Chapter 14

Services for VME Drivers

This chapter provides an overview of the kernel services needed by a kernel-level VME device driver. It contains a complete example driver.

Kernel Services for VME

The kernel provides services for mapping the VME bus into the kernel virtual address space for PIO or DMA, and for transferring data using maps. It also provides services for allocating interrupt vector numbers.

Mapping PIO Addresses

A PIO map is a system object that represents the mapping from a location in the kernel's virtual address space to some small range of addresses on a VME or EISA bus. After creating a PIO map, a device driver can use it in the following ways:

- Use the specific kernel virtual address that represents the device, either to load or store data, or to map that address into user process space.
- Copy data between the device and memory without learning the specific kernel addresses involved.
- Perform bus read-modify-write cycles to apply Boolean operators efficiently to device data.

The kernel virtual address returned by PIO mapping is not a physical memory address and is not a bus address. The kernel virtual address and the VME or EISA bus address need not have any bits in in common.

The functions used with PIO maps are summarized in Table 14-1.

Table 14-1 Functions to Create and Use PIO Maps

Function	Header Files	Can Sleep	Purpose
pio_mapalloc(D3)	pio.h & types.h	Y	Allocate a PIO map.
pio_mapfree(D3)	pio.h & types.h	N	Free a PIO map.
pio_badaddr(D3)	pio.h & types.h	N	Check for bus error when reading an address.
pio_badaddr_val(D3)	pio.h & types.h	N	Check for bus error when reading an address and return the value read.
pio_wbadaddr(D3)	pio.h & types.h	N	Check for bus error when writing to an address.
pio_wbadaddr_val(D3)	pio.h & types.h	N	Check for bus error when writing a specified value to an address.
pio_mapaddr(D3)	pio.h & types.h	N	Convert a bus address to a virtual address.
pio_bcopyin(D3)	pio.h & types.h	Y	Copy data from a bus address to kernel's virtual space.
pio_bcopyout(D3)	pio.h & types.h	Y	Copy data from kernel's virtual space to a bus address.
pio_andb_rmw(D3)	pio.h & types.h	N	Byte read-and-write.
pio_andh_rmw(D3)	pio.h & types.h	N	16-bit read-and-write.
pio_andw_rmw(D3)	pio.h & types.h	N	32-bit read-and-write.
pio_orb_rmw(D3)	pio.h & types.h	N	Byte read-or-write.
pio_orh_rmw(D3)	pio.h & types.h	N	16-bit read-or-write.
pio_orw_rmw(D3)	pio.h & types.h	N	32-bit read-or-write.

A kernel-level device driver creates a PIO map by calling **pio_mapalloc()**. This function performs memory allocation and so can sleep. PIO maps are typically created in the *pfx***edtinit()** entry point, where the driver first learns about the device addresses from the contents of the *edt_t* structure (see "Entry Point edtinit()" on page 144).

The parameters to **pio_mapalloc()** describe the range of addresses that can be mapped in terms of

- the bus type, ADAP_VME or ADAP_EISA from *sys/edt.h*
- the bus number, when more than one bus is supported
- the address space, using constants such as PIOMAP_A24N or PIOMAP_EISA_IO from *sys/pio.h*
- the starting bus address and a length

This call also specifies a "fixed" or "unfixed" map. The usual type is "fixed." For the differences, see "Fixed PIO Maps" on page 328 and "Unfixed PIO Maps" on page 329.

A call to **pio_mapfree()** releases a PIO map. PIO maps created by a loadable driver must be released in the *pfx***unload()** entry point (see "Entry Point unload()" on page 167 and "Unloading" on page 240).

Testing the PIO Map

The PIO map is created from the parameters that are passed. These are not validated by **pio_mapalloc()**. If there is any possibility that the mapped device is not installed, not active, or improperly configured, you should test the mapped address.

The **pio_badaddr()** and **pio_badaddr_val()** functions test the mapped address to see if it is usable for input. Both functions perform the same operation: operating through a PIO map, they test a specified bus address for validity by reading 1, 2, 4, or 8 bytes from it. The **pio_badaddr_val()** function returns the value that it reads while making the test. This can simplify coding, as shown in Example 14-1.

Example 14-1 Comparing pio_badaddr() to pio_badaddr_val()

```
unsigned int gotvalue;
piomap_t *themap;
/* Using only pio_badaddr() */
   if (!pio_badaddr(themap,CTLREG,4)
   {
      (void) pio_bcopyin(themap,CTLREG,&gotvalue,4,4,0);
      ...use "gotvalue"
/* Using pio_badaddr_val() */
   if (!pio_badaddr_val(themap,CTLREG,4,&gotvalue))
   {
      ...use "gotvalue"
```

The **pio_wbadaddr()** function tests a mapped device address for writeability. The **pio_wbadaddr_val()** not only tests the address but takes a specific value to write to that address in the course of testing it.

Using the Mapped Address

From a fixed PIO map you can recover a kernel virtual address that corresponds to the first bus address in the map. The **pio_mapaddr()** function is used for this.

You can use this address to load or store data into device registers. In the *pfx***map()** entry point (see "Concepts and Use of mmap()" on page 158), you can use this address with the **v_mapphys()** function to map the range of device addresses into the address space of a user process.

You cannot extract a kernel address from an unfixed PIO map, as explained under "Unfixed PIO Maps" on page 329.

Using the PIO Map in Functions

You can apply a variety of kernel functions to any PIO map, fixed or unfixed. The **pio_bcopyin()** and **pio_bcopyout()** functions copy a range of data between memory and a fixed or unfixed PIO map. These functions are optimized to the hardware that exists, and they do all transfers in the largest size possible (32 or 64 bits per transfer). If you need to transfer data in specific sizes of 1 or 2 bytes, use direct loads and stores to the mapped addresses.

The series of functions **pio_andb_rmw()** and **pio_orb_rmw()** perform a read-modify-write cycle on the VME bus. You can use them to set or clear bits in device registers. A read-modify-write cycle is faster than a load followed by a store since it uses fewer system bus cycles.

Fixed PIO Maps

On a Challenge or Onyx system, a PIO map can be either "fixed" or "unfixed." This attribute is specified when the map is created. (On a Crimson system, only fixed maps are created regardless of which type is requested.)

The Challenge and Onyx architecture provides for a total of 15 separate, 8 MB windows on VME address space for each VME bus. Two of these are permanently reserved to the kernel, and one window is reserved for use with unfixed mappings. The remaining 12 windows are available to implement fixed PIO maps.

When the kernel creates a fixed PIO map, the map is associated with one of the 12 available VME mapping windows. The kernel tries to be clever, so that whenever a PIO map falls within an 8 MB window that already exists, the PIO map uses that window. If the desired VME address is not covered by an open window, one of the twelve windows for that bus is opened to expose a mapping for that address.

It is possible in principle to configure thirteen devices that are scattered so widely in the A32 address space that twelve, 8 MB windows cannot cover all of them. In that unlikely case, the attempt to create the thirteenth fixed PIO map will fail for lack of a mapping window.

In order to prevent this, simply configure your PIO addresses into a span of at most 96 MB per bus (see “Configuring Device Addresses” on page 310).

Unfixed PIO Maps

When you create an unfixed PIO map, the map is not associated with any of the twelve mapping windows. As a result, the map cannot be queried for a kernel address that might be saved, or mapped into user space.

You can use an unfixed map with kernel functions that copy data or perform read-modify-write cycles. These functions use the one mapping window that is reserved for unfixed maps, repositioning it in VME space if necessary.

The *lboot* command uses an unfixed map to perform the *probe* and *exprobe* sequences from VECTOR statements (see “Configuring the System Files” on page 311). As a result, these probes do not tie up mapping windows.

Mapping DMA Addresses

A DMA map is a system object that represents a mapping between a buffer in kernel virtual space and a range of VME bus addresses. After creating a DMA map, a driver uses the map to specify the target address and length to be programmed into a VME bus master before a DMA transfer.

The functions that operate on DMA maps are summarized in Table 14-2.

Table 14-2 Functions That Operate on DMA Maps

Function	Header Files	Can Sleep	Purpose
dma_map(D3)	dmamap.h & types.h & sema.h	N	Load DMA mapping registers for an imminent transfer.
dma_mapbp(D3)	dmamap.h & types.h & sema.h	N	Load DMA mapping registers for an imminent transfer.
dma_mapaddr(D3)	dmamap.h & types.h & sema.h	N	Return the "bus virtual" address for a given map and address.
dma_mapalloc(D3)	dmamap.h & types.h & sema.h	Y	Allocate a DMA map.
dma_mapfree(D3)	dmamap.h & types.h & sema.h	N	Free a DMA map.
vme_adapter(D3)	vmereg.h & types.h	N	Determine VME adapter that corresponds to a given memory address.

A device driver allocates a DMA map using **dma_mapalloc()**. This is typically done in the *pfx***edtinit()** entry point, provided that the maximum I/O size is known at that time (see "Entry Point edtinit()" on page 144). The important argument to **dma_mapalloc()** is the maximum number of pages (I/O pages, the unit is IO_NBPP declared in *sys/immu.h*) to be mapped at one time.

Note: In the Challenge and Onyx systems, a limit of 64 MB of mapped DMA space per VME adapater is imposed by the hardware. Some few megabytes of this are taken early by system drivers. Owing to a bug in IRIX 5.3 and 6.1, a request for 64 MB or more is not rejected, but waits forever. However, in any release, a call to **dma_mapalloc()** that requests a single map close to the 64 MB limit is likely to wait indefinitely for enough map space to become available.

DMA maps created by a loadable driver should be released in the *pfx***unload()** entry point (see “Entry Point unload()” on page 167 and “Unloading” on page 240).

Using a DMA Map

A DMA map is used prior to a DMA transfer into or out of a buffer in kernel virtual space. The function **dma_map()** takes a DMA map, a buffer address, and a length. It assigns a span of contiguous VME addresses of the specified length, and sets up a mapping between that range of VME addresses and the physical addresses that represent the specified buffer.

When the buffer spans two or more physical pages (IO_NBPP units), **dma_map()** sets up a scatter/gather operation, so that the VME bus controller will place the data in the appropriate page frames.

It is possible that **dma_map()** cannot map the entire size of the buffer. This can occur only when the buffer spans two or more pages, and is caused by a shortage of mapping registers in the bus adapter. The function maps as much of the buffer as it can, and returns the length of the mapped data.

You must always anticipate that **dma_map()** might map less than the requested number of bytes, so that the DMA transfer has to be done in two or more operations.

Following the call to **dma_map()**, you call **dma_mapaddr()** to get the bus virtual address that represents the first byte of the buffer. This is the address you program into the bus master device (using a PIO store), in order to set its starting transfer address. Then you initiate the DMA transfer (again by storing a command into a device register using PIO).

Allocating an Interrupt Vector Dynamically

When a VME device generates an interrupt, the Silicon Graphics VME controller initiates an interrupt acknowledge (IACK) cycle on the VME bus. During this cycle, the interrupting device presents a data value that characterizes the interrupt. This is the *interrupt vector*, in VME terminology.

According to the VME standard, the interrupt vector can be a data item of 8, 16, or 32 bits. However, Silicon Graphics systems accept only an 8-bit vector, and its value must fall in the range 1-254 inclusive. (0x00 and 0xFF are excluded because they could be generated by a hardware fault.)

The interrupt vector returned by some VME devices is hard-wired or configured into the board with switches or jumpers. When this is the case, the vector number should be written as the *vector* parameter in the VECTOR statement that describes the device (see “Configuring the System Files” on page 311).

Some VME devices are programmed with a vector number at runtime. For these devices, you omit the *vector* parameter, or give its value as an asterisk. In the device driver, you use the functions in Table 14-3 to choose a vector number.

Table 14-3 Functions to Manage Interrupt Vector Values

Function	Header Files	Can Sleep	Purpose
vme_ivec_alloc(D3)	vmereg.h & types.h	N	Allocate a VME bus interrupt vector.
vme_ivec_free(D3)	vmereg.h & types.h	N	Free a VME bus interrupt vector.
vme_ivec_set(D3)	vmereg.h & types.h	N	Register a VME bus interrupt vector.

Allocating a Vector

In the *pfx***edtinit()** entry point, the device driver selects a vector number for the device to use. The best way to select a number is to call **vme_ivec_alloc()**, which returns a number that has not been registered for that bus, either dynamically or in a VECTOR line.

The driver then uses **vme_ivec_set()** to register the chosen vector number. This function takes parameters that specify

- The vector number
- The bus number to which it applies
- The address of the interrupt handler for this vector—typically but not necessarily the name of the *pfx***intr()** entry point of the same driver
- An integer value to be passed to the interrupt entry point—typically but not necessarily the vector number

The **vme_ivec_set()** function simply registers the number in the kernel, with the following two effects:

- The **vme_ivec_alloc()** function does not return the same number to another call until the number is released.
- The specified handler is called when any device presents this vector number on an interrupt.

Multiple devices can present the identical vector, provided that the interrupt handler has some way of distinguishing one device from another.

Note: If you are working with both the VME and EISA interfaces, it is worth noting that the number and types of arguments of **vme_ivec_set()** differ from the similar EISA support function **eisa_ivec_set()**.

Releasing a Vector

There is a limit of 254 vector numbers per bus, so it is a good idea for a loadable driver, in its *pfx***unload()** entry point, to release a vector by calling **vme_ivec_free()** (see “Entry Point unload()” on page 167 and “Unloading” on page 240).

Vector Errors

A common problem with programmable vectors in the Challenge or Onyx systems is the appearance of the following warning in the SYSLOG file:

```
Warning: Stray VME interrupt: vector =0xff
```

One possible cause of this error is that the board is emitting the wrong interrupt vector; another is that the board is emitting the correct vector but with the wrong timing, so that the VME bus adapter samples all-binary-1 instead. Both these conditions can be verified with a VME bus analyzer. In the Challenge or Onyx hardware design, the most likely cause is the presence of empty slots in the VME card cage. All empty slots must be properly jumpered in order to pass interrupts correctly.

Supporting Early IO4 Cache Problems

VME drivers that support DMA to buffers that are not cache-aligned multiples of a cache-line need to take special precautions in a Challenge system; see Appendix B, “Challenge DMA with Multiple IO4 Boards.”

Sample VME Device Driver

The source module displayed in Example 14-2 contains a complete character device driver for a hypothetical VME device. Although it is a character driver, it contains a strategy routine (the **cdev_strategy()** function). Both the *pfx***read()** and *pfx***write()** entry points call the strategy routine to perform the actual I/O. As a result, this driver could be installed as either a block device driver or a character driver, or as both.

The driver is multiprocessor-aware, so its *pfx***devflag** global contains D_MP. It uses two locks. A basic lock (*board.cd_lock*) is used for short-term mutual exclusion, to block a potential race between the strategy routine and the interrupt routine. A semaphore (*board.cd_rwsema*) is used for long-term mutual exclusion to make sure that only one process uses the device for reading or writing at any time.

Example 14-2 Example VME Character Driver

```
/*********************************************************************\
*       File:           cdev.c                                        *
*                                                                     *
*       The following is an example of how a device driver for a VME  *
*       character device might be written.  The sample driver         *
*       illustrates how to write code which performs DMA into both    *
*       kernel and user address space, as well as how a sample        *
*       driver's registers would be mapped into user address space.   *
*                                                                     *
\*********************************************************************/
#include <sys/types.h>          /* Contains basic kernel typedefs */
#include <sys/param.h>
#include <sys/immu.h>           /* Contains VM-specific definitions (map) */
#include <sys/region.h>         /* Contains VM data structure defs (map) */
#include <sys/conf.h>           /* Contains cdevsw and driver flag defs */
#include <sys/vmereg.h>         /* Contains VME bus-specific definitions */
#include <sys/edt.h>            /* Contains definition of edt structs */
#include <sys/dmamap.h>         /* Definitions for dma structs and routines */
#include <sys/pio.h>            /* Definitions for pio structs and routines */
#include <sys/cmn_err.h>        /* Definitions for cmn_err constants */
#include <sys/errno.h>          /* Define classic error numbers */
#include <sys/open.h>           /* Define open types used in otyp open parm */
#include <sys/cred.h>           /* Contains credential structure declaration */
#include <sys/ksynch.h>         /* Define ddi-compliant synch primitives */
#include <sys/sema.h>           /* Include semaphore prototypes */
#include <sys/ddi.h>            /* Include the ddi-compliant stuff */
/* Some constants used throughout the driver */
#define CDEV_MAX_XFERSIZE 65536
#define VALID_DEVICE 0x0acedeed
/* The following structure is provided so that we can memory map the
 * device's control registers.  For purposes of illustration, we
 * provide a couple of generic registers; a real device would have
 * completely different mappings.
 */
#define CMD_READ        0x1
#define CMD_WRITE       0x2
#define CMD_CLEAR_INTR  0x4
#define CMD_RESET       0x8
typedef struct deviceregs_s {
    volatile unsigned short  cr_status; /* The device's status register */
    volatile unsigned short  cr_cmd;    /* The device's command register */
    volatile unsigned int    cr_dmaaddr;/* The DMA address */
    volatile unsigned int    cr_count;  /* The number of bytes to xfer */
```

```
    volatile unsigned int    cr_devid;  /* The device ID register */
    volatile unsigned int    cr_parm;   /* A device parameter */
} deviceregs_t;
/* The cdevboard structure contains about a device which the
 * driver needs to maintain.  In general, each instance of a
 * device in the system has an associated cdevboard structure
 * which contains driver-specific information about that board.
 */
#define STATUS_PRESENT          0x1
#define STATUS_OPEN             0x2
#define STATUS_INTRPENDING      0x4
#define STATUS_TIMEOUT          0x8
#define FLAG_SET(_x, _y)        (((_x)->cd_status) |= (_y))
#define FLAG_CLEAR(_x, _y)      (((_x)->cd_status) &= (~(_y)))
#define FLAG_TEST(_x, _y)       (((_x)->cd_status) & (_y))
typedef struct cdevboard_s {
    lock_t              cd_lock;        /* Used for mutual exclusion */
    sema_t              cd_rwsema;      /* Prevents simult. read & write */
    volatile deviceregs_t *cd_regs;     /* Memory-mapped control regs */
    dmamap_t            *cd_map;        /* DMA Map for this device */
    unsigned int        cd_ctlr;        /* The controller # of this device */
    unsigned int        cd_status;      /* The board's status. */
    unsigned int        cd_strayintr;   /* Counts stray interrupts */
    struct buf         *cd_buf;         /* Pointer to buffer */
    unsigned int        cd_count;       /* Count of bytes being transferred */
    toid_t              cd_tout;        /* Timeout handle */
} cdevboard_t;
/* We need to tell the kernel what kind of interface this driver
 * expects.  For a simple, non-MP driver, the devflag can be set to
 * 0.  Since we're going to be a little more ambitious, we'll tell
 * the kernel that we are capable of running MP.
 */
int cdev_devflag = D_MP;
/* Forward declarations of general driver functions */
int  cdev_intr(int board);
int  cdev_strategy(struct buf *bp);
void cdev_timeout(cdevboard_t *board);
/* Driver global data structures; to minimize memory use, we create
 * an array of pointers to audioboard structures and only allocate the
 * actual structure if the corresponding board is configured.
 */
#define CDEV_MAX_BOARDS 4
static cdevboard_t *CDevBoards[CDEV_MAX_BOARDS + 1];
#if DEBUG
#define DPRINTF(_x)     debug_printf _x
```

```
void debug_printf(char *fmt, ...)
{
        va_list ap;
        extern void icmn_err();
        va_start(ap, fmt);
        icmn_err(CE_NOTE, fmt, ap);
        va_end(ap);
}
#else
#define DPRINTF(_x)
#endif

/************************************************************************
 * edtinit is the first routine all VME drivers need to provide.
 * This function is called early during kernel initialization, and
 * drivers generally use it to set up driver-global data structures
 * and device mappings for any devices which exist.  The kernel calls
 * it once for each VECTOR line in the appropriate .sm file.
 */
void
cdev_edtinit(struct edt *e)
{
    piomap_t *piomap;           /* Control register mapping descriptor */
    dmamap_t *dmamap;           /* DMA mapping for read/write buffers */
    volatile deviceregs_t *base;   /* Base address of device’s control regs */
    vme_intrs_t *intrs;         /* Pointer to VME interrupt information */
    int intr_vec;               /* Actual vector to use */
    int ctlr;                   /* Board number to be configured */
    cdevboard_t *board;         /* New board data structure */
    /* Make sure that the the controller number is within range */
    ctlr = e->e_ctlr;
    if (ctlr < 0 || ctlr > CDEV_MAX_BOARDS) {
        cmn_err(CE_WARN, “cdev%d: controller number is invalid”, ctlr);
        return;
    }
    /* Allocate a programmed I/O mapping structure for the particular
     * device.  The kernel uses the data in the e_space field to figure
     * out both the VME base address and the total size of the register area.
     */
    piomap = pio_mapalloc(e->e_bus_type, e->e_adap, e->e_space,
                          PIOMAP_FIXED, “cdev”);
    /* XXX Check for the success of piomap allocation */
    if (piomap == (piomap_t *)NULL){
        cmn_err(CE_WARN, “cdev%d: Could not allocate piomap”, ctlr);
        return;
```

```
}
/* Now that the map is allocated, we position it so that it overlays
 * the device’s hardware registers.  Since this is a fixed map, we
 * just pass in the base address of the control register range.
 * iobase comes from the VECTOR line in the .sm file.
 */
base = (volatile deviceregs_t*) pio_mapaddr(piomap, e->e_iobase);
/* We’re going to need to DMA map the user’s buffer during read and
 * write requests, so we preallocate a fixed number of dma mapping
 * entries based on the constant CDEV_MAX_XFERSIZE.  If we allowed
 * multiple users to perform reads and writes simultaneously we’d
 * probably want to allocate one map for reads and one for writes.
 * Since we only allow one operation to occur at any given time,
 * though, we can get away with only one.
 *
 * IMPORTANT NOTE: There are only a limited number of dma mapping
 * registers available in a system; you should be somewhat conservative
 * in your use of them.  It is reasonable to consume up to 100 per
 * device (you can use more if you expect that only a couple devices
 * will be attached for each driver.  If, for example, this driver
 * will never control more than two devices, you could probably use
 * up to 512 mapping registers for each device.  If however, you’d expect
 * to see hundreds of devices, you’d need to be more conservative.
 */
dmamap = dma_mapalloc(DMA_A24VME, e->e_adap,
                       io_btoc(CDEV_MAX_XFERSIZE) + 1, 0);
if (dmamap == (dmamap_t*) NULL) {
    cmn_err(CE_WARN, “cdev%d: Could not allocate dmamaps”, ctlr);
    pio_mapfree(piomap);
    return;
}
/* The next step would be to probe the device to determine whether
 * it is actually present.  To do this, we attempt to read some
 * registers which behave in a manner unique to this particular
 * hardware.  We need to protect ourselves in the event that the
 * device isn’t actually present, however, so we use the badaddr
 * and wbadaddr routines.  For our example, we assume that the
 * device is present if it’s device
 */
if ((badaddr(&(base->cr_devid), 4) == 0) &&
    (base->cr_devid == VALID_DEVICE)) {
    DPRINTF((“cdev%d: found valid device”, ctlr));
} else {
    /* It doesn’t look like the device is there. */
    cmn_err(CE_WARN, “cdev%d: cannot find actual device”, ctlr);
```

```
    pio_mapfree(piomap);
    dma_mapfree(dmamap);
    return;
}
/* Now we set up the interrupt for this device.
 * It is possible to specify a vector and priority level on the
 * VECTOR line in the .sm file, so we check to see if such was the case.
 */
intrs = (vme_intrs_t*) e->e_bus_info;
intr_vec = intrs->v_vec;
/* If intr_vec is non-zero, user specified specific vec in .sm file.
 * If the interrupt was specified on the VECTOR line, the kernel has
 * already established a vector for us, so we don't need to do it
 * ourselves.
 */
if (intr_vec == 0) {
    intr_vec = vme_ivec_alloc(e->e_adap);
    /* Make sure that we got a good interrupt vector */
    if (intr_vec == -1) {
        cmn_err(CE_WARN, "cdev%d: could not allocate intr vector\n", ctlr);
        pio_mapfree(piomap);
        dma_mapfree(dmamap);
        return;
    }
    /* Associate this driver's interrupt routine with the acquired vec */
vme_ivec_set(e->e_adap, intr_vec, cdev_intr, 0);
}
/* Initialize the board structure for this board */
board = (cdevboard_t*) kmem_alloc(sizeof(cdevboard_t));
if (board == (void*) 0) {
    cmn_err(CE_WARN, "cdev%d: kmem_alloc failed", ctlr);
    pio_mapfree(piomap);
    dma_mapfree(dmamap);
    /* XXX Need to check whether it is allocated?? */
    vme_ivec_free(e->e_adap, intr_vec);
    return;
}
board = CDevBoards[ctlr];
board->cd_regs = base;
board->cd_ctlr = ctlr;
board->cd_status = STATUS_PRESENT;
board->cd_strayintr = 0;
board->cd_map = dmamap;
initnsema(&board->cd_rwsema, 1, "CDevRWM");
/* Finally, call any one-time-only device initialization routines;
```

```
     * this particular device doesn't have any.
     */
    return;
}
/***********************************************************************
 * cdev_open -- When opening a device, we need to check for mutual
 * exclusion (if desired) and then set up an additional data structures
 * if this is the first time the device has been opened.  Remember that
 * the OS usually doesn't call close until all users close the device,
 * so you can't count on being able to set up unique data for each user
 * of the device unless you either disallow multiple opens at the same time
 * or mark the device as being a layered (otype = O_LYR) device.
 */
int
cdev_open(dev_t *dev, int flag, int otyp, cred_t *cred)
{
    minor_t             ctlr;           /* Controller # of cdev being opened */
    cdevboard_t         *board;         /* per-board data for opened cdev*/
    int                 s;              /* Opaque lock value */
    /* We assume that the minor number encodes the ctlr number, so
     * we just go ahead and use it to index the CDevBoards array once
     * we've validated it.
     */
    ctlr = geteminor(*dev);
    if (ctlr > CDEV_MAX_BOARDS) {
        DPRINTF(("cdev%d: open: minor number out of range", ctlr));
        return ENXIO;
    }
    board = CDevBoards[ctlr];
    if (FLAG_TEST(board, STATUS_PRESENT) || (board->cd_ctlr != ctlr)) {
        DPRINTF(("cdev%d: open: device not found", ctlr));
        return ENXIO;
    }
    /* If exclusiveness is desired, we now need to atomically insure that
     * we are the owners of the device.
     */
    s = LOCK(&board->cd_lock, splhi);
    if (FLAG_TEST(board, STATUS_OPEN)) {
        UNLOCK(&board->cd_lock, s);
        return EBUSY;
    } else {
        ASSERT(board->cd_status == STATUS_PRESENT);
        FLAG_SET(board, STATUS_OPEN);
    }
    UNLOCK(&board->cd_lock, s);
```

```
    return 0;
}

/************************************************************************
 * cdev_close -- Called when the open reference count drops to zero.
 *      Cleans up any leftover data structure and marks the device as
 *      available.
 */
int
cdev_close(dev_t dev, int flag, int otyp, cred_t *cred)
{
    int                 ctlr;           /* Controller # of dev being closed */
    cdevboard_t         *board;         /* per-board data structure */
    ctlr = geteminor(dev);
    ASSERT(ctlr <= CDEV_MAX_BOARDS);
    board = CDevBoards[ctlr];
    ASSERT(board && FLAG_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    /* Do any cleanup required here */
    /* Reset the board's status flags (to clear the OPEN flag) */
    FLAG_CLEAR(board, STATUS_OPEN);
    return 0;
}
/************************************************************************
 * cdev_intr -- Called when an interrupt occurs.  We check to see if
 *      a process was waiting for an I/O operation to complete and
 *      re-activate that process if such is the case.
 */
#ifdef EVEREST /* IO4 fix for Challenge */
extern int io4_flush_cache(caddr_t piomap);
#endif
int
cdev_intr(int ctlr)
{
    cdevboard_t         *board;         /* per-board data structure pointer */
        int s;  /* lock return value */
    /* Make sure that the controller value is legitimate */
    ASSERT(ctlr <= CDEV_MAX_BOARDS);
    board = CDevBoards[ctlr];
    ASSERT(board && FLAG_TEST(board, STATUS_PRESENT));
#ifdef EVEREST /* flush IO4 cache */
    (void)io4_flush_cache((caddr_t)board->cd_regs);
#endif
        /*
         * Get exclusive use of the board. This ensures that the strategy
         * routine is completely finished setting STATUS_INTRPENDING before
```

```
         * we examine it.
         */
        s = LOCK(&board->cd_lock, splhi);
    /* It's possible that we could get a stray interrupt if the hardware
     * is flaky, so we keep a count of bogus interrupts and ignore them.
     */
    if (FLAG_TEST(board, STATUS_OPEN|STATUS_INTRPENDING)) {
        board->cd_strayintr++;
        return 0;
    }
    /* Acknowledge the interrupt from the device */
    board->cd_regs->cr_cmd = CMD_CLEAR_INTR;
    FLAG_CLEAR(board, STATUS_INTRPENDING);
    /* Remove the timeout request */
    untimeout(board->cd_tout);
    /* Update the buffer's parameters */
    ASSERT(board->cd_buf->b_bcount > 0);
    board->cd_buf->b_bcount     -= board->cd_count;
    board->cd_buf->b_dmaaddr    += board->cd_count;
    /* Release the mutual exclusion on the board. */
    UNLOCK(&board->cd_lock,s);
    /* If the transfer count is 0, then we've transferred all of the
     * bytes in the request, so we call iodone to awaken the user process.
     * Otherwise, we call cdev_strat to initiate another transfer.
     */
    if (board->cd_buf->b_bcount == 0)
        iodone(board->cd_buf);
    else
        cdev_strategy(board->cd_buf);
    return 0;
}
/************************************************************************
 * cdev_read -- reads data from the device.  We employ the uiophysio
 *      routine to perform all the requisite mapping of the buffer
 *      for us and then call the cdev_strat routine.  The big advantage
 *      of uiophysio() is that it sets up memory such that the device can
 *      DMA directly into the user address space.  The strategy routine
 *      is responsible for actually setting up and initiating the transfer.
 *      The process will block in uiophysio until the interrupt handler
 *      calls iodone() on buffer pointer.
 */
int
cdev_read(dev_t dev, uio_t *uio, cred_t *cred)
{
    int         ctlr;
```

```
    cdevboard_t *board;
    int         error = 0;
    ASSERT(ctlr >= 0 && ctlr <= CDEV_MAX_BOARDS);
    ctlr = geteminor(ctlr);
    ASSERT(board && FLAG_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    board = CDevBoards[ctlr];
    /* Since we allocated only a single DMA buffer, we need to block
     * if a previous transfer hasn't completed.
     */
    psema(&board->cd_rwsema, PZERO+1);
    error = uiophysio(cdev_strat, NULL, dev, B_READ, uio);
    /* Check to see if the transfer timed out */
    if (FLAG_TEST(board, STATUS_TIMEOUT)) {
        FLAG_CLEAR(board, STATUS_TIMEOUT);
        error = EIO;
    }
    vsema(&board->cd_rwsema);
    return error;
}
/**********************************************************************
 * cdev_write -- writes data from a user buffer to the device.
 *      We employ the uiophysio routine to set up the mappings for us.
 *      Once the mappings are established, uiophysio will call the
 *      given strategy routine (cdev_strat) with a buffer pointer.
 *      The strategy routine is then responsible for kicking off the
 *      transfer.  The process will block in uiophysio until the
 *      interrupt handler calls iodone() on the buffer pointer.
 */
int
cdev_write(dev_t dev, uio_t *uio, cred_t *cred)
{
    int         ctlr;
    cdevboard_t *board;
    int         error = 0;
    ASSERT(ctlr >= 0 && ctlr <= CDEV_MAX_BOARDS);
    ctlr = geteminor(ctlr);
    ASSERT(board && FLAG_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    board = CDevBoards[ctlr];
    psema(&board->cd_rwsema, PZERO+1);
    error = uiophysio(cdev_strat, NULL, dev, B_WRITE, uio);
    /* Check to see if the transfer timed out */
    if (FLAG_TEST(board, STATUS_TIMEOUT)) {
        FLAG_CLEAR(board, STATUS_TIMEOUT);
        error = EIO;
    }
```

```
    vsema(&board->cd_rwsema);
    return error;
}
/***********************************************************************
 * cdev_strat -- Called by uiophysio, cdev_strat actually performs all
 *      the device-specific actions needed to initiate the transfer,
 *      such as establishing the DMA mapping of the transfer buffer and
 *      actually programming the device.  There is an implicit assumption
 *      that the device will interrupt at some later point when the I/O
 *      operation is complete.
 */
int
cdev_strategy(struct buf *bp)
{
    int ctlr;                   /* Controller # being accessed */
    cdevboard_t *board;         /* Board data structure */
    int mapcount;               /* Count */
    int s;                      /* opaque lock value */

    /* Get a reference to the actual board structure */
    ctlr = geteminor(bp->b_edev);
    ASSERT(ctlr >= 0 && ctlr <= CDEV_MAX_BOARDS);
    board = CDevBoards[ctlr];
    ASSERT(board && FLAG_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    /* We start by mapping the appropriate region into VME address space.
     * Because of the mapping registers we don't have to worry about the
     * fact that the physical pages backing the data regions may be
     * physically discontinuous; in effect, the DMA mapping is taking the
     * place of scatter/gather hardware.  Nonetheless, in order to avoid
     * consuming an excessive number of translation entries we limit the
     * size of the transfer to CDEV_MAX_XFERSIZE.
     */
    mapcount = MIN(bp->b_bcount, CDEV_MAX_XFERSIZE);
    mapcount = dma_map(board->cd_map, bp->b_dmaaddr, mapcount);
    ASSERT(mapcount > 0);
    /* Before starting the I/O, get exclusive use of the board struct.
     * This ensures that, if this CPU is interrupted and we are slow to
     * set STATUS_INTRPENDING, cdev_intr() will be locked out until we do.
     */
    s = LOCK(&board->cd_lock, splhi);
    /* Now we start the transfer by writing into memory-mapped registers */
    board->cd_regs->cr_dmaaddr = dma_mapaddr(board->cd_map, bp->b_dmaaddr);
    board->cd_regs->cr_count = mapcount;
    board->cd_regs->cr_cmd = ((bp->b_flags & B_WRITE) ? CMD_WRITE : CMD_READ);
    /* Schedule a timeout, just in case the device decides to hang forever */
```

```
    itimeout(cdev_timeout, board, 2000, splhi);
    /* Finally, we update some of the board data structures */
    board->cd_buf   = bp;
    board->cd_count = mapcount;
    FLAG_SET(board, STATUS_INTRPENDING);
    /* Release the board struct, so the interrupt handler can use it. */
    UNLOCK(&board->cd_lock, s);
    /* Upon returning, uiophysio will block until cdev_intr calls iodone() */
    return 0;
}
/**************************************************************************
 * cdev_ioctl -- Not too exciting.  We'll assume that the device has
 *      one controllable parameter which can be both written and received.
 *      To help users avoid errors, we use unusual constants for the ioctl
 *      values.  In a real driver, the CDIOC definitions would go into a
 *      header file.
 */
#define CDIOC_SETPARM   0xcd01
#define CDIOC_GETPARM   0xcd02
int
cdev_ioctl(dev_t dev, int cmd, int arg, int mode, cred_t *cred)
{
    int             ctlr;           /* Controller number */
    cdevboard_t    *board;          /* Per-controller data */
    int             error = 0;      /* Error return value */
    ctlr = geteminor(dev);
    ASSERT(ctlr >= 0 && ctlr <= CDEV_MAX_BOARDS);
    board = CDevBoards[ctlr];
    ASSERT(board && FLAG_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    switch (cmd) {
      case CDIOC_SETPARM:
        board->cd_regs->cr_parm = arg;
        break;
      case CDIOC_GETPARM:
        {
            int value;
            value = board->cd_regs->cr_parm;
            if (copyout(&value, (void*) arg, sizeof(int)))
                error = EFAULT;
        }
        break;
      default:
        error = EINVAL;
        break;
    }
```

```
    return error;
}
/************************************************************************
 * cdev_timeout -- If an I/O request takes a really long time to complete
 *      for some reason (if, for example, someone takes the device offline),
 *      it is better to warn the user than to simply hang.  This timeout
 *      routine will cancel any pending I/O requests and display a message.
 *      A more sophisticated routine might try resetting the device and
 *      re-executing the operation.
 */
void
cdev_timeout(cdevboard_t *board)
{
    /* Clear the pending request from the device.  This operation
     * is extremely dependent on the actual device.  This driver
     * pretends that we simply can use the reset command.
     */
    board->cd_regs->cr_cmd = CMD_RESET;
    /* Make a note that the operation timed out */
    FLAG_SET(board, STATUS_TIMEOUT);
    /* Display a warning */
    cmn_err(CE_WARN, "cdev%d: device timed out", board->cd_ctlr);
    /* Notify the user process that the operation has "finished". */
    iodone(board->cd_buf);
}
/************************************************************************
 * cdev_map -- For illustrative purposes, we show how one would go about
 *      mapping the device's control registers.
 */
int
cdev_map(dev_t dev, vhandl_t *vt, off_t off, int len, int prot)
{
    int                 ctlr;           /* Controller number */
    cdevboard_t        *board;          /* Per-controller data */
    ctlr = geteminor(dev);
    ASSERT(ctlr >= 0 && ctlr <= CDEV_MAX_BOARDS);
    board = CDevBoards[ctlr];
    ASSERT(board && FLAGS_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    if (v_mapphys(vt, (void*) board->cd_regs, len))
        return ENOMEM;
    else
        return 0;
}
/************************************************************************
 * cdev_unmap -- Called when a region is unmapped.  We don't actually
```

```
 *      need to do anything.
 */
int
cdev_unmap(dev_t dev, vhandl_t *vt)
{
    /* No need to do anything here; unmapping is handled by upper levels
     * of the kernel.
     */
    return 0;
}
```