Chapter 7

Writing Kernel-level General Memory-mapping Device Drivers

This chapter explains how to write kernel-level general memory-mapping device drivers.

It contains the following sections:

This chapter describes how a kernel-level device driver can map VME device hardware or main (kernel) memory to user space. It introduces two system calls, **mmap**(2) and **munmap***(2)*, as well as describing two IRIX driver functions, *drv***map**() and *drv***unmap**()[1]. Because your driver allows the user to map the device user space, your driver may not need to include *drv***read**() and *drv***write**() functions.

---

[1] SVR4 also uses a *drv***mmap**() function and an optional *drv***unmap**() function.

Including a Memory-mapping Device Driver in the Kernel

For a VME device driver, refer to Chapter 3, "Writing a VME Device Driver," for details on adding a VME device driver to the kernel.

For a EISA device driver, see Chapter 4, "Writing an EISA Device Driver," for details on including this type of driver.

For a SCSI device driver, refer to Chapter 5, "Writing a SCSI Device Driver," for details on including this type of driver to the kernel.

For a GIO device driver, see Chapter 6, "Writing Kernel-level GIO Device Drivers," for details.

Mapping and Unmapping Functions

The functions for mapping memory and registers are **mmap**(), *drv***mmap(), munmap**(), and **v_mapphys**().

mmap – Mapping the Device
*drv*mmap – Mapping the Device

When a user-level program wants to map device memory into its address space, the user program opens the special file corresponding to the particular device and uses the **mmap**() system call. Your driver needs to call *drv***mmap**() — which you need to write — when a **mmap** call maps into a user's address space. *drv***mmap** is logically similar to a *drv***open** routine; make sure it does whatever work your device requires. (See Chapter 2, "Writing a Device Driver," and the mmap(D2) man page for more details.)

The section of a user-level program that maps device memory into its own addressing space could look like:

```
#include "fcntl.h"
#include "sys/mman.h"

fd = open(special_file, O_RDWR);
addr = mmap(0, len, PROT_READ|PROT_WRITE,
            MAP_PRIVATE, fd, off);
```

After the kernel performs basic sanity checking on the system call arguments, if the file descriptor passed to the **mmap**(2) system call represents a special file, the kernel looks in your driver object module and calls that device mapping function.

Synopsis

```
drvmap(dev, vt, off, len, prot)
       dev_t     dev;      /* device number*/
       vhandl_t  *vt;      /* handle to caller's
                           /* virtual address space */
       off_t     off;      /* offset into device */
       int       len;      /* # of bytes to map */
       int       prot;     /* protections */
```

Arguments

dev Gives your *drv***map**() function the device major and minor numbers. Use the **major** and **minor** macros to extract this information from *dev*.

vt Gives your *drv***map** function a pointer to the kernel-level data structure that describes the virtual space to which the device memory will be mapped. Your driver needs this pointer when calling certain kernel service functions.

Caution: Your driver must treat this pointer as an "opaque" handle and try not to set any of the member values directly. The specifics of this structure are likely to change from release to release. Your *drv***map**() function may change the member values of this structure indirectly, but only by calling kernel service functions.

off This offset within device memory, at which mapping begins, gives your *drv***map**() function the kernel's virtual address for the device.

len Gives your *drv***map**() function the length of the device memory to be mapped into the user's address space.

prot Gives your *drv***map**() function the protections that the user program specified when it called **mmap**().

munmap – Unmapping the Device

To unmap a device, the user program calls the **munmap**(2) system call:

`munmap (`*addr*`,` *len*`);`

where *addr* is the device virtual address returned by the **mmap**(2) function and *len* is the length of the mapped area. After performing device-independent unmapping in the user's space, the **munmap**(2) system call calls your driver's *drv***unmap**() function if it is defined as an entry in the device switch table.

Synopsis

```
drvunmap(dev, vt);
dev_t     dev;              /* device number */
vhandl_t  *vt;              /* handle to caller's virtual
                               address space */
```

The *vt* and *dev* parameters are the same as for *drv***map**(), above.

Note: If a driver provides a mapping function but does not provide an unmapping function, the **munmap**() system call returns the ENODEV error condition to the user. Therefore, it is a good idea for your driver to provide a dummy unmapping function even if your driver does not need to perform any action to unmap the device.

v_mapphys – Mapping Device Control Registers and On-board Memory

Your driver can allow the user to map device registers and local memory from the user's virtual space to physical memory if your driver's *drv***map**() function calls **v_mapphys**().

Synopsis

```
int v_mapphys(vt, addr, len);
              vhandl_t *vt;
              caddr_t addr;
              int len;
```

Arguments

vt — Your *drv***map**() function must give this parameter the opaque handle to the user's virtual address space. (The internals of the structure for the opaque handle are likely to change from release to release.)

addr — Your *drv***map**() function must give this parameter the kernel virtual address by which the device is accessed. (See "VME Slave Addressing" in Appendix A.) The system takes this address from the user's call to **mmap**(2) and hands it to the *off* parameter of your *drv***map**() function.

len — Your *drv***map**() function must give this parameter the number of bytes to be mapped. The system takes this value from the user's call to **mmap**(2) and hands it to the *len* parameter of your *drv***map**() function.

If successful, **v_mapphys**() returns 0. If **v_mapphys**() fails, it sets *errno* and returns -1. After a successful call to **v_mapphys**(), the device's registers at *addr* are mapped into the user's address space as designated by *vt*. You do not need any special unmapping, so the *drv***unmap**() function does not need actions specified within it.

Caution: Your driver must be very careful when it maps device registers to a user process. It must carefully check the range of addresses that the user requests and make sure that the request references only the requested device. Because protection is available only up to a page boundary, configure the addresses of I/O cards so that they do not overlap a page. If they are allowed to overlap, an application process may be able to access more than one device, possibly a system device (for example, a disk controller or Ethernet). This can cause system secuity problems or other problems that are hard to diagnose.

Sharing Kernel Memory with a User Program

To allocate memory that is shared between the driver and the application process, your *drv***map**() function must do steps 1 and 2 of the procedure below. To free that memory later, your *drv***unmap**() function must do step 3.

1. Use the **kmem_alloc**() kernel function to allocate some memory pages in the kernel:

   ```
   caddr_t *kaddr = kmem_alloc (len , KM_CACHEALIGN);
   ```

2. Map the memory pages into the user's address space by calling **v_mapphys**():

   ```
   v_mapphys (vt, kaddr, nbytes)
   ```

3. To free the memory, your driver's unmapping function, *drv***unmap**(), must call **kmem_free**():

   ```
   kmem_free(kaddr, len);
   ```

kmem_alloc() allocates physical memory and returns a kernel virtual address associated with that memory. The physical memory is not subject to paging. **v_mapphys**() returns 0 upon success and -1 upon failure. The parameters for these calls are:

vt — Your *drv***map**() function must give this parameter the opaque handle to the user virtual address space. (The internals of this structure are likely to change from release to release.)

kaddr — Your *drv***map**() function must give this parameter the virtual address returned by **kmem_alloc**(). Your *drv***unmap**() function must give this parameter the kernel virtual address returned by **kmem_alloc***()*.

len — Your *drv***map**() function must give this parameter the number of bytes to be mapped. (This value must not be greater than the number of pages allocated in step 1.)

Example Program

Suppose the mythical VT device wants to share memory with a user program. Its *drv***map**() and *drv***unmap**() functions would look something like this:

```
#include <sys/sysmacros.h>
struct mpd {
   unsigned int d_id;    /* id of memory segment */
   caddr_t  d_addr;      /* address of allocated memory */
   int      d_npages;    /* number of pages allocated */
   struct mpd *d_last, *d_next;   /* links */
};

struct mpd vdk_list;    /* at init, this becomes a doubly
                        /*linked ring */

int vdkmap(dev_t dev, vhandl_t *vt, off_t off, int len,
           int prot)
{
   struct mpd *d;

   /* initial sanity checking (not shown) */
   ...

   /* allocate some temporary storage */
   if( (d = kmem_alloc(sizeof(struct mpd)), 0 )
      == NULL )
      return ENOMEM;

   d->d_npages = btoc(len);
   if( (d->d_addr = kmem_alloc(ctob(d-
>d_npages))KM_CACHEALIGN) == NULL ) {
      kmem_free(d,sizeof(struct mpd));
      return ENOMEM;
   }

   /* map it into the user's address space */
   if( v_mapphys(vt,d->d_addr,len) ) {
      kmem_free(d->d_addr,ctob(d->d_npages));
      kmem_free(d,sizeof(struct mpd));
      return ENOMEM;
   }

   d->d_id = v_gethandle(vt);
```

```
    /* initialize the memory */
    bzero(d->d_addr,ctob(d->d_npages));

    /* add to the list */
    d->d_next = vdk_list.d_next;
    d->d_last = &vdk_list;
    d->d_next->d_last = d->d_last->d_next = d;

    return 0;
}
```

See “Returning Opaque Handle Data,” for a description of the **v_gethandle**() system function. In the **vdkunmap**() function, you can find the piece of memory to be deallocated by searching the above list. The driver can then call the **kmem_free**() kernel function with the address and length of this section of memory:

```
int vdkunmap(dev_t dev, vhandl_t *vt)
{
    struct mpd *d;
    int id;

    id = v_gethandle(vt);

    /* Find chunk of memory corresponding to it. */
    for(d = vdk_list.d_next; d != &vdk_list ; d = d->d_next )
        if( d->d_id == id )
            break;

    /* Make sure we found it. */
    if( d == &vdk_list )
        return 0;

    /* remove from list */
    d->d_next->d_last = d->d_last;
    d->d_last->d_next = d->d_next;

    /* free up resources */
    kmem_free(d->d_addr,ctob(d->d_npages));
    kmem_free(d,sizeof(struct mpd));

    return 0;
}
```

Returning Opaque Handle Data

Use the **v_gethandle** macro to get the unique identifier associated with *vt*, the opaque handle to the user's virtual address space. (The term “opaque” indicates that your code does not directly deal with the members of this structure, which is likely to change from release to release.)

```
#include "sys/region.h"
unsigned  v_gethandle(vt);
vhandl_t  *vt;
```

Because the virtual handle points into the kernel stack, it is likely to be overridden. Use **v_gethandle** if your driver must “remember” several virtual handles. Various other macros are also defined in *sys/region.h*, including macros that get the user virtual address to which the device space is mapped; macros that get the inode associated with the special file; and macros that get the length (in pages) of the user's mapped space.