Driver Example

This chapter displays the code of a complete device driver. The driver implements a "RAM drive," a block of memory that simulates a disk drive. Since it has no hardware dependencies, this example driver can be used for experimentation in any IRIX system.

Note: It is not sensible to use a RAM drive in a system like IRIX, where there is an effective implementation of virtual memory. The RAM drive only occupies memory that is better used as buffers for the paging system. This driver is useful only as a test-bed for experiments with the driver-kernel interface and with *symmon* and other debugging tools. Do not use this driver in a production system.

Installing the Example Driver

Use the following steps to install and test the example driver. Each step is expanded in the following topics.

1. Obtain the source code files.
2. Compile the source to obtain an object file.
3. Set up the appropriate configuration files.
4. Reboot the system and verify driver operation.
5. Install special device files and make and mount a filesystem.

Obtaining the Source Files

The example driver consists of the following four source files:

ramdrive.c	Source code of the executable module.
ramdrive.h	Header file containing a few declarations.
ramdrive	Descriptive file to place in */var/sysgen/master.d*
ramdrive.sm	Example VECTOR line for */var/sysgen/system*

These files (and other example code from this book) are available from the Silicon Graphics FTP server, ftp://ftp.sgi.com/sgi/

Alternatively, a patient person could recreate the files by copying and pasting from the online manual. However, owing to bugs in the Insight viewer support for X-paste, this is an error-prone process and is not recommended in the current release.

Compiling the Example Driver

Compile *ramdrive.c* using the techniques described under "Compiling and Linking" on page 228. An example compilation is shown in Example 12-1,

Example 12-1 Compiling the Example Driver for a 32-bit Kernel

```
% set CFLAGS = "-DDEBUG -D_K32U32 -D_KERNEL -DSTATIC=static
-D_PAGESZ=4096 -D_MIPS2 -DIP22 -DR4000 -G 0 -non_shared -elf -xansi
-fullwarn -32 -mips2 -Wc,-pic0"
% cc $CFLAGS  -c ramdrive.c
```

When the driver is compiled with the -DDEBUG option, all its informational displays are enabled. Without that option, it only displays messages related to serious errors during initialization.

Configuring the Example Driver

Before you configure the example driver into the kernel, you should set the system with a debugging kernel, as described under “Preparing the System for Debugging” on page 243.

Configure the example driver to IRIX by copying files as follows:

- Copy the *ramdrive.o* file to */var/sysgen/boot*.
- Edit the *ramdrive* descriptive file and make any necessary changes. For example:
  - The file specifies major device number 77. If there is another driver in the system using this major number, change 77 to any unused number in the range 60-79 (see “Major Device Number” on page 34).
  - The fourth option, #DEV, is set to 4. This controls how many unique minor devices the driver supports.
- Copy the edited *ramdrive* descriptive file to */var/sysgen/master.d*.
- Edit the *ramdrive.sm* file and make any desired changes. For example,
  - Change a *base*= value to change the size of a RAM drive
  - Add or remove VECTOR lines to change the number of minor devices that are initialized.
- Copy the edited *ramdrive.sm* file to */var/sysgen/system*.
- Reboot the system.

If you compiled the example driver with -DDEBUG, it displays several informational lines to the system console from its **rd_init()**, **rd_start()**, and **rd_edtinit()** entry points. The display from **rd_edtinit()** gives the address of the *rd_info_t* structure. You can display that area with *idbg*. Example 12-2 shows the result of displaying a simulated volume header structure, using an address displayed from **rd_edtinit()** (the actual address will differ).

Example 12-2 Displaying Simulated Volume Header Using idbg

```
# idbg hd 0x8841f604 0x80
0x8841f604: 0b e5 a9 41  00 00 00 01  00 00 00 00  00 00 00 00 ...A............
0x8841f614: 00 00 00 00  00 00 00 00  00 00 00 00  02 00 00 00 ................
0x8841f624: 00 01 00 00  00 00 00 08  02 00 00 01  00 00 00 00 ................
0x8841f634: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f644: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f654: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f664: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f674: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f684: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f694: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6a4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6b4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6c4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6d4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6e4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6f4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f704: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f714: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f724: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f734: 00 00 00 00  00 00 00 00  00 00 0f ff  00 00 00 01 ................
0x8841f744: 00 00 00 03  00 00 00 00  00 00 10 00  00 00 00 03 ................
0x8841f754: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f764: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f774: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f784: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 0f ff ................
0x8841f794: 00 00 00 01  00 00 00 03  00 00 00 01  00 00 00 00 ................
0x8841f7a4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f7b4: 00 00 10 00  00 00 00 00  00 00 00 06  00 00 00 00 ................
0x8841f7c4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f7d4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f7e4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f7f4: 00 00 00 00  00 00 00 00  f0 19 16 a5  00 00 00 00 ................
```

Creating Device Special Files

For each VECTOR line in */var/sysgen/system/ramdrive.sm* you can create a pair of device special files, one block device and one character device. Each file's inode contains the major device number established in descriptive file */var/sysgen/master.d/ramdrive*, and a minor number between 0 and 1 less than the limit established in the descriptive file.

The preferred command is *install* (see the install(1) reference page). The commands in Example 12-3 show the creation of the character and block devices for minor device number 0. The files created are */dev/ramblk0* and */dev/ramchr0*.

Example 12-3 Install Command to Create Device Special File

```
# install -u root -g sys -f /dev -blk 77,0 ramblk0
# install -u root -g sys -f /dev -chr 77,0 ramchr0
```

In addition to device special files, you need to create ordinary directories that can be used as mount points for the mounted filesystem, for example

```
# mkdir /RAM0
```

The example driver supports as many minor device numbers as you specify in the descriptive file */var/sysgen/master.d/ramdrive*. You can create as many device special files that contain the example driver's major number as you like—or as few. Device special files are independent of the driver configuration until they are opened. However,

- A device with a minor number in excess of the limit set in */var/sysgen/master.d/ramdrive* cannot be initialized with a VECTOR line in */var/sysgen/system/ramdrive.sm*, nor can it be opened.
- A device whose minor number was not initialized by a VECTOR line cannot be opened.

Verifying Driver Operation

You can verify operation of the driver by creating an EFS file system on one of its devices. As a first step, check the simulated volume header contents using *prtvtoc*, as shown in Example 12-4.

Example 12-4 Applying prtvtoc to a RAM Drive of 2 MB

```
# prtvtoc /dev/ramchr0
ramdrive open: flag 1 copen 1 bopen 0 xopen 0
DIOCGETVH on 20185088
ioctl copy kmem 8841f604 -> usr 10006868 for 512
ramdrive close: flag 1 copen 0 bopen 0 xopen 0
* /dev/ramchr0 (bootfile "")
*     512 bytes/sector
*       8 sectors/track
*       1 tracks/cylinder
*     512 cylinders
*       0 cylinders occupied by header
*     512 accessible cylinders
* No space unallocated to partitions
Partition  Type  Fs   Start: sec   (cyl)    Size: sec    (cyl)  Mount
Directory
 0          raw                1 (   0.1)        4095  ( 511.9)
 7          raw                1 (   0.1)        4095  ( 511.9)
 8       volhdr                0  (    0)           1  (   0.1)
10       volume                0  (    0)        4096   ( 512)
```

You can make an EFS filesystem on a RAM drive device by applying *mkfs* to the character special device (*mkfs* is applied to the character device because it uses **ioctl()**, which is not supported for block devices). This is shown in Example 12-5. The voluminous debugging displays have been truncated in the display.

Example 12-5 Making a Filesystem on a RAM Drive

```
# mkfs -t efs /dev/ramchr0
ramdrive open: flag 1 copen 1 bopen 0 xopen 0
DIOCGETVH on 20185088
ioctl copy kmem 8841f604 -> usr 10010c50 for 512
ramdrive close: flag 1 copen 0 bopen 0 xopen 0
ramdrive open: flag 1 copen 1 bopen 0 xopen 0
DIOCGETVH on 20185088
ioctl copy kmem 8841f604 -> usr 10010c50 for 512
ramdrive close: flag 1 copen 0 bopen 0 xopen 0
ramdrive open: flag 1 copen 1 bopen 0 xopen 0
DIOCGETVH on 20185088
ioctl copy kmem 8841f604 -> usr 10010c50 for 512
ramdrive close: flag 1 copen 0 bopen 0 xopen 0
ramdrive open: flag 1 copen 1 bopen 0 xopen 0
ramdrive close: flag 1 copen 0 bopen 0 xopen 0
ramdrive open: flag 3 copen 1 bopen 0 xopen 0
```

```
mkfs_efs: /dev/ramchr0: blocks=4095 inodes=616
mkfs_efs: /dev/ramchr0: sectors=8 cgfsize=4091
mkfs_efs: /dev/ramchr0: cgalign=1 ialign=1 ncg=1
mkfs_efs: /dev/ramchr0: firstcg=3 cgisize=154
mkfs_efs: /dev/ramchr0: bitmap blocks=1
rd_write entered for dev 20185088
rd_strategy: edev 20185088, flags 8018, blkno 3
           : offset 600, count 13400, dmaadr c026b610
        : write c026b610 to c0000600 for 13400
rd_write entered for dev 20185088
rd_strategy: edev 20185088, flags 18, blkno 0
           : offset 0, count 200, dmaadr 8b292cc0
        : write 8b292cc0 to c0000000 for 200
rd_read entered for dev 20185088
rd_strategy: edev 20185088, flags 19, blkno 3
           : offset 600, count 200, dmaadr c2c38a40
        : read c0000600 to c2c38a40 for 200
(many debugging displays omitted)
rd_write entered for dev 20185088
rd_strategy: edev 20185088, flags 8018, blkno ffe
           : offset 1ffc00, count 200, dmaadr c022def0
        : write c022def0 to c01ffc00 for 200
ramdrive close: flag 3 copen 0 bopen 0 xopen 0
```

After making a filesystem you can mount the filesystem, as shown in Example 12-6. You specify the block device when mounting an EFS filesystem, because the filesystem code calls the *pfx***size()** and *pfx***strategy()** driver entry points, which are supplied only by a block driver.

Example 12-6 Mounting a RAM Drive Filesystem

```
# mount -t efs /dev/ramblk0 /RAM0
ramdrive open: flag 3 copen 0 bopen 1 xopen 0
rd_size entered for dev 20185088
rd_strategy: edev 20185088, flags 9, blkno 1
           : offset 200, count 200, dmaadr 889f5800
        : read c0000200 to 889f5800 for 200
rd_strategy: edev 20185088, flags 9, blkno 2
           : offset 400, count 200, dmaadr 889f5a00
        : read c0000400 to 889f5a00 for 200
rd_strategy: edev 20185088, flags 8, blkno 2
           : offset 400, count 200, dmaadr 889f5a00
        : write 889f5a00 to c0000400 for 200
```

```
rd_strategy: edev 20185088, flags 9, blkno 3
           : offset 600, count 1000, dmaadr 88ef6000
        : read c0000600 to 88ef6000 for 1000
rd_strategy: edev 20185088, flags 1000008, blkno 1
           : offset 200, count 200, dmaadr 889f5800
        : write 889f5800 to c0000200 for 200
# df /RAM0
Filesystem             Type  blocks     use     avail  %use Mounted on
/dev/ramblk0            efs     3937       22     3915   1  /RAM0
```

Example Driver Source Files

The four source files of the example driver are displayed in the following topics:

- "Descriptive File" on page 278 displays the */var/sysgen/master.d* file that describes the driver to *lboot*.
- "System File" on page 279 displays the */var/sysgen/system* file that contains the VECTOR statements to initialize the driver.
- "Header File" on page 279 displays the driver's header file.
- "Source File" on page 281 displays the source file.

Descriptive File

```
*
* IRIX 6.2 Example driver "ramdrive" -- not for production use
*
* Flags used:
* b: block type device
* c: character type device (yes, both)
* d: dynamically loadable kernel module
* n: driver is semaphored
* s: software device driver
* w: driver is prepared to perform any cache write back operation
*
* External major number (SOFT) is an arbitrary choice from
* the range of numbers reserved for customer drivers.
*
* #DEV is passed in to the driver and used to configure its info array.
*
```

```
*FLAG   PREFIX  SOFT    #DEV    DEPENDENCIES
*bcdnsw  rd_     77      4
bcnsw  rd_     77      4

$$$

int rd_e_major = ##E;
int rd_numdevs = ##D;
int rd_ctrlrs = ##C;
```

System File

```
*
* Lboot config file for IRIX 6.2 example driver "ramdrive"
*   base= size of RAM drive, e.g. 0x00200000 is 2MB
*   ctlr= minor device number, 0 to 3
* Store as /var/sysgen/system/ramdrive.sm
*
VECTOR: module=ramdrive ctlr=0 base=0x00100000
*VECTOR: module=ramdrive ctlr=1 base=0x00080000
```

Header File

```
/**************************************************************************
 *                                                                        *
 *               Copyright (C) 1993, Silicon Graphics, Inc.               *
 *                                                                        *
 *  These coded instructions, statements, and computer programs  contain  *
 *  unpublished  proprietary  information of Silicon Graphics, Inc., and  *
 *  are protected by Federal copyright law.  They  may  not be disclosed  *
 *  to  third  parties  or copied or duplicated in any form, in whole or  *
 *  in part, without the prior written consent of Silicon Graphics, Inc.  *
 *                                                                        *
 **************************************************************************/

/**************************************************************************
| This is ramdrive.h, containing declarations that are used in both the
| driver and in the unit-test application code.  Aside from the unit-test
| module, no user-level code ever needs these declarations.  The ram drive
| is accessed through the file system like any other disk.
**************************************************************************/

/**************************************************************************
```

```
| The driver name for lboot and configuration is "ramdrive."
| The driver prefix is "rd_"
*************************************************************************/

#define DRIVER_PFX "rd_"
#define DRIVER_NAME "ramdrive"

/************************************************************************
| MAX_RD_DEVS declares the maximum number of distinct devices supported.
| Ram drive device special files are /dev/dsk/ramblk<n> for block devices
| and /dev/rdsk/ramchr<n> for character devices.  In each case <n> is the
| device minor number, between 0 and MAX_RD_DEVS-1.
************************************************************************/

#define MAX_RD_DEVS 4

/************************************************************************
| An array of MAX_RD_DEVS structures of the following type is maintained
| in the driver. VECTOR lines for up to MAX_RD_DEVS devices are written
| in /var/sysgen/system/ramdrive.sm, causing that many entries to the
| rd_edtinit() entry point, each entry initializing one structure.
|
| base:    address of allocated memory for the "drive."  If NULL, this
|          minor number has not been initialized or failed initialization.
|
| size:    size of the allocated memory in bytes, always rounded down to
|          a multiple of IO_NBPP.
|
| copen:   count of successful character opens, cleared to 0 in rd_close().
| bopen:   count of successful block opens (0 or 1), cleared in rd_close().
| xopen:   nonzero when an FEXCL open has succeeded.
|
| nmmap:   count of rd_map() entries, decremented in rd_unmap().  When the
|          any of copen, bopen, and nmaps over all devices is nonzero, the
|          driver returns EBUSY to the rd_unload() entry point.
|
| queue:   semaphore used to serialize access for reading and writing.
|          (Note however that in a multiprocessor, a user process can
|          perform an unsynchronized write to a mapped character device.)
|
| vh:      volume header structure prepared in edtinit(), and used to
|          initialize block 0 when "formatting" the drive.  The block-0
|          version can be modified by /etc/mkfs.
************************************************************************/
```

```
typedef struct rd_info {
    caddr_t     *base;
    off_t       size;
    __uint32_t  copen, bopen, xopen, nmmap;
    sema_t      queue;  /* requires sys/sema.h */
    struct volume_header vh; /* requires sys/dvh.h */
} rd_info_t;
```

Source File

```
/*************************************************************************
 *                                                                       *
 *                  Copyright (C) 1993, Silicon Graphics, Inc.           *
 *                                                                       *
 *  These coded instructions, statements, and computer programs  contain *
 *  unpublished  proprietary  information of Silicon Graphics, Inc., and *
 *  are protected by Federal copyright law.  They  may  not be disclosed *
 *  to  third  parties  or copied or duplicated in any form, in whole or *
 *  in part, without the prior written consent of Silicon Graphics, Inc. *
 *                                                                       *
 *************************************************************************/
/*************************************************************************
| This sample IRIX device driver implements a "ram disk" -- a block of
| kernel memory accessed as if it were a disk.  The driver supports both
| block and character interfaces and is loadable and unloadable.
|
| N   N   OO   TTTTT  EEEEE      It does not make sense to use a ram disk
| NN  N  O  O    T    E          in a system like IRIX that implements
| N N N  O  O    T    EEE   ::   effective virtual memory.  This device
| N  NN  O  O    T    E          driver is useful as an example because
| N   N   OO     T    EEEEE ::   it has no hardware dependencies, and so
|                                can be tried out in any IRIX system.
| However, this driver SHOULD NOT be employed in a production system!
| It WILL NOT give better performance.  It WILL consume kernel memory
| that would be better used for buffers.
|************************************************************************/

#include <sys/ddi.h>        /* gets also sys/types.h and sys/buf.h */
#include <sys/conf.h>       /* for driver flags D_MP etc */
#include <sys/kmem.h>       /* kmem_alloc and friends */
#include <sys/sema.h>       /* the rd_info_t contains a semaphore */
#include <sys/dvh.h>        /* the rd_info_t contains struct volume_header */
#include "ramdrive.h"       /* declare rd_info_t, etc. */
#include <sys/edt.h>        /* declare edt_t for edtinit() */
```

```
#include <sys/errno.h>      /* error codes to return */
#include <sys/cmn_err.h>    /* cmn_err() and related constants */
#include <sys/cred.h>       /* cred_t for prototypes */
#include <sys/dkio.h>       /* DKIOC* constants for ioctl */
#include <sys/param.h>      /* NBPSC bytes per sector */
#include <sys/immu.h>       /* IONBPP bytes per I/O page, btod() */
#include <sys/file.h>       /* FEXCL and other open flags */
#include <sys/open.h>       /* OTYP_CHR, OTYP_BLK */
#include <sys/region.h>     /* for vhandl_t */
#include <sys/mload.h>      /* only for M_VERSION */
/**************************************************************************
| Debug display macros: one each for cmn_err calls with 0, 1, 2, or 3
| variable arguments.
|**************************************************************************/
#ifdef DEBUG
#define DBGMSG0(s) cmn_err(CE_DEBUG,s)
#define DBGMSG1(s,x) cmn_err(CE_DEBUG,s,x)
#define DBGMSG2(s,x,y) cmn_err(CE_DEBUG,s,x,y)
#define DBGMSG3(s,x,y,z) cmn_err(CE_DEBUG,s,x,y,z)
#define DBGMSG4(s,x,y,z,w) cmn_err(CE_DEBUG,s,x,y,z,w)
#else
#define DBGMSG0(s)
#define DBGMSG1(s,x)
#define DBGMSG2(s,x,y)
#define DBGMSG3(s,x,y,z)
#define DBGMSG4(s,x,y,z,w)
#endif
/**************************************************************************
| Driver flag: this driver is MP-safe.  Also version flag for mload.
|**************************************************************************/
unsigned rd_devflag = D_MP;
char *rd_mversion = M_VERSION;
/**************************************************************************
| Array of rd_info_t objects, one per allowed minor device.  We rely on
| the loader to ensure these static globals are zero until initialized!
| Also defined: two convenience macros for frequent expressions.
|**************************************************************************/
static rd_info_t *rd_array;
#define INFOPTR(dev) &rd_array[geteminor(dev)]
#define VALIDIO(prd,off,len) (((off_t)(off) + (off_t)(len)) <= prd->size)
/**************************************************************************
| rd_basic() is called from rd_edtinit() to allocate the rd_array based
| on the global rd_numdevs, an integer set to ##D in the configuration
| file /var/sysgen/master.d/ramdrive.  Also display the other available
| globals for debugging purposes.
```

```
|*************************************************************************/
extern int rd_e_major, rd_numdevs, rd_ctrlrs;
int
rd_basic(void)
{
    if (!rd_array)
    {
        register int size;
        DBGMSG3("ramdrive basic: ##E=%d, ##D=%d, ##C=%d\n",
            rd_e_major, rd_numdevs, rd_ctrlrs);
        if (size = rd_numdevs*sizeof(rd_info_t))
            rd_array = (rd_info_t *)kmem_zalloc(size,KM_SLEEP);
        else
            cmn_err(CE_ALERT,"ramdrive: confused");
    }
    return (0 != rd_array);
}
/*************************************************************************
| rd_init() is included solely to demonstrate that this entry point
| can be called in addition to rd_edtinit() and rd_start().
|*************************************************************************/
int
rd_init(void)
{
    DBGMSG0("rd_init entry point called\n");
    return 0;
}
/*************************************************************************
| rd_start() is included solely to demonstrate that it, too can be called
| in addition to rd_edtinit() and rd_init().
|*************************************************************************/
int
rd_start(void)
{
    DBGMSG0("rd_start entry point called\n");
    return 0;
}
/*************************************************************************
| rd_format() is a subroutine of both rd_edtinit() and rd_ioctl() which
| "formats" the ramdrive to zeros with a reasonable volume header.
| The volume header (set in both the info struct and "sector 0")
| describes standard SGI partitions:
|   10 == the whole "drive"
|    8 == the volume header, only one sector in this case
|    7 == all sectors except the volume header
```

```
|     0 == data ("root") same as 7
|     1 == swap contains 0 sectors
| For versimilitude we arbitrarily say we have 1 track/cylinder
| and 8 sectors/track.  This assumes that nsectors is a multiple of 8,
| which is a good bet when the allocated size is a multiple of IO pages
| and sectors are 512 bytes.
|*********************************************************************/
void
rd_format(register rd_info_t *prd)
{
    register struct volume_header *pvh = &prd->vh;
    register int nsectors = btod(prd->size);/* immu.h */

    bzero((void *)pvh,sizeof(struct volume_header));
    pvh->vh_magic = VHMAGIC; /* in sys/dvh.h */
    pvh->vh_rootpt = 0;
    pvh->vh_swappt = 1;
    pvh->vh_dp.dp_cyls = nsectors/8; /* number of cylinders */
    pvh->vh_dp.dp_trks0 = 1; /* tracks/cyl */
    pvh->vh_dp.dp_secs = 8; /* sectors/track */
    pvh->vh_dp.dp_secbytes = NBPSCTR; /* param.h */
    pvh->vh_dp.dp_interleave = 1;

    pvh->vh_pt[10].pt_firstlbn = 0;
    pvh->vh_pt[10].pt_nblks = nsectors;
    pvh->vh_pt[10].pt_type = PTYPE_VOLUME;
    pvh->vh_pt[ 8].pt_firstlbn = 0;
    pvh->vh_pt[ 8].pt_nblks = 1;
    pvh->vh_pt[ 8].pt_type = PTYPE_VOLHDR;
    pvh->vh_pt[ 8].pt_firstlbn = 0;
    pvh->vh_pt[ 7].pt_firstlbn = 1;
    pvh->vh_pt[ 7].pt_nblks = nsectors-1;
    pvh->vh_pt[ 7].pt_type = PTYPE_RAW;
    pvh->vh_pt[ 0] = pvh->vh_pt[ 7];
    pvh->vh_pt[ 1].pt_firstlbn = nsectors;
    pvh->vh_pt[ 1].pt_nblks = 0;
    pvh->vh_pt[ 1].pt_type = PTYPE_RAW;

    pvh->vh_csum = -vh_checksum(pvh);
    bzero(prd->base,prd->size); /* clear all sectors */
    bcopy(pvh,prd->base,sizeof(prd->vh)); /* vh in sec 0 */
}
/*********************************************************************
| rd_edtinit() is called whenever the driver is loaded, once for each
| VECTOR that names this driver.  A typical VECTOR line would be:
```

```
|     VECTOR module=ramdrive ctrl=2 base=0x00040000
| which says, initialize minor number 2 for a size of 256K.
|************************************************************************/
int
rd_edtinit(register edt_t *pedt)
{
    register rd_info_t *prd;
    register __psint_t size;
    register int nsectors;
    register int ctlr = pedt->e_ctlr;
    /*
    || If this is the first time, allocate the rd_array of info structures.
    || Exit immediately if that fails.
    */
    if (!rd_basic())
    {
        return ENODEV;
    }
    DBGMSG3("ramdrive edtinit bustype %d adap %d ctlr %d\n",
        pedt->e_bus_type, pedt->e_adap, pedt->e_ctlr);
    DBGMSG3("                 e_space[0] iopaddr %x size %x vaddr %x\n",
        pedt->e_space[0].ios_iopaddr,pedt->e_space[0].ios_size,
        pedt->e_space[0].ios_vaddr);
    /*
    || Diagnose and reject an invalid minor dev# from VECTOR ctlr=
    */
    if (ctlr > rd_numdevs)
    {
        cmn_err(CE_ALERT,"ramdrive: ctlr=%d invalid minor dev#",ctlr);
        return ENODEV;
    }
    /*
    || Address the info structure and diagnose multiple initialization
    */
    prd = INFOPTR(ctlr);
    if (prd->base)
    {
        cmn_err(CE_ALERT,"ramdrive: duplicate VECTOR for ctlr=%d",ctlr);
        return EBUSY;
    }
    /*
    || The desired size of the ramdrive is encoded as the base=# value,
    || which is passed as the ios_vaddr value in the edt_t.
    || Diagnose 0 size (omitted base=). Round the size to a
    || multiple of *memory* (not necessarily I/O) pages.
```

```
    */
    size = (__psint_t) pedt->e_space[0].ios_vaddr;
    if ((0 == size)||(-1 == size))
    {
        cmn_err(CE_ALERT,
        "ramdrive: no size (base=) specified for ctlr=%d",ctlr);
        return EINVAL;
    }
    size = (size + (NBPP-1)) & (-NBPP); /* in sys/immu.h */
    /*
    || Allocate the kernel memory. Report an error if not possible.
    */
    prd->size = size;
    prd->base = kmem_alloc(size,KM_SLEEP);
    if (!prd->base)
    {
        cmn_err(CE_ALERT,
        "ramdrive: unable to allocate %x bytes for dev %d",size,ctlr);
        return ENOMEM;
    }
    nsectors = btod(size); /* immu.h bytes to disk sectors */
    DBGMSG3("ramdrive: dev# %d allocated %x = %x sectors\n",
                ctlr,size,nsectors);
    /*
    || Initialize the semaphore.
    */
    initnsema(&prd->queue,1,"ramdrive");
    /*
    || Initialize the "volume."
    */
    rd_format(prd);
    DBGMSG2("              info at 0x%x  vh at 0x%x \n",
        prd, (__psint_t)(&prd->vh) );
    return 0;
}
/***************************************************************************
| rd_open() is called for each open() of a character device /dev/ramchr<n>,
| and during a mount of a block device /dev/ramblk<n>.  We can distinguish
| between types of open from the otyp.
|**************************************************************************/
int
rd_open(dev_t *pdev, int oflag, int otyp, cred_t *pcred)
{
    register rd_info_t *prd = INFOPTR(*pdev);
    register int error = 0;
```

```
/*
|| Make sure the device being opened was initialized by a VECTOR.
*/
if (!prd->base)
{
    cmn_err(CE_NOTE,"ramdrive: open of uninitialized dev %d",*pdev);
    return ENODEV;
}
/*
|| Seize the device semaphore so that prd->rd_info can be updated
|| without error on a multiprocessor.
*/
psema(&prd->queue,PZERO+1 | PCATCH);
/*
|| Implement FEXCL (exclusive) open for a privileged process only.
|| Exclusivity applies to the entire minor device, under both its
|| block and character special devices.
*/
if (oflag & FEXCL)
{
    if (drv_priv(pcred)) /* not privileged */
    {
        DBGMSG0("ramdrive: reject FEXCL with EPERM\n");
        error = EPERM;
    }
    else if (prd->copen+prd->bopen+prd->nmmap) /* current use? */
    {
        DBGMSG0("ramdrive: reject FEXCL with EBUSY\n");
        error = EBUSY;
    }
    else
    {
        prd->xopen = oflag; /* note device open exclusively */
    }
}
else /* nonexclusive request can be blocked by exclusive open */
{
    if (prd->xopen)
    {
        DBGMSG0("ramdrive: reject normal open for exclusivity\n");
        error = EBUSY;
    }
}
if (!error)
{
```

```
        /*
        || Count the open so we don't unload with open devices.
        */
        if (otyp & OTYP_CHR)
            ++prd->copen;
        else
            ++prd->bopen;
        DBGMSG4("ramdrive open: flag %x copen %d bopen %d xopen %d\n",
            oflag, prd->copen, prd->bopen, prd->xopen);
    }
    vsema(&prd->queue);
    return error;
}
/**************************************************************************
| rd_close() is not called for each close() but for the final close of a
| given device (character or block).  Clear the respective count of opens
| and note whether exclusivity is being given up.  Since a close() in
| one CPU could happen concurrently with an open() in another CPU, we
| need to grab the semaphore before updating the rd_info.
| NOTE: the flag passed to close does not contain FEXCL even if it was
| given in the flag passed to open.
|*************************************************************************/
int
rd_close(dev_t dev, int flag, int otyp, cred_t *pcred)
{
    register rd_info_t *prd = INFOPTR(dev);
    psema(&prd->queue,PZERO+1 | PCATCH);
    if (flag & FEXCL)
    {
        /* this is never entered */
    }
    if (otyp & OTYP_CHR)
    {
        prd->copen = 0;
    }
    else
    {
        prd->bopen = 0;
    }
    /* if all opens are closed, an exclusive one is closed */
    prd->xopen = 0;
    vsema(&prd->queue);
    DBGMSG4("ramdrive close: flag %x copen %d bopen %d xopen %d\n",
            flag, prd->copen, prd->bopen, prd->xopen);
    return 0;
```

```
}
/*************************************************************************
| rd_ioctl() is called for ioctl(2), which can only be used on a character
| device. Disk ioctl command numbers for are in sys/dkio.h.
| DIOCREADCAPACITY: supported just for fun.
| DIOCGETVH: supported because /etc/mkfs and other tools use it (which
| explains why you apply mkfs to the character, not the block, device).
| DIOCSETVH: allows a program to change the "volume header" info.
| DIOCFORMAT: clears the device contents to 0, rewrites the vol header.
|
| The DIOC(S|G)ETVH calls use only the info in the per-device structure
| in memory. We make no attempt to keep that info in step with the
| contents of sector 0 of the simulated media.  This is consistent with
| other current IRIX disk drivers.  This has the implications that:
|   - you can change the driver's idea of the disk geometry on the fly,
|     without actually formatting the disk, this is useful for scsi.
|   - if you want to make a permanent change in the volume header,
|      -- one, that's a bad idea, use dvhtool(1) instead, but
|      -- two, if you insist, you need both a write to sector 0 and
|         a call to ioctl(,DIOCSETVH) to keep the driver up to date.
|
| Neither DIOCSETVH nor DIOCFORMAT hold the semaphore. You are strongly
| advised to do an exclusive open before calling them (but mkfp doesn't).
|************************************************************************/
int
rd_ioctl(dev_t dev, int cmd, caddr_t arg, int mode, cred_t *pcred, int *rval)
{
    register rd_info_t *prd = INFOPTR(dev);
    register int error = 0;
    register caddr_t kmemadr;
    register int len = 0;
    register int dir = 0; /* copyout */
    int capacity;
    switch(cmd)
    {
    case DIOCGETVH:
        {
            kmemadr = (caddr_t)(&prd->vh);
            len = sizeof(prd->vh);
            DBGMSG1("DIOCGETVH on %d\n",dev);
            break;
        }
    case DIOCREADCAPACITY:
        {
            capacity = prd->size/NBPSCTR;
```

```
                kmemadr = (caddr_t)(&capacity);
                len = sizeof(capacity);
                DBGMSG2("DIOCREADCAPACITY on %d = %d\n",
                            dev,capacity);
                break;
            }
        case DIOCSETVH:
            {
                kmemadr = (caddr_t)(&prd->vh);
                len = sizeof(prd->vh);
                dir = 1; /* copyin */
                DBGMSG1("DIOCSETVH on %d done\n",dev);
                break;
            }
        case DIOCFORMAT:
            {
                rd_format(prd);
                DBGMSG1("DIOCFORMAT done on %d!\n",dev);
                break;
            }
        default:
            {
                DBGMSG2("ramdrive invalid ioctl %x on %d\n",cmd,dev);
                error = EINVAL;
            }
        } /* switch(cmd) */
        /*
        || Perform the copy to or from user space if needed.
        */
        if ((!error) && (len))
        {
            if (!dir)
            {
                DBGMSG3("ioctl copy kmem %x -> usr %x for %d\n",
                    kmemadr, arg, len);
                error = copyout(kmemadr,arg,len);
            }
            else
            {
                DBGMSG3("ioctl copy usr %x -> kmem %x for %d\n",
                    arg, kmemadr, len);
                error = copyin(arg,kmemadr,len);
            }
    #ifdef DEBUG
            if (error)
```

```
            DBGMSG1("error %d on ioctl copy\n",error);
#endif
    }
    *rval = error; /* ensure user gets correct code */
    return error;
}
/***************************************************************************
| I/O Operations:
|
| rd_strategy() performs all actual I/O.  Called directly by file systems
| to read and write full I/O page units aligned on I/O page boundaries.
| Called indirectly to implement character I/O in any length and alignment.
|
| rd_read() and rd_write are called by read()/write() to a character
| device. They defer to rd_strategy via uiophysio().  This is consistent
| with the operation of other IRIX disk drivers.
|
| The strategy code simply does a bcopy. This is highly unrealistic.
| A real device driver would have to deal with efficient sequencing of
| track numbers and with asynchronous interrupts.
|***************************************************************************/
int
rd_strategy(register struct buf *pbuf)
{
    register rd_info_t *prd = INFOPTR(pbuf->b_edev);
    register __psint_t offset = pbuf->b_blkno * NBPSCTR;
    register __psint_t count = pbuf->b_bcount;
    register caddr_t target = (caddr_t)((__psint_t)prd->base)+offset;
    DBGMSG3("rd_strategy: edev %d, flags %x, blkno %x\n",
                    pbuf->b_edev,pbuf->b_flags,pbuf->b_blkno);
    DBGMSG3("           : offset %x, count %x, dmaadr %x\n",
                    offset,count,(caddr_t)pbuf->b_dmaaddr);
    if (!VALIDIO(prd,offset,count))
    {
        DBGMSG0("rejecting strategy with ENOSPC\n");
        pbuf->b_error = ENOSPC;
        iodone(pbuf);
        return 0;
    }
    /*
    || Ensure that pbuf->b_dmaaddr is a valid kernel address.
    || This is never needed when called via uiophysio, only when
    || called from the file system or paging subsystem.  (Goodness!
    || wouldn’t it be fun to use a ramdrive for swapping?)
    || NOTE: while a simple bp_mapin() call works, this approach
```

```
    || would impose unnecessary overhead in a real driver when
    || the device does not support scatter/gather.
    */
    if (!BP_ISMAPPED(pbuf))
    {
        bp_mapin(pbuf);
        DBGMSG1("        : after bp_mapin dmaadr %x\n", pbuf->b_dmaaddr);
    }
    /*
    || Grab the device semaphore. Note: this ensures consistency
    || between reads and writes, but does not control modifications
    || made through memory-mapped access.
    */
    psema(&prd->queue,PZERO+1 | PCATCH);
    /*
    || Perform the "read" or "write."
    */
    if (pbuf->b_flags & B_READ)
    {
        DBGMSG3("        : read %x to %x for %x\n",
            target,pbuf->b_dmaaddr,pbuf->b_bcount);
        bcopy(target,pbuf->b_dmaaddr,pbuf->b_bcount);
    }
    else
    {
        DBGMSG3("        : write %x to %x for %x\n",
            pbuf->b_dmaaddr,target,pbuf->b_bcount);
        bcopy(pbuf->b_dmaaddr,target,pbuf->b_bcount);
    }
    vsema(&prd->queue);
    iodone(pbuf);
    return 0;
}
int
rd_read(dev_t dev, uio_t *puio, cred_t *pcred)
{
    DBGMSG1("rd_read entered for dev %d\n",dev);
    return uiophysio(rd_strategy,0,dev,B_READ,puio);
}
int
rd_write(dev_t dev, uio_t *puio, cred_t *pcred)
{
    DBGMSG1("rd_write entered for dev %d\n",dev);
    return uiophysio(rd_strategy,0,dev,B_WRITE,puio);
}
```

```
int
rd_size(dev_t dev)
{
    DBGMSG1("rd_size entered for dev %d\n",dev);
    return rd_array[geteminor(dev)].size/NBPSCTR;
}
/**************************************************************************
| Memory mapping: rd_map() (one "m") is called to implement an mmap()
| request on a character device.  We permit read and write mappings, which
| means that in a multiprocessor, one CPU could be updating the kernel
| memory that represents the medium while another CPU executes a read()
| on the same memory.
|
| Since a map can persist after the corresponding FD is closed, we
| keep track of mappings separately from opens.
**************************************************************************/
int
rd_map(dev_t dev, vhandl_t *pvh, off_t off, int len, int prot)
{
    register rd_info_t *prd = INFOPTR(dev);
    int error;

    DBGMSG3("map request on %d at %x for %x\n",dev,off,len);
    if (VALIDIO(prd,off,len))
    {
        error = v_mapphys(pvh,prd->base+off,len);
#ifdef DEBUG
        if (error)
            DBGMSG1("v_mapphys returns %d\n",error);
#endif
    }
    else
    {
        DBGMSG0("rejecting map with ENOSPC\n");
        error = ENOSPC;
    }
    if (!error)
        ++prd->nmmap;
    return error;
}
rd_unmap(dev_t dev, vhandl_t *pvh)
{
    register rd_info_t *prd = INFOPTR(dev);
    if (prd->nmmap)
    {
```

```
        --prd->nmmap;
        DBGMSG2("unmap on %d, map count now %d\n",dev,prd->nmmap);
    }
    else
    {
        DBGMSG1("unmap on %d when map count 0 ?!?!?!?\n",dev);
    }
    return 0;
}
/**************************************************************************
| Unload support: rd_unload() is called when ml(1) is asked to unload
| this driver.  We test to make sure that none of our devices that have
| been initialized, are in use.  When any are in use, we return EBUSY
| and so will not be unloaded.
**************************************************************************/
int
rd_unload(void)
{
    int j;
    for (j = 0; j<rd_numdevs; ++j)
    {
        if (( rd_array[j].base )
        &&  ( rd_array[j].copen
            ||rd_array[j].bopen
            ||rd_array[j].nmmap) )
        {
            DBGMSG1("rejecting unload because dev %d busy\n",j);
            return EBUSY;
        }
    }
    DBGMSG0("accepting unload, byeeeee\n");
    return 0;
}
```