3 Character Drivers

In this chapter we will develop a simple character driver named SCULL (for Simple Character Utility for Loading Localities; again, don't blame me for the name). As in the previous chapter, the module we are going to develop in this chapter is a rough translation of the C code found in Rubini & Corbet's *Linux Driver Drivers (Second Edition)* text. However, unlike the *skull* driver of the previous chapter (which was almost a line for line translation of the C code), the *scullc* driver I'm going to present in this chapter is quite a bit less capable than the C version of LDD2. This has nothing to do with the language choice; rather, it's an issue of pedagogy. The *scullc* driver that R&C present is far more sophisticated than it needs to be; indeed, it presents a good example of software engineering and design. However, their *scullc* driver, like the one of this chapter, is really nothing more than a 'toy' device driver that is useful mainly for teaching how to write character device drivers. While one could argue that all the additional complexity of their device driver forces you to think about the problem more thoroughly, I also feel that their driver contains a lot of code that really has nothing to do with writing device drivers. Therefore, I've opted to write a *scullc* driver that has many limitations compared with their driver. After all, you're not going to actually use the *scullc* driver in your system other than to test out creating a character driver; so any superfluous code that exists in the driver serves only to get between you and learning how to write Linux character device drivers in assembly language. Hopefully, I've boiled the *scullc* device driver down to its essence; the minimal code you need to write a character device driver.

3.1 The Design of scullc

In LDD2, R&C have chosen to develop a single device driver that implements several different views of the device to real-world users. In a real-world device driver, this is exactly what you want to do and I heartily recommend that you read Chapter Three in LDD2 to see how to create "polymorphic" device drivers that present several different types of devices to Linux through the use of minor numbers. For simplicity, however, I will implement each of these device drivers as separate drivers within this text

. This allows us to focus on the examples at hand (while presenting the source code for the full example) without having to deal with extra code intended for use by other forms of the *scullc* device. Although this chapter does not ultimately merge the device drivers together (leaving that as an exercise for the user), doing so is not difficult at all. Please check out the *scull* design in LDD2 for more details.

This chapter presents the code for up to four instances of a simple memory-based character device that we'll refer to as *scull0, scull1, scull2*, and *scull3*. Note that these are four independent devices that are all handled by the same device driver, *scullc*. These devices have the following characteristics:

- The devices are memory-mapped. This means that data you write to the device is stored in sequential memory locations and data you read from the device is read from those same sequential memory locations. R&C's *scullc* driver provides some sophisticated dynamic memory management that allows the size of the buffer area to expand and contract as applications use this device. Such memory management would be necessary if *scullc* were a real device; however, since *scullc* is really just a demonstration of a device driver, I felt that this sophisticated memory management was completely unnecessary and provides very little additional knowledge[1]. In place of dynamic memory allocation of the *scullc* devices, the *scullc* this chapter presents uses fixed 16 kilobyte blocks to represent the devices. This would be a severe limitation in the real world, but keep in mind that this is not a real-world device driver and 16K is more than enough memory to test out the operation of the device driver.
- The *scullc* memory device is global. This means that if an application (or group of applications) opens the device multiple times, the various file descriptors that share the device see the same data (rather than a separate instance for each open file descriptor).

1. It does teach you how to use kmalloc and kfree; but I promise I'll teach you about those memory management functions in the chapter without the complexity of a dynamic *scullc* device.

- The *scullc* memory device is persistent. This means that data one application writes to the device remains in the internal memory buffer until either another application overwrites the data, you remove the *scullc* driver from memory, or the system loses power.

The interesting thing about the scullc device is that you can manipulate it without resorting to writing special applications that use the device. Any application that writes (less than 16K of) text to a file or reads data from a file can access the device. For example, you can use commands like *ls, cp,* and *cat* to manipulate this device (and instantly see the device operate).

3.2 Major and Minor Numbers

An application accesses a character device driver using 'filenames' in the file system. These are the filenames of special device files and you normally find them in the */dev* subdirectory. The following is a (very) short list of some of the files you will typically find in the /dev subdirectory:

```
crw--w----    1 root     root       4,   0 Apr 18 14:16 /dev/tty0
crw-------    1 root     root       4,   1 Apr 18 14:16 /dev/tty1
crw-------    1 root     root       4,   2 Apr 18 14:16 /dev/tty2
```

The 'c' appearing in this first column of the '*ls -l /dev/tty[0-2]*" display indicates that tty0, tty1, and tty2 are all character devices. Other device types would place a different character here (e.g., block devices place a 'b' in column one).

Note the two columns of numbers between the columns containing 'root' and 'Apr'. These numbers are the major device number and the minor device number, respectively. In the listing above, the major number is 4 (all three devices) and the minor numbers are 0, 1, and 2. The major number identifies the device driver associated with a device. Therefore, tty0, tty1, and tty2 all three share the same device driver, since they have the same major number. The kernel uses the major number, when you open a file, to determine which device driver should receive the open call.

The kernel simply passes the minor number along to the device driver. The kernel, in no way, manipulates or notes the value of the minor number. This value is strictly for use by the device driver. Typically, a device driver will use the minor number to differentiate between several instances of the same device type. For example, if your computer has two peripheral adapters with the same hardware, the system will only need one device driver (and one major number) to control the devices. The device driver can use the minor number to differentiate between the two actual devices.

Both the major and minor numbers are eight-bit values at this time. This is a severe limitation that Linux kernel developers are trying to remedy by the v2.6 kernel release, but so many applications and device drivers out there are 'aware' of the internal representation of device numbers under Linux that this is going to be a difficult task to achieve. The problem is that there are far more than 256 devices available for PCs today. Unfortunately, Linux inherited this problem from older UNIX variations and, as you may recall, UNIX originated on old DEC equipment way back in the 1970's. Back then, there weren't many devices available for machines like PDP-8's and PDP-11's (the original machines on which UNIX ran). Therefore, 256 seemed like infinity at the time (also, it was the case that holding both the major and minor number in a single word of memory was a big deal because systems didn't have much memory at the time (4K words was a typical system). Today, of course, memory and devices are plentiful, but we're still suffering from the limitations inherited from the original UNIX systems.

To add a new device to the system you need to assign a major number to that device. You should do this within the init_module procedure by calling the following function[2]:

```
procedure linux.register_chrdev
(
            major   :dword;
            _name   :string;
    var     fops    :linux.file_operations
);
```

2. This is actually a macro, not a procedure, see *kernel.hhf* for more details.

This function returns a result in EAX. If the return result is negative, then there was some sort of error. If the function returns zero or a positive value, then the registration of the major number was successful. The major parameter is the major number you're requesting (which, even though it's a dword parameter, must be a value in the range 0..255). The _name parameter is a string containing the name of the device. Linux will display this name as the device name when you list the contents of the */proc/devices* file. The fops parameter is a pointer to a table of addresses that provide the optional kernel entry points to your driver. We'll discuss this parameter a little later in this chapter.

Note that you only pass a major number to register_chrdev. As noted earlier, the kernel doesn't use the minor value, so linux.register_chrdev has no need for it.

Note that applications open your device by specifying its name, not a major/minor number combination. Therefore, we need some way to identify this device by name rather than number. Unfortunately, the name you pass as a string to linux.register_chrdev does not do the trick. This is because that string specifies a driver name rather than an explicit device name. Remember, a single device driver can actually control multiple devices concurrently (via the minor number). Applications will need a separate name by which they can reference each device instance that your device driver can handle. To properly access a device driver, an application references a special file name in the */dev* subdirectory, so we need some way to create new file entries in the */dev* subdirectory and we need some way to connect entries in the */dev* subdirectory with our device drivers. This is done using the *mknod* (make device node) utility. The *mknod* utility, which only the superuser may execute, uses the following syntax:

```
mknode  /dev/devicename  c  major  minor
```

For example, if you want to create a *scull0* device that responds to major number 254 and minor number zero, you'd use the following command:

```
mknode /dev/scull0 c 254 0
```

The 'c' parameter, as you might guess, tells Linux that this is a character device (as opposed to a block device, which uses a 'b' in this argument position).

Note that *mknod* simply creates a special file associated with a character or block device. It doesn't require that there actually be a device driver that responds to the major number when you execute *mknod* (though you must have installed a device driver that responds to the major number before you attempt the first open of this device). After you execute a mknod command like the one above, you should be able to see the device/special file in the */dev* subdirectory by simply issuing an "*ls -l /dev*" command.

Once you've created a special device file with mknod, it remains in the system until you explicitly remove it with the *rm* command. If you've actually executed the command above to test the creation of a special device file, now would be a good time to delete that file so that it doesn't create any conflicts with a device. To do so, just type:

```
rm /dev/scull0
```

3.3 Dynamic Allocation of Major Numbers

The "biggie" device drivers (disks, consoles, serial ports, parallel ports, etc.), all have fixed major device numbers assigned to them. You can find a list of the statically allocated devices in the *Documentation/devices.txt* file of your Linux source distribution. As you can see by looking at this file, a large percentage of the device numbers are already assigned (and re-assigned!), so finding a major device number that doesn't conflict with some other device is difficult. Unless you're planning to use your device driver only on your personal system, simply choosing a number of a device that isn't present in your system will not suffice. Someone else who wants to use your device driver might very well have that equipment installed and a conflict could occur. Fortunately, there is a way out: dynamic allocation of major numbers.

If you pass zero as the major number to register_chrdev, this function will not use zero as the major number; instead, it uses the value zero as a flag to indicate that the caller is requesting that register_chrdev pick a handy, currently unused, major device number. When you call register_chrdev in this manner, the

function returns the major number as the function return result. Since major numbers are always positive, you can differentiate a dynamically assigned major number from an error code by checking the sign of the return result; if it's positive, it's a valid major number; if it's negative, then it's an error return value. Note that if you specify a non-zero major number when calling register_chrdev, then the function returns zero (rather than the major number) to indicate success.

For private use, dynamically allocated major numbers are the best way to go. However, if you're planning on supplying your device driver for mainstream use by Linux users, you'll probably have to request a major number for specific use by your device. The *documentation/devices.txt* file mentioned earlier discusses the procedure for doing this.

The problem with dynamically assigned major numbers is that you'll not be able to create the device file entry in the */dev* subdirectory prior to actually installing your driver (since you won't know the major number, something that *mknod* requires, until you've successfully installed the driver). Once your driver is installed, however, you can read its (dynamically assigned) major number from the /proc/devices file. Here's a (truncated) example of that file as it appeared on my system while writing this:

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 14 sound
 29 fb
 36 netlink
128 ptm
136 pts
162 raw
180 usb
253 scullc
254 iscsictl

Block devices:
  1 ramdisk
  2 fd
  8 sd
  9 md
 22 ide1
 65 sd
 66 sd
```

As you can see in this example, this file is broken into to sections: character and block devices. In each section, the lines list the currently installed devices. The first column for each entry is the major number and the second column is the driver's name. As you can see in this example, the scullc device has been (dynamically) assigned the major number 253. You'll notice in the listing above that block devices and character devices share some common major numbers. Linux uses separate tables for character and block devices, so the major character numbers are distinct from the major block numbers (note that Linux uses the major number as an index into the table of devices; since there are two separate tables, the indicies are independent).

Although you cannot create a special device file prior to actually loading your device, you may create the special file immediately after installing your device driver. In fact, you can create an installation script that automates the installation of your driver and the creation of the special device file. The following is a slight modification of the *scullc_load* Linux shell script that Rubini and Corbet provide:

```
#!/bin/sh
module="scullc"
device="scullc"
```

```
mode="666"


# remove stale nodes (scull0, scull1, scull2, scull3):

rm -f /dev/${device}?

# invoke insmod with all arguments we got ("$*")
# and use a pathname, as newer modutils don't look in . by default

/sbin/insmod -f ./$module.o $* || exit 1

# Search for the major number associated with the module
# we just installed and set the shell variable 'major' to this
# value:

major=`cat /proc/devices | awk "\\$2==\"$module\" {print \\$1}"`

# Create the four scull devices (scull0, scull1, scull2, scull3)
# using the major number we obtained from the above shell cmd:

mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3
ln -sf ${device}0  /dev/${device}

# give appropriate permissions so anyone can write to this device:

chmod $mode  /dev/${device}[0-3]
```

Listing 4.1 Linux scullc_load Shell Script to Install and Initialize a Driver

This shell script creates four devices (corresponding, to minor numbers 0, 1, 2, and 3) because the *scullc* driver we're going to develop in this chapter directly supports four devices. If you decide you want fewer or more *scullc* devices, simply change the number of *mknod* statements above (of course, you'll need to make a one-line change to the *scullc.hhf* header file, too). Note that this script is easy to modify to install any driver by simply changing the *module, device*, and *mknode* lines in this script.

Of course, you'll probably want a script that automatically cleans up the device files when you want to remove the device driver. R&C provide the following *scullc_unload* shell script that does the trick:

```
#!/bin/sh
module="scullc"
device="scullc"

# invoke rmmod with all arguments we got
/sbin/rmmod $module $* || exit 1

# remove nodes
rm -f /dev/${device}[0-3] /dev/${device}

exit 0
```

Listing 4.2 Linux scullc_unlock Shell Script to Unload a Driver

Rubini & Corbet suggest that the best way to assign major numbers is to have linux.register_chrdev generate a dynamic major number by default, but allow the system administrator to specify a fixed major number when loading the driver. The code to accomplish this is similar to that for autodetection of port numbers given in the previous chapter. Using this approach, the system administrator can specify a major number on the *insmod* command line (or, by modifying the *scullc_load* script above, by supplying the major number on the *scullc_load* command line). If the optional command line isn't present, the system can default to dynamic major number assignment.

Assuming we use the dword variable scullc_major to hold the major number parameter, we can do this dynamic assignment using code like the following:

```
// Note: scull_major is a static dword variable initialized to zero
// by the compiler and possibly set to some other value specified by
// an insmod command line parameter.

linux.register_chrdev( scull_major, "scullc", scullc_fops );
if( (type int32 eax ) < 0 ) then

    // There was an error.

    kdebug( linux.printk( "<1>scullc: can't get major %d\n", scull_major ));

elseif( scull_major = 0 ) then

    mov( eax, scull_major );

endif;
```

3.4 Removing a Driver From the System

Just before a module unloads itself from the system, it must relinquish its major number. You'll generally do this in the cleanup_module procedure by calling the following function:

```
procedure linux.unregister_chrdev( major:dword; _name:string );
```

The parameters are the major number and device name that you passed to the linux.register_chrdev function in the module_init code. This function returns a negative error code in EAX if either parameter does not agree with the original values.

You must always ensure that you unregister the major number before your driver unloads itself. Failure to do so leaves the kernel in an unstable state that may cause a segmentation fault the next time someone reads the */proc/devices* file. The best solution is to simply reboot the system and correct your driver so that it properly unregisters the major number.

When you unload a driver, you should also delete the device files for the driver you've just removed. The script given earlier does this task automatically if you use it to unload the driver and remove the device files. For more details, as always, see LDD2.

3.5 dev_t and kdev_t

Historically, UNIX has used the dev_t data type to hold the major and minor numbers as a single unit. As UNIX grew up on limited-memory 16-bit systems, it's not at all surprising to find that the original UNIX

designers made the major and minor values eight-bit values and combined them into a single data type: *dev_t*. The major number value is kept in the high order byte and the minor number is kept in the low order byte. As noted earlier, the Linux system needs more than 256 major numbers (and there are good reasons for extending the number of minor numbers, too), but too many application programs know the internal data format for *dev_t* objects and insist on manually extracting the major/minor numbers (without relying on a library routine to do the work for them). As a result, any attempt to change the underlying *dev_t* will break a large number of applications.

Despite the problems with changing the dev_t type, there are plans to extend the major and minor numbers to 16 bits (each). According to R&C, doing so is a stated goal for the Linux v2.6 release. In preparation for this change, kernel developers have defined a new type, kdev_t, that holds the major and minor number. Currently (for compatibility with existing code), kdev_t is a 16-bit value with the major and minor values appearing in the H.O. and L.O. bytes, just like dev_t. However, the Kernel developers have decreed that no software (including the kernel) is to know the underlying data type. Instead, all access to kdev_t objects must be through a set of macros that extract the major/minor numbers, construct a kdev_t object from a major/minor number pair, and convert between integer and kdev_t types. These macros appear in the *kernel.hhf* header files and are the following:

```
major( kdev_tVal )
minor( kdev_tVal )
mkdev( maj, min )
kdev_t_to_nr( kdev_tVal )
to_kdev_t( i )
```

These macros have been carefully written to be very efficient so that assembly programmers will avoid the temptation to access the bytes of a kdev_t object directly. To ensure future compatibility with the Linux kernel, you should always use these macros when manipulating kdev_t objects. Of course, as an assembly language programmer, you may feel it's your right to chose how you'll access the data (and this is true), just keep in mind that the data structure is going to change soon and your code will break if it's aware of kdev_t's internal representation. You have been forewarned.

The major macro accepts a kdev_t value as an argument and returns the major number associated with that value in the EAX register. This macro is quite sophisticated as far as the *linux.hhf* macros go. It allows constant parameters, register parameters, and kdev_t memory arguments. It carefully considers the type of the argument and attempts to generate the best code that will load EAX with the major number component of the argument. There is one exception, however, if you specify a constant kdev_t value as an argument, then the major macro returns a constant (rather than the EAX register). This allows you to use the major macro in constant expressions when supplying a constant argument (btw, the only legitimate way to supply a constant to this macro is via some constant you've initialized with the mkdev macro; since other constants require that you know the internal representation of a kdev_t value). If major returns a constant value, it must be the operand of some other machine instruction (since the macro generates no machine code otherwise), e.g.,

```
mov( major( $1234 ), ebx ); // current equals mov( $12, ebx );
```

The source code for the major macro is interesting reading for those who'd like to see how a macro can check the type of the argument the caller supplies.

The minor macro is just like the major macro except, of course, it returns the minor number (as a constant or in EAX) rather than the major number. You should always use this macro to extract the minor number from a kdev_t object.

The mkdev macro takes two integer constants (currently they must be in the range 0..255) that represent the major and minor numbers, and combines them to form a kdev_t value in EAX, unless both operands are constants, in which case the mkdev macro returns a constant (see the discussion of major for more details).

The kdev_t_to_nr and to_kdev_t macros translate to/from kdev_t and dword objects. They generally compile to a single instruction that leaves the value in EAX.

3.6 File Operations

As you may recall from the discussion of linux.register_chrdev earlier, your driver communicates the facilities it provides to the kernel (beyond init_module and cleanup_module) using the fops argument to linux.register_chrdev. In this section we'll discuss the fops table (type file_operations).

The file_operations data type is a table of function pointers. Each entry in this table contains either NULL (zero) or the address of some procedure that provides some sort of functionality for the device driver. Linux checks the entries in this table when invoking a particular device function. If the entry of interest contains zero, the Linux will perform some default action (like returning an error code); if the entry is non-zero, then Linux will call the procedure at the address specified by this table entry. As a device driver writer, it is your responsibility to decide which functionality you're going to provide, write the procedures to provide this functionality, fill in a file_operations table with the addresses of the functions you've written, and then pass the address of this table to the linux.register_chrdev during driver initialization.

We'll discuss the meaning of each table entry (and its corresponding function) momentarily, but first we need to have a brief discussion of the file_operations data type and it's impact on the way you write your code.

The file_operations data structure is not a static object across different Linux versions. As Linux matures, kernel developers add new fields to this structure and they rearrange fields in this structure (see "Version Dependency and Installation Issues" on page 20 for the reasons and the resulting problems). Here's what the structure looked like in Linux v2.4.7-10:

```
// file_operations record for drivers:

file_operations:record
    owner       :dword;  // Pointer to module_t;
    llseek      :procedure
                (
                   file    :dword; // Pointer to file
                   offset  :qword; // 64-bit offset
                   whence  :dword  // Type of seek.
                ); @cdecl;

    read        :procedure
                (
                       file    :dword; // Pointer to file
                       buf     :dword; // Buffer address.
                       size    :dword; // Size of transfer
                   var offset  :qword  // Store new ofs here.
                ); @cdecl;

    write       :procedure
                (
                       file    :dword; // Pointer to file
                       buf     :dword; // Buffer address.
                       size    :dword; // Size of transfer
                   var offset  :qword  // Store new ofs here.
                ); @cdecl;

    readdir     :procedure
                (
                   file    :dword; // Pointer to file.
                   buf     :dword; // data buffer.
                   count   :dword  // ignored (?)
                ); @cdecl;

    poll        :procedure
```

```
                (
                   file:dword;
                   poll_table_struct:dword
                ); @cdecl;

_ioctl      :procedure( inode:dword; file:dword ); @cdecl;

mmap        :procedure
                (
                   file:dword; // pointer to file
                   vmas:dword  // pointer to vm_area_struct
                ); @cdecl;

open        :procedure
                (
                   inod    :dword; //pointer to inode
                   file    :dword  //pointer to file
                );  @cdecl;

flush       :procedure
                (
                   file    :dword  //pointer to file
                ); @cdecl;

release     :procedure
                (
                   inod    :dword; //pointer to inode
                   file    :dword  //pointer to file
                );

fsync       :procedure
                (
                   inod    :dword; //pointer to inode
                   de      :dword; //pointer to dentry
                   datasync:dword
                ); @cdecl;

fasync      :procedure
                (
                   fd      :dword; //file descriptor
                   file    :dword; //pointer to file
                   on      :dword
                );  @cdecl;

lock        :procedure
                (
                   file    :dword; //file pointer
                   typ     :dword;
                   filock  :dword  //pointer to file_lock
                ); @cdecl;

readv       :procedure
                (
                   file    :dword; //pointer to file
                   iov     :dword; //pointer to iovec
                   count   :dword;
                   offs    :dword
                ); @cdecl;

writev      :procedure
                (
```

```
                        file    :dword; //pointer to file
                        iov     :dword; //pointer to iovec
                        count   :dword;
                        offs    :dword
                    ); @cdecl;

        sendpage    :procedure
                    (
                            file    :dword; // Pointer to file.
                            thePage :dword; // Pointer to page struct.
                            pgNum   :dword; // ???
                            size    :dword;
                        var offset  :qword
                    );

        get_unmapped_area:
                    procedure
                    (
                        file    :dword;
                        u1      :dword;
                        u2      :dword;
                        u3      :dword;
                        u4      :dword
                    );
    endrecord;
```

Listing 4.3 Linux v2.4.7-10 file_operations Data Structure

HLA provides the ability to define record constants, so you could initialize the table of pointers with the addresses of several procedures using code like the following:

```
    scullc_fops     :linux.file_operations :=
                        linux.file_operations:
                        [
                            0,                  // Owner
                            &scullc_llseek,     // llseek
                            &scullc_read,       // read
                            &scullc_write,      // write
                            0,                  // readdir
                            0,                  // poll
                            0,                  // ioctl
                            0,                  // mmap
                            &scullc_open,       // open
                            0,                  // flush
                            &scullc_release,    // release
                            0,                  // fsync
                            0,                  // fasync
                            0,                  // lock
                            0,                  // readv
                            0,                  // writev
                            0,                  // sendpage
                            0                   // get_unmapped_area
                        ];
```

Listing 4.4 Example Code That Initializes a file_operations Record

In this particular example (which just happens to be reasonable for the SCULL driver we're developing in this chapter), table entries for the llseek, read, write, open, and release functions are provided. The driver using this table will not support any of the other (optional) device driver functions.

Consider, however, what will happen in a different version of Linux if the kernel developers decide to rearrange a couple of entries in the structure (and also assume that the appropriate header file contains the change). As long as the number of entries in the structure are the same, HLA will continue to compile your code without complaint. However, the device driver won't work proper. To understand why, let's assume that the kernel developers decided to swap the llseek and read entries in the structure (so the kernel calls the read function using the second field in the table and it calls llseek using the third field in the table above). If your file operations table contains the entries above, the kernel will actually call read when it attempts to call llseek and vice-versa. Obviously, this is a problem.

This problem exists in C as well as assembly. However, Linux kernel programmers working in C can take advantage of a GCC extension that allows them to specify a record constant using a label notation rather than positional notation for each of the fields. That is, rather than having to specify all 18 fields in the file_operations struct (and in the order they're declared), GCC programmers need only provide the initializers for the non-NULL fields in the struct and they may specify the fields in any order. To give GCC a clue about what it's supposed to do with the struct constant, the programmer must attach a label to each field constant they supply, e.g.,

```
struct file_operations fops =
       {
           llseek: &seekfunc,
           open: &openfunc,
           read: &readfunc,
           write: &writefunc,
           release: &releasefunc
       };
```

When GCC sees the field labels, it automatically moves the value (following the label) to its appropriate spot in the struct and fills the empty slots with zero bytes (NULL). This feature that GCC provides is very powerful and makes it much easier to maintain device drivers across Linux versions. Now the kernel developers can move the fields in file_operations around to their heart's content and the device driver author need only compile the module against the new header files and GCC does the rest.

Unfortunately, HLA does not provide this nifty feature for initializing record constants. However, HLA does provide a very powerful compile-time language and macro facility that lets you create a macro that provides this same functionality. The following listing provides an HLA macro[3] that you can use to initialize a file_operations constant:

```
// The fileops_c macro allows the user to create
// a file_operations record constant whose fields
// are specified by name rather than position, e.g.,
//
//  linux.fileops_c
//  (
//      read:&readproc,
//      open:&openproc,
//      release:&releaseproc,
//      llseek:&llseekproc
//  );
```

3. This macro is a part of the *linux.hhf* header file set, so you don't actually have to enter this code into your drivers. Just refer to this macro by linux.fileops_c.

```
        //
        //  Entries that are absent in the list are filled with NULLs.
        //  The entries may appear in any order.
        //
        // Using this macro rather than a file_operations record
        // constant to initialize a file_operations variable helps
        // reduce maintenence of your driver when the file_operations
        // record structure changes (as it does every now and then).

    const
        _fops_fields:= @locals( file_operations );

        #macro fileops_c(__ops[]):
                __opsIndex,
                __exit,
                __syntaxError,
                __namesIndex,
                __maxIndex,
                __maxNames,
                __curLine,
                __curVal,
                __curName,
                __curField,
                __thisField;

            // This is a file_operations record constant, so output
            // some syntax to begin the constant:

            linux.file_operations:[

            // Now generate the "guts" for this constant:

            ?__curVal   :string;
            ?__curName  :string;
            ?__maxIndex := @elements( __ops );
            ?__maxNames := @elements( linux._fops_fields );
            ?__namesIndex := 0;
            ?__syntaxError := false;
            #while( __namesIndex < __maxNames & !__syntaxError )

                ?__curField := linux._fops_fields[ __namesIndex ];
                ?__opsIndex := 0;
                ?__exit := false;
                ?__thisField := "0";
                #while( __opsIndex < __maxIndex & !__exit )

                    ?__curLine :string := __ops[ __opsIndex ];
                    ?__exit :=
                        @uptoChar
                        (
                            __curLine,
                            ':',
                            __curVal,
                            __curName
                        );

                    #if( !__exit )

                        #error
                        (
                            "Syntax error in file_operations constant: "+
```

```
                    __curLine
                )
                ?__exit := true;
                ?__syntaxError := true;

            #else

                ?__curName := @trim( __curName, 0 );
                ?__exit := __curName = __curField;
                #if( __exit )

                ?__thisField := @substr( __curVal, 1, 1024 );

                #endif

            #endif
            ?__opsIndex += 1;

        #endwhile

        // If not the first table entry, emit a comma:

        #if( __namesIndex <> 0 )
            ,
        #endif

        // emit the table entry:

        @text( __thisField )

        ?__namesIndex += 1;

    #endwhile

    // Okay, close up the constant:

    ]

    // Now, to be on the safe side, verify that there
    // weren't any extra fields in the parameter list:

    ?__opsIndex := 0;
    #while( __opsIndex < __maxIndex & !__syntaxError )

        ?__namesIndex := 0;
        ?__exit := false;
        #while( __namesIndex < __maxNames & !__exit )

            ?__exit :=
                @uptoChar
                (
                    __ops[ __opsIndex ],
                    ':',
                    __curVal,
                    __curName
                );

            ?__curName := @trim( __curName, 0 );
            ?__exit :=
                __curName = linux._fops_fields[ __namesIndex ];
```

```
                ?__namesIndex += 1;

            #endwhile
            #if( !__exit )

                #error
                (
                    "Unexpected field in fileops_c (" +
                    __curName +
                    ")"
                )

            #endif

            ?__opsIndex += 1;

        #endwhile

    #endmacro;
```

Listing 4.5 linux.fileops_c Macro to Initialize file_operations Constants

AA brief description of how this macro operates may be helpful to those who are interested in creating their own macros to initialize records in this manner.

First of all, you use this macro in the declaration section as the following example demonstrates:

```
static
    fops :linux.file_operations :=
            linux.fileops_c
            (
                llseek: &seekfunc,
                open: &openfunc,
                read: &readfunc,
                write: &writefunc,
                release: &releasefunc
            );
```

Notice that the argument list for the macro uses the exact same syntax as the C example given earlier. Hopefully, this demonstrates the power of the HLA compile-time language: if HLA doesn't provide some facility you'd like, it's fairly easy to come up with a compile-time program (e.g., macro) that extends the language in exactly the way you want. Kernel C programmers generally have to live with the features provided by GCC; kernel assembly programmers don't, they can easily extend HLA as they wish.

To understand how this macro works, first consider the following statement:

```
const
    _fops_fields := @locals( file_operations );
```

The @locals compile-time function returns an array of strings with each string containing one of the field names of the record you pass as an argument[4]. The important thing to note is that element zero of this string array constant contains the name of the first field, element one contains the name of the second field, etc.

The fileops_c macro supports a variable argument list. You may specify zero or more parameters to this macro (specifying zero arguments will initialize the file_operations constant with all zeros). Each parameter must take the following form:

4. @locals will also return the names of local symbols in a procedure as well as the fieldnames of a union object.

```
fieldName : initial_value
```

The fieldname label must be one of the fieldnames in the file_operations record. The initial_value component should be the name of an HLA procedure with the address-of operator ("&") as a prefix to the value. If a given parameter does not take this form, then the macro will report an error (see the code that sets the __syntaxError compile-time variable to true for details).

For each field in the file_operations record, the macro scans through all the macro parameters to see if it can find an argument whose label matches the current field name. This processing is done by the first two (nested) #while loops appearing in the code. The outermost #while loop steps through each of the file_operations fields, the inner-most #while loop steps through each of the macro parameters. Inside that innermost #while loop, the code uses the @uptoChar, @trim, and @substr compile-time functions to extract the label and the value from the parameter string (see the HLA documentation for more details on these functions). The macro then compares the label it just extracted with the current field name and emits the value it extracted if the field name matches the label.

The second pair of nested #while loops scan through the names and macro arguments a second time searching for any labels that are not field names in the record. Without this second pair of loops, the macro would quietly ignore any incorrect field names you try to initialize (e.g., typographical errors). With these last two loops present, the macro will report an error if it encounters an unknown field name.

Since the linux.fileops_c macro is so convenient to use (much easier than manually supplying a file_operations record constant), there probably isn't any reason you'll not use it. But most importantly, you should use this macro because it will help make your device drivers easier to maintain across Linux kernel versions.

Having described how to initializes the file_operations record object that you must pass to linux.register_chrdev, perhaps it's now time to describe the more-important functions whose addresses you place in the file_operations record. The following sub-sections will handle that task.

3.6.1 The llseek Function

```
procedure llseek( var f:linux.file; offset:linux.loff_t; whence:dword );
    @cdecl;
    returns( "edx:eax" );
```

This procedure is responsible for changing the read/write position in a file. Note that the linux.loff_t type is a 64-bit unsigned integer type (qword). Note that C code calls this function, so you must declare your code using the @cdecl procedure option (or manually handle accessing parameters on the stack). Also note that this procedure returns the new 64-bit offset in EDX:EAX.

As far as Linux is concerned, it believes that it is calling a function written in C that exhibits the appropriate C calling conventions. In addition to the @cdecl parameter passing convention, GCC also assumes that the llseek procedure preserves all registers except EAX, EDX, and ECX (obviously, you don't preserve EDX and EAX because the function returns the new 64-bit offset in these two registers; however, you should note that you do not have to preserve the ECX register). If you use EBX, ESI, or EDI, you must preserve those registers[5].

If your driver has a problem on the llseek call, it should return one of the negative error codes found in the errno namespace (see *errno.hhf*). Don't forget that all error codes are negative, so you must return a negative value in EDX:EAX. Since all the errno codes are less greater than -4096 (and less than zero), they fit easily into EAX. However, you must return a 64-bit negative number, so don't forget to sign extend EAX into EDX after loading EAX with the negative error code (either use CDQ or move -1 into EDX).

Providing an llseek function, like all other file_operations functions, is optional. If you place a NULL in the llseek field, then Linux will attempt to simulate llseek for you. This simulation is done for seeks relative to the beginning of the file or from the current file position by simply updating the position in the f (linux.file) variable (described a little later) while seeks relative to the end of the file always fail.

5. Ditto for EBP and ESP, but it's a really bad idea to attempt to use these registers as general purpose registers.

3.6.2 The read Function

```
procedure read
(
    var    f      :linux.file;
    var    buf    :var;
           size   :dword;
    var    offset :linux.loff_t
);
    @cdecl;
    @returns( "eax" );
```

The read function is responsible for transferring data from the device to a user buffer (specified by buf). The read operation starts at the file offset specified by the 64-bit value pointed at by offset. Note that, *and this is very important*, buf is the address of a buffer in user space. You cannot directly dereference this pointer. We'll get into the details of user-space access a little later in this chapter. Just keep in mind that you cannot simply load buf into a 32-bit register an use register indirect addressing on that pointer.

The size parameter specifies the number of bytes to transfer from the device to the user's buffer. Although Linux supports files that are much larger than 4GB, the size field is only 32-bits. The reason should be obvious: the user's address space is limited to 4GB (less, actually) so there is no way to transfer more than 4GB in one operation; in fact, few devices would support such a large, continuous, transfer, anyway. So 32 bits is fine for the size parameter.

The offset parameter is a pointer to the current file position associated with the read. This is usually (but not always) the f_pos field of the f parameter that Linux also passes in. However, you should never update f.f_pos directly, always use the offset pointer to update the file position when you're done moving data from the device to the user's buffer. Generally, updating the file position consists of adding the value of the size parameter to the 64-bit value pointed at by offset.

If this function succeeds, it returns a non-negative value the specifies the number of bytes that it actually transferred to the user's buffer. If this function fails, it must return an appropriate negative error code in EAX indicating the problem.

If the read field in the file_operations record contains zero (NULL), then Linux will return errno.einval whenever an application attempts to read data from the device driver.

3.7 The write Function

```
procedure write
(
    var    f      :linux.file;
    var    buf    :var;
           size   :dword;
    var    offset :linux.loff_t
);
    @cdecl;
    @returns( "eax" );
```

The write function is responsible for transferring data from the user buffer (specified by buf) to the device starting at the file offset specified by the 64-bit value addressed by offset. Note that, *and this is very important*, buf is the address of a buffer in user space. You cannot directly dereference this pointer. We'll get into the details of user-space access a little later in this chapter. Just keep in mind that you cannot simply load buf into a 32-bit register an use register indirect addressing on that pointer.

The size parameter specifies the number of bytes to transfer from the user's buffer to the device. Although Linux supports files that are much larger than 4GB, the size field is only 32-bits. The reason should be obvious: the user's address space is limited to 4GB (less, actually) so there is no way to transfer

more than 4GB in one operation; in fact, few devices would support such a large, continuous, transfer, anyway. So 32 bits is fine for the size parameter.

The offset parameter is a pointer to the current file position associated with the write operation. This is usually (but not always) the f_pos field of the f parameter that Linux also passes in. However, you should never update f.f_pos directly, always use the offset pointer to update the file position when you're done moving data from the device to the user's buffer. Generally, updating the file position consists of adding the value of the size parameter to the 64-bit value pointed at by offset.

If this function succeeds, it returns a non-negative value the specifies the number of bytes that it actually transferred to the device. If this function fails, it must return an appropriate negative error code in EAX indicating the problem.

If the write field in the file_operations record contains zero (NULL), then Linux will return errno.einval whenever an application attempts to write data to the device driver.

3.8 The readdir Function

This function is only used for file systems, never by device drivers. Therefore, this field in the file_operations record should contain NULL (zero).

3.8.1 The poll Function

```
procedure poll( var f:linux.file; var poll_table_struct:dword );
    @cdecl;
    @returns( "eax" );
```

The Linux system calls poll and select ultimately call this function for the device. This function returns status information indicating whether the device can be read, can be written, or is in a special state. The poll function returns this information as a bitmap in the EAX register (we'll talk about the explicit bit values later). Note that the second parameter is actually a pointer to the poll record data; we'll discuss it's layout later.

If there is no poll function entry in the file_operations table, then Linux treats the device as though it's always readable, always writeable, and is not in any kind of special state.

3.8.2 The _ioctl Function

```
procedure _ioctl( var ino:linux.inode; var f:linux.file );
    @cdecl;
    returns( "eax" );
```

The _ioctl function is a "catch-all" function that lets you send device-specific commands from an application to a device driver. Note that the true fieldname is "ioctl" in the C code; the "_" prefix appears in the HLA code because "ioctl" is a macro for the ioctl system call and HLA will expand that macro directly for the ioctl fieldname in the file_operations structure, thus creating all kinds of havoc. Hence the "_" prefix on this field name.

Like most Linux functions, this function returns a negative value to indicate an error. Whatever non-negative value you return in EAX, Linux returns on through to the application that made the original ioctl system call.

If the _ioctl entry is NULL in the file_operations table, the Linux returns an errno.enotty error code[6] to the application that invoked ioctl.

6. See LDD2 for the reasons behind this choice of an unusal error return value.

3.8.3 The mmap Function

```
procedure mmap( var f:linux.file; vmas:dword );
    @cdecl;
    returns( "eax" );
```

This function is called when an application is requesting that the system map the device's memory to a portion of the application's address space. We'll cover this function in detail later on in this text.

If the mmap field of the file_operations table contains NULL, then Linux returns errno.nodev to the caller when they attempt to make this call.

3.8.4 The open Function

```
procedure open( var ino:linux.inode; var f:linux.file );
    @cdecl;
    returns( "eax" );
```

Before an application can talk to a device driver, it must first open the file associated with that device. When the application does that, Linux ultimately calls this function (if you've implemented it). We'll cover this function in depth a little later in this chapter, so there's no need to exhaustively explain what you can do with it here. Note that if the file_operations record entry for open contains NULL, then Linux always informs the application that opens the device that the open has succeeded. Generally, though, you'll want some indication that an application has opened the device for use, so you'll probably implement this function.

3.8.5 The flush Function

```
procedure flush( var f:linux.file );
    @cdecl;
    returns( "eax" );
```

This call informs the device that it should wait for the completion of any I/O on the device prior to returning. Applications call this function indirectly whenever they close a file. If the field entry in the file_operations table for flush contains NULL, the system simply doesn't call flush.

3.8.6 The release Function

```
procedure release( var ino:linux.inode; var f:linux.file);
    @cdecl;
    returns( "eax" );
```

The system calls this function after the last process sharing a file structure closes the file (generally, there is only one process using the file structure, but fork and dup operations allow multiple processes to share the same open file). If the corresponding entry in the file_operations table is NULL, then Linux simply doesn't call this function. Normally, however, you will want to implement this function.

3.8.7 The fsync Function

```
procedure fsync( var ino:linux.inode; var de:linux.dentry; datasync:dword );
    @cdecl;
```

```
    returns( "eax" );
```

Whenever an application makes an fsync system call on a file associated with a device, that system call ultimately winds up making this call to the corresponding device driver. This function should flush any pending data to/from the device prior to returning. If the file_operations table doesn't implement this call, Linux returns errno.einval to the caller.

3.8.8 The fasync Function

```
procedure fasync( fd:dword; var f:linux.file; on:dword );
    @cdecl;
    returns( "eax" );
```

Linux calls this function in your driver to notify it of a change in its FASYNC flag. We'll discuss the issue of asynchronous notification in a later chapter on advanced character devices. If you choose not to implement this function, then your device driver will not support asynchronous notification.

3.8.9 The lock Function

```
procedure lock( var f:linux.file; ltyp:dword; var fl:linux.file_lock );
    @cdecl;
    returns( "eax" );
```

This function implements file locking. Most device drivers don't deal with locking and, therefore, leave this function undefined.

3.8.10 The readv and writev Functions

```
procedure readv
(
    var    f      :linux.file;
    var    iov    :linux.iovec;
           count  :dword;
           offs   :linux.loff_t
);
    @cdecl;
    returns( "eax" );

procedure writev
(
    var    f      :linux.file;
    var    iov    :linux.iovec;
           count  :dword;
           offs   :linux.loff_t
);
    @cdecl;
    returns( "eax" );
```

These functions implement scatter/gather read/write operations. This provide high-performance read/write operations when writing blocks of memory that are scattered around the system. If the file_operations table contains NULL in their corresponding fields, the Linux implements readv and writev by making successive calls to read and write.

3.8.11 The owner Field

The owner field in the file_operations table is not a procedure pointer like the other fields. Instead, this field contains a pointer to the module that owns this structure. Linux uses this pointer to gain access to the module so it can maintain that module's usage counter.

When initializing the file_operations table, you should always initialize the owner field as well. To do so, you can simply specify "owner : &linux.__this_module" in the fileops_c initializer. This only works in Linux v2.4 or later; but since this text only deals with device drivers for Linux v2.4 or later, this isn't a problem.

3.9 The file Record

After the linux.file_operations record, the linux.file record is probably the second most important data structure that Linux device driver authors will use. The linux.file structure is what the Linux kernel uses to track open files in the system; indeed, the file descriptor value that Linux returns to an application is really nothing more than an index that selects a linux.file structure inside the kernel. Note that the linux.file structure really has little to do with the C Standard Library FILE object; this is a kernel-only data structure, user-space applications do not have access to the linux.file objects.

The linux.file structure is is actually quite large, containing many fields, since the kernel uses it to keep track of information about any open file (not just device driver files). Therefore, a lot of the information held within the linux.file structure is of no interest to device driver writers (indeed, the kernel doesn't use many of the fields when using the structure to represent an open device file); furthermore, many of the fields are for internal kernel use only and, likewise, are of no interest to device driver authors. Still, there are many fields that a device driver writer will need to access. This section briefly describes many of those fields.

3.9.1 file.f_mode : linux.mode_t

The file mode field contains two bits, that you can test with the bitmask values linux.fmode_read and linux.fmode_write, that determine whether the file is readable, writable, or both. Note that you do not need to test these bits from your read and write functions since the kernel verifies that the file is readable before calling read and writable before calling write. However, if you do any read or write operations from another call (e.g., _ioctl) then you may want to check these bits bits for their proper values before doing the operation.

3.9.2 file.f_pos : linux.loff_t

This is the current reading or writing position associated with this file variable. This is a 64-bit value since many devices support files larger than 4GB. Your driver may read this value (to determine the current file position) but it should never directly write to this field. Linux will pass the address of this field (or a comparable field) to those calls that must update the file's read/write position. This is necessary because Linux sometimes holds the file position value in a different location and if you update this field directly you may not be updating Linux's idea of the file position.

3.9.3 file.f_flags : dword

Linux stores the file open flags, like linux.o_rdonly, linux.o_wronly, linux.o_nonblock, and linux.o_sync, in this field. Your driver, however, should not check these flags to see if the file was opened for reading or writing; use the f_mode field (above) for this purpose. Drivers will typically check this field for nonblocking operations and nothing else.

3.9.4 file.f_op : linux.file_operations

This is a pointer to the linux.file_operations table for the device associated with this file. The kernel assigns a value to this field as part of the open operation and then never writes to this field again. From that point forward, the kernel simply dispatches device I/O calls through this table when there is an I/O request. As a result, your driver can change the value of this field to redirect I/O operations through a different table. Many of the drivers in this text will do exactly this when dealing with a driver that has multiple personalities (selected via the minor device number). This is a common way of implementing several behaviors in a device driver without any overhead; by changing this pointer, the kernel automatically calls the appropriate handler routines without any code ever having to test the minor device number (and, thus, incurring additional overhead).

3.9.5 file.private_data : dword

The (kernel) open system call sets this field to NULL prior to calling the open function associated with the driver. Other than that, the kernel ignores this field. The device driver is free to use this field for any purpose it chooses. For example, the driver may allocate some storage via the linux.kmalloc function and store a pointer to the storage in the private_data field. Or the driver can use this field to point at some other important data structure in the system. In the *scullc* driver we're developing in this chapter, for example, the driver will use this field to point at the *scull* device data (that is specific to the scullc driver).

3.9.6 file.f_dentry : linux.dentry

The f_dentry field hols a pointer to the directory entry associated with the file. Device driver writers rarely deal with directory entries (since most devices do not support directories). However, the d_inode field of the f_dentry object is a pointer to the inode associated with the file. Accessing the inode structure is common in device drivers, so a device driver writer will often use the f_dentry field to gain access to the inode information.

3.10 Open and Release

Now that we've briefly covered the linux.file_operations and linux.file data structures, we can begin discussing some of the functions associated with the entries in the linux.file_operations table. Since the open and release functions are among the most fundamental, it's a good idea to start with the explanation of these two procedures.

3.10.1 The Open Procedure

The kernel calls the device driver's open procedure to allow the device driver to do any device-specific it requires after the kernel has done much of it's work associated with opening the file (e.g., filling in the linux.file data structure). A typical device driver will do the following in its open procedure:

- Check for device-specific errors (such as hardware not operational).
- Initialize the device if the system is opening it for the first time.
- Identify the minor number and do whatever is necessary to handle the particular sub-device (e.g., update the f_op pointer, if necessary).
- Allocate and initialize any data structure associated with the private_data field.

In kernels earlier than version 2.4, it was also the responsibility of the device driver to increment the module's usage count. However, as of Linux v2.4, the kernel started handling this job automatically. You may see some device drivers doing this anyway (it doesn't hurt), but it's unnecessary in Linux v2.4 and later.

A typical device driver will first look at the minor number to determine exactly what initialization it should do upon entry to the open procedure. Many of the drivers in this text, that use the minor number to provide different device behaviors, will do exactly that. For more details on this, please see LDD2.

The *scullc* driver that this chapter presents uses the minor number to select from among four identical *scullc* devices (*scull0, scull1, scull2*, and *scull3*). Since the devices are identical, there is no need to change the f_ops pointer in the linux.file structure, but the private_data field must point at a separate scull_device object for each of the four devices. So the *scullc* open procedure uses the minor number to select which scull_device pointer it stores into the private_data field.

The following listing is the code for the *scullc* open procedure:

```
// scullc_open-
//
//  Handles the open call to this device.

procedure scullc_open
(
    var inode   :linux.inode;
    var filp    :linux.file
);
const
    sdECX       :text := "(type scull_dev [ecx])";

begin scullc_open;

    reg_save( ecx, edx );

    // Get the kdev_t value (device number) from the
    // inode.i_rdev field and extract this device's
    // minor number:

    mov( inode, eax );
    linux.minor( (type linux.inode [eax]).i_rdev );

    kdebug( linux.printk( "<1>scullc_open, minor#=%d\n", eax ) );

    begin exitOpen;

        // Verify that the minor number is within range for
        // this device:

        if( eax >= scullc_devs_c ) then

            kdebug
            (
                linux.printk( "<1>scullc_open:minor # too big\n" )
            );
            mov( errno.enodev, eax );  // No such scull device!
            exit exitOpen;

        endif;

        // Compute the address of the scull[minor] device
        // structure in the scullc_devices array.  Get the
        // address of the particular scullc device into ECX:

        intmul( @size( scull_dev ), eax );
        lea( ecx, scullc_devices[eax] );
```

```
            // Initialize the filp->private_data field so that it
            // points at the particular device data for the device
            // selected by the minor number:

            mov( filp, eax );
            mov( ecx, (type linux.file [eax]).private_data );

            // Trim the length of the device down to zero
            // if the open was write-only:

            mov( (type linux.file [eax]).f_flags, edx );
            and( linux.o_accmode, edx );
            if( edx = linux.o_wronly ) then

                kdebug
                (
                    linux.printk
                    (
                        "<1>scullc_open: opened write-only\n"
                    )
                );

                // If someone is already accessing the data structures
                // for this device, make the kernel wait until they
                // are through.

                if( linux.down_interruptible( sdECX.sem ) ) then

                    mov( errno.erestartsys, eax );
                    exit exitOpen;

                endif;

                // Okay, clear out the existing data in the device:

                scullc_trim( sdECX ); // ignore any errors.

                // Free the semaphore we grabbed above.

                linux.up( sdECX.sem );

            endif;

            xor( eax, eax );    // return success

        end exitOpen;

        kdebug
        (
            linux.printk( "<1>scullc_open: exit (returns %d)\n", eax )
        );
        reg_restore;

    end scullc_open;
```

Listing 4.6 The scullc open Procedure

The first thing to take notice of are the paramters to open: inode and filp. The inode parameter is a pointer to the linux.inode_t structure associated with this device. The piece of information we need from this structure is the i_rdev field that holds the kdev_t device number. The filp parameter is the linux.file structure whose fields we need to access (and in some cases, initialize) for the current open operation.

The first statement in the body of the procedure is the following:

```
reg_save( ecx, edx );
```

reg_save is a macro I've written to make it easier to preserve and restore registers in code. Note that reg_save is not a part of the *linux.hhf* header file set – this macro declaration actually appears within the *scullc* body (though this macro is so genuinely useful, it probably should become part of the HLA Standard Library header files). The reg_save macro emits code to push all the parameters you supply onto the 80x86 stack. It also keeps track (in a global HLA compile-time variable) of all the registers you specify in the argument list. Later in the code, you can restore all the register by simply invoking the reg_restore macro as follows:

```
reg_restore;
```

Note that you don't have to supply any parameters to the reg_restore macro. This macro accesses the information that reg_save keeps in the global compile-time variable to figure out which registers to pop from the stack (and the order by which reg_save should pop them). The operation of reg_save and reg_restore is relly handy because it makes maintaining your code a whole lot easier. Whenever you need to change the regsiters you push onto the stack at the beginning of the procedure, you don't have to search for each location where you pop the data (and this may occur in more than one location as you'll see in the open procedure). Instead, just add (or remove) the affected registers in the reg_save invocation and the reg_restore(s) within that same procedure will automatically emit code to deal with your changes. This can help prevent several programming errors that occur when you miss popping a register you've added to your list of pushed registers (or forgetting to remove a pop from some pop sequence when you remove a push). For those who are interested in how this operates, the following listing provides the code for these macros:

```
val
    lastRegSet  :dword;  //Keeps track of registers saved by reg_save

// reg_save - pushes a set of 32-bit registers
// onto the stack and saves the register set so that
// reg_restore can automatically pop them.

#macro reg_save( regs[] ):rindex, rmax, reg;
    ?rmax   := @elements( regs );
    #if( rmax > 0 )
        ?lastRegSet :string[ rmax ];
    #endif
    ?rindex := 0;
    #while( rindex < rmax )

        ?reg :text := regs[ rindex ];
        #if( @isreg( reg ))

            #if( @size( reg ) = 4 )

                push( reg );

            #else

                #error( "Expected a 32-bit register" )

            #endif
```

```
        #else

            #error
            (
                "Expected a 32-bit register, encountered: '" +
                regs[ rindex ] +
                "'"
            );

        #endif
        ?lastRegSet[ rindex ] := regs[ rindex ];
        ?rindex := rindex + 1;

        // We have to change reg to string before next loop
        // iteration, or the assignment to reg above will choke.

        ?@tostring:reg :string;

    #endwhile

#endmacro;

// reg_restore - pops the registers last pushed by
// the reg_save macro.  This macro doesn't do any
// error reporting because that was done by reg_save.

#macro reg_restore:rindex, reg;
    ?rindex :int32  := @elements( lastRegSet ) - 1;
    #while( rindex >= 0 )

        ?reg :text := lastRegSet[ rindex ];
        #if( @isreg( reg ))

            #if( @size( reg ) = 4 )

                pop( reg );

            #endif

        #endif
        ?rindex := rindex - 1;
        ?@tostring:reg :string;

    #endwhile

#endmacro;
```

Listing 4.7 The reg_save and reg_restore Macros

Note that Linux assumes that the open procedure it's calling is a C function. This means that it thinks that the procedure can wipe out the EAX, EDX, and ECX registers. Therefore, the "reg_save(ecx, edx);" macro invocation, strictly speaking, isn't necessary. However, the assembly programmer's convention is (generally) to preserve all registers the procedure modifies (and doesn't use to return values; EAX is the function return result in this example which is why open doesn't preserve EAX). Therefore, you may remove the reg_save and reg_restore invocations if you're really concerned about the few extra bytes and cycles the code associated with these macros consumes. I stick these instructions in the code simply because preserving registers is a good habit to have and maintain. Furthermore, open doesn't get called that often, the the

few cycles needed to push and pop two registers is of little consequence to the running time of the function (nor are the four bytes the pushes and pops consume).

The next couple of instructions extract the minor number from the inode.i_rdev field. This code also writes out the minor number to the log file if debugging is enabled (by setting KNDEBUG false). I'll have more to say about this debugging output a little later in this chapter. Immediately after extracting the minor number, the code also checks the minor number to make sure it is valid (in the range 0..3 for the scullc device):

```
// Get the kdev_t value (device number) from the
// inode.i_rdev field and extract this device's
// minor number:

mov( inode, eax );
linux.minor( (type linux.inode [eax]).i_rdev );

kdebug( linux.printk( "<1>scullc_open, minor#=%d\n", eax ) );
          .
          .
          .
    // Verify that the minor number is within range for
    // this device:

    if( eax >= scullc_devs_c ) then

        kdebug
        (
            linux.printk( "<1>scullc_open:minor # too big\n" )
        );
        mov( errno.enodev, eax );  // No such scull device!
        exit exitOpen;

    endif;
```

Once the open procedure is happy with the minor number, it uses the minor number to compute an index into the array of the scull_devices array. We'll discuss the scull_device object a bit later in this chapter. Suffice to say for now, that a scull_device object is where the *scull* driver keeps the data associated with a particular device. Since the *scullc* driver supports four independent *scullc* devices, the driver has an array of four scull_device values (the scull_devices array). The open procedure obtains the address of one of these array elements (based on the minor number) and stores the address of the specified array element in the filp.private_data field. Having this address in the filp.private_data field makes it easy to access the *scullc* device data in future calls to the device driver. Here's the code that initializes the filp.private_data field:

```
// Compute the address of the scull[minor] device
// structure in the scullc_devices array.  Get the
// address of the particular scullc device into ECX:

intmul( @size( scull_dev ), eax );
lea( ecx, scullc_devices[eax] );

// Initialize the filp->private_data field so that it
// points at the particular device data for the device
// selected by the minor number:

mov( filp, eax );
mov( ecx, (type linux.file [eax]).private_data );
```

The *scullc* device supports persistent data. This means that data written to the device is maintained across opens and closes of the device. At some point, however, the system must be able to initialize (or re-initialize) the device to write brand-new data to it. The scullc device initializes the device whenever some

process opens the device as a write-only device. The next section of code in the open procedure checks the filp.f_flags field to see if the device was opened as a write-only device and, if so, it initializes the device by calling scullc_trim:

```
        // Trim the length of the device down to zero
        // if the open was write-only:

        mov( (type linux.file [eax]).f_flags, edx );
        and( linux.o_accmode, edx );
        if( edx = linux.o_wronly ) then

            kdebug
            (
                linux.printk
                (
                    "<1>scullc_open: opened write-only\n"
                )
            );

            // If someone is already accessing the data structures
            // for this device, make the kernel wait until they
            // are through.

            if( linux.down_interruptible( sdECX.sem ) ) then

                mov( errno.erestartsys, eax );
                exit exitOpen;

            endif;

            // Okay, clear out the existing data in the device:

            scullc_trim( sdECX ); // ignore any errors.

            // Free the semaphore we grabbed above.

            linux.up( sdECX.sem );

        endif;
```

This is one of those few instances where you'll actually use the filp.f_flags field (rather than filp.f_mode field) to see if the file is writable. The catch, of course, is that we're not checking to see if the file is writable, but, rather, we're checking to see how the file was actually opened. The call to scullc_trim in the code above is actually responsible for initializing the device. We'll return to the meaning of the linux.down_interruptible and linux.up procedures a little later in this chapter.

If the open procedure successfully gets past scullc_trim, then it loads EAX with zero (the success return code) and returns to the caller. If there was an error along the way, then the open procedure loads an appropriate error code into EAX and returns without initializing the device (the EXIT statement skip out of the 'begin exitOpen..end exitOpen' block if you're not familiar with the BEGIN..EXIT..END statements in HLA).

The only real "operation" that the open procedure accomplishes is clearing the data buffer associated with the *scullc* device when an application opens the device in a write-only mode. This is necessary because (in the standard design) the *scullc* device only provides a 16KB buffer and will ignore any additional write requests once the buffer fills up. When that happens, some process will need to open the device in a write-only mode to clear out the data.

3.10.2 The release Procedure

The release procedure roughly corresponds to the close call. The kernel calls the release function when the last application using a particular linux.file object closes the file. This, by the way, does not imply that the kernel calls release whenever an application calls the close function. If a process has duplicated a file handle (which is, effectively, a linux.file object)[7], then the kernel will only call release after the application closes all file handles associated with the linux.file object.

The release procedure should do the following:

- Deallocate any storage that open allocated (using kfree to deallocate storage allocated with kmalloc).
- Shut down the device on the last close (if applicable).
- Any other clean-up associated with shutting down the device.

The *scullc* device of this chapter has no hardware to put in a quiescent state nor does it need to deallocate any memory, so the release procedure is very simple; it just prints a message to the log file and the returns a success code (zero) in EAX:

```
procedure scullc_release
(
    var inode   :linux.inode;
    var filp    :linux.file
);  @noframe;
begin scullc_release;

    kdebug( linux.printk( "<1>scullc_release\n" ) );
    xor( eax, eax );
    ret();

end scullc_release;
```

Listing 4.8 The scullc release Procedure

If you look at the release procedure in some other device drivers, you'll notice that they decrement the module use counter. This is no longer necessary (starting with Linux v2.4).

3.10.3 Kernel Memory Management (kmalloc and kfree)

Although the *scullc* driver this chapter presents does not require any dynamic memory allocation and deallocation, this text has mentioned kmalloc and kfree enough times already that it's worthwhile to take some time out from our regularly schedule program to discuss these functions.

The kernel provides the kmalloc and kfree functions as analogs to the malloc and free routines you'll find the in the C (and HLA) Standard Library. The HLA *linux.hhf* header file set places the prototypes for these functions in the linux namespace, so you'll actually refer to these functions as linux.kmalloc and linux.kfree.

The linux.kmalloc function differs from the standard library malloc procedure insofar as it requires a second parameter that specifies the priority of the access. Normally, you will use the constant linux.gfp_kernel as the value of the second linux.kmalloc parameter. See LDD2 and the Linux kernel documentation for the

7. The fork and dup2 system calls can create copies of a file handle that will require multiple closes in order for Linux to call the release function.

meaning of the other possible values you can supply in this parameter position. This document will generally specify only linux.gfp_kernel for the second linux.kmalloc argument.

Note that there is a very limited about of "heap space" in the kernel and attempts to allocate large blocks of kernel memory (especially with the linux.gfp_kernel priority) will fail. Therefore, you should not allocate huge blocks of memory using this function; doing so may dramatically affect the performance of the system.

The linux.kfree function returns storage you allocate via linux.kmalloc back to the kernel heap. You pass the pointer that linux.kmalloc returns as an argument to linux.kfree (just as you would with malloc/free). Unlike linux.kmalloc, the linux.kfree procedure does not require a second parameter.

The *scullc* driver appearing in this chapter could be easily modified to allocate storage for the four scull devices using dynamic memory allocation (linux.kmalloc and linux.kfree). Such a policy would allow whomever opens a file to request the maximum size the file could grow to (rather than fixing the size at 16KB). The original scull driver in LDD2 used a sophisticated memory management scheme (with kmalloc and kfree) that actually allows the memory area for the *scull* device to grow and shrink as necessary. However, as noted at the beginning of this chapter, such sophistication is introducing far more complexity than a simple character driver example probably should have, hence the use of statically allocated 16KB data arrays in the example appearing within this chapter.

3.10.4 The scull_device Data Type

The previous section mentioned that LDD2's scull driver used dynamic allocation to set up the scull data object whereas the scull device of this chapter uses static allocation. Perhaps now is a good time to exactly describe the data structure this chapter's driver uses. Without further ado, here's the definition that appears in the *scullc.hhf* header file:

```
const
    maxScullcSize_c := 16*1024;         // Default 16K size for device.
    scullc_devs_c   := 4;               // # of devices to create.

type
    scull_dev :record
        len     :dword;                     // Size of real data.
        sem     :linux.semaphore;           // Mutual exclusion semaphore.
        data    :byte[ maxScullcSize_c ];   // Scullc data area.
    endrecord;
```

As noted throughout this chapter, the *scullc* driver actually controls four *scullc* devices: *scull0, scull1, scull2*, and *scull3*. In fact, the number of supported devices is very easy to change in this driver by changing the value of the *scullc_devs_c* constant. However, four is a reasonable number to use (being the number of scull devices that LDD2 supports).

The other constant in the *scullc.hhf* header file that affects the scullc device is the *maxScullcSize_c* constant definition. This constant tells HLA how many bytes to reserve for an array that *scullc* uses to hold the data associated with the device. The default is 16KB per *scullc* device, which is probably a reasonable number for demonstration purposes. However, if you want to reserve more or less data for the device, feel free to change the constant's declaration. Note that this value is the per-device amount. Therefore, with the default four devices, the *scullc* driver actually reserves a little bit more than 64K for all four devices[8].

The scull_dev record is the data structure that the scullc driver uses to maintain the device. As you can see, there's not much too this device's data structure, just the len, sem, and data fields. The len field specifies how many data bytes actually appear in the data field (that is, len is an index into the array and specifies the next available position for writing in the data array). The sem field is a semaphore that the *scullc* driver uses to control access to the structure (we'll talk briefly about semaphores in the next section). Finally, the data field is where the readers and writers read and write their data.

8. There are a few additional bytes of data needed for each device in addition to the 16KB buffer.

Whenever an application opens a scullc device as a write-only file, the scullc open procedure (see the code earlier) calls the scullc_trim function which stores writes a zero to the len field of the scull_dev data type. After this point, all further writes to the device will store their data starting at position zero in the data array. The following listing is the scullc_trim device initialization code:

```
// scullc_trim-
//
//  Initializes one of the scullc devices for writing.

procedure scullc_trim( var sd: scull_dev in eax );
begin scullc_trim;

    kdebug( linux.printk( "<1>scullc_trim: entry\n" ) );
    kassert( sd <> 0 );

    // Set the length to zero:

    mov( 0, (type scull_dev [eax]).len );

    // Return success:

    xor( eax, eax );

    kdebug
    (
        linux.printk
        (
            "<1>scullc_trim: exit (returning %d)\n",
            eax
        );
    );

end scullc_trim;
```

Listing 4.9 The scullc_trim Device Initialization Code

By convention, scullc_trim returns a zero in EAX to indicate success. In fact, there is no way for this function to fail, so returning zero in EAX is somewhat redundant. However, by convention functions like this one should return a success/error code in EAX. This code does that to adhere to convention (so you won't have to remember whether it really returns a value or not).

If you prefer, you could easily modify this function so that it also zeros out the data array in the scull_dev data structure. However, since the device driver never allows read/write access to the data beyond the byte indexed by the scull_dev len field, there is really no need to do this.

The only other field in the scull_dev data structure is the sem field. It turns out that the init_module procedure initializes this field, not scullc_trim.

3.10.5 A (Very) Brief Introduction to Race Conditions

Because of the sophisticated nature of the data structure in LDD2's *scull* example, two processes can get into trouble if they attempt to simultaneously access the dynamically allocated data buffer area that holds the *scull* data. For that reason, the code that manipulates the LDD2 dynamic data structure is not reentrant – that

is, only one process at a time may be executing code that manipulates the data structures. The LDD2 *scull* example uses semaphores to ensure that only one process at a time accesses these data structures.

Strictly speaking, the *scullc* example of this chapter does not require semaphores because it doesn't use the sophisticated data structures that the LDD2 example employs to manage the device's data. Nevertheless, the code in this example does use a semaphore to protect access to the array during an initialization operation[9]. This is done more as an example than anything else.

You declare Linux semaphores in HLA using the linux.semaphore data type. Before you can use a semaphore, you must initialize it with the following HLA macro:

```
linux.sema_init( semaphoreObject, resourceCount );
```

For binary semaphores (the kind that we'll normally use, that provide mutual exclusion), the resourceCount argument should be the value '1'. Here's the code in init_module that initializes the four semaphores in the scullc_devices array (this code also calls scullc_trim to initialize the data area associated with each scullc_devices element):

```
// Clear out the scullc_devices array:

    kdebug
    (
        linux.printk
        (
            "<1>Clearing out the scullc_devices array\n"
        );
    );
    mov( 3 * @size( scull_dev ), ecx );  // Select last element.
    repeat

        scullc_trim( scullc_devices[ecx] );

        // Initialize the object's semaphore:

        linux.sema_init( (type scull_dev scullc_devices[ecx]).sem, 1 );

        // Move on to the previous element of the scull array:

        sub( @size( scull_dev ), ecx );

    until( @s ); // ecx < 0
```

Semaphores protect some resource. That resource could be just about anything, though we'll typically use semaphores to control access to some data structure (like a scull_device object) or to some sequence of machine instructions (that is, we'll only allow one process at a time to execute the instruction sequence).

To use a semaphore, you call the Linux system functions linux.down and linux.up. The linux.down function decrements its counter (hence the name down) and the blocks if the semaphore's value is less than zero (the resourceCount value you pass to linux.sema_init above provides the initial value that that linux.down decrements; initializing this with one says that only one process at a time can use the resource the semaphore protects). If the semaphore's count value is one or greater when you call linux.down, then the function immediately returns after decrementing the semaphore's counter. If the counter is one, this tells Linux that the resource is available and the call to linux.down "gives" the resource to whomever calls linux.down. Until that process "gives up" the resource, any further calls to linux.down (with the same semaphore object) will force the new process requesting the semaphore (resource) to wait until the original process is done with the resource.

9. It doesn't protect much. It simply doesn't allow one process to initialize the device while another process is reading or writing the structure (or vice versa). This may seem important, but really it isn't that important since the very next read or write after the device has been reinitialized will fail anyway.

To give up a resource held by a semaphore, a process calls the linux.up procedure. This increments the semaphore counter and releases any process(es) waiting on the semaphore so they can grab the resource.

This is a rather trivial discussion of semaphores. LDD2 provides a little more information (so check it out), but the truth is that semaphores and synchronization in general are somewhat complex and there is all kinds of trouble you can get into. I'd recommend you have a look at a good operating systems text to get more comfortable with the concept of semaphores, synchronization, and the problems you'll run into when using semaphores (like deadlock).

Before moving on, I would be remiss not to mention that you'll rarely call the linux.down procedure. Instead of linux.down, you'll usually call linux.down_interruptible (as *scullc* does). The linux.down_interruptible returns under one of two conditions: when some other process releases the semaphore or if the system sends a signal to the waiting process. The linux.down_interruptible function returns zero if it obtains the semaphore. It returns a non-zero code if it was awoken because of a signal rather than having obtained the semaphore. Therefore, you'll want to test the return value (in EAX) from linux.down_interruptible before assuming you've obtained the semaphore.

3.10.6 The read and write Procedures

Since the read and write procedures in the scullc module are nearly identical, it makes sense to discuss these two procedures together in the same section. Here are their prototypes:

```
procedure scullc_read
(
    var filp    :linux.file;
    var buf     :var;
        count   :linux.size_t;
    var f_pos   :linux.loff_t
);  @use eax;
    @cdecl;
    @external;

procedure scullc_write
(
    var filp    :linux.file;
    var buf     :var;
        count   :linux.size_t;
    var f_pos   :linux.loff_t
);  @use eax;
    @cdecl;
    @external;
```

The filp parameter is a pointer to the linux.file structure for the object. The count parameter is the number of bytes to read or write (depending on the procedure). The f_pos parameter is a pointer to the 64-bit file position pointer for the file; usually (but not always) this is the f_pos field of the filp structure; you should, however, never count on this because sometimes f_pos can point at a different variable. The buf parameter points at a buffer in user space. For the scullc_read operation, buf points at a buffer where the driver will store data; it must be sufficiently large to hold at least count bytes. For the scullc_write procedure, buf points at a buffer where data will be taken from to write to the device; there must be at least count bytes of (valid) data in this buffer.

Note that buf is a pointer into user-space and you cannot directly dereference this pointer in your driver. In particular, you cannot use memcpy or a similar function to transfer data between the user's buffer and the *scullc* device. There are several reasons for this, not the least of which that applications on the x86 use a completely different address space than the kernel. This means that the data appearing at the kernel memory address held in buf is not at all the data in the user's buffer.

Another big difference between kernel-space addresses and user-space adddresses is that user-space addresses can be swapped out to disk. Dereferencing a user-space address (even once you map it into kernel

space) could generate a page fault, which is something you want to let the kernel, proper, deal with rather than having to deal with this in your device driver.

Linux provides special functions to handle cross-space copies (user/kernel space). This text will explain most of these functions in a later function of this text. For now, however, we'll get by with two generic functions: linux.copy_to_user and linux.copy_from_user. Here are their prototypes:[10]

```
procedure linux.copy_to_user( var _to:var; var from:var; count:dword );
    @use eax;
    @cdecl;
    @external;

procedure linux.copy_from_user( var _to:var; var from:var; count:dword );
    @use eax;
    @cdecl;
    @external;
```

You use these functions just like memcpy with a few caveats. First, since the user-space buffer could be currently swapped out to disk, it's quite possible for these functions to go to sleep pending the kernel swapping the data back to memory. While the process is sleeping, waiting for the data to become valid, another process could write data to the *scullc* device and that process could reenter your driver. This is one of the reasons you must write reentrant code in your device driver.

These functions do more than simply copy data between user and kernel space, they also validate the user pointer to ensure it's valid. These functions returns number number of bytes still left to copy in the EAX register. If this is non-zero, then the scullc_read and scullc_write procedures return an errno.efault error code.

It is scullc_read's responsibility to copy data from the *scullc* device to user space; it must copy count bytes starting at the file position specified by the 64-bit value at the address the f_pos parameter contains. However, scullc_read will fail if the file position is greater than the number of bytes current written to the scull_dev object (that is, if the file position is greater than the value held in the len field of the scull_dev object for the current scullc device). The scullc_read procedure handles this with the following code (note that ECX points at the scull_dev object and sdECX is equivalent to "(type scull_dev [ecx])":

```
// 64-bit comparison of f_pos with filp->size.
// Exit if the file position is greater than the size.
// Although the size of the device will never exceed
// 32-bits, some joker could llseek to some value beyond
// 2^32, so we have to do a 64-bit comparison.  However,
// since we know the device will never exceed 2^32 bytes
// in size, we only need to compare the H.O. dword against
// zero.

mov( f_pos, edx );

if
(#{
    cmp( (type dword [edx+4]), 0 );
    jne true;
    mov( [edx], ebx );
    cmp( ebx, (type dword sdECX.len) );
    jb false;
}#) then

    kdebug
    (
        linux.printk
```

10. Actually, these are macros so what you're seeing aren't the true prototypes, but the functions they ultimately call have very similar prototypes.

```
            (
                "<1>scullc_read: Attempt to read beyond device\n"
            )
        );

        exit nothing; // Exits scullc_read

    endif;
```

If the file pointer is less than the scull_dev's len value, we're still not out of the woods. It could turn out that the count value plus the starting file position would take us beyond the end of the data written to the *scullc* device. Therefore, the scullc_read procedure checks for this condition and reduces the value of count, if necessary, using the following code:

```
    // Okay, we know that the current file position is
    // within the size of the scullc device, so we can
    // stick to 32-bit arithmetic here (since the scullc
    // device is never greater than 4GB!).
    //
    // Okay, check to see if the current read operation
    // would take us beyond the end of the file, if so,
    // then decrement the count so that we read the rest
    // of the device and no more.
    //
    // Note that if we get down here, EBX will contain
    // the L.O. dword of the file position.

    add( count, ebx );  // Compute the end of data to read.
    if( ebx > (type dword sdECX.len)) then

        // Okay, we'd read beyond the end of the device.
        // Truncate this read operation.

        mov( (type dword sdECX.len), ebx );
        sub( [edx], ebx );
        mov( ebx, count );

    endif;
```

Once we get past the code above, there's little left to do except actually transfer the data. Here's the code that does this:

```
    mov( [edx], ebx );  // Get the current file position.
    lea( edx, sdECX.data[ebx] );
    linux.copy_to_user( buf, [edx], count );
```

Whereas the linux.copy_to_user function returns the number of bytes left to transfer, scullc_read's responsibility is to return the number of bytes it actually transferred. The scullc_read procedure must also update the file position (pointed at by the f_pos parameter) so that it points at the next byte to read from the device. The scullc_read procedure uses the following code to accomplish this:

```
        mov( count, eax );   // Return # of bytes actually read
        mov( eax, rtnVal );
        mov( f_pos, edx );   // Update the file position.
        add( eax, [edx] );
```

Beyond the code snippets given above, the principle thing that scullc_read does is some error checking and ensuring that no other process holds the device's semaphore before scullc_read attempts to transfer data from the device. Here's the full code to the scullc_read procedure:

```
// scullc_read-
//
//  Reads data from the 'character device' and copies
// the data to user space.  Returns number of bytes
// actually read in EAX.

procedure scullc_read
(
    var filp    :linux.file;
    var buf     :var;
        count   :linux.size_t;
    var f_pos   :linux.loff_t
);
const
    sdECX   :text := "(type scull_dev [ecx])";
var
    rtnVal  :dword;

begin scullc_read;

    reg_save( ebx, ecx, edx );
    kdebug
    (
        mov( f_pos, ebx );
        linux.printk
        (
            "<1>scullc_read: entry, count=%d, "
            "f_pos=%d:%d, buf=%x\n",
            count,
            (type dword [ebx+4]),
            (type dword [ebx]),
            buf
        )
    );

    // Get a pointer to the current scull_dev object into ECX:

    mov( filp, eax );
    kassert( eax <> 0 );
    mov( (type linux.file [eax]).private_data, ecx );
    kassert( ecx <> 0 );

    // See if it's cool to access the scull_dev object:

    if( linux.down_interruptible( sdECX.sem )) then

        // down_interruptible returns 0 on success, a negative
        // error code on failure (which leads us to this point).
        // Failure means the scullc_dev object is currently being
        // used so we have to wait for it to be free.

        kdebug
        (
            linux.printk( "<1>scullc_read: device in use\n" )
        );
        reg_restore;
        mov( errno.erestartsys, eax );
        exit scullc_read;
```

```
endif;

begin nothing;

    // If the count was zero, immediately return.

    mov( 0, rtnVal );           // Return zero as byte count.
    exitif( count = 0 ) nothing;


    // 64-bit comparison of f_pos with filp->size.
    // Exit if the file position is greater than the size.
    // Although the size of the device will never exceed
    // 32-bits, some joker could llseek to some value beyond
    // 2^32, so we have to do a 64-bit comparison.  However,
    // since we know the device will never exceed 2^32 bytes
    // in size, we only need to compare the H.O. dword against
    // zero.

    mov( f_pos, edx );

    if
    (#{
        cmp( (type dword [edx+4]), 0 );
        jne true;
        mov( [edx], ebx );
        cmp( ebx, (type dword sdECX.len) );
        jb false;
    }#) then

        kdebug
        (
            linux.printk
            (
                "<1>scullc_read: Attempt to read beyond device\n"
            )
        );

        exit nothing;

    endif;

    // Okay, we know that the current file position is
    // within the size of the scullc device, so we can
    // stick to 32-bit arithmetic here (since the scullc
    // device is never greater than 4GB!).
    //
    // Okay, check to see if the current read operation
    // would take us beyond the end of the file, if so,
    // then decrement the count so that we read the rest
    // of the device and no more.
    //
    // Note that if we get down here, EBX will contain
    // the L.O. dword of the file position.

    add( count, ebx );  // Compute the end of data to read.
    if( ebx > (type dword sdECX.len)) then

        // Okay, we'd read beyond the end of the device.
        // Truncate this read operation.
```

```
            mov( (type dword sdECX.len), ebx );
            sub( [edx], ebx );
            mov( ebx, count );

        endif;

        // Okay, copy the data from the current file
        // position of the specified length (count) to the
        // user's buffer:

        kdebug
        (
            linux.printk
            (
                "<1>scull_read: copying %d bytes at posn %d\n",
                count,
                (type dword [edx])
            )
        );

        mov( [edx], ebx );  // Get the current file position.
        lea( edx, sdECX.data[ebx] );
        linux.copy_to_user( buf, [edx], count );

        kdebug
        (
            linux.printk
            (
                "<1>scull_read: %d bytes left to copy\n",
                eax
            )
        );

        // If copy_to_user returns non-zero,  we've got an error
        // (return value is the number of bytes left to copy).

        if( eax = 0 ) then

            mov( count, eax );
            mov( eax, rtnVal );
            mov( f_pos, edx );
            add( eax, [edx] );

            // Technically, the file position is a 64-bit value
            // and we should add in the carry from the add above
            // into the H.O. dword.  However, the device never
            // has more than 2^32 bytes, so we know the carry
            // will always be clear.

        else

            // If the transfer count was zero, then
            // return an efault error.

            mov( errno.efault, rtnVal );

        endif;

    end nothing;

    // Release the semaphore lock we've got on this scull device:
```

```
    linux.up( sdECX.sem );

    // Return the number of bytes actually read (or the error code)
    // to the calling process:

    mov( rtnVal, eax );
    kdebug
    (
        linux.printk( "<1>scullc_read: exit (returns %d)\n", eax )
    );

    reg_restore;

end scullc_read;
```

Listing 4.10 The scullc_read Procedure

The value that scullc_read returns in EAX is the number of bytes that it actually transferred to the user's buffer. If this value is equal to the original count value, then the read was successful.

If the return value is positive, but less than count's original value, then only part of the data was transferred. This is not an error and it is the application's responsibility to issue the read again (though a few ill-behaved applications may treat this as an error).

If the return value is zero, then the device has reached the end of file.

If scull_read returns a negative value, then it's an error code and the return value is one of the errno.XXXX values.

It's also possible for scullc_read to return an indication that "there is some data, but it will arrive later." We'll discuss this possibility in a later chapter when we discuss blocking I/O.

Note that it's quite possible for the file position to be beyond the end of the device's data in the data buffer. This can occur, for example, if the user calls the llseek function and advances the file pointer beyond the end of the data. It can also occur if a process is reading data from the device file and another process opens the file as a write-only device, thus resetting the scull_dev len field to zero. Should this occur, the scullc_read procedure will simply return zero indicating the end of file.

The scullc_write procedure is nearly line-for-line identical to the scullc_read code. There are only a couple of important differences. First, it copies data from the user-space buffer to the device (rather than vice-versa). Second, it extends the value of the scull_dev len field as well as the file position. Finally, another important difference is that it rejects writes that go beyond the end of the data buffer (rather than writes that go beyond len). Without further ado, here's the code for the scullc_write procedure:

```
// scullc_write-
//
//  Writes data to the device by copying the data
// from a user buffer to the device data buffer.
// Returns the number of bytes actually written is
// returned in EAX.

procedure scullc_write
(
    var filp    :linux.file;
    var buf     :var;
        count   :linux.size_t;
    var f_pos   :linux.loff_t
);
```

```
const
    sdECX   :text := "(type scull_dev [ecx])";

var
    rtnVal  :dword;

begin scullc_write;

    reg_save( ebx, ecx, edx );
    kdebug
    (
        mov( f_pos, ebx );
        linux.printk
        (
            "<1>scullc_write: entry, count=%d, "
            "f_pos=%d:%d, buf=%x\n",
            count,
            (type dword [ebx+4]),
            (type dword [ebx]),
            buf
        )
    );

    // Get a pointer to the current scull_dev object into ECX:

    mov( filp, eax );
    kassert( eax <> 0 );
    mov( (type linux.file [eax]).private_data, ecx );
    kassert( ecx <> 0 );


    // See if it's cool to access the scull_dev object:

    if( linux.down_interruptible( sdECX.sem )) then

        // down_interruptible returns 0 on success, a negative
        // error code on failure (which leads us to this point).
        // Failure means the scullc_dev object is currently being
        // used so we have to wait for it to be free.

        kdebug
        (
            linux.printk( "<1>scullc_read: device in use\n" )
        );
        reg_restore;
        mov( errno.erestartsys, eax );
        exit scullc_write;

    endif;

    begin nothing;


        // If the count was zero, immediately return.

        mov( 0, rtnVal );
        exitif( count = 0 ) nothing;


        // 64-bit comparison of f_pos with filp->size.
        // Exit if the file position is greater than the size.
```

```
// Although the size of the device will never exceed
// 32-bits, some joker could llseek to some value beyond
// 2^32, so we have to do a 64-bit comparison.  However,
// since we know the device will never exceed 2^32 bytes
// in size, we only need to compare the H.O. dword against
// zero.

mov( f_pos, edx );

if
(#{
    cmp( (type dword [edx+4]), 0 );
    jne true;
    cmp( (type dword [edx]), maxScullcSize_c );
    jb false;
}#) then

    kdebug
    (
        linux.printk
        (
            "<1>scullc_write: "
            "Attempt to write beyond device, pos=%d:%d\n",
            (type dword [edx+4]),
            (type dword [edx])
        )
    );
    mov( errno.enospc, rtnVal );
    exit nothing;

endif;

// Okay, we know that the current file position is
// within the size of the scullc device, so we can
// stick to 32-bit arithmetic here (since the scullc
// device is never greater than 4GB!).
//
// Okay, check to see if the current write operation
// would take us beyond the end of the file, if so,
// then decrement the count so that we write to the end
// of the device and no more.

mov( [edx], ebx );
add( count, ebx );  // Compute the end of data to read.
if( ebx > maxScullcSize_c ) then

    // Okay, we'd write beyond the end of the device.
    // Truncate this write operation and bail if it
    //  turns out we'd transfer zero bytes.

    mov( maxScullcSize_c, ebx );
    sub( [edx], ebx );
    mov( ebx, count );
    exitif( ebx = 0 ) nothing;

endif;

// Okay, copy the data from the current file
// position of the specified length (count) to the
// user's buffer:
```

```
            kdebug
            (
                linux.printk
                (
                    "<1>scull_write: copying %d bytes to posn %d\n",
                    count,
                    (type dword [edx])
                )
            );

            mov( [edx], ebx );  // Get the current file position.
            lea( edx, sdECX.data[ebx] );

            linux.copy_from_user( [edx], buf, count );

            kdebug
            (
                linux.printk( "<1>scull_write: %d bytes left to copy\n", eax )
            );


            // If copy_to_user returns non-zero,  we've got an error
            // (the return value is the number of bytes left to copy).

            if( eax = 0 ) then

                mov( count, eax );
                mov( eax, rtnVal );
                mov( f_pos, edx );
                add( eax, [edx] );
                add( eax, sdECX.len );

                // Technically, the file position is a 64-bit value
                // and we should add in the carry from the add above
                // into the H.O. dword.  However, the device never
                // has more than 2^32 bytes, so we know the carry
                // will always be clear.

            else

                // If the transfer count was zero, then
                // return an efault error.

                mov( errno.efault, rtnVal );

            endif;

        end nothing;

        linux.up( sdECX.sem );
        mov( rtnVal, eax );
        kdebug
        (
            linux.printk( "<1>scullc_write: exit (returns %d)\n", eax )
        );

        reg_restore;

    end scullc_write;
```

Listing 4.11 The scullc_write Procedure

The scullc_write function returns the number of bytes actually written. You should interpret the return result as follows:

- If the value is equal to count, then scullc_write has successfully transferred the desired number of bytes.
- If the value is positive but less than count, then scullc_write has written fewer than the specified number of bytes. The application program should retry writing the remaining bytes that were not transferred.
- If the value is zero this does not indicate an error, but only that the application should retry the write later. Blocking I/O (which we'll cover in a later chapter) returns this value as well.
- A negative value is a typical errno.XXXX error code and the application should deal with the error accordingly.

As it turns out, the scullc_write function returns an error if it cannot transfer all the data for one reason or another. So the discussion above involving non-negative values less than count apply to device driver write procedures in general, but not specifically to scullc_write.

A careful analysis of the write procedure will suggest a minor security flaw. As noted earlier, the scullc_trim procedure doesn't zero out the scull_dev data array when a process opens the device as write-only. I mentioned earlier that this isn't a problem because the *scullc* device doesn't read beyond the len barrier. Strictly speaking, this statement is true. However, as you'll soon see, a process could open the scullc device as a write-only device and then do a scullc_llseek operation (explained next) to advance the file position. Then a write of one byte to the *scullc* device will advance the len value beyond all the data previously written to the device (by other processes). This allows the new process to peek at the data written by some other process (a minor security gaffe).

The current *scullc* driver ignores this issue because the *scullc* device is global and persistent. In particular, if the current process wants to read the previous data written by the last process to access the device, it could do so simply by opening the device in read-only mode rather than write-only mode. If the previous process really wanted to prevent any new processes from reading the data once it was done, it could overwrite the device file with zeros. Since the whole purpose of the *scullc* device is to share data, providing security to the data is counter-productive (*scull* devices we'll develop in later chapters will demonstrate how to implement security policies in device drivers).

In any case, I just wanted to point out this issue so you're aware of it in the event you use this code as a template for your own device drivers.

3.11 The scullc Driver

Here's the full source code for the scullc driver (note that the getversion.hhf header file was presented in the previous chapter and does not reappear here).

3.11.1 The scullc.hhf Header File

```
#if( !@defined( scullc_hhf) )
?scullc_hhf := true;

const
    maxScullcSize_c := 16*1024;         // Default 16K size for device.
    scullc_devs_c   := 4;               // # of devices to create.

type
    scull_dev :record
```

```
        len     :dword;                     // Size of real data.
        vmas    :dword;                     // Counts active mappings.
        sem     :linux.semaphore;           // Mutual exclusion semaphore.
        data    :byte[ maxScullcSize_c ];   // Scullc data area.
    endrecord;


static
    scullci :dword; external;

    procedure scullc_trim( var sd: scull_dev in eax ); forward;

    procedure scullc_open( var inode: linux.inode; var filp:linux.file );
        @use eax;
        @cdecl;
        @external;


    procedure scullc_release( var inode: linux.inode; var filp:linux.file );
        @use eax;
        @cdecl;
        @external;

    procedure scullc_read
    (
        var filp    :linux.file;
        var buf     :var;
            count   :linux.size_t;
        var f_pos   :linux.loff_t
    );  @use eax;
        @cdecl;
        @external;

    procedure scullc_write
    (
        var filp    :linux.file;
        var buf     :var;
            count   :linux.size_t;
        var f_pos   :linux.loff_t
    );  @use eax;
        @cdecl;
        @external;

    procedure scullc_llseek
    (
        var filp    :linux.file;
            off     :linux.loff_t;
            whence  :dword
    );
        @use eax;
        @cdecl;
        @external;


    procedure init_module; @external;
    procedure cleanup_module; @external;

#endif
```

3.12 The scullc.hla Source File

```
/*
 * main.c -- the bare scullc char module
 *
 * Copyright (C) 2001 Alessandro Rubini and Jonathan Corbet
 * Copyright (C) 2001 O'Reilly & Associates
 * Copyright (C) 2002 Randall Hyde
 *
 * The source code in this file can be freely used, adapted,
 * and redistributed in source or binary form, so long as an
 * acknowledgment appears in derived source files.  The citation
 * should list that the code comes from the book "Linux Device
 * Drivers" by Alessandro Rubini and Jonathan Corbet, published
 * by O'Reilly & Associates.   No warranty is attached;
 * we cannot take responsibility for errors or fitness for use.
 */


#include( "getversion.hhf" )

unit scullc;

// "linux.hhf" contains all the important kernel symbols.
// Must define __kernel__ prior to including this value to
// gain access to kernel symbols.  Must define __smp__ before
// the include if compiling for an SMP machine.

// ?__smp__ := true; //Uncomment for SMP machines.
?__kernel__ := true;

#includeonce( "linux.hhf" )


// Enable debug code in this module.
// Note that kernel.hhf (included by linux.hhf) sets this
// val object to true, so we must set it to false *after*
// including linux.hhf.
//
// Be sure to set this to true for production code!
// (Setting KNDEBUG true automatically eliminates all
// the debug code in this file, assuming there are no
// other assignments to KNDEBUG in this source file).

?KNDEBUG := false;


// Skull-specific declarations:

#includeonce( "scullc.hhf" )

// Set up some global HLA procedure options:

?@nodisplay := true;
?@noalignstack := true;
?@align := 4;



static
```

```
    scullci         :dword;         // This module's reference counter.
    scullc_major    :dword := 0;    // Default: use dynamic major #.

    // scullc_fops- here's the "file_operations" functions that
    // this driver supports (the non-NULL entries):

    scullc_fops     :linux.file_operations :=
                        linux.fileops_c
                        (
                            llseek  :&scullc_llseek,
                            read    :&scullc_read,
                            write   :&scullc_write,
                            open    :&scullc_open,
                            release :&scullc_release
                        );


    // Storage for the devices.

    scullc_devices  :scull_dev[ scullc_devs_c ];


val
    lastRegSet  :dword;

// reg_save - pushes a set of 32-bit registers
// onto the stack and saves the register set so that
// reg_restore can automatically pop them.

#macro reg_save( regs[] ):rindex, rmax, reg;
    ?rmax   := @elements( regs );
    #if( rmax > 0 )
        ?lastRegSet :string[ rmax ];
    #endif
    ?rindex := 0;
    #while( rindex < rmax )

        ?reg :text := regs[ rindex ];
        #if( @isreg( reg ))

            #if( @size( reg ) = 4 )

                push( reg );

            #else

                #error( "Expected a 32-bit register" )

            #endif

        #else

            #error
            (
                "Expected a 32-bit register, encountered: '" +
                regs[ rindex ] +
                "'"
            );

        #endif
```

```
            ?lastRegSet[ rindex ] := regs[ rindex ];
            ?rindex := rindex + 1;

            // We have to change reg to string before next loop
            // iteration, or the assignment to reg above will choke.

            ?@tostring:reg :string;

        #endwhile

    #endmacro;

    // reg_restore - pops the registers last pushed by
    // the reg_save macro.  This macro doesn't do any
    // error reporting because that was done by reg_save.

    #macro reg_restore:rindex, reg;
        ?rindex :int32  := @elements( lastRegSet ) - 1;
        #while( rindex >= 0 )

            ?reg :text := lastRegSet[ rindex ];
            #if( @isreg( reg ))

                #if( @size( reg ) = 4 )

                    pop( reg );

                #endif

            #endif
            ?rindex := rindex - 1;
            ?@tostring:reg :string;

        #endwhile

    #endmacro;



    // scullc_open-
    //
    //  Handles the open call to this device.

    procedure scullc_open
    (
        var inode   :linux.inode;
        var filp    :linux.file
    );
    const
        sdECX       :text := "(type scull_dev [ecx])";

    begin scullc_open;

        reg_save( ecx, edx );

        // Get the kdev_t value (device number) from the
        // inode.i_rdev field and extract this device's
        // minor number:

        mov( inode, eax );
        linux.minor( (type linux.inode [eax]).i_rdev );
```

```
kdebug( linux.printk( "<1>scullc_open, minor#=%d\n", eax ) );

begin exitOpen;

    // Verify that the minor number is within range for
    // this device:

    if( eax >= scullc_devs_c ) then

        kdebug
        (
            linux.printk( "<1>scullc_open:minor # too big\n" )
        );
        mov( errno.enodev, eax );  // No such scull device!
        exit exitOpen;

    endif;

    // Compute the address of the scull[minor] device
    // structure in the scullc_devices array.  Get the
    // address of the particular scullc device into ECX:

    intmul( @size( scull_dev ), eax );
    lea( ecx, scullc_devices[eax] );

    // Initialize the filp->private_data field so that it
    // points at the particular device data for the device
    // selected by the minor number:

    mov( filp, eax );
    mov( ecx, (type linux.file [eax]).private_data );

    // Trim the length of the device down to zero
    // if the open was write-only:

    mov( (type linux.file [eax]).f_flags, edx );
    and( linux.o_accmode, edx );
    if( edx = linux.o_wronly ) then

        kdebug
        (
            linux.printk
            (
                "<1>scullc_open: opened write-only\n"
            )
        );

        // If someone is already accessing the data structures
        // for this device, make the kernel wait until they
        // are through.

        if( linux.down_interruptible( sdECX.sem ) ) then

            mov( errno.erestartsys, eax );
            exit exitOpen;

        endif;

        // Okay, clear out the existing data in the device:
```

```
                scullc_trim( sdECX ); // ignore any errors.

                // Free the semaphore we grabbed above.

                linux.up( sdECX.sem );

            endif;

            xor( eax, eax );    // return success

        end exitOpen;

        kdebug
        (
            linux.printk( "<1>scullc_open: exit (returns %d)\n", eax )
        );
        reg_restore;

    end scullc_open;



    // scullc_release-
    //
    //  Not much to do here, but we do need to decrement the
    // module counter that was incremented in scullc_open.

    procedure scullc_release
    (
        var inode   :linux.inode;
        var filp    :linux.file
    );  @noframe;
    begin scullc_release;

        //linux.mod_dec_use_count;

        kdebug( linux.printk( "<1>scullc_release\n" ) );
        xor( eax, eax );
        ret();

    end scullc_release;




    // scullc_read-
    //
    //  Reads data from the 'character device' and copies
    // the data to user space.

    procedure scullc_read
    (
        var filp    :linux.file;
        var buf     :var;
            count   :linux.size_t;
        var f_pos   :linux.loff_t
    );
    const
        sdECX   :text := "(type scull_dev [ecx])";
    var
        rtnVal  :dword;
```

```
begin scullc_read;

    reg_save( ebx, ecx, edx );
    kdebug
    (
        mov( f_pos, ebx );
        linux.printk
        (
            "<1>scullc_read: entry, count=%d, "
            "f_pos=%d:%d, buf=%x\n",
            count,
            (type dword [ebx+4]),
            (type dword [ebx]),
            buf
        )
    );

    // Get a pointer to the current scull_dev object into ECX:

    mov( filp, eax );
    kassert( eax <> 0 );
    mov( (type linux.file [eax]).private_data, ecx );
    kassert( ecx <> 0 );

    // See if it's cool to access the scull_dev object:

    if( linux.down_interruptible( sdECX.sem )) then

        // down_interruptible returns 0 on success, a negative
        // error code on failure (which leads us to this point).
        // Failure means the scullc_dev object is currently being
        // used so we have to wait for it to be free.

        kdebug
        (
            linux.printk( "<1>scullc_read: device in use\n" )
        );
        reg_restore;
        mov( errno.erestartsys, eax );
        exit scullc_read;

    endif;

    begin nothing;

        // If the count was zero, immediately return.

        mov( 0, rtnVal );           // Return zero as byte count.
        exitif( count = 0 ) nothing;


        // 64-bit comparison of f_pos with filp->size.
        // Exit if the file position is greater than the size.
        // Although the size of the device will never exceed
        // 32-bits, some joker could llseek to some value beyond
        // 2^32, so we have to do a 64-bit comparison.  However,
        // since we know the device will never exceed 2^32 bytes
        // in size, we only need to compare the H.O. dword against
        // zero.
```

```
mov( f_pos, edx );

if
(#{
    cmp( (type dword [edx+4]), 0 );
    jne true;
    mov( [edx], ebx );
    cmp( ebx, (type dword sdECX.len) );
    jb false;
}#) then

    kdebug
    (
        linux.printk
        (
            "<1>scullc_read: Attempt to read beyond device\n"
        )
    );

    exit nothing;

endif;

// Okay, we know that the current file position is
// within the size of the scullc device, so we can
// stick to 32-bit arithmetic here (since the scullc
// device is never greater than 4GB!).
//
// Okay, check to see if the current read operation
// would take us beyond the end of the file, if so,
// then decrement the count so that we read the rest
// of the device and no more.
//
// Note that if we get down here, EBX will contain
// the L.O. dword of the file position.

add( count, ebx );  // Compute the end of data to read.
if( ebx > (type dword sdECX.len)) then

    // Okay, we'd read beyond the end of the device.
    // Truncate this read operation.

    mov( (type dword sdECX.len), ebx );
    sub( [edx], ebx );
    mov( ebx, count );

endif;

// Okay, copy the data from the current file
// position of the specified length (count) to the
// user's buffer:

kdebug
(
    linux.printk
    (
        "<1>scull_read: copying %d bytes at posn %d\n",
        count,
        (type dword [edx])
    )
);
```

```
            mov( [edx], ebx );  // Get the current file position.
            lea( edx, sdECX.data[ebx] );
            linux.copy_to_user( buf, [edx], count );

            kdebug
            (
                linux.printk
                (
                    "<1>scull_read: %d bytes left to copy\n",
                    eax
                )
            );

            // If copy_to_user returns non-zero,  we've got an error
            // (return value is the number of bytes left to copy).

            if( eax = 0 ) then

                mov( count, eax );
                mov( eax, rtnVal );
                mov( f_pos, edx );
                add( eax, [edx] );

                // Technically, the file position is a 64-bit value
                // and we should add in the carry from the add above
                // into the H.O. dword.  However, the device never
                // has more than 2^32 bytes, so we know the carry
                // will always be clear.

            else

                // If the transfer count was zero, then
                // return an efault error.

                mov( errno.efault, rtnVal );

            endif;

        end nothing;

        // Release the semaphore lock we've got on this scull device:

        linux.up( sdECX.sem );

        // Return the number of bytes read (or the error code)
        // to the calling process:

        mov( rtnVal, eax );
        kdebug
        (
            linux.printk( "<1>scullc_read: exit (returns %d)\n", eax )
        );

        reg_restore;

    end scullc_read;
```

```
// scullc_write-
//
//  Writes data to the device by copying the data
// from a user buffer to the device data buffer.

procedure scullc_write
(
    var filp    :linux.file;
    var buf     :var;
        count   :linux.size_t;
    var f_pos   :linux.loff_t
);
const
    sdECX   :text := "(type scull_dev [ecx])";

var
    rtnVal  :dword;

begin scullc_write;

    reg_save( ebx, ecx, edx );
    kdebug
    (
        mov( f_pos, ebx );
        linux.printk
        (
            "<1>scullc_write: entry, count=%d, "
            "f_pos=%d:%d, buf=%x\n",
            count,
            (type dword [ebx+4]),
            (type dword [ebx]),
            buf
        )
    );

    // Get a pointer to the current scull_dev object into ECX:

    mov( filp, eax );
    kassert( eax <> 0 );
    mov( (type linux.file [eax]).private_data, ecx );
    kassert( ecx <> 0 );


    // See if it's cool to access the scull_dev object:

    if( linux.down_interruptible( sdECX.sem )) then

        // down_interruptible returns 0 on success, a negative
        // error code on failure (which leads us to this point).
        // Failure means the scullc_dev object is currently being
        // used so we have to wait for it to be free.

        kdebug
        (
            linux.printk( "<1>scullc_read: device in use\n" )
        );
        reg_restore;
```

```
    mov( errno.erestartsys, eax );
    exit scullc_write;

endif;

begin nothing;


    // If the count was zero, immediately return.

    mov( 0, rtnVal );
    exitif( count = 0 ) nothing;


    // 64-bit comparison of f_pos with filp->size.
    // Exit if the file position is greater than the size.
    // Although the size of the device will never exceed
    // 32-bits, some joker could llseek to some value beyond
    // 2^32, so we have to do a 64-bit comparison.  However,
    // since we know the device will never exceed 2^32 bytes
    // in size, we only need to compare the H.O. dword against
    // zero.

    mov( f_pos, edx );

    if
    (#{
        cmp( (type dword [edx+4]), 0 );
        jne true;
        cmp( (type dword [edx]), maxScullcSize_c );
        jb false;
    }#) then

        kdebug
        (
            linux.printk
            (
                "<1>scullc_write: "
                "Attempt to write beyond device, pos=%d:%d\n",
                (type dword [edx+4]),
                (type dword [edx])
            )
        );
        mov( errno.enospc, rtnVal );
        exit nothing;

    endif;

    // Okay, we know that the current file position is
    // within the size of the scullc device, so we can
    // stick to 32-bit arithmetic here (since the scullc
    // device is never greater than 4GB!).
    //
    // Okay, check to see if the current write operation
    // would take us beyond the end of the file, if so,
    // then decrement the count so that we write to the end
    // of the device and no more.

    mov( [edx], ebx );
    add( count, ebx );  // Compute the end of data to read.
    if( ebx > maxScullcSize_c ) then
```

```
        // Okay, we'd write beyond the end of the device.
        // Truncate this write operation and bail if it
        //  turns out we'd transfer zero bytes.

        mov( maxScullcSize_c, ebx );
        sub( [edx], ebx );
        mov( ebx, count );
        exitif( ebx = 0 ) nothing;

    endif;

    // Okay, copy the data from the current file
    // position of the specified length (count) to the
    // user's buffer:

    kdebug
    (
        linux.printk
        (
            "<1>scull_write: copying %d bytes to posn %d\n",
            count,
            (type dword [edx])
        )
    );

    mov( [edx], ebx );  // Get the current file position.
    lea( edx, sdECX.data[ebx] );

    linux.copy_from_user( [edx], buf, count );

    kdebug
    (
        linux.printk
        (
            "<1>scull_write: %d bytes left to copy\n",
            eax
        )
    );


    // If copy_to_user returns non-zero,  we've got an error
    // (the return value is the number of bytes left to copy).

    if( eax = 0 ) then

        mov( count, eax );
        mov( eax, rtnVal );
        mov( f_pos, edx );
        add( eax, [edx] );
        add( eax, sdECX.len );

        // Technically, the file position is a 64-bit value
        // and we should add in the carry from the add above
        // into the H.O. dword.  However, the device never
        // has more than 2^32 bytes, so we know the carry
        // will always be clear.

    else

        // If the transfer count was zero, then
```

```
            // return an efault error.

            mov( errno.efault, rtnVal );

        endif;

    end nothing;

    linux.up( sdECX.sem );
    mov( rtnVal, eax );
    kdebug
    (
        linux.printk( "<1>scullc_write: exit (returns %d)\n", eax )
    );

    reg_restore;

end scullc_write;


// scullc_llseek-
//
//  Seeks to a new file position.
// Returns new 64-bit file position in EDX:EAX.

procedure scullc_llseek
(
    var filp    :linux.file;
        off     :linux.loff_t;
        whence  :dword
);
const
    sdECX   :text := "(type scull_dev [ecx])";

begin scullc_llseek;

    reg_save( ecx );
    mov( filp, ecx );
    mov( whence, eax );
    if( eax = linux.set_seek ) then

        mov( (type dword off), eax );
        mov( (type dword off[4]), edx );

    elseif( eax = linux.seek_cur ) then

        mov( (type dword off), eax );
        mov( (type dword off[4]), edx );
        add( (type dword (type linux.file [ecx]).f_pos), eax );
        adc( (type dword (type linux.file [ecx]).f_pos[4]), edx );

    elseif( eax = linux.seek_end ) then

        mov( (type linux.file [ecx]).private_data, edx );
        mov( (type scull_dev [edx]).len, eax );
        xor( edx, edx ); // Device size is 32-bits or less.
        add( (type dword (type linux.file [ecx]).f_pos), eax );
        adc( (type dword (type linux.file [ecx]).f_pos[4]), edx );

    else
```

```
            mov( errno.einval, eax );
            mov( -1, edx );

        endif;

        if( (type int32 edx) >= 0 ) then

            mov( eax, (type dword (type linux.file [ecx]).f_pos) );
            mov( edx, (type dword (type linux.file [ecx]).f_pos[4]) );

        endif;
        reg_restore;

    end scullc_llseek;



    // scullc_trim-
    //
    //  Initializes one of the scullc devices for writing.

    procedure scullc_trim( var sd: scull_dev in eax );
    begin scullc_trim;


        kassert( sd <> 0 );

        // Set the length to zero:

        mov( 0, (type scull_dev [eax]).len );

        // Return success:

        xor( eax, eax );

        kdebug
        (
            linux.printk
            (
                "<1>scullc_trim: exit (returning %d)\n",
                eax
            );
        );

    end scullc_trim;


    procedure init_module; @noframe;
    begin init_module;

        reg_save( ecx, edx );


        kdebug( linux.printk( "<1>init_module\n" ) );

        // Initialize the module owner field of scullc_fops:

        mov( &linux.__this_module, scullc_fops.owner );

        // Clear out the scullc_devices array:
```

```
kdebug
(
    linux.printk
    (
        "<1>Clearing out the scullc_devices array\n"
    );
);
mov( 3 * @size( scull_dev ), ecx );  // Select last element.
repeat

    scullc_trim( scullc_devices[ecx] );

    // Initialize the object's semaphore:

    linux.sema_init( (type scull_dev scullc_devices[ecx]).sem, 1 );

    // Move on to the previous element of the scull array:

    sub( @size( scull_dev ), ecx );

until( @s ); // ecx < 0



// Register the major device number and accept a dynamic #:

kdebug
(
    linux.printk
    (
        "<1>Registering the character device, scullc_major=%d\n",
        scullc_major
    );
)

linux.register_chrdev( scullc_major, "scullc", scullc_fops );

kdebug
(
    linux.printk( "<1>Returned major #=%d\n", eax );
);

if( (type int32 eax) >= 0 ) then

    // If we're using a dynamically assigned major number,
    // scullc_major will contain zero;  that being the case,
    // store the returned major number into scullc_major:

    if( scullc_major = 0 ) then

        mov( eax, scullc_major );

    endif;

    // Unlike Rubini and Corbet, this version of "scullc"
    // only creates two scullc devices, it does not allow
    // the superuser to specify the number of scullc devices
    // at load time.  If you're comparing this code against
    // the 'C' code in LDD2, you'd normally expect to find
    // code that dynamically allocates memory for the
    // scullc devices here.  That's not necessary in this
```

```
            // program as we can allocate the storage space statically.

            xor( eax, eax ); // return success.

        endif;

        kdebug
        (
            linux.printk
            (
                "<1>scullc: init_module return code = %d\n",
                eax
            )
        );

        reg_restore;
        ret();

    end init_module;

    procedure cleanup_module; @noframe;
    begin cleanup_module;

        // Since this module uses static allocation for the
        // Scullc devices, all we really have to do is
        // unregister the "scullc" device's major number:

        kdebug( linux.printk( "<1>scullc: cleanup_module\n" ) );

        linux.unregister_chrdev( scullc_major, "scullc" );
        ret();

    end cleanup_module;


    end scullc;
```

3.13 Debugging Techniques

One of the greatest problems facing device driver (and kernel) programmers is the fact that it's very difficult to debug kernel code. It's difficult to run a kernel or device driver under a debugger nor is it easy to trace kernel code. Kernel errors are difficult to reproduce and track down because they often crash the whole system requiring a reboot (which tends to destroy the data you could use to track down the problem). Even if you're fortunately enough to have a hardware debugging device like an in-circuit emulator (ICE), it's difficult to trace through the kernel because it's just a set of "library functions" that application programs execute. You can't start and stop a "kernel process" like you can a normal application. These problems create lots of challenges for kernel developers.

This section attempts to introduce some techniques you can use to help trace through kernel code and track down problems that develop in your code.

3.13.1 Code Reviews

Without question, one of the most poweful tools you can use to track down problems in your device driver code is analysis. Software engineering research indicates that software engineers tend to find fewer than 60% of the defects in a program during normal testing and debugging. Since kernel testing and debug-

ging can hardly be considered 'normal' one would expect to find even fewer defects in one's kernel code when relying specifically on testing and debugging techniques.

Those same software engineers have found that a programmer can also discover about 60-70% of the defects in a program through code reviews and code analysis (the two sums are greater than 100% because there is considerable overlap between the defects the two mechanisms discover). Although normal testing and debugging operations are much difficult in kernel code, there is really no additional effort to review kernel code than the effort required to review an application with compariable complexity. Therefore, a thorough code review should be the first tool a software engineer uses when attempting to discover the defects in a device driver. Unfortunately, my experience suggests that few device driver engineers even think to carefully analyze their code in an attempt to locate defects. For example, Rubini and Corbet devote a whole chapter to *Debugging Techniques* in *Writing Linux Device Drivers, Second Edition*, and nowhere do they mention using a code review to locate defects in a device driver (I will give them the benefit of the doubt and assume that they assume the reader has already analyzed their code; based on my observations, I will not make this assumption about all my readers).

This chapter, indeed this text, is a far to short to go into all the details behind a full code analysis and review process. However, it's easy to make two simple suggestions that are surprisingly effective in tracking down defects that would be difficult to find using normal debugging techniques: (1) read your code and convine yourself it's correct, and (2) have someone else read your code and have them figure out what it's doing.

The first suggestion may seem somewhat stupid. After all, you've written your code, how is reading it going to prove anything about it? Simple, it's called the mental block. If you're like me, your mind often thinks one thing while your fingers type something else during the coding process. The bad news is that it's easy to read what you were thinking rather than what you actually typed when scanning over your code. Therefore, when personally reviewing your code, it's important to look at one statement at a time and ask yourself "Do I really understand *exactly* what this statement is doing?" Be sure that you understand both the main intent of the statement and any side effects the statement produces before going on to the next statement. Look at the effects of that statement both locally and globally. When you've convinced yourself that the statement is correct, go on to the next one. This may seem laborious (and it is), but it's amazing how many defects you can find in the code by following this procedure. In the long run, this takes far less time to find many defects than you'll take using traditional testing and debugging techniques. Best of all, this technique finds defects in code sections that hardly (if ever) execute as easily as it finds defects in code that execute all the time (these latter ones are the ones that are most easily found using traditional testing and debugging techniques).

One problem with reviewing your own code is that you can develop mental blocks and no matter how many times you read over a section of code, you're likely to always see what you *think* you see rather than what's actually present in the source. Therefore, once you've made a couple of passes over your code, it's a good idea to have someone else take a look as well. Since they're operating from a different frame of reference, they'll see problems that you continue to miss. Do keep in mind, however, that you shouldn't waste another engineer's time; it's important that you review your own code first and eliminate all the "low lying fruit" before taking up someone else's time in a review process. Remember, when someone else is reviewing your code, you're consuming two man hours for each real hour since two of you are participating in the review. And you should always participate; never hand the code off to someone else and ask them to get back to you later. You'll learn a lot more about your code and the two of you will find far more defects together. Just be sure the easy stuff is taken care of before you involve someone else.

A good way to do a code review with another engineer is to give them the code (and design documentation!) ahead of time so they can read through the code and learn how it operates. Then have them explain how they believe it operates to you. This is actually the best way to do the reviews. It's amazing how many misconceptions they'll have about the code and, although their misconceptions may be nothing more than misconceptions, this feedback will help you document your code better. On the other hand, the things that they just don't understand in your code could truly be due to defects in your code. This is the most powerful form of code review you can do involving other engineers; unfortunately, those other engineers often have their own projects and don't "own" your code; therefore, they're unlikely to invest the time that's necessary to understand the code to the level needed to pull this type of review off really well. This type of review works best when your manager or a project lead is the one who must understand your code (so they can handle project scheduling better) or if you've got someone who is going to take over your code when you're

done with it. That is, this process works well when you've got someone who has a vested interest in your code.

In a perfect world, every project has a back-up engineer who can take over the project should something happen to the primary engineer (such as leaving for a better opportunity). In the real world, project schedules, engineering costs, the marketing department pressures, and the lack of engineering resources all conspire to prevent things from getting done the "right way." Therefore, code reviews as described immediately above rarely take place in many companies.

A second way to do a code view involving one or more engineers beyond the principal engineer is to have the projects "owner" hand out listings to the code and then go through the code line by line, explaining it to the engineers in attendance. Although there is the danger that this explanation may propogate the mental blocks from the principal engineer to the others in attendance. However, just as often you'll have one of the reviewers pipe up and say something "No! That's not what this code does!" Indeed, in my personal expeience, simply explaining my code to others and answering their questions about the code often causes me to discover the problems myself. I could have read the code over by myself a dozen times and missed the problem each time, but the moment I attempt to explain it to someone else, the problem becomes clear to me.

3.13.2 Debugging By Printing

The most common kernel debugging technique is monitoring (or instrumenting) your code. This is accomplished by printing important data at various points in the device driver code. When something goes wrong, you can inspect the debug log in an attempt to determine the cause of the problem.

3.13.2.1 linux.printk

As you've seen already, the linux.printk function is the basic tool for dumping data during the execution of your device driver. In past examples, this document has made a couple of simplifying assumptions: (1) you're familiar with the C standard library printf function and (2) printk behaves just like printf except for the mysterious "<1>" prefix that must appear in the format string. I'll continue to make assumption (1). If you're not completely familiar with the C standard library's printf function, you should check out the description found in the Linux man page or read about it in LDD2 or some other text. As for assumption (2), it's time to look at the operation of linux.printk in a little more detail.

One of the big differences between linux.printk and the C printf functions is that "<1>" prefix to the format string. This prefix specifies the severity or *loglevel* of the message. The *kernel.hhf* header file (included by *linux.hhf*) defines the following symbolic names for the eight loglevels that Linux supports:

```
const

    // The following kernel constants also exist outside the
    // linux namespace because they get used so often it's too
    // much of a pain to always type "linux." as a prefix to them:

    kern_emerg      :text := """<0>"""; // system is unusable
    kern_alert      :text := """<1>"""; // action must be take immediately
    kern_crit       :text := """<2>"""; // critical conditions
    kern_err        :text := """<3>"""; // error conditions
    kern_warning    :text := """<4>"""; // warning conditions
    kern_notice     :text := """<5>"""; // normal, but significant condition
    kern_info       :text := """<6>"""; // informational
    kern_debug      :text := """<7>"""; // debug-level messages
```

As you'll note by reading the comment above, these constants are defined outside the linux namespace, so you don't need to prefix them with "linux." in your source code. This was done because you'll probably use (some of) these constants often and their names are unlikely to conflict with names in your code, so saving some typing is worthwhile. The comments appearing next to each constant pretty much state the severity level associated with the prefix. Here's how C&R describe each of these severity levels:

kern_emerg	Used for emergency message; i.e., those that immediately precede a kernel crash (such as an assertion failure right before a NULL reference).
kern_alert	A situation requiring immediate action or the kernel will fail.
kern_crit	Critical conditions, typically related to hardware or software failures (e.g., an assertion failure that probably won't cause the kernel crash, but will definitely cause the device driver to malfunction).
kern_err	Used to report error conditions. For example, device drivers will often use this loglevel to report errors returned by the hardware (as opposed to a hardware failure, which is a bit more serious).
kern_warning	Warnings about problematic situations that do not, by themselves, create problems for the system (though cascading warnings might...)
kern_notice	Situations that are normal, but still worthy of note. For example, many kernel routines report security related issues at this log level.
kern_info	Informational messages. Many device drivers print messages at this loglevel when logging information about the hardware they find and the initialization they perform.
kern_debug	Typical "Here I Am" and instrumentational messages you'll print during debugging.

The kernel may treat messages differently depending on the log level if you're running on a text-based console. However, if you're running under XWindows (e.g., Gnome or KDE) then the messages are always written to the */var/log/messages* file (which is probably better, since if you really crash the system the output will be waiting for you in this file after you reboot Linux). For more details on how Linux treats these different loglevel prefixes, please see LDD2. This text assumes that the system always writes the output messages to the log file.

When calling the linux.printk function, you can use the symbols above as follows:

```
linux.printk( kern_debug "Debug message" nl );
```

Note that there is no comma between the kern_debug and the "Debug message" items. HLA expands this to the following statement:

```
linux.printk( "<7>" "Debug message" nl );
```

When HLA finds two juxtaposed strings like "<7>" and "Debug message" above, the compiler concatenates the two strings to produce:

```
linux.printk( "<7>Debug message" nl );
```

(note, by the way, that the nl item above also expands to a string constant containing the new line character, so the above linux.printk parameter actually expands to nothing more than a single string constant.)

The important thing to note is that you can specify the Linux names for the log levels as though they were pseudo parameters and leave it up to HLA to do the rest. Just remember not to place a comma between the kernel loglevel string constant and the linux.printk format string.

3.13.2.2 Turning Debug Messages On and Off

During the earliest stages of development on your driver, or when tracking down a partricularly nasty defect, you'll probably want to print just about everything in site. As your driver becomes stable, and certainly by the time you release your driver, you'll want to eliminate the debugging output it produces. Removing all the linux.printk calls in your code is problematic for two reasons: (1) you'll undoubtedly miss some of the calls and your driver will continue to write data to the log file, even once you've releaseed it. Since writing data to the log file is very slow (as well as it wipes out other, possibly important, log data), this is not a good situation. (2) The moment you eliminate a linux.printk because you think you'll never need it again, you'll discover a defect that makes you wish you'd not removed the printk call.

You've already seen a global solution to this problem – use the kdebug macro to enclose debugging statements (like calls to linux.printk) that you'd like to be able to turn on and off by setting the value of the KNDEBUG compile-time variable.

If you're an extremely lazy typist and you don't want to have to type "kdebug(--)" around your calls, or even worse, you don't even want to have to type the "linux." prefix, you can get sneaky and create an HLA sequence like the following:

```
#if( KNDEBUG )
    #macro printk( dummy[] ); // dummy arg allows multiple parameters
    #endmacro;
#else
    const
        printk :text := "linux.printk";
#endif;
```

If KNDEBUG is false, then an HLA statement of the form

```
printk( kern_debug "Hello" );
```

expands to

```
linux.printk( "<7>Hello" );
```

On the other hand, if KNDEBUG is true, then HLA expands the printk invocation above to nothing (since the prink macro above is empty). The sequence above is not a part of the linux.hhf header file set, but it's easy enough to add to your own code if you want to be able to control code emission of the printk statements in your code without the expense of typing kdebug(--) everywhere.

3.13.2.3 Debug Zones

Earlier, this text mentioned that you can control debugging output in your device drivers by sprinkling statements like "?KNDEBUG := false;" and "?KNDEBUG := true;" throughout your code. In the range of statements between the points where you set KNDEBUG false and where you set it true, debug output will be enabled. Conversely, after you set KNDEBUG true (and up to the point you set it false), debugging output will be disabled. This is cool and useful, but it suffers from the same problem as burying bare linux.printk calls in your code – if you forget to remove the statements that set KNDEBUG to false from your production code, then your released driver will be writing debug code to the log file (which is bad). One solution to this problem is to set up debug sets or debug zones.

The idea behind a debug zone or a debug set is to classify your debug statements into sets and enable or disable your debug statements as sets. A simple way to do this is to create a 32-bit compile-time variable (HLA VAL object) and use each bit of that object to represent one debug zone (set element). At the beginning of a source file you can control up to 32 sets of debug statements by modifying a single statement in the program. The following example demonstrates how to do this with four debug zones:

```
const
    dbgz0 := @{0};  // bit zero is set
    dbgz1 := @{1};  // bit one is set
    dbgz2 := @{2};  // bit two is set
    dbgz3 := @{3};  // bit three is set

val
    dzone :dword := dbgz0 | dbgz2 | dbgz3; // activate zones 0, 2, and 3.

    #macro zdebug( zone, instrs );
        #if( !KNDEBUG & ((dzone & zone) != 0 ))
            returns
            ({
                instrs
            },"")
        #endif
```

```
#endmacro;
```

You use the zdebug macro as follows:

```
zdebug
( dbgz2,  // Do the following if debug zone 2 (dbgz2) is active:
    linux.printk( kern_debug "Some message goes here" nl );
);
```

If the global dzone compile-time variable has bit two set (i.e., you've OR'd in dbgz2 in the VAL statement above) and the KNDEBUG compile-time variable contains false, then HLA will emit the code that displays the message above at run-time. On the other hand, if bit two is clear (or KNDEBUG is true), then HLA will not emit zone two debugging code to the object file.

If the use of zdebug above looks a little clumsy, you can always create another macro that simplifies the zone two invocations as follows:

```
#macro dzone2( instrs );
    zdebug( dbgz2, instrs )
#endmacro;
```

You can invoke the dzone2 macro as follows:

```
dzone
(
    linux.printk( kern_debug "Some message goes here" nl );
);
```

This isn't any more difficult than the kdebug macro that the *linux.hhf* header file set provides.

Since dzone is a dword object, you can have up to 32 debug zones in your code. Note that this doesn't limit you to 32 debug statements, but rather it limits you to being able to independently control 32 different sets of debug statements. Many programmers, for example, will place all function entry/exit output statements into one set, initialization-related debugging statements will go into a second set, etc. If this still isn't enough debug zones for you, you can always create a compile-time array of dwords to maintain more debug zones (though the check in the #if statement starts to get a little nastier).

3.13.3 Debugging by Querying

A big problem with printing lots of data is that is slows down the system considerably. Besides the annoyance factor and the possiblity that it make take a long time to encounter the problem with all the debugging output taking place, there are some other problems as well. First of all, degrading the performance of your driver could cause it to fail. Some devices may produce data faster than your driver can accept it while the driver is producing a ton of debugging output. Even if this is not the case, it could turn out that slowing down your driver eliminates some race condition and it mysteriously starts working when debug output is turned on. Therefore, producing volumes of output is not always a good way to attempt to debug your code. Another solution is to query your driver about its status while it's running. The advantage of querying is that the slow stuff (formatting and output) takes place in an application program, not in your driver. There are other advantages to querying as well, but it does suffer from a couple of major disadvantages: specifically you must add code to your driver to handle query requests, and second, a query could miss important information not available outside the driver. Nevertheless, querying is a good way to get information from the driver in a high-performance fashion; especially if that information is persistent within the driver.

Querying issues are not specific to any one give language. Therefore, I will defer this discussion to LDD2. Please see LDD2 for an in-depth discussion of this debugging technique. The LDD2 chapter on debugging techniques presents several other language-independent suggestions for debugging the kernel, in the interest of moving on to more assembly-language related topics, I once again defer further discussion to LDD2.