Writing Linux Device Drivers in Assembly Language

(Full Contents)

Writing Linux Device Drivers in Assembly Language

Written by Randall Hyde

0 Preface and Introduction

This document will attempt to describe how to write Linux device drivers (modules) in assembly language. This document is not self-contained; that is, you cannot learn everything you need to know about Linux device drivers (assembly or otherwise) from this document. Instead, this document is based on the text "Linux Device Drivers, Second Edition" by Rubini & Corbet (published by O'Reilly & Associates, ISBN 0-596-00008-1). That text explains how to write device drivers using C, this document parallels that text, converting the examples to assembly language. However, to keep this document relatively short, this document does not copy the information that is language-independent from the text. Therefore, you'll need a copy of that text to read along with this document so the whole thing makes sense.

Rubini & Corbet have graciously chosen to make their text freely available under the GNU Free Documentation License 1.1. Therefore, this text inherits that license. You can learn more about the GNU Free Documentation License from

`http://www.oreilly.com/catalog/linuxdrive2/chapter/licenseinfo.html`

The programming examples in this text are generally translations of the C code appearing in "Linux Device Drivers" so they also inherit Rubini & Corbet's licensing terms. Please see the section on licensing in "Linux Device Drivers, Second Edition," orthe text file LICENSE with the distributed software for more details on the licensing terms.

If you're not reading this copy of "Webster" you'll probably want to check out the Webster website at

`http://webster.cs.ucr.edu`

where you'll find the latest copy of this text and its accompanying software.

0.1 Randy's Introduction

As an embedded engineer, I've had the opportunity to deal with Linux device drivers in the past (back around Linux 1.x when the device driver world was considerably different). Most of the device drivers for Linux I'd dealt with were quite simple and generally involved tweaking other device drivers to get the functionality I was interested in. At one point I needed to modernize my Linux device driver skills (Linux v2.x does things way differently). As I mastered the material, I wrote about it in this document so I could share my knowledge with the world at large.

0.2 Why Assembly Language?

Rather than take the "just hack an existing driver" approach, I wanted to learn Linux device drivers inside and out. Reading (and doing the examples in) a book like *Linux Device Drivers* is a good start, but as I get tired reading I tend to gloss over important details. I'm not the kind of person who can read through a text once or twice and completely master the material; I need to actually *write code* before I feel I've mastered the material. Furthermore, I've never been convinced that I could learn the material well by simply typing code out of a textbook; to truly learn the material I've always had to write my own code implementing the concepts from the text. The problem with this approach when using a text like *Linux Device Drivers, Second Edition* is that it covers a lot of material dealing with real-world devices. Taking the time to master the particular peripherals on my development system (specific hard drives, network interface cards, etc.) doesn't seem like a good use of my time. Dreaming up new pseudo devices (like the ones Rubini & Corbet

use in their examples) didn't seem particularly productive, either. What to do? It occurred to me that if I were to translate all the examples in C to a different programming language, I would have to really understand material. Of all the languages besides C, assembly is probably the most practical language with which one can write device drivers (practical from a capability point of view, as opposed to a software engineering point of view). The only reasonable language choice for Linux device drivers other than C is assembly language (that is, I *know* that I'd be able to write my drivers in assembly since GCC emits assembly code; I'm not sure at all that it would be possible to do this in some other language I have access to).

Rewriting Rubini & Corbet's examples in a different language certainly helps me understand what they're doing; rewriting their examples in assembly language really forces me to understand the concepts because C hides a lot of gory details from you (which Rubini & Corbet generally don't bother to explain). So that was my second reason for using assembly; by using assembly to write these drivers I *really* have to know what's going on. This experience was valuable not only because it forced me to learn Linux device drivers to a depth I would have never otherwise attained, but it also taught me a lot of in-depth *Linux* programming, as well. You won't fully appreciate the complexity of the Linux system until you've converted a large number of C header files to assembly language (and verified that the conversion is correct).

Note that all the examples in this text are pure assembly language. I don't write a major portion of the driver in C and then call some small assembly language function to handle some operation. That would defeat the purpose for (my) using assembly language in the first place, that is, forcing me to really learn this stuff.

Of course, many people really want to know how to write Linux device drivers in assembly language. Either they prefer assembly over C (and many people do, believe it or not), or they need the efficiency or device control capabilities that only assembly language provides. Such individuals will probably find this document enlightening. While those wanting more efficient code or more capbility could probably use the C+assembly approach, they should still find this document interesting.

Of course, any die-hard Unix/Linux fan is probably screaming "don't, don't, don't" at this point. "Why on Earth would anyone be crazy enough to write a document about assembly language drivers for Linux? " they're probably saying. "Doesn't this fool (me) know that Linux runs on different platforms and assembly drivers won't be portable?" Of course I realize this. I'm also assuming that anyone bright enough to write a Linux device driver *also* realizes this. Nevertheless, there are many reasons for going ahead and writing a device driver in assembly, portability be damned. Of course, portability isn't that much of an issue to most people since the vast majority of Linux systems use the x86 processor anyway (and, most likely, the devices that such people are writing drivers for may only work on x86 systems anyway).

0.3 Assembly Language Isn't *That* Bad

Linux & Unix programmers have a pathological fear of assembly language. Part of the reason for this fear is the aforementioned portability issue. *NIX programmers tend to write programs that run on different systems and assembly is an absolute no-no for those attempting to write portable applications. In fact, however, most *NIX programmers only write code that runs on Intel-based systems, so portability isn't *that* much of an issue.

A second issue facing Linux programmers who want to use assembly is the toolset. Traditionally, assemblers available for Linux have been very bad. There's Gas (as), the tool that's really intended only for processing GCC's output. Most people attempting to learn Gas give up in short order and go back to their C code. Gas has many problems, not the least of which it's *way* underdocumented and it isn't entirely robust. Another assembler that has become available for Linux is NASM. While much better than Gas in terms of usability, NASM is still a bit of work to learn and use. Certainly, your average C programmer isn't going to pick up NASM programming overnight. There are some other (x86) assemblers available for Linux, but I'm going to use HLA in this document.

HLA (the High Level Assembler) is an assembler I originally designed for teaching assembly language programming at UC Riverside. This assembler is public domain and runs under Windows and Linux. It contains comprehensive documentation and there's even a University-quality textbook ("The Art of Assembly Language Programming") that newcomers to assembly can use to learn assembly and HLA.

I'm using HLA in the examples appearing herein for several reasons:

- I designed and wrote HLA; so allow me to toot my own horn,
- HLA source code is quite a bit more readable than other assembly code,
- HLA includes lots of useful libary routines,
- HLA is easy to learn by those who know C or Pascal,.
- HLA is far more powerful than the other assemblers available for Linux.

For more information on HLA, to download HLA, or to read "The Art of Assembly Language Programming" go to the Webster website at

`http://webster.cs.ucr.edu`

1 An Introduction to Device Drivers

Linux Device Drivers, Second Edition (LDD2), Chapter One, spends a fair amount of time discussing the role and position of device drivers in the system. Much of the material in this chapter is independent of implementation language, so I'll refer you to that text for more details.

There are, however, a couple of points that are made in LDD2 that are worth repeating here. First, this document, like LDD2, deals with device drivers implemented as *modules* (rather than device drivers that are compiled in with the kernel). Modules have several advantages over traditional device drivers including: (1) they are easier to write, test, and debug, (2) a (super) user can dynamically load and unload them at run-time, (3) modules are not subject to the GPL as are device drivers compiled into the system; hence device driver writers are not compelled to give away their source code which might leave them at a commercial disadvantage.

Rubini & Corbet graciously grant permission to use the code in their book on the condition that any derived code maintain some comments describing the origin of that code. I usually put my stuff directly in the public domain, but since my code is (roughly) based on their code, I will keep the copyright on this stuff and grant the same license. That is, you many use any of the code accompanying this document as long as you maintain comments that describe its source (specifically, LDD2 and Rubini & Corbet, plus mentioning the fact that it's an assembly translation by R. Hyde).

In chapter one, Rubini & Corbet (R&C here on out) suggest that you join the Kernel Development Community and share your work. While this is a good idea, I'd strongly recommend a flame-resistant suit if you're going to submit drivers written in assembly language to the Linux community at large. I anticipate that assembly drivers will not be well-received by the Linux/Unix community. Don't say you weren't warned.

2 Building and Running Modules

Before you can learn how to write a real-world device driver in assembly langauge, you need to learn how to use the device driver development tools that Linux provides. In particular, you need to learn how to compile your modules[1], install them into Linux, and remove them.

The first thing to realize is that you do not compile your device drivers into executable programs. Instead, you only compile them to object files (".o" files). This may seem strange to someone who is used to using ld to link together separate object files to produce an executable; after all, how do you resolve external symbol references? Well, as it turns out, most of the external references that will appear in your drivers will be references to kernel code. There is no "library" you can link in to satisfy these references. It turns out to be the Linux kernel's responsibility to read your object files and perform any necessary linkage directly to kernel code. Because of the way this process works, you generally shouldn't call any functions in the HLA Standard Library that directly or indirectly invoke a Linux system call. Furthermore, it's generally a real bad idea to use HLA exception handling (unless you can guarantee that your code handles all possible exceptions that could come along and you're willing to initialize the exception handling system). Based on these constraints, there are actually only a few HLA Standard Library routines you can call. If you're a long-time HLA programmer, this may bring tears to your eyes, but it's best to avoid much of the HLA Standard Library until you have a good feel for what's legal and what's not (alternately, you have access to the HLA Standard Library source code, so feel free to strip out the exception code in the library routines that don't call Linux; then you'll be able to use those routines without any problems).

The first thing to do when writing a bunch of code that makes kernel calls is to decide what to name the functions, types, and data. At first, this may seem to be a trivial exercise – we'll use the same names that the Linux kernel uses. There are, however, a couple of problems with this approach. One problem is *namespace pollution*; that is, there are thousands of useful symbols that refer to objects in the kernel. Despite the best efforts by Linux kernel developers to mangle these names, there is still the likelihood that the kernel will use a name that you're attempting to use for a different purpose in your programs. The solution to this problem is clear, we'll use an HLA namespace declaration to prevent namespace pollution. This is a common technique the HLA Standard Library uses to avoid namespace pollution (and is the reason you have function names like stdout.put and linux.write; the stdout and linux identifers are namespace IDs). In theory, we could just add all of our new kernel symbols to the existing linux namespace. The only problem with this idea is that the linux namespace contains symbols that Linux application programmers use; placing kernel declarations in the linux namespace would suggest that someone could use those symbols in normal application programs; fortunately, we'll use the same trick the kernel does to hide kernel-only symbols from the user – we'll require the declaration of the "__kernel__" symbol in order to make kernel symbols visible. This, plus the fact that all standard kernel symbols are unique within the linux namespace, will prevent conflicts with symbols in our device driver code.

Using the "linux" namespace reduces the namespace pollution problem, but even within that namespace there are a couple of reasons this document won't simply adopt all the Linux kernel identifiers. The first reason is one of style: following the C/C++ tradition, most constants and macros in the kernel header files are written in all upper case. This is horrible programming style because uppercase characters are much harder to read than lowercase. Since this text deals with assembly language, not C/C++, I do not feel compelled to propogate this poor programming practice in my examples. However, using completely different identifiers would create problems of its own (since there is a lot of documentation that refers to the C identifiers, e.g., LDD2). Therefore, I've adopted the convention of simply translating all uppercase symbols to all lower case (even when mixed case would probably be a better solution). This makes translating C identifiers to HLA identifiers fairly easy.

By the way, if you get tired of prepending the string "linux." to all the Linux identifiers, you can always create an HLA text constant that expands to "linux" thusly:

```
const
    k :text := "linux";
```

1. HLA v1.x programmers use the term "compile" rather than "assemble" to describe the process of translating HLA source code into object code. This is because HLA v1.x is truly a compiler insofar as it emits assembly code that Gas must still process in order to produce an ".o" file.

With this text constant appearing your your program you need only type "k.*linux_namespace_id*" instead of "linux.*linux_namespace_id*". This can save a small amount of typing and may even make your programs easier to read if you use a lot of linux namespace identifiers in your code[2].

Another problem with C identifiers is case sensitivity and the fact that structs and functions have different name spaces. Many Linux/Unix kernel programmers have taken advantage of this lexical "feature" to create different identifiers in the program whose spelling only differs by alphabetic case, or, they use the same exact identifiers for structures and functions. Since HLA doesn't allow this, I have had to change the spelling of a few identifiers in order to satisfy HLA. A related problem is the fact that HLA and C have different sets of reserved words and some of the Linux kernel identifiers conflict with HLA reserved words. Again, slight changes were necessary to accomodate HLA. Here are the conventions I will generally employ when changing names:

- All uppercase symbols will become all lowercase.
- If a struct ID and a function ID collide, I will generally append "_t" to the end of the structure ID (a typical Unix convention; I wonder why they didn't follow it consistently in Linux).
- In the case of any other conflict, I will usually prepend an underscore to one of the identifiers to make them both unique (generally, this occurs when there is a conflict between a C identifier and an HLA reserved word).

2.1 The "Hello World" Driver Module

The "Hello World" program (or something similar) is the standard first program any programmer writes for a new system. Since Linux device driver programming is significantly different than standard C or assembly programming, it makes since to begin our journey into the device driver realm by writing this simply program. Since any HLA StdLib routine that ultimately calls Linux is out, this means that all the standard output stuff is verboten. This presents a problem since stdout.put is a favorite debugging tool of HLA programmers. Fortunately, the kernel supplies a debug print routine, printk, that we can use. The printk procedure is very similar to the C printf function, see TDD2/Chapter One for more details. Unfortunately, printk, like printf, is one of those pesky C functions that is difficult to call from an HLA program because it relies upon variable parameter lists and HLA's high level procedure declarations and invocations don't allow variable parameter lists. Of course, we can always push all the parameters on the stack manually and then call printk like one would with any standard assembler, but this is a pain in the rear for something you'll use as often as *printk*. So we'll write an HLA macro (HLA macros do support variable parameter lists) that handles the gross work for us. The printk macro takes the following form:

```
namespace linux;

    // First, we have to tell HLA that the _printk (the actual
    // printk function) is an external symbol.  The Linux
    // kernel will supply the ultimate target address of this
    // function for use.  Use the identifier "_printk" to
    // avoid a conflict with the "printk" macro we're about
    // to write.

    procedure _printk; external( "printk" );

    macro printk( fmtstr, args[]):index, msgstr;

        ?index :int32 := @elements( args );
        #while( index >= 0 )
```

2. The letter "k" was chosen for "kernel" rather than "l" for Linux. "l" is a bad choice because it looks like the digit one in your listings. Sometimes you will see me use "k.identifier" in sample code because I personally adopt this convention. However, I will attempt to use linux.*identifier* in most examples to avoid ambiguity.

```
            push( @text( args[index] ));
            ?index := index - 1;

        #endwhile
        readonly
            msgstr:byte; @nostorage;
                   byte fmtstr, 0;
        endreadonly;
        pushd( &msgstr );
        call linux._printk;
        add( (@elements( args ) + 1)*4, esp );

    endmacro;

end linux;
```

Program 3.1 printk Macro Declaration

For those who are not intimately familiar with HLA's macro and compile-time language facilities, just think of the linux.printk macro as a procedure that executes during compilation. The macro declaration states that the caller must supply at least one parameter (for fmtstr) and may optionally have additional parameters (held in the args array, which will be an array of strings, one string holding each parameter). index is a local compile-time variable that this macro uses to step through the elements of the array. The HLA compile-time function @elements returns the number of elements in the args array. Therefore, the #while loop in this macro steps through the array elements, from last to first, assuming there is at least one element. Within this #while loop, the macro emits a push instruction that pushes the specified macro parameter. Note, however, that any parameters you supply beyond the format string must be entities that you can legally push on the stack[3].

The fmtstr must be a string constant. Again, HLA provides the capability of verifying that this is a string constant, but for the sake of keeping this example as simple as possible, we'll assume that fmtstr is always a string constant. Since you cannot push a string constant with the x86 pushd instruction, the macro above emits the string constant to the readonly segment and pushes the address of this constant. Although HLA strings are upwards compatible with C's zero-terminated string format, the linux._printk function cannot take advantage of this additional functionality, so the k.printk macro emits a simple zero-terminated string for use by linux._printk. Note the use of the macro's local msgstr symbol to guarantee that each invocation of linux.printk generates a unique msgstr label.

Note that linux._printk uses the C calling convention, so it's the caller's resposibility to pop all parameters off the stack upon return. This is the purpose of the add instruction immediately following the call instruction. Each parameter on the stack is four bytes long, so adding four times the number of parameters (including the fmstr parameter) to esp pops the parameter data off the stack.

The following code snippets provide an example of how the linux.printk macro expands its parameter list. (Note that the expansion of the local symbols will be slightly different in actual practice; these examples create their own unique IDs just for the purposes of illustration.)

```
// linux.printk( "<1>Hello World" ); expands to:

    readonly
        L_0001:    byte; @nostorage;
```

3. HLA's macro facilities are actually sophisticated enough to detect constants and emit proper code for them. However, 99% of the time we're only going to be passing dword parameters (if we're passing any parameters beyond the fmtstr at all), so we'll ignore the extra complexity that would be required to handle other data types. For the other 1% of the time we do need other types, we can manually push the parameters.

```
                byte "<1>Hello World", 0;
    endreadonly;
    pushd( &L_0001);
    call linux._printk;
    add( 4, esp );

// linux.printk( "<1>My Variable V=", v );

    pushd( v );
    readonly
        L_0002:    byte; @nostorage;
                   byte "<1>My Variable v=", 0;
    endreadonly;
    pushd( &L_0002 );
    call linux._printk;
    add( 8, esp );
```

Program 3.2 printk Macro Invocation Examples

There are a few important differences between the Linux printk and C printf functions. First of all, don't bother trying to print multiple lines by injecting newlines into your strings. The printk function's design assumes one line of text per call. Furthermore, printk prints some additional information at the beginning of each line (to aid in debugging). This is a kernel debugging aid, not a general purpose output routine; please keep this in mind. Also note that your format strings should begin with "<1>" or a similar string with a low-valued numeric digit within angle brackets. This sets the *priority* of the message. See LDD2 or documentation for printk for more details. If you don't want to bother learning about printk at this level of detail, just always stick "<1>" in front of your format strings. If this gets to be a pain, just modify the linux.printk macro[4] so it does this for you automatically, e.g.,

```
msgstr:    byte; @nostorage;
           byte "<1>" fmtstr, 0;
```

Armed with an output procedure, we're almost to the point where we can write our "Hello World" program as a device driver module. There are only three additional issues we've got to deal with and then we'll be ready to create and run our first assembly language device driver module.

The first issue is that device drivers must be compiled for a specific version of the Linux kernel. We need some mechanism for telling the kernel what version our device driver expects. The kernel will reject our module if the kernel version it assumes does not match the version that is actually running. Therefore, we have to insert some additional information into the object module to identify the kernel version that we expect. Unfortunately, HLA does not contain ELF/Linux specific directives that let us supply this information. Fortunately, however, HLA does provide the #asm..#endasm directive that let us emit Gas code directly into HLA's output stream. Since Gas does provide directives to set up this version information, we can supply this information between the #asm and #endasm directives. An example of this appears in the following listing:

```
#asm
           .section.     modinfo,"a",@progbits
           .type         __module_kernel_version,@object
           .size         __module_kernel_version,24

__module_kernel_version:
           .string"      kernel_version=2.4.7-10"
#endasm
```

4. Note that this macro declaration appears in the *kernel.hhf* header file.

Program 3.3 Code to Emit Version Information to a Module's Object File

These statements create a special *modinfo* record in the ELF object file that the kernel will check when it loads your device driver module. If the string specified by the __module_kernel_version symbol does not match the kernel's internal version, Linux will reject your request to load the module. Therefore, you will need to adjust the version number following "kernel_version=" in the example above to exactly match your kernel's version. Note that you must also adjust the value 24 appearing at the end of the .size directive if the length of your string changes when you change the version number.

The only problem with placing this code directly in your assembly file is that you'll have to modify the source file whenever the Linux version changes (which it does, frequently). If you've got to compile your driver for multiple versions of Linux (new versions are coming out all the time and you'll often have to support older versions of Linux as well as the latest), or if you've got several different drivers and you need to compile them for the newest version of Linux, going in and editing the version numbers in all these files can be a real pain. Fortunately, HLA's powerful compile-time language facilities come to the rescue. A little bit later you'll see how to write a compile-time program to automatically extract this information.

Another issue with writing device drivers is that you don't write a traditional HLA program for your device drivers. Indeed, device drivers don't have a "main program" at all. Instead, we'll write each device driver as an HLA unit that exports two special symbols: init_module and cleanup_module. The kernel will call the init_module procedure when it first loads your driver; this roughly corresponds to the "main program" of the module. The init_module procedure is responsible for any initialization your driver requires, including registering other functions with the kernel. If your init_module successfully initializes, it must return zero in eax. It should return -1 upon failure. Note that the driver does not quite upon return from init_module. It remains in system memory until the system explicitly removes the driver.

The kernel calls your cleanup_module procedure just prior to removing your driver from memory. This allows you to gracefully shut down any devices your driver is controlling and release any system resources your driver is using. Note that cleanup_module is a procedure (i.e., a C void function) and doesn't return any particular value in eax.

For the purposes of our "Hello World" device driver example, we'll simply print a couple of strings within the init_module and clean_up module functions. The following listing provides the code for these two routines:

```
procedure init_module; @nodisplay; @noframe;
begin init_module;

    linux.printk( "<1>Hello World\n" );
    xor( eax, eax );                // Return success.
    ret();

end init_module;

procedure cleanup_module; @nodisplay; @noframe;
begin cleanup_module;

    linux.printk( "<1>Goodbye World\n" );
    ret();

end cleanup_module;
```

Program 3.4 The init_module and cleanup_module Functions for the "Hello" Driver

There is only one more detail we've got to worry about before we can create a working "Hello World" device driver module. The Linux kernel maintains a *reference count* for each module it loads. As different processes open (i.e., use) a device driver, Linux increments that driver's reference count. As each process closes the device, Linux decrements the driver's reference count. Linux will only remove a module from memory if the reference count is zero (meaning no process is currently using the device). When writing a device driver, we have to declare and export this reference count variable. This variable must be the name of the module with an "i" appended to the name. For example, if we name our module "khw" (kernel hello world) then we must declare and export a dword "khwi" variable. The following program is the complete "khw.hla" driver module that implements the ubiquitous "Hello World" program as a Linux device driver module in assembly language. Note that the *kernel.hhf* header file (included by *linux.hhf*) contains the linux.printk macro, among other things (though this example only uses the linux.printk declarations given earlier; you could substitute that code for the #include if you're so inclined).

Note: this document often refers to the *linux.hhf* header file as though it contains all the Linux declarations. In fact, the *linux.hhf* header file is very small, containing only a series of #includeonce statements that include in all of the linux-related include files (found in the */usr/hla/include/os* subdirectory). You could actually speed up the compilation of your modules a tiny amount by including on the files you need from the *include/os* subdirectory, but it's far more convenient just to include linux.hhf from the standard include directory. That is the approach the examples in this document will take.

```
#include( "getversion.hhf" )

unit kernelHelloWorld;
#include( "linux.hhf" )
procedure init_module; external;
procedure cleanup_module; external;

static
    khwi:dword; external;
    khwi:dword;

procedure init_module; @nodisplay; @noframe;
begin init_module;

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

end init_module;

procedure cleanup_module; @nodisplay; @noframe;
begin cleanup_module;

    linux.printk( "<1>Goodbye World\n" );
    ret();

end cleanup_module;

end kernelHelloWorld;
```

Program 3.5 The hello.hla Device Driver Module

Like C/C++, HLA uses the "external" directive to export (that is, make global) names appearing in the source file. Hence, the external definitions for init_module, cleanup_module, and khwi, above are actually "public" or "global" declarations; they don't suggest that these identifers have a declaration in a separate file.

Now that we've got an HLA source file to play with, the next step is to figure out how to actually build and install this device driver module. The first step is to compile the "khw.hla" source file to an object module using the following command:

```
hla -c khw.hla
```

The "-c" command line parameter is very important. Remember, device driver modules are object (".o") files, not executable files. If you leave off the "-c" parameter, HLA will attempt to run the *ld* program to link khw.o and produce an executable program. Unfortunately, *ld* will fail and complain that printk is an undefined symbol (it will also complain about a missing "_start" symbol, since this module has no main program)[5].

Once you've successfully created the "khw.o" object module, the next step is to load the module (and then unload it). This is accomplished via the Linux *insmod* and *rmmod* commands. Note that you need to be the super-user to run these commands. If you've logged in as root (never a good idea) you may execute these commands directly; if not, execute the Linux *su* command to switch to super-user mode[6]. The paths for the root account and other accounts are often different, so you may need to run the *insmod* and *rmmod* commands by supplying a full prefix to these programs (usually /sbin/insmod and /sbin/rmmod). Alternately, you can set the *path* environment variable to include the directory containing *insmod* and *rmmod*. This document will assume you have done so and I will assume you can simply type "insmod" or "rmmod" at the command line.

Assuming you're now the super-user, you can run the "Hello World" driver module via the following commands:

```
insmod ./khw.o
rmmod khw
```

If you're like most modern Linux users and you're running under Gnome or KDE, you won't see any output. However, if you're running in a text-based console window (as opposed to a GUI), then you will see "Hello world" printed immediately after you run *insmod* and you'll see "Goodbye world" displayed after you run *rmmod*. When operating in an X-terminal window (i.e., under Gnome or KDE), the Linux kernel writes all printk output to a special log file. This file is usually the "/var/log/messages" file, though the path may vary on some systems. To see the result of your driver's execution, simply dump this file using the *cat* or *tail* command, e.g.,

```
cat /var/log/messages
```

Presumably, you'll see the output from your driver as the last few lines of the *messages* file. Congratuations, you've just written and run your first Linux kernel driver module in assembly language!

2.2 Compiling and Linking Drivers

In the previous section, this document explains that device drivers must be ".o" files rather than executable files. Furthermore, because we can't use HLA's exception handling or calls to Linux, we don't include their header files or link the HLA Standard Library with our module. However, it is quite possible that you'll want to write your own library modules (or link in some "safe" HLA StdLib functions) with your main device driver module. The question is "How do we merge separate '.o' files to produce a single object file (versus an executable) that still has unresolved labels (e.g., printk)"? The answer? Use *ld's* "-r" command line option. Assuming you had split the khw.hla module into two modules, init_hello.hla and cleanup_hello.hla, you could compile and link them with the following commands:

```
hla -c init_hello.hla
hla -c cleanup_hello.hla
ld -r -o khw.o init_hello.o cleanup_hello.o
```

5. Technically, if these missing symbol errors are the only errors *ld* reports, you can ignore them since HLA has produced a perfectly valid ".o" file. However, those error messages often hide other error messages, so it's a good idea to recompile the code with the "-c" option anyway, just to be sure there are no other problems.
6. Presumably, if you're attempting to install device drivers on your machine, you've got super-user access.

After compiling and linking these modules in this fashion, you could use *insmod* and *rmmod* to load and unload khw.o exactly as before.

Obviously, if you're going to be working with separate compilation and linking together different object files when building your modules, it makes a lot of sense to use a *Makefile* to control the module's build. I will assume that you are familiar with make files. In the examples I provide, I will avoid the temptation to use fancy *Make* features and stick to simple rules and actions. That way, you should be able to read my makefiles without having to resort to a manpage or text on *Make*. The examples in this text are sufficiently short that there is no need to inject really crazy operations into the makefiles.

2.3 Version Dependency

As noted earlier, when you compile a driver module for Linux you must compile it for a specific version of the kernel. Linux will reject an driver whose version number does not match the kernel's. You can generally find the kernel's version number in the <linux/version.h> header file. Here's what that source file looks like on my development system:

```
#include <linux/rhconfig.h>
#if defined(__module__smp)
#define UTS_RELEASE "2.4.7-10smp"
#elif defined(__module__BOOT)
#define UTS_RELEASE "2.4.7-10BOOT"
#elif defined(__module__enterprise)
#define UTS_RELEASE "2.4.7-10enterprise"
#elif defined(__module__debug)
#define UTS_RELEASE "2.4.7-10debug"
#else
#define UTS_RELEASE "2.4.7-10"
#endif
#define LINUX_VERSION_CODE 132103
#define KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))
```

The very last UTS_RELEASE definition (in this particular file) is the one that defines the version number of my kernel for the standard distribution. This is the string you must cut an paste into the ".string" directive at the beginning of the *hello.hla* module (or any other device driver module you write). Since Linux kernel versions seem to change every time somebody sneezes, manually making these changes to your code can be a real bear. It's probably much better to write a little program to extract this information and build the Gas directives for you. Tools like AWK and PERL come to mind; if you're familiar with those tools, you could easily create a short script that extracts the information from the version.h file and builds the Gas directives. However, since this is a book that describes the use of assembly language, you guessed it, we're going to develop some code to handle this activity in HLA.

We're not actually going to write the code to extract the version number in HLA (although with HLA's pattern matching library, this is a trivial task). Instead, we're going to use HLA's compile-time language facilities and actually extract this information from a C header file during compilation!

```
// Open the C header file containing the version # string:

#openread( "/usr/include/linux/version.h" )

// Predeclare some compile-time variables (needed by the
// #while loop and the pattern matching routines):

?brk := false;
?r1 :string;
?r2 :string;
?r3 :string;
```

```
?r4 :string;
?version :string;
?lf := #$a;

// Until we hit the end of the version file, or we find the
// version number string, repeat the following loop:

#while( !brk & @read( s ) )

    // Look for lines that begin with "#define"

    #if( @matchstr( s, "#define", r1 ))

        // ...followed by any amount of whitespace...
        #if( @oneOrMoreWS( r1, r2 ))

            // ...followed by 'UTS_RELEASE "'
            #if( @matchstr( r2, "UTS_RELEASE """, r3 ))

                //...followed by a string of digits, dots, or dashes.
                // This will be our version number!

                #if( @oneOrMoreCset( r3, {'0'..'9', '.', '-'}, r4, version ))

                    // Using the version number we just extracted, create
                    // an include file with the directives we need to pass
                    // along to Gas:

                    #openwrite( "version.hhf" )
                    #write( "#asm", lf )
                    #write( " .section  .modinfo,""a"",@progbits", lf )
                    #write( " .type  __module_kernel_version,@object", lf )
                    #write
                    (
                        " .size  __module_kernel_version,",
                        @length(version) + 15,
                        lf
                    )
                    #write( "__module_kernel_version:", lf )
                    #write( ".string    ""kernel_version=", version, """", lf )
                    #write( "#endasm", lf )
                    #closewrite
                    ?brk := true;

                #endif

            #endif

        #endif

    #endif

#endwhile
#closeread

// Okay, include the header file we just create above:

#include( "version.hhf" )


// Now for the actual device driver code:
```

```
unit hello;
#include( "k.hhf" )
procedure init_module; external;
procedure cleanup_module; external;

static
    helloi  :dword; external;
    helloi  :dword;

procedure init_module; nodisplay; noframe;
begin init_module;

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

end init_module;

procedure cleanup_module; nodisplay; noframe;
begin cleanup_module;

    linux.printk( "<1>Goodbye World\n" );
    ret();

end cleanup_module;

end hello;
```

Program 3.6 Automatically Setting the Version Number Within HLA

The code above is somewhat tricky. Upon finding a matching statement in the C header file, the compile-time program above extracts the version number as a string and then writes a header file containing the Gas directives that embed the version number in the object file.

This compile-time program (the code up to the first #include above) uses HLA's compile-time file I/O facilities (#openwrite, #openread, #write, @read, #closewrite, and #closeread) to dynamically create a new include file while compiling your source file. It uses HLA's compile-time file I/O capabilities to read the kernel's "version.h" header file and search for a #define statement that defines the version number, extracts the appropriate information, and writes the Gas code to the *version.hhf* header file, and the program then includes this file it just created. The *version.hhf* header file written contains the #asm..#endasm sequence given in the previous section; except it fills in the current version of the Linux kernel. Note that if you need to compile your code for a different version than the current kernel version you're running, you will need to modify the #openread statement so that it opens the appropriate C header file containing the version number you're interested in.

This code is so useful, it appears as a macro in the *getversion.hhf* header file. It couldn't go in the *linux.hhf* header file because you need to include the version extraction code before the HLA unit declaration and the *linux.hhf* header file must appear after the start of the unit. Simply including *getversion.hhf* before your unit statement does all the work for you (it does, however, assume that the version.h file uses the format found on my machine and appears in the /usr/include/linux subdirectory).

2.4 Kernel Modules vs. Applications

There are considerable differences between the way kernel modules and Linux applications operate. As you may have noted in the previous section, modules remain resident even after their "main program"

(init_module) returns. This section will alert you to some of the major differences between kernel modules and application programs.

Since you've already seen that the system doesn't remove a module from memory when the init_module procedure returns, it's probably a good idea to start by explaining how the kernel invokes the module after it returns. Every module exports two well-known symbols: init_module and cleanup_module[7]. Running the *insmod* utility loads your module and causes the Linux kernel to call the init_module procedure; conversely, running the *rmmod* utility tells the Linux kernel to run the cleanup_module procedure and then remove the module from memory (assuming the reference count decrements to zero). Beyond this, there are some standard entry point address that the Linux kernel uses to communicate with your driver, *but all of these other entry points are optional*. For example, the "hello world" driver appearing earlier doesn't implement *any* of the optional entry points. For real-world device drivers, you will need to implement some of these optional entry points; your driver has to tell the kernel about the functionality it supports. As you see in a little bit, it is the responsibility of the init_module procedure to tell Linux which entry points the driver provides. It is the cleanup_module's responsibility to inform the kernel that these entry points are no longer valid prior to Linux removing the driver from memory.

Another difference, also noted earlier, is that module authors must be careful about calling library routines. Those that directly or indirectly make Linux system calls are an absolute no-no in a Linux module. Note that some seemingly innocuous HLA Standard Library functions may still invoke Linux if an exception occurs (e.g., one would think that an str.cpy call would be safe since it basically copies data from one memory location to another; however, if there is a string bounds vioation, str.cpy will raise an exception). Therefore, it's a good idea to avoid using calls to HLA Standard Library functions unless you really know what you are doing. If you want to use some HLA Standard Library code in your modules, you should probably copy the library procedure's source code directly into your project (this would also allow you to remove raise statements and calls to Linux from the procedures, thus making them safe for use in a module).

Note that the linux namespace is defined within the *linux.hhf* header file – the same file that application authors include to access Linux features. This was done to avoid having duplicate sets of definitions in multiple header files (one set for application developers, one set for module developers). To prevent inadvertent system calls from modules (and inadvertent kernel calls from applications), the *linux.hhf* header file uses conditional assembly (the #if statement) to selectively remove certain sets of symbols from the header file. The *linux.hhf* header file checks to see if the symbol __kernel__ is defined prior to the inclusion of the header file. If so, then certain kernel symbols are made available to the including source file. Conversely, if __kernel__ is not defined prior to *linux.hhf* file inclusion, then the kernel symbols are left undefined. To activate the kernel symbols, simply insert a statement like the following before you include the *linux.hhf* header file:

```
?__kernel__ := true; // Type doesn't matter, but boolean is the convention.
```

Linux modules written in assembly language should not include *stdlib.hhf* or any of the other Standard Library header files other than *linux.hhf*. If you feel it's truely safe to call a particular Standard Library routine, copy the external declaration from one of the HLA Standard Library header files into your own header file or into your module's source file. This will help prevent an accidental call to a Standard Library function.

If you intend to run your driver on an SMP machine (symmetrical multiprocessor), then you will need to create separate drivers for SMP machines. As this is being written, the *linux.hhf* header file and support code doesn't provide complete support for SMP modules. However, at some point all the pieces will be in place for SMP support. To compile your module for SMP machines, you must include the following statement prior to the inclusion of the *linux.hhf* header file:

```
?__smp__ := true; // Type doesn't matter, but boolean is the convention.
```

Conversely, when compiling for uniprocessor systems, be sure this symbol is not defined prior to including the linux.hhf header file. As this is being written, SMP support in *linux.hhf* is incomplete. Therefore, you should never define the __smp__ symbol. When SMP support is reliable, the *linux.hhf* header file will con-

7. Newer Linux kernels allow you to specify a different name for these procedures; this document, however, will assume that you're using these names.

tain comments to this effect. So check this file for the appropriate comments if you need SMP support. If the comments are not present, then *linux.hhf* won't support the creation of SMP modules unless you are willing to make the appropriate changes yourself.

In general, you should never use the raise or try..endtry statements in an Linux device driver module. These statements assume that an HLA main program has properly set up exception handling within your code. Since modules are written as units, not programs and have no (HLA aware) main program, the code that sets up exception handling has not executed. It is possible to support limited exceptions in a module (user exceptions only, none that depend upon Linux signals), but this is probably more work than it's worth in the kernel environment. If you want to use asserts within your module, do not use the assert macro found in the HLA Standard Library exceptions module. The *kernel.hhf* header file provides a special linux.kassert macro for this purpose.

2.5 Kernel Stack Space and The Current Process

Many of the calls that the system makes to your driver will be made from some application program. Sometimes your driver may need some information about the process that made the system call that invoked your driver. This is done by accessing the current task object whose type is linux.task_struct. Since it is possible for two or more processes to be executing simultaneously on multiprocessor systems, Linux cannot keep this data in a global variable. Therefore, it hides this data structure in the kernel stack area reserved for each process. To gain access to this data, you need to understand about kernel stacks and user stacks.

Whenever program execution switches from user (application) mode to kernel mode, the x86 processor automatically switches to an auxillary stack before pushing any return addresses or other data. This is one of the principal reasons that Linux system calls pass their parameters in registers rather than on the stack – because if they were pushed on the stack they'd be on the user stack, not the kernel stack that Linux operates from. Indeed, if the application examines the stack data below ESP before and after a system call, it won't see any difference in the data below the location where ESP points (this is necessary for security as well as other reasons). Linux, however, does not use a single kernel stack – every process gets its own kernel stack. This is necessary for several reasons, but a good example of why this is necessary is because multiple processes could be running in the kernel concurrently on multiprocessor systems, and they will each need their own stack. Although in theory, a stack data structure is unbounded, the Linux kernel has to allocate real memory for this structure and memory the kernel allocates is scarce. Therefore, the kernel only allocates eight kilobytes of kernel stack space for each process. The Linux kernel uses far less than 8K to handle any given system call, yet 4K may not be sufficient (the kernel allocates memory in page-sized, 4K, chunks, so the allocation amout was going to be 4K or 8K), hence the choice of 8K. This places a very important liimitation on device drivers you write: device driver modules operate from the kernel stack, so there is a very limited amout of stack space available. So be careful about allocating large automatic variables or writing heavily recursive procedures in module code; doing so may overflow the kernel stack and crash the system.

Don't forget, your module is not the only user of the kernel stack. When a process makes a system call that ultimately invokes your driver, Linux' system call facilities have pushed important information onto the stack before calling your code. Also, as briefly noted above (and explained below), Linux maintains the task_struct information in part of the 8K kernel stack region. Therefore, your module shouldn't count on being able to use more than about three kilobytes of stack space during any call to your module. If you need more space than this, you will need to call the kmalloc function (explained in a later chapter) to dynamically allocate such storage.

The Linux kernel allocates each process' 8K stack space on an 8K boundary. Since the x86 hardware stack grows downward, Linux' designers placed the task_struct information at the beginning of this 8K block (that is, at the bottom of the stack area). This allows the stack to grow as large as possible before it collids with the task_struct object and also allows kernel code to easily determine the address of the task_struct object. The *linux.hhf* header file even includes a macro, linux.get_current, that computes the address of the task_struct object for you. This macro emits the following two instructions:

```
kernel.get_current;
```

-is equivalent to-

```
mov( esp, eax );
and( !8192, eax );  // round down to previous 8K boundary.
```

This code leaves a pointer to the current process' task structure (the current pointer) in EAX. Note that the execution of this code sequence is only reasonable while operating in kernel mode since that's the only time ESP is pointing at the kernel stack (and that's the only time you can assume that ESP's value, rounded down to the previous 8K boundary, is the address of the current process' task information structure).

In general, you should use the kernel.get_current macro rather than emitting these two instructions yourself. However, since this is assembly language, you can always do whatever the situation requires (e.g., you might want the current pointer in a register other than EAX). The nice thing about the macro invocation is that if the Linux designers ever decide to increase the kernel stack space to 16K (the next reasonable amount), then one change to the kernel.get_current macro and a recompile of y our modules is all that's necessary to handle the change. On the other hand, if you've manually computed the task_struct address with discrete instructions, you'll have to find each and every one of those instruction sequences and manually change them.

We revisit the use of the current pointer a little later in this text. Until then, just keep in mind that you've got a very limited amount of stack space to play within in your module before you overwrite the task_struct information at the bottom of the stack area (with disasterous consequences). Until then, here's a little trick you can play to display the name of the current process that has called your module:

```
linux.get_current;
linux.printk
(
    "<1>The process is \"%s\" (pid %i)\n",
    (type linux.task_struct [eax]).comm, //process name.
    (type linux.task_struct [eax]).pid   //process id
);
```

2.6 Compiling and Loading

This rest of this chapter is devoted to creating a skeleton module. That is a module that compiles, loads, runs, and can be removed, but offers nothing more than trivial functionality. The purpose of this module is to provide a "shell" or "template" of a device driver that you can use to construct real device drivers. The name of this driver is *skull* which stands for *Simple Kernel Utility for Loading Localities* (hey, don't blame me, I didn't make this name up! It comes from LDD2).

2.6.1 A Make File for SKULL

Since most modules will consist of multiple source files, the best place to start is with the creation of a makefile for the skull project. The skull project consists of two modules that compile the *skull_init.hla* and *skull_cleanup.hla* source files (containing the init_module and cleanup_module procedures, respectively). A real device driver may very well have additional source modules; however, skull offers no functionality requiring additional modules, hence their absence here. This text assumes that you are somewhat familiar with makefiles and know how to add additional dependencies and commands to a makefile.

Before presenting a simple makefile, it's probably worthwhile to discuss how we'll combine the multiple object files our (multiple) source files produce into a single object file (required by the *insmod* utility). The *ld* utility supports a special command line option, "-r", that tells the linker to produce an partial linking (that is, it produces a single object file by linking together several object files and it doesn't complain if the resulting object file still has some undefined references). The "-r" option stands for relocatable, by the way, since the resulting object file is still in relocatable format (which *insmod* requires). The following is a typical invocation of *ld* using the "-r" option:

```
    ld -r skull_init.o skull_cleanup.o -o skull.o
```

The "-o skull.o" command line option provides the output object file name (*skull.o* in this case).

The following is a simple makefile that will build the skull project:

```
all: skull.o

skull.o: skull_init.o skull_cleanup.o
    ld -r skull_init.o skull_cleanup.o -o skull.o

skull_init.o: skull_init.hla skull.hhf
    hla -c -d__kernel__ skull_init.hla

skull_cleanup.o: skull_cleanup.hla skull.hhf
    hla -c -d__kernel__ skull_cleanup.hla

clean:
    rm -f *.inc
    rm -f *.asm
    rm -f *.o
    rm -f *~
    rm -f core
```

The "-d__kernel__" HLA command line options define the __kernel__ symbol prior to compilation just in case you forget to define __kernel__ prior to including *linux.hhf*. Since both *skull_init.hla* and *skull_cleanup.hla* define this symbol, this command line option has no effect on the compilation; but defining __kernel__ in the makefile is probably a good idea in general.

Of course, more advanced *make* users will complain that the file could be made much shorter by defining some rules and *make* variables. I encourage advanced *make* users to modify this makefile (and other makefiles appearing in this text) to their heart's content. Yes, it's possible to create a single universal makefile that will successfully compile every program in this text. However, my experience suggests that while most people know the basic syntax of makefiles, a much smaller percentage know *make's* more advanced features. Since I generally value clarity over brevity, I'll usually stick to simple makefiles that are easy enough for everyone to understand rather than seeing how short I can make them. For those who are interested in a small taste of how powerful *make* can be, I'd suggest taking a look at the makefile for the linux module in the HLA Standard Library or, better yet, pick up a copy of O'Reilley's text on the subject.

2.7 Version Dependency and Installation Issues

This subject is independent of implementation language. Please see LDD2 for more details. Later in this chapter I will have more to say about version dependency and include files as they pertain to assembly language device drivers.

2.8 Platform Dependency

Since we're writing our drivers in x86 assembly language, the whole issue of platform dependency is a moot point. Our code will be platform specific to the x86 processor. If you're interested in writing platform independent code, assembly language isn't the right choice.

2.9 The Kernel Symbol Table

The Linux kernel maintains a table of kernel symbol names and their corresponding addresses. Whenever you attach a module to the the system, Linux adds the symbols your module exports to the kernel's symbol table. You can display the current symbols in the symbol table by dumping the contents of the */proc/ksyms* file. Each line in this file corresponds to one symbol in the kernel symbol table. The first value on each line is the hexadecimal address of the object in (kernel) memory, the second item on each line is the actual symbol. LDD2 has a lot of interesting things to say about the kernel symbol table. You'll definitely want to take a look at what it has to say. In this section, however, we'll concentrate on how you can export symbols from your module so that they're placed in the global kernel symbol table.

To export a symbol from a module is pretty simple: just make that symbol external and the kernel will automatically add the symbol to it's symbol table. The only problem with this approach is that you will need to use the HLA external directive to communicate symbols between object files in your modules. Unfortunately, the use of this directive will make all external symbols a part of the Linux symbol table. This can create some grave problems if the symbol names you choose conflct with existing kernel symbol names (i.e., namespace pollution occurs). There are two solutions to this problem: use (external) names that are guaranteed to be unique, or export only a subset of your module's symbols.

The first solution is a bit of a kludge, but is probably less error-prone than the second version (which is why some people favor it). The usual solution module writers use to guarantee that all external names are unique is to prepend some common name to the external identifiers; typically this is the module name. For example, if you have an external symbol "myProc" which is really used just to communicate information between two separate object files of your module, you could change the name to something like skull_myProc to guarantee uniqueness in the global kernel symbol table (assuming, of course, that your module's name is relatively unique). The symbol still winds up in the kernel symbol table, but its prefix helps prevent namespace pollution problems. Despite the fact that this is an obvious kludge, many kernel developers prefer this approach because having those symbols in the global symbol table is sometimes useful when debugging kernel modules.

The second solution is to explicitly specify those symbols that you want exported to the kernel's global symbol table. This is accomplished with the two macros linux.export_proc and linux.export_var. These two macros use the following invocation syntax:

```
linux.export_proc( procedure_name );
linux.export_var( static_variable_name );
```

The *procedure_name* argument must be the name of a procedure or statement label (generally, specifying a statement label here is not a good idea, but it's certainly possible if you know what you're doing). The *static_variable_name* argument must be a variable you declare in a static, storage, or readonly section of your program. Note that automatic objects (var variables and parameters) are not legal here. These macros do not check the types or classes of their arguments. If you specify an improper argument, HLA may happily accept it and you will probably get an error when Gas attempts to assemble the code HLA emits (since these macros use the #emit compile-time statement to directly emit Gas code to the assembly output file). So take care when you use these macros[8]. Also note that these macros emit code to a data section they define. Therefore, *never* invoke this macros inside a procedure as this will mess up HLA's code generation. Generally, you should be all your exports either near the top or near the end of your source file.

2.10 Initialization and Shutdown

As briefly mentioned earlier, the init_module procedure is responsible for telling the kernel about any entry points beyond init_module and cleanup_module that the driver optionally supports. The process of telling the kernel which optional entry points the module supports, and specifying the addresses of these entry

8. Actually, HLA's compile-time language provides the ability to verify the correctness of the arguments before emitting the code based on them. Someday, I'll probably get around to extending these macros to check their arguments to make sure they're proper.

points, is called *registration*. The init_module procedure registers the optional entry points and the cleanup_module procedure *unregisters* those entry points (so the kernel knows not to call them after module removal). We'll discuss these optional entry points in great detail later in this text. The important thing to note right now is that a module must clean up after itself and unregister everything before the module kernel removes it from memory.

Please see LDD2 for a further discussion of module initialization and shutdown.

2.11 Error Handling in init_module

If an error occurs during module initialization, the init_module procedure must explicitly unregister any functionality it has already registered prior to the error. This is because Linux doesn't keep track of the facilities your module has registered; it's strictly up to your code to take of this problem.

Probably the most "structured" approach is to use nested HLA begin..exitif..end blocks as follows (the following code assumes that each code block registering some functionality leaves zero in EAX if the registration was successful and a non-zero error code in EAX if there was an error):

```
// code inside init_module procedure:

<< code to register functionality #1 with Linux >>
exitif( eax <> 0 ) init_module;
begin func1;

    << code to register functionality #2 with Linux >>
    exitif( eax <> 0 ) func1;
    begin func2;

        << code to register functionality #3 with Linux >>
        exitif( eax <> 0 ) func2;

        << repeat nested begin..end blocks for each functionality to add >>

            // After the nth exitif statement, put the following:

            mov( 0, eax );
            exit init_module; // return from init_module with success code (0).

    end func2;
    << If we get here, func #3 failed, so clean up func #2 which was
           successful >>

end func1;
<< If we get here, func #2 failed but func #1 was successful, clean up
    by unregistering func #1 >>

end init_module;
```

The important thing to note about this set of nested begin..end blocks is that once failure occurs, the program falls through all remaining unregister operations for all of the previously successful registration operations. This is the "structured" (and, usually, more readable) way to handle error recovery in init_module.

The structured solution falls apart if the module needs to register a large number of facilities with the kernel. The problem is that the indentation of the nested blocks tends to wander off the right hand side of your screen after about eight or ten levels. In this case, the structured solution is probably harder to read than an unstructured solution that simply uses JMP instructions:

```
// code inside init_module procedure:
```

```
<< code to register functionality #1 with Linux >>
exitif( eax <> 0 ) init_module;

<< code to register functionality #2 with Linux >>
jt( eax <> 0 ) func1;

<< code to register functionality #3 with Linux >>
jt( eax <> 0 ) func2;

<< repeat sequence for each functionality to add >>

// After the nth exitif statement, put the following:

mov( 0, eax );
exit init_module; // return from init_module with success code (0).

<< Labels and unregistration code for func3, func4, ..., funcn >>

func2:
<< If we get here, func #3 failed, so clean up func #2 which was successful >>

func1:
<< If we get here, func #2 failed but func #1 was successful, clean
     up by unregistering func #1 >>

end init_module;
```

Note that the jt statements above are completely equivalent to:

```
test( eax, eax );
jnz funcn;
```

Feel free to use this latter form if you perfer "pure" machine instructions (or know that the zero flag already contains the error status so you don't have to test the value in EAX).

Depending on the complexity of your init_module handling code, even the jmp solution make not scale well to a large number of "undo levels". Or perhaps, you've come from a high level language background and all those jmps (gotos) really bother you. Another solution, that scales well to a large number of facility registrations is to keep track of exactly what facilities you register and then pass this information on to cleanup_module to undo what has been done up to that point. The nice thing about this approach is that it is easy to do using tables so that the init_module and cleanup_module procedures simply iterator through a table of entries and act upon those entries. Another solution is to use a boolean (or bitmap) array to keep track of facilities that need to be undone by cleanup_module. If the init_module completes successfully, it sets all of the boolean array elements (or bits) to true and the cleanup_module unregisters everything; if the init_module procedure aborts early (and calls cleanup_module directly to do the clean up), then cleanup_module only undoes the resource allocation specified by the entries in the map. The examples in this text will generally use this approach since it's a bit more aesthetic and is usually more effecient (in terms of lines of code) as well.

2.12 The Usage Count

The kernel maintains a count of the number of processes that use a module in order to determine whether it can safely remove a module (when you use *rmmod*). In modern kernels, the kernel automatically maintains this counter. Still, there are times when you might need to manipulate this counter directly. The linux.hhf header file contains several macros that provide access to this counter variable. These macros are:

```
linux.mod_inc_use_count;  // Adds one to the count for the current module.
linux.mod_dec_use_count;  // Subtracts one from the current module count.
linux.mod_in_use;         // Returns true in EAX if count is non-zero.
```

Note that the module count variable is the unsigned integer variable that you must declare with your module that is the module name with an "i" suffix (e.g., the "khwi" variable in the hello world example given early). However, you must avoid the temptation to access this variable directly. The macros above don't simply expand to an increment, decrement, or test of this variable. For example, the `linux.mod_inc_use_count` macro expands to the following:

```
push( eax );
mov( linux.__this_module.uc.usecount.counter, eax );
lock.inc( (type dword [eax]) );
or
(
    linux.mod_visited | linux.mod_used_once,
    linux.__this_module.flags
);
pop( eax );
```

Now it turns out that `linux.__this_module.uc.usecount.counter` contains a pointer to the module counter variable you declare, but as you can see, there's more to this operation than simply incrementing the variable. If you're willing to take chances with future kernel compatibility, you can always use the following code sequence instead of the above:

```
lock.inc( khwi );  // Increment your module's counter variable.
or
(
    linux.mod_visited | linux.mod_used_once,
    linux.__this_module.flags
);
```

While a little bit shorter and faster, this code will fail if the kernel developers decide to change how the module counter variable works (which they did between kernels 2.0 and 2.2 if you don't believe this can happen; see LDD2 for more details).

There are many other issues surrounding the use of the module's usage counter. See LDD2 for more details on this object.

2.13 Resource Allocation (I/O Ports and Memory)

Device drivers generally need to use system resources such as I/O ports or memory-mapped I/O locations. Unlike the wild and wooly days of DOS, a Linux device driver cannot simply start poking around at some I/O port address. Some other device may have exclusive access to that port and the actions of your driver could break the system. Therefore, the Linux kernel allocates ports and other system resources to device drivers and rejects requests for those resources if they are already in use. For a general discussion of I/O Ports and I/O memory, see LDD2; this document assumes that the reader (an assembly language programmer) is confortable with these concepts. What an assembly language programmer may not know is that Linux manages I/O port and I/O memory locations just like any other resource. The device driver writer must request an I/O port or I/O memory location just like any other resource. The writer of a parallel port device driver, for example, simply cannot assume that the ports at addresses $378..$37A are available for use, even if the device driver is begin written specifically for the parallel port at that address. After all, it could be the case that another device driver also manipulates the parallel port at that address range. Therefore, a well-behaved device driver must request ports prior to using them and return those resources to the system upon termination.

The Linux kernel provides three functions that manage the allocation and deallocation of port-related addresses: linux.check_region, linux.request_region, and linux.release_region. Their function prototypes are as follows[9]:

```
check_region( start:dword; len:dword )
request_region( start:dword; len:dword; theName:string )
release_region( start:dword; len:dword)
```

The check_region function is no longer needed as of Linux v2.4, though this document will still discuss its use. See LDD2 for more details. The check_region function checks a range of ports ($0000..$FFFF port addresses to see if they are *all* available. This function returns a negative value (an errno.XXXX value like errno.ebusy or errno.einval) if the ports are not all available.

The request_region function, as of kernel v2.4, returns NULL if it cannot allocate the specified range of port addresses. It returns a non-NULL value otherwise (your code should ignore the actual return value if it is non-zero). Note that as of kernel v2.4, request_region handles both the task of checking for port availability and allocating the port; in earlier kernels, request_region did not return a value so you had to use check_region to see if the range was available. This was changed in v2.4 to help avoid some race conditions. In Linux v2.4 and later you shouldn't really call check_region; just use request_region to allocate ports and test to see if they are available. See LDD2 for more details. Note that the string parameter for request_region should me the name of your module that you request with the kernel (e,.g., "skull" in the example we are developing in this chapter).

Here's a typical instruction sequence that demonstrates how you could use check_region and request_region to request a set of I/O ports:

```
procedure skull_detect( port:dword; range:dword );
begin skull_detect;

    linux.check_region( port, range );
    exitif( (type int32 eax) < 0 );   // return if error.
    exitif( skull_probe_hw( port, range ) <> 0 ) skull_detect
    linux.request_region( port, range ); // Can't fail, see check_region call.
    xor( eax, eax );  // return success.

end skull_detect;
```

This code first checks to see if the requested port region is available. It returns with the appropriate error code if the ports are available (there is, after all, no sense in probing for the hardware if the ports are unavailable.

The skull_probe_hw function, not given here, scans the actual port addresses to determine if the hardware device is present at the specified port addresses. Since this function is hardware dependent, we won't discuss it here other than to say it returns zero in EAX if it finds the device, it returns errno.enodev (device not found) if it can't determine that the device is present.

Note that since linux.check_region returns successfully if we execute linux.request_region, we know that the request_region call will be successful[10]. Hence the function simply sets EAX to zero and returns once it gets back from linux.request_region.

The linux.release_region function releases the port resources so other drivers can use the ports. Obviously, any device driver should free up the ports it uses when it's done with those ports. The skull code does this in the skull_release procedure:

```
procedure skull_release( port:dword; range:dword );
begin skull_release;

    linux.release_region( port, range );

end skull_release;
```

9. These are actually macros which hide the fact that these functions use the C calling convention and require a little bit of stack clean-up on return as well as passing some extra parameters ; see the linux.hhf header file for details.

10. Note that the kernel will only install one module at a time, so you don't have to worry about another processor sneaking in and grabbing the ports; paranoid coders, however, can feel free to check kernel.request_region's return result. Just keep in mind that this function did not return a value prior to Linux v2.4.

Note that if you need to request a non-contiguous range of ports in your code and you allocate them as you check them (that is, you don't call linux.check_region), don't forget that if an error ever occurs you've got to release those resources you've successfully allocated. The recovery process is similar to that as we discussed for facilities, earlier.

In addition to port address, the Linux kernel also allocates I/O memory addresses (that is, the memory-mapped regions associated with peripherals) in a manner quite similar to I/O port addresses. To obtain and relinquish a range of I/O memory addresses, you use the following kernel functions[11]:

```
check_mem_region( start:dword; len:dword )
request_mem_region( start:dword; len:dword; theName:string )
release_mem_region( start:dword; len:dword )
```

You use these functions in a manner identical to the I/O functions given earlier. See LDD2 for more details.

For an interesting discussion of resource allocation in Linux v2.4, see LDD2.

2.14 Automatic and Manual Configuration

In the best of all worlds, we could query the hardware and it would tell use whether a device was present in the system, what port addresses, memory addresses, interrupts, and other system resources it uses (or provides), and our device driver could configure itself based on these values. Many system busses (e.g., PCI) come close to this ideal. Unfortunately, many "legacy" devices still use the ISA bus which doesn't provide this capability. In some cases, the software can probe various I/O port addresses looking for certain values and determine whether some special piece of hardware is present. The problem with this approach is that arbitrarily poking around at hardware address may cause some devices (besides the one you're looking for) to malfunction. Therefore, automatic probing isn't always a safe way to determine whether a piece of hardware is present. Because of this problem, sometimes you'll need to provide certain configuration information via parameters the user must pass in to the module during initialization. Obviously, we'd like to avoid requiring users to supply such information (often, they are not technically qualified to do so), but some times there is no other choice.

The *insmod* utility provides the ability to pass parameters to the init_module procedures on the command line. For example, you could pass an integer and a string parameter via *insmod* using a command line like the following:

```
/sbin/insmod ./skull.o intVar=12345  strVar="Hello World"
```

In order for *insmod* to change the value of variables within your modules, you must tell *insmod* about those objects. This is done with the linux.module_parm macro found in the *linux.hhf* header file. For example, to handle the two parameters above, you could use code like the following:

```
static
    intVar     :dword := 0;
    strVar     :pointer to char := 0;  // Initialize with NULL.

kernel.module_parm( intVar, "i" )
kernel.module_parm( strVar, "s" )
```

Note that *insmod* assumes that string objects are C-style strings (that is, a zero-terminated array of characters), not HLA strings. The *insmod* utility will allocate storage for the string data and store a pointer to the actual string data in the variable the module_parm macro specifies; hence the use of the dword type for strVar in the example above. Of course, HLA strings are pointer objects, so we could have declared strVar as an HLA string object, but this could create some confusion, hence the use of the "pointer to char" type above. Feel free to use the string type in your code as long as you don't forget that these objects point at zero-terminated strings (that aren't completely compatible with HLA strings). Typically the confusion between HLA

11. Like the I/O port functions, these are actually macros. See *linux.hhf* for more details.

strings and C strings won't be a problem in modules because you can't use the HLA Standard Library string functions anyway (unless you modify them to remove the code that raise exceptions).

The linux.module_parm macro requires two arguments. The first must be the name of an HLA static or storage variable. The second parameter must be a string constant that specifies the type of the object. Module parameters currently support five type strings: "b" for a byte, "h" for a word, "i" for an int32 object, "l" for a long object (also int32), and "s" for a string. See LDD2 for more information about this feature.

Note that the linux.module_parm macro emits data to a special ELF segment; like the earlier 'export' macros, you should never place these macro invocations inside a procedure or in the middle of a particular declaration section, doing so will cause the code to malfunction[12]. It's best to put these macro invocations wherever a procedure declaration would normally go.

You should declare all module parameter variables in a static declaration section and give them an initial default value. The *insmod* utility only modifies a parameter variable's value if a command line parameter is present for the particular variable.

Another useful macro in the linux.hhf header file related to parameter is the linux.module_parm_desc macro. You invoke this macro as follows:

```
linux.module_parm_desc( parmName, "comment describing parameter" )
```

This invocation buries the specified string in the object code. Certain administrative tools can display this information (e.g., objdump) so system administrators can figure out what parameters are possible for a given module. Example:

```
static
    basePort := $300;

    linux.module_parm( basePort, "i" )
    linux.module_parm_desc( basePort, "The base I/O port (default 0x300)" )
```

(User-friendliness note: note the use of the 'C' notation for hexadecimal values rather than HLA's notation. Most system administrators know about C's hexadecimal notation, but they might not understand HLA's notation.)

2.15 The SKULL Module

This section presents the complete SKULL module source code and describes how it works, the features it provides, and how to use it as a starting point for your own device drivers.

The following listing is the makefile for the skull project. Since we've already discussed its content earlier, this code is offered without further explanation:

```
all: skull.o

skull.o: skull_init.o skull_cleanup.o
    ld -r skull_init.o skull_cleanup.o -o skull.o


skull_init.o: skull_init.hla skull.hhf
    hla -c -d__kernel__ skull_init.hla


skull_cleanup.o: skull_cleanup.hla skull.hhf
    hla -c -d__kernel__ skull_cleanup.hla
```

12. HLA actually provides the facility in the compile-time language to determine if you invoke a macro within a procedure's body. Someday, when I get the time, I may clean up this macro and issue an error if you invoke it in the wrong place.

```
clean:
    rm -f *.inc
    rm -f *.asm
    rm -f *.o
    rm -f *~
    rm -f core
```

Program 3.7 Skull Project Makefile

The *getversion.hhf* header inserts the appropriate kernel version number information into the object file. Since the operation of this compile-time code was discussed earlier, there is no need to repeat that discussion here. One point worth noting is that a device driver project should only include this header file in one of the source files, otherwise there will be multiply-defined symbols in the object file. The *skull* project includes this header file in the *skull_init.hla* source file. Here's the source code for this header file:

```
#openread( "/usr/include/linux/version.h" )
?brk := false;
?r1 :string;
?r2 :string;
?r3 :string;
?r4 :string;
?UTS_RELEASE :string;
?lf := #$a;

#while( !brk & @read( s ) )

   #if( @matchstr( s, "#define", r1 ))

      #if( @oneOrMoreWS( r1, r2 ))

         #if( @matchstr( r2, "UTS_RELEASE """, r3 ))

            #if( @oneOrMoreCset( r3, {'0'..'9', '.', '-'}, r4, UTS_RELEASE ))

               #openwrite( "version.hhf" )
               #write( "#asm", lf )
               #write( " .section   .modinfo,""a"",@progbits", lf )
               #write( " .type    __module_kernel_version,@object", lf )
               #write
               (
                  " .size   __module_kernel_version,",
                  @length(UTS_RELEASE)+16,
                  lf
               )
               #write( "__module_kernel_version:", lf )
               #write( ".string  ""kernel_version=", UTS_RELEASE, """", lf )
               #write( "#endasm", lf )
               #closewrite
               ?brk := true;

            #endif

         #endif

      #endif
```

```
   #endif

#endwhile
#closeread

// Create the LINUX_VERSION_CODE value:

?r1 := UTS_RELEASE;
?lvc2:dword;
?lvc1:dword;
?lvc0:dword;
?r3 := @matchIntConst( r1, r1, lvc2 ) & @oneChar( r1, '.', r1 );
?r3 := @matchIntConst( r1, r1, lvc1 ) & @oneChar( r1, '.', r1 );
?r3 := @matchIntConst( r1, r1, lvc0 );
?LINUX_VERSION_CODE :dword := (lvc2 << 16) + (lvc1 << 8) + lvc0;


#include( "version.hhf" )
```

Program 3.8 getversion.hhf

The *skull* project uses the *skull.hhf* header file to define all the external symbols the various source modules share with one another and the kernel. Note the traditional use of conditional compilation to prevent problems with multiple inclusions (even though no source file ever attempts to include this more than once and this is even doubly redundant since the source files use #includeonce to include this file).

The skull_fn1, skull_fn2, and skull_variable declarations exist solely to demonstrate sharing symbols between modules and the kernel. They provide no functionality to the module and you may remove them without any effect on *skull's* operation. The init_module, cleanup_module, skull_port_base, and skulli declarations define the symbols the module uses to communicate with the kernel (the two source modules in this particular example don't actually communicate information between one another, just between themselves and the kernel). The skulli variable, of course, is the module counter variable that an assembly language programmer must provide for the kernel.

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

   // From skull_init.hla:

   procedure init_module; external;
   procedure skull_fn1; external;
   procedure skull_fn2; external;

   // From skull_cleanup.hla:

   procedure cleanup_module; external;


static

   // From skull_init.hla:

   skulli         :dword; external;
   skull_variable :dword; external;
   skull_port_base   :dword; external;
```

```
#endif // skull_hhf defined.
```

Program 3.9 skull.hhf Local Header File

The *skull_init.hla* module contains the source file holding the init_module function and several support procedures. The following paragraphs describe the code that follows them; you may want to jump ahead and read along in the code while you're reading each of these paragraphs so that they make the most sense.

This particular module demonstrates how to probe the ISA bus at various port addresses and at various I/O memory addresses. Since this driver was not designed for a specific device, but strictly as a demo, the initialization code only demonstrates how one might write the code to probe for the device; this code doesn't actually look for a real world device.

The skull_fn1 and skull_fn2 procedures and the skull_variable objects exist in this source simply to demonstrate how to export names to the kernel using the linux.export_proc and linux.export_var macros. They provide no other functionality nor does any code in this module (or the kernel) otherwise reference these labels.

This code demonstrates how to inform *insmod* about module parameters using the linux.module_parm and linux.module_parm_desc macros. This particular module allows the *insmod* user to specify the base port address of the device on the command line for the skull_port_base variable. If no such command line parameter exists, this program attempts to probe for the hardware to determine it's base address.

The skull_register function is responsible for registering additional functionality that this module provides. The *skull* module doesn't provide any additional facilities beyond init_module and clean_up module, so this particular procedure is currently empty. If you use this source code as a skeleton for your own modules, you'll want to add the code that registers kernel facilities to this procedure.

The skull_probe_hw function is passed a base port address and a value that specifies the range of port addresses. This function is responsible for determining if there is an instance of your hardware device at the port address specified. This function should return zero it if determines that the desired hardware is present at the given port address; it should return some non-zero value (e.g., -1) if it cannot find the hardware at the specified address. Since this skull project isn't dealing with real hardware, the skull_probe_hw function always returns -1 to indicate "hardware not found."

The skull_init_board procedure is responsible for actually initializing the hardware once this code determines that there is hardware present in the system. This function should return zero if it can successfully initialize the hardware, it should return a non-zero value if there is a problem initializing the hardware (note, however, that this particular *skull* implementation doesn't currently check the return value; you should change this if it is possible for your hardware to have an error during initialization). Since this procedure is hardware dependent, and this version of *skull* doesn't deal with actual hardware, the implementation of skull_init_board in this source file simply returns zero to indicate successful initialization.

The skull_detect procedure checks to see if a range of port addresses (specified by the parameters) is already in use. If so, skull_detect returns failure. If not, then this function calls the skull_probe_hw to see if the hardware is actually present. If no, skull_detect returns failure. If so, then skull_detect reserves those addresses and returns success (zero) in EAX. Note that if skull_detect returns success and later on the initialization code fails, the code must return the port range resources to the system. Since skull_probe_hw always returns failure in this particular example, skull_detect never allocates any port addresses so there is no need for this code to clean up after itself. In a real driver, however, you must be cogniscent of this issue.

The skull_find_hw procedure is the actual procedure that scans through the port addresses searching for actual hardware. It iterates across the legal set of port addresses calling the above functions to see if there is hardware present in the system. Note that if multiple instances of the hardware are present, then skull will reserve all their port addresses; you'll need to modify the code (by adding a break) if you want the detection to stop on the first device it finds. Note that this procedure only probes through the range of addresses $280..$300 if the user has not specified a port base address on the *insmod* command line. If the user has not specified a command-line parameter with the base address, then the skull_port_base variable will contain

zero and the repeat..until loop in this procedure will iterate between the addresses $280 and $300. However, if skull_port_base contains a non-zero value, then the loop will execute only once and will test the port(s) at the base address specified by the skull_port_base command line parameter.

The last procedure in this source module is the init_module procedure. This procedure, of course, is the module's "main program" – the procedure that *insmod* calls when it first loads your program into memory. Normally, this procedure would do two things: (1) call skull_find_hw to detect the hardware and reserve the I/O ports associated with it, and (2) call skull_register to register any additional facilities this module provides. This particular version of init_module does these two tasks, but it also demonstrates how to scan through ISA I/O memory by calling kernel.ioremap (that uses the PCI<->ISA bus bridge to map ISA memory addresses to a virtual memory address) and then scanning ISA memory locations from $A_0000 to $F_FFFF (where peripheral memory lies in the original PC memory map). This code writes an explanation of it's findings to the */var/log/messages* file via linux.printk. This code is purely gratuitous and exists for demonstration purposes only. You should remove this code prior to use *skull* as a skeleton for your own device drivers.

```
/*
 * skull_init.hla -- sample typeless 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 skull;

// "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" )

// Skull-specific declarations:

#includeonce( "skull.hhf" )

// Set up some global HLA procedure options:

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

```
const

   // Port ranges: the device may reside between $280 and $300
   // steps of $10.  It uses $10 ports.

   skull_port_floor_c   := $280;
   skull_port_ceil_c := $300;
   skull_port_range_c   := $10;

   // Here is the region we look at:

   isa_region_begin_c   := $0A_0000;
   isa_region_end_c  := $10_0000;
   step_c            := 2048;



static

   skulli         :dword;        // This module's reference counter.

   skull_port_base   :dword := 0;   // 0 forces autodetection.
   skull_variable :dword;        // Dummy variable to test exporting.


// Here are some dummy procedures we can use to test exporting
// symbols:

procedure skull_fn1; begin skull_fn1; end skull_fn1;
procedure skull_fn2; begin skull_fn2; end skull_fn2;


linux.export_proc( skull_fn1 );
linux.export_proc( skull_fn2 );
linux.export_var( skull_variable );
linux.export_var( skull_port_base );
linux.module_parm( skull_port_base, "i" );
linux.module_parm_desc( skull_port_base, "Base I/O port for skull" );


// Register and export names & functionality.

procedure skull_register; returns( "eax" ); @noframe;
begin skull_register;

   // << insert code to register symbols here >>

   mov( 0, eax );
   ret();

end skull_register;


// Probe for the hardware devices:

procedure skull_probe_hw( port:dword; range:dword );  returns( "eax" );
begin skull_probe_hw;
```

```
   // <<Insert code to probe for the hardware here>>

   // This function returns zero to indicate success (it's found
   // the board).  We're returning failure here because there
   // isn't any real hardware associated with this "driver" (yet).

   mov( -1, eax );      // HW not found.

end skull_probe_hw;



procedure skull_init_board( port:dword ); returns( "eax" );
begin skull_init_board;

   // <<insert code to initialize the hardware device here >>

   mov( 0, eax ); // Indicates we've successfully initialized the hw.

end skull_init_board;



// Detect the device and see if its ports are free:

procedure skull_detect( port:dword; range:dword ); returns( "eax" );
var
   err:dword;
begin skull_detect;

   // check_region returns a negative error code if
   // the specified port region is in use:

   linux.check_region( port, range );
   exitif( (type int32 eax) < 0 ) skull_detect;

   // Note: skull_probe_hw returns non-zero (-1) if it
   // can't find the hardware.  We'll just return this
   // value back to skull_detect's caller if it's non-zero
   // as the error code.

   exitif( skull_probe_hw( port, range ) <> 0 ) skull_detect;

   // At this point, request_region will succeed.  So upon
   // return from request_region, return zero to skull_detect's
   // caller to indicate success.

   linux.request_region( port, range, "skull" );
   mov( 0, eax );

end skull_detect;


// skull_find_hw-
//
// This procedure scans port addresses from
// skull_port_floor to skull_port_ceil in increments
// of skull_port_range searching for the actual
// hardware.
```

```
procedure skull_find_hw; returns( "eax" );
var
   boardCnt:   dword;

begin skull_find_hw;

   push( edx );

   mov( 0, boardCnt );

   // If skull_port_base contains zero, then there was
   // no command line parameter specifying the base address,
   // so start with the floor value and work our way up from there,
   // otherwise start with the base address supplied by the user.

   mov( skull_port_base, edx );
   if( edx = 0 ) then

      mov( skull_port_floor_c, edx );

   endif;

   // If no command line parameter was specified for skull_port_base,
   // the the following loop scans through the address range specified
   // by skull_port_floor..skull_port_ceil.  If there was a command
   // line parameter specified, this loop repeats only once to validate
   // the parameter value.

   repeat

      if( skull_detect( edx, skull_port_range_c ) = 0 ) then

         skull_init_board( edx );
         inc( boardCnt );

      endif;
      add( skull_port_range_c, edx );

      // Note" skull_port_base contains a non-zero value
      // if there was a command line parameter.

   until( skull_port_base != 0 || edx >= skull_port_ceil_c );

   // Return a count of the boards we found as the function
   // result (zero indicates no boards were present).

   mov( boardCnt, eax );
   pop( edx );

end skull_find_hw;


procedure init_module; @noframe;
begin init_module;

   push( ecx );
   push( edx );
   push( ebx );
   push( esi );

   ////////////////////////////////////////////////////////////
```

```
//
// The following code is just for fun.  It demonstrates how to
// scan through the I/O memory space.  You'll probably strip
// this out when writing a "real" device driver.
//
////////////////////////////////////////////////////////////

 /*
  * Print the isa region map, in blocks of 2K bytes.
  * This is not the best code, as it prints too many lines,
  * but it deserves to remain short to be included in the book.
  */

// Use ioremap to get a pointer to the region:

linux.ioremap
(
   isa_region_begin_c,
   isa_region_end_c - isa_region_begin_c
);

// Compute the 32-bit base address of the 1Meg block
// into which Linux maps the ISA memory addresses.
// This is done so that we can get by printing 20-bit
// "ISA-Friendly" addresses down below:

sub( isa_region_begin_c, eax );
mov( eax, ebx );           // Save as base address.

// Check the region in 2K blocks to see if it's already allocated:
// Check out the ISA memory-mapped peripheral range from
// $A_0000 to $F_FFFF:

for
(
   mov( isa_region_begin_c, esi );
   esi < isa_region_end_c;
   add( step_c, esi)
) do


   // Check for an already allocated region:

   if( linux.check_mem_region( esi, step_c ) ) then

      linux.printk( "<1>%lx: Allocated\n", esi );
      continue;

   endif;

   // Read and write the beginning of the region and
   // see what happens.

   pushfd();
   cli();

   mov( [ebx+esi], ch );   // Get old byte.
   mov( ch, cl );
   not( cl );
   mov( cl, [ebx+esi] );   // Write value.
   linux.mb();           // Force write to RAM (mem barrier).
```

```
            mov( [ebx+esi], cl );   // Reread data from RAM.
            mov( ch, [ebx+esi] );   // Restore original value.
            popfd();          // Restore interrupts.

            // The XOR produces $FF for RAM since we inverted the value
            // we wrote back to memory (x xor !x = $FF).  The xor below
            // will produce $00 for ROM since we'll always read back the
            // original value (not what we wrote) when ROM is mapped in.
            // Note, however, that it's also possible for $00 to be
            // returned if there is no device associated with the address.
            // LDD2 provides a better check for ROM than this, but since
            // this code is just a demo, who cares?
            // If we get something other than $00 or $FF back, then
            // either the space is not associated with a device
            // (and we're reading garbage) or there is a device but
            // it returns a different value each time you read it
            // that has no relationship to the written value.

            xor( ch, cl );
            if( cl = $ff ) then

               // We reread our change, so there's RAM present.

               linux.printk( "<1>%lx: RAM\n", esi );
               continue;

            endif;
            if( cl <> 0 ) then

               // Read back random bits:

               linux.printk( "<1>%lx: Empty\n", esi );
               continue;

            endif;

            // See LDD2 for a fancier test for ROM.  Hardly seems
            // worthwhile here since all this code goes away in a
            // real device driver.

            linux.printk( "<1>%lx: Possible ROM or empty\n", esi );

        endfor;

        ////////////////////////////////////////////////////////////
        //
        // The above was just for fun.  Here's the real code you'd
        // normally put in your driver:

        // Find your hardware:

        skull_find_hw();

        // Register your symbol table entries:

        skull_register();

        xor( eax, eax );
        pop( esi );
        pop( ebx );
        pop( edx );
```

```
    pop( ecx );
    ret();

end init_module;


end skull;
```

Program 3.10 skull_init.hla Source Module

The *skull_cleanup.hla* source module contains two procedures that *skull* uses to clean up resources prior to module removal: skull_release and cleanup_module. It is the responsibility of the skull_release procedure to release the ports that were allocated by the *skull_init.hla* module. Currently, the *skull_init.hla* module does not communicate this information to the *skull_cleanup.hla* module. In a real driver you should rectify this by saving the allocated port base address(es) in a global, external, variable so skull_release (or cleanup_module, that calls skull_release) has access to this information. As you can see in the code below, cleanup_module doesn't actually call skull_release, so nothing actually gets freed by this code (which is fine, since the *skull_init.hla* module doesn't actually allocate anything). One point you might consider when turning this into a real module; if your code, like the skull_find_hw procedure in the *skull_init.hla* module can allocate multiple instances of a device in the I/O port range, then make sure that the skull_release procedure frees each instance that skull_find_hw allocates (the code for skull_release below isn't operational, so it's doesn't bother with this important detail).

```
/*
 * skull_cleanup.hla -- sample typeless 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.
 */


unit skull;

// "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" )

// Include file with SKULL-specific declarations:

#includeonce( "skull.hhf" )
```

```
// Set up some global HLA procedure options:

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


procedure skull_release( port:dword; range:dword ); @noframe;
begin skull_release;

   linux.release_region( port, range );

end skull_release;


procedure cleanup_module;
begin cleanup_module;

   /* should put real values here */

   // skull_release( 0, 0 );

end cleanup_module;

end skull;
```

Program 3.11 skull_cleanup.hla Source Module

You can actually build this module, use insmod to load it, and rmmod to remove it from the system. When *insmod* loads this module (*skull.o*), the init_module procedure writes an ISA memory map log to the log file. Then then module remains in a quiescent state until you remove it with the *rmmod* command. For more information on the *skull* module, please see LDD2.

Of course, skull isn't a *real* Linux device driver. It really doesn't provide any support for reading and writing data to a real (or even a pseudo) device. Fear not, however, we'll get around to writing a real driver in the very next chapter.

2.16 Kernel Versions and HLA Header Files

Header files one links with application programs tend to remain static. That is, if you write an application that uses a particular structure defined in a library header file you can generally expect to be able to compile and run that application without any changes on later versions of Linux. When kernel designers make changes to system calls necessitating the modification of such structures, they usually add the new support by creating a new structure and leave the old one alone. This design strategy maintains backwards compatibility with applications while allowing newer applications to take advantage of new system features; the drawback is that the libraries wind up carrying around a lot of baggage to support obsolete system calls and data structures. Kernel developers, however, do not take the same approach with the kernel code; if there is a good reason to change a kernel data structure or interface to some kernel function, the kernel developers will make the change and expect people who use that structure or function to make appropriate modifications to their code before they can integrate it back into the kernel. While this helps keep the kernel free of excess baggage, it plays hell with module developers.

Kernel interface functions and data structures change for a wide variety of reasons. Obviously, many changes occur as kernel developers add functionality to the kernel. Sometimes this involves adding new fields to existing data structures so that new functions can provide additional functionality. Obviously, any module compiled with old header files specifying those data structures will not work in the new kernel since

the data structures are different and the fields may appear at different offsets within the structure. Sometimes kernel developers rearrange fields in a structure[13]; again, this invalidates any old code that assumed the fields appeared at their original offsets in the object.

The problem noted above is the primary reason why modules compiled for one kernel version may not work properly on a different version. For this reason, the Linux kernel enforces strict versioning and the kernel will not load a module whose version does not match the kernel's version number. So when you compile a module, you need to compile that module using the header files for the system on which you wish to run the module. This usually isn't a problem for module developers who work in C since the kernel developers maintain the header files for the kernel and provide a set with each kernel release. However, those same kernel developers (probably) do not maintain the HLA header files concurrent with the C header files. This plays havoc with assembly/HLA developers.

As this document was first being written, the *linux.hhf* header files are based on the Linux v2.4.7-10 header files that came with Red Hat Linux v7.2. If you're using Linux v2.4.7-10 you should be able to use these header files as-is. However, if you're using a different version of Linux, use the kernel symbols in the *linux.hhf* header files at your own peril.

2.16.1 Converting C Header Files to HLA and Updating Header Files

Since the creation of the HLA header files for Linux was not a trivial task, this isn't a process one would want to go through whenever a Linux version change occurs (which, as noted above, seems to happen every time somebody sneezes). Fortunately, the changes between minor versions are relatively minor; but even one incorrect constant definition can cause your device driver to fail. So how can we handle this problem?

The bad news is that you're going to have to get good at translating C header files into HLA header files. This is true even if you decide to stick with Linux v2.4.7-10. The existing HLA/Linux header files are not complete; at some point or another you're going to want to use a constant, data type, or function whose definition doesn't appear in the *linux.hhf* header file. When this happens, you should add the appropriate definition to the header file (and email me the change!) so you can make use of that definition in future projects.

If you are working with a version of Linux other than v2.4.7-10[14], then you should definitely make a quick check of the HLA header files to verify that they're "in-sync" with your kernel's version. The way to do this is to take the Linux v2.4.7-10 kernel/module header files and *diff* them against your kernel version's header files. Where differences exist, you should check the corresponding declarations (if present) in the HLA header files and update them as necessary. If you do upgrade the HLA header files to a different version, I'm sure many Linux developers out there would appreciate it if you'd send me a copy so that I can post those header files on Webster[15].

The best way to get comfortable with translating C header files to HLA is to compare the existing HLA header files with their C counterparts. Although the HLA header files are similar in content to the corresponding C header files (e.g., *fs.hhf* typically contains the definitions found in *fs.h*), there isn't an exact one-to-one correspondance between the HLA header files and the C header files. The bottom line is that the *grep* utility will quickly become your friend when doing this kind of work (I'll assume you know about *grep*, if not, check out the man page for it).

Simple data types are the easiest to convert between C and HLA. The following table provides a quick reference for GCC vs. HLA types (on the x86):

13. They'll do this to speed up the system by placing often-used fields together in the same CPU cache line, for example.

14. Note that this document assume that you are using a Linux v2.4 or later kernel. You're on your own if you need to work with an earlier kernel. See LDD2 for details concerning Linux v2.0 and Linux v2.2.

15. Note that the HLA header files are public domain, so you are not bound by the GPL to release your work. However, it would be really nice of you to make your work available for everyone's use. You may email such code to me at rhyde@cs.ucr.edu.

Table 1: C vs. HLA Data Types

C Type	HLA Primary	HLA Alternates
char	byte	char
unsigned char	byte	uns8, char
signed char	byte	int8
short, signed short	word	int16
unsigned short	word	uns16
unsigned, unsigned int, unsigned long int, unsigned long	dword	uns32
int long int	dword	int32
long long, long long int	qword, dword[2]	
float	real32	dword
double	real64	qword
long double	real80	tbyte
any pointer object (except char*)	dword, pointer to *type*	
char *	string	pointer to char

Generally, I used the HLA untyped data types (byte, word, dword, qword) for most integral one, two, four, and eight byte values. This reduces type coercion syntax in the assembly code (at the expense of letting some logical/semantic errors slip through; I assume that an assembly programmer writing modules know what s/he is doing). For certain data types that are obviously an integer or unsigned type, I may use the UNSxx or INTxx types, but this is rare. For character arrays and objects that are clearly characters (and not eight-bit integral types) I'll use the HLA char type; bytes for everything else.

I'll often use the HLA string type in place of C's char* type. HLA's strings, strictly speaking, are not completely compatible with C's. However, the string data type is a pointer object (just like char*) and HLA's literal string constants are upwards compatible with C's[16].

16. Do keep in mind that HLA maintains an extra pair of dwords with each string to hold the current and maximum string lengths. Since C (Linux) will not be able to use this information, so programmers may prefer not to use HLA literal strings except in BYTE statements in order to save eight bytes of overhead for each string. However, this will create a bit more work for such programmers.

2.16.2 Converting C Structs to HLA Records

HLA's records and C's structs are, perhaps, the greatest problem facing the person converting C header files to HLA. HLA, by default, aligns all fields of a record on byte boundaries. GCC (at least the version that compiles Linux v2.4.7-10 as provided by Red Hat) aligns each field on a boundary that is equal to the field's size[17]. Prior to HLA v1.34, one could align fields of an HLA record using the ALIGN directive inbetween fields of the record. Inserting all those ALIGN directives in a large record (of which there are many in Linux) gets old really fast. Furthermore, all though ALIGN directives clutter up the code. Therefore, I added a couple of record alignment options in HLA v1.34 in order to ease the creation of HLA records compatible with Gcc's structs (and other C compilers, for that matter). Consider the following HLA record declaration template:

```
type
    identifier : record ( const_expr )
        <field declarations>
    endrecord;
```

In particular, note the "(*const_expr*)" component above. This expression, appearing within parentheses immediately after the RECORD reserved word tells HLA to align the fields of the record on the boundary specified by the constant (which must be an integer constant in the range 0..16). If you specify an alignment value of one, then the fields of the record are packed (that is, they are aligned on a byte boundary). If you specify a value greater than one, then HLA aligns each field of the record on the specified boundary. The most interesting option is specifying zero as the alignment value. In this case, HLA will align each field on a boundary that is compatible with that field (e.g., bytes on any boundary, words on even boundaries, dwords on four-byte boundaries, etc.). Note that HLA will only align up to a 16-byte boundary, even if the object is larger than 16-bytes. This is the option you're mostly likely to use when converting Gcc structs to HLA records (since this is, roughly, how Gcc aligns fields in structs). Note that in the current crop of HLA header files I've created, you'll still find a lot of ALIGN directives since much of the conversion was done prior to adding the RECORD alignment feature to HLA.

If you're as paranoid as I am, you'll not set some HLA record alignment value (or even manually insert ALIGN directives) and assume the records match Gcc's. The compiler changes; the options Linux developers specify for structs change, and there are bugs too (in both HLA and Gcc). Therefore, it's wise to always verify that each field of an HLA record appears at the same offset as the corresponding field in the C struct.

The easiest way I've found to verify the size of a C structure compiled by GCC is to create a short GCC program that sets a global variable to the size of the struct and creates an array of pointers to each of the fields of the structure. The following code gives a quick example of this:

```
struct mystruct
{
    int a;
    char b;
    short c;
    long d;
    char e;
    int f;
};

struct mystruct f;
unsigned long mystructsize = sizeof( struct mystruct );

void *i[] =
{
```

17. Strictly speaking, this isn't completely true. Gcc aligns fields that are structures on a boundary compatible with it's largest object and it aligns arrays on boundaries compatible with an array element. There are some other minor differences, too.

```
    &f.a,
    &f.b,
    &f.c,
    &f.d,
    &f.e,
    &f.f
};

int main( int argc, char **argv )
{
}
```

Program 3.12 Example GCC Program to Test struct Field Sizes

If you compile this file to an assembly output file (using the GCC "-S" command line option) then you can easily determine the size of the structure and the offsets to each of the fields by looking at the GAS output that GCC produces.

In HLA, there are a couple of easy ways to get offset and size information. First of all, you can use the "-sym" command line option to tell HLA to generate a symbol table dump after compilation. The symbol table dump displays the offsets of the record fields along with the record's total size. This works great when you can isolate the record (or the record and a few other data types) into a single HLA compilation unit. However, if you're including lots of header files (e.g., linux.hhf) and/or have many symbol declarations in the file you're compiling, dumping the symbol table is not a good idea because it's just too much information to view all at once[18]. Another solution is to use HLA's #print compile-time statement to display the specific information you want. Using the @offset and @size compile-time functions, you can determine the size of a record and the offsets of individual fields, e.g.,

```
type
    r:record
        a:byte;
#print( "a's offset: ", @offset(a))
        b:word;
#print( "b's offset: ", @offset(b))
        c:dword;
#print( "c's offset: ", @offset(c))
        d:qword;
#print( "d's offset: ", @offset(d))
        e:tbyte;
#print( "e's offset: ", @offset(e))
    endrecord;
#print( "size=", @size( r ))
        .
        .
        .
```

When you compile this program containing the code above, it will display the field offsets and the record size on the console. You can compare these numbers against the ones that GCC produced.

It may seem like a tremendous amount of work to compare field offsets as I've suggested here. However, I'd point out that I uncovered several bugs in the HLA *linux.hhf* header files by going through this process. Despite the fact that this is labor-intensive, it is a process that pays off handsomely when attempting to produce defect-free code.

18. Of course, you could send the symbol table dump to a file and view that file with a text editor, but we'll ignore that possibility here.

2.16.3 C Calling Sequences and Wrapper Functions/Macros

Another issue you have to deal with is the function calling sequence that Linux uses. This is a two-way street since your code will call Linux functions and Linux will call your functions. Obviously, the parameter passing mechanisms and parameter clean-up strategies must agree. Since Linux is written (mostly) in GCC, this means that your assembly code must conform to the C calling mechanism. Unfortunately, GCC provides several different calling mechanisms through the use of compiler pragmas, and Linux uses several different options for the functions you may want to call. This means that you'll have to do a little kernel research when writing an HLA prototype for a function in order to determine the calling sequence.

In general, most Linux functions you'll call use the standard C calling sequence. That is, the caller pushes the parameters onto the stack in the reverse order of their declaration and it is the caller's responsibility to remove the parameters from the stack upon return. If you see the macro/attribute "asmlinkage" before a C function declaration, you can be assured that the kernel is using this calling mechanism. Usually, if you don't see any specific attribute in front of a function declaration, you can also assume that the function uses the standard C calling sequence. If you see the macro "FASTCALL(*type*)" in front of a function declaration, then GCC will pass the first three (four byte or smaller) parameters in EAX, EDX, and ECX (respectively); GCC passes additional parameters on the stack (though there are rarely more than three parameters in a function the kernel declares with the FASTCALL attribute). Obviously, upon return from a FASTCALL function, the caller must not remove data from the stack that was passed in the registers.

If you're calling a Linux-based function, the first step is to create an HLA procedure prototype for the C function. If that function's name is CFunc and it has dword parameters a, b, and c, the HLA prototype will probably look something like the following:

```
procedure CFunc( a:dword; b:dword; c:dword ); @cdecl; @external;
```

Note the "@cdecl" attribute! This is very important and it's absence is a common source of defects in HLA code that attempts to call a C function. Without this procedure attribute, HLA will push the parameters in the wrong order when you call CFunc from your HLA code (with disasterous results, usually). Of course, you could always manually push the parameters yourself and avoid this issue, e.g.,

```
push( cParmValue );
push( bParmValue );
push( aParmValue );
call CFunc;
add( 12, esp );
```

Rather than:

```
CFunc( aParmValue, bParmValue, cParmValue );
add( 12, esp );
```

However, the former sequence is harder to write, harder to read, and harder to maintain. Better to just ensure you've got the @cdecl procedure option attached to the HLA prototype and use the high-level calling sequence HLA provides.

Especially note the "add(12, esp);" instruction following the function call in the two examples above. Another common error in assembly code that calls C functions is forgetting to remove the parameters from the stack upon return from the function. Obviously, this can lead to problems later on in your code.

Another issue you must deal with when calling functions written in C is GCC's register preservation convention. As I write this (gcc v2.96), GCC preserves all general-purpose registers except EAX, EDX, and ECX across a function call (the same registers, incidentally, that it uses to pass parameters when using the FASTCALL attribute[19].

GCC functions return one-, two-, and four-byte function results in EAX; it returns eight-byte parameters in EDX:EAX (this is true for structure return results as well as scalar return results). GCC returns larger

19. Actually, FASTCALL is not a GCC attribute, it is a macro that expands to __attribute__(regparm(3)) which tells GCC to pass up to three parameters in register; those registers being EAX, EDX, and ECX, respectively.

function results differently, but fortunately you won't have to deal with this issue in modules since the kernel functions you're likely to call don't return structure results (they may return pointers to structs, but they don't return structures by value). Clearly, a call to a non-void function will disturb at least EAX and may disturb EDX as well when the function returns 64-bit values.

Because calls to GCC functions can wipe out the EAX, ECX, and EDX registers, you must preserve these registers across a function call (even if it's a void function) if you need to maintain their values across the call. Although the C calling convention treats EAX, ECX, and EDX as volatile values (that is, their values can change across the function call), the standard assembly convention is to preserve register values across function calls unless the function explicitly returns a value in said register(s). If you normally follow this convention in your assembly code, it's easy to forget that the C code can wipe out EAX, ECX, and/or EDX and have your code fail when the function call destroys an important value you need preserved across the call. Of course, assembly programmers are supposed to know what they're doing and keeping track of register usage across procedure calls is part of the assembly language programmer's responsibility. One of the advantages of a high level language like C is not having to deal with these issues. On the other hand, explicitly dealing with these issues is why assembly language code is often more efficient than compiled C code. Nevertheless, in many instances maintainability and readability are more important than efficiency (e.g., if you only call function once in an initialization routine, who really cares if it executes a few extra machine instructions?). Fortunately, HLA's macro facilities let you enjoy the best of both worlds – you can write easy to use and maintain code that might be slightly less efficient or (within the same program) you can write tight code when efficiency is a concern.

If you look at the *linux.hhf* header files you'll notice that the HLA Standard Library modules provide *wrapper functions* for most Linux system calls. These wrapper functions use the standard HLA (Pascal) calling sequence where you pass the parameters on the stack and the wrapper function takes the responsibility for removing the parameters from the stack, placing them in registers, and executing the INT($80) instruction to actually call Linux. Upon return, the wrapper function automatically cleans up the stack before returning to the application program that made the call.

Using wrapper functions has several advantages. It lets you change the calling sequence, rearrange parameters, and check the validity of the parameters before calling Linux. Unfortunately, there is some overhead associated with calling a wrapper function (you have to push the parameters on the stack for the wrapper, you have an extra call and return, and there is some code needed to set up the procedure's activation record inside the wrapper function). Because calls to Linux involve a user-mode/kernel-mode switch (which is expensive), the extra overhead of a wrapper function in an application program is negligible. In a device driver, however, there is no user-mode/kernel-mode context switch since the device driver module is already operating in kernel mode. Module code makes direct calls to functions in the Linux kernel. So although you can still write wrapper functions, their overhead isn't always negligible compared to the execution time required by the actual Linux function they invoke. Nevertheless, writing a wrapper function is sometimes useful because it lets you do other things (such as check the validity of parameters, insert debugging code, preserve registers, and clean up the stack after the actual Linux call) that would be painful to do on each and every call to the Linux function. Use the mythical Linux kernel CFunc example above, here's how you'd typically write a wrapper function to use in your module code:

```
procedure _CFunc( a:dword; b:dword; c:dword ); @cdecl; @external("CFunc");

procedure CFunc( a:dword; b:dword; c:dword );
    @nodisplay;
    @noalignstack;
    returns( "eax" );
begin CFunc;

    push( ecx );  // preserve ECX and EDX across call
    push( edx );  // (assume function result comes back in EAX.
    _CFunc( a, b, d );
    add( 12, esp );  // Clean up stack upon return.
    pop( edx );      // Restore the registers.
    pop( ecx );
```

```
end CFunc;
```

There are several things here to take notice of. First, note that this code names the external Linux function _CFunc in the HLA code but uses the external name "CFunc" (so that the kernel will properly link calls to _CFunc to the kernel's CFunc code). Then the code above names the wrapper function CFunc so you can call the wrapper function using the original Linux function name; this is better than using a different name because you won't accidentally call the original function when you attempt to call the wrapper function.

One problem with the scheme above occurs if you want to make the CFunc wrapper function external so you can call it from several different source files in your device drivers. This is easily accomplished by using an external declaration for CFunc (wrapper function) like the following:

```
procedure CFunc( a:dword; b:dword; c:dword ); @cdecl; @external("_CFunc");
```

Note that this external declaration, combined with the previous code, swaps the internal and external names of the CFunc and _CFunc functions (that is, internally CFunc is the wrapper function and _CFunc is the original Linux code while externally _CFunc is the wrapper function and CFunc is the Linux code).

One thing nice about wrapper functions like the above in assembly language is that you're not forced to use them. If a particular call to the Linux CFunc code needs to be as efficient as possible, you can always explicitly call the original C code yourself using the _CFunc label, e.g.,

```
_CFunc( aValue, bValue, cValue );
add( 12, esp );
```

One reason you might want to do this is because you're making several successive Linux calls and you only want to clean up the stack after the entire sequence. The following example demonstrates how to make three successive calls to _CFunc and clean up the stack only after the last call:

```
_CFunc( 0, 5, x );
_CFunc( eax, 5, y );  // uses previous _CFunc return result.
_CFunc( eax, 0, -1 ); // uses previous _CFunc return result.
add( 36, esp );       // Clean up stack.
```

This saves two instructions, which is useful on occasion. Don't forget that the code above can obliterate ECX and EDX!

Since wrapper functions can introduce a fair amount of overhead, especially for simple Linux functions, the use of wrapper functions when calling Linux functions isn't always appropriate. A better solution is to expand the wrapper function in-line rather than call the wrapper function. In HLA, of course, this is accomplished by using macros. A macro version of CFunc would look like the following:

```
procedure _CFunc( a:dword; b:dword; c:dword ); @cdecl; @external("CFunc");

#macro CFunc( a,b,c );
    returns
    ({
        push( ecx );  // preserve ECX and EDX across call
        push( edx );  // (assume function result comes back in EAX.
        _CFunc( a, b, d );
        add( 12, esp );  // Clean up stack upon return.
        pop( edx );      // Restore the registers.
        pop( ecx );
    }, "eax" )
#endmacro
```

The code above embeds the macro body in a returns statement so you may invoke this macro anywhere EAX is legal (in addition to being able to invoke this macro as an HLA statement). That is, both of the following invocations are legal:

```
CFunc( 0, 5, x );
    .
    .
    .
```

```
if( CFunc( 0, 1, y ) = 5 ) then
    ...
endif;
```

Since most Linux functions you'll call return some result, using the returns statement in the macro is very handy.

Most of the Linux interface functions appearing in the *linux.hhf* header file are actually macros very similar to the CFunc macro above. See the *linux.hhf* header file (and the files it includes) for the exact details. In particular, if you want to call the original Linux function directly, you'll need to call the actual procedure, not invoke the macro (like _CFunc above, most external Linux function prototypes use the original name with a "_" prefix on the name).

2.16.4 Kernel Types vs. User Types

One minor issue of which you should be aware is that there are some differences between data types in the kernel headers and data types in user programs. Some types are words in user-land and they are dwords in kernel-land. When converting types from C to HLA, be sure you are using the kernel types. However, since application programs as well as device drivers include the *linux.hhf* header file, a problem exists – how to define the type so that applications see word (for example) and the kernel sees dword? The solution is to use conditional compilation and the __kernel__ symbol. Remember, device drivers (and other kerrnel code) must define the symbol __kernel__ prior to including *linux.hhf* (or any other Linux releated header file). Therefore, you can test for the presence of this symbol using the #if statement to determine how to define the object as the following example demonstrates:

```
type
    #if @defined( __kernel__ ))

        typ    :dword; // inside kernel, it's a dword.

    #else

        type   :word;  // In user-land, its a word.

    #endif
```

2.17 Some Simple Debug Tools

Although this text will cover debugging in greater depth in a later chapter, it's worthwhile to present some simple debugging tools early on so you'll be able to debug the code you're working with over the next few chapters. You've already seen the *linux.printk* function, which is probably the primary kernel debugging tool you'll use. In this section, we'll take a look at a couple of additional macros the *linux.hhf* header file provides to help you debug your system.

The *linux.hhf* header file provides two macros, kdebug and kassert, that let you inject debugging code into your device driver during development and easily remove that code for production versions of your module. Here's these macros as they appear in the *kernel.hhf* header file:

```
// kdebug is outside any namespace because we're going
// to use it fairly often.
// Ditto for kassert (text constant).


#if( @defined( __kernel__ ))
```

```
    ?KNDEBUG :boolean := boolean(1); // default is true (debug off).

    #macro kdebug( instrs );
        #if( !KNDEBUG )
            pushad();
            pushfd();
            instrs;
            popfd();
            popad();
        #endif
    #endmacro;


const
    kassert :text :=
        "?linux.kassertLN := @linenumber; "
        "?linux.kassertFN := @filename; "
        "linux.kassert";

namespace linux;

val
    kassertLN: dword;
    kassertFN: string;


    #macro kassert( expr ):skipCode,msg,fn,ln;
        #if( !KNDEBUG )
            readonly
                msg :string := @string:expr;
                fn  :string := linux.kassertFN;
                ln  :dword := linux.kassertLN;
            endreadonly;

            pushfd();
            jt( expr ) skipCode;
            pushad();
            linux.printk
            (
                "Kernel assertion failed: '%s' (line:%d, file: %s)\n",
                msg,
                ln,
                fn
            );
            popad();

            // We can't really abort the kernel, so just keep going!

            skipCode:
            popfd();

        #endif
    #endmacro;
end linux;

#endif
```

Program 3.13 kassert and kdebug Macros

Both of these macros (kassert and kdebug) are controlled by the current value of the KNDEBUG val constant in your source file[20]. By default, KNDEBUG (K*ernel* N*o* DEBUG) contains true, meaning that debugging is disabled. You may activate kernel debugging using the following HLA statement:

```
?KNDEBUG := false;
```

Since KNDEBUG is a compile-time variable, you can explictly turn debugging on and off in sections of your source file by sprinkling statements like the following throughout your code:

```
?KNDEBUG := false;

<< statements that run with debugging turned on >>

?KNDEBUG := true;

<< statements that run with debugging turned off >>

?KNDEBUG := false;

<< statements that run with debugging turned on >>

etc...
```

Generally, you'll just turn on or off debugging for the whole source file with a single assignment to KNDEBUG. However, if you're getting too much debugging output, you can select which sections of your code have active debug sections by using statements like the above.

We'll take a look at the kdebug macro first, since it's the simplest of the two macros to understand. The whole purpose of this macro is to let you insert debugging code into your source file that you can easily remove for production code (but leave in the source file so you can reactivate it later if further debugging is necessary). The kdebug macro use the following syntax:

```
kdebug
(
    << sequence of debugging statements >>
);
```

The kdebug macro above is simply a shorthand version of the following conditional code:

```
#if( !KNDEBUG )
    pushad();
    pushfd();

    << sequence of debugging statements >>

    popfd();
    popad();

#endif
```

There are two important things to note about this sequence: first, it preserves the integer registers and flags across your debugging statements, so you may use the registers and flags as you see fit within the kdebug macro without worrying about their side effects in the device driver; the second thing to note about this code sequence is that it disappears if KNDEBUG contains true. Therefore, as noted earlier, you can make all

20. KNDEBUG is a compile-time variable; that is, a constant whose value you may change at various points during compilation, but whose value is constant at run-time.

these debugging statements go away (or be present) by changing the compile-time KNDEBUG value. Since the kdebug macro expands to the code above, you could easily subsitute the #if code in place of the kdebug macro invocation. However, the kdebug invocation is certainly more convenient, hence it's inclusion in the *kernel.hhf* header file. However, keep this equivalence in mind if you need a debug sequence that does not preserve the registers or flags (i.e., in debug mode you want to force one of the registers to have a different value).

The kassert macro actually isn't a macro, but a text constant that sets up the linux.kassertLN and linux.kassertFN constants with the line number and filename (respectively) and then invokes the linux.kassert macro. The linux.kassert macro (just kassert, hereafter) expects a run-time boolean expression (like you'd use with IF, WHILE, etc.). If this boolean expression evaluates true, kassert does nothing. However, if the expression evaluates false, then kassert uses linux.printk to write an "assertion failure" message to the log file. Those who are familiar with C's assert macro or HLA's ex.assert macro should realize that a kassert failure does not abort the program (which would mean aborting the kernel). Instead, control continues with the next statement following the kassert macro invocation, whether the assertion succeeds or fails. If the assertion causes the kernel to segfault (or otherwise misbehave), at least you'll have a record of the assertion failure in the */var/log/messages* file.

Of course, you can also write your own code and macros to inject debug code into your module. It's not a bad idea to use the KNDEBUG value to control code emission of debug code in your module (as kassert and kdebug do) so that you can easily control all debug code emission with a single statement.

When writing a device driver module, especially in assembly language, it's a real good idea to insert a lot of debug statements into your code so you can figure out what happened when your device driver crashes. Often when there is a kernel fault, you'll have to reboot before you can rerun the driver. Therefore, the more debug information you write to the /var/log/messages file, the fewer reboots you'll need to track down the problem. As noted earlier, we'll return to the subject of debugging device drivers in a later chapter.

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.

Index