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.