Creating a Linux Device Driver

Much of the following was taken from “The Linux Kernel Module Programming Guide” by Peter Salzman (http://www.tldp.org/LDP/lkmpg/)

1. Define the hardware device.

This requires specifying the name of the device and major and minor device numbers. Normally, device numbers are assigned to a manufacturer and are unique for each device. Fortunately, Linux has reserved some device numbers for demonstrations and experiments: Major 60-63 and minor 0-255.

Devices are normally defined by special files in **/dev**. A program reads or writes to a device by treating it as a ‘file’. Devices are specified to be character- or block- oriented depending upon whether they need a buffering. To define a simple device, use **mknod** as superuser. The following example defines **/dev/my_device** as character oriented with major and minor numbers of 60 and 128.

$ **mknod /dev/my_device c 60 128**

2. Write a device driver

A device driver program is an example of a Linux *module*. A module is different from a user-program in that a) there is no *main(),* b) each module must include members *init_module()* and *cleanup_module(),* and c) a module may not call library functions – only functions available in the kernel as listed in **/proc/ksyms**.

The *init_module()* function is invoked when the module is installed. The *cleanup_module()* is invoked when the driver is removed. The primary responsibility of the *init_module()* function is to register the module as the device driver for the hardware device you defined in step #1 above. For a character device, this is done with the *register_chrdev()*. The arguments to this function are the major device number, the name of the device (used in the **mknod** command), and a *struct file_operations* structure that Linux will add to the master device table.

The minor number isn’t really used by Linux – it is available to the device driver to differentiate between a family of similar devices. As such, family of devices sharing the same major number may be serviced by a single device driver.

The structure is used to list the functions within the driver that are to be invoked by Linux when the user performs system calls for file operations; *open(), close(), read(), write(), ioctl(), etc.* Remember, all C++ library functions that do I/O are wrappers around these Linux system calls.

Here’s the skeleton of a valid device driver – it just doesn’t do anything. The important sections are documented below by label;

```
// Demonstration device driver:  hello.c

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

#define DEVICE_NAME            "hello"
#define MAJOR_DEVICE_NUMBER    60

MODULE_AUTHOR("Scott Cannon <scott.cannon@usu.edu>");       // A
MODULE_LICENSE("GPL");

static int my_read
```

```
      (struct file *filp, char *buf, int length, int *offset);
static int my_open (void);
static int my_close (void);                                 // B
int init_module (void);
void cleanup_module (void);

struct file_operations fops =                               // C
{     .read = my_read,
      .open = my_open,
      .release = my_close
};

static int my_open (void)                                   // D
{     return 0;
}

static int my_close (void)                                  // E
{     return 0;
}

static int my_read                                          // F
      (struct file *filp, char *buf, int length, int *offset)
{     return 0;
}

int init_module (void)                                      // G
{  major = register_chrdev (MAJOR_DEVICE_NUMBER, DEVICE_NAME, &fops);
   if (major < 0)
      return major;
   else
      return 0;
}

void cleanup_module (void)                                  // H
{  unregister_chrdev (major, DEVICE_NAME);
}
```

Section explainations

A. When a driver module is installed, the module should contain a special header that specifies the license and author of the module. Otherwise, Linux may not install the module when the time comes. These two macros create this header with the given strings. The string used for license is examined by Linux when the module is installed for a recognizable value. The string “GPL” can be used for demonstration and experimentation.

B. This section is simply the prototypes of all module functions. This isn’t normally necessary in a C/C++ program, but is required for a kernel module.

C. The special **struct file_operations** section is where the table of member routines is specified for each Linux system call associated with the hardware device. This syntax may be new to many C++ programmers -- it simply assigns function names to members of the structure. Those that aren’t defined default to NULL. To see what system call functions are available to be defined, look in **fs.h**. *Release* refers to the *close()* function – the other two are obvious.

D. This function is invoked in response to a Linux *open()* call for **/dev/my_device**

E. This function is invoked in response to a Linux *close()* call.

F. This function is invoked in response to a Linux read() call. The parameters are defined below;

file *filp,	// a pointer to a special structure containing the state of the device
char *buf,	// a pointer to the user's array when data is to be inserted
int length,	// the number of bytes requested by the caller
int *offset	// the current position within the file.

(In the example below, the first and last parameters are simply ignored.)

G. The function init_module() is invoked by Linux when the module is installed. This is where the driver is associated or bound to the **/dev/my_device**; by passing the *register_chrdrv()* the major device number, the name of the driver module, and the list of driver functions to be associated with system I/O calls. If the registration fails, the return code will be an error.

H. This function is invoked when the device driver is removed

The device driver given below is a complete example of a read-only device. In this case, the hardware device defined in step #1 above really doesn't exist. This driver simply 'generates' characters when the device is opened. Normally, the device driver would access the physical registers of the hardware device to input chars. Let's look over the code and then explain each section in detail:

```
// Demonstration device driver:  hello.c

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

#define DEVICE_NAME            "hello"          // I
#define MAJOR_DEVICE_NUMBER    60
#define BUF_LEN                128

static char msg[BUF_LEN];                       // J
static char *msg_ptr;
static int device_open = 0;
static int major;
static int count = 0;

MODULE_AUTHOR("Scott Cannon <scott.cannon@usu.edu>");
MODULE_LICENSE("GPL");

static int my_read (
      struct file *filp, char *buf, int length, int *offset);
static int my_open (void);
static int my_close (void);
int init_module (void);
void cleanup_module (void);

struct file_operations fops =
{     .read = my_read,
      .open = my_open,
      .release = my_close
};

static int my_open (void)
{     if (device_open)
            return -EBUSY;                      // K
      device_open = 1;
      sprintf (msg, "Hello from the device on open # %d!\n", count++);
      msg_ptr = msg;                            // L
      return 0;
}

static int my_close (void)
{     device_open = 0;
      return 0;
}

static int my_read
      (struct file *filp, char *buf, int length, int *offset)
{     int bytes_read = 0;
      if (*msg_ptr == 0) return 0;              // M
      while (length && *msg_ptr)
      {     put_user (*(msg_ptr++), buf++);     // N
            length--;
            bytes_read++;
      }
      return bytes_read;                        // P
}

int init_module (void)
{  major = register_chrdev (MAJOR_DEVICE_NUMBER, DEVICE_NAME, &fops);
   if (major < 0)
      return major;
   else
      return 0;
}

void cleanup_module (void)
{     unregister_chrdev (major, DEVICE_NAME);
}
```

Section explainations

I. The module name need not be the same as the hardware device name used in the **mknod** command. Typically, this is the name of the source file that contains the driver (sans .c extension).

J. Different driver functions usually communicate with each other via global variables. To retain the values of these variables when the driver is not active, we qualify declarations with the **static** keyword.

K. If the device has already been opened, return a pre-defined error code for failure.

L. Here is where the driver ‘generates’ characters for the device. It simply copies a fixed literal string into an array that the *read()* function will access.

M. The variable *msg_ptr* points to the next char to be read. If it is pointing to the end of the string, *read()* returns a failure or zero bytes read.

N This loop copies bytes from the driver array into the user’s array. Since these two arrays are in completely different environments (user space vs. kernel space), we can’t simply use an assignment statement. A special kernel function *put_user()* is available to do the job.

P. By definition, *read()* should return the number of chars successfully read.

3. Compiling the device driver

Since the device driver isn't linked to any libraries, you should skip the linking stage of compilation. In other words, the .o file is all the kernel needs.

You might ask how the driver has access to kernel functions and variables if it isn't linked into the kernel. Linux 'exports' all available symbols when it is compiled in much the same way as the –g debug option under gcc/g++ provides symbol access to a debugger. In this way, the driver can be dynamically added into the Linux kernel without recompiling the entire kernel. The driver can also be dynamically removed from the kernel.

Compilation is normally done with all warnings enabled, since a bug can be catastrophic. You also want to reference the .h files used when your kernel was compiled, rather than the standard .h file. Here is a simple makefile for the above device driver;

```
TARGET      := hello
WARN        := -W -Wall -Wstrict-prototypes -Wmissing-prototypes
INCLUDE     := -isystem /lib/modules/`uname -r`/build/include
CFLAGS      := -O2 -DMODULE -D__KERNEL__ $(WARN) $(INCLUDE)
CC          := gcc

$(TARGET).o :     $(TARGET).c
```

To use this makefile, use the following command line;

```
$ make –f hello.mak
```

You will see several warnings. Check them over before you ignore them.

4. Installing the device driver

This is done with the **insmod** command (as superuser);

```
$ insmod –r –N hello.o
```

(You can examine a list of all modules currently installed using the **lsmod** command. If the above was successful, you will see the module listed. To remove a module, use the **rmmod** command.)

5. Using the device driver

The device can now be treated as a Linux file. For example;

```
$ cat /dev/my_device
Hello from the device on open #0!
$ cat /dev/my_device
Hello from the device on open #1!
```

Similarly in a user program;

```
char word1[32], word2[32], word3[32];
ifstream infile;

infile.open("/dev/my_device");
infile >> word1 >> word2 >> word3;
cout << word1 << endl;
cout << word2 << endl;
cout << word3 << endl;
```

Produces the following output;

```
Hello
from
the
```

Be sure to un-register the driver when you are through with it. Also, delete the **/dev/my_device** file since you don't want some future program to attempt to open it – since *my_open()* is no longer there.