Embedded Systems
Laboratory Exercise 3

Character Device Drivers

Part I

The purpose of this laboratory exercise is to teach you how to write and implement character device drivers. *Character device drivers* are kernel modules that provide a file-based interface to user-level programs. When a character device driver is inserted into the Linux* kernel, a special type of file associated with the driver is created, usually in the filesystem folder */dev*. For example, if the driver is named *chardev* then the associated file would be */dev/chardev*. A user-level program can read or write to this file to communicate with the driver. In this exercise we will develop a character device driver that can communicate with the user to print different text strings on the Terminal window. While this is not an especially interesting driver, it will serve to illustrate how the code for a character device driver has to be structured. In later laboratory exercises we will use the knowledge developed here to write drivers that perform more useful functions, such as controlling hardware devices like switches, pushbuttons, LEDs, timers, video and audio output, and an accelerometer.

An example of code for a character device driver is given in Figure 1. This is a trivial example that is meant to illustrate how the code for a driver has to be structured. If a user reads from this driver, it simply replies with the text `Hello from chardev`. One way to read from the driver is to execute the Linux command `cat /dev/chardev`. This driver supports both reading and writing. The user can change the character string provided by the driver by writing to it. One way to write to the driver is to use the Linux command `echo "New message" > /dev/chardev`.

Linux provides a number of programming interfaces that can be used to specify a character device driver. For this exercise we will use an interface that is intended for making *miscellaneous* drivers for custom hardware. The driver can be specified by using some special types of variables and data structures, as well as a library function that *registers* the driver in the Linux kernel. Important lines of code in Figure 1 are described below. Lines 1 to 4 include some header files that are required for character device drivers. Lines 7 to 10 provide prototype declarations for the functions `device_open`, `device_release`, `device_read`, and `device_write`. These are the functions that will be called by the Linux kernel when a user program performs an open, close, read, or write operation, respectively, on the file */dev/chardev*. To support these functions the driver declares the variable *chardev_fops* in Line 12, which has the type `struct file_operations`. It provides references to the functions from Lines 7 to 10.

Line 23 declares a variable named *chardev*, which is used to represent our character device driver. It has the type `struct miscdevice`. For our character device driver the fields *.minor* and *.mode* can be set to the values shown in this example. The *.name* field specifies the name of the driver, which is *chardev* in this example. Finally, the *.fops* field contains a pointer to the driver's `file_operations` data structure.

The driver is created in the *start_chardev* function shown in lines 34 to 46. It calls the Linux library function *misc_register* with an argument that is a pointer to the *chardev* variable. If this function returns a value ≥ 0, then the character device driver will have been successfully created by the Linux kernel. The `start_chardev` function is executed when the character device driver is *inserted* into the Linux kernel. When this driver is removed from the kernel, the function `stop_chardev` is called, shown in lines 48 to 53. Note that it is possible to print error or debug information on the Linux terminal window, by calling the *printk* kernel function. This function works similarly to the C library function *printf*, and has the same types of formatting options. Examples of *printk* are shown in Lines 37 and 40. When *printk* is given the `KERN_ERR` argument, its output appears on the Linux Terminal window. But if the `KERN_INFO` argument is used, then the output is stored into a Linux *log* file. You can examine the content of this log file by executing the Linux command `dmesg`.

```c
#include <linux/fs.h>           // struct file, struct file_operations
#include <linux/module.h>       // for module init and exit macros
#include <linux/miscdevice.h>   // for misc_device_register and struct miscdev
#include <asm/io.h>

/* Kernel character device driver /dev/chardev. */
static int device_open (struct inode *, struct file *);
static int device_release (struct inode *, struct file *);
static ssize_t device_read (struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

static struct file_operations chardev_fops = {
    .owner = THIS_MODULE,
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};

#define SUCCESS 0
#define DEV_NAME "chardev"

static struct miscdevice chardev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DEV_NAME,
    .fops = &chardev_fops,
    .mode = 0666
};
static int chardev_registered = 0;

#define MAX_SIZE 256 // assume that no message longer than this will be used
static char chardev_msg[MAX_SIZE];  // the string that can be read or written

static int __init start_chardev(void) {
    int err = misc_register (&chardev);
    if (err < 0) {
        printk (KERN_ERR "/dev/%s: misc_register() failed\n", DEV_NAME);
    }
    else {
        printk (KERN_INFO "/dev/%s driver registered\n", DEV_NAME);
        chardev_registered = 1;
    }
    strcpy (chardev_msg, "Hello from chardev\n");

    return err;
}

static void __exit stop_chardev(void) {
    if (chardev_registered) {
        misc_deregister (&chardev);
        printk (KERN_INFO "/dev/%s driver de-registered\n", DEV_NAME);
    }
}
```

Figure 1: The character device driver code. (Part a)

```
54 /* Called when a process opens chardev */
55 static int device_open(struct inode *inode, struct file *file) {
56     return SUCCESS;
57 }
58
59 /* Called when a process closes chardev */
60 static int device_release(struct inode *inode, struct file *file) {
61     return 0;
62 }
63
64 /* Called when a process reads from chardev. Provides character data from
65  * chardev_msg. Returns, and sets *offset to, the number of bytes read. */
66 static ssize_t device_read(struct file *filp, char *buffer, size_t length,
       loff_t *offset) {
67     size_t bytes;
68     bytes = strlen (chardev_msg) - (*offset);  // how many bytes not yet sent?
69     bytes = bytes > length ? length : bytes;   // too much to send at once?
70
71     if (bytes)
72         (void) copy_to_user (buffer, &chardev_msg[*offset], bytes);
73     *offset = bytes; // keep track of number of bytes sent to the user
74     return bytes;
75 }
76
77 /* Called when a process writes to chardev. Stores the data received into
78  * chardev_msg, and returns the number of bytes stored. */
79 static ssize_t device_write(struct file *filp, const char *buffer, size_t
       length, loff_t *offset) {
80     size_t bytes;
81     bytes = length;
82
83     if (bytes > MAX_SIZE - 1) // can copy all at once, or not?
84         bytes = MAX_SIZE - 1;
85     (void) copy_from_user (chardev_msg, buffer, bytes);
86     chardev_msg[bytes] = '\0';   // NULL terminate
87     // Note: we do NOT update *offset; we keep the last MAX_SIZE or fewer bytes
88     return bytes;
89 }
90
91 MODULE_LICENSE("GPL");
92 module_init (start_chardev);
93 module_exit (stop_chardev);
```

Figure 1. The character device driver code. (Part *b*)

Lines 54 to 62 in Figure 1 give the code for `device_open` and `device_release`. For most character device drivers nothing needs to be done in these functions, and hence the code simply *returns*. Lines 64 to 75 show the `device_read` function. The first argument to this function, `filp`, is not used in this example. The second argument, `buffer`, is used to pass character data from the driver back to the user-level process that read from */dev/chardev*. The `length` argument specifies the maximum number of bytes that can be stored in the `buffer`. The final argument, `offset`, will be discussed later. Lines 68 to 69 determine how much data can be sent back to the kernel. If `length` is greater than `bytes` then the whole message can be sent at once. Otherwise only `length` characters can be sent. The value of `bytes` is calculated in Line 68.

To understand the way that the `offset` variable determines how many bytes are returned to the user in the read operation, consider the following scenario. When a user opens the file */dev/chardev*, and then performs a read from the file, `*offset` will be 0. Thus, `bytes` for this read operation will be set to the length of the character

string that is returned when the driver is read. In Linux the kernel does not directly access the memory addresses, or variables, in a user-level program. Similarly, user-level code cannot access memory addresses within the kernel. The kernel provides special functions for passing data between itself and user-level programs. One such function, `copy_to_user`, is called in Line 72 of Figure 1. It copies character data from the kernel back to the user-level program via the `buffer`. The kernel variable `offset` is set to the value of the variable `bytes` in line 73. This value of `offset`, associated with */dev/chardev*, is stored in the kernel as long as the file remains open. In line 74 the `device_read` function provides as a return value the number of bytes stored into the `buffer`. A typical user-level program that reads from the device driver (e.g., `cat /dev/chardev`) will read from the file until `device_read` returns 0, which indicates *end-of-file*. The way this works in our example is as follows. The first time `device_read` is called it copies the character string back to the `buffer` and returns the *string length*. But a second call to `device_read` will copy nothing into the `buffer` and will return 0. This mechanism is facilitated by the way in which `offset` is used in the code.

Lines 77 to 89 show the `device_write` function. The first argument to this function, `filp`, is not used in this example. The second argument, `buffer`, is used to get character data from the user program into the device driver. The `length` argument specifies the amount of data that is to be transferred. The `offset` argument is not used in this example. Data transferred from the user is stored into `chardev_msg`, overwriting the previous message. Line 83 checks for overflow, so that the amount of data can be reduced if needed. In line 85 the kernel function `copy_from_user` is called to get the user-data and copy it into `chardev_msg`.

The `device_write` function provides as a return value the number of bytes copied into `chardev_msg`. A typical user-level program (e.g., `echo "New message" > /dev/chardev`) will continue to call the `device_write` function until a total of `length` bytes have been received by the driver. In our example each call to `device_write` overwrites the data stored in `chardev_msg`.

Perform the following:

1. Create a C source-code file named *chardev.c* for the device driver code in Figure 1. The source code can be obtained from the design files that accompany this lab exercise on the *FPGAcademy.org* website.

2. Create a Makefile for your character device driver, following the format given in the tutorial *Using Linux on DE-series Boards*. Compile the code to create the kernel module *chardev.ko*, and insert this module into the Linux kernel. Check the filesystem folder */dev* to see that the file */dev/chardev* was created as a result of inserting your kernel module. Type the command `cat /dev/chardev` and observe that your character device driver responds with the message `Hello from chardev`. Overwrite the default message by typing a command such as `echo "New Message" > /dev/chardev`. Then, issue the `cat` command again to see that the driver responds with the new message.

3. In addition to using commands like `cat` and `echo`, you can write your own user-level programs that read and write to your character device driver. An example program is given in Figure 2. It uses the kernel functions `open`, `read`, `write`, and `close` to communicate with the character device driver through the file */dev/chardev*. Create a C source-code file called *part1.c* for this program, and compile it using a command such as `gcc -Wall -o part1 part1.c`. The source code in Figure 2 is provided in the design files for this exercise. Run the resulting executable program and observe the output that it produces.

```c
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>

#define BYTES 256    // max # of bytes to read from /dev/chardev

volatile sig_atomic_t stop;
void catchSIGINT(int signum){
    stop = 1;
}

/* This code uses the character device driver /dev/chardev. The code reads the
 * default message from the driver and then prints it. After this the code
 * changes the message in a loop by writing to the driver, and prints each new
 * message. The program exits if it receives a kill signal (for example, ^C
 * typed on stdin). */
int main(int argc, char *argv[])
{
    int chardev_FD;              // file descriptor
    char chardev_buffer[BYTES];  // buffer for chardev character data
    int ret_val, chars_read;     // number of characters read
    char new_msg[128];           // space for the new message
    int i_msg;

    // catch SIGINT from ^C, instead of having it abruptly close this program
    signal(SIGINT, catchSIGINT);

    // Open the character device driver for read/write
    if ((chardev_FD = open("/dev/chardev", O_RDWR)) == -1) {
        printf("Error opening /dev/chardev: %s\n", strerror(errno));
        return -1;
    }

    i_msg = 0;
    while (!stop) {
        chars_read = 0;
        while ((ret_val = read (chardev_FD, chardev_buffer, BYTES)) != 0)
            chars_read += ret_val;          // read the driver until EOF
        chardev_buffer[chars_read] = '\0';  // NULL terminate
        printf ("%s", chardev_buffer);

        sprintf (new_msg, "New message %d\n", i_msg);
        i_msg++;
        write (chardev_FD, new_msg, strlen(new_msg));

        sleep (1);
    }

    close (chardev_FD);
    return 0;
}
```

Figure 2: A program that communicates with */dev/chardev*.

An alternative version of the `read_device` function is given in Figure 3. Instead of calling the kernel function `copy_to_user`, this code copies one byte at a time by calling the kernel function `put_user`. Similarly, an alternative version of the `device_write` function that transfers one byte at a time instead of calling `copy_from_user` is shown in Figure 3. These alternative versions of the code are meant to provide additional examples of how device driver code can be written. The two versions of the code perform exactly the same functions.

```
/* Called when a process reads from chardev. */
static ssize_t device_read(struct file *filp, char *buffer, size_t length,
    loff_t *offset) {
    size_t bytes_read = 0;
    char *msg_Ptr = &(chardev_msg[*offset]);

    // Write to user buffer
    while (length && *msg_Ptr)  {
        put_user(*(msg_Ptr++), buffer++);
        length--;
        bytes_read++;
    }
    (*offset) = bytes_read;
    return bytes_read;
}

/* Called when a process writes to chardev */
static ssize_t device_write(struct file *filp, const char *buffer, size_t
    length, loff_t *offset) {
    int i;

    for (i = 0; i < length; ++i)
        chardev_msg[i] = buffer[i];  // assume that data won't overflow
    chardev_msg[i] = '\0';           // NULL terminate
    return length;
}
```

Figure 3: Alternative versions of `device_read` and `device_write`

Part II

The discussion below assumes that the reader is using the DE1-SoC board. If you are using the DE10-Standard board, then the same discussion applies. But if you are using the DE10-Nano board, then some minor differences should be considered, because this board has fewer pushbutton KEYs and SW switches.

In some Linux systems user-level programs are not permitted to access the physical memory addresses of I/O devices. In such systems I/O devices can only be accessed via device drivers. In this part you are to implement two character device drivers. One driver provides the state of the KEY pushbutton switches in the DE1-SoC Computer, via the file */dev/KEY*. The other driver provides the state of the SW slider switches, via the file */dev/SW*.

Perform the following:

1. Create a new kernel module in a file *KEY_SW.c*. Write the code for *both* character device drivers in this module. Declare separate variables of type `file_operations` and `miscdevice` for each driver. Initialize each of these variables by following the steps shown in Figure 1. When setting up the `file_operations` data structure (see line 12 in Figure 1), each of your character device drivers needs to have a function for `opening`, `releasing`, and `reading` its */dev/file*. These drivers do not require a function for `writing`, since a user would not need to write anything to */dev/KEY* or */dev/SW*.

Your module needs to have an initialization function, like the one beginning on Line 34 of Figure 1. If this function were named `init_drivers`, then it would be declared using the syntax

```
static int __init init_drivers(void)
```

Your code has to identify the module initialization function by using the statement

```
module_init (init_drivers);
```

In *KEY_SW.c* declare global variables that will hold virtual addresses for the KEY and SW ports in the DE1-SoC Computer. Initialize these variables in `init_drivers`, using the kernel function `ioremap_nocache`, as illustrated in the tutorial *Using Linux on DE-series Boards*. Note that you do not need to use interrupts for this part of the exercise.

Write code for the `open`, `release`, and `read` functions for your drivers. For the KEY driver you should read the state of the KEYs from the port's *Edgecapture* register. The programmer registers in the KEY port of the DE1-SoC Computer are illustrated in Figure 4. Return the KEY values to the user as *character* data (ASCII) in the `read` function for the KEY driver. One way to convert binary data into character data is to make use of a library function such as `sprintf`. You could encode the binary data using ASCII in various schemes. For example, you could use one hexadecimal digit to encode the data, and then send the ASCII value of that digit. If the KEY data were equal to `0011`, then you would return the character `3`, and if the data were `1111` you would return `F`. Choose whatever method you prefer for representing this data. For the SW driver, read the slider switch settings from the port's *Data* register, illustrated in Figure 5. Return these values to the user in the form of *character* data, via the driver's `read` function. You can choose to encode the 10 data bits in different ways. For example, if you use hexadecimal and the data were `1100001010`, then you would return the ASCII characters `30A`.

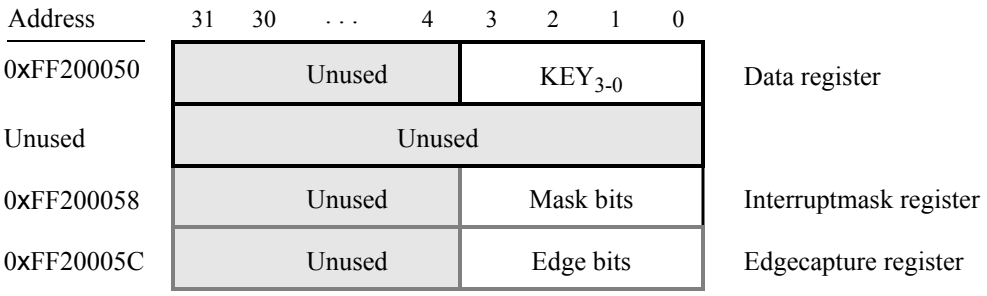

Figure 4: The KEY pushbutton switch port.

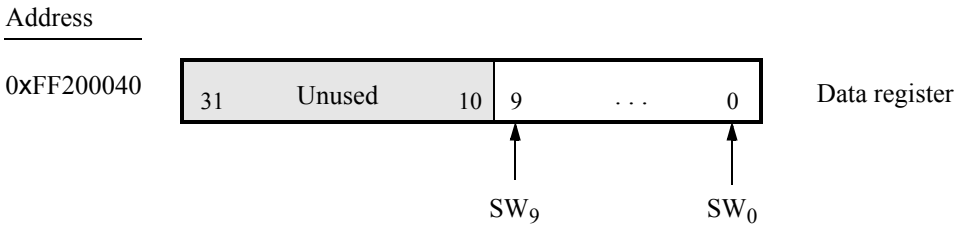

Figure 5: The SW slider switch port.

2. Create a *Makefile* that can be used to compile your kernel module to produce the file *KEY_SW.ko*. Insert this module into the kernel and verify that it creates the character device files */dev/KEY* and */dev/SW*.

3. Test your drivers by using the commands `cat /dev/KEY` and `cat /dev/SW`. Verify that the drivers provide correct values for the switches.

Part III

This part of the exercise assumes that the reader is using the DE1-SoC board, to control its LEDs and seven-segment displays. The same discussion applies for the DE10-Standard board. But if the DE10-Nano board is being used instead, then two differences should be considered: it has fewer LED lights, and it has no seven-segment displays. The reader can still perform the rest of the exercise, but should ignore any parts of the discussion that refer to seven-segment displays.

For this part you are to write another two character device drivers. One driver controls the LEDR lights in the DE1-SoC Computer, via the file */dev/LEDR*. The other driver controls the seven-segment displays HEX5−HEX0, via the file */dev/HEX*. The LEDR port in the DE1-SoC Computer is illustrated in Figure 6, and the seven-segment display ports are depicted in Figure 7. Your driver should be able to display decimal digits $0 - 9$ on each of the six displays. Perform the following:

1. Create a kernel module source-code file called *LEDR_HEX.c*. Implement both character device drivers in this module. For both drivers you should implement functions for `open`, `release`, and `write` operations. These drivers do not require a `read` function.
2. Create a Makefile, compile your module, and insert it into the kernel.
3. Test the LEDR and HEX character device drivers by using the `echo` Linux command. For example, if you designed your LEDR module to accept a three-digit hexadecimal value, then the command `echo 3FF > /dev/LEDR` would turn on all ten LEDs.

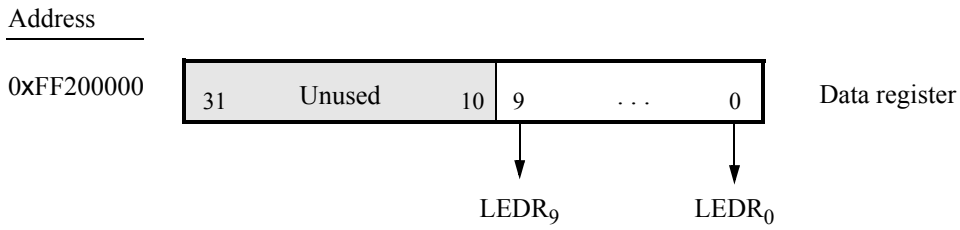

Figure 6: The LEDR red light port.

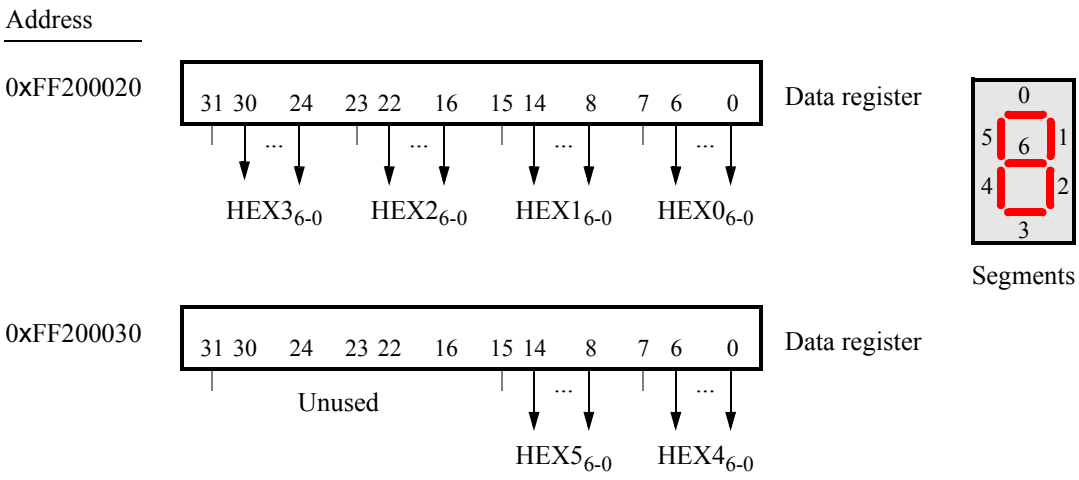

Figure 7: The seven-segment display ports.

Part IV

For this part you are to write a user-level program called *part4.c* that makes use of the character device drivers from Parts II and III. As we mentioned in Part III, if you are using the DE10-Nano board, then ignore the parts of the solution that require seven-segment displays. Perform the following:

1. In your program open the files */dev/KEY* and */dev/SW* for reading, and open the files */dev/LEDR* and */dev/HEX* for writing. Make a loop in your program that does the following. Whenever a KEY is pressed capture the values of the SW switches. Display these values on the LEDR lights. Also, keep a running accumulation of the SW values that have been read, and show this sum on the HEX displays, as a decimal value.

2. Compile your program using a command such as `gcc -Wall -o part4 part4.c`.

3. Run your program and make sure that it works properly. Each time a KEY is pressed, the values of the SW switches should immediately appear on the LEDR lights and the sum should appear on the HEX displays. As an example, if the SW switches are set to 0000000101, then the first time a KEY is pressed 000005 should be shown on the HEX displays, and the second time a KEY is pressed 000010 should be displayed.

Copyright © FPGAcademy.org. All rights reserved. FPGAcademy and the FPGAcademy logo are trademarks of FPGAcademy.org. This document is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the document or the use or other dealings in the document.

*Other names and brands may be claimed as the property of others.