Embedded Systems
Laboratory Exercise 7

Using the ADXL345 Accelerometer

The purpose of this exercise is to experiment with the ADXL345 accelerometer chip that is included on the DE1-SoC board. You will add support in the Linux kernel for the ADXL345 chip by creating a character device driver.

Part I

To learn about the operation of the ADXL345 accelerometer chip, read the tutorial called *Using the Accelerometer in DE-SoC Boards*. This tutorial is provided as part of the design files that accompany this exercise. We assume that you are using the DE1-SoC board, but the procedures for using the accelerometer are the same for other boards, such as the DE10-Standard and DE10-Nano. An example program that demonstrates use of the accelerometer is included with the tutorial. It assumes that no operating system is being used (i.e., a "bare metal" environment), and therefore uses simple dereferencing of pointers to communicate with the device. But, since your code needs to run under Linux on the DE1-SoC Computer, you will have to map the required physical addresses into virtual addresses.

Perform the following:

1. Read through the tutorial *Using the Accelerometer in DE-SoC Boards*. Create a new version, which uses virtual addresses and can run under Linux, of the C program that accompanies the tutorial.
2. In your program, configure the ADXL345 device to provide an output data rate of 12.5 Hz, using a fixed 10-bit resolution and +/- 16 g (*gravity*) range. These settings provide an appropriate level of sensitivity in the ADXL345 device, of about 31 mg per least-significant bit.
3. Use a Linux Terminal to compile and run your program. Tilt the DE1-SoC board in various directions and observe the acceleration data that is printed by your program. Since the z axis of the accelerometer chip on the DE1-SoC board is aligned with earth's gravity, your program should read acceleration of approximately 1,000 mg in the z direction when the DE1-SoC board is stationary. You should end up with a program that prints out acceleration data in the format `<X-axis acceleration in mg>, <Y-axis acceleration in mg>, <Z-axis acceleration in mg>`.

Part II

In Part I you wrote a user-level program to directly read/write registers in the ADXL345 device by using memory-mapped I/O. In this part, you will use a different approach, by adding support to the Linux kernel for accessing the device. You are to create a character device driver that provides an interface to the accelerometer. The driver must create the file */dev/accel* in the Linux filesystem. A read of this file should return accelerometer data in the format `R XXXX YYYY ZZZZ SS`, where *R* is 1 if new accelerometer data is being provided, *XXXX*, *YYYY*, and *ZZZZ* are acceleration data in the x, y, and z axes, and *SS* is the scale factor in mg/LSB for the acceleration data. As an example, if the ADXL345 has new data to report, then a read of the file might return: "1 0 -1 32 31", which would represent 0 mg acceleration in the x axis, -31 mg acceleration in the y axis, and 992 mg acceleration in the z axis. If you were to perform another read from */dev/accel* immediately, then the device might not be ready to provide new data; it would then respond with "0 0 -1 32 31", indicating old data.

An outline of the required code for the accelerometer device driver is given in Figure 1. Lines 1 to 7 include header files that are needed for the driver. Global variables that are used to access the accelerometer, which will be described later, are declared in lines 10 and 11. Line 14 is a placeholder for the declarations of function prototypes

and variables that are needed for the character device driver. Prototypes have to be declared for the functions that are executed when opening, reading, and closing the driver. A variable of type `miscdevice` has to be declared and initialized in the function *start_accel*, shown in Lines 17 to 31, which is executed when the accelerometer driver is inserted into the Linux kernel. Refer to Exercise 3 for a more detailed discussion about the functions and variables that are needed for character device drivers.

To provide the kernel module with access to the *I2C controller*, which is used to communicate with the ADXL345 device, line 22 calls the `ioremap_nocache` function. Information about this function can be found in the tutorial *Using Linux on the DE1-SoC*. Line 23 computes a virtual address for the *system manager*, which has to be configured for use with the ADXL345 device. The next three lines call the `Pinmux_Config`, `I2C0_Init`, and `ADXL345_Init` functions. You should be able to reuse the code that you wrote for these functions in Part I of this exercise. You should initialize the ADXL345 with the same settings as for Part I.

Line 34 is a placeholder for the various functions that are needed to read and write to registers in the ADXL345 device. You should be able to reuse the code from Part I for these functions. The remainder of the code in Figure 1 refers to the functions that are needed to open, close, and read from the character device driver. A detailed discussion relating to these functions can be found in Laboratory Exercise 3.

Perform the following:

1. Create a file named *accel.c* and fill in the missing code from Figure 1. Using a Makefile, compile your code to create the kernel module *accel.ko* and insert this module into the Linux kernel.
2. Test your device driver by executing a command such as `cat /dev/accel` using a Linux Terminal window. Try tilting the board in various directions and confirm that the driver provides appropriate results each time that you read from */dev/accel*.
3. Once you have confirmed correct operation of your character device driver, write a user-level program called *part2.c* that uses the driver. A skeleton of an example program is given in Figure 2. Fill in the rest of the code so that it performs some operations using the device driver. For example, you could write a loop that continuously reads from the device and prints the results whenever new data is available. Compile your program using a command such as `gcc -Wall -o part2 part2.c`, and test the program.

Part III

In Part II your character device driver supports only read operations. For this part, you are to add the following commands that can be written to your device driver:

- `device`: prints on the Terminal (using *printk*) the ADXL345 device ID.
- `init`: re-initializes the ADXL345
- `calibrate`: executes a calibration routine
- `format F G`: sets the data format to fixed 10-bit resolution ($F = 0$), or full resolution ($F = 1$), with range G = +/- $2, 4, 8$, or 16 g
- `rate R`: sets the output data rate to R Hz. Your code should support a few examples of data rates, such as 25 Hz, 12.5 Hz, and so on.

Perform the following:

1. Augment your device-driver code from Part II to support *write* operations, and implement the commands listed above.

```
1  #include <linux/module.h>
2  #include <linux/kernel.h>
3  #include <linux/fs.h>
4  #include <linux/cdev.h>
5  #include <linux/device.h>
6  #include <asm/io.h>
7  #include <asm/uaccess.h>
8
9  // Declare global variables needed to use the accelerometer
10 volatile int * I2C0_ptr;   // virtual address for I2C communication
11 volatile int * SYSMGR_ptr; // virtual address for System Manager communication
12
13 // Declare other variables and prototypes needed for a character device driver
14 ... code not shown
15
16 /* Code to initialize the accelerometer driver */
17 static int __init start_accel(void) {
18    // initialize the miscdevice and other data structures
19    ... code not shown
20
21    // generate virtual addresses
22    I2C0_ptr = ioremap_nocache (0xFFC04000, 0x00000100);
23    SYSMGR_ptr = ioremap_nocache (0xFFD08000, 0x00000800);
24    if ((I2C0_ptr == 0) || (SYSMGR_ptr == NULL))
25       printk (KERN_ERR "Error: ioremap_nocache returned NULL\n");
26
27    Pinmux_Config ();
28    I2C0_Init ();
29    ADXL345_Init ();
30    return 0;
31 }
32
33 // Functions needed to read/write registers in the ADXL345 device
34 ... code not shown
35
36 static void __exit stop_accel(void) {
37    /* unmap the physical-to-virtual mappings */
38    iounmap (I2C0_ptr);
39    iounmap (SYSMGR_ptr);
40
41    /* Remove the device from the kernel */
42    ... code not shown
43 }
44
45 // Code for device_open and device_release
46 ... code not shown
47
48 static ssize_t device_read(struct file *filp, char *buffer, size_t length,
       loff_t *offset) {
49    ... code not shown
50 }
```

Figure 1: An outline of the ADXL345 device driver code.

2. Write a user-level program, called *part3.c*, that uses your character device driver. Implement a simple graphical demo that draws a "bubble" on the screen. When the DE1-SoC board is stationary the bubble should appear in the middle of the screen. Tilting the board right, left, up, or down should cause the bubble to "move" accordingly. Also, print in the top-left of the screen the values of x, y, and z acceleration each time new data is available from your device driver.

3. To implement graphics, you can use either ASCII commands in a Terminal window, or you can connect a VGA monitor to the DE1-SoC board and use VGA graphics. ASCII and VGA graphics were described in Lab Exercises 5 and 6, respectively.

4. Experiment by writing different commands to */dev/accel* and examining the effect on your graphics display. Depending on the data format that you set in the accelerometer, the graphics displayed by your program may appear "jittery". To ameliorate this effect, you can compute a running average of the acceleration data. For example, to compute an average acceleration for the x-axis, an appropriate calculation could be

   $av\_x = av\_x * \alpha + accel\_x * (1 - \alpha)$

   Here, *av_x* is the average acceleration, and *accel_x* is the most recent x-axis data provided by the ADXL345. Smaller/larger values of α can be used to decrease/increase the effect of new data on the average.

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

// number of characters to read from /dev/accel
#define accel_BYTES 20

int main(int argc, char *argv[])
{
   int accel_FD;                  // file descriptor
   char accel_buffer[accel_BYTES];     // buffer for accel char data
   // Declare other variables
   ... code not shown

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

   // Perform some operations using the ADXL345 device
   ... code not shown

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

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

Part IV

In addition to providing acceleration data, the ADXL345 device also supports "tap" and "double-tap" detection. Perform the following:

1. Add tap and double-tap detection to your device-driver code from Part III. To learn how to configure the accelerometer for tap detection, consult the product *Data Sheet*, which is available on Analog Device's website (*www.analog.com*). Some recommended tap-detection settings are

Tap threshold:	3 g
Maximum tap duration:	0.02 seconds
Tap latency:	0.02 seconds
Double-tap window:	0.3 seconds

2. Modify the read function for your character device driver so that it returns data in the format `HH XXXX YYYY ZZZZ SS`. Here, *HH* is a two-digit hexadecimal value corresponding to the contents of the ADXL345 *Sources of Interrupt* register. In addition to reporting when new acceleration data is available this register also indicates when tap or double-tap actions have been detected.

3. Write a user-level program, called *part4.c*, that demonstrates the detection of tap and double-tap actions. One possibility is to augment the user-level program from Part III such that the animation displayed changes based on whether a tap or double-tap has occurred.