Lab Assignment 3
Memory-Mapped I/O and Object-Oriented Programming

Lab 3.0 Introduction

In this lab we introduce the concept of memory-mapped I/O to access devices available on the ZedBoard, including the LEDs, the switches, and the push buttons. Starting with a simple program given in this lab manual that controls the state of the LEDs, we will inspect the state of the other input devices, and make them all interact. Finally, we will use object-oriented programming to abstract some of the functionality related with memory-mapped I/O into a C++ class providing additional data protection and modularity.

Lab 3.1 Becoming *root* on the ZedBoard

As we studied in class, UNIX operating systems use a file system where the currently logged in user has a dedicated home folder with full read and write privileges. Outside of this folder, however, the operating system limits access permissions in order to guarantee the system's integrity and security. This lab assignment involves certain I/O-related actions on the ZedBoard that require a higher privilege level than that granted to a regular user. This will force you to manually upgrade your privilege level every time you run your programs on the ZedBoard.

In Linux, a special user called *root*, present in every system, acts as the system administrator. In general, it is not a good practice to work with *root* privileges even if you own the *root* account, since critical system settings could be accidentally affected if the wrong command is typed in the shell.

To avoid working as *root*, a special group of users exists called the *sudoers*. A regular user that belongs to this group is allowed to carry out system administration tasks by prepending the sudo command to any command that requires additional privileges, but will keep restricted permissions otherwise.

An example of an action unauthorized for a regular user is displaying the content of file /etc/shadow, which contains a list of encrypted passwords for all users in the system. You can try the following command:

```
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
```

Since we can access the ZedBoard with a root account, you can use the following steps to add yourself as a *sudoer* in the system:

- Log in as root with MobaXTerm. The password is *root* (yes, not very secure).
- Run the following command to change the permissions of executable file sudo, allowing you to access it once you log out of the *root* account:

  ```
  $ chmod 4755 /usr/bin/sudo
  ```

- Add your user name to the *sudoers* group with the following command, replacing <user> by your user name on the ZedBoard:

```
$ adduser <user> sudo
```

- Now log out from the *root* account, and log back in with your user name. If you run the following command, you should be able to perform privileged actions on the machine. When you're prompted for a *sudo* password, just enter your user password.

```
$ sudo cat /etc/shadow
```

Lab 3.2 Controlling the LEDs

The following program is our first attempt to control the devices present in the ZedBoard, starting with the LEDs. This program asks the user to enter a number on the keyboard between 0 and 255, and represents that number in binary (base 2) using the 8 available LEDs. An LED will light up if the associated digit in the binary number is set to 1.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

// Physical base address of GPIO
const unsigned gpio_address = 0x400d0000;

// Length of memory-mapped IO window
const unsigned gpio_size = 0xff;

const int gpio_led1_offset = 0x12C;  // Offset for LED1
const int gpio_led2_offset = 0x130;  // Offset for LED2
const int gpio_led3_offset = 0x134;  // Offset for LED3
const int gpio_led4_offset = 0x138;  // Offset for LED4
const int gpio_led5_offset = 0x13C;  // Offset for LED5
const int gpio_led6_offset = 0x140;  // Offset for LED6
const int gpio_led7_offset = 0x144;  // Offset for LED7
const int gpio_led8_offset = 0x148;  // Offset for LED8

const int gpio_sw1_offset = 0x14C;  // Offset for Switch 1
const int gpio_sw2_offset = 0x150;  // Offset for Switch 2
const int gpio_sw3_offset = 0x154;  // Offset for Switch 3
const int gpio_sw4_offset = 0x158;  // Offset for Switch 4
const int gpio_sw5_offset = 0x15C;  // Offset for Switch 5
const int gpio_sw6_offset = 0x160;  // Offset for Switch 6
const int gpio_sw7_offset = 0x164;  // Offset for Switch 7
const int gpio_sw8_offset = 0x168;  // Offset for Switch 8

const int gpio_pbtnl_offset = 0x16C;  // Offset for left push button
const int gpio_pbtnr_offset = 0x170;  // Offset for right push button
const int gpio_pbtnu_offset = 0x174;  // Offset for up push button
const int gpio_pbtnd_offset = 0x178;  // Offset for down push button
const int gpio_pbtnc_offset = 0x17C;  // Offset for center push button

/**
 * Write a 4-byte value at the specified general-purpose I/O location.
 *
 * @param pBase         Base address returned by 'mmap'.
 * @parem offset   Offset where device is mapped.
 * @param value    Value to be written.
 */
void RegisterWrite(char *pBase, int offset, int value)
{
     * (int *) (pBase + offset) = value;
}

/**
 * Read a 4-byte value from the specified general-purpose I/O location.
 *
 * @param pBase         Base address returned by 'mmap'.
 * @param offset   Offset where device is mapped.
 * @return         Value read.
 */
int RegisterRead(char *pBase, int offset)
{
     return * (int *) (pBase + offset);
}

/**
 * Initialize general-purpose I/O
 *  - Opens access to physical memory /dev/mem
 *  - Maps memory at offset 'gpio_address' into virtual address space
 *
 * @param fd  File descriptor passed by reference, where the result
 *            of function 'open' will be stored.
 * @return    Address to virtual memory which is mapped to physical,
 *            or MAP_FAILED on error.
 */
char *Initialize(int *fd)
{
     *fd = open( "/dev/mem", O_RDWR);
     return (char *) mmap(
               NULL,
               gpio_size,
               PROT_READ | PROT_WRITE,
               MAP_SHARED,
               *fd,
               gpio_address);
}

/**
 * Close general-purpose I/O.
 *
 * @param pBase    Virtual address where I/O was mapped.
 * @param fd  File descriptor previously returned by 'open'.
 */
void Finalize(char *pBase, int fd)
{
     munmap(pBase, gpio_size);
     close(fd);
}
```

```
/**
 * Show lower 8 bits of integer value on LEDs
 *
 * @param pBase    Base address of I/O
 * @param value    Value to show on LEDs
 */
void SetLedNumber(char *pBase, int value)
{
     RegisterWrite(pBase, gpio_led1_offset, value % 2);
     RegisterWrite(pBase, gpio_led2_offset, (value / 2) % 2);
     RegisterWrite(pBase, gpio_led3_offset, (value / 4) % 2);
     RegisterWrite(pBase, gpio_led4_offset, (value / 8) % 2);
     RegisterWrite(pBase, gpio_led5_offset, (value / 16) % 2);
     RegisterWrite(pBase, gpio_led6_offset, (value / 32) % 2);
     RegisterWrite(pBase, gpio_led7_offset, (value / 64) % 2);
     RegisterWrite(pBase, gpio_led8_offset, (value / 128) % 2);
}

int main()
{
     // Initialize
     int fd;
     char *pBase = Initialize(&fd);

     // Check error
     if (pBase == MAP_FAILED)
     {
          perror("Mapping I/O memory failed - Did you run with 'sudo'?\n");
          return -1;
     }

     int value = 0;
     printf("Enter a value less than 256: ");
     scanf("%d", &value);
     printf("value = %d\n", value);

     // Show the value on the Zedboard LEDs
     SetLedNumber(pBase, value);

     // Done
     Finalize(pBase, fd);
     return 0;
}
```

The code given above uses a technique called memory-mapped I/O in order to access devices. This technique consists in accessing a virtual file representing a set of I/O devices using the open() system call, and then mapping a set of control flags into memory locations using function mmap(), which lets you modify or read the device state by just dereferencing a pointer.

Pre-lab assignment

Before this lab session, please prepare the following material:

a) On the COE Linux machines, copy and paste the code in a file named ~/Lab3/LedNumber.c. Compile it and then run it with command ./LedNumber. Since obviously the COE machines do not provide the same hardware environment as the ZedBoards, the program is expected to fail. Where does it fail? Why? Copy and paste the output or errors you get and your response in your PDF turn in.

b) Using the UNIX manual pages (man), as well any necessary online documentation, explain the code in functions Initialize(), Finalize(), RegisterRead(), RegisterWrite() and SetLedNumber().

Submit your PDF pre-lab on Blackboard before coming to the lab.

Lab Assignment

- Copy and paste this code into a file named LedNumber.c on the ZedBoard (or download it from Blackboard), created in a new directory named Lab3.
- Compile your program and run it on the ZedBoard. You need privileged access in order to access the ZedBoard's I/O devices, so you need to prepend the sudo command to the name of the executable file:

```
$ sudo ./LedNumber
```

- Run your program at least three times, entering different numbers in each execution. Before continuing with the following lab assignments, verify that the LEDs represent the proper binary number entered as the program input. Congratulations, you have written your first piece of software controlling a device on the ZedBoard!

Lab 3.3 Controlling individual LEDs

Our next goal is controlling the state of each separate LED, without affecting the state of the rest of the LEDs each time.

- On the ZedBoard, copy file LedNumber.c into a new file called LedOnOff.c, still in directory Lab3. Modify the code in such a way that the program asks the user for an LED identifier (a number between 0 and 7) and an LED state, where 0 means *off* and 1 means *on*. The program should turn on or off the given LED as per the entered state, and then exit.

- Encapsulate the code that modifies the LED state in a function with the following prototype, and behavior, described here using comments in Doxygen format:

```
/** Set the state of the LED with the given index.
 *
 * @param pBase         Base address for general-purpose I/O
 * @parem led_index   LED index between 0 and 7
 * @param state       Turn on (1) or off (0)
 */
void SetLedState(void *pBase, int led_index, int state);
```

Assignment 2

Compile and run your program on the ZedBoard, and verify that its behavior is correct. Test your code for different values for the LED index and state. In your report, copy and paste the content of file LedOnOff.c, and describe the combinations of inputs tested during the execution of your program.

Lab 3.4 Controlling the switches

We will now introduce the switches, as probably the simplest type of input device on the ZedBoard. The state of the switches can be read using function RegisterRead() function with the appropriate offset chosen from the list of constants at the beginning of the source code in LedNumber.c (gpio_sw1_offset and following).

A switch is considered to be in the *on* position when it is closest to the LED underneath it. In this case, a value equal to 1 can be read at the corresponding memory location in the memory-mapped I/O region.

- Copy file LedNumber.c into a new file called SwitchToLed.c in directory Lab3.
- Modify the content of the new file with a main program that reads the values for all switches and sets the same value for the LED located right underneath it. In order for this program to respond to any changes in the switches and reflect it in the LEDs in real time, it needs to run in an infinite loop, where the switch-to-LED operation is repeated in each iteration.
- As the program running an infinite loop will not terminate, you can interrupt its execution by pressing *Control+C*. This combination of keys sends an interruption signal to the running program. The program immediately terminates and returns control to the shell.

Assignment 3

Compile your new program and test it on the ZedBoard.

a) List the content of file SwitchToLed.c in your report.

b) Demonstrate the implemented features to the instructor or the TA. They will check you off on completion of this part.

Lab 3.5 Controlling the push buttons

Push buttons are another simple kind of input devices available on the ZedBoard. Placing the ZedBoard with the power connector facing forward, the push buttons are located in the bottom-right corner. There are five push buttons identified as follows:

- PBTNU (up)
- PBTND (down)
- PBTNR (right)
- PBTNL (left)
- PBTNC (center)

Write a program that interprets the value of the switches as a binary number for a counter. Initially, the program should reflect the value of the counter in the state of the LEDs, using part of the functionality implemented in previous steps. When the user presses the *up* button, the counter should be incremented by 1, and this change should be reflected right away into the LEDs. When the user presses the *down* button, the counter should be decremented by one. When the user presses the *right* button, the current count should be shifted right one bit position, inserting one 0 on the left (e.g., 00010111 → 00001011). When the user presses the *left* button, the current count should be shifted left one bit position, inserting a 0 on the right (e.g., 00010111 → 00101110). Finally, when the user presses the *center* button, the counter should be reset to the value specified in the switches.

- Copy file LedNumber.c into a new file called PushButton.c in the same directory.

- Write a function named PushButtonGet() that returns 0 if no push button is pressed, and a value between 1 and 5 if any push button is pressed. Each number identifies a particular push button (1 = *up*, 2 = *down*, etc).

- Notice that reading the state of the push buttons is tricky because of debouncing on the push buttons. You want your program to increment or decrement the value of a counter only once for each time a push button is pressed, that is, when the PushButtonGet() function changes its return value, and not necessarily every time PushButtonGet() returns a value other than 0. In order to deal with this situation, you will need to store the current state of the push buttons, and only act if you detect a difference in this state.

Assignment 4

Compile and test your program.

a) List the content of file PushButton.c in your report.

b) Demonstrate the implemented features to the instructor or the TA. Make sure that you demonstrate all features, including incrementing the counter, decrementing it, and resetting it to the values given by the switches. In the absence of the instructor or TA, record a short video that illustrates the correct behavior of the platform.

Lab 3.6 Using C++ objects

In the last part of this lab, we will use object-oriented programming to abstract the functionality related with I/O operations on the ZedBoard, including the initialization and finalization of the I/O memory maps, and the read or write operations on I/O memory locations.

We will do this with a new C++ class called ZedBoard, whose constructor and destructor will take care of initialization and finalization operations, respectively—those implemented in functions Initialize() and Finalize() in the previous C code. In C, these functions shared information through arguments and return values, including the base pointer to the I/O memory (variable pBase) and the file descriptor of the virtual device file (variable fd). These two variables should now be private members of class ZedBoard.

Functions ReadRegister() and WriteRegister() also took the base pointer pBase as the first argument. Now, these functions will also be part of class ZedBoard, and will read the base pointer from class member pBase. This approach avoids having to pass this argument around.

Your class should look like this:

```
class ZedBoard
{
     char *pBase;
     int fd;

public:

     ZedBoard()
     {
          ...
     }

     ~ZedBoard()
     {
          ...
     }

     void RegisterWrite(int offset, int value)
     {
          ...
     }

     int RegisterRead(int offset)
     {
          ...
     }
};
```

- Copy your program PushButton.c into a new file named PushButtonCpp.cpp located in directory Lab3.

- Modify the code in PushButtonCpp.cpp to convert it into a proper C++ program with the suggestions provided above. You can compile the new C++ program with the following command:

```
$ g++ PushButtonCpp.cpp -o PushButtonCpp -Wall -Werror
```

Assignment 5

Compile and test your program. List the content of file PushButtonCpp.cpp in your report.

Lab 3.7 Counter with speed control

As an optional assignment, design an automatic counter whose direction and speed can be controlled with the push buttons, using a C++ implementation based on the code given in the previous assignment. The effect of the push buttons in the counter should be the following:

- Initially, the speed of the counter is 0 ticks per second, that is, the counter does not change its value automatically.
- When the *center* push button is pressed, the LEDs should load the value represented by the current state of the switches.
- When the *up* push button is pressed, the counting speed is incremented by 1 tick per second. When this button is pressed for the first time, the counting speed transitions from 0 to 1 tick per second, that is, it starts incrementing its value periodically without any action from the user. Pressing *up* multiple times makes the counter increase at a faster pace.
- When the *down* button is pressed, the speed of the automatic increments is reduced by 1 tick per second. If the current speed is 1 tick per second and *down* is pressed again, the new speed becomes 0 again, and the counter halts.
- The *right* and *left* buttons should control the counting direction. Initially, the value of the counter goes up by 1 at each tick. Pressing the *left* button should change the direction to make the counter count down instead. Pressing the *right* button should make it count up again, always at the corresponding speed.

Extra credit

Copy your C++ implementation of PushButtonCpp.cpp into a new file called CounterSpeed.cpp in directory Lab3, and extend it to provide the functionality presented above.

a) List the code of file CounterSpeed.cpp in your lab report.
b) Demonstrate the implemented features to the instructor or the TA, or record a video with a maximum duration of 20 seconds where you demonstrate the correct behavior of the system, including direction control, speed control, and loading new values from the switches. Attach a file named extra.mov to your submission.

Lab Report

Submit a complete formal lab report. You should follow the lab report outline provided on Blackboard. Your report should be developed on a word processor (e.g., OpenOffice or LaTeX), and should include graphics when trying to present a large amount of data. Include the output of compiling and running your programs. Upload the lab report on blackboard. Attach a listing of each program that you wrote in the appendix to your lab report.