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

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 3.0 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
```

Your username has been added to the *sudoers* group. If you run the command above, preceded by the `sudo` 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
```

Controlling the LEDs

The following program is our first attempt to control the devices present in the ZedBoard, starting with the LEDs. We will write a program that controls the LEDs, switches, and push buttons using memory mapped I/O. Study the program below and complete the pre-lab.

```
#include <iostream>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
using namespace std;

// 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);
}


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

    // Check error
    if (pBase == MAP_FAILED)
    {
         cerr << "Mapping I/O memory failed - Did you run with 'sudo'?\n";
         exit(1); // Returns 1 to the operating system;
    }

    // ************** Put your code here **********************


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

The code given above uses a technique called memory-mapped I/O in order to access devices. This technique consists of 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) Using the UNIX manual pages (man), as well any necessary online documentation, explain the code in functions Initialize(), Finalize(), RegisterRead(), and RegisterWrite().

b) Write a function for controlling each LED individually. The function should have the interface below: The function calls `RegisterWrite(…)` to write the state to the specified LED.

```
/** Changes the state of an LED (ON or OFF)
*  @param pBase     base address of I/O
*  @param ledNum    LED number (0 to 7)
*  @param state     State to change to (ON or OFF)
*/
void Write1Led(char *pBase, int ledNum, int state);
```

In this function *ledNum* selects LED to control (0 … 7) and *state* controls if LED is off (0), or on (1). For example:

`Write1Led(pBase, 0, 1);` Should turn on LED 0.

Hints: See the definition of each LEDs offset address (*#define* at top of the file). What is the address distance between LED offsets, are they constant? Can you compute an offset based on an LED number (0 … 7)? Compute first the offset for the desired LED and store it in variable *int ledOffset*.

c) Write a function analogous to the one above but this time the function should read a switch instead. The function should have the interface below. In this function *switchNum* selects the switch to read from (0 … 7). The function calls `RegisterRead(…)` to read the specified switch and returns the value read.

```
/** Reads the value of a switch
 *  - Uses base address of I/O
 * @param pBase     base address of I/O
 * @param switchNum Switch number (0 to 7)
 * @return          Switch value read
 */
int Read1Switch(char *pBase, int switchNum);
```

Submit a single PDF file with your responses and pre-lab code on Blackboard before coming to the lab.

Lab 3.1 Lab Assignment

- Copy and paste the code above into a file named LedNumber.cpp on the ZedBoard (or download it from Blackboard), created in a new directory named Lab3.

- Copy and paste your `Write1Led(…)` and `Read1Switch(…)` functions from the pre-lab and confirm their functionality from the main function.

    - In the main function, implement code to ask the user to enter an LED number (0 to 7) and the state (0 or 1) to turn on or off any LED. Confirm the correct behavior of `Write1Led(…)`.

    - Similarly, implement code to ask the user to enter a switch number (0 to 7) to read that switch. Confirm the correct behavior of `Read1Switch(…)`.

- 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
```

Assignment 1

Run your program at least three times, entering different numbers for LEDs and switches for each test. Congratulations, you have written your first piece of software controlling a device on the ZedBoard!

Lab 3.2 Controlling all LEDs

Our next goal is to write a function that writes to all LEDs. This function receives, as arguments, the base address `pBase` and an integer `value` between 0 and 255 that represents the decimal equivalent of the eight LEDs. The function extracts the individual bits from the value and writes them to the corresponding LEDs using the `Write1Led(…)` function tested above.

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

```
/** Set the state of the LEDs with the given value.
 *
 * @param pBase         Base address for general-purpose I/O
 * @param value         Value between 0 and 255 written to the LEDs
 */
void WriteAllLeds(void *pBase, int value);
```

Assignment 2

Compile and run your program on the ZedBoard, and verify that its behavior is correct from the main function. Test your code for different values between 0 and 255. In your report, copy and paste the contents of the function `WriteAllLeds(…)`, and describe the combinations of inputs tested during the execution of your program.

Lab 3.3 Controlling all the switches

Our next goal is to write a function that reads all the switches. This function receives, as argument, the base address `pBase`, and returns an integer `value` between 0 and 255 that represents the decimal equivalent of the value of the switches. The function uses the `Read1Switch(…)` function tested above to read each switch in turn and concatenates the values into a single integer.

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

```
/** Reads all the switches and returns their value in a single integer.
 *
 * @param pBase         Base address for general-purpose I/O
 * @return              A value that represents the value of the switches
 */
int ReadAllSwitches(void *pBase);
```

Assignment 3

Compile and run your program on the ZedBoard, and verify that its behavior is correct from the main function. Test your code for different settings of the switches. In your report, copy and paste the contents of the function `ReadAllSwitches(…)`, and describe the combinations of switch settings tested during the execution of your program.

Lab 3.4 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.cpp into a new file called PushButton.cpp 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.cpp 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 3.5 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, the 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.cpp into a new file named PushButtonClass.cpp located in directory Lab3.

- Modify the code in PushButtonClass.cpp to convert it into a proper C++ object oriented program with the suggestions provided above.

Assignment 5

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

Lab 3.6 Counter with speed control – Extra Credit

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 PushButtonClass.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.