Lab 7

Memory-Mapped I/O and Controlling HW with SW

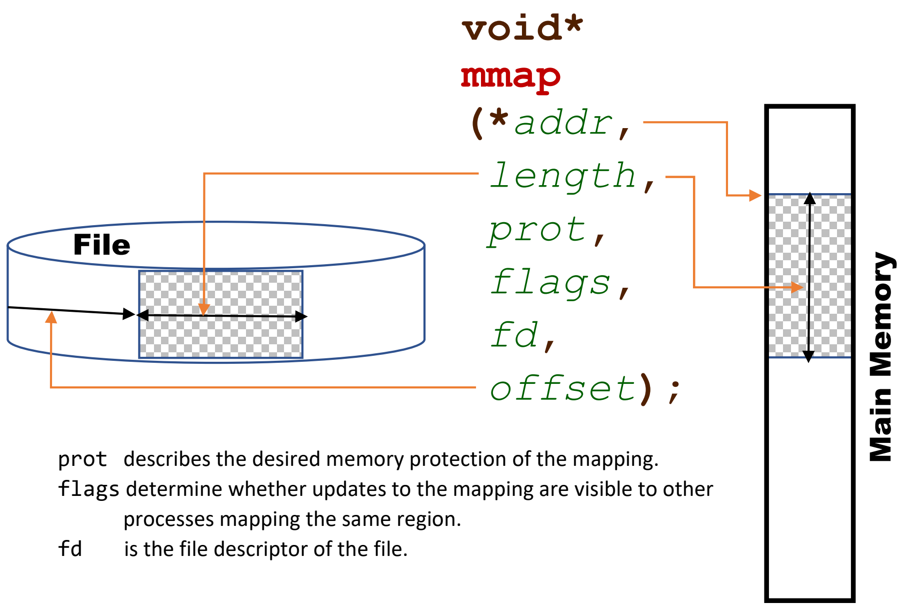
1. Introduction

In this lab we introduce the concept of memory mapped I/O to access devices available on the DE1-SoC, including the LEDs, the switches, and the push buttons. In later labs, we will also use MMIO to control the spider robot's servos. 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.

2. Memory Mapping

A processor can communicate with Input/Output devices (e.g., keyboard, monitor) the same way it communicates with the memory (i.e., through *Load* (read) and *Store* (write) instructions). In this case a portion of the memory "address" space is reserved for I/O devices rather than the regular memory that is used to store running programs' data and instructions. This method of communicating with I/O devices is called memory mapped I/O. The (now less common) alternative is for processors to use dedicated instructions to access I/O devices.

Memory mapping is a fast and efficient method to **securely** allow programs to access physical devices. Normally, the Operating System oversees physical devices, and every I/O access must go through the OS (via a **system call**). Unfortunately, system calls are very slow operations. Memory mapping makes an I/O device directly accessible as a file in the program's address space (I.e., memory addresses that it is allowed to read and write without system calls). In this way, the program only needs **one system call** to set up the mapping, and thereafter can access the I/O device directly. The standard Linux function for establishing a memory map is called mmap(). The following figure illustrates how mmap() works.


void*
mmap
(*addr,
length,
prot,
flags,
fd,
offset);
File
Main Memory
prot describes the desired memory protection of the mapping.
flags determine whether updates to the mapping are visible to other processes mapping the same region.
fd is the file descriptor of the file.

The addr parameter is a pointer to *where* the application would like the device to be mapped; if NULL is passed, then the kernel chooses the address at which to create the mapping.

The other important parameter is fd, which stands for *file descriptor* and is the C/C++ representation of a particular file. In this lab, we will memory map the file /dev/mem, which is a special file that gives us access to the physical memory on our DE1-SoC board. Part of this memory space contains the addresses of the board I/O device registers. We will use the mmap() function to map the device registers into the user address space so that we can read/write them with pointers.

2.1 Address of DE1-SoC LEDs, Switches, and Push Button

The following diagrams are taken from the **DE1-SoC Computer System with ARM Cortex A9** document (pp. 7-9) on Canvas:

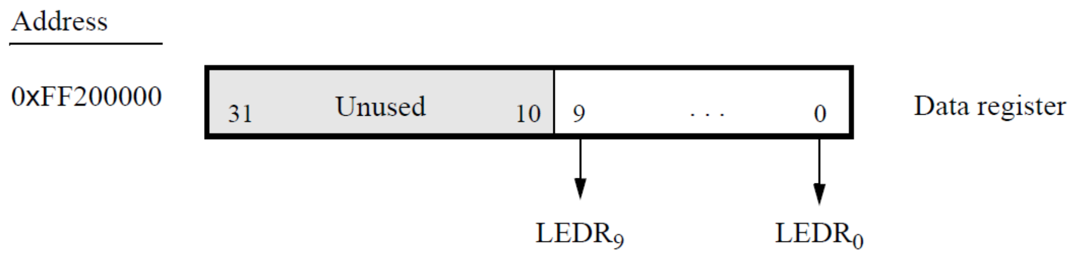


Figure 1: Output parallel port for LEDR

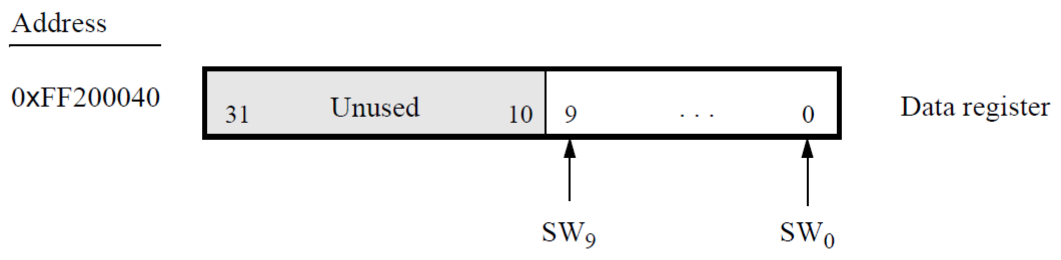


Figure 2: Data register in the slider switch parallel port

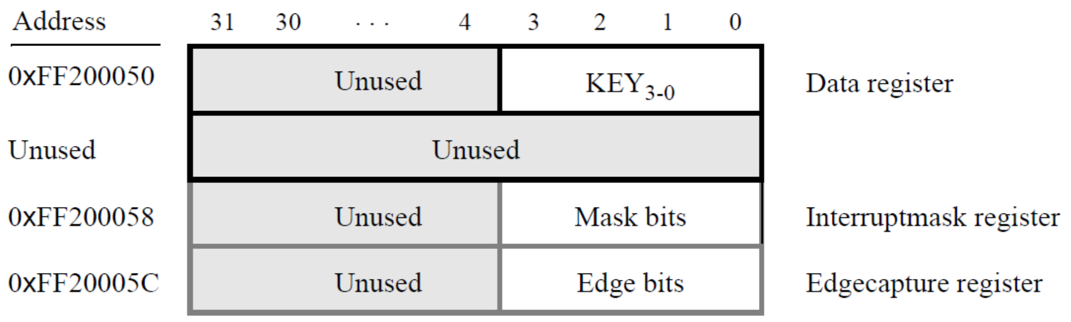


Figure 3: Registers used in the pushbutton parallel port. Note: For this lab, we will only use the Data register to determine the push button pressed.

3. Interfacing with the LEDs and Switches

Study the code **LedNumber.cpp** which is available on Canvas. The program prompts the user for an integer; if the integer is –1 then all LEDS will be set to the value of their respective switches; if the integer is between 0 and 9 then *only the LED with that index* should be modified to reflect the state of the switch with the same index.

Make sure you understand all of the functions in the code.

1. char *Initialize(int *fd)
2. void Finalize(char *pBase, int fd)
3. void RegisterWrite(char *pBase, unsigned int reg_offset, int value)
4. int RegisterRead(char *pBase, unsigned int reg_offset)
5. int ReadAllSwitches(char *pBase)
6. int Read1Switch(char *pBase, int switchNum)

7. void WriteAllLeds(char *pBase, int value)
8. void Write1Led(char *pBase, int ledNum, int state)
9. int main()

Compile and run the code on the DE1 board to make sure it works. **Show it to your TA or instructor.**

4. Interfacing with the Push Buttons

Push buttons are another simple kind of input device available on the DE1-SoC. There are four push buttons to the right of the switches identified as follows:

- KEY3, KEY2, KEY1, KEY0

Write a program that:

- Interprets the current value of the switches as the initial value of a counter.
- The program starts by reflecting this counter value on the LEDs, using the functions you implemented in the previous assignments.
- **All changes to the counter value** should be reflected immediately on the LEDs so that the LEDs always display the current binary value of our counter.
- When the user presses the *KEY0* button,
  - The current counter value should be incremented by 1.
- When the user presses the *KEY1* button,
  - The counter should be decremented by 1.
- When the user presses the *KEY2* 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 *KEY3* 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 multiple buttons, the counter should be reset to the value specified by the current state of the switches.

Perform the following tasks:

- Copy file LedNumber.cpp into a new file called PushButton.cpp in the same directory.
- Write a function named PushButtonGet(…) that **returns -1** if no push button is pressed, and a

value between 0 and 3 identifying the push button pressed. If multiple push buttons (any two or more) are pressed it should return a value of 4.

- Implement the **main** program described above.
- *Hint 1: To detect a* button press *you will need to keep track of the state of the buttons the **LAST TIME** that you checked them. If a button is held down, then PushButtonGet will repeatedly return the same value; this **does not** mean you should repeatedly update the counter value, it should only be updated **once per press**.*

Hint 2: To keep the program running, use an infinite loop in your main function.

Assignment 1

Compile and test your program. Demonstrate the implemented features by recording a short video. Make sure that you demonstrate all features, including incrementing the counter, decrementing it, and resetting it to the values given by the switches.

5. Outputting to 7-Segment Display

- Copy file PushButton.cpp into a new file called PushDisplay.cpp in the same directory.
- Write a function called SevenSeg () that takes a number from 0-9 and displays it on a 7-segment display chip on the DE1 board. The function should take two parameters: the number to display, and the number of the 7-segment display to use (0 to 3). See the memory map of the 7-segment displays on the last page of this document.
- Write a function called DigitDissect() that takes a number from 0 – 1023 turns it into 1 to 4 char (8-bit) variables each containing one digit from the number. You should somehow know the position of each digit in the decimal version of the number. Note: 10 LEDs can display a number from 0 to 1023.
- Use the above two functions and modify your Main() code to output the "counter" value on 7-segment display digits 0 to 3. There should be no leading zeros in the number just like in Lab 3.

Assignment 2

Compile and test your program. Demonstrate the implemented features by recording a short video. Make sure that you demonstrate all features, including incrementing the counter, decrementing it, and resetting it to the values given by the switches (as in assignment 1) with the value of counter continuously displayed on the Hex display chips.

4. Lab Submission Instructions

- Each lab consists of a set of **pre-lab questions** and **lab assignments**. You need to submit one lab report with the **answers** to the **pre-lab questions** and the **lab assignments**. If working in a group of maximum two students, only one student must submit one version of the lab report and the source code files with the names of both students on the report cover page and at the top of each source code.
- Write your lab report following the report template provided on Canvas.
- Any source code must be **well-commented** by explaining the purpose of your code. Every function should have a comment explaining its inputs, outputs, and effects. At the beginning of your source code files write your full name, students ID, and any special compiling/running instruction (if any).
- For each lab, submit the following on Canvas before the announced due date/time:
- The lab report, as a single Word or PDF document.
- The source code (.cpp, .cc, or .h files) for lab assignments with C/C++ programming.
- Any files the assignments might ask for (e.g., a demonstration video file).
- Submit each of the above files separately (do not upload compressed files).
- You can submit multiple attempts for this lab; however, only what you submit in the last attempt will be graded (i.e., all required reports and files must be included in this last attempt).

5. Grading Rubric

Section	**Items**	**Points**
Prelab	Commented code explaining all the functions (1 – 9, see pp. 3-4) LedNumber.cpp	15
Assignment 1	Commented source code of program	15
	Screen shots of program	5
	Video Demonstration	20
Assignment 2	Commented source code of program	15
	Screen shots of program	5
	Video Demonstration	20
Lab Report Quality		5
	Total Points Lab Report	60
	Total Points Demonstrations	40
	Grand Total Points	100

7-Segment Displays Parallel Port

There are two parallel ports connected to the 7-segment displays on the DE1-SoC board, each of which comprises a 32-bit write-only Data register. As indicated in Figure 8 below, the register at address 0xFF200020 drives digits HEX3 to HEX0, and the register at address 0xFF200030 drives digits HEX5 and HEX4. Data can be written into these two registers, and read back, by using word operations. This data directly controls the segments of each display, according to the bit locations given in Figure 8. The locations of segments 6 to 0 in each seven-segment display on the DE1-SoC board is illustrated on the right side of the figure.

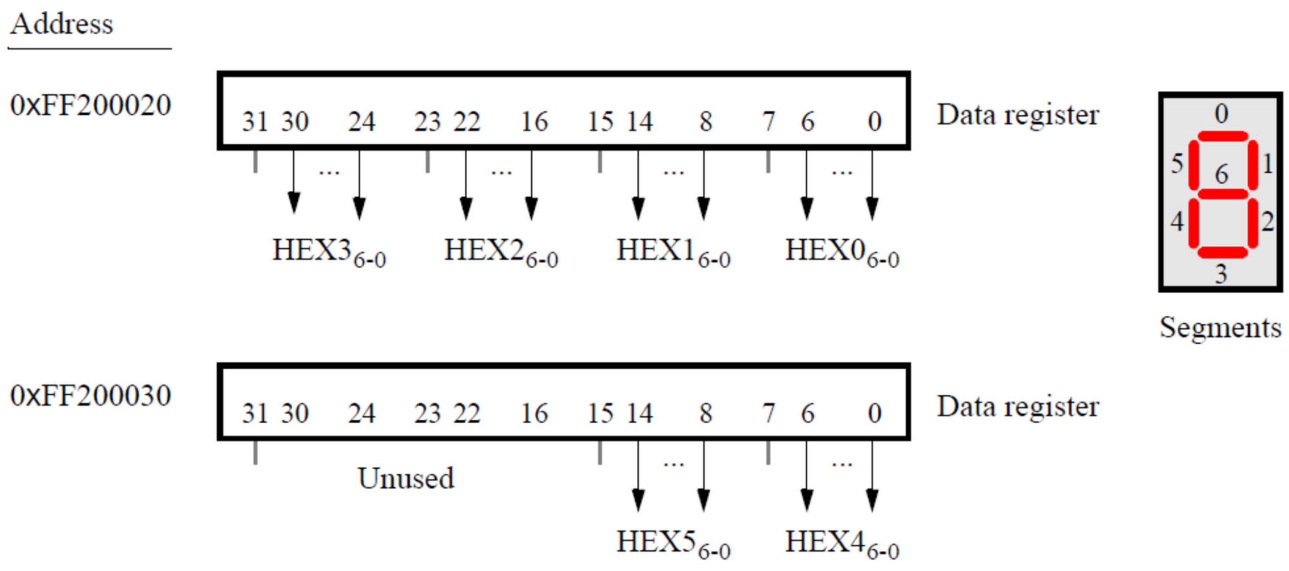


Figure 8. Bit locations for the 7-segment displays parallel ports.

From **DE1-SoC Computer System with ARM Cortex A9** document (p. 8) on Canvas: