Lab 1
Using TS-7250 I/O

Objective
In this lab students will learn how to access the I/O registers on the TS-7250 from a user-space program and a kernel module.

Prelab
- Before coming to the lab, look up the use of the functions `mmap` and `__ioremap` and their arguments.
- Know where the control registers for the LEDS are located (refer to the ep93xx user manual)
- Before we can start with the first lab assignment, we need to set up the TS-7250 boards. For that, we will use the document “Lab How Tos.pdf”, which can be found on the BlackBoard: “ECE 4220/ ECE 7220: Course Contents > Laboratory > Extra Information” Have this document ready for the first lab session. You should also take a look at the document “Lab Guidelines”, which can be found on the same directory.

Background

The TS-7250 has 20 Digital I/O lines available which are located on the two headers labeled “DIO” and “LCD”. All of the I/O lines can be programmed individually as inputs or outputs. All I/O registers are 8 bits wide and are aligned on 32-bit boundaries with each register being given a letter name such as Port A and Port B. Each of the bottom 8 bits of a register correspond to an individual I/O line, while the top 24 bits of each register are not modified when written to and read back 0’s. Every I/O pin has two registers used to access it: a data register and a data direction register, which is denoted as the DDR register. Modifying bits in the DDR register changes whether an I/O pin is an input or an output, while writing to or reading from the data register changes/returns the value of the pins.

In order to use the I/O registers in our applications, we will have to map these register’s addresses into our program memory. For applications, this is done by using the `mmap()` function while for kernel modules this is done using the `__ioremap()` function. Something to note is that `mmap()` requires a file descriptor, so you will have to open up the special file `/dev/mem` in your program first, using the `open()` command. Since we are mapping 32-bit registers into our program, we can use an unsigned long ptr for access to each port.
Lab Procedure

Week 1: Creating a Square Wave
1. In the lab you will be provided with an auxiliary board which contains 5 push buttons, 3 LEDs and a speaker (see figure above). The 5 push buttons and 3 LEDs are connected to the I/O lines associated with port B, while the speaker is connected to the I/O line associated with bit 1 of port F.

2. For this part of the lab, you are to create a program that first scans for user input. Valid entries are integers from 0 to 4. Depending on the number entered, the corresponding push button will then be used to activate the generation of a square wave on the speaker. This is done by constantly toggling the value of the speaker’s I/O line, which is located on Port F. For example, if the user enters a “1”, then the square wave should start after the button connected to port B1 is pushed. If any other button is pushed, it should be ignored. Try different frequencies until you hear a “nice” sound.

*Hint:* Other I/O lines associated with port F are used for system processes, so when writing to port F make sure that you only change the bit that is associated with the speaker.

Week 2: Creating a Simple Kernel Module
1. For this part of the lab, you will create a kernel module that simply turns on the red and yellow LEDs of the auxiliary board when it is installed, and turns them off when removed.
Hint: Remember to use __ioremap__ when accessing the I/O registers in a kernel module. You will need to include a specific header file.

2. Something to note about kernel modules is that they do not contain a main(), instead they contain two functions:

```c
int init_module(void)
void cleanup_module(void)
```

The `init_module` function contains code that is run when the module is installed. This function should return 0 unless an error has occurred. `cleanup_module` is the code that is run when the module is removed.

3. There are three helpful shell commands to use when dealing with modules:
   - `lsmod`: lists all of the modules currently installed in the kernel
   - `insmod NAME.o`: installs the module whose filename is NAME.o
   - `rmmod NAME`: removes the module with name: NAME (notice the .o is not included)

4. Last, to compile the module, you need to define the symbols: `MODULE` and `__KERNEL__`. This can be done one of two ways. First, you can define the symbols at the beginning of your source code, which would look like `#define MODULE`. Another way is to define the symbols when you compile in Eclipse by going to the properties and clicking on symbols under the compiler settings.

5. To avoid warnings when installing the module, add the following line to your source code `MODULE_LICENSE("GPL");`

6. In order to debug your module code, you cannot use the `printf()` function. Instead, you can use the `printk()` function and then check the printed lines using `dmesg` command in the terminal. Modify your kernel module so that the message “MODULE INSTALLED” is printed when the module is installed, and the message “MODULE REMOVED” is printed when the module is removed. Install and remove your module several times. What do you see when you run the `dmesg` command?

PostLab Questions:

1. Assuming that PortB has 8 I/O lines associated with it, what value would you assign to Port B’s data direction register so that the I/O lines associated with bits 0, 3 and 7 are inputs, and the rest are outputs?
2. Assuming that PortB has 8 I/O lines which are all configured as outputs, what would you write to Port B’s data register so that bits 2, 4, and 5 are set to one, bits 1, 3, and 6 are set to 0 and bits 7 and 0 retain their original value?

3. Imagine that in the user-space program (Part-1) you want to terminate the sound by pressing a key on the keyboard. How could you do that? Are there any implementation difficulties? (You don’t have to implement it).

4. How do you think you could solve the frequency issues to get a “nice”, consistent sound in the speaker?