Lab 3: Interrupts and Realtime

Overview

At this point, we have learned the basics of how to write kernel driver module, and we wrote a driver kernel module for the LCD+shift register. Writing kernel driver module is one way to build an embedded system with Linux. However, we can program the board by only writing user-space C code (like the C code in all homework). The advantages of programming the board with user-space C code are listed here:

1. It is easier to write user-space program than to write kernel module because writing kernel module requires us to follow some "templates" and steps to "insert" and "remove" the module.
2. It is easier to debug the user-space program than to debug kernel module.
3. User-space C program is more mainstream, and in general more professionals write user-space C programs instead of kernel modules (but knowing how to do that is still a good skill). Learning user-space C programming does not only help you understand concepts in embedded systems, but also help you understand programming, computers in general.

Therefore, starting from this lab, we are moving from using kernel module and to user-space C program. We will use **interrupt, which is one of the most important concepts in embedded systems.** we are going to add distance sensors and motor-driver chip to the system using interrupt. At the end of this lab, you will be impressed how new features you have added to the system and how much you have learned!

In this lab, you will learn to understand and work with interrupts and use them to implement the distance sensor of the RoboTank system. You will also update your button code from lab0 to run more efficiently by replacing polling with interrupt driven software, and use the button to control LCD display.

Background

As you learned in lecture, the majority of embedded systems rely on interrupt driven software. This is because embedded systems are integrated very closely with peripheral hardware and often want to respond only when a certain event occurs (a user pushing a button for example). Assuming we have a system that requires low power, and If the processor were polling the button, it would consume unnecessary power and CPU cycles. Instead if we use interrupt, the system could stay in a low-power mode until an interrupt "wakes it up". Then the processor handles the interrupt by executing the interrupt handler function and then goes back to the low-power mode. Although our project doesn't have any power-related constraint, interrupts are still important to allowing us do multiple things concurrently, and that makes our project closer to a real-time embedded system. A real-time system is a reactive system that responds to or reacts to signals from the environment in order to meet various timing and other constraints (how strict the timing constraints depends on specific situations). In our case, a self-driving tank reacts to the environment in a way that it detects obstacles depending on the distance sensor data and avoids them by controlling the motor of the tank.

We will use 2 kinds of interrupt in this lab, and they are **timer interrupt**, which executes a handler function on a preset frequency, and **general interrupt**, which executes a handler function when some user-defined conditions are met. So far we have talked a lot about interrupts, but don't worry, the whole interrupt story will make perfect sense when you get started reading some sample code.

Besides interrupt, we will learn how to user IPC (Inter-process communication) as well. We need to go through several terminologies first. First of all, what is a "process"? The answer is that any running program is a process. For example, the PDF reader or browser you are using to read this specification is a process. If you write a C program with a "while(1) " infinite loop, and run it, the running program will be a process. Don't feel strange about the "while(1) " infinite loop, because most application programs have an infinite loop. This is because you want your program to exit when you tell it to do so or when it encounters some problems (unlike the bash programs such as "ls", "cd" and etc, which you run it and expecting it to finish so you can see the results). The cool part about processes is that the operating system can run many of them concurrently and let them send information to each other or interrupt each other! For example, when you copy and paste a text line from your PDF reader to your text editor, the information (text line) is sent through using the Inter-process communication.

The basic idea behind IPC is that process A can create a special file, and process B can write to the file. Then process A can get what B writes to the file by reading from there. The "special" file is called fifo file, and it is a pipe that connects multiple processes. The name for this IPC method is **named pipe.** Refers to **pipe_Receiver.c** and **pipe_Sender.c** as reference code.

```
>>gcc -o receiver pipe_Receiver.c
>>gcc -o sender pipe_Sender.c
```

After you compiled both of them, run "receiver" first and open another tab or new terminal window to run "sender", and you will see what happens.

So far the communication via the named pipe not dynamic. Because one process can write something in a file "silently" without notifying other processes to read from there. Therefore, after one process A writes something in a file, then it interrupts another process B, and in the interrupt handler of process B, it reads the data from that file. Now the question comes down to ow does a process interrupt another process? The answer is a process send a **signal** to another process with its process ID (once a process starts running, the OS will assigns a process ID to that process). So any process can send signal to any process (including itself) if it knows the other one's process ID. Refers to **signal_receive_test.c**, **signal_send_test.c** and **send_Signal.sh** as sample code. Compile **signal_receive_test.c** on your own computer with the below command:

```
>> gcc -o signal_receive_test signal_receive_test.c
```

Run "./signal_receive_test", and then run "./send_Signal.sh" to see what happens.

All we have talked about here are very basic concepts of the IPC. You are highly encouraged to google keywords such as "named pipe", "signals", "Linux IPC" and etc, and you will find a lot of useful sources. Because they are very common and popular concepts in Unix-based operating systems.

After all the concepts, how do we do that in our system? Basically you can have one program "taking care of" one hardware. For example, you can have a "Master" program running, and let a "Slave" program keep sampling the ADC data and basing on the sampled data to make decision when to interrupt the "Master". When the "Master" is interrupted, it reads from some specific fifo file to get the information the "Slave" sends to it. Then the "Master" will control the entire system basing on information from "Slave". Let's look at the hardware we are going to use in this lab, and then you will find this paragraph makes more sense.

Part 1 (GPIO kernel interrupt)

We used a button to control LCD display by using polling in the previous lab. In this lab, we are going to use interrupt to reimplement the same button effect. Basically, rather than keep checking the button state, we will create an interrupt handler in the loadable kernel module(LKM), and the code in that interrupt handler will be executed only when the button is pressed. The interrupt is initialized and registered when the LKM is inserted into the kernel. With the interrupt implemented, the code in userspace doesn't have to keep checking the value of any GPIO pins. Because our focus for this lab is to write user space program, we will provide you with the code for this part, and you just need to insert the module and use it to replace the polling code. But still try to read through the given code and understand as much as you can.

Part 2 (ADC sampling with software timer interrupt)

We should have the LCD display system for the tank finished completely by far. Now we will add the “eyes” of the tank, that is the distance sensor. Start by wiring up your sensor. The red and black wires are of course power (5 V) and ground respectively. The white wire should be plugged into the one of the ADC pins on the X69 header. The voltage on white wire will start at 0 V when no object is in front of the sensor and increase as an object approaches. The ADC module of the CPU will read the analog value at corresponding pin and digitize it so that the data can be read by programmer using “cat involtagex_raw ”. The next question is that even though the data can be read by the shell command or file I/O function calls, how do we know when to request the data? This is where the **timer interrupt** comes into play. we request the ADC data at a certain frequency. For example, we can read the ADC device file 10 times every 1 second so that we will have 10 ADC values indicating how the distance changes in this 1 second. This is how our tank will know its distance to peripheral obstacles (Imaging if we get 1 ADC data every 2 seconds, that frequency would be too slow and the tank is very likely to run into obstacles. Therefore the system is not real-time).
The timer interrupt is set up in the userspace instead of the LKM. The below URL provides in depth explanation and sample code to create a software timer interrupt in C. Note that the code to set up a timer to run on the Phytec board is different from the code to set up a timer to run on your computer as in:

http://man7.org/linux/man-pages/man2/timer_create.2.html

To compile the source code, we need to add “-lrt” as an argument flag for the compiler so that it knows to link with the right library. For example:

```
$(EXECUTABLE): $(SOURCES)
$(CC) -std=gnu99 -g -o $@ $^ -lrt
```

In the handler function, we should sample the ADC value. The sampling rate is up to you to decide. For example, it could be 300 Hz, which means getting 300 sample values first, then taking the average of all samples. This process should watch the average value. When the average value is greater or less than some preset threshold value, this process should send a signal to interrupt the “Master” process. Then the “Master” makes decisions.

The system should have 4 distance sensors, which means 4 ADCs are used to convert the analog voltage to digital values that we can access in our C code.

Part 3 (H-bridge and putting things together)

At this point, we should have the “eyes” of the tank set up.The next step is to control the motor basing on what the tank “sees”. We are using something called H-bridge (TB6612FNG) as a motor driver. To read more about what an H-bridge is, how does it work, go to the below URL:

https://www.sparkfun.com/products/9457

To figure out how to use the H-bridge, read the datasheet which could be found in the above URL as well (understanding datasheet and learning to pull out the most useful information are among the most important skills for an electrical engineer in general). The table on page 4 could help understand the relation between input and output. One thing to pay close attention to is that **DO NOT mixed Vm and Vcc.** Vm is the power supply for the motor, while Vcc is the power supply for the H-bridge. Once they are mixed up, the H-bridge will be burned, and we have limited number of backup H-bridge.

In general, the “Master” process controls H-bridge to move the tank basing on the distance information provided by “Slave” processes.

What to turn in

README.md // simple overview to your code, explain all of your *.c and *.h files
All corresponding *.c and *.h files
Makefile

You should type `make clean` before pushing anything to the repository to be graded. We want to be able to type `make` ourselves and look through the source code directly. Additional scripts are allowed as well, such as a file that makes and uploads the file to the Phyboard.

Comment and style do make a decent portion of the final grade. Appropriate use of header files, consistent style, appropriately segmenting code into methods, and concise comments all will be taken into consideration. All of the pins you use should be apart of preprocessor define statements. Tab size should be set to 3, and all tabs should be filled as spaces. This is an option available in just about every text editor and ensures that formatting remains consistent on every machine. Lines should not exceed 80 characters. Grading will not be as strict as CSE 14X courses, but will have similar guidelines for commenting expectations. Remember to include names and dates at the top of all documents!

All of the variables and function prototypes you include in your header should also be static. Unless stated otherwise, you can assume you only want them to exist within the scope of the containing file.

The readme and files are intended to require no more than 1 hour of work. The readme will mostly explain how to use your code and any other basic functionalities a git user might be interested in knowing if they were to pull your code. The lab report is intended to purely contain diagrams and tables of wiring for hardware. You can imagine that it is a minimalistic datasheet. We do want to see a very clean and concise diagram of the wiring you did for each project though, so you should expect to turn in a report every lab. You are welcome to include any other text in the lab report. For this lab, you should include a diagram of all wiring in and out of the Phyboard, ADCs and the H-bridge. Make sure to label any pins used. A circuit-like diagram is the best example of what we are looking for, but you are given stylistic freedom to convey the information how you please. Only turn in .pdf files for the report!

Demo

Pull the code from your repository and walk through the code with your TA. Show working ADCs and motor controlling. Printing the the sensor values on the console, and moving some object away and close to the sensor while seeing the changing of the value as a response to the distance of the object could be a good way to demonstrate. For H-bridge, you can set up a patterns of motor movements, such a 2 wheels both moving forward for 1 second, then 2 wheels moving in opposite directions for 1 second, then 2 wheels both moving backward for 1 second and etc. Show your TAs that the motor movement matched up with how you programed it, and expected it to be.