MICROPROCESSOR II AND EMBEDDED SYSTEM DESIGN

Lab Number: 3

Date: 04/06/16

Student Name: Trever Wagenhals

Student ID: 01425986

Group Name: Steel Elephants

PERSONAL CONTRIBUTION TO THE LAB ASSIGNMENT

For this lab, I ended up writing the new code to read and print the RTC register values. Since the code was actually fairly short once writing it was understood, it didn't require more than just one person to implement the code. I also creating a function to write a new time to the RTC in case the clock ends up getting off for some reason.

I also handled building the new Linux kernel, mainly because I already had a Linux OS installed to my laptop so it made it the most convenient.

PURPOSE OF THE LAB

In this lab, we were tasked with building a Yocto Linux kernel from scratch using the instructions provided to us. With the building of the Linux kernel, we had to make sure that I2C functionality is installed properly so that the rest of the lab can be executed within the user-space. Also, this lab wanted to demonstrate the idea of building a custom driver. For the lab, we were tasked with creating a demo "Hello World" driver that would print our team name to the screen when loaded and disabled.

Once the kernel was built, the next task was to learn how to use I2C communication to access a slave ds1307 Real Time Clock and be able to read and write to the registers of the device. The values of these registers each represent a portion of a displayed time and finding a good way to print a nicely transformed representation of the time was part of the project. This new ability to read time from a slave device was incorporated with the previous lab, allowing us to add a timestamp to ADC values to follow the changes of light intensity based on the time of the day.

INTRODUCTION OF THE LAB AND REPORT

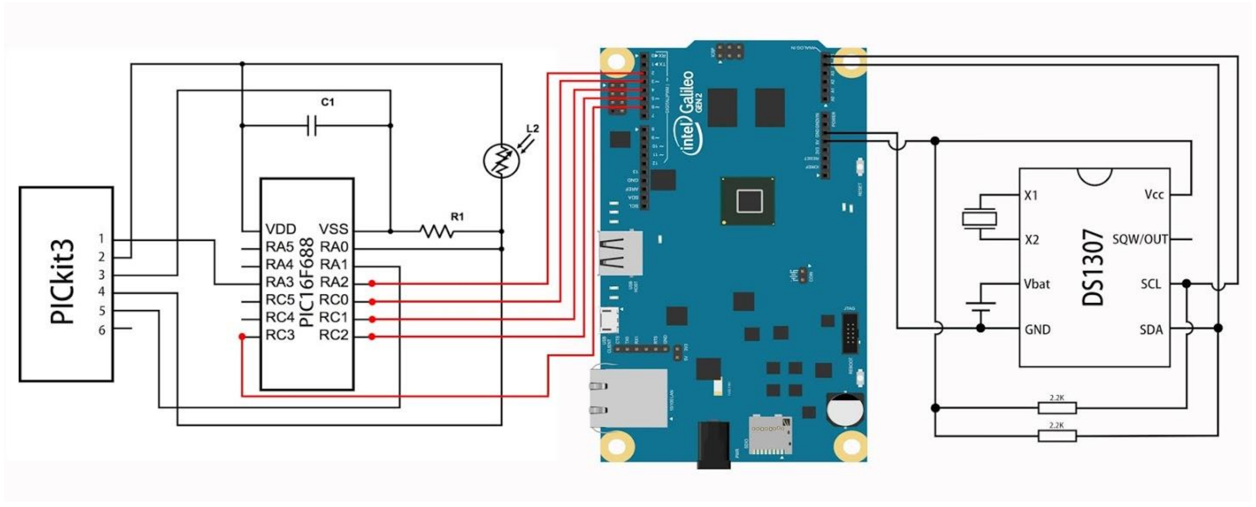
This lab built right on top of Lab 2 and just added another device for the Galileo to communicate to aside from the microprocessor. In conjunction with the GPIO bitbanging, we would now use I2C libraries and API to communicate with a real time clock stored at a specific address and alter each register to reflect the current time. This would then be used as a timestamp to reflect light intensities based on the time of day.

The Yocto Linux image traditionally doesn't come with I2C functionality, and so a custom image is expected to be built to introduce this functionality. Because of this, it was also a good time to introduce the concept of custom driver design by creating a simple hello-mod driver that outputs information based on your group name. This lab will discuss procedures, results, and potential pitfalls related to all of these ideas for future experiments.

MATERIALS, DEVICES AND INSTRUMENTS

PICkit3
MPLAB-X IDE
Galileo 2nd Generation
Breadboard
Wires
10k Ohm Resistor
Two 4.7k Ohm Resistors
32MHz Oscillating Crystal
Watch Battery
0.1 uF Capacitor
Laptop for remote accessing Galileo
DS1307 RTC
Linux Operating System
MicroSD Card
Photoresistor

SCHEMATICS



LAB METHODS AND PROCEDURE

For the building of the Linux, the procedure was identical to the one described on the GitHub. The building process involved adding additional functionalities to the kernel, and customizing the output of the hello-mod driver for your group. Once these two details were done, the image could be built with the bitbake command. Once the build was finished, the files were renamed to match the format required to be recognized by the Galileo.

Once the Linux image was built, it was time to concentrate on the RTC. In order to start working on the RTC, the Galileo needed to first be set up to using I2C functionality. In order to use this, GPIO 60 had to be set low to mux the functionality. At first, simply exporting and opening the files manually could be done with the following commands:

```
echo -n "60" > /sys/class/gpio/export

echo -n "out" > /sys/class/gpio/gpio60/direction

echo -n "0" > /sys/class/gpio/gpio60/value
```

Once these commands have been done, the ds1307 circuit should be set up and connected to the Galileo, as shown in the schematic.

Once set up, you should verify that the RTC is being recognized by using the following command:

I2cdetect –y –r 0

Once it is detected, you can immediately start setting and getting the register values with the commands:

I2cget –y 0 0x68

I2cset –y 0 0x68 (value to set)

These are just to get familiar with the register values and to make minor adjustments if needed. Automating all of this functionality in a program is the next step to execute.

As far as the actual programming portion of the lab, the program only required a total of three additional functions. One function was designed to open the appropriate GPIO to use the I2C functionality, and also open the I2C device in a read/write mode. Once the device was set up to be accessed, the function then points to the appropriate address among all the addresses on the device. The second function is the function that actually allows proper reading of the device address registers. This function uses a write command to point to the appropriate register, without actually writing anything, and then once the appropriate register is identified, it can be read. The value that is read is stored to an array to be accessed when needed in the future. This occurs in a third function that allows us to easily print out the registers in the order that we found the most appropriate when reading time.

When we set up this function initially, we had all of the formatting set up properly before actually trying to write the values to the proper ones, just so it was visually easy to see what needed to be fixed. Once set up, we used the write command that was also found in our second function to point to the register we wanted to alter. Instead of just pointing to it this time though, we would store a second value to the buf array so that value would be written to the register. By doing this, we were able to alter one register at a time until the clock finally output the correct result.

Once we had the clock time correct, we waited 24 hours with the clock unplugged to see what the result would be. The clock ended up being 6 minutes off, but this is something that is possible based on crystal accuracy and temperature conditions. In order to offset this though, we decided to write an additional function that allows the user to easily fix the time in case this happens again. This function just prompted the user for values based on the time, and then stored those values to an array, which are eventually passed to the buf array when the user confirms it. At this point, we were more comfortable with the writing portion of the lab, and so we decided to pass all the values through at once instead of doing it individually like we did originally.

TROUBLE IN THE LAB AND HOW DID YOU SOLVE IT

As far as the lab went with building the Linux image, there were not a whole lot of troubles as much as just inconsistencies in the notes to perform the lab. The first issue to note is that the GitHub references version 1.2.0, although the newest version is now 1.2.1, so the ability to copy and paste a lot of the instructions to the command prompt were eliminated. This just went that extra attention had to be given to ensure the instructions were followed properly. The next issue also involved downloading the hello-mod file off the GitHub. The GitHub did not have a downloadable zip, so the individual had to use the "git clone" command or "wget" to download the files. I was not familiar with "git clone" so I opted to use "wget" and just reconstructed the file once I downloaded each individual piece.

The next issue that occurred was a "tmpdir error" when trying to execute the bitbake command. This probably occurred because I am using Linux Mint as my OS of choice, while the instructions were for Ubuntu. I did manage to find a page that went over this issue though, which gave a list of programs to sudo apt-get install, re-running setup.sh, and then executing a "touch conf/sanity.conf" command. The combination of these allowed the bitbake command to execute properly.

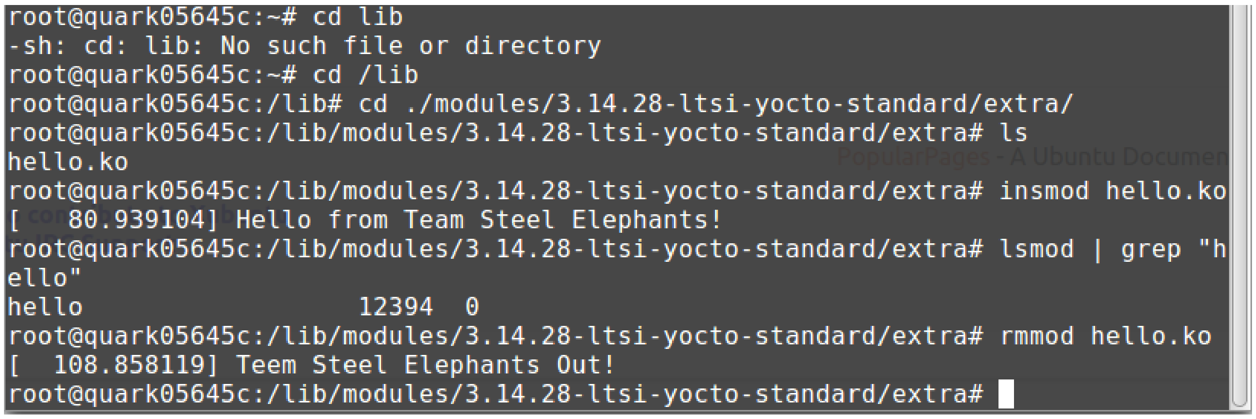
The first time I tried to do the build, I did not realize that I did not have enough space for to actually download all of the requirements to build the image, and so I found that my build failed halfway through. To resolve this issue, I had to extend my Linux partition to have enough room to finish the process.

Lastly, once the Linux image was built, getting it to properly load on the Galileo seemed to be its own challenge. I do not know exactly what was causing the issue, but I kept getting kernel fail messages. After trying to re-format, re-copy, and re-name the files several times, I finally got it to boot into the OS properly.

As far as issues with the RTC programming aspect, we actually did not encounter a whole lot of issues. The main issue that we had was probably the change in GPIO I2C muxing from gen1 to gen2, which was not clarified properly in the notes. Instead of setting GPIO 29 to low, we had to set GPIO 60 to low so the RTC would show up in the I2C tree.

The overall programming of the DS1307 posed no issues because adequate research was done before executing and the functionality was implemented one step at a time. The only detail that we had originally missed was a decimal to BCD conversion. Because of this, the hours, minutes, and seconds registers would sometimes look messed up depending on how you wrote data to them. So, for example, sometimes the seconds would go above 60, sometimes the format would be hexadecimal, and sometimes the minutes/hours wouldn't change when the cycle reset. This occurred mainly because writing to the registers appeared fine in some instances, such as when the value written was below 9. Also, in cases such as the year where you can write a value such as int 16 and have it read properly the whole time even though it is functionally wrong made it really hard to diagnose. (int 16 = 0001 0000, BCD = 0001 0110) Once the BCD conversions were implemented, we tested it by setting every register to their highest value and made sure all registers reset properly.

RESULTS AND CONCLUSION



```
root@quark05645c:~# cd lib
-sh: cd: lib: No such file or directory
root@quark05645c:~# cd /lib
root@quark05645c:/lib# cd ./modules/3.14.28-ltsi-yocto-standard/extra/
root@quark05645c:/lib/modules/3.14.28-ltsi-yocto-standard/extra# ls
hello.ko
root@quark05645c:/lib/modules/3.14.28-ltsi-yocto-standard/extra# insmod hello.ko
[   80.939104] Hello from Team Steel Elephants!
root@quark05645c:/lib/modules/3.14.28-ltsi-yocto-standard/extra# lsmod | grep "h
ello"
hello                  12394  0
root@quark05645c:/lib/modules/3.14.28-ltsi-yocto-standard/extra# rmmod hello.ko
[  108.858119] Teem Steel Elephants Out!
root@quark05645c:/lib/modules/3.14.28-ltsi-yocto-standard/extra# 
```

Overall, this lab gave a lot of insight to numerous topics in regards to embedded systems. The concept of building a custom Linux image, building custom drivers, and using the I2C communication protocol were all executed. This showed the simplicity of I2C in comparison to bitbanging GPIO values. Comparing the length of GPIO code to I2C code, it can be seen how much simpler communication could be between devices.