MICROPROCESSOR II AND EMBEDDED SYSTEM DESIGN

Lab Number: 4

Date: 4/29/16

Student Name: Trever Wagenhals

Student ID: 01425986

Group Name: Team Steel Elephants

PERSONAL CONTRIBUTION TO THE LAB ASSIGNMENT

For this lab, I decided to take the initiative and actually ended up producing most of the code in relation to both the threading and the server communication.

PURPOSE OF THE LAB

The purpose of this lab was to introduce the concepts of multi-threaded applications to develop parallel computing, and to be able to transfer data collected to a password protected server.

INTRODUCTION OF THE LAB AND REPORT

This lab built entirely on the same concepts of the previous labs; a programmed microprocessor through a PICkit communicated with a Galileo via GPIO to fetch an ADC value adjusted by a photoresistor. In parallel with this functionality, a real time clock was set up to fetch the current time and create a timestamp to accompany the ADC value so the adjusted values over time can be analyzed. In order to preserve this data, this lab involved sending the data collecting to a server.

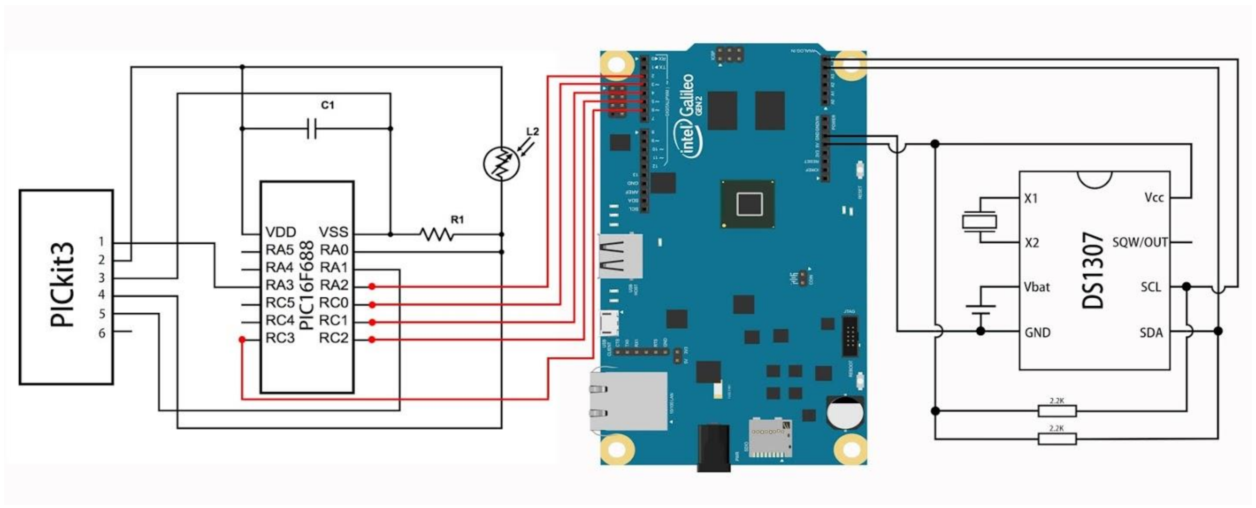
This lab involved several steps necessary to result in a final product. Firstly, the professor needed to establish a server for the Galileo devices to eventually communicate with. Because this was not originally established, concepts of simulating our own server in python script were discussed in order to give us a better understanding.

Once the new functionality was implemented, each task was expected to be divided into separate threads to better understand the concept of multi-threaded systems. The lab required at least three separate threads for user interface, PIC communication, and server communication. As a bonus, each group could also establish dual-way communication with the server.

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
Ethernet Cable
Established Server with Password and ID

SCHEMATICS



LAB METHODS AND PROCEDURE

The first step of this lab should be to verify proper communication with the server. The first thing to be done is to go to the server URL and attempt to manually enter the assigned group information. By doing this, several things are verified; the password and ID that were assigned by the professor were correct, and a sample group URL is generated to visually see how the information should be sent to the server properly.

Once communication was established manually, utilizing the libcurl library was the next step to be implemented to ensure proper communication. By using the group generated URL from the manual test, and the test.c example on the Github, server communication through a C program could be attempted. Instead of using the two functions and passing all the parameters to a string and then passing that string to the curl_easy_setopt function built into the libcurl header file, the string could be replaced with the exact URL that the manual attempt generated. This is how server communication through C programming was verified by our group.

Once the functionality was verified, a server function was written and integrated in to the rest of the program, calling on the newly generated ADC value and timestamp each time a user requested it. Also, another global variable was established with the acknowledge check so that the status of the read would be properly represented. This was important in case the user reads an ADC value that was no acknowledged, meaning that the data is invalid. All of these variables were appended to a string in the same fashion as the test.c example, and the string was passed to the libcurl function. Once all of the new functionality was implemented, it was time to create separate threads for each task.

The lab description called for at least three threads to be created, a UI thread, a user choice computation thread for microprocessor communication, and a server thread to send data to the server. Although three threads were mentioned, our group ended up creating a fourth thread as a standby thread. This thread allowed the user to decide if they want to allow server communication when the user is inactive and specify how long the inactive time must be. The lab description required this functionality built into the lab, and because the timer we created relied on a sleep call to function, it did not seem plausible to put it in one of the other threads without affecting speed. Also, this allowed the user to go above just having a predefined standby action that is always running and choose if and how long it will run.

Creating the threads required four unique threads to be created, two mutex locks, and 3 conditional variables. The main three thread described shared a single lock, which ensured that only one thread used a variable at a single time. Each thread also had a conditional call so that a thread can be woken up when the previous one finishes. The standby thread we created had its own lock and did not have a conditional signal because the way we designed it had it always running in the background. The thread never needed a signal from another thread to execute anything, and only sent signals to the other threads when it was executed.

Through threading, the main function was dissolved from holding all of the functions necessary to only the four threads being initialized that hold their respective functions. Doing just this however would have the main thread race to the end and terminate. In order to fix this, each thread was put in an indefinite while loop so that they would never terminate, and a call to join the threads was put at the end of the main loop to join the threads when they terminate. Because they were set to never terminate, this ensured the program did not exit.

Once the program was properly broken up into the three threads necessary to imitate initial functionality, the four thread was developed. This thread waited on user input to set a flag to signal to start counting to the time the user specified, and lock all variables while it executes code so that there is no race condition.

Once the basic concepts of the lab were completed, we decided to try to implement the bonus and have dual way communication to the server. Through examples, the WRITEFUNCTION and WRITEDATA were found to enable dual way communication. Through these two commands, two additional functions were created, one to fetch the data that the server printed, and another to save the fetched data to a string. Once this was accomplished, the string could be printed out to verify that it was received correctly.

Once the string was received, it now had to be analyzed to determine what command the server was executing. Looking for the command was easy, as it just required an else if statement to check if the command string was located in the full string. Once the command was located, the time delay needed to be determined. The way we did this was to just copy a certain character from the string until it was the proper character. Once the character was copied, it was then converted to an integer and a sleep function was used to create the delay. This was difficult, because each string was a different length because the length of the command strings varied, so each location in the string varied. As a group choice, we decided to only have the dual way communication active when the user inputs data and not when the standby thread does, so an additional flag was created to check for this.

TROUBLE IN THE LAB AND HOW DID YOU SOLVE IT

The first issue that was faced was one caused by our absence. For the first lab session in relation to this project, all of my groupmates had decided that we were not going to go to the lab session since we had other matters to attend to. This lab session was when the libcurl library and the python server simulation code was discussed. Because of this, looking at all of the files on the Github was fairly overwhelming. The lack of commenting in the programs also did not assist us. To counteract this, a lot of reading into the libcurl header file had to be done and examples viewed to understand the concept. Once the examples were viewed, it was easy to see that the python code was unnecessary in the file product, making the direction to take much clearer.

When the threads were written for the first time and implemented into our function, we did not use the concept discussed above of only running the threads in the main loop and signaling each thread. Instead, the thread was created right when it needed to be executed inside the previous thread, and then they were set to join at the end to ensure that the previous thread waited for it to finish. Of course this worked, but it involved creating a new thread every time action needed to be taken, which was less efficient than just declaring them once. Talking to the teach clarified this issue and allowed us to re-examine how we initiated them.

Another minor error due faced was compiling the programs with the new libraries called. The pthread and libcurl libraries require the the –pthread and –lcurl commands are also present when compiling, otherwise undeclared variable messages would be spawned. This was a small fix but is very important to note.

Once we rewrote the threads to all be initiated at the beginning and run indefinitely, getting a clear understanding of how the lock worked needed to be done. Initially, the concept of declaring the lock at the top of each thread and having it wait in a while loop for a signal was not clearly understood. The fact that the condition_wait command actually released the lock while it was sleeping and only grabbed it again when awake was not understood, and so for quite a while the unlocking and locking timing was messed up for our code. This resulted in both race conditions, as well as deadlocks where the program would just freeze. After further research on the concepts though, this issue was resolved.

Due to having so many threads running at once and wanting the functionality to be as free as possible for the user, multiple flags had to be used in order to ensure events occurred only at specific times. Figuring out when to set and reset these flags was difficult, but it allowed for the user to have more control over the functionality.

RESULTS AND CONCLUSION

This lab was once again a very knowledgeable experience to undergo. Having built up the entire project throughout the course, the potential and power of IoT devices such as the Galileo is able to be realized. The network of every device surrounding us is even more pronounced through this project, where so many pieces came together to make a working, data-collecting system.