Microprocessors 2

EECE-4800

Lab 2: Interfacing with a Sensor Device on an Embedded Computer System

Instructor: Yan Luo

TA: Ioannis Smanis

Group 10

Author: Kyle Marescalchi (me)

Teammate: Derek Teixera

Teammate: Hans Hoene

Handed in 10/30/17

Due 10/30/17

Section 2: Contributions /1 points

1. Group Member 1 – Kyle Marescalchi (Me) ………………………………….

   In this lab, I altered the previous counter-based PWM methodology to implement a more direct and focused iteration using timer delays in a loop. This was created so that the servo motor could be turned to a more direct degree than before, where it was simple a +/-90* turn only. Further, I did research into the process of setting the GPIO of the Galileo 2 board, designed, and wrote the code for such.

2. Group Member 2 – Derek Teixera …………………….

   Derek focused on the Intel Galileo and how it functioned, implementing most if not all of the physical portions of communicating with the Galileo. This includes installing the Linux OS onto it, learning how to interface with usb media, and assisting on implementing and explaining the process of GPIO. Further, he handled the physical design of the communications, and assisted largely in debugging.

3. Group Member 3 – Hans Hoene…………………….

   Hans developed the most of the communication code between the Intel Galileo and the PIC, also drafting and explaining the process in which the communication worked. He tweaked code to his needs and was able to get it to work properly. Further, Hans implemented the working interrupt code. Working together with Derek and I, we implemented the code into each other's portions to have full functionality, and debugged it together.

Section 3: Purpose /0.5

The purpose of this lab was to understand how a microcontroller can communicate with a microprocessor. Using what is primarily the same basic functions from the first lab, these were adapted to instead be called by the Intel Galileo so that the processes can be operated and requested by the microcontroller. The lab demonstrated the use of strobe signals, the GPIO pins, and the general process of interrupting.

Section 4: Introduction /0.5

Using the same ADC and PWM servo as lab one, lab two created a communication-based process between the PIC16F18857 and the Intel Galileo 2 board. This was performed using the general purpose pins of the PIC and the GPIO pins on the Intel Galileo 2. The communication process between the two was handled using five pins, one for the strobe and four for communication. These four specific pins need to switch between input and output for the necessary communications; the Galileo sends a strobe to signal that there will be a command, the PIC prepares to receive the signal, the Galileo sends out a 4-bit signal, then the communication results are sent back to the Galileo or an error is triggered. In the case of the ADC, the Galileo takes three series of four-bit data to receive its overall information.

Section 5: Materials, Devices and Instruments /0.5

- Intel Galileo 2
- PIC-16F18857 Microcontroller operating at 3.3V
- PIC-Kit interface
- LDR Sensor (Photoresistor)
- Red LED
- 2x 10k-Ohm Resistors
- Servo Motor
- Breadboard
- Oscilloscope
- Analog Discovery
- Benchtop Power Supply
- Wires

Section 6: Schematics /0.5

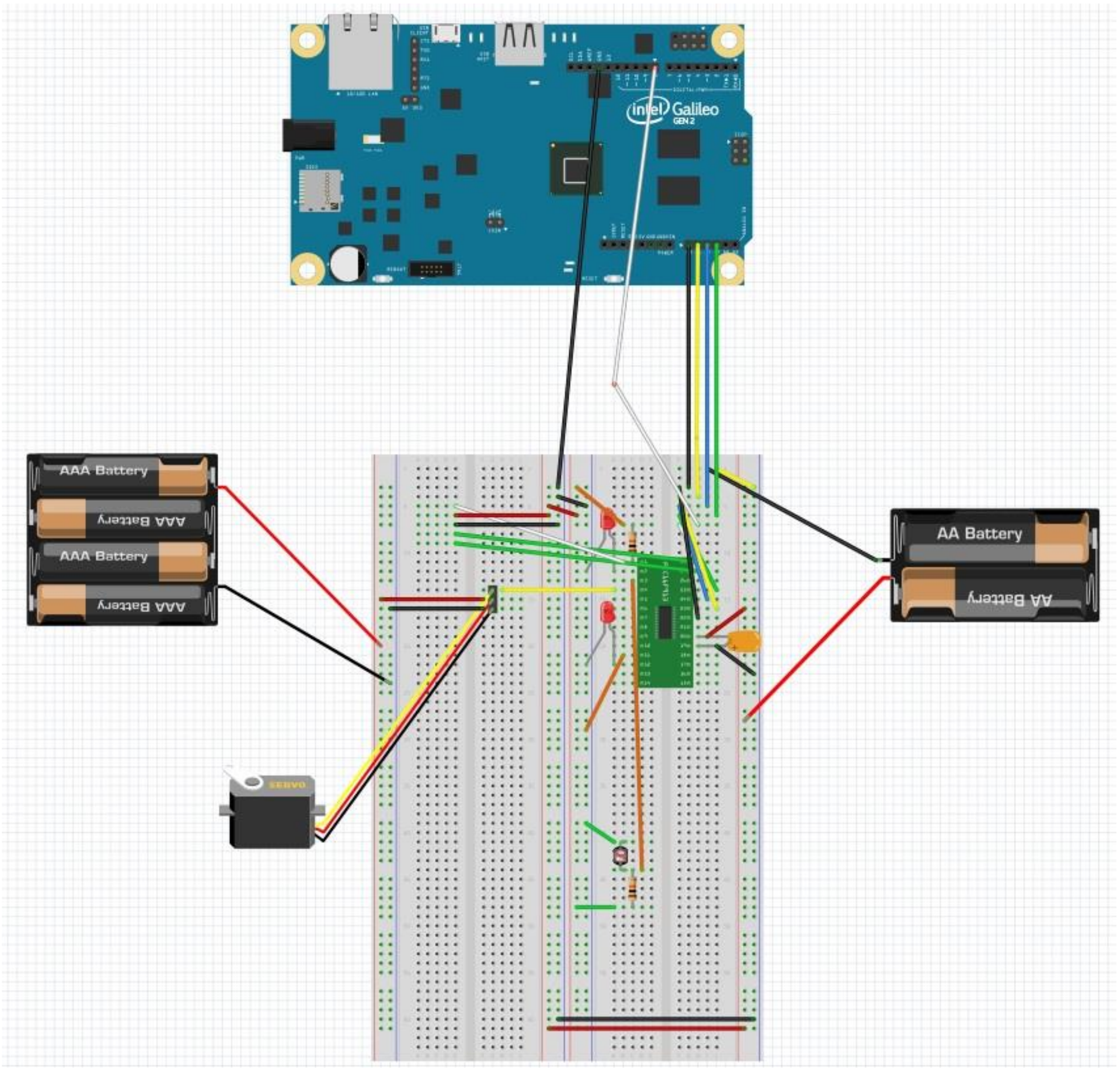


Figure 1: Lab Schematic

Section 7: Lab Methods and Procedure /2

Hardware design:

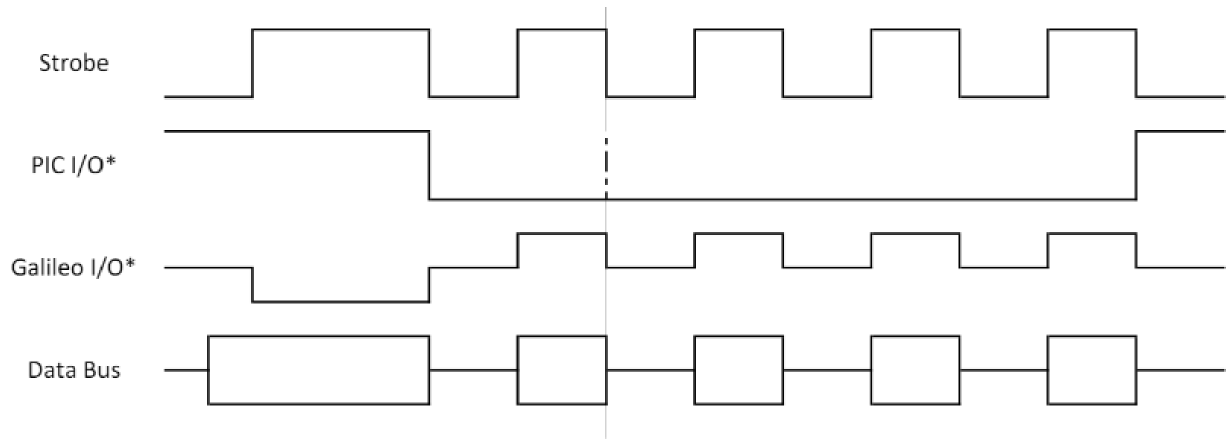
In our circuit, the “B1” through “B5” pins were used to communicate with the PIC kit, and the GPIO pins A1->4 on the Intel Galileo, along with pin 8 for the strobe. This decision was made largely for sake of not having a functioning GPIO pin 0, which was something that we could not find a resolution to. The strobe functioned exactly as desired, however, so this did not become an error. Other than such, an extra LED along with a loose wire was extended to the circuit so that there could be a physically operated interrupt (not necessary to the lab, used for our experimentation), where the LED’s flashing would indicate a hard reset.

Software design:

The largest focus of this process was the way the GPIO pins would be initialized then operated, and as such there were several sections of the software that needed to be focused on. First there is the initializing process, which operates in a simulation of the Linux coding. Using the sprint, open, read, and close functions the program would create the necessary files for the individual GPIO pins as they were called, and set the direction for their operation. This was done by creating and writing into system files that set the operation of each pin on an individual basis, which can be seen in Appendix 1.

The second process was implementing the bussing feature, which was necessary so that the PIC could handle what it was doing *before* having the operation be performed, so that there would be no direct overrides or half-performed functions. This code can be seen in Appendix 2, and allows for the queueing of functions (which is often unnecessary, given the speed at which our functions operate).

The process in which the communication is handled can be seen in Figure 2, which is the timing diagram. The actual implementation of this can be seen in Appendix 3, but it operates roughly as follows: The Galileo will write the strobe high to signal that the PIC is to receive a command, and waits until an acknowledgement. When the PIC’s function begins, it writes the ACK signal to the Galileo to have it prepare for the data to be received. When taking the ADC in, it was a result of three separate bursts of four signals, a requirement based on the 10-bit resolution of the PIC ADC. On the Galileo side, it outputs a result stating the digital value of the sensor as well as the actual voltage.



Note: dotted line only passed when GET request.
The more solid line is the path that would be followed if the request is **not** GET.

Figure 2: Communication Process

Section 8: Trouble Shooting /1

Issue 1: **The Galileo was, no matter what, only receiving an error back from the PIC – indicating that there would be no resulting acknowledgement code signaling a received request.**

Resolution: It was found in the code that the Galileo was not properly drawing the Strobe signal HIGH and LOW. Using an oscilloscope, the signal from the strobe was tested when a function was sent. It was seen that, instead of drawing low after sending a signal, the Galileo's strobe would remain HIGH at all times. The error was on the Galileo side, then, and was found to be an error in one of the calls for the strobe signal. A mix of the filehandleGPIO_S and the Strobe definitions for the pins were used, which led to an improper mixture of defining variables for the Galileo's initialization variables. When resolved, the signal received properly.

Issue 2: **The servo had an incorrect setting for its distances, and when attempting to set it to 30* or 120* as the function calls for, it was never correct.**

A mixture of issues led to the overall problem with this. The servo motor would result in incorrect distances regardless of direction *unless* it was either 0*, 90*, or 180*. Firstly, the operating voltage of the motor was originally set to 3.3V from the power supply, whereas it is called to be a 4-5.8V volt supply. When increased in voltage, it began to move more swiftly, however it was still in the incorrect directions. Only through experimentation it was found that our motor was more responding to the 0.5ms to 2.5ms pulse widths for the -90* or 90* directions. The cause of this is likely due to the 3.3V signal from the PIC, instead of the roughly guessed 5.0V signal.

Issue 3: **'Corruption' of further signals after initial errors or rapidly placed orders.**

We had a problem with being unable to perform a proper reset after an error; this was resolved by the creation of a buffer system for our signal so that there would not be an overridden signal to the PIC. Example: "4 4" would cause the PIC to take in the Ox4 signal, then immediately force another Ox4 signal. This was also resolved further by Hans 'flushing' all results after the first, so that "4 4" would read only the first 0x4.

Section 9: Results /0.5

The final results of the lab are exactly as they should have been. Every signal was able to be handled and processed by the Intel Galileo and the PIC16F18857 microprocessor alike, with a seamless communication between the two. Results were near instantaneous, with the only oddity in the results being with the servo motor – which seemed to have fluctuating (but roughly accurate) results based off the power generator used to supply it. No errors occurred in our function (that we could easily determine) and we were capable of pinging, resetting, calling the ADC, and turning the servo to desired degrees. The largest consideration of how this could be improved on is finding a better way to handle the servo motor, which was operating with relatively crude accuracy. It is unclear how the differences between the individual power generators are the issue with this, but it is known that it is functioning regardless. With tuning of a power generator depending on its specific model, the servo motor could be fine-tuned to a better functionality. When operating the ADC, it would return the digital value and the actual voltage of the photoresistor – something that was tested and found (approximately) correct for multiple light levels in different environments.

Section 10: Appendix

A1.

```
Int openGPIO(int gpio, int direction) {

        int handle;              //file variable
        char buf[256];
        int inout;


        //simple command to enable pin A0


        handle = open("/sys/class/gpio/export", O_WRONLY);


        sprintf(buf, "%d", gpio);


        write(handle, buf, strlen(buf));


        close(handle);


        sprintf(buf, "/sys/class/gpio/gpio%d/direction", gpio);


        handle = open(buf, O_WRONLY);


        // Set out direction
        switch (direction) {
        case GPIO_DIRECTION_OUT:
                write(handle, "out", 3);
                inout = O_WRONLY;
                break;
        case GPIO_DIRECTION_IN:
        default:
                write(handle, "in", 2);
                inout = O_RDONLY;
                break;
```

```
            }
            // write(handle, "out", 3);
            // Set in direction




            close(handle);


            sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio);


            switch (direction) {
            case GPIO_DIRECTION_OUT:
                   handle = open(buf, inout);
                   break;
            case GPIO_DIRECTION_IN:
            default:
                   handle = open(buf, inout);
            }
            // handle = open(buf, O_WRONLY);


            return handle;


    }
```

```
A2.
void writeBus(int value, int *bus) {
       writeGPIO(bus[0], value & 0x1);
       writeGPIO(bus[1], (value >> 1) & 0x1);
       writeGPIO(bus[2], (value >> 2) & 0x1);
       writeGPIO(bus[3], (value >> 3) & 0x1);
}

int readBus(int *bus) {
       // readGPIO must return 0 or 1 !!!
       int value = 0;
       value |= (readGPIO(bus[0]));
       value |= (readGPIO(bus[1]) << 1);
       value |= (readGPIO(bus[2]) << 2);
       value |= (readGPIO(bus[3]) << 3);
       return value;}
```

A3.
```
/*
               START STEP 1
                      1) open all pins as outputs
                      2) put data on bus
                      3) flip strobe on
                      4) Give pic 10ms to read data
               */

               // 1
               data[0] = openGPIO(GP_4, GPIO_DIRECTION_OUT);
               data[1] = openGPIO(GP_5, GPIO_DIRECTION_OUT);
               data[2] = openGPIO(GP_6, GPIO_DIRECTION_OUT);
               data[3] = openGPIO(GP_7, GPIO_DIRECTION_OUT);

               writeBus(input & 0xF, data);          // 2
               writeGPIO(strobe, HIGH);              // 3
               usleep(10000);                               // 4


               /*END STEP 1*/

               /*STEP 2 -- read data from PIC*/
               flag = 0;
               response = 0;
               while ((flag < 4 && input == MSG_GET) || flag < 1) {
                      // if msg_get, read 4 times
                      // else, just read response

                      /*
                      READ FROM PIC
                      1) bring strobe low
                      2) remove data from bus
                      3) make pins inputs after closing them
                      4) give PIC some auxiliary some extra time to generate
                      response
                      5) raise strobe high
                      6) give PIC time to convert pins from inputs to outputs
                      7) read bus
                      */

                      writeGPIO(strobe, LOW);
       // 1
                      writeBus(0, data);
       // 2

                      // 3
                      closeGPIO(GP_4, data[0]);
                      closeGPIO(GP_5, data[1]);
                      closeGPIO(GP_6, data[2]);
                      closeGPIO(GP_7, data[3]);
                      data[0] = openGPIO(GP_4, GPIO_DIRECTION_IN);
                      data[1] = openGPIO(GP_5, GPIO_DIRECTION_IN);
                      data[2] = openGPIO(GP_6, GPIO_DIRECTION_IN);
                      data[3] = openGPIO(GP_7, GPIO_DIRECTION_IN);

                      usleep(2000);
       // 4
```

```
                writeGPIO(strobe, HIGH);                      // 5
                usleep(2000);
// 6

                // 7
                if (input == MSG_GET) {
                        response += readBus(data) << (4 * (3 - flag));
// 7 + extra
                } else {
                        response = readBus(data);
                }

                ++flag;

        }
        /*END STEP 2*/
```