Section 1: General Lab Info */0.5 points*

Microprocessors II and Embedded Systems

EECE 4800

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

Prof. Yan Luo, TA. Ioannis Smanis

Group # 1

Andrew MacGregor

Hand in: October 30, 2017

Due: October 30, 2017

Section 2: Contributions */1 points*

1. Group Member 1 - Andrew MacGregor (me)

Wrote GPIO API for Galileo

- `setGPIOMode()`, `setGPIODirection()`, `openGPIO()`, `writeGPIO()`, `readGPIO()`

Wrote strobe bus interface for Galileo

- `readNibble()`, `writeNibble()`

Implemented sent commands in the Galileo.

- `reset()`, `ping()`, `adc_value()`, `servo_30()`, `servo_90()`, `servo_120()`

Debugging:

- Debugged/tested all of the Galileo GPIO code.
- Fixed a few bugs in Galileo's `main()`
- Fixed the bus timing in `sendADCResults()`, on the PIC.
- Used Logic Analyzer to work out bus timing

2. Group Member 2 - Grayson Colwell

Wrote all of the code that is on the PIC

- `servoRotate0()`, `servoRotate30()`, `servoRotate90()`, `servoRotate120()`, `set_receive()`, `receive_msg()`, `sensorReset()`, `ADV_Init()`, `ADC_conversion_results()`, `sensorPing()`, `sendADCResults()`, `main()`

Debugging:

- Fixed the bus timing in `sendADCResults()`, on the PIC.
- Fixed the bus timing in `sendADCResults()` on the pic
- Used the Logic Analyzer to work out bus timing.
- Identified and corrected general bus timing issues in `readNibble()` and write`Nibble()`,

3. Group Member 3 - Jose Velis

Wrote Galileo `main()`

- Configures GPIO ports, handles user input, dispatches events etc.

Circuit design

- Designed ADC and LED circuiit

Debugging:

- Used the Logic Analyzer to work out bus timing.
- Identified and corrected general bus timing issues in `readNibble()` and write`Nibble()`,

Section 3: Purpose */0.5 points*

The purpose of this project was to understand how to interface two devices together with a strobe bus. An Intel Galileo board was used to interface with the sensor created in Lab1. The goal was to understand how the strobe bus works at a low level (with timing diagrams included). In the process, one has to understand how to use the GPIO interface on the Galileo.

Section 4: Introduction */0.5 points*

The final deliverable of the project is a master-slave system, consisting of the PIC sensor unit as the slave, and the Galileo as the master. The devices are connected on a strobe-type bus that is defined in the Lab2.pdf handout. The Galileo is able to ping the PIC and command it to move its servo motor to different positions. The Galileo can also prompt the PIC to send the value read by its analog to digital converter (ADC). A user controlling this system is able to access these commands through a command line application running on the Galileo

Section 5: Materials, Devices and Instruments */0.5 points*

- PIC16F18857 at 3.3V
- PICkit3 using low voltage programming
- LED
- Servo Motor
- 330Ω resistor
- 1kΩ resistor
- Photoresistor
- Breadboard & wiring kit
- Intel Galileo with wifi adaptor
- 5 wires to connect the Galileo and PIC
- Analog Devices logic analyzer
- FTID/serial cable

Section 6: Schematics */0.5 points*

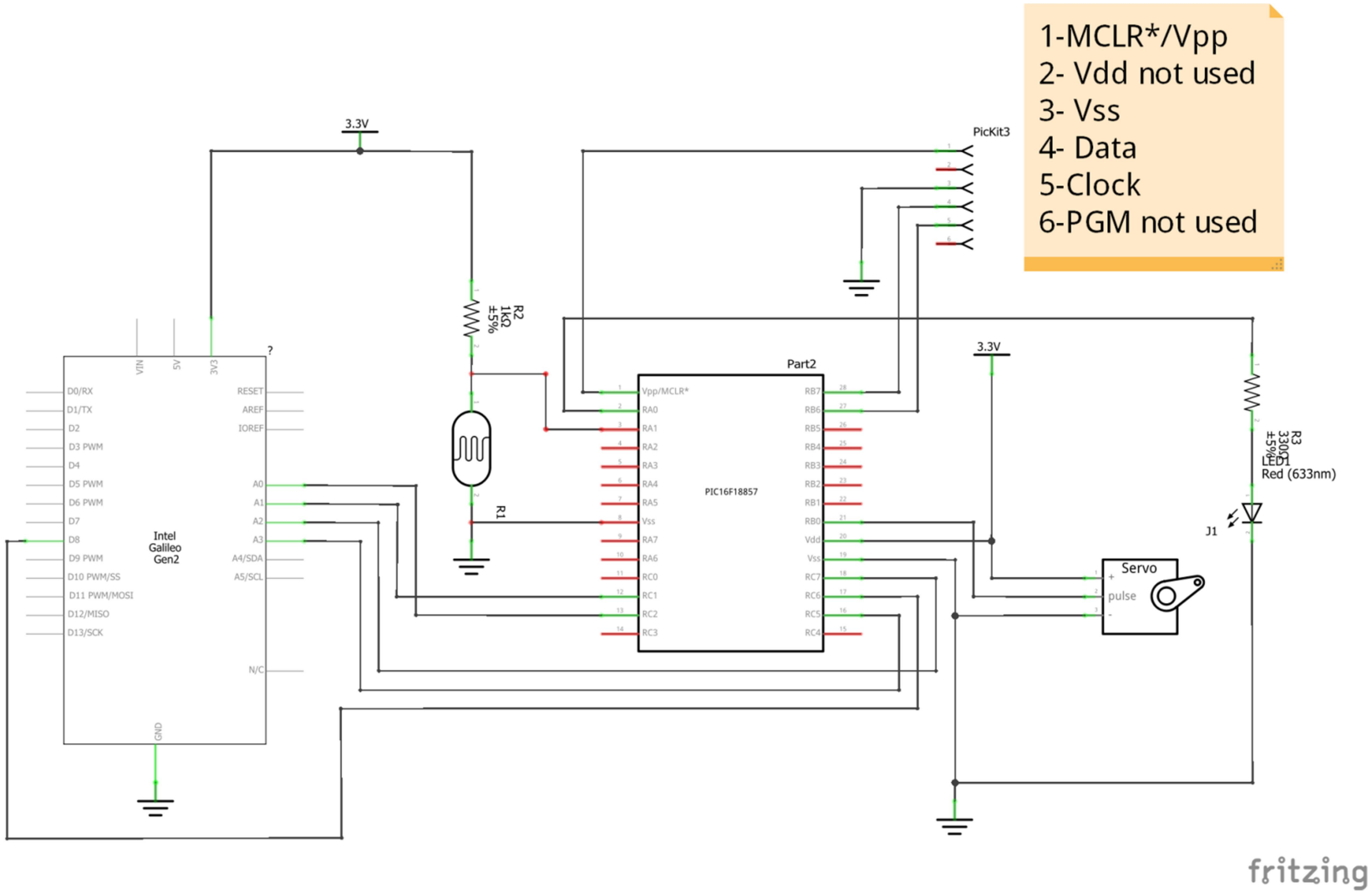


Figure 1: Complete wiring diagram of the system

Section 7: Lab Method and Procedure /2 points

Hardware design:

LED:

The LED was connected in series with a 330Ω resistor in between an output port of the PIC and ground. The 330 Ω resistor was chosen so the LED would shine without burning out. 3.3V was selected as Vdd so none of the components would be damaged at 5V.

Servo:

The servo was connected as follows. The red wire was connected to Vdd, the brown wire was connected to ground, and the orange wire was connected to a square wave PWM signal at 50Hz from an output port from the PIC. The brown wire was grounded. We controlled the servo position with

Photoresistor:

The Photoresistor was connected in series with a 1k Ω resistor in between Vdd and ground. The node between the Photoresistor and the 1k Ω was connected to an ADC input port. The photoresistor's resistance was measured to vary between about 800 Ω and 3kΩ depending on the intensity of the light. 1k Ω was chosen as the series resistor because it lies somewhere between 800 and 3k Ω. The voltage input to the ADC was calculated using the voltage divider equation below.

$$V_{ADC\ in} = R_{Pr}\frac{V_{dd}}{1k + R_{Pr}}$$

PICkit3:

The PICkit was connected as specified in its datasheet with two exceptions. The reset signal was not tied to Vdd with a pull up resistor and pin 6 was left unconnected. The lack of a pull-up resistor may have caused us problems during programming, but we didn't catch our mistake until Ioannis pointed it out during the demonstration.

Bus connection:

There was no special hardware in the bus. It was just five wires connected between corresponding ports on the PIC and the Galileo. The Galileo's GPIO pins were put in high-impedance mode when reading in data and the same pins were put into "strong" mode when outputting data. This is to minimize current flow along the bus.

Software design:

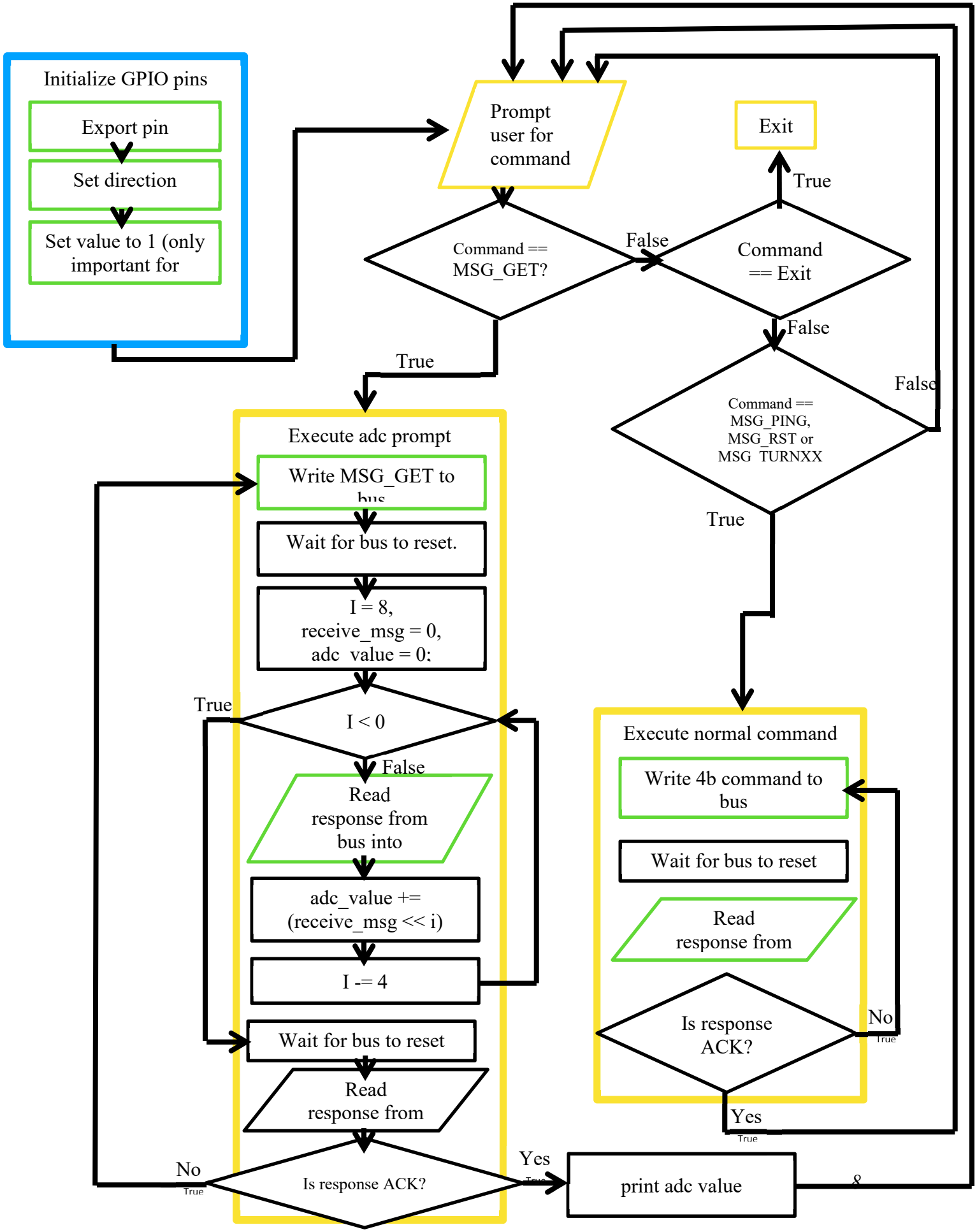
Galileo:

Since the Intel Galileo was running code on Linux, we used the sysfs interface to control the GPIO pins though the file system rather than modifying hardware registers. Controlling the GPIO pins this way required two basic steps. First, to access any of the functions for a given GPIO pin, it had to be exported first. Writing the GPIO pin number to the /sys/class/gpio/export file prompts the kernel to expose the interface for that gpio pin to userspace located in the

directory /sys/class/gpio/gpioN/. After a pin is exported, it can be manipulated by reading and writing values to the various files in its GPIO folder. We used /sys/class/gpio/gpioN/direction to control whether the pin was set to input or output and /sys/class/gpio/gpioN/mode to force the pins to enter high-impedance mode when inputting and strong mode when outputting. To read or write values from pin N, we read/wrote to the /sys/class/gpioN/value file. We used functions from stdio.h to do all of the file manipulation required to control the GPIO pins. Appendix A2 shows the code used to write to /sys/class/gpioN/value. It takes a string containing the directory of the gpio pin to write to and the value to write as parameters. the first six lines of the function uses string manipulation functions to form the full path name of the value file ("/sys/class/gpioN/value" with N substituted for the real GPIO pin), and formats the value to write as a string. The rest of the function writes the formatted value string to the file and makes sure the write was successful. Functions openGPIO(), readGPIO(), setGPIODirection() and setGPIOMode() are very similar.

The main() function first exports all needed GPIO pins, then prompts the user to enter a command to send a message to the PIC. The main function sends a command based on the user input. All command-sending functions have the same structure except for the function requesting the PIC's ADC value. This function sends one message containing MSG_GET, the code for this command, and expects to receive four messages from the PIC. The first three messages it expects are three nibbles containing the PIC's ADC value in big endian order. Then it expects an ACK message from the PIC. If an ACK isn't received, then the transfer restarts. Appendix A3 shows the Galileo code that handles a MSG_GET command. All other commands involve one sent message (the command code) and one received message (an ACK indicating completion). If an ACK is not received, the transfer is restarted. Bus read-write functions are explained in detail at the end of this section.

The application logic of the main function is illustrated in the following flow-chart. Green boxes represent functions covered by the GPIO/bus control functions.


Initialize GPIO pins
Export pin
Set direction
Set value to 1 (only important for
Prompt user for command
Command == MSG_GET?
False
Command == Exit
True
Exit
False
Command == MSG_PING, MSG_RST or MSG_TURNXX
False
True
True
Execute adc prompt
Write MSG_GET to bus
Wait for bus to reset.
I = 8, receive_msg = 0, adc_value = 0;
I < 0
True
False
Read response from bus into
adc_value += (receive_msg << i)
I -= 4
Wait for bus to reset
Read response from
Is response ACK?
No
True
Yes
True
Execute normal command
Write 4b command to bus
Wait for bus to reset
Read response from
Is response ACK?
No
True
Yes
True
print adc value
8

Appendix A1 shows the writeNibble() function that shows how the Galileo writes a command to the bus using the right timing. When not in use, the strobe line of the bus is high. The first thing the Galileo does is pull the strobe low to initiate a bus operation. Next, it outputs the data onto the four data lines. Once the data is written to the bus, the Galileo pulls the strobe line high and waits at least 10ms to signal that the data is ready to be read. Next, the Galileo pulls the strobe line low to indicate the end of the write and clears the data lines. After waiting for more than 10ms, the Galileo pulls the bus high again to reset the bus. The readNibble() function works in almost the same exact way, but with the PIC outputting to the data lines instead.

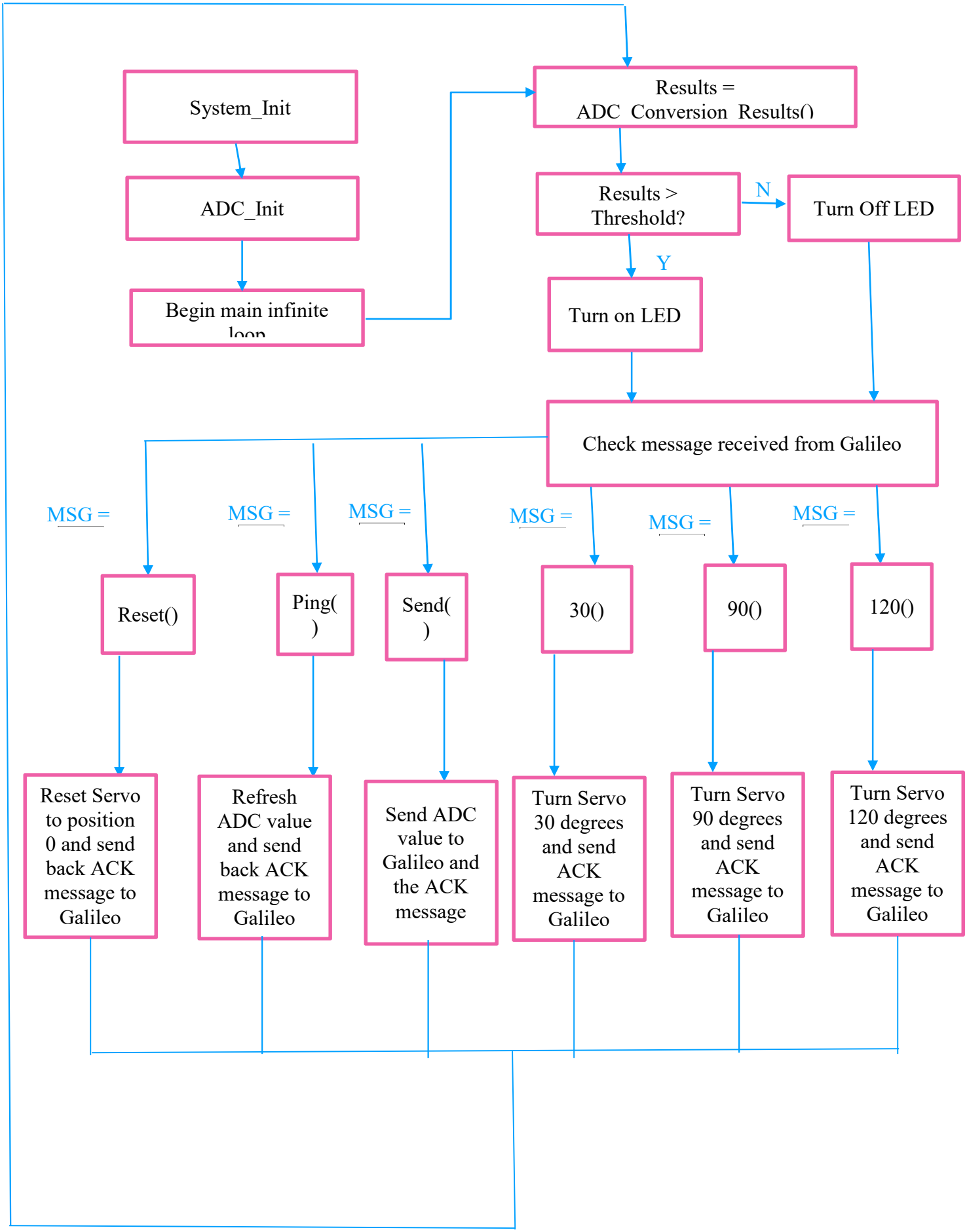
PIC:

Because the PIC used all of the functions from Lab1 (servo and ADC), much of the code was recycled from Lab1. The ADC was configured to take inputs from ANA1 using the ADPCH channel, and the input port A1 was configured for analog input using the ANSELA and TRISA registers. Using the ADCON0 register, the ADC results were right justified and the modified was enabled.

Also, the servo's PWM control was recycled from Lab1. It uses delay cycles to generate 50 cycles of a 20ms period signal with variable duty cycle from 1.1ms to 2.1ms high periods. We chose to use 50 cycles of the signal because we found that was enough time to move the servo motor from any one position to another. Appendix A4 shows an example of this servo code that moves the rotor to a neutral position.

The PIC's main() function initializes the bus GPIO pins then enters an infinite loop of waiting for a command from the bus then executing a command. Each command function performs an action, then finishes by sending an ACK message to the bus. The function returning the adc value from the PIC sends three messages before the ACK containing the adc value broken up into nibbles. The first message contains 00XX, where X are the two most significant bits of the 10-bit adc value. The next message contains the next four most significant bits of the adc value and the last message contains the rest of the adc value. Appendix A1 shows how the Galileo handles bus timing, and Appendix A5 shows the most succinct function that demonstrates how the PIC handles bus timing when receiving a ping message. The most important parts of the PIC bus code are the several while loops that wait for RC6 (the strobe pin) to change, indicating a change in bus state. Without these controlled waits, the PIC would be completely out of sync with the Galileo and would be unable to communicate.

The following page shows a flowchart describing the PIC software.


System_Init
ADC_Init
Begin main infinite loop
Results = ADC_Conversion_Results()
Results > Threshold?
N
Turn Off LED
Y
Turn on LED
Check message received from Galileo
MSG =
MSG =
MSG =
MSG =
MSG =
MSG =
Reset()
Ping( )
Send( )
30()
90()
120()
Reset Servo to position 0 and send back ACK message to Galileo
Refresh ADC value and send back ACK message to Galileo
Send ADC value to Galileo and the ACK message
Turn Servo 30 degrees and send ACK message to Galileo
Turn Servo 90 degrees and send ACK message to Galileo
Turn Servo 120 degrees and send ACK message to Galileo

Section 8:Troubleshooting /1 points

Issue 1: PIC programming issues

We experienced trying to program the PIC from my laptop. Despite using a benchtop power supply and having no components connected to the PIC, MPLAB X would be unable to program the PIC and would say that the target circuit may draw more power than the PICKit can provide.

We solved this issue by switching to a Lab computer and making sure to use a USB2.0 port. We found that USB3.0 ports were unreliable.

Issue 2: Broken pin on PIC

During Lab 1, I accidentally broke off pin C6 of the PIC trying to remove it from the breadboard, and forgot about it. During initial phases of this lab, we noticed that this pin was missing, and chose to use a different pin on the C port instead.

Issue 3: Bus timing mismatch between PIC and Galileo

Most of the time spent debugging our circuit was spent debugging the interface between PIC and Galileo. We were experiencing issues such as the PIC not reading the Galileo's command, the Galileo receiving gibberish instead of an ACK from the PIC and other communication breakdowns.

We ultimately solved the issue by using the Analog Devices logic analyzer to confirm that each computer was behaving the way it was expected to. When we realized that the PIC and Galileo expected a slightly different sequence of events on the bus, we restructured the bus code on both the PIC and Galileo to make sure that they explicitly expected the same events.

Issue 4: Errors resulting from consecutive servo commands

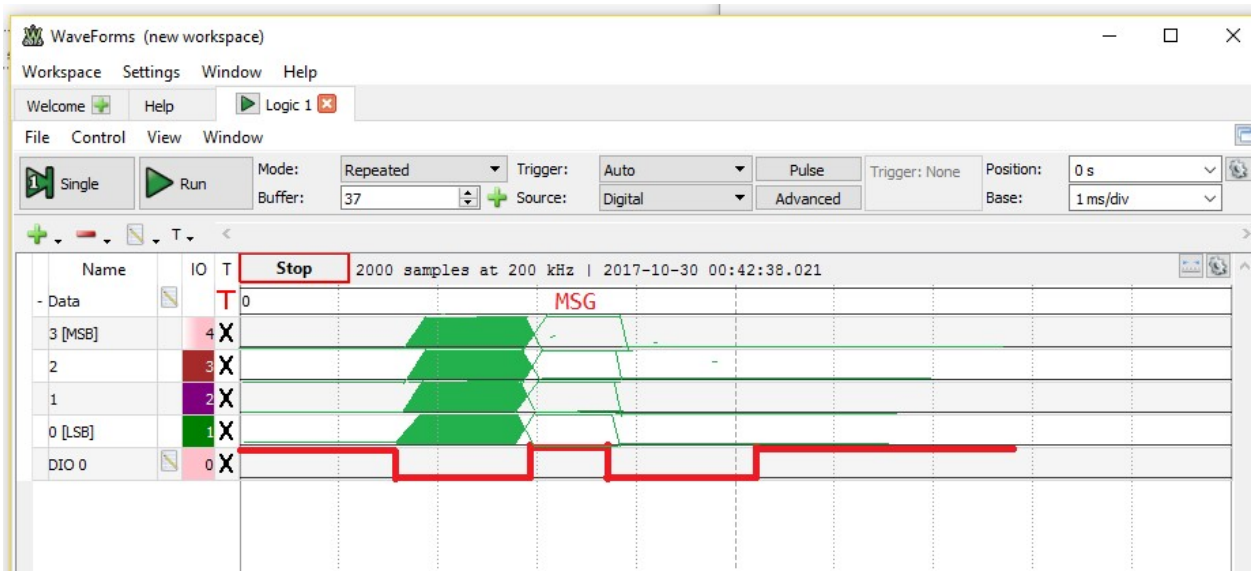
We found that if the Galileo sent a servo command and then another command quickly, the second command would not be executed. This is because the servo function uses delay commands and bit-banging to make a PWM signal. 50 cycles of a 20ms signal takes 1s to complete. During that interval, the bus isn't monitored for any new activity, so a command sent will get ignored.

We solved this issue by ignoring it in the code. During the demo we made sure to space out our commands by at least a second. I'd imagine that the only way to fix this would be to use the PIC's PWM module to control the servos instead of delays. However, we tried to take this route during Lab1 and it took too long, so until we come to a point where we can't use delay functions to control the servo, we won't change that code.

Section 9: Results */0.5 points*

In this lab, the most tangible results were seeing the PIC respond to each individual command. For reset and each of the servo messages, we could verify that the system was working by observing the servo move and seeing the Galileo receive an ACK from the bus. For the ping message, we connected an LED to turn on when the message was received during the debugging session. The ADC value was easy to see results because we kept the same circuit from Lab1. In Lab 1, we found that the ADC read a value greater than 912 if the Photoresistor was covered, and a value less than 912 if the Photoresistor was exposed. We tried sending the get_adc message with the Photoresistor covered and uncovered and saw that the response was reasonable.

We used a logic analyzer to test our output signal. We don't have any screenshots of us debugging the actual signal, but the following is a screenshot with an explanation of the functioning bus.



The bottom signal with the thick red line shows where the strobe signal would be. The four green signals above represent D0-D3. The signal on top shows the value of the nibble on the bus as a whole. Once strobe is pulled low, the data would be written to D0-D. Once strobe pulses high, D0-D3 become frozen, and the message on the bus is read into whatever device is receiving. Afterwards, the bus clears and is reset.

References:

[1] https://www.kernel.org/doc/Documentation/gpio/sysfs.txt

Section 10:Appendix

A1. WriteNibble function, explaining Galileo bus code

```
void writeNibble(unsigned char data,
   char* d0,
   char* d1,
   char* d2,
   char* d3,
   char* strobe)
{
   //set all the ports to output
   setGPIODirection(d0, GPIO_DIRECTION_OUT);
   setGPIODirection(d1, GPIO_DIRECTION_OUT);
   setGPIODirection(d2, GPIO_DIRECTION_OUT);
   setGPIODirection(d3, GPIO_DIRECTION_OUT);
   setGPIODirection(strobe, GPIO_DIRECTION_OUT);

   //start the bus protocol
   //1: pull strobe low
   writeGPIO(strobe, LOW);

   //2: output the nibble to the bus
   writeGPIO(d0, data & 1);
   writeGPIO(d1, (data & 2) >> 1);
   writeGPIO(d2, (data & 4) >> 2);
   writeGPIO(d3, (data & 8) >> 3);

   //3: raise strobe and wait at least 10ms
   writeGPIO(strobe, HIGH);
   usleep(STROBE_DELAY);

   //4: Pull strobe low again
   writeGPIO(strobe, LOW);
   usleep(STROBE_DELAY); //and delay a little bit

   //5: clear the bus
   writeGPIO(d0, LOW);
   writeGPIO(d1, LOW);
   writeGPIO(d2, LOW);
   writeGPIO(d3, LOW);

   //....let the bus float high again
   writeGPIO(strobe, HIGH);
   usleep(STROBE_DELAY); //and delay a little bit
}
```

A2. WriteGPIO, explaining Galileo GPIO interface

```
int writeGPIO(char* gpioDirectory, int value)
{
```

```
    char gpioValue[BUFFER_SIZE];
    strcpy(gpioValue, gpioDirectory);
    strcat(gpioValue, "value");
    FILE* valueFh = fopen(gpioValue, "w");

    char numBuffer[5];
    snprintf(numBuffer, 5, "%d", value);

    int n = fputs(numBuffer, valueFh);
    fclose(valueFh);
    if(n < 0)
    {
        printf("Error writin to gpio value file in %s", gpioDirectory);

        return ERROR;
    }

    return value;
}
```

A3. Galileo MSG_GET function

```
void adc_value()
{
    int i;
    int receive_msg = 0;
    int adc_value = 0;
    while(receive_msg != MSG_ACK)
    {
        adc_value = 0;
        receive_msg = 0;
        printf("Starting to send ADC_request\n");
        writeNibble(MSG_GET,
                fileHandleGPIO_4,
                fileHandleGPIO_5,
                fileHandleGPIO_6,
                fileHandleGPIO_7,
                fileHandleGPIO_S);

        //expect to receive 3 messages containing ADC values, msn first
        usleep(STROBE_DELAY);
        for(i = 8; i >= 0; i -= 4)
        {
            receive_msg = readNibble(fileHandleGPIO_4,
                            fileHandleGPIO_5,
                            fileHandleGPIO_6,
                            fileHandleGPIO_7,
                            fileHandleGPIO_S);
            adc_value += ((unsigned) receive_msg) << i;
            printf("Received ADC Nibble: 0x%x\n", receive_msg);
            usleep(STROBE_DELAY);
        }
```

```
        //expect one last ACK message
        receive_msg = readNibble(fileHandleGPIO_4,
                                 fileHandleGPIO_5,
                                 fileHandleGPIO_6,
                                 fileHandleGPIO_7,
                                 fileHandleGPIO_S);
        printf("Received ADC ACK(?): 0x%x\n", receive_msg);
        usleep(STROBE_DELAY);
    }
    printf("adc message received successfully: 0x%x\n", adc_value);
}
```

A4. PIC servo code

```
void servoRotate0() //0 Degree -> reset servo position
{
  unsigned int i;
  for(i=0;i<50;i++)
  {
    PORTB = 1;
    __delay_ms(1.4);
    PORTB = 0;
    __delay_ms(18.6);
  }
}
```

A5. PIC bus interface code

```
void sensorPing()
{
    printf("PING\n");
    //wait for the bus to be pulled low
    while(PORTCbits.RC6 == 1)
    {

    }

    TRISC = 0b01000000;
    //send the ACK message to the galileo
    PORTCbits.RC2 = 0;
    PORTCbits.RC1 = 1;
    PORTCbits.RC7 = 1;
    PORTCbits.RC5 = 1;
    //wait for the Galileo signal its ready to receive (Strobe High)
    while(PORTCbits.RC6 == 0)
    {

    }
    //Let the strobe go back low
    while(PORTCbits.RC6 == 1)
    {
```

```
    }
    //Wait for the bus to reset
    while(PORTCbits.RC6 == 0)
    {

    }
}
```