Section 1: General Lab Info /0.5 points

Microprocessors II and Embedded Systems

EECE 4800

Lab 4: Multithreaded Programming

Prof. Yan Luo, TA. Ioannis Smanis

Group #1

Andrew MacGregor

Hand in: December 20, 2017

Due: December 22, 2017

Section 2: Contributions /1 points

For future versions of this course, could you use git to see exactly who contributed what?

1. **Group Member 1 - Andrew MacGregor (me)**

Wrote & tested Thread #3 (HTTP POST loop)
Wrote new PIC code
    Added PWM control with Timer2
    Added Timer0 counter to increment pwm duty cycle in scan mode
    Added logic for new scan mode bus commands
    Updated bus code from lab 2 to work with new PWM module
Installed required libraries on Galileo
Wrote Makefile
Debugged mutexes + thread interaction
    It turns out you need a mutex to guard the 5-wire bus when two threads can access it
Final debugging and polishing

2. Group Member 2 - Grayson Colwell

Wrote logic for thread #1 (user input thread)
    Added printf menu and scanf + switch dispatcher
Wrote logic for thread #2 (sensor interface thread)
    Continuously probe i2c and pic adc sensors and update their shared value
    Triggers a picture taken if above a threshold.

3. Group Member 3 - Jose Velis

Figured out i2c bus logic.
Wrote & tested i2c temp sensor access code
Helped debug thread #2.
Drew schematic

Section 3: Purpose /0.5 points

The purpose of this project was to understand how to design a multithreaded program for an embedded system. Specifically, POSIX thread libraries were used to create different threads and synchronize events. Also, basic client-server communication was demonstrated with HTTP POST method using libcurl. Indirectly, this lab demonstrated the importance of writing maintainable code from the past labs.

Section 4: Introduction /0.5 points

The final deliverable of this project is an application on the Galileo that monitors sensors on its busses and updates a server with data collected. Additionally, the Galileo prompts a user to enter commands to more specifically control a webcam mounted on a servo. If the attached temperature sensor reads above a user-defined threshold, then the webcam takes a picture and the filename of the most recently taken picture is sent to the remote server.

Section 5: Materials, Devices and Instruments /0.5 points

- Intel Galileo with wifi adaptor and with OpenCv and libmraa installed (IO pins set to 3V3)
- PIC microcontroller (3V3)
- LED
- light dependent resistor
- FTID/serial cable
- Breadboard
- Temperature sensor (TMP 102)
- Assorted wires and resistors
- USB 2.0 webcam
- servo motor
- Analog Discovery oscilloscope/logic analyzer (for debugging)
- pizza box and duct tape (for presenting the project)

Section 6: Schematics /0.5 *points*

The PIC was connected to the servo and LDR like in Lab1. Additionally, the PIC and temperature sensor were connected to the Galileo with the same connections as in Lab 2 and Lab 3 respectively. The Galileo and PIC share a ground connection. This circuit is essentially made of recycled designs from previous labs.

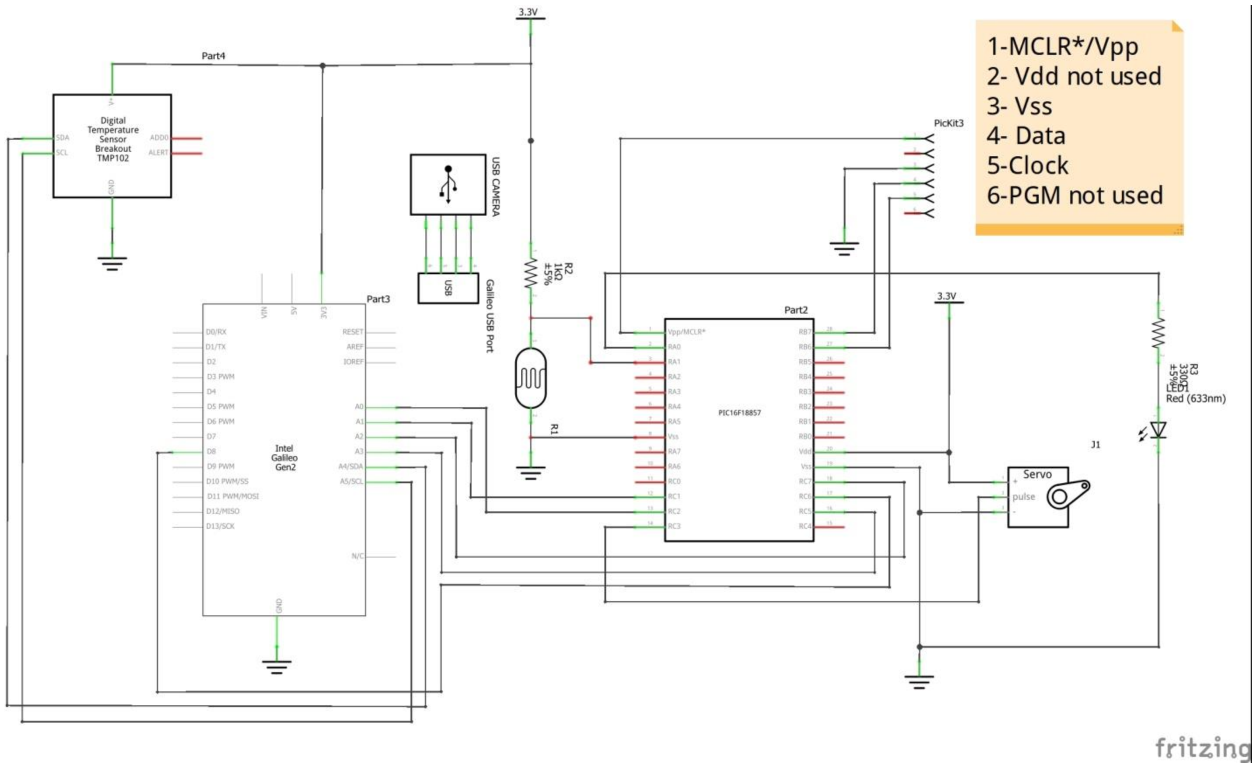


Figure 1: *Schematic for Lab 4 circuit*

Hardware design:

New hardware design was minimal for this lab. All circuit components were recycled from earlier labs. The only major change was the port selection on the PIC. The PWM module we selected output on RC3 which is what we were using for the D1 line of the strobe bus, so we changed D1 to use RC1 and changed the servo's PWM connection to RC3.

Like before, the LDR was connected to the PIC through a voltage divider and an LED was connected to the PIC. Additionally, the temperature sensor was connected to the Galileo's I2C port without external pullup resistors. The five-wire strobe bus was logically connected the same as in Lab2.

Software design:

I^{2}C temperature sensor

We used i2c-detect to verify the temperature sensor's address. i2c-detect output is shown in Appendix A1.

We reused our code from Lab 3 to control the i2c bus. Using the mraa library, the code to read the temperature sensor is very small. The code snippet to read from the sensor is shown in Appendix A2.

Opencv/Webcam

We reused the opencv code from Lab 3 to control the webcam. We used opencv to access the webcam and save an image. We used C library functions to format a timestamp based filename. The opencv code snipped it shown in Appendix A3.

HTTP POST

We used the provided libcurl template code and modified it slightly to send our HTTP message. First we used the sensor state information to format a url that delivers the correct information. Formatting code is shown in Appendix A5. Then we used curl in easy mode to to send the default HTTP message to the formatted url at the AWS server (I think this code actually sends a GET, but it still works with the server that was set up). The curl code is shown in Appendix A4.

PWM and scan mode

The PWM module was configured on the PIC to perform PWM. The capture&compare module CCP1 was used to generate PWM. CCP1 outputs to port RC3, and uses the TMR2 module for timing. First RC3 was configured for digital output. and timer2 was directed to input to CCP1 using the CCP1CON register. The PWM frequency and duty cycle were calculated with the following equations. We used an oscillator frequency of 1/4MHz and a TMR2 prescale value of 64.

$$PWM\ period = 20ms = (PR2 + 1) * 4 * 1 \frac{}{XTAL_{FREQ}} * TMR2Prescale$$

$$PR2 = 20ms/(\frac{4}{\_XTAL\_FREQ} * 64) \approx \mathbf{77}$$

$$Pulse\ width = CCPR1H{:}CCPR1L * \frac{64}{\frac{\_XTAL\_Freq}{4}}$$

$$Pulse\ width = 1ms, CCPR1H{:}CCPR1L = \mathbf{15}$$

$$Pulse\ width = 2ms, CCPR1H{:}CCPR1L = \mathbf{31}$$

These equations found a value to assign to PR2 and an appropriate range of CCPR1L values to modify the duty cycle with. We were interested in pulse widths between 1ms and 2ms because the servo datasheet specified maximum rotation at those two values.

Next, TMR2 was configured to use FOSC/4 as its input source and to use a 64x prescaler. If TMR2's input is not set to FOSC/4 it will not work with the CCP module for PWM. Lastly the TMR2 interrupt flag was cleared and TMR2 was started. Timer2 and CCP setup code is shown in Appendix A6.

Scan was achieved by incrementing the pulse width intermittently. Timer0 was set to trigger its interrupt flag at about 1Hz. In our code, we manually checked for Timer0's interrupt flag, and if it was set, we stepped the servo position up or down by one increment. It would be way better to use an interrupt to do this work. The upper and lower bounds of the servo scan are able to be set to either ±180 or ±90 by sending a message from the Galileo.

An LED on the PIC indicates whether or not the PIC is in scanning mode.

<u>Strobe bus connection</u>:

The code for the strobe bus was reused from Lab 2. On the Galileo side, it consisted of GPIO writes, reads, and delays in a specified order. The PIC waited for events on the strobe pin, read commands when specified, and wrote back to the bus if an ACK or adc data was expected on the bus.

The PIC's strobe code was slightly changed from Lab2. In Lab2, delay functions were used to do PWM to move the server. These delay functions helped keep the PIC out of states that would leave it unresponsive to bus commands from the Galileo. Small delay commands were added to the end of every PIC bus interface to do the same.

<u>Multithreading:</u>

The code on the Galileo was divided between three threads. Thread 1 took commands from a user and dispatched events (on the same thread). Thread 2 continuously probed the PIC for its ADC value and the temperature sensor for its temperature value. If the temperature sensor surpassed some temperature threshold, Thread 2 takes a picture with the webcam. Thread 3 continuously formats the sensor's state data into a url and uses curl to post it to the server.

All three threads are created at startup. There are two mutexes in the code. One mutex protects the sensor's state information from a race condition between threads trying to read and update state at the same time. The other mutex protects the strobe bus from simultaneous access. Simultaneous strobe bus access disrupts the timing on the bus and garbles the bus data.

Galileo Flow chart:
Orange = Thread 1 (user control)
Blue = Thread 2 (continuous sensor read)
Green = Thread 3 (HTTP POST)
red text = critical section accessing shared state

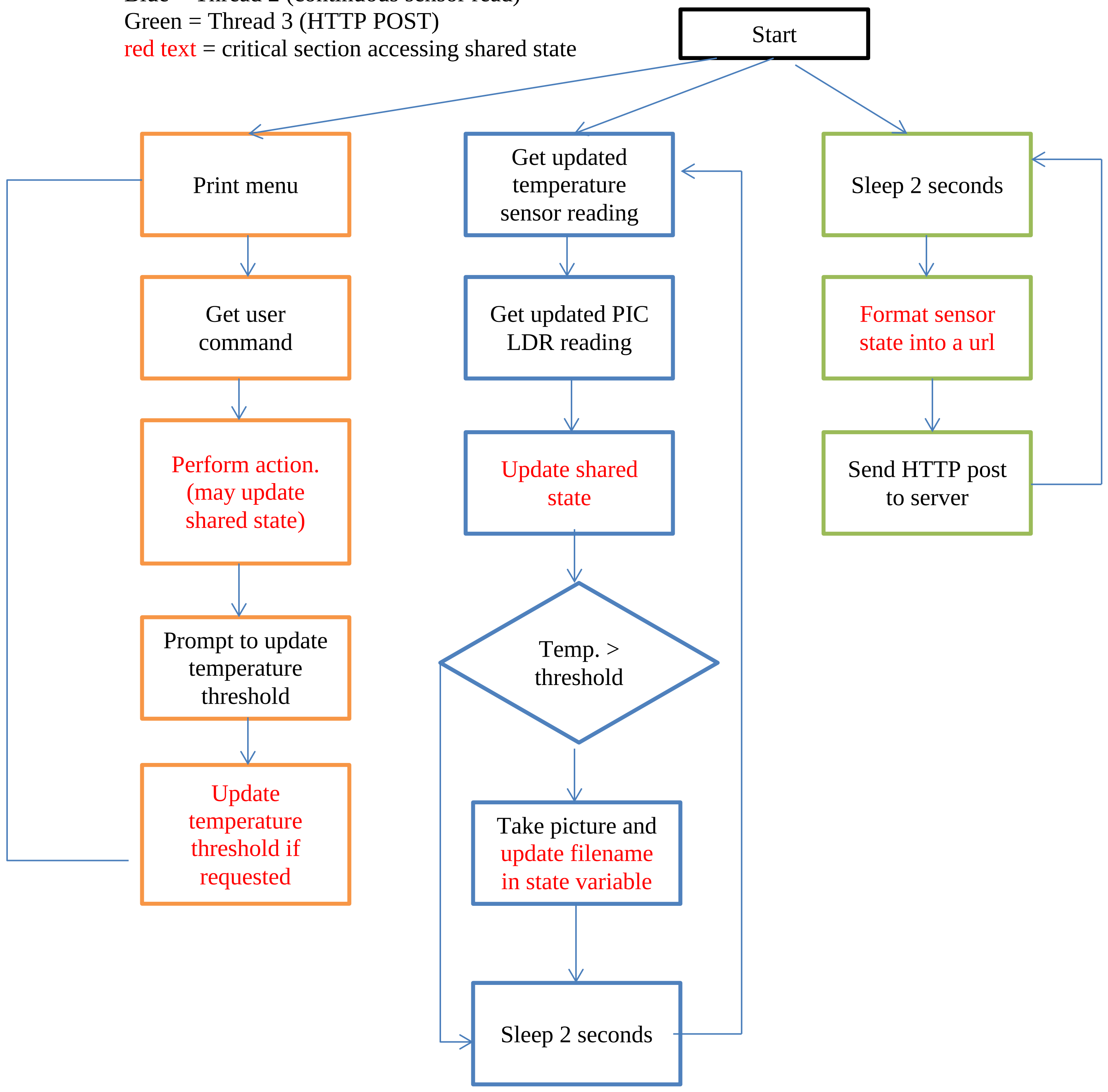

Start
Print menu
Get user command
Perform action. (may update shared state)
Prompt to update temperature threshold
Update temperature threshold if requested
Get updated temperature sensor reading
Get updated PIC LDR reading
Update shared state
Temp. > threshold
Take picture and update filename in state variable
Sleep 2 seconds
Sleep 2 seconds
Format sensor state into a url
Send HTTP post to server

***Figure 2**: Flow chart showing Galileo thread structure*

Section 8: Trouble Shooting /1 points

Issue 1: We were having trouble using delay functions to implement scanning mode on the PIC

We used the PIC's CCP module to do PWM without delay functions

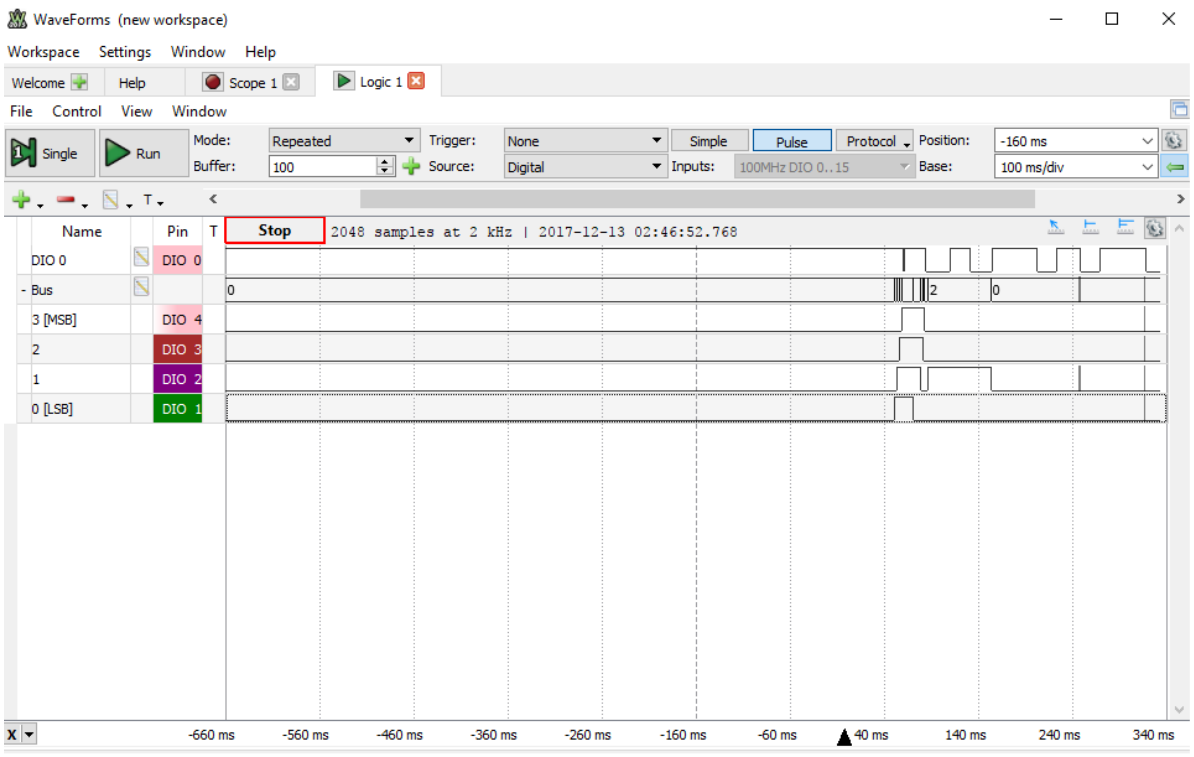
Issue 2: We were having trouble getting any CCP module to work.

The datasheet instructions have to be followed completely. We initially weren't using FOSC/4 as the input source for TMR2 (it isn't the default source). The PWM module wouldn't output until we changed the TMR2 oscillator source to FOSC/4.

Issue 3: We were having trouble getting the CCP4 module to work, but CCP1 worked fine.

CCP4 output to an unused pin so we wanted to use it but couldn't get it to work. We used CCP1 instead and moved D1 to a different pin.

Issue 4: When initializing the Galileo's GPIO pins, a pulse goes out to the strobe bus that put the PIC in a state that prevented it from working correctly.


WaveForms (new workspace)
Workspace Settings Window Help
Welcome Help Scope 1 Logic 1
File Control View Window
Single Run
Stop
Name Pin T
-660 ms -560 ms -460 ms -360 ms -260 ms -160 ms -60 ms 40 ms 140 ms 240 ms 340 ms

Mode: Repeated
Buffer: 100
Trigger: None
Source: Digital
Simple Pulse Protocol
Inputs: 100MHz DIO 0..15
Position: -160 ms
Base: 100 ms/div

2048 samples at 2 kHz | 2017-12-13 02:46:52.768

Name	Pin
DIO 0	DIO 0
- Bus	
3 [MSB]	DIO 4
2	DIO 3
1	DIO 2
0 [LSB]	DIO 1

Figure 3: *Logic Analyzer screen capture showing the pulse at the beginning of the signal when the Galileo initializes its GPIO pins.*

First, we disconnected the PIC from the strobe bus every time we initialized the Galileo's GPIO pins.

After solving issue 5, this issue went away.

Issue 5: The PIC would sometimes get into a state where it wouldn't follow the bus protocol and wouldn't leave that state.

We found that the delay code to control the servo in Lab 2 prevented this state from occurring. We added a small delay to every PIC function to prevent this state.

Issue 6: We couldn't get the timer0 interrupt handler to work.

We added an if statements to check if the timer0 interrupt flag was set while waiting for the strobe bus state to change.

Issue 7: *When first using i2c-detect, the temperature sensor wasn't showing up.*

The PIC and the Galileo didn’t share a ground connection so the temperature sensor didn’t recognize the Galileo’s voltage levels.

After adding a common ground, the temperature sensor communicated with no issues.

Issue 8: *Strobe bus occasionally was garbled.*

This happened when the user prompted a message to be sent to the PIC and Thread 2 probed the PIC ADC at the same time. A mutex was added to protect the strobe bus.

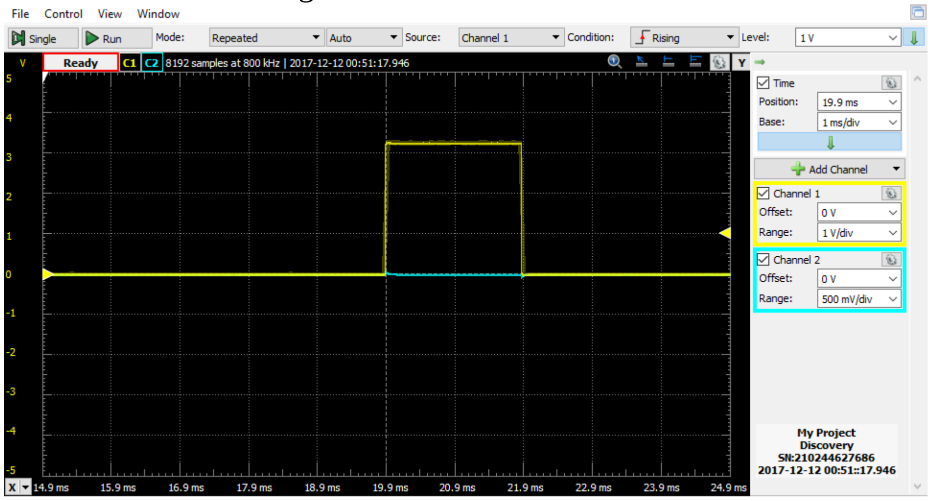
Issue 9: *PWM from CCP1 came out of pin RC3 instead of pin RC2 like the data sheet indicated*.

I think its just a typo in the data sheet. The servo was connected to RC3 instead of RC2.

Issue 10: *libcurl post wasn’t working with sample code, even with a correctly formatted url.*

I removed the line: `curl_easy_setopt(curl, CURLOPT_POST, 1);` from the sample code and it worked fine. I don’t know why this worked.

Scanning mode on the PIC correctly sweeps the PWM pulse width between minimum and maximum values. The following screenshots show scan results.



Mode: Repeated | Auto | Source: Channel 1 | Condition: Rising | Level: 1 V

8192 samples at 800 kHz | 2017-12-12 00:51:17.946

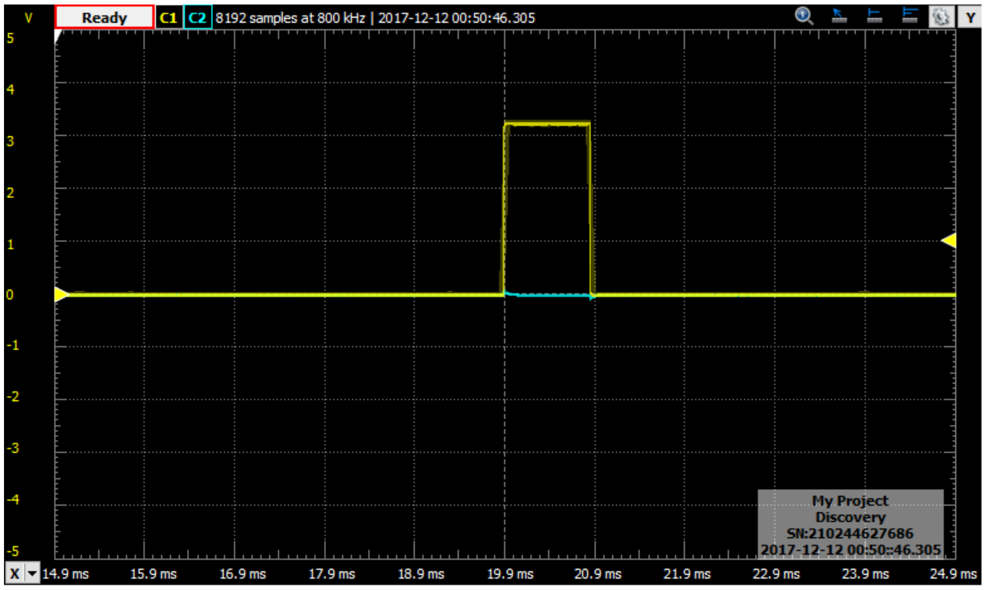
Time — Position: 19.9 ms; Base: 1 ms/div

Channel 1 — Offset: 0 V; Range: 1 V/div

Channel 2 — Offset: 0 V; Range: 500 mV/div

My Project
Discovery
SN:210244627686
2017-12-12 00:51::17.946

Figure 4: *Maxmum PWM pulse width*



8192 samples at 800 kHz | 2017-12-12 00:50:46.305

My Project
Discovery
SN:210244627686
2017-12-12 00:50::46.305

Figure 5: *Minimum PWM pulse width*

The webcam worked correctly as well. The following picture is a picture taken by the webcam in response to the temperature sensor reading a value above a set threshold.

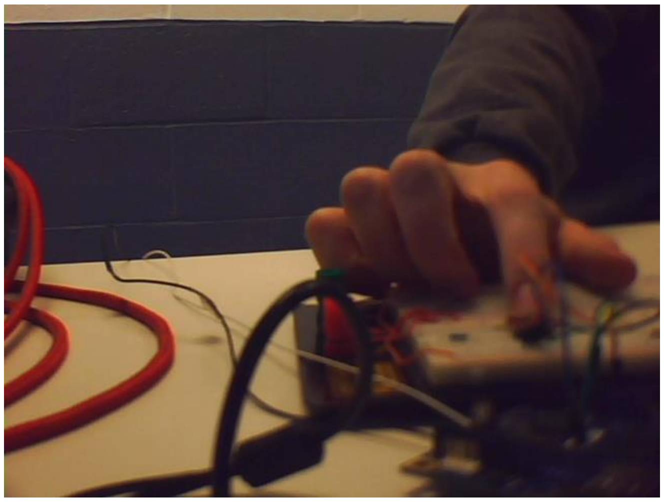
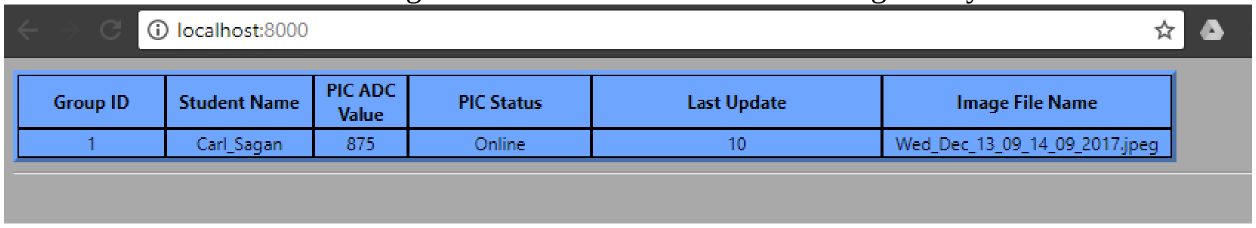


Figure 6: *Webcam-captured image of one of us with our finger on the temperature sensor.*

The libcurl messages successfully updated the remote server. The following screenshot shows the result of a test message that was sent to our server running locally.



Group ID	Student Name	PIC ADC Value	PIC Status	Last Update	Image File Name
1	Carl_Sagan	875	Online	10	Wed_Dec_13_09_14_09_2017.jpeg

Figure 7: *Updated table from a Galileo post message.*

A1. i2cdetect output with temperature sensor connected

```
root@galileo:~/Lab3# i2cdetect -y -r 0
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- UU UU UU -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- UU 48 -- -- -- -- -- -- --
50: -- -- -- -- UU UU UU UU -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: 70 -- -- -- -- -- -- --
root@galileo:~/Lab3#
```

A2. Function to get temperature

```
double get_temp(){

//create a new I2c instance on the i2c-0 interface
I2c i2c(0);

//sets this I2c instance to work with a specific address
//   TMP102Address is a constant set to 0x48
i2c.address(TMP102Address);

uint8_t dataReg [2];

//read two bytes from the registers
int buffer = i2c.read(dataReg,2);

//The data expected is a 12 bit value representing the temperature
//it is sent big endian and left justified.
int temperature = ((dataReg[0]<<8 | dataReg[1]) >> 4);

//The received temperature value is in units of C°/16
//The multiplier below converts to C°
return temperature*0.0625;
}
```

A3. Function to capture/save image (without error checking)

```
bool capture_and_save_image(char* filename)
{
  //init webcam on video0 interface
  VideoCapture ourCam;
  ourCam.open(0);
  Mat image;

  //read a frame from the vid camera into image
  ourCam.read(image);

  //make jpeg format parameters in form of Key, Value
  vector<int> compression_params;
  //indicates that the next parameter sets the JPEG quality
  compression_params.push_back(IMWRITE_JPEG_QUALITY);
  //sets JPEG quality to 95%
  compression_params.push_back(95);

  //make filename
  string fn = string(filename) + string(".jpeg");

  //try to save to file
  imwrite(fn, image, compression_params);
  return true;
}
```

A4. Curl code to send POST

```
//url is a correctly formatted url
void HTTP_POST(const char* url){
  CURL *curl;
  CURLcode res;

  curl = curl_easy_init();
  if(curl)
  {
    //set url to curl structure
    curl_easy_setopt(curl, CURLOPT_URL, url);

    //write data to stderr so it can be diverted from the user
interface.
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, stderr);

    //perform POST
    res = curl_easy_perform(curl);

    //Handle failure
    if(res != CURLE_OK)
      fprintf(stderr, "curl_easy_perform() failed:
%s\n",curl_easy_strerror(res));

    //cleanup curl object (required)
    curl_easy_cleanup(curl);
  }
}
```

A5. POST message format

```
void getPostRequest(char* buffer, int length, int picAdcValue, char*
picStatus, char* timeStamp, char* filename)
{

  //get timestamp and set member
  //YYYY-MM-DD_HH:MM:SS
  time_t date = time(NULL);
  strftime(timeStamp, MAX_TIMESTAMP, "%F_%T", localtime(&date));

  //format the whole url with one big snprintf
  int ret = snprintf(buffer,length,
"http://%s:%d/update?id=%d&password=%s&name=%s&data=%d&status=%s&times
tamp=%s&filename=%s",
    SERVER_HOSTNAME, SERVER_PORTNUMBER, GROUP_ID, PASSWORD,
STUDENT_NAME, picAdcValue,
    picStatus, timeStamp, filename);

}
```

A6. PWM module init.

```
////init PWM module
//Set whole port C to digital output
TRISC = 0;
ANSELC = 0; //ansel is 1 by default

//set frquency of PWM
//PWM period = 20ms = [(PR2) + 1]] * 4* _XTAL_FREQ * TMR2 prescale
// PR2 = 20ms / (4 * 1/_XTAL_FREQ * 64) ~= 77
PR2 =  77; //set timer2 period

//use CCP1 because it goes to RC1
CCP1CON = 0x8F; //set enabled.. in PWM mode

//load CCPRxL and CCPRxH to set the pulse width

//pulse width = CCPR4H:CCPR4L * 64 / FOSC
//duty = 0 -> PW = 1ms, PR = 15
//duty = 100 -> PW = 2ms, PR = 31
CCPR1H = 0;
CCPR1L = 25; //some value between 15 and 31

//point timer2 at CCP1
CCPTMRS0 = 0x01;

//configure + start TIMER2
T2CLKCON = 0x01; // set timer source to FOSC/4
T2CON = 0xE0; //set timer2 on; prescaler = 64

//clear TMR2IF of PIR4
PIR4 = 0;

//start timer2
T2CON |= 0x80;
```