# Section 1: General Lab Info /0.5 points



Microprocessor Systems II & Embedded Systems

EECE.4800 – 201

Laboratory 3: Building Linux Kernel and Controlling an I2C Device

Professor Yan Luo

Group #10

Hans-Edward Hoene

Derek A Teixeira

Kyle W Marescalchi

18 November 2017

Due Date: 20 November 2017

# Section 2: Contributions /1 points

1. Group Member 1 – Hans-Edward Hoene (Me)

* Set up batch files and documentation for using Ethernet to quickly move files from Windows PC to Galileo and vice versa
* Worked on I2C for communicating with TMP102 sensor
* Worked on webcam functions for capturing and saving images

1. Group Member 2 - Derek A Teixeira

* Set up hardware
* Worked on I2C for communicating with TMP102 sensor
* Worked on webcam functions for capturing and saving images

1. Group Member 3 - Kyle W Marescalchi

* Debugged errors in reading temperature (discussed in “Troubleshooting” section)
* Debugged error in reading temperature from buffers
* Researched the Open CV functions for documentation

# Section 3: Purpose /0.5 points

The purpose of this laboratory was to use I2C o Linux to communicate with a sensor and to use an open source Linux C library for using a webcam. In short, the idea is to continuously poll the TMP102 temperature sensor until the temperature crosses a specific threshold. Once that occurs, an image will be captured and stored on the SD card that is responsible for booting the Linux. The three objectives are as follows: program I2C devices from Linux using libraries and APIs, program Linux with a library to capture and store images from a webcam, and use a temperature sensor to trigger the capture of images.

# Section 4: Introduction /0.5 points

The Galileo is nothing more than a miniature computer. It boots Linux from an SD card and can be controlled via Linux terminal over putty. In this laboratory, the Galileo will be communicating with both a temperature sensor and USB webcam.

The temperature sensor is interfaced via the I2C protocol. I2C is a serial bus protocol with masters and slaves. In this laboratory, the Galileo is the master and the temperature sensor is the slave slowing it to be controlled and interfaced by a master on the network. A master initiates communication by raising signals and putting an address on the serial bus. The slave with that address will then listen in and communicate. Data is sent serially by putting a bit on SDA and raising SCL to indicate a clock cycle. Linux has C libraries for communicating via I2C without writing functions to raise and lower signals.

The webcam is connected via USB. The webcam is controlled and interfaced via the open source Open CV library. The Open CV library is used to capture and store images from the webcam. Before using this library though, it had to be installed from the internet.

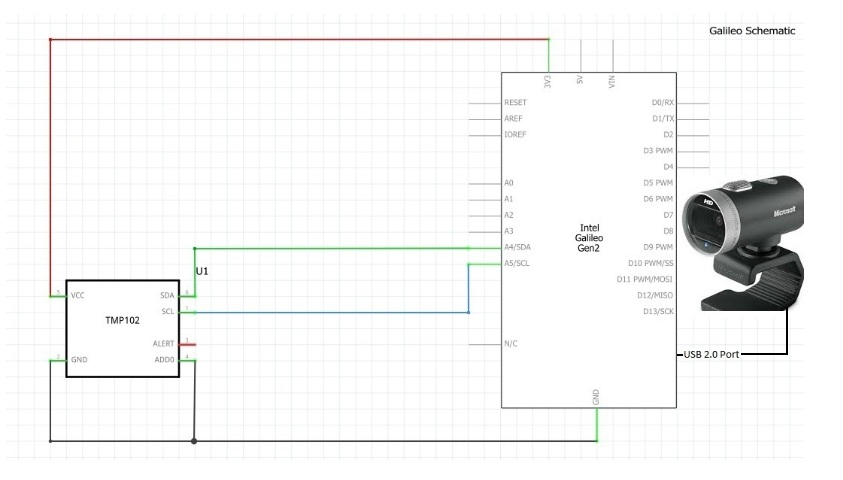
# Section 5: Materials, Devices and Instruments /0.5 points

|  |  |  |
| --- | --- | --- |
| Device Name | Model Number | Description |
| Galileo Gen2 | Intel Quark x1000 | Boots Linux and runs our program |
| Temperature sensor | TMP102 | Temperature sensor that communicates with Galileo via I2C |
| USB webcam | N/A | Connects to Galileo via USB 2.0; captures images |

# Section 6: Schematics /0.5 points

**Figure 1** below consists of the entire design. The I2C pins on the TMP102, SDA and SCL, are connected to the respective pins on the Galileo, so that the devices can communicate. And the USB webcam is plugged into the Galileo’s USB2.0 slot. Additionally, an Ethernet cable connects the Galileo to the Local Area Network (LAN). This allows files to be moved over the network from the Galileo to a PC and vice versa.

**Figure 1** (made by Derek)

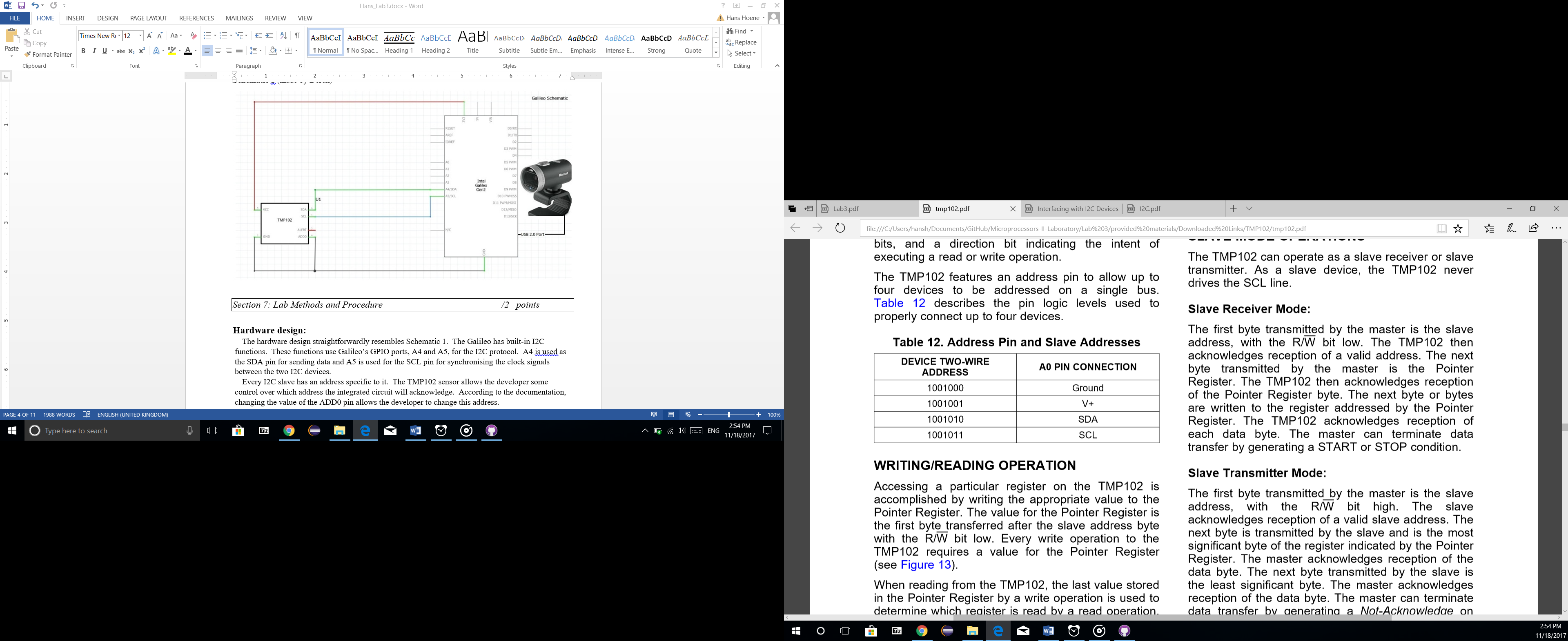


# Section 7: Lab Methods and Procedure /2 points

**Hardware design:**

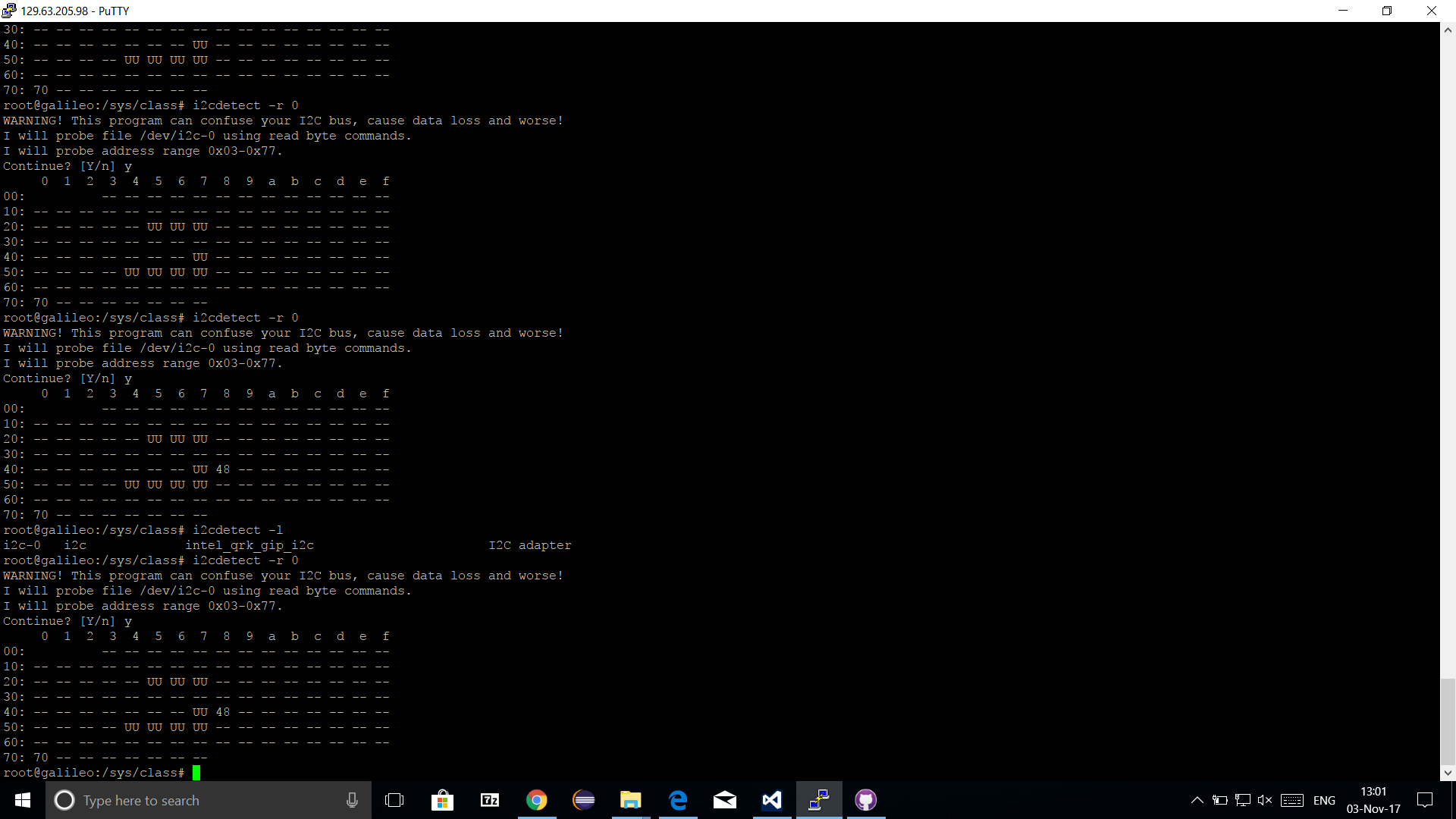
The hardware design straightforwardly resembles **Figure 1**. The Galileo has built-in I2C functions. These functions use Galileo’s GPIO ports, A4 and A5, for the I2C protocol. A4 is used as the SDA pin for sending data and A5 is used for the SCL pin for synchronising the clock signals between the two I2C devices.

Every I2C slave has an address specific to it. The TMP102 sensor allows the developer some control over which address the integrated circuit will acknowledge. According to the documentation, changing the value of the ADD0 pin allows the developer to change this address.

(from TMP102 specification)

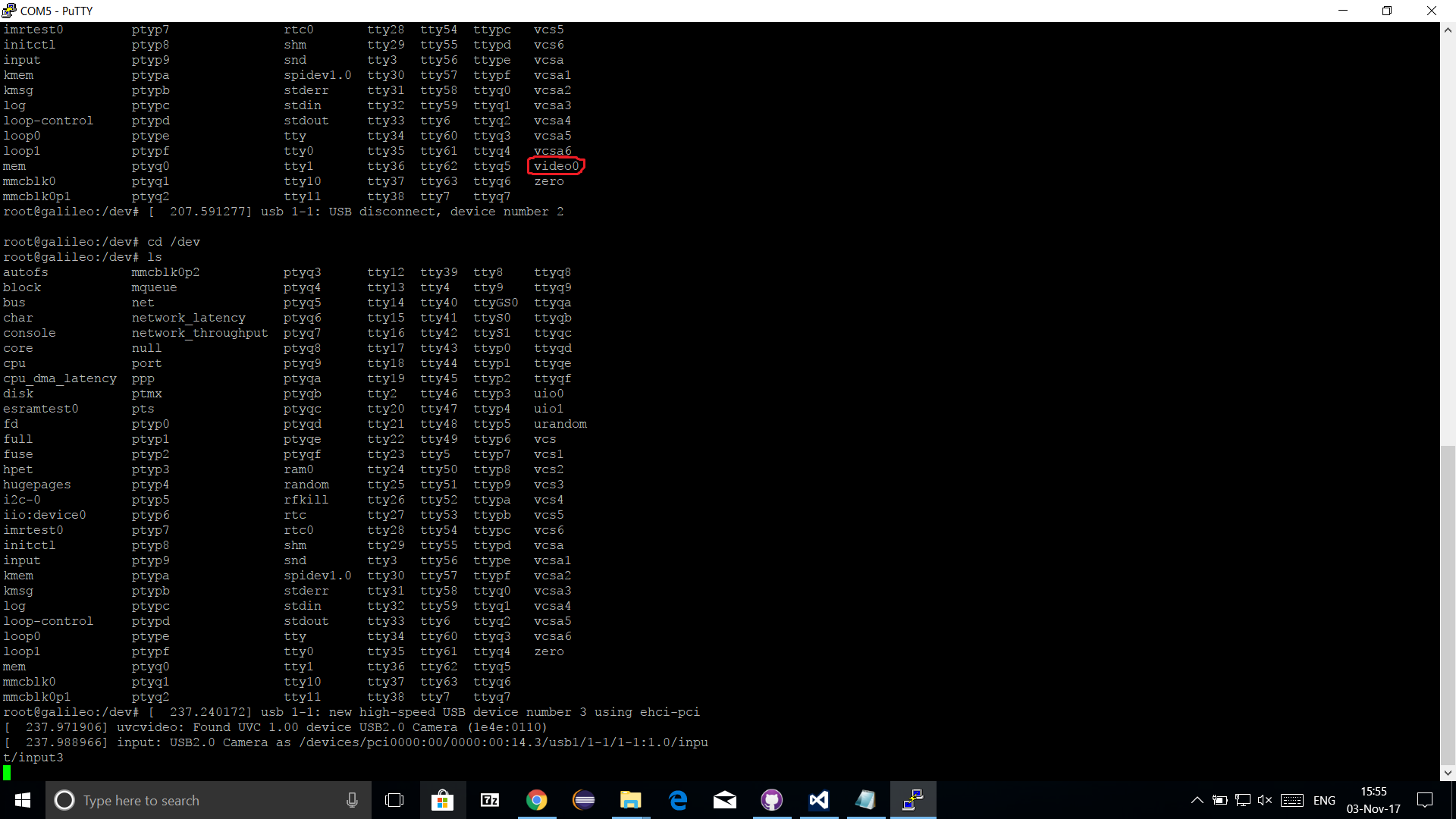
In our circuit, we grounded the ADD0 pin meaning that the TMP102 will acknowledge all I2C calls to address 0x48. We were able to verify this on the Galileo command line. We ran the command “i2cdetect –l” on the Galileo Linux terminal to poll for I2C devices that were connected. We saw one device called “i2c-0”, which implies that there is an I2C device using adapter zero. To evaluate a little further, we ran “i2cdetect –r 0” and saw that there was in fact a device connected at address 0x48. Below in **Figure 2** is a screenshot of these two commands being run.

**Figure 2**



After we were sure that the TMP102 sensor was connected, we went to connect the USB webcam. All we needed to do was connect the webcam’s USB chord into the Galileo USB2.0 port. We verified that the webcam was connected by inspecting the “/dev” folder through the Linux terminal. Linux treats everything as a file, so to find a device, like a USB webcam, navigate to the folder of virtual device files and inspect the contents. **Figure 3** below shows the contents of the “/dev”. The contents are displayed first with the device connected, then again with the device disconnected. Notice how in the first list of contents of “/dev”, there is a directory named, “video0”. This is the USB webcam. Once it is disconnected and the contents of “/dev” are reprinted to the terminal, the directory “vidoe0” is no longer there. This was very helpful for verifying that this directory was in fact the webcam.

**Figure 3**



After this step, we were sure that the USB webcam was connected with a identification number of zero (“video0”), and that the TMP102 device was connected via I2C at address 0x48 with adapter number zero. Let us now discuss the software, which complements the connections we have set up here.

**Software design:**

To manage the temperature sensor, we created files specifically communicating with the temperature sensor. The function declarations are in “i2c.h”, which is shown in **Appendix 1**.

**Appendix 1**

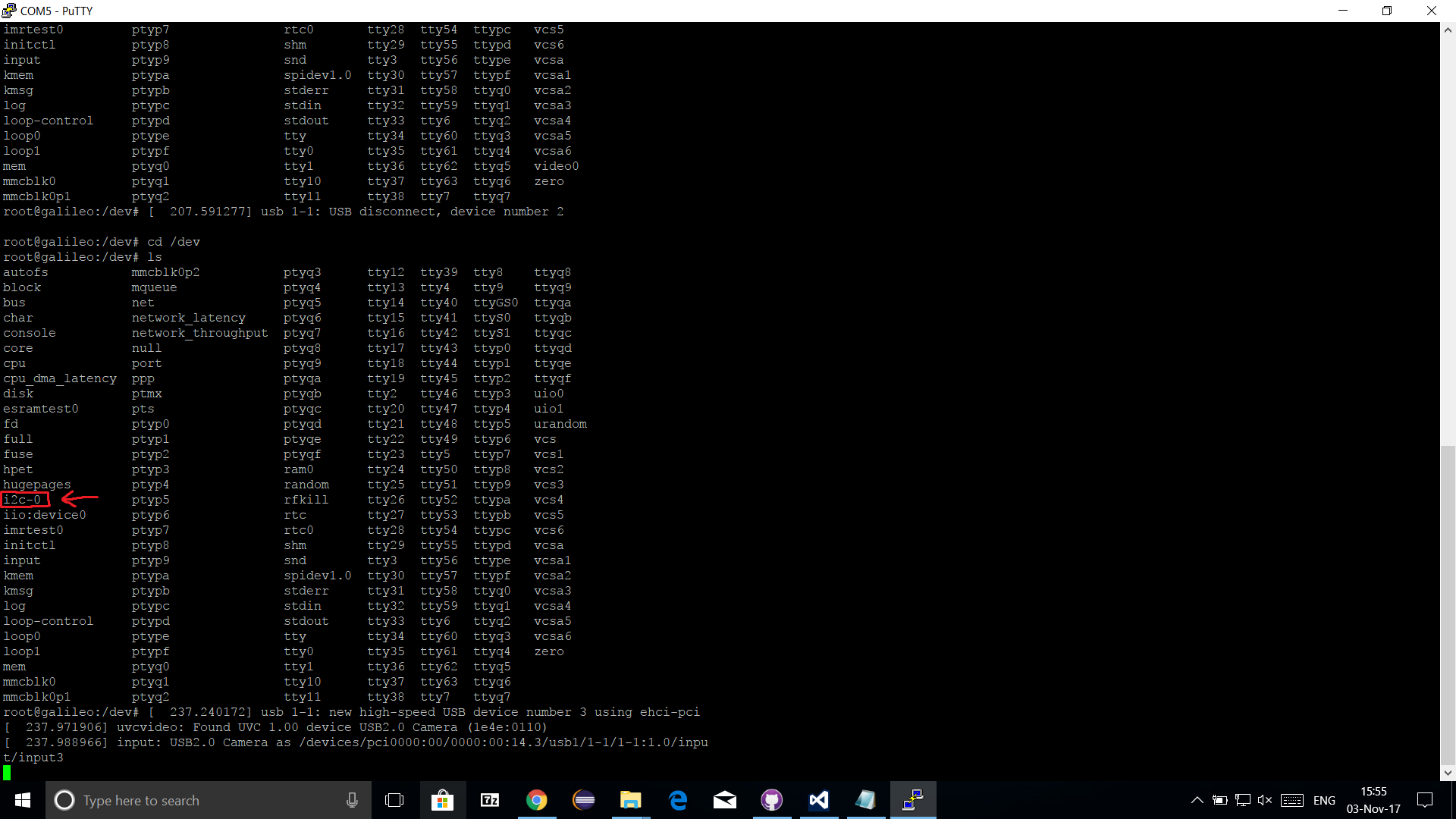


These functions are used to initialise the TMP102 device and subsequently read temperatures from it. “InitTempDevice” will open an I2C connection to the connected TMP102 and it will set the device as a read-only temperature sensor slave. “readTemp” will be used afterwards to get the current temperature in Celsius. “sampleTemp” will not be further discussed because it does nothing more than read the temperature for a set number of times and return the average. The function is used for smoothing out irregular jumps in temperature. We want to avoid sensor sensitivity so we sample the temperature rather than directly read it from the main function. Let us look at the source code below in **Appendix 2**. This code has been taken from “i2c.c”.

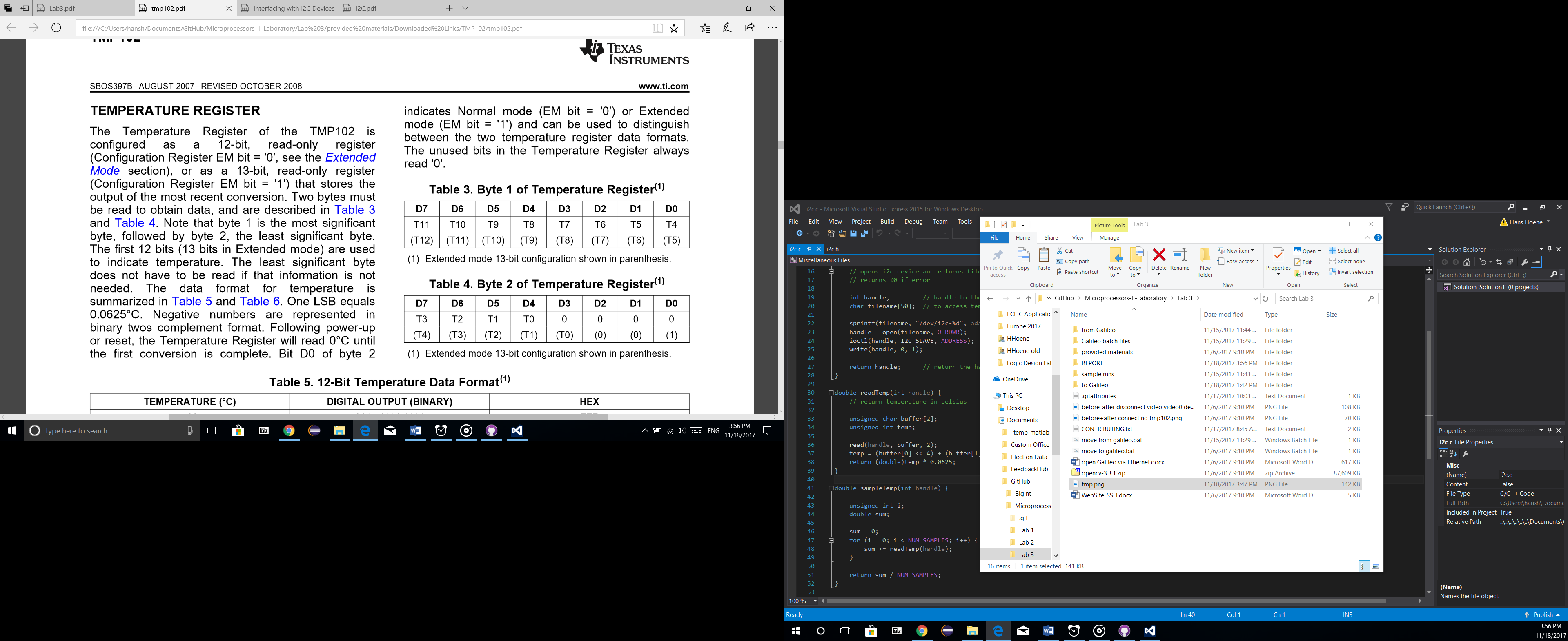


Do you remember how we verified that the temperature sensor was connected? If not, refer to **Figure 2**. We ran commands in the Linux terminal that would detect any I2C devices that were connected. As we discussed earlier, Linux treats everything like a file, so we could have also inspected the “/dev” folder. Let us go back and look at the “/dev” folder in **Figure 4** below. Notice that “i2c-0” is listed there as a device. If you look at the definition for the “InitTempDevice” function above, the function is returning the handle to a file called, “/dev/i2c-0” when the “adapter\_number” argument is zero. The handle is literally a handle to a virtual file. In the lines that follow, the code is specifying that in order to send and read from the file, the I2C protocol to a specific address (0x48) should be used. This code specifies to the hardware that it is time to read and write to file using the Galileo pins A4 and A5, otherwise known SDA and SCL. At the very end, the function sends a zero to the I2C device to tell it that it will be read-only before returning the handle to that virtual file.

**Figure 4**



To read the temperature now, refer back to **Appendix 2**’s “readTemp” function. The handle return from “InitTempDevice” is used to read from the device as if it were a file. The TMP102 stores the temperature as twelve bytes. Refer to the screenshot from the sensor documentation below to see the setup of the two bytes. The first byte hols the most significant eight bits and the second byte holds the least significant four bits as the most significant four bits. This requires the first byte to be shifted to the left by four bits in order to leave room for the least significant four bits. The second byte needs to be shifted to the right by four bits because its four bits are stored in the most significant bits of that byte. After shifting the first byte left four bits and the second bytes right by four bits, the two can be added to get temperature. This integer represents the number of resolution increments in the temperature. The resolution, per the specification, is 0.0625 degrees Celsius. So if the the temperature holds a value of one-thousand, meaning one-thousand increments of resoltuon, the temperatre is 6.25 degrees Celsius, which is the value that wuld be returned by “readTemp” in **Appendix 2**.

 (from TMP102 specification)

Now that we understand how communication was achieved to the TMP102 sensor, let us discuss the USB webcam, which had only one job: capture images. Below in **Appendix 3** is the only function needed in this laboratory, “takePicture”. This code snippet is from “pic.h” and “pic.c”. Since there is only one function, we will not discuss the declaration; rather we will go right ahead and discuss the definition.

**Appendix 3**



These functions almost exclusively use Open CV, which requires that gcc receive a vast number of linking arguments (see **Appendix 3** comments). First, a file name is setup. In this case, all images are being stored to the SD card with an identification number as a name. Next, a frame is captured from the camera. Note that the argument in “cvCaptureFromCAM” can be zero, since there is only one camera connected at “video0”, but the way that we have it works as well because it selects any camera that is connected. Next, data is retrieved from the captured frame. The data is then saved as a JPG at the specified file name. Finally, all of the data is released and the function returns.

The last part of the software design resides in the main function seen in **Appendix 4** below. Using all of the functions that we have described thus far, the main function receives a handle to the temperature sensor, determines what the threshold should be based on whatever temperature someone’s hand is, and finally, the program will poll the temperature until it crosses the threshold. Once it crosses a threshold, a picture will be taken and given incremental names (1.jpg, 2.jpg, …). Once a set number of pictures are taken, the program will exit, so that we are not forced to abruptly abort.

**Appendix 4**



# Section 8: Trouble Shooting /1 points

***Issue 1:***

***Issue 2:***

The next major issue was with the Galileo GPIO pin input setting. Derek and I were able to get the GPIO pins to function properly as outputs, and we were able to get the GPIO pins to open as inputs. However, the inputs were not reading valid values. In test programs, the GPIO pins would read a one or zero once before reading odd numbers such as twenty-nine. We discussed the issue and concluded that it was likely that GPIO pins needed to be consistently re-opened as inputs in order to read multiple values. We made this change to the test program, and afterwards, everything functioned as expected.

***Issue 3:***

The most time-consuming error occurred in the final stages of the project. We were testing both the PIC and the Galileo trying to get them to communicate with each other. The protocol would never finish though. The Galileo user-interface was outputting that a connection error occurred meaning that no valid acknowledgement had been received from the PIC microcontroller. We added very long delays to parts of the protocol in order to analyse portions of the strobe protocol.

During our tests, we observed that the PIC was reading proper values from the Galileo, but the Galileo was not receiving anything back. With some further testing, Kyle noticed with a multimeter that the strobe pin was high at the end of the protocol, which is not right. I went through the Galileo and PIC communication protocol implementations and found our fatal error: on the functions that were supposed to set the strobe pin low again, a variable “Strobe” was being used rather than a different variable called “strobe” (lowercase). This means that in the beginning of a protocol, the strobe pin would go high and the PIC would read the data.

Afterwards though, the strobe would remain high and the PIC would never add the instruction to the execution queue nor generate an acknowledgement code response. Meanwhile, the Galileo would continue communicating as if there was no problem and it would read data off of an empty data bus from the PIC, which caused the error message. Once this coding error was changed, the program worked immediately!

# Section 9: Results /0.5 points

Figure 5

Screenshot of terminal-interfacing program on Galileo

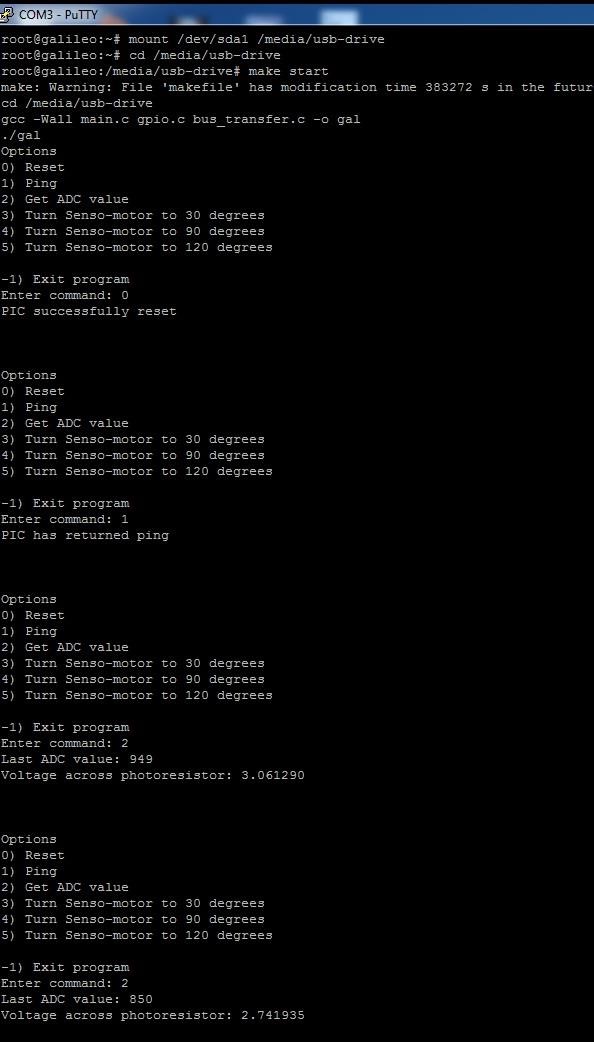


Figure 6

Screenshot of PWM signal used to turn senso-motor to thirty degrees

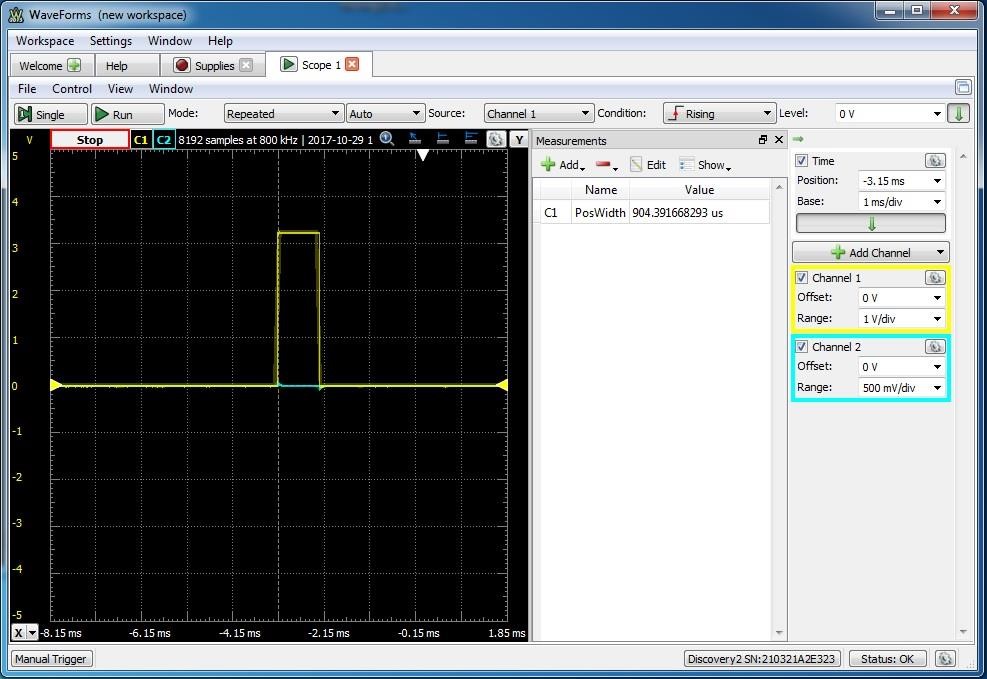


Figure 7

Screenshot of PWM signal used to turn senso-motor to ninety degrees

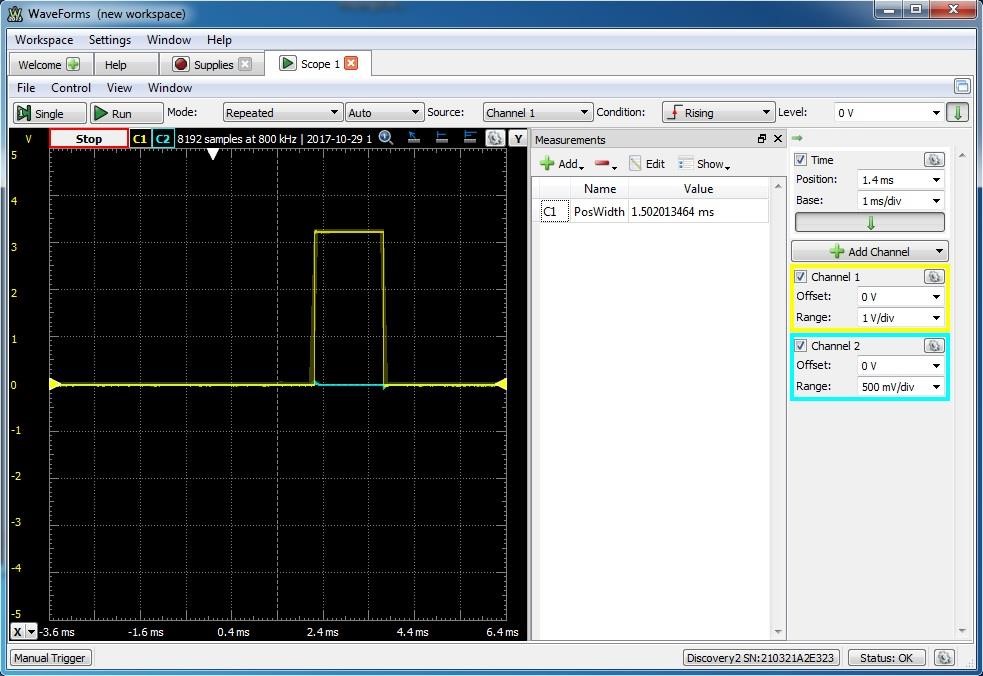
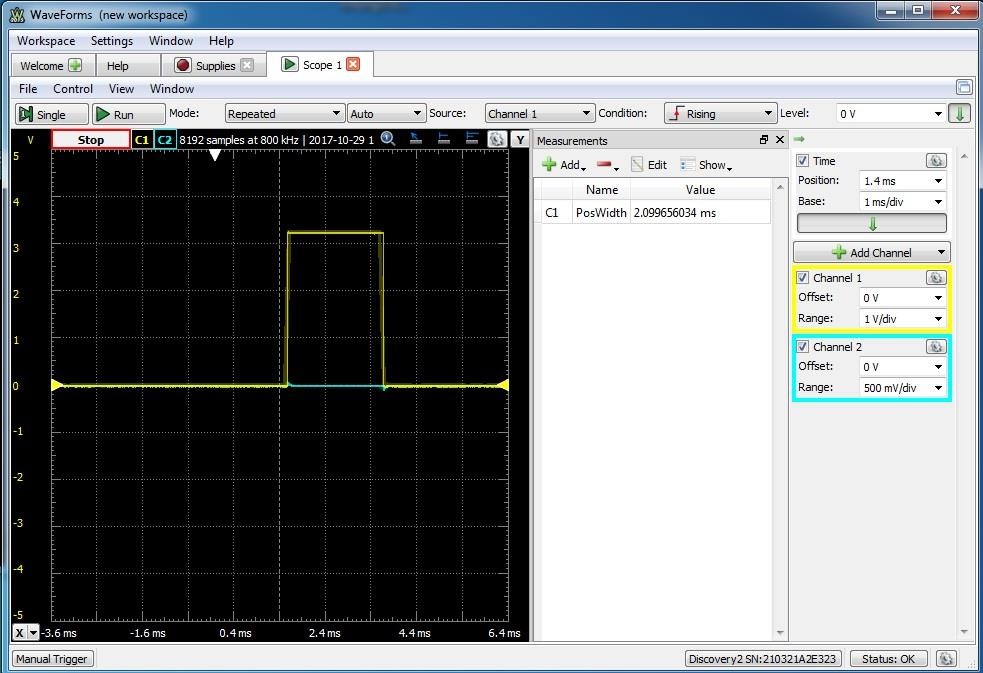


Figure 8

Screenshot of PWM signal used to turn senso-motor to one hundred twenty degrees



# Section 10: Appendix

A1.

/\*this switch statement is called every interrupt.

communication\_counter represents that phase of the communication.\*/ switch (communication\_counter) { case 0:

// first interrupt, read value from GPIO bus // GPIO bus pins should already be set as inputs

instruction = read(); ++communication\_counter; break; case 1:

// computer is done outputting signal // start processing and set up outputs already if (instruction == MSG\_GET) { write((adc\_value >> 8) & 0x3);

} else { enqueue(&execution\_queue, instruction); write(MSG\_ACK);

}

++communication\_counter; break; case 2:

// computer raises signal

// reading has begun

TRISB &= 0xE1; // set up outputs

++communication\_counter;

break; case 3:

// computer done reading value if (instruction == MSG\_GET) {

// write bit again

write((adc\_value >> 4) & 0xF);

++communication\_counter;

} else {

TRISB |= 0x1E; // back to inputs (high impedance)

communication\_counter = 0; // next edge will be new command

}

break; case 4:

// only get will come this far

// reading has begun ++communication\_counter; break; case 5:

// computer done reading // write one more value! write(adc\_value & 0xF); ++communication\_counter; case 6:

// reading has begun ++communication\_counter; break; case 7:

// reading done write(MSG\_ACK); ++communication\_counter; break; case 8:

++communication\_counter;

// reading

break; case 9:

// all done w/ everything TRISB |= 0x1E;

communication\_counter = 0; // next edge is new instruction break; default:

// handle error

communication\_counter = 0; // next edge is new instruction break;

}

A2.

/\*

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\*/

/\*START STEP 3 -- just switch strobe to low to indicate that communication is over, and close pins\*/

writeGPIO(strobe, LOW); closeGPIO(GP\_4, data[0]); closeGPIO(GP\_5, data[1]); closeGPIO(GP\_6, data[2]); closeGPIO(GP\_7, data[3]); /\*END STEP 3\*/

A3.

while (1) {

/\*START LIGHT SENSOR AND LED PART\*/ if (led\_counter < 0) { // adc conversion is running

// only if conversion is done,

// update LED based on LED\_THRESHOLD (pre-defined)

if (ADCON0bits.GO == LOW) {

// conversion = done

// update led

adc\_value = (ADRESH << 8) + ADRESL; if (adc\_value > max) { max = adc\_value;

threshold = (max + min) >> 1;

}

if (adc\_value < min) { min = adc\_value;

threshold = (max + min) >> 1;

}

// debounce

if (LATAbits.LATA0 == HIGH) {

/\*LED is off; to turn on, adc\_value must go below the lower offset\*/ if (adc\_value < threshold - OFFSET) {

LATAbits.LATA0 = LOW;

}

} else {

if (adc\_value > threshold + OFFSET) {

LATAbits.LATA0 = HIGH;

}

}

led\_counter = 0; // start counter over again

}

} else if (led\_counter >= LED\_ROLLOVER) {

// it is time to start an adc conversion ADCON0bits.GO = HIGH; // start conversion

led\_counter = -1; // indicates that conversion is running

} else {

// neither so just update counter

++led\_counter;

}

/\*END LIGHT SENSOR AND LED PART\*/ /\*Queue execution\*/ if (!isEmpty(&execution\_queue)) { switch (dequeue(&execution\_queue)) { case MSG\_RESET:

min = 1023;

max = 0;

threshold = 0; break; case MSG\_TURN30: PWM\_Turn30(); break; case MSG\_TURN90: PWM\_Turn90();

break; case MSG\_TURN120: PWM\_Turn120();

break;

}

}

} // end infinite loop