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

2. 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

3. 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.

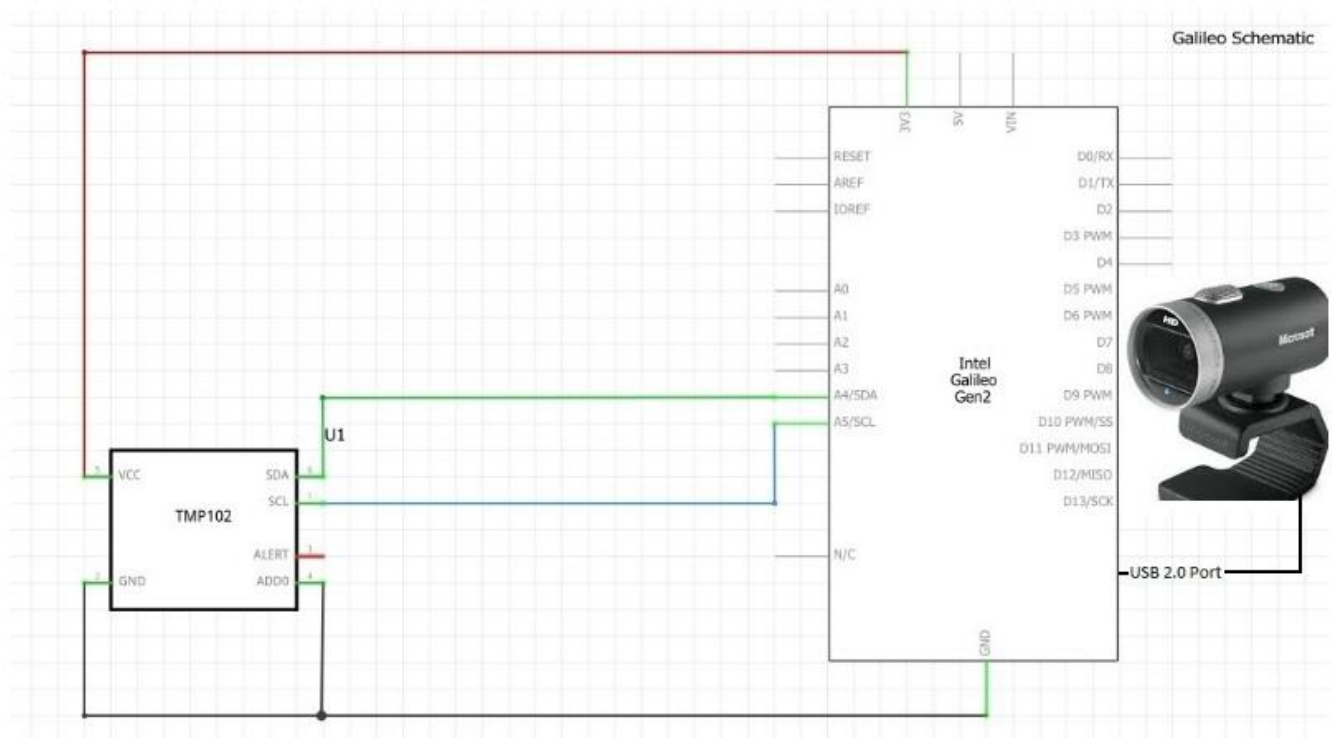
The webcam is connected via USB. The webcam is controlled and interfaced via the open source Open CV library. The Open CV library is used to capture and store images from the webcam. Before using this library though, it had to be installed from the internet.

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

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.

Table 12. Address Pin and Slave Addresses

DEVICE TWO-WIRE ADDRESS	A0 PIN CONNECTION
1001000	Ground
1001001	V+
1001010	SDA
1001011	SCL

(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

```
root@galileo:/sys/class# i2cdetect -l
i2c-0   i2c             intel_qrk_gip_i2c                       I2C adapter
root@galileo:/sys/class# i2cdetect -r 0
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-0 using read byte commands.
I will probe address range 0x03-0x77.
Continue? [Y/n] y
     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:/sys/class#
```

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

COM5 - PuTTY

```
imrtest0        ptyp7             rtc0       tty28  tty54  ttypc   vcs5
initctl         ptyp8             shm        tty29  tty55  ttypd   vcs6
input           ptyp9             snd        tty3   tty56  ttype   vcsa
kmem            ptypa             spidev1.0  tty30  tty57  ttypf   vcsa1
kmsg            ptypb             stderr     tty31  tty58  ttyq0   vcsa2
log             ptypc             stdin      tty32  tty59  ttyq1   vcsa3
loop-control    ptypd             stdout     tty33  tty6   ttyq2   vcsa4
loop0           ptype             tty        tty34  tty60  ttyq3   vcsa5
loop1           ptypf             tty0       tty35  tty61  ttyq4   vcsa6
mem             ptyq0             tty1       tty36  tty62  ttyq5   video0
mmcblk0         ptyq1             tty10      tty37  tty63  ttyq6   zero
mmcblk0p1       ptyq2             tty11      tty38  tty7   ttyq7
root@galileo:/dev# [  207.591277] usb 1-1: USB disconnect, device number 2

root@galileo:/dev# cd /dev
root@galileo:/dev# ls
autofs          mmcblk0p2         ptyq3      tty12  tty39  tty8    ttyq8
block           mqueue            ptyq4      tty13  tty4   tty9    ttyq9
bus             net               ptyq5      tty14  tty40  ttyGS0  ttyqa
char            network_latency   ptyq6      tty15  tty41  ttyS0   ttyqb
console         network_throughput ptyq7     tty16  tty42  ttyS1   ttyqc
core            null              ptyq8      tty17  tty43  ttyp0   ttyqd
cpu             port              ptyq9      tty18  tty44  ttyp1   ttyqe
cpu_dma_latency ppp               ptyqa      tty19  tty45  ttyp2   ttyqf
disk            ptmx              ptyqb      tty2   tty46  ttyp3   uio0
esramtest0      pts               ptyqc      tty20  tty47  ttyp4   uio1
fd              ptyp0             ptyqd      tty21  tty48  ttyp5   urandom
full            ptyp1             ptyqe      tty22  tty49  ttyp6   vcs
fuse            ptyp2             ptyqf      tty23  tty5   ttyp7   vcs1
hpet            ptyp3             ram0       tty24  tty50  ttyp8   vcs2
hugepages       ptyp4             random     tty25  tty51  ttyp9   vcs3
i2c-0           ptyp5             rfkill     tty26  tty52  ttypa   vcs4
iio:device0     ptyp6             rtc        tty27  tty53  ttypb   vcs5
imrtest0        ptyp7             rtc0       tty28  tty54  ttypc   vcs6
initctl         ptyp8             shm        tty29  tty55  ttypd   vcsa
input           ptyp9             snd        tty3   tty56  ttype   vcsa1
kmem            ptypa             spidev1.0  tty30  tty57  ttypf   vcsa2
kmsg            ptypb             stderr     tty31  tty58  ttyq0   vcsa3
log             ptypc             stdin      tty32  tty59  ttyq1   vcsa4
loop-control    ptypd             stdout     tty33  tty6   ttyq2   vcsa5
loop0           ptype             tty        tty34  tty60  ttyq3   vcsa6
loop1           ptypf             tty0       tty35  tty61  ttyq4   zero
mem             ptyq0             tty1       tty36  tty62  ttyq5
mmcblk0         ptyq1             tty10      tty37  tty63  ttyq6
mmcblk0p1       ptyq2             tty11      tty38  tty7   ttyq7
```

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

```
/* CODE FROM "i2c.h" */
#ifndef I2C_H_
#define I2C_H_

#define ADDRESS 0x48                 // because ADD0 on TMP102 is grounded
                                     // validated with "i2cdetect -r 0"
#define ADAPTER_NUMBER 0             // determined with "i2cdetect -l"
#define NUM_SAMPLES 10

int InitTempDevice(int adapter_number);   // returns handle to TMP102
double readTemp(int handle);              // reads temperature using I2C handle
double sampleTemp(int handle);       // sample temperature to get rid of irregularities

#endif
```

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".

Appendix 2

```c
/* CODE FROM "i2c.c" */
#include <stdio.h>
#include "i2c.h"

#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>          // ioctl function
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int InitTempDevice(int adapter_number) {
	int handle;         // handle to the temperature sensor; will be returned
	char filename[50];  // to access temperature sensor

	sprintf(filename, "/dev/i2c-%d", adapter_number);
	handle = open(filename, O_RDWR);          // gets handle to temperature sensor
	ioctl(handle, I2C_SLAVE, ADDRESS);       // io control;set as I2C slave at ADDRESS
	write(handle, 0, 1);                      // set as read-only temperature sensor

	return handle;
}

double readTemp(int handle) {
	// return temperature in celsius

	unsigned char buffer[2];
	unsigned int temp;

	read(handle, buffer, 2);                         // read data
	temp = (buffer[0] << 4) + (buffer[1] >> 4);      // get as int
	return (double)temp * 0.0625;                    // multiply by resolution
}

double sampleTemp(int handle) {
	unsigned int i;
	double sum;

	sum = 0;
	for (i = 0; i < NUM_SAMPLES; i++) {
		sum += readTemp(handle);
	}
	return sum / NUM_SAMPLES;
}
```

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

```
root@galileo:/dev# cd /dev
root@galileo:/dev# ls
autofs            mmcblk0p2            ptyq3      tty12  tty39  tty8    ttyq8
block             mqueue               ptyq4      tty13  tty4   tty9    ttyq9
bus               net                  ptyq5      tty14  tty40  ttyGS0  ttyqa
char              network_latency      ptyq6      tty15  tty41  ttyS0   ttyqb
console           network_throughput   ptyq7      tty16  tty42  ttyS1   ttyqc
core              null                 ptyq8      tty17  tty43  ttyp0   ttyqd
cpu               port                 ptyq9      tty18  tty44  ttyp1   ttyqe
cpu_dma_latency   ppp                  ptyqa      tty19  tty45  ttyp2   ttyqf
disk              ptmx                 ptyqb      tty2   tty46  ttyp3   uio0
esramtest0        pts                  ptyqc      tty20  tty47  ttyp4   uio1
fd                ptyp0                ptyqd      tty21  tty48  ttyp5   urandom
full              ptyp1                ptyqe      tty22  tty49  ttyp6   vcs
fuse              ptyp2                ptyqf      tty23  tty5   ttyp7   vcs1
hpet              ptyp3                ram0       tty24  tty50  ttyp8   vcs2
hugepages         ptyp4                random     tty25  tty51  ttyp9   vcs3
i2c-0             ptyp5                rfkill     tty26  tty52  ttypa   vcs4
iio:device0       ptyp6                rtc        tty27  tty53  ttypb   vcs5
imrtest0          ptyp7                rtc0       tty28  tty54  ttypc   vcs6
initctl           ptyp8                shm        tty29  tty55  ttypd   vcsa
input             ptyp9                snd        tty3   tty56  ttype   vcsa1
kmem              ptypa                spidev1.0  tty30  tty57  ttypf   vcsa2
kmsg              ptypb                stderr     tty31  tty58  ttyq0   vcsa3
log               ptypc                stdin      tty32  tty59  ttyq1   vcsa4
loop-control      ptypd                stdout     tty33  tty6   ttyq2   vcsa5
loop0             ptype                tty        tty34  tty60  ttyq3   vcsa6
loop1             ptypf                tty0       tty35  tty61  ttyq4   zero
mem               ptyq0                tty1       tty36  tty62  ttyq5
mmcblk0           ptyq1                tty10      tty37  tty63  ttyq6
mmcblk0p1         ptyq2                tty11      tty38  tty7   ttyq7
```

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 would be returned by "readTemp" in **Appendix 2**.

Table 3. Byte 1 of Temperature Register(1)

D7	D6	D5	D4	D3	D2	D1	D0
T11 (T12)	T10 (T11)	T9 (T10)	T8 (T9)	T7 (T8)	T6 (T7)	T5 (T6)	T4 (T5)

(1) Extended mode 13-bit configuration shown in parenthesis.

Table 4. Byte 2 of Temperature Register(1)

D7	D6	D5	D4	D3	D2	D1	D0
T3 (T4)	T2 (T3)	T1 (T2)	T0 (T1)	0 (T0)	0 (0)	0 (0)	0 (1)

(1) Extended mode 13-bit configuration shown in parenthesis.

(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

```
/* CODE FROM "pic.h" and "pic.c" */
/*
must be compiled with the following gcc args in order to access the Open CV library:

-I/usr/local/include/opencv -I/usr/local/include/opencv2 -L/usr/local/lib/ -lm -lo-
pencv_core -lopencv_imgproc -lopencv_highgui -lopencv_ml -lopencv_video -lopencv_fea-
tures2d -lopencv_calib3d -lopencv_objdetect -lopencv_contrib -lopencv_legacy -lo-
pencv_stitching
*/
// Open CV Header Files
#include <opencv2/objdetect/objdetect.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv/cv.h>
#include <opencv/highgui.h>

#include <stdio.h>

#define DEST_FOLDER "/media/card/to PC"   // pictures end up here
#define PICTURE_LIMIT 5             // maximum # of pictures that will be taken

void takePicture(unsigned int id) {

	char filename[200];
	CvCapture *capture;
	IplImage *image;

	sprintf(filename, "%s/%u.jpg", DEST_FOLDER, id);
	capture = cvCaptureFromCAM(CV_CAP_ANY);
	image = cvQueryFrame(capture);
	cvSaveImage(filename, image, 0);                // save image to file as JPG
	cvReleaseCapture(&capture);                     // release capture
	cvReleaseImage(&image);                                // release image

	return;
}
```

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.

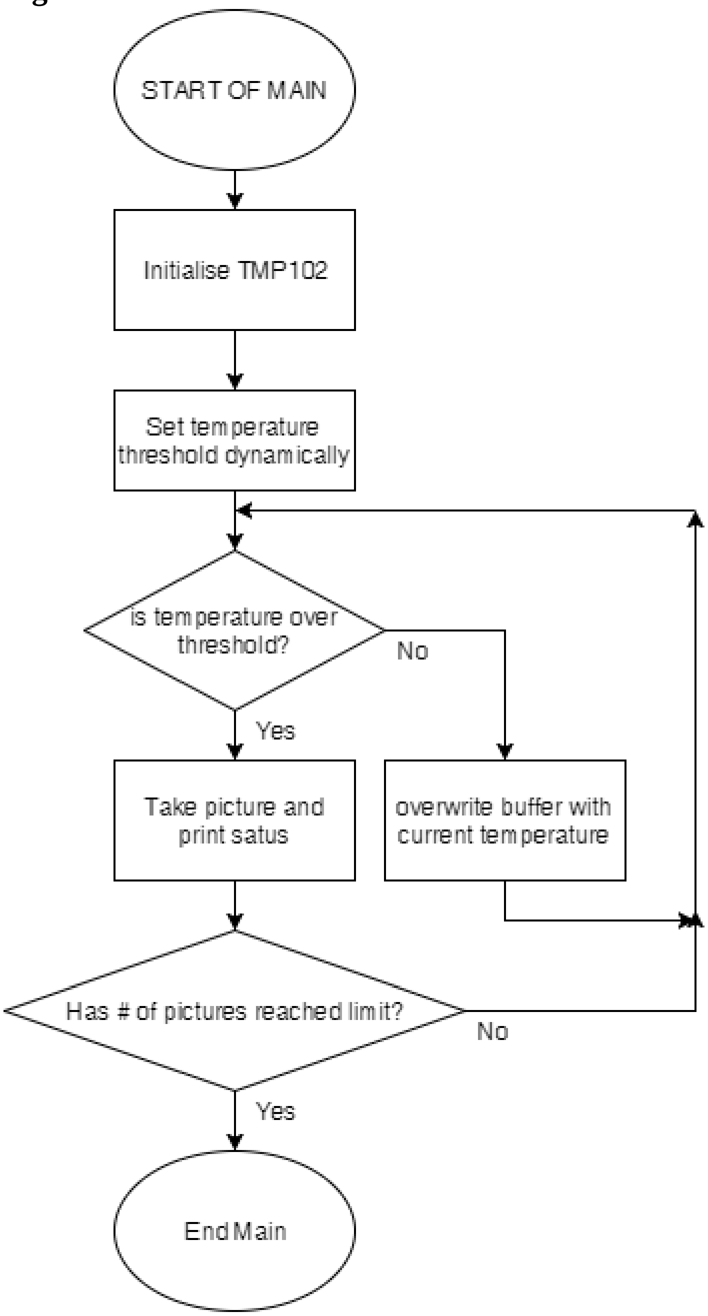
The last part of the software design resides in the main function seen in **Appendix 4** below. Using all of the functions that we have described thus far, the main function receives a handle to the temperature sensor, determines what the threshold should be based on whatever temperature someone’s hand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

Appendix 4

```c
#include <linux/i2c-dev.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "i2c.h"
#include "pic.h"

int main() {

	unsigned int pic_counter;		// # of pictures taken
	int temp_sensor_handle;
	double temp, temp_threshold;

	temp_sensor_handle = InitTempDevice(ADAPTER_NUMBER);// get handle to temp sensor
	pic_counter = 0;

	/* protocol to determine temperature threshold dynamically */
	puts("Get ready to put hand on the sensor...");
	sleep(5);
	puts("Put hand on temperature sensor. Do not remove until instructed to do so.");
	sleep(5);
	temp_threshold = sampleTemp(temp_sensor_handle);
	puts("Now take your hand off the sensor.");

	printf("Threshold: %2.2lf degrees Celsius\nProgram will begin in 5 seconds...\n\n", temp_threshold);
	sleep(5);

	// infinite loop - exit from inside
	while (1) {
		temp = sampleTemp(temp_sensor_handle);
		if (temp > temp_threshold) {
			// temperature is above threshold, take picture
			++pic_counter;
			printf("\rYour picture is being taken. Temperature (C) = %2.2lf\n_", temp);
			takePicture(pic_counter);

			if (pic_counter >= PICTURE_LIMIT) {
				// if enough pictures have been taken, exit
				return 0;
			}

		} else {
			// if temperature is not above threshold, overwrite line with current temperature
			printf("\r%2.2lf", temp);
		}
	}

	return 0;		// the code shall never reach this point

} // end main
```

In order to better understand the flow of this program through its many files, consult the flowchart below in **Figure 5**.

Figure 5



Issue 1: **USB webcam malfunctioning**

We were able to connect the TMP102 sensor without any issues. Fairly quickly, we were also able to connect a USB webcam and take pictures. However, we noticed that about 60% to 70% of the captured images were distorted in their colour. Consult **Figure 6** below. This is a typical picture taken of the ceiling with a light. Although this picture is normal, it does not represent the majority of the pictures that were being taken. Some pictures, like **Figure 6**, appeared normal while most appeared inundated in an odd green pigment. We originally thought that some night mode feature was being triggered to switch on and off without and precedence. We actually tested this by pointing the webcam towards light. What we discovered immediately invalidated our hypothesis. **Figure 7** shows light as a purple shade. This is one of many odd pictures taken. It seems that the entire colour scheme would become distorted at random. That night, Derek tried running our program with his ow personal USB webcam attached instead. The pictures were consistently superb, so we assumed we had a hardware issue with the webcam that we were given. We contacted our TA, Ioannis Smith, who confirmed our suspicions and gave us a new USB webcam. After using a different webcam, we were able to take non-distorted pictures and finish the program logic.

Figure 6

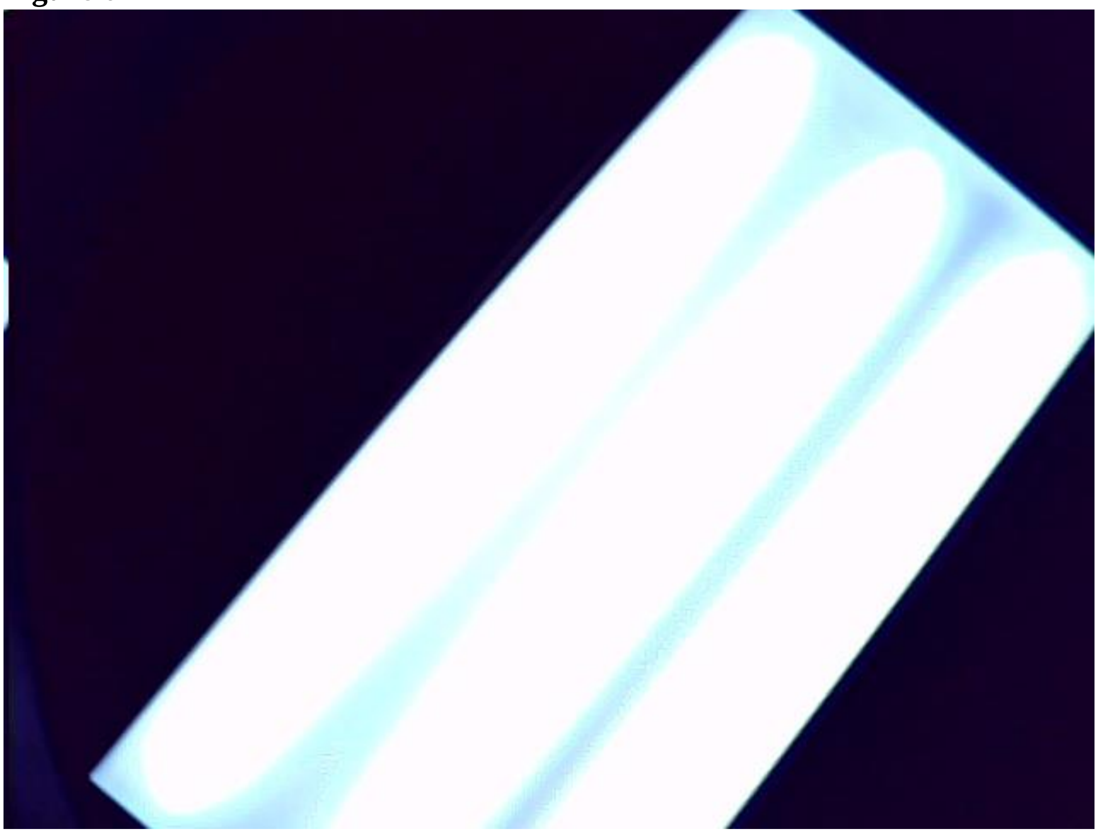
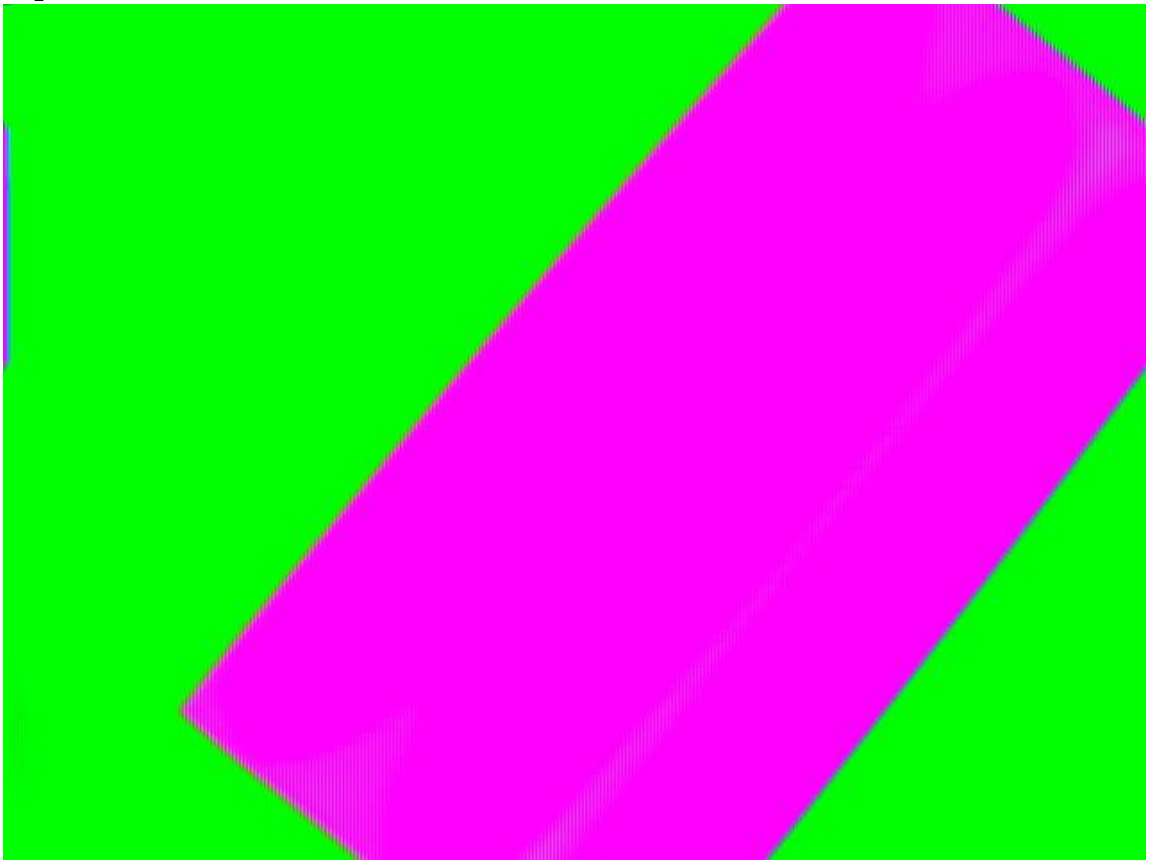


Figure 7



Issue 2: **Incorrect Temperature Reading**

In an attempt to make the code more logically sound, I made several changes to the code and a bug arose. The temperature was no longer reading correctly. We collectively reviewed all my changes, and fortunately, Kyle pointed out the error. In "i2c.c", I made the following change inside "readTemp".

```
temp = (buffer[0] << 4) + (buffer[1] >> 4);      // original line

temp = (buffer[0] << 4) | (buffer[1] >> 4);      // new line (OR instead of ADD)
```

Even after all this time, none of can say why the second line does not work. In fact, it is possible that this line was not even the problem. It is possible that we accidentally changed something else back in the process. All that we know is that the last time we tried to change that line, it didn't work, so we plan on just leaving it as it is.

Issue 3: **Incorrect Temperature Reading Again**

Another thing we noticed is that it seemed that communication with the temperature sensor would sometimes fail and a negative number would be read often screwing up test runs. We were unable to resolve the issue. In retrospect, I think we most likely had a poor hardware connection. It is possible that the problem no longer even exists. But this problem has been made irrelevant nevertheless. To fix the problem on the spot, we added a function, "sampleTemp" in "i2c.h" and "i2c.c". We used this function to read the temperature. Rather than directly reading the temperature from the sensor, we decided to sample the temperature by making several measurements and taking the average. Therefore, if a bad value was read, it was voided because it was averaged out.

Section 9: Results *_/0.5 points_*

This section of the laboratory will quickly look at the output of the program.

In the beginning of the program, the threshold is set dynamically. The program asks the user to place their hand on the temperature sensor for five seconds. The program measures the temperature that the sensor achieves before asking their user to remove their hand and allow the sensor to cool. From this point on, this temperature must be obtained to take a picture. Whenever a picture is taken, a message along with the temperature is printed. Otherwise, the current temperature is printed on the line and that is constantly overwritten in the buffer until next picture is taken. Consult **Figure 8** to see the output in the middle of the program. Notice how the bottom line is being rewritten with the current temperature. **Figure 9** is the output of the program after it has finished. He program finishes once it captures the maximum amount of images allowed by the program logic. In this case, that limiting number was five.

Figure 8

```
root@galileo:~/Documents/from PC# ./gal.out
Get ready to put hand on the sensor...
Put hand on temperature sensor. Do not remove until instructed to do so.
Now take your hand off the sensor.
Threshold: 25.81 degrees Celsius
Program will begin in 5 seconds...

Your picture is being taken. Temperature (C) = 25.82
Your picture is being taken. Temperature (C) = 25.83
Your picture is being taken. Temperature (C) = 25.84
25.81
```

Figure 9

```
root@galileo:~/Documents/from PC# ./gal.out
Get ready to put hand on the sensor...
Put hand on temperature sensor. Do not remove until instructed to do so.
Now take your hand off the sensor.
Threshold: 25.81 degrees Celsius
Program will begin in 5 seconds...

Your picture is being taken. Temperature (C) = 25.82
Your picture is being taken. Temperature (C) = 25.83
Your picture is being taken. Temperature (C) = 25.84
Your picture is being taken. Temperature (C) = 25.83
Your picture is being taken. Temperature (C) = 25.88
_root@galileo:~/Documents/from PC#
```

Section 10: Appendix

Throughout the laboratory report, code snippets have been put inline for easy reference. Collaboratively, those snippets can consist of the entire program. Here they are again all in one location.

Appendix 1 – i2c.h

```
/* CODE FROM “i2c.h” */
#ifndef I2C_H_
#define I2C_H_

#define ADDRESS 0x48              // because ADD0 on TMP102 is grounded
                                  // validated with “i2cdetect –r 0”
#define ADAPTER_NUMBER 0          // determined with “i2cdetect –l”
#define NUM_SAMPLES 10

int InitTempDevice(int adapter_number);   // returns handle to TMP102
double readTemp(int handle);              // reads temperature using I2C handle
double sampleTemp(int handle);    // sample temperature to get rid of irregularities

#endif
```

Appendix 2 – i2c.c

```
/* CODE FROM "i2c.c" */
#include <stdio.h>
#include "i2c.h"

#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>          // ioctl function
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int InitTempDevice(int adapter_number) {
       int handle;         // handle to the temperature sensor; will be returned
       char filename[50];  // to access temperature sensor

       sprintf(filename, "/dev/i2c-%d", adapter_number);
       handle = open(filename, O_RDWR);          // gets handle to temperature sensor
       ioctl(handle, I2C_SLAVE, ADDRESS);       // io control;set as I2C slave at ADDRESS
       write(handle, 0, 1);                      // set as read-only temperature sensor

       return handle;
}

double readTemp(int handle) {
       // return temperature in celsius

       unsigned char buffer[2];
       unsigned int temp;

       read(handle, buffer, 2);                          // read data
       temp = (buffer[0] << 4) + (buffer[1] >> 4);      // get as int
       return (double)temp * 0.0625;                     // multiply by resolution
}

double sampleTemp(int handle) {
       unsigned int i;
       double sum;

       sum = 0;
       for (i = 0; i < NUM_SAMPLES; i++) {
              sum += readTemp(handle);
       }
       return sum / NUM_SAMPLES;
}
```

Appendix 3 – pic.h and pic.c

```
/* CODE FROM “pic.h” and “pic.c” */
/*
must be compiled with the following gcc args in order to access the Open CV library:

-I/usr/local/include/opencv -I/usr/local/include/opencv2 -L/usr/local/lib/ -lm -lo-
pencv_core -lopencv_imgproc -lopencv_highgui -lopencv_ml -lopencv_video -lopencv_fea-
tures2d -lopencv_calib3d -lopencv_objdetect -lopencv_contrib -lopencv_legacy -lo-
pencv_stitching
*/
// Open CV Header Files
#include <opencv2/objdetect/objdetect.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv/cv.h>
#include <opencv/highgui.h>

#include <stdio.h>

#define DEST_FOLDER "/media/card/to PC"   // pictures end up here
#define PICTURE_LIMIT 5            // maximum # of pictures that will be taken

void takePicture(unsigned int id) {

	char filename[200];
	CvCapture *capture;
	IplImage *image;

	sprintf(filename, "%s/%u.jpg", DEST_FOLDER, id);
	capture = cvCaptureFromCAM(CV_CAP_ANY);
	image = cvQueryFrame(capture);
	cvSaveImage(filename, image, 0);			// save image to file as JPG
	cvReleaseCapture(&capture);			// release capture
	cvReleaseImage(&image);					// release image

	return;
}
```

Appendix 4 – main.c

```c
#include <linux/i2c-dev.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "i2c.h"
#include "pic.h"

int main() {

	unsigned int pic_counter;		// # of pictures taken
	int temp_sensor_handle;
	double temp, temp_threshold;

	temp_sensor_handle = InitTempDevice(ADAPTER_NUMBER);// get handle to temp sensor
	pic_counter = 0;

	/* protocol to determine temperature threshold dynamically */
	puts("Get ready to put hand on the sensor...");
	sleep(5);
	puts("Put hand on temperature sensor. Do not remove until instructed to do so.");
	sleep(5);
	temp_threshold = sampleTemp(temp_sensor_handle);
	puts("Now take your hand off the sensor.");

	printf("Threshold: %2.2lf degrees Celsius\nProgram will begin in 5 seconds...\n\n", temp_threshold);
	sleep(5);

	// infinite loop - exit from inside
	while (1) {
		temp = sampleTemp(temp_sensor_handle);
		if (temp > temp_threshold) {
			// temperature is above threshold, take picture
			++pic_counter;
			printf("\rYour picture is being taken. Temperature (C) = %2.2lf\n_", temp);
			takePicture(pic_counter);

			if (pic_counter >= PICTURE_LIMIT) {
				// if enough pictures have been taken, exit
				return 0;
			}

		} else {
			// if temperature is not above threshold, overwrite line with current temperature
			printf("\r%2.2lf", temp);
		}
	}

	return 0;		// the code shall never reach this point

} // end main
```