EE3580 Lab Report

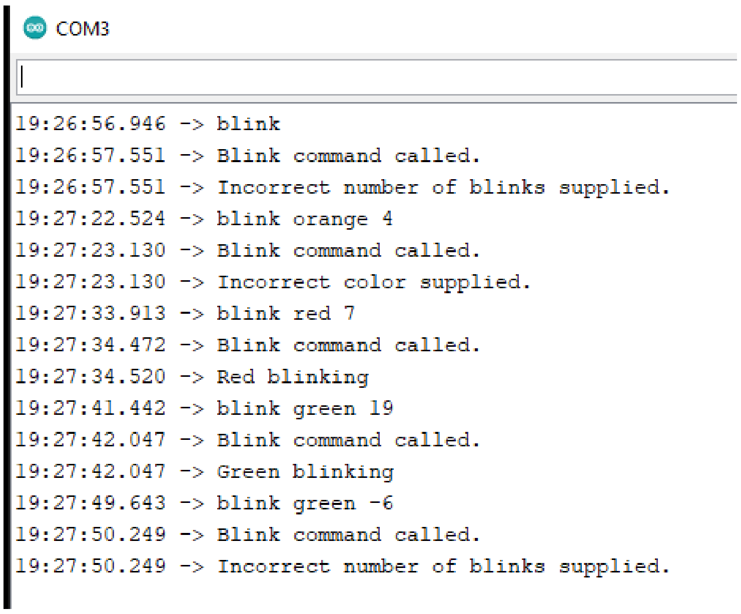
Piotr Rucinski

Table of Contents

1. Serial Terminal with Arduino

1.1 Overview of the task

The purpose of the task is to create a system to communicate with Arduino Uno. Exercises 1 & 2 will not be explored, as it is simply testing of the connections, which would be hard to demonstrate to work without a video. However, some COM commands can be seen for Exercise 3 that might be useful.

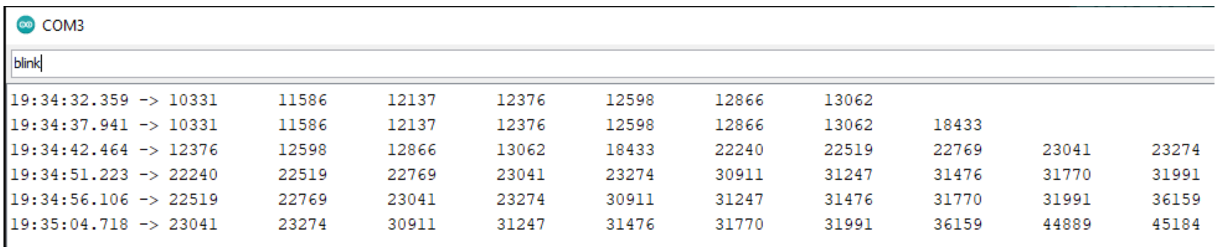


```
19:26:56.946 -> blink
19:26:57.551 -> Blink command called.
19:26:57.551 -> Incorrect number of blinks supplied.
19:27:22.524 -> blink orange 4
19:27:23.130 -> Blink command called.
19:27:23.130 -> Incorrect color supplied.
19:27:33.913 -> blink red 7
19:27:34.472 -> Blink command called.
19:27:34.520 -> Red blinking
19:27:41.442 -> blink green 19
19:27:42.047 -> Blink command called.
19:27:42.047 -> Green blinking
19:27:49.643 -> blink green -6
19:27:50.249 -> Blink command called.
19:27:50.249 -> Incorrect number of blinks supplied.
```

Figure 1.1. Console output of Exercise 3.

Exercise 4 expands on the previous exercises too. The "blink" command remains the same and code for it is unchanged (with the helper debugging commands mostly removed). A circular array was implemented to store the button presses times, which is programmatically traversed upon looping. Using the long int variable comfortably grants us 24855 days of having the program run without overflowing. Button presses are counted within an interrupt service routine – a method of gathering data that will prove very useful in the following exercises. Please do see the console outputs below.

The code for exercise 3 & 4 is attached in the appendix.



```
19:34:32.359 -> 10331    11586    12137    12376    12598    12866    13062
19:34:37.941 -> 10331    11586    12137    12376    12598    12866    13062    18433
19:34:42.464 -> 12376    12598    12866    13062    18433    22240    22519    22769    23041    23274
19:34:51.223 -> 22240    22519    22769    23041    23274    30911    31247    31476    31770    31991
19:34:56.106 -> 22519    22769    23041    23274    30911    31247    31476    31770    31991    36159
19:35:04.718 -> 23041    23274    30911    31247    31476    31770    31991    36159    44889    45184
```

Figure 1.2. Console output of calling "blink" repeatedly, after pressing the button a couple of times.

2. Direct Digital Synthesis

2.1 Overview of the tasks ahead

The purpose of this exercise was to use Arduino as a signal generator, and then test it both visually on the Oscilloscope (or the Serial Plotter for Task 4, as I didn't manage to do it on time), as well as have the signal be played by a buzzer.

Code for task 1 has been provided in the appendix. As per the example, duty cycle is at 75% at 440 Hz.

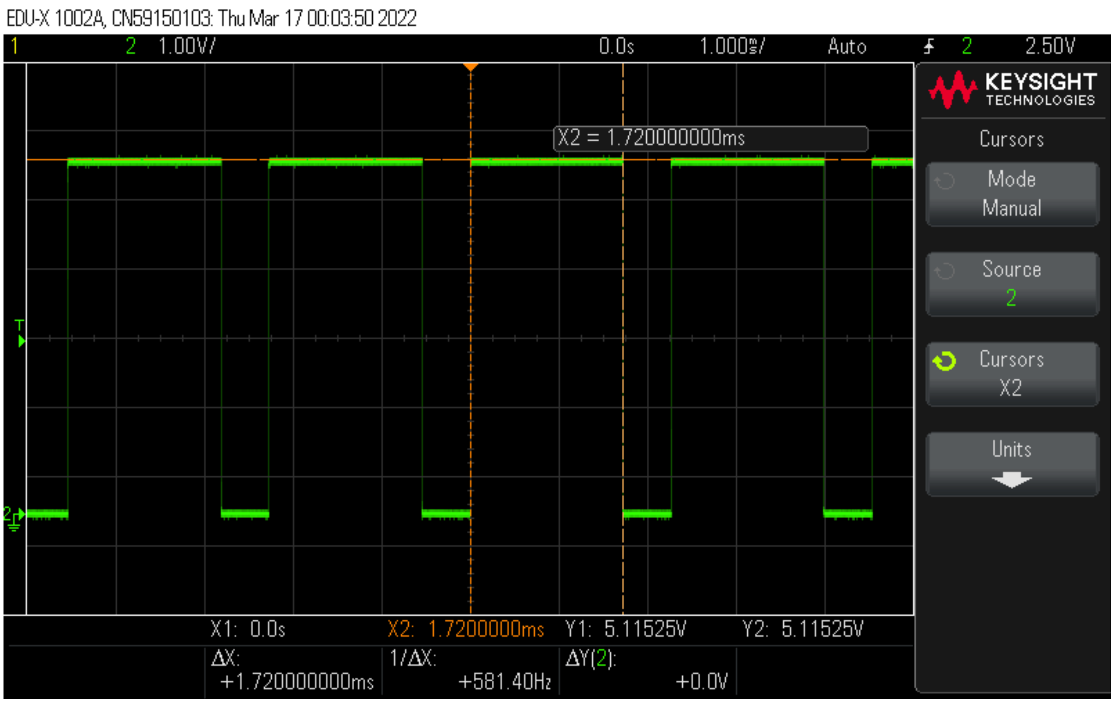
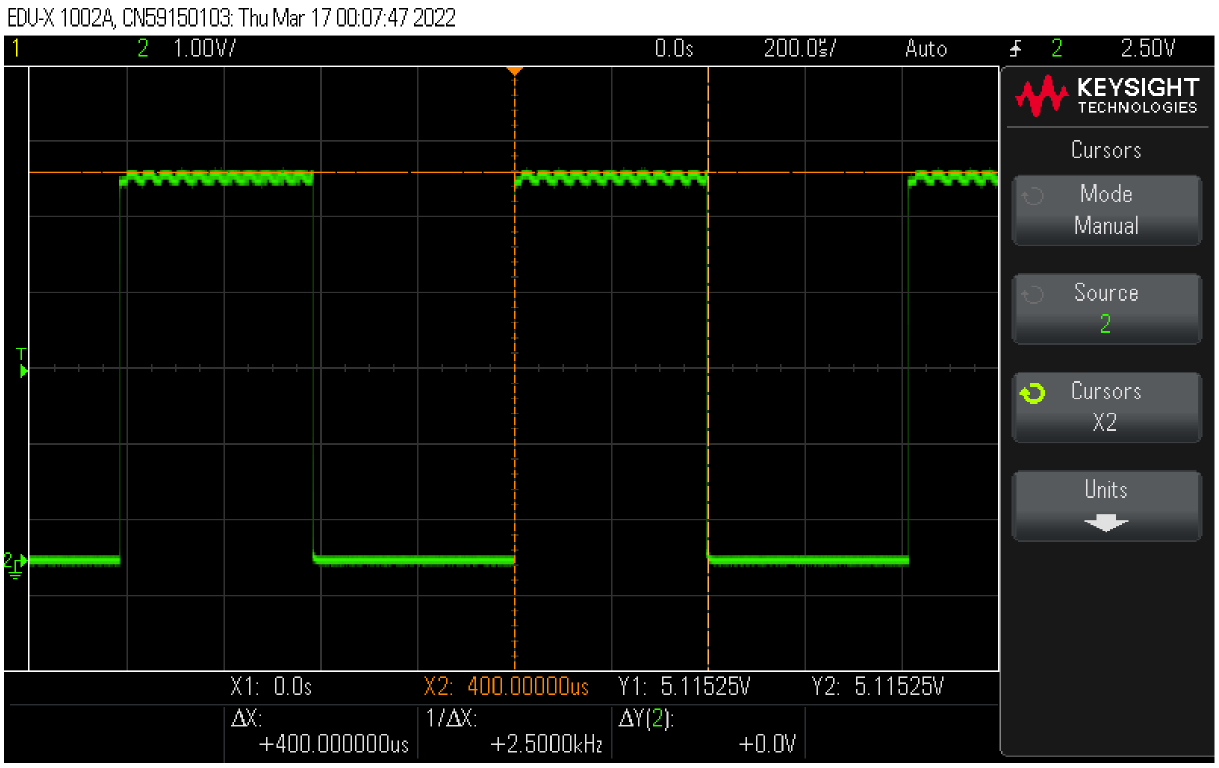


Figure 2.1. Square wave at 440 Hz with a duty cycle of 0.75. As per Figure 5 in lab instructions, OC0B is at HIGH for ~1.7ms – the hardware has been implemented properly.

2.2 Square wave generator

Function setup_timer() has been modified so as to take in a frequency desired by the user. In order to do so, I had to establish maximum and minimum frequency that can be taken in for the system. Using the equation provided in the lab sheet, I have determined these to be 244 and 31250Hz. The upper bound could theoretically be double that, however for robustness of the code I am not allowing any of the registers be at 0. First, inside the function, the value for OCR0A needs to be determined from the equation F = 16000000/(N*(OCR0A + 1) – then, OCR0B has to be found via p = (OCR0B + 1)/(OCR0A + 1). Initially, excessive bracketing on my part has led to me receiving bizarre results (such as zero) – I have learned that when operating on ints, bracketing them as (duty_cycle/256) is a terrible idea as it will evaluate to 0.

I have mapped the potentiometer to a linear function. This is probably not ideal as humans will find any noises above 1500Hz unpleasant, but the entire spectrum of the signal has been covered this way. Modifying OCR0A strictly influences the frequency at which the waveform is generated. Please see appendix for code.



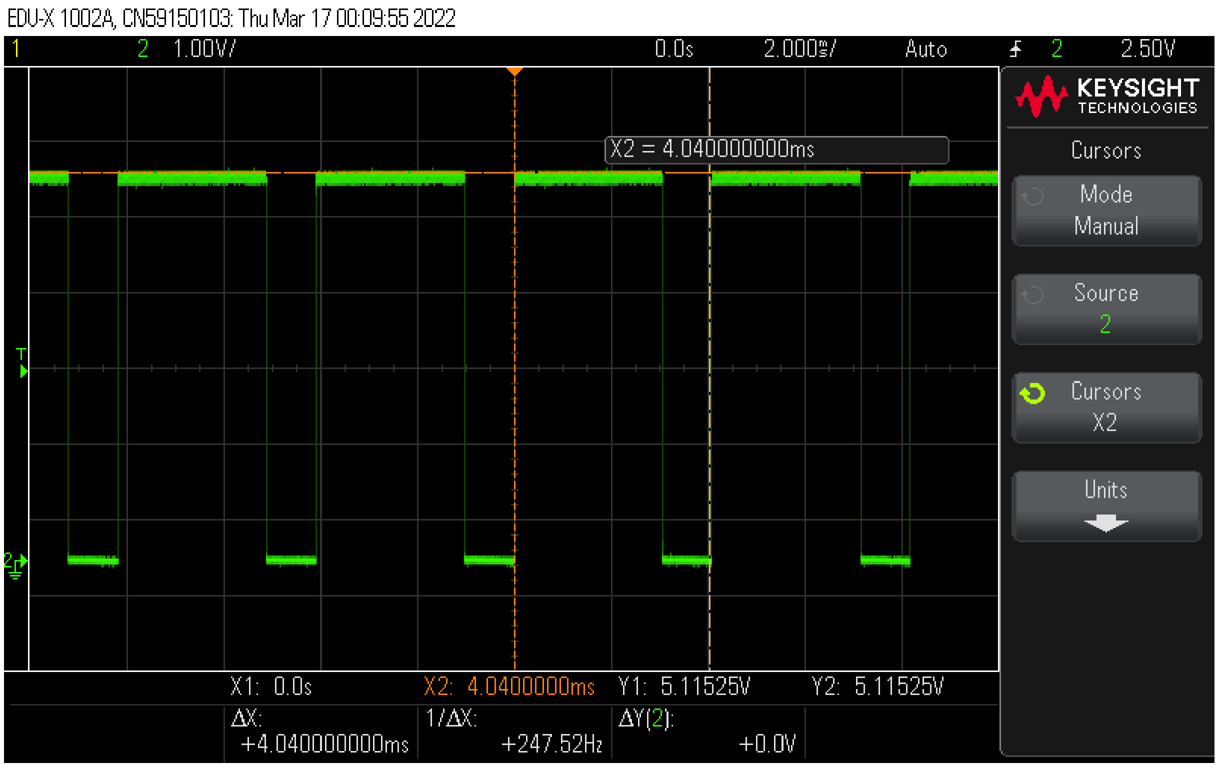
EDU-X 1002A, CN59150103: Thu Mar 17 00:07:47 2022

1 2 1.00V/ 0.0s 200.0us/ Auto 2 2.50V

X1: 0.0s X2: 400.00000us Y1: 5.11525V Y2: 5.11525V

ΔX: +400.000000us 1/ΔX: +2.5000kHz ΔY(2): +0.0V

Figure 2.2. Square wave at supplied values F = 1204 Hz, duty cycle 50%. As evidenced by the reading, F = 1/800ns = 1250Hz. I have tried going fairly close to the intended 1kHz frequency. There is still a small discrepancy between the reading and the actual value – this could be caused by an imprecise reading and/or integer division (and our registers also have limited flexibility in terms of values).



EDU-X 1002A, CN59150103: Thu Mar 17 00:09:55 2022

1 2 1.00V/ 0.0s 2.000ms/ Auto 2 2.50V

X2 = 4.040000000ms

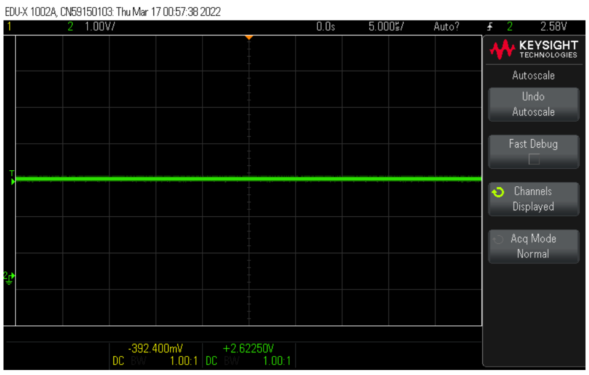
X1: 0.0s X2: 4.0400000ms Y1: 5.11525V Y2: 5.11525V

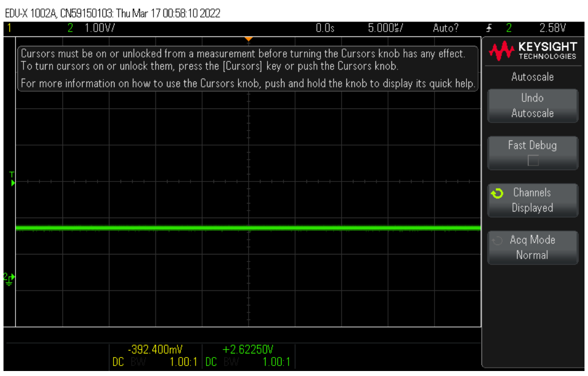
ΔX: +4.040000000ms 1/ΔX: +247.52Hz ΔY(2): +0.0V

Figure 2.3 I have not taken the readings of frequency 1kHz at 25% duty cycle, but I have taken a measurement of frequency 244Hz at a duty cycle of 75%. F = 1/4.04ms = 247 Hz. These frequencies are much more pleasant to listen to.

2.3 Low pass filter

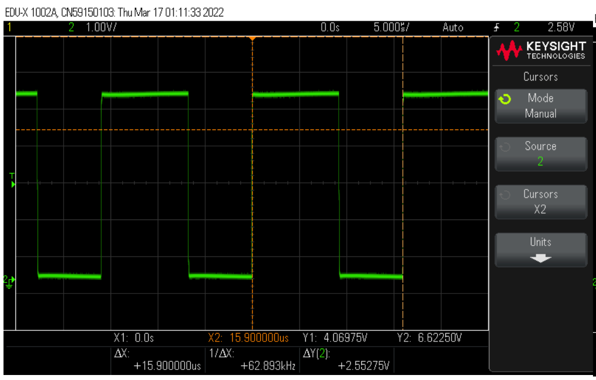
For this exercise, a low-pass filter has been built to extract the first harmonics of a square wave. First harmonics will be a constant – therefore we will end up with a flat wave – something that is easier to spot on the serial plotter rather than the oscilloscope. As it can be seen from the readings, modifying the value of duty cycle with the potentiometer influences the voltage on the pin (power of the signal). Some sample values are shown below. Please see code for appendix.

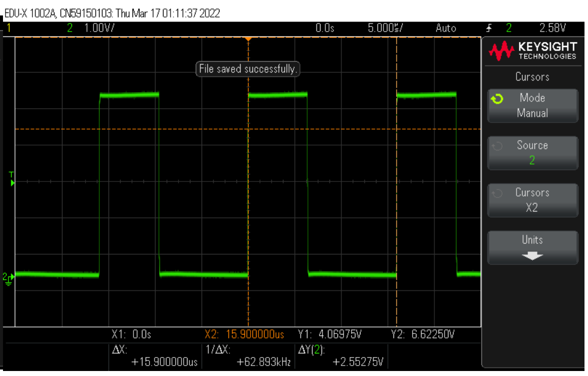




Cursors must be on or unlocked from a measurement before turning the Cursors knob has any effect.
To turn cursors on or unlock them, press the [Cursors] key or push the Cursors knob.
For more information on how to use the Cursors knob, push and hold the knob to display its quick help.

Figures 2.4 & 2.5. Outputs of the low-pass filter. See how they differ with regards to GND.





Figures 2.6 & 2.7. Variable duty cycles, changed via the potentiometer.

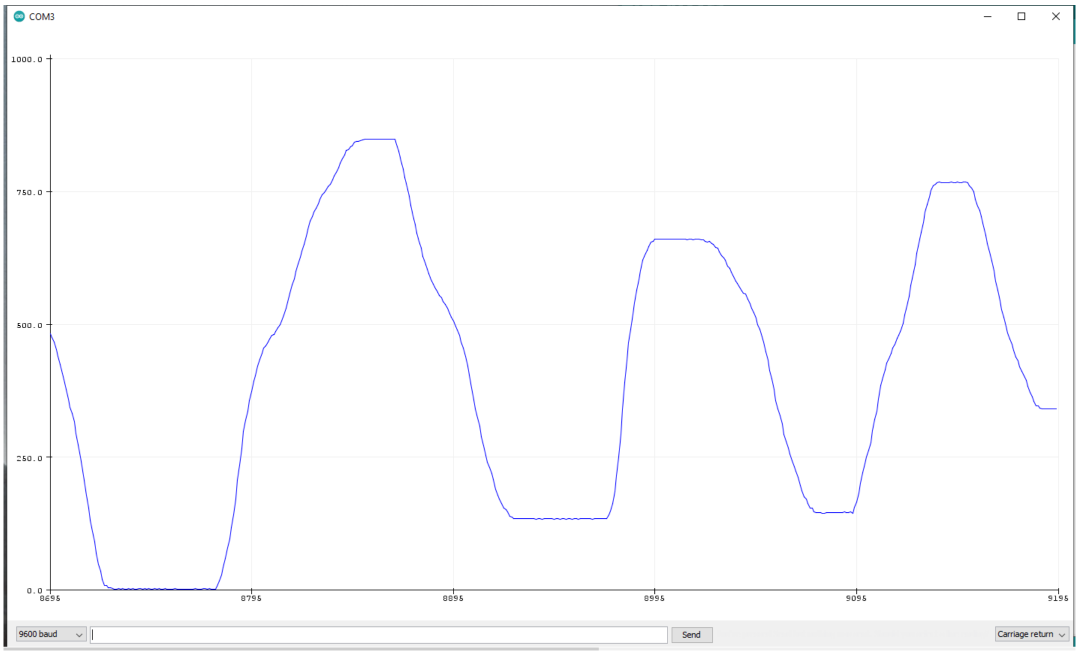
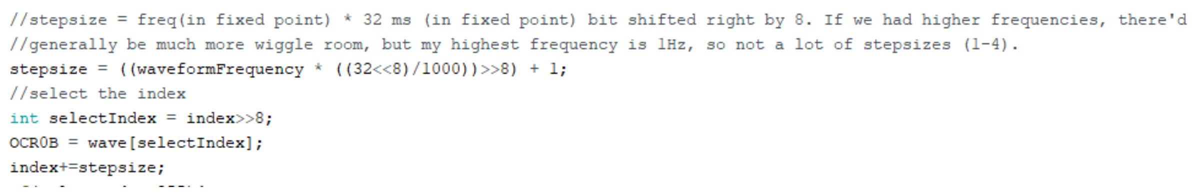


Figure 2.8. Manually varying the duty cycle.

2.4 Direct Digital Synthesis

This task was the hardest one out of the entire lab, as it required quite a lot of complex understanding of the underlying algorithms. In the end, I have successfully managed to implement the various waveforms.

Underlying the new waveforms is a square wave of 62kHz – however, it could be any frequency, as we are only extracting the first harmonics of this wave (which is not frequency-dependent). I have followed the algorithm that is presented in the lab sheet. The logic I followed for this is shown below.



```
//stepsize = freq(in fixed point) * 32 ms (in fixed point) bit shifted right by 8. If we had higher frequencies, there'd
//generally be much more wiggle room, but my highest frequency is 1Hz, so not a lot of stepsizes (1-4).
stepsize = ((waveformFrequency * ((32<<8)/1000))>>8) + 1;
//select the index
int selectIndex = index>>8;
OCR0B = wave[selectIndex];
index+=stepsize;
```

Figure 2.9. Algorythm logic.

Surprisingly enough, in reality the frequency is slightly lower than 1Hz (and for consecutive frequencies). I have decided for the maximum frequency to be 1024Hz for the ease of conversion from the potentiometer, as evidenced inside the code.

After this, figuring out the rest of the task was fairly easy. I have implemented some additional functions for the sawtooth and sinusoidal functions in the library. Their outputs can be seen below.

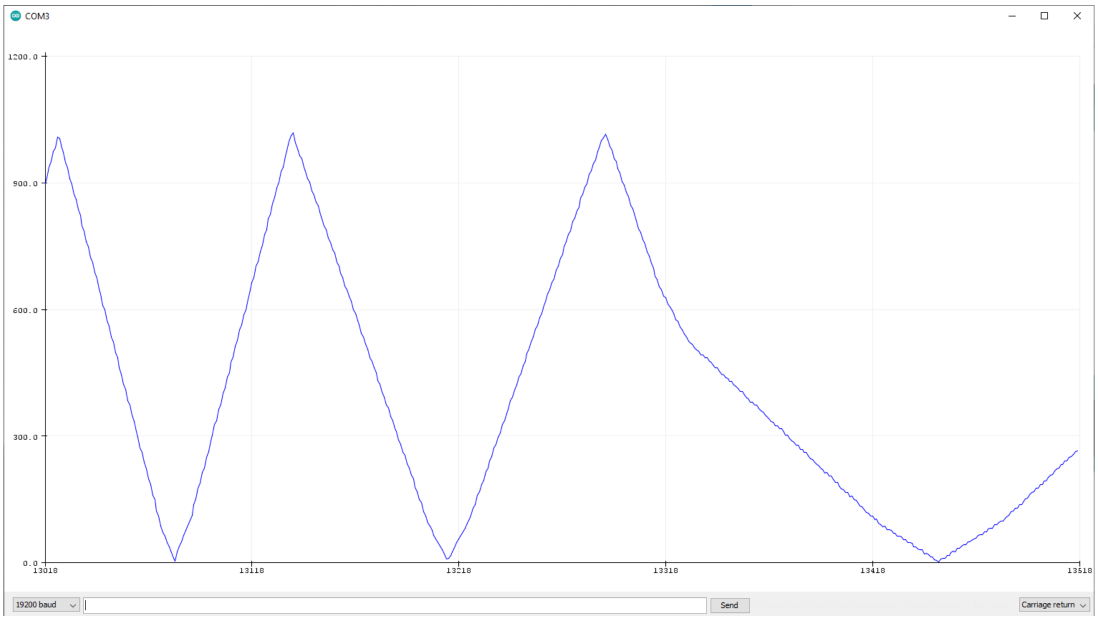


Figure 2.10. Triangular wave.

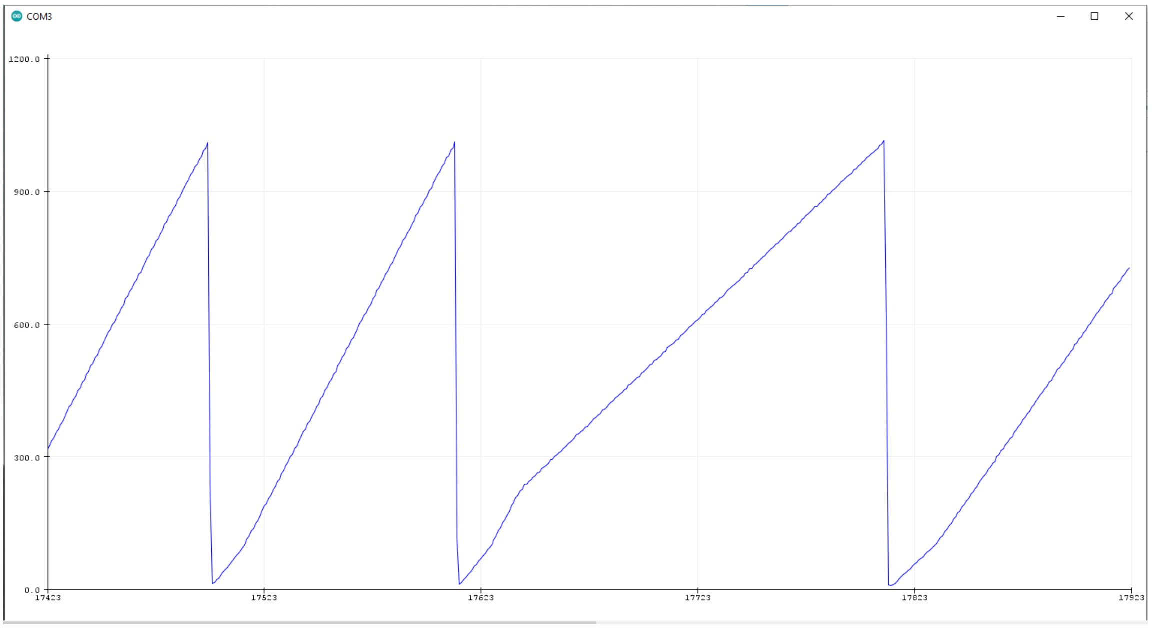


Figure 2.11. Sawtooth wave.

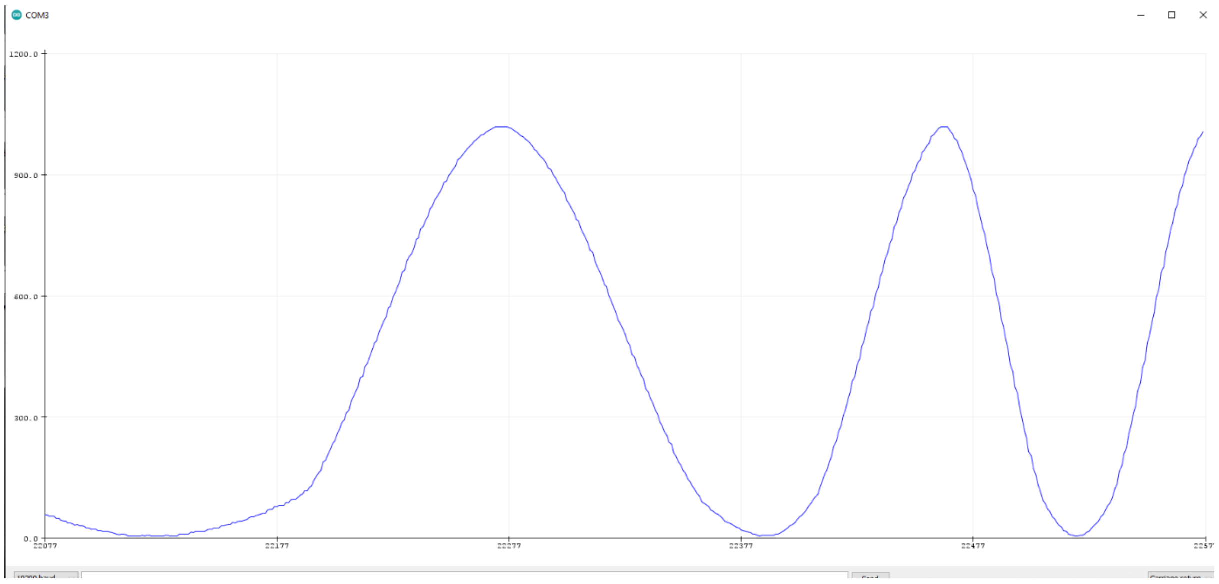


Figure 2.12. Sinusoidal wave.

3. Heat Detector Alarm

3.1 Overview of the task & the system

The purpose of this task was to implement a PCB that can be used to implement an alarm system. I have chosen the Arduino Shield variant – the main reason influencing my design decision was my familiarity with using a shield. In order to produce the schematics below, DesignSpark was used.

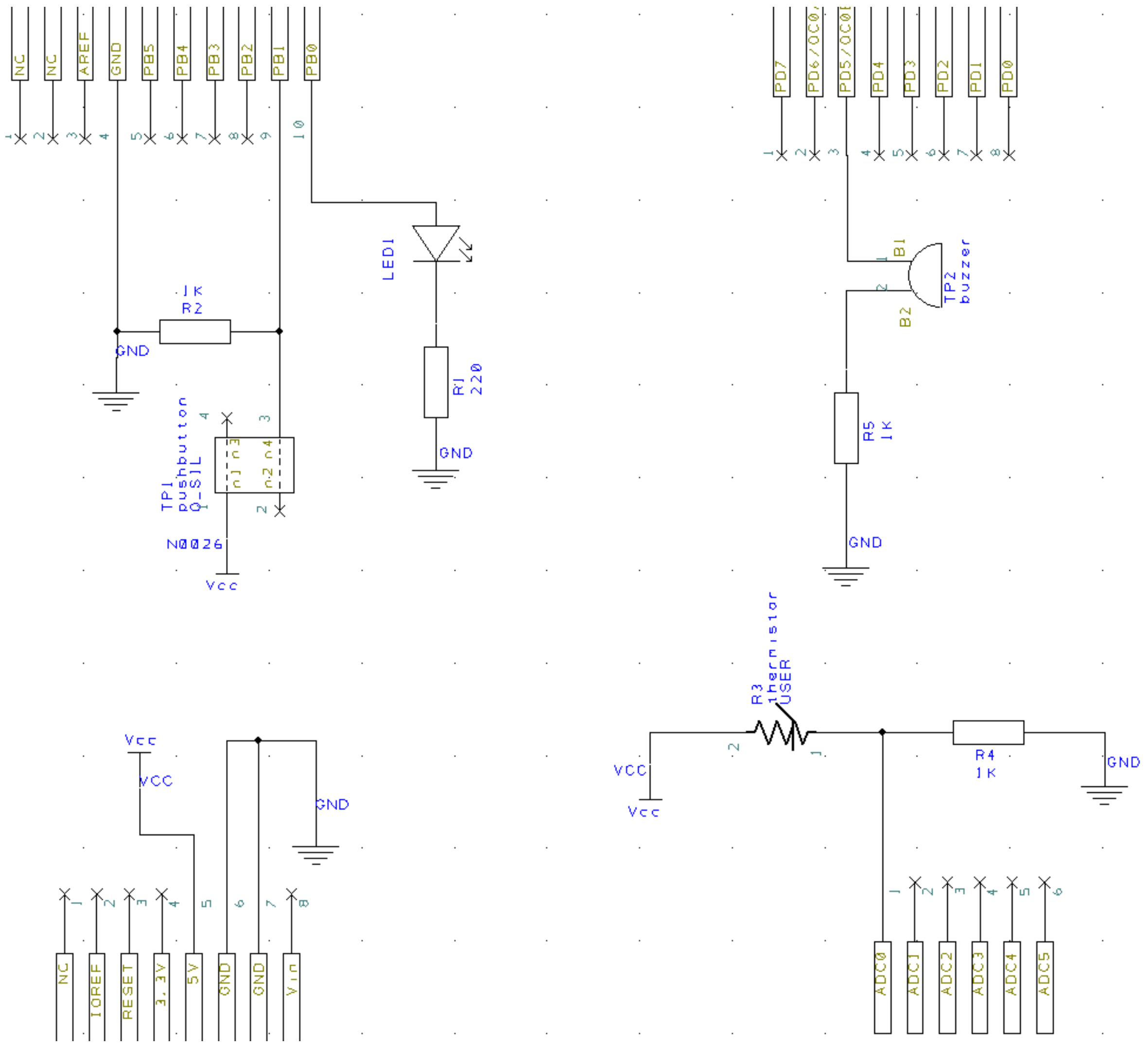


Figure 3.1. Heat Detector System's schematic.

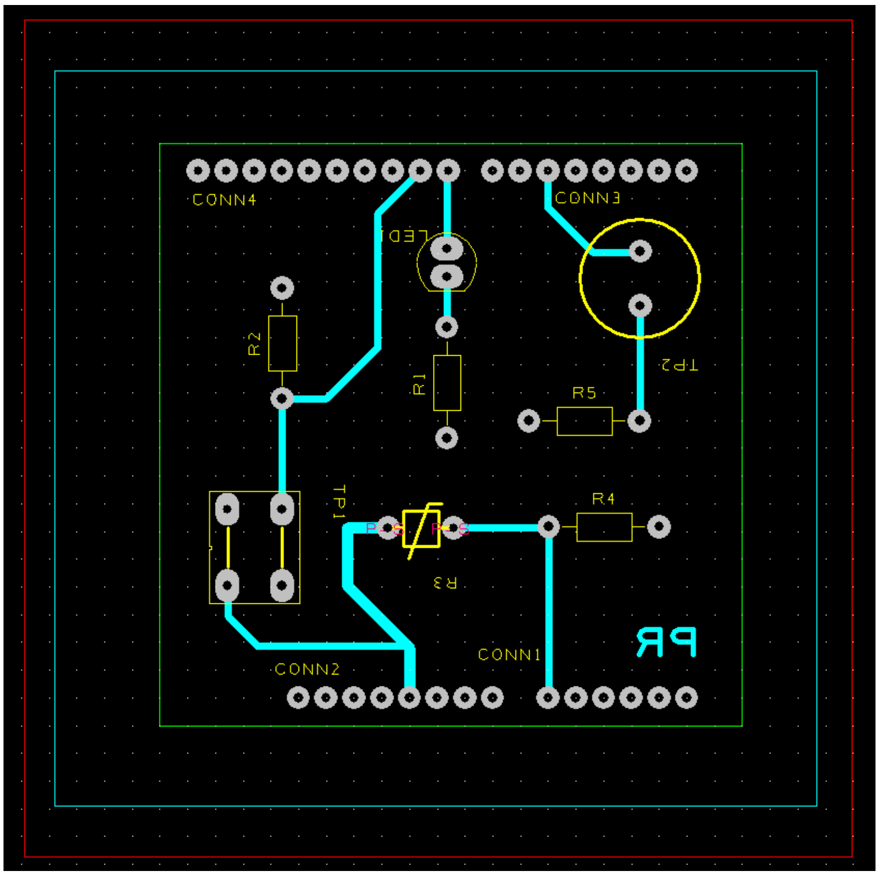


Figure 3.2. PCB Schematic without copper pour.

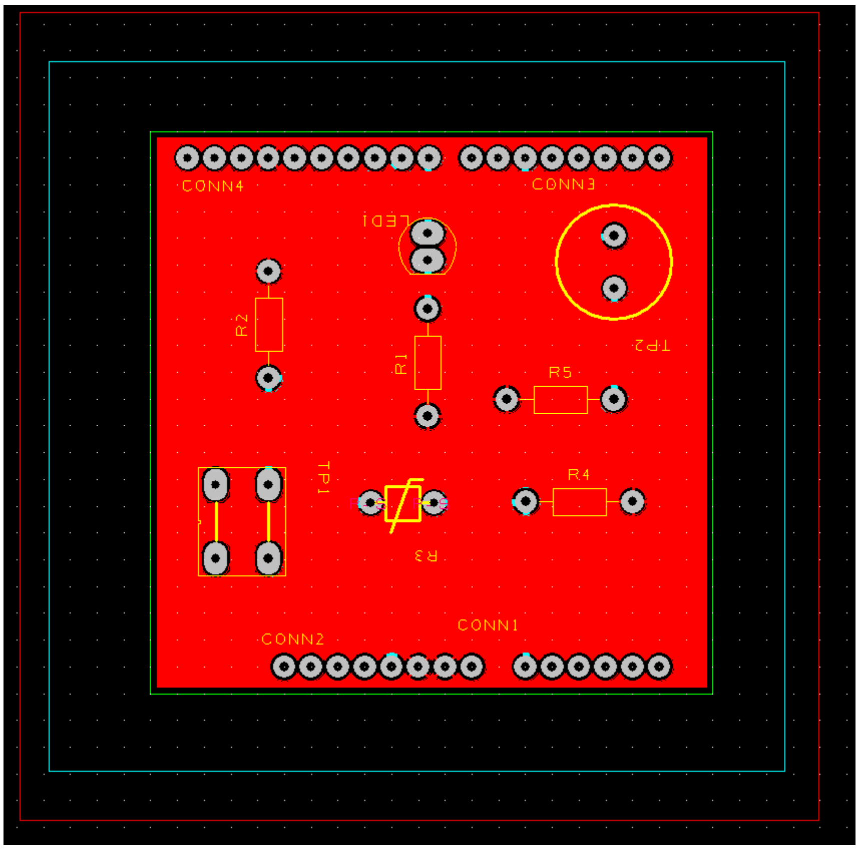
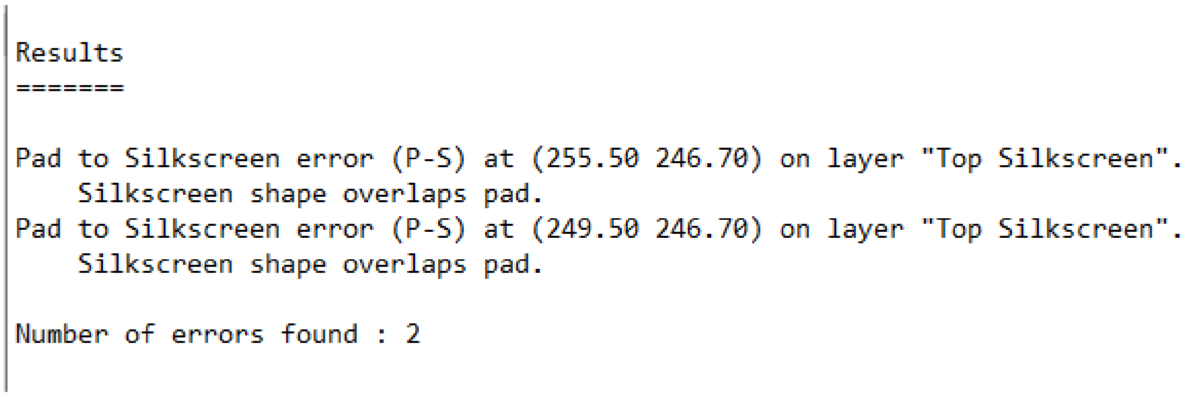


Figure 3.3. Schematic of the PCB board, copper poured.

This design was then verified to be correct by the Lab Technician. The results of the design rule check can be seen below. What is interesting is the padding error – which I believe to be caused by the insufficient spacing between the component and pads on the schematics. Sadly, I have not received my board and the consecutive experiments have been performed on the generic one.

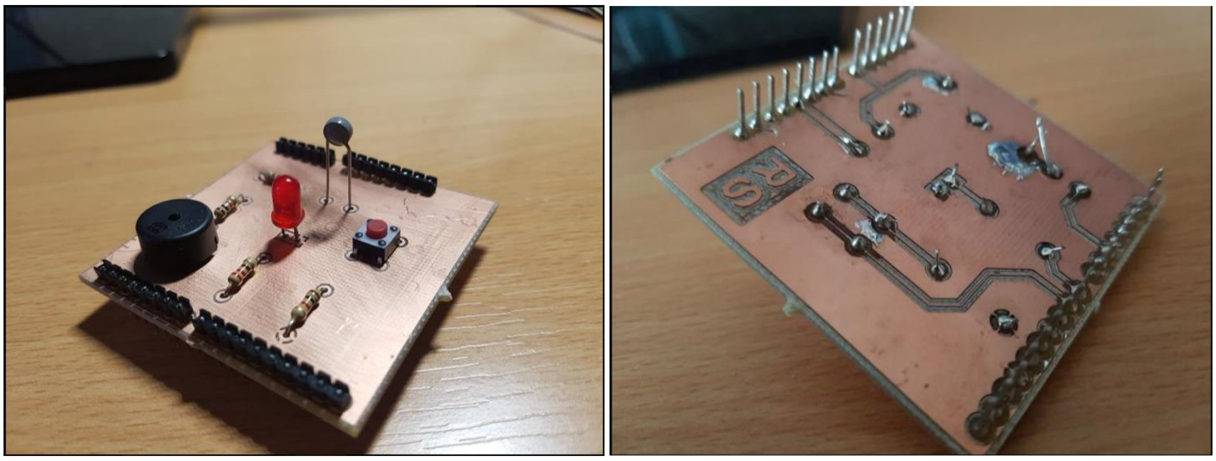


```
Results
=======

Pad to Silkscreen error (P-S) at (255.50 246.70) on layer "Top Silkscreen".
    Silkscreen shape overlaps pad.
Pad to Silkscreen error (P-S) at (249.50 246.70) on layer "Top Silkscreen".
    Silkscreen shape overlaps pad.

Number of errors found : 2
```

Figure 3.4 Design check results.



Figures 3.4 & 3.5. Top and bottom of the board after welding all the components in. All connections working. One of the Arduino plug-ins was welded very poorly, as I forgot to start with the leftmost pin and then move onto the rightmost one. You can also see the sky bridge between the buzzer and resistor – that had to be implemented after the poorly-maintained soldering tip stopped yielding appropriate results.

3.2 Testing the system & implementing the FSM

First and foremost, I have tested whether all the tested components were soldered on properly – I have created a simple script (please see appendix) to check all the connections. Having finished that, I went on to design.

The Finite-State Machine is fairly straightforward to make when following the instructions posted in the lab notes. I have decided to implement the Moore Machine that strictly associates it’s output with the state it is in – a decision based on preference. Please see the state diagram below.

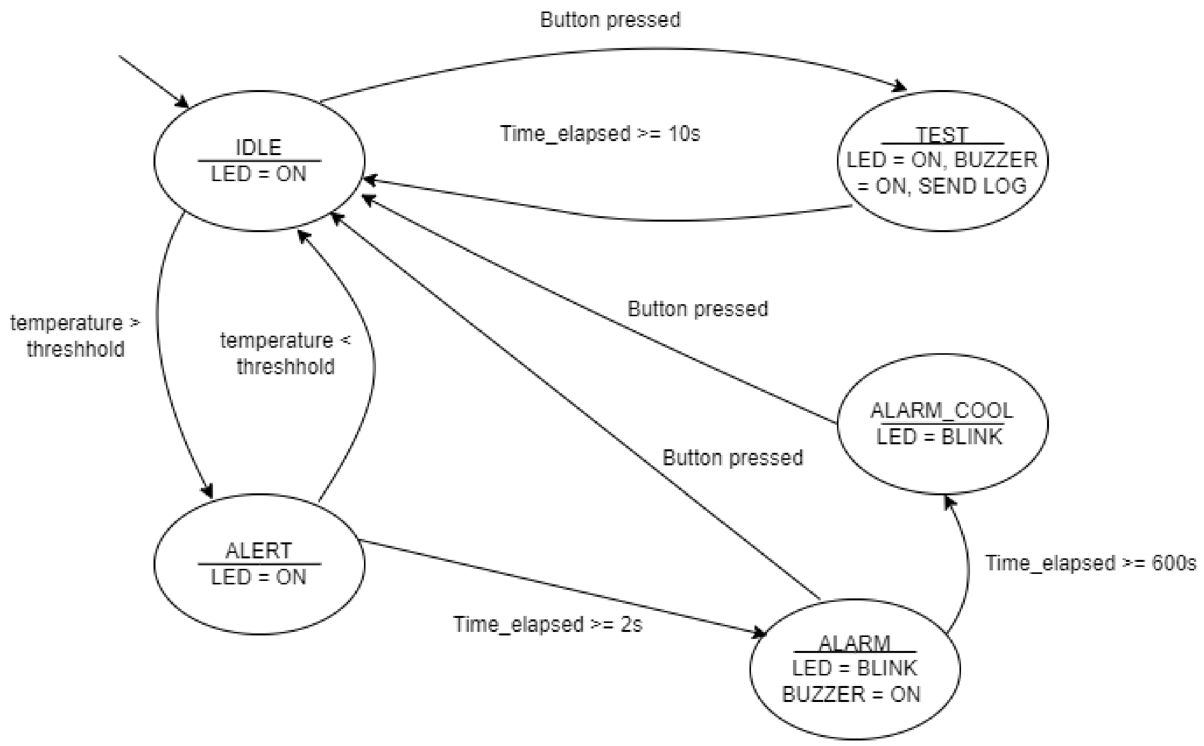


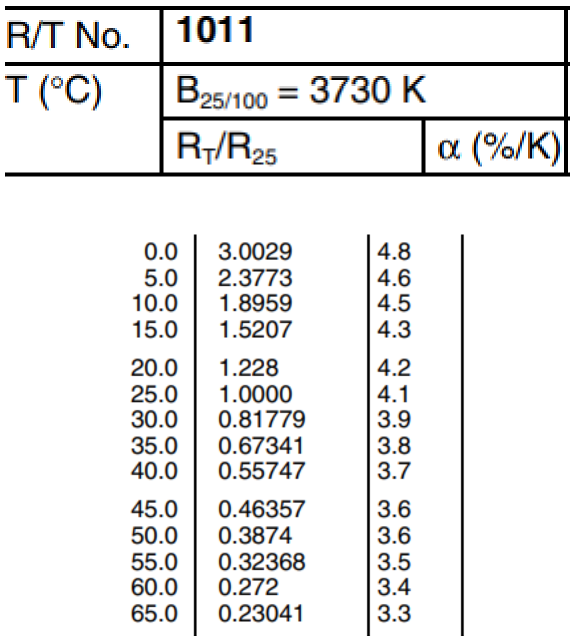
Figure 3.6. Design of the FSM. This FSM is not ideal as no alarm will occur after the first one without user acknowledgement, so we're assuming the user to be diligent enough to check the alarm after vacating the room for more than 10 minutes. However, for the sake of simplicity, it is good enough.

Next, the code has been implemented. For testing purposes of the thermistor, I set the threshold to 540 – a value that should activate the alarm after a few seconds of heating the thermistor up with human body heat (e. g. putting it between your fingers). For testing purposes, I set all the values in the code to some testable ones – for example, instead of 10 minutes I set 6 seconds to see if it properly switches states – the tests were successful. An interesting thing I have observed is when using the provided, secondary board (not the one that I soldered), my button transitions are on rising edge (e. g. when the button is let go), and the test state goes off upon initialization. The reason for that is rather interesting – it is because the old board designs have their buttons connected to GND, while the new designs are connected to Vcc. Fortunately, this does not affect the performance of the system in any way (besides the one outlined above) and can still be used normally.

I have set the delays of checking between states of 100ms. I still find that rather high, however it is enough to find any spurious readings without "tiring" the Arduino too much (if any – all the components have pulldown/up resistors). This results in slight inconsistencies as, for example, the alarm will go off after 2.2 seconds instead of 2 seconds. This could be quite obviously easily remedied, however for code's readability I left it as is – I didn't find it changing anything in terms of design.

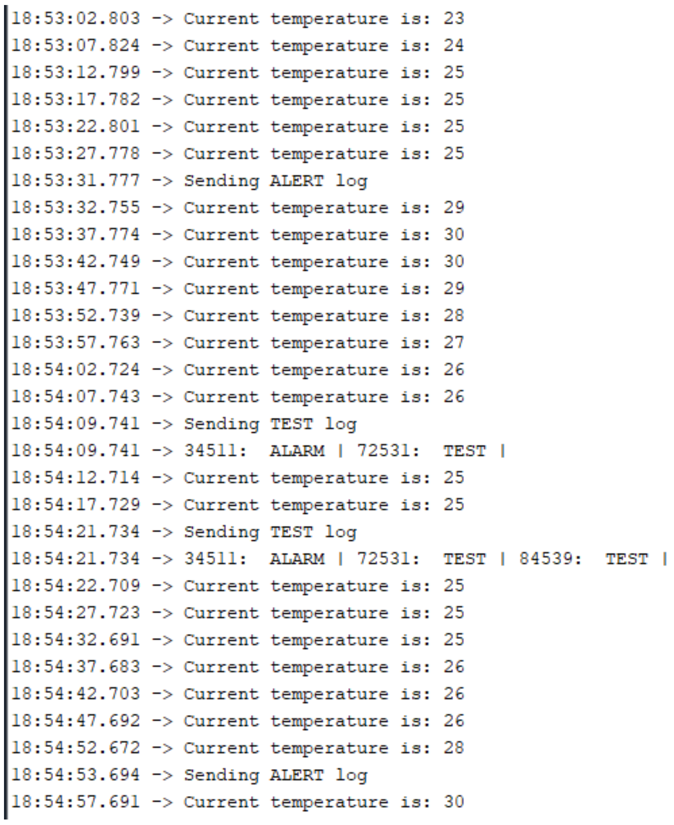
Now, after implementing timestamp logs of both AVG and TEST, there was time to determine the temperature of the resistor. Now, as luck would have it, both the thermistor and resistor are 1 kiloohm, meaning that at room temperature the voltage should split evenly – and roughly it does (1024/2 = 512), which complies with the findings. The higher the resistance of the thermistor, the lower will be the resistance – and as such, higher will the temperature reading be. The resulting reading on the pin will be: V = 1024*(1000Ohm/(1000Ohm + Rtherm)). Out of that, we have to extract Rtherm to be able to determine the current temperature: Rtherm = 1024000/V – 1000. Now, the relationship is not linear – however, assuming that the electronics system will operate only in temperatures 0 to 60 degrees, I will assume that the alpha factor is constant and is 4%/K for a somewhat accurate reading. Having done some linear algebra (at 1000 Ohm, 25 deg and at 960 Ohm, 26 deg) with that assumption, T= -0.025*Rtherm + 50.

A much better solution would be to implement a lookup table of sorts – but the current solution is sufficient for this application.



R/T No.	1011	
T (°C)	$B_{25/100}$ = 3730 K	
	R_T/R_{25}	α (%/K)
0.0	3.0029	4.8
5.0	2.3773	4.6
10.0	1.8959	4.5
15.0	1.5207	4.3
20.0	1.228	4.2
25.0	1.0000	4.1
30.0	0.81779	3.9
35.0	0.67341	3.8
40.0	0.55747	3.7
45.0	0.46357	3.6
50.0	0.3874	3.6
55.0	0.32368	3.5
60.0	0.272	3.4
65.0	0.23041	3.3

Figure 3.7. The non-linearity of the alpha factor. To be found in the data sheet for TDK-manufactured NTC thermistors.



```
18:53:02.803 -> Current temperature is: 23
18:53:07.824 -> Current temperature is: 24
18:53:12.799 -> Current temperature is: 25
18:53:17.782 -> Current temperature is: 25
18:53:22.801 -> Current temperature is: 25
18:53:27.778 -> Current temperature is: 25
18:53:31.777 -> Sending ALERT log
18:53:32.755 -> Current temperature is: 29
18:53:37.774 -> Current temperature is: 30
18:53:42.749 -> Current temperature is: 30
18:53:47.771 -> Current temperature is: 29
18:53:52.739 -> Current temperature is: 28
18:53:57.763 -> Current temperature is: 27
18:54:02.724 -> Current temperature is: 26
18:54:07.743 -> Current temperature is: 26
18:54:09.741 -> Sending TEST log
18:54:09.741 -> 34511:  ALARM | 72531:  TEST |
18:54:12.714 -> Current temperature is: 25
18:54:17.729 -> Current temperature is: 25
18:54:21.734 -> Sending TEST log
18:54:21.734 -> 34511:  ALARM | 72531:  TEST | 84539:  TEST |
18:54:22.709 -> Current temperature is: 25
18:54:27.723 -> Current temperature is: 25
18:54:32.691 -> Current temperature is: 25
18:54:37.683 -> Current temperature is: 26
18:54:42.703 -> Current temperature is: 26
18:54:47.692 -> Current temperature is: 26
18:54:52.672 -> Current temperature is: 28
18:54:53.694 -> Sending ALERT log
18:54:57.691 -> Current temperature is: 30
```

Figure 3.8. Testing the system with human touch and a bit of rubbing. Room temperature of roughly 22-23 degrees.

Appendix.

Code for Lab1_Ex3

```
#include <EE3580_lab1.h>

#include <stdio.h>


char command[30];

char myBufferCommand[50];

char blinkCommand[] = "blink";

char color[10];

char green[] = "green"; //PB4

char red[] = "red";    //PB5

int noOfBlinks = -1;


void setup() {

 Serial.begin(9600); // initialized the UART

 DDRB = (1<<PB5 | 1<<PB4); // set the direction of PB5 and PB4 as output

 pinMode(11, INPUT_PULLUP);

}

void loop()

{

 // if blink found,

 getstring(myBufferCommand);

 Serial.println(myBufferCommand);

 sscanf(myBufferCommand, "%s %s %d", command, color, &noOfBlinks);

 delay(600);

 int isEqual = strcmp(command, blinkCommand);

 if(isEqual==0){

  Serial.println("Blink command called.");

  if(noOfBlinks >20 || noOfBlinks <0){

    Serial.println("Incorrect number of blinks supplied.");

    return;
  }
  if(strcmp(color,green)==0){
   Serial.println("Green blinking");
   for(int i = 0; i <noOfBlinks*2; i++){
   PORTB ^= (1<<PB4);
   delay(200);
   }
  }
  else if(strcmp(color, red)==0){
   Serial.println("Red blinking");
   for(int i = 0; i <noOfBlinks*2; i++){
   PORTB ^= (1<<PB5);
   delay(200);
   }
  }
  else{
    Serial.println("Incorrect color supplied.");
    return;
  }
 }
}
```

<u>Code for Lab1_Ex4</u>

```
#include <EE3580_lab1.h>
#include <stdio.h>

char command[30];
char myBufferCommand[50];
char blinkCommand[] = "blink";
char printCommand[] = "print";
```

```
//globals for interrupt routine
long int timer;
long int timestampArray[10];
int currentIndex = 0;
int previousButton = 8;
int currentButton;
bool isCircular = false;

char color[10];
char green[] = "green"; //PB4
char red[] = "red";     //PB5
int noOfBlinks = -1;

ISR(TIMER2_COMPA_vect){
  timer++;
  currentButton = (PINB & (1<<PB3));
  //transition on FALLING EDGE (e.g. moment it's pressed, not moment it's let go)
  if(currentButton - previousButton < 0){
    if(currentIndex ==10){
      currentIndex=0;
      //if is circular, we have to keep that in mind when we're printing.
      isCircular = true;
    }
    timestampArray[currentIndex] = timer;
    currentIndex+=1;
  }
  previousButton = currentButton;

}

void setup() {
 Serial.begin(9600); // initialized the UART
 DDRB = (1<<PB5 | 1<<PB4); // set the direction of PB5 and PB4 as output
 pinMode(11, INPUT_PULLUP); //NO EXTERNAL PULLUP RESISTOR.
 setup_timer();
}
void loop()
{
 // if blink found,
 getstring(myBufferCommand);
 sscanf(myBufferCommand, "%s %s %d", command, color, &noOfBlinks);
 delay(600);
 int isEqualBlink = strcmp(command, blinkCommand);
 if(isEqualBlink==0){
  Serial.println("Blink command called.");
  if(noOfBlinks >20 || noOfBlinks <0){
    Serial.println("Incorrect number of blinks supplied.");
    return;
  }
  if(strcmp(color,green)==0){
   for(int i = 0; i <noOfBlinks*2; i++){
   PORTB ^= (1<<PB4);
   delay(200);
   }
  }
  else if(strcmp(color, red)==0){
   for(int i = 0; i <noOfBlinks*2; i++){
   PORTB ^= (1<<PB5);
   delay(200);
   }
  }
  else{
    Serial.println("Incorrect color supplied.");
    return;
  }
 }
  int isEqualPrint = strcmp(command, printCommand);
  if(isEqualPrint == 0){
    //If array is circular, we have to print all 10 entries
    //and know when to "reset" the counter.
    if(isCircular){
    for(int i =0; i < 10; i++){
      currentIndex+=1;
      Serial.print(timestampArray[currentIndex-1]);
      Serial.print("      ");


      if(currentIndex == 10){
        currentIndex = 0;
      }
      }
      Serial.println("");
    }
    //Array is not circular. Print as usual.
    else{
      for(int i =0; i < currentIndex; i++){
        Serial.print(timestampArray[i]);
        Serial.print("      ");
      }
      Serial.println("");
    }


  }
}
```

}

Code for lab2_Ex2

```
  //minimum frequency - 16000000/(256*(255+1)) = 244,1 Hz
 //maximum frequency - 16000000/(256*(1+1)) = 31 250 Hz
  #define MAX_FREQ 31250
  #define MIN_FREQ 244

  int prevRead = -1;
  uint8_t dutyCycle = 128;


void timer_setup(uint32_t freq, uint8_t duty_cycle)
{

 if(freq < MIN_FREQ || freq > MAX_FREQ){
  return;
 }
 cli();
 // Fast PWM, prescaler 256
 TCCR0A = 0x23;
 // prescaler 256 (0x0C = 256)
 TCCR0B = 0x0C;
 OCR0A = round(16000000/(256*freq) - 1); //how long till reset?
 //This wont work:
 //OCR0B = (duty_cycle/256)*(OCR0A + 1) - 1; //how long wave is on?
 //This will as duty_cycle/256 will not be zero anymore
 OCR0B = (duty_cycle*(OCR0A+1)/256) - 1;
 DDRD = (1<<PD5);
 sei();
}

void setup() {
  //timer_setup(1000, 128);
  //timer_setup(10000, 64);
  timer_setup(440, 255);
  Serial.begin(9600);
  Serial.println(OCR0A);
  Serial.println(OCR0B);
  }
void loop() {
  int thisRead = analogRead(0);
  //ensuring noise is irrelevant
  if(abs(thisRead - prevRead) > 4){
    //31 250 / 1024 distinct values = jump of 30.5 Hz + base address
    uint32_t newFreq = thisRead*30 + MIN_FREQ;
    Serial.println("Chosen frequency:");
    Serial.println(newFreq);
    timer_setup(newFreq,dutyCycle);
    prevRead = thisRead;
    Serial.println("The new values for the registers are:");
    Serial.println(OCR0A);
    Serial.println(OCR0B);
  }
  //Serial.println(analogRead(0));
  }
```

Code for Lab2_Ex3

```
#include <EE3580_lab2.h>

int prevRead = -1;

void setup() {
  carrier_62k(128);
```

```
 Serial.begin(9600);
 Serial.println(OCR0A);
 Serial.println(OCR0B);
 }
void loop() {
 int thisRead = analogRead(0);
 //ensuring noise in potentiometer is irrelevant
 if(abs(thisRead - prevRead) > 4){
  uint8_t dutyCycle = thisRead/4;
  //dutyCycle varies from 0 to 255, thisRead from 0 to 1023
  carrier_62k(dutyCycle);
  prevRead = thisRead;
 }
 Serial.println(analogRead(1));
 }
```

Code for lab2_ex4

```
#include <EE3580_lab2.h>
int prevRead = 0;
long int time = 0;
uint8_t wave[256]; // Look up table
uint16_t index, stepsize; // index and stepsize
uint32_t waveformFrequency; //making it 32 bit allows for operating on higher numbers temporarily.
uint16_t selectIndex;

void setup() {
 //start frequency at max frequency
 waveformFrequency = 256;
 index = 0;
 stepsize = 1;
 //lut_setup_triangular(wave);
```

```
  //lut_setup_sawtooth(wave);
  lut_setup_sinusoidal(wave);
  //setup of the "square wave". It doesn't mnatter
  //what wave there is as we are only extracting the
  //DC component anyway.
  carrier_62k(128);
  //setting up the interrupt for updating OCR0B
  timer1_setup();
  Serial.begin(19200);



  }

ISR(TIMER1_COMPA_vect){
  time++;
  //stepsize = freq(in fixed point) * 32 ms (in fixed point) bit shifted right by 8. If we had higher
frequencies, there'd
  //generally be much more wiggle room, but my highest frequency is 1Hz, so not a lot of stepsizes (1-
4).
  stepsize = ((waveformFrequency * ((32<<8)/1000))>>8) + 1;
  //select the index
  int selectIndex = index>>8;
  OCR0B = wave[selectIndex];
  index+=stepsize;
  if(selectIndex>255){
    index = 0;
  }
}
void loop() {
  int thisRead = analogRead(0);
  //ensuring noise in potentiometer is irrelevant
  if(abs(thisRead - prevRead) > 4){
```

```
  //turn potentiometer into fixed-point representation,
  //with max being 1.024 Hz (Because it's the easiest ;))
  waveformFrequency = thisRead>>3;
 }
 if(time%100 == 0){
  Serial.println(analogRead(1));
 }
 }
```

Code for Lab3_test

```
int button;
int led;
int buzzer;
int thermistor;

void setup() {
 // put your setup code here, to run once:
 // PB1 (pin 9) - button. PB0 (pin 8) - LED.
 //PD5 (pin 5) - BUZZER. ADC0 (A0) - thermistor.
 buzzer = 5;
 led = 8;
 button = 9;
 thermistor = A0;

 pinMode(buzzer, OUTPUT);
 pinMode(led, OUTPUT);
 pinMode(button, INPUT_PULLUP);
 //no need for pinMode for thermistor

 Serial.begin(9600);
}

void loop() {
 // put your main code here, to run repeatedly:
 delay(1000);
 tone(buzzer, 100);
 Serial.println(analogRead(thermistor));
 if(digitalRead(button) == HIGH){
   digitalWrite(led, LOW);
 }
 else{
   digitalWrite(led, HIGH);
 }

}
```

Code for Lab3_FSM

```
int button;
int led;
int buzzer;
int thermistor;

//counting to show temperature.
int counter;

//analysing button clicks
int previousButton = 1;
int currentButton;



enum {IDLE, TEST, ALERT, ALARM, ALARM_COOL} state;
```

```
long int lastPressedTime;
int thermistorRead;
int thermistorThreshhold;
bool litUp = true;
bool firstPressed = false;

//yet again I am reusing code from previous labs, for circular arrays.
long int timestampArray[10];
bool testEventArray[10];
int currentIndex = 0;
bool isCircular = false;



void setup() {
  // put your setup code here, to run once:
  // PB1 (pin 9) - button. PB0 (pin 8) - LED.
  //PD5 (pin 5) - BUZZER. ADC0 (A0) - thermistor.
  buzzer = 5;
  led = 8;
  button = 9;
  thermistor = A0;

  //pinMode(buzzer, OUTPUT);
  pinMode(led, OUTPUT);
  pinMode(button, INPUT_PULLUP);
  //no need for pinMode for thermistor

  //for testing purposes
  thermistorThreshhold = 540;
  state = IDLE;
  Serial.begin(9600);
```

```
}
//record an alarm event.
void sendLog(bool type){
  if(currentIndex ==10){
      currentIndex=0;
      //if is circular, we have to keep that in mind when we're printing.
      isCircular = true;
    }
  timestampArray[currentIndex] = millis();
  testEventArray[currentIndex] = type;
  currentIndex+=1;
}

//check if a button was pressed.
boolean buttonPressed(){
  currentButton = digitalRead(button);
  //transition on falling edge
  if(currentButton - previousButton < 0){
    previousButton = currentButton;
    return true;
  }
  previousButton = currentButton;
  return false;
}

//print the last 10 events.
void printLog(){
  if(isCircular){
    for(int i =0; i < 10; i++){
      currentIndex+=1;
      Serial.print(timestampArray[currentIndex-1]);
      Serial.print(":  ");
      if(testEventArray[currentIndex -1]){
        Serial.print("ALARM");
      }
      else{
        Serial.print("TEST");
      }
      Serial.print(" | ");

      if(currentIndex == 10){
        currentIndex = 0;
      }
      }
      Serial.println("");
    }
    //Array is not circular. Print as usual.
    else{
      for(int i =0; i < currentIndex; i++){
        Serial.print(timestampArray[i]);
        Serial.print(":  ");
        if(testEventArray[i]){
          Serial.print("ALARM");
        }
        else{
          Serial.print("TEST");
        }
        Serial.print(" | ");
      }
      Serial.println("");
    }
```

```
}

void update_fsa(){
 //check state every 500ms. Needs button to be held up to 500ms.
 delay(500);
 thermistorRead = analogRead(thermistor);

 if(counter%10 == 0){
  //Serial.println(thermistorRead);
  Serial.print("Current temperature is: ");
  long int thermistorVal = 1024000/thermistorRead -1000;
  int temp = -1*thermistorVal/40 + 50;
  Serial.println(temp);
 }
 counter+=1;

 switch(state){
  case IDLE:
   digitalWrite(led, HIGH);
   noTone(buzzer);
   if(thermistorRead > thermistorThreshhold){
    state = ALERT;
   }
   else if(buttonPressed()){
    state = TEST;
   }
   break;

  case TEST:
   //run this only on entering the state.
   if(!firstPressed){
    firstPressed = !firstPressed;
    //For test purposes, we treat it as an "ALARM" event
    Serial.println("Sending TEST log");
    sendLog(false);
    printLog();
    lastPressedTime = millis();
  }
  digitalWrite(led, HIGH);
  tone(buzzer, 100);
  //10 seconds
  if(millis() - lastPressedTime > 10000){
    state = IDLE;
    //reset firstPressed variable.
    firstPressed = !firstPressed;
  }
  break;

case ALERT:
  if(!firstPressed){
    firstPressed = !firstPressed;
    //record every ALERT event.
    Serial.println("Sending ALERT log");
    sendLog(true);
    lastPressedTime = millis();
  }
  digitalWrite(led, HIGH);
  if(thermistorRead < thermistorThreshhold){
    state = IDLE;
    firstPressed = !firstPressed;
  }
  //2 seconds before alert
  else if(millis() - lastPressedTime > 2000){
    state = ALARM;
    firstPressed = !firstPressed;
  }

  break;
//alternate.

case ALARM:
  litUp = !litUp;
  if(!firstPressed){
    firstPressed = !firstPressed;
    lastPressedTime = millis();
  }
  digitalWrite(led, litUp);
  //arbitrary values.
  if(litUp){
    tone(buzzer, 200);
  }
  else{
    tone(buzzer, 100);
  }
  if(buttonPressed()){
    state = IDLE;
    firstPressed = !firstPressed;
  }
  //10 minutes
  if(millis() - lastPressedTime > 600000){
    state = ALARM_COOL;
    firstPressed = !firstPressed;
  }
```

```
      break;

    case ALARM_COOL:
      litUp = !litUp;
      digitalWrite(led, litUp);
      noTone(buzzer);
      if(buttonPressed()){
        state = IDLE;
      }
      break;

  }
}

void loop() {
  update_fsa();
}
```

Modified EE3580_lab2 library

```
#ifndef _EE3580_lab2
#define _EE3580_lab2

#define BASE_FREQ   62500UL
#define MIN_FREQ       245

void lut_setup_triangular(uint8_t* wave) {
    int i;
    for(i=0; i<128; i++)
        wave[i] = 2*i;
    for(i=128; i<256; i++)
        wave[i] = 511-2*i;
}

void lut_setup_sawtooth(uint8_t* wave) {
    int i;
    for(i=0; i<256; i++){
        wave[i] = i;
              }
}

void lut_setup_sinusoidal(uint8_t* wave) {
    double i;
    for(i=0; i<256; i++){
        wave[(int)i] = 128*sin((i/256)*2*3.14)+128;
    //debugging relic really, could convert this to lut_setup_triangular
    //but I left it this way
        if(i >= 128){
            wave[(int)i] = 128*sin((i/256)*2*3.14)-128;
        }
    }

}




void timer_setup(uint32_t freq, uint8_t duty_cycle)
{
uint32_t x;

     // check that the frequency in the correct range
     if (freq>BASE_FREQ || freq<MIN_FREQ)
         return;

     cli();
     // Fast PWM, prescaler 256
     TCCR0A = 0x23;
     TCCR0B = 0x0C;

     x = round(BASE_FREQ/freq);

     OCR0A = x - 1;
     OCR0B = ((x*duty_cycle)>>8) - 1;

     DDRD = (1<<PD5);
     sei();
}


void carrier_62k(uint8_t duty_cycle)
{
     cli();
     // Fast PWM, prescaler 256
     TCCR0A = 0x23;
     TCCR0B = 0x09;

     OCR0A = 0xFF;
     OCR0B = duty_cycle;

     DDRD = (1<<PD5);
     sei();
}
```

```
void timer1_setup() {
    cli();
    TCCR1A = 0x00;
    TCCR1B = 0x01 | (1 << WGM12);
    OCR1A = 0x7D0;
    TIMSK1 = (1<<OCIE1A);
    sei();
}

#endif
```