2022

Technology Verdieping

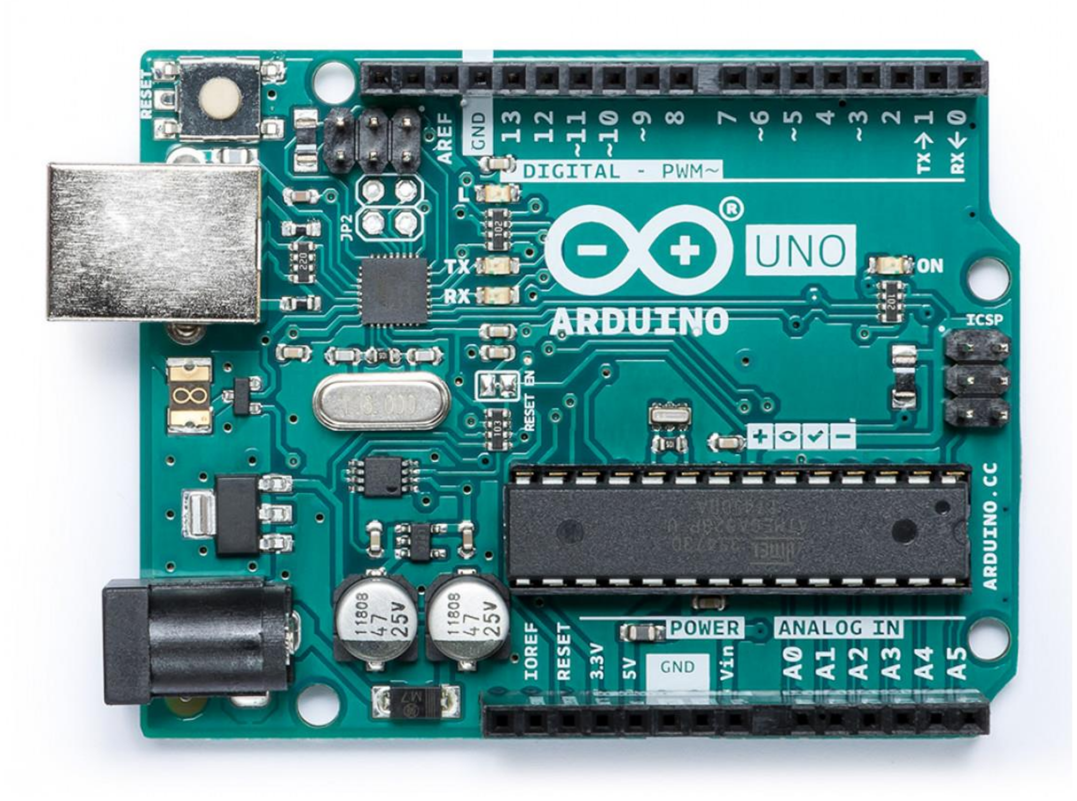
MAJOR SWITCH PREPERATION

MEES VAN LEUSDEN

Inhoud

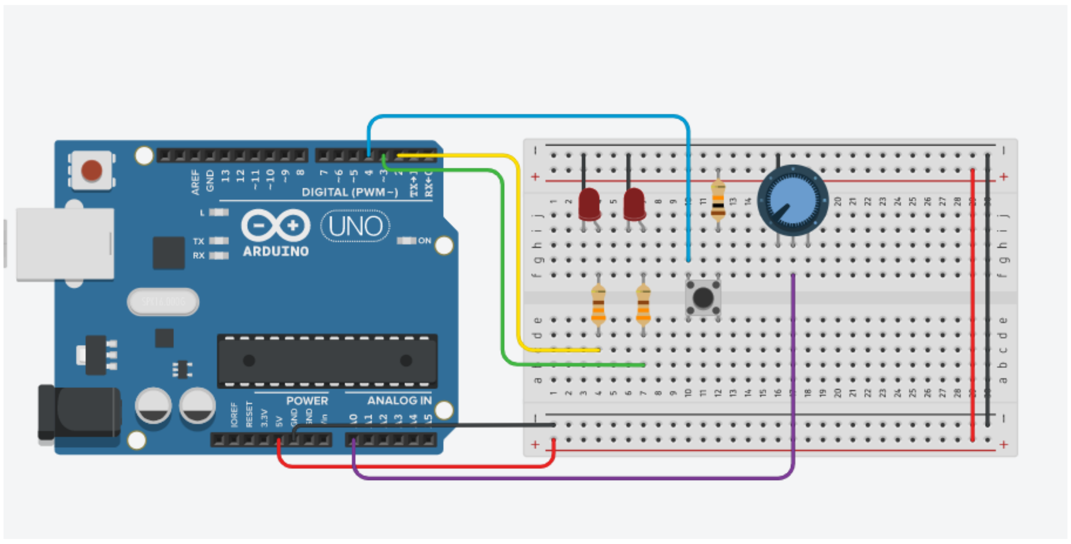
Github Repository: https://github.com/Tmoemes/TechS1PrepMees

2.2:Board Markings



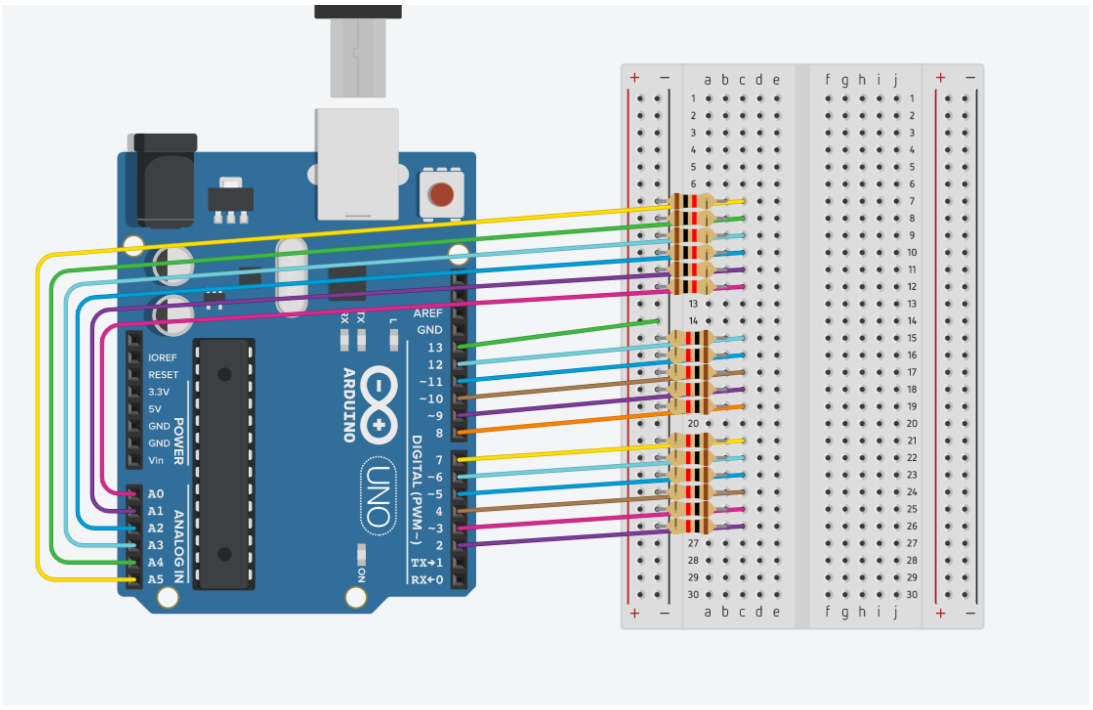
Pins a0-a5 are analog, 0-13 are digital. Pins 3,5,6,10 and 11 have a ~ which means they are capable of pwm output. Digital 0 and 1 are dedicated for serial communication and should therefore be generally avoided if possible.

2.5.1/2:Led Fade



2.5.3:Board Testing

My idea was to make the Arduino test itself by choosing one pin as the central test pin, lets say the last one so 13. To test the digital pins you set the central test pin first to INPUT and one by one ever other pin to OUTPUT into that pin, while doing this keep track for which pin the central one detects and signal. Do the reverse to check the INPUT of the pins. For the digital pin to the same but only for INPUT and also check the voltage that is measured.



This is a circuit I thought of, each pin is connected to pin 13 with a resistor. This way ever digital pin can automatically be tested for both input and output and the analog for input. There are a couple problem though, if pin 13 has a problem all the tests will fail but this will help you see that something is wrong. The other problem is that is don't know about automatically testing the full analog range of the analog inputs, you could do these by hand with a potentiometer but that would take a lot longer.

3.2.1:Blinking

This exercise I initially didn't read correctly so I did it differently from what the exercise suggests. Instead of making a timing function I made a led blink function. This does work similarly to a timing function but has the drawback of needing to create a timing variable for each led manually so it isn't very scalable.

```
long BlinkingLed(byte pin,long prevTime,long delay){
    if(currentTime - prevTime >= delay){
        digitalWrite(pin,!digitalRead(pin));
        return currentTime;
    }
    return prevTime; }
```

After reading the exercise again I created another version that is more scalable and less manual work.

```
long timers[10];
unsigned long currentTime = 0;
void StartTimers(){
    for(int i; i<sizeof(timers); i++){
        timers[i] = 0;
}  }
bool CheckTimer(int index,int delay){
    if(currentTime - timers[index] >= delay){
        timers[index] = currentTime;
        return true;
    } return false;}
```

This time I made an array to keep track of the timing of each timer. With this method you can use many timers at the same time without much extra code. In this instance there are 10 timers available but by increasing the size of the *timers* array there can be practically infinite. For both these methods you need to make sure to update the *currentTime* variable each loop.

3.2.2:Debounce

This one I also did in 2 parts, first in the loop to understand the logic.

```
void loop(){
    int state = digitalRead(buttonPin);
    if(state != prevState){ debounceTime = millis();}
    if(millis() - debounceTime >= debounceDelay){

        if(state != buttonState){
            buttonState = state;
            if(buttonState == HIGH){digitalWrite(ledPin,!digitalRead(ledPin));}
        }
    } prevState = state;
}
```

This is a very basic and confusing way to do this, it is also not very expandable because it uses a lot of code and variables for a single button.

3.2.3:debounce function

This exercise suggests making a separate function to make this code cleaner but that would still require many variables to work. I thought it would be more interesting to immediately make it into a class to fix most of the problems and start creating a sort of library of basic functions.

```
byte Button::read(){
    int state = digitalRead(_pin);
    if(state != _prevState){ _debounceTime = millis();}
    if(millis() - _debounceTime >= _debounceDelay){

        if(state != _buttonState){
            _buttonState = state;
            if(_buttonState == HIGH){return HIGH;}
        }
    }
    _prevState = state;
    return LOW;
}
```

This is how I made the function to read the button state, it is mostly the same as the previous one except it return the value instead of immediately using it.

To us classes it also requires a header file.

```
#ifndef Button_h
#define Button_h
#include "Arduino.h"
class Button{
    public:
    Button(byte pin, unsigned long debounceTime);
    byte read();
    void setupButton();
    private:
    byte _pin;
    unsigned long _debounceDelay;
    unsigned long _debounceTime;
    int _buttonState;
    int _prevState;
};
#endif
```

This took me a while to figure out and get working because it is quite a bit different from how I'm used to making classes. I understand the header file is kind of like an interface in other languages but the usage is different.

Extra Classes:

After figuring out how to create classes I also made a timer and an led class to more easily do these basic functions in future projects.

The timer class works pretty much exactly the same as before only this time in a different class.

```
bool Timer::CheckTimer(int index,int delay){
    currentTime = millis();
    if(currentTime - timers[index] >= delay){
    timers[index] = currentTime;
    return true;
    }
return false;
}
```

I did have a bit of a problem with the timer, I wanted to size of the timers array to be assignable when initialising the timer object. I could sadly not figure out how to initialise an array in the header with without specifying a size to then change it in the cpp file.

```
#ifndef Timer_h
#define Timer_h
#include "Arduino.h"
class Timer{
    public:
    Timer();
    void init();
    bool CheckTimer(int index, int delay);
    private:
    long timers[32];
    unsigned long currentTime = 0;
}; #endif
```

For a temporary solution I ended up setting the amount of timers to 32 because this seemed like a decent amount for most everything I need right now.

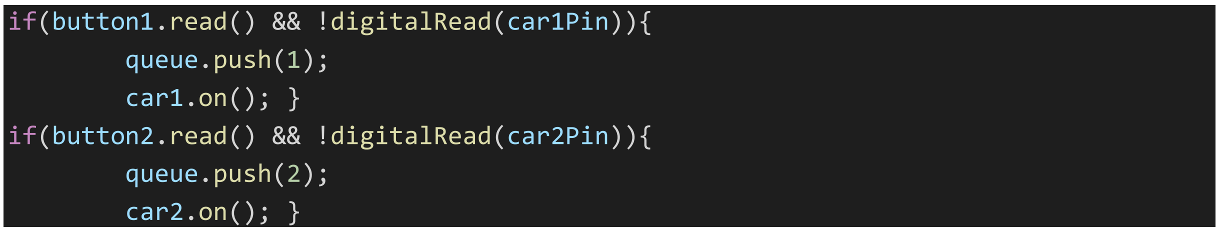
The other class I made was for controlling leds and is a lot simpler.

```
Led::Led(byte pin) {
    this->pin = pin;
    pinMode(pin,OUTPUT);
    off();
}
void Led::on(){
    digitalWrite(pin,HIGH);
}
void Led::off(){
    digitalWrite(pin,LOW);
}
void Led::toggle(){
    digitalWrite(pin,!digitalRead(pin));
}
```

So far I only added the functions for on, off and toggle. This class does the least but help a bit with the amount of code that has to be written.

3.3.1:states

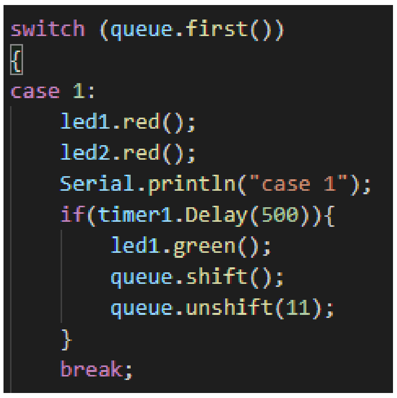
When I first read this challenge it took me a little while to fully understand how the crossing was supposed to function and how to implement the concept of state machines into that. I immediately realised that making a state for every possible combination of lights and waiting cars wasn't feasible. Now I had the problem of keeping track of the cars while using a lot less states. My solution was to have the states only keep track of a single traffic light but keep track of when they should switch to green using a queueing system. For the queueing system I used a library called CircularBuffer. The queueing system works in that whenever a car drives up to a traffic light the initial state for that light gets added to the end of the queue.



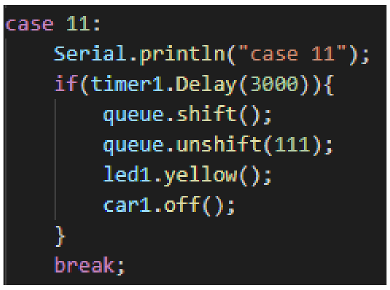
```
if(button1.read() && !digitalRead(car1Pin)){
        queue.push(1);
        car1.on(); }
if(button2.read() && !digitalRead(car2Pin)){
        queue.push(2);
        car2.on(); }
```

These 2 if statements represent the 2 sensors in the road. They check if the sensor(in this case a button) is pressed and if there isn't already a car there by checking if the light I used to represent the cars is off(this could also be done with variables instead but this was simpler in this case). When a car is detected the corresponding light's state gets pushed to the end of the queue and a light representing a waiting car gets turned on.

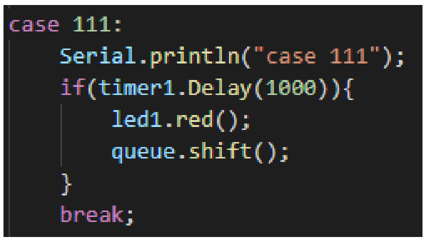
With this system the first car that arrives at the intersection will have it's light turn green first and a second car can arrive but it's light won't be touched until the first one is done.



```
switch (queue.first())
{
case 1:
    led1.red();
    led2.red();
    Serial.println("case 1");
    if(timer1.Delay(500)){
        led1.green();
        queue.shift();
        queue.unshift(11);
    }
    break;
```



```
case 11:
    Serial.println("case 11");
    if(timer1.Delay(3000)){
        queue.shift();
        queue.unshift(111);
        led1.yellow();
        car1.off();
    }
    break;
```



```
case 111:
    Serial.println("case 111");
    if(timer1.Delay(1000)){
        led1.red();
        queue.shift();
    }
    break;
```

The current state the program has to act on will always be the first one in the queue so that is the only thing the switch case looks at. So when a car arrives at traffic light 1, a 1 gets added to the queue. When the 1 at the front of the queue gets read the switch case goes to case one. In case 1 all the traffic lights get first all set to red to definitely make sure they are red at that point and then a delay of 500ms gets started. After the first time case 1 gets executed it will keep looping until the 500ms delay is over, then the light will be set to green and the first item in the queue is replaced with 11(1->11). Now the second case, case 11, will be executed, it does mostly the same except that it also turns off the car light to signify it has passed. Then lastly case 111, here the light gets set to red after 1sec and the first item in the queue finally get fully removed to allow the next state to process. So in total when a car arrives ,when it's their turn, the light will turn green for 3sec after 500ms, then the light turns yellow for 1sec and then finally back to red. The initial 500ms delay might seem strange but this is to allow a buffer between one light turning back to red and then another one turning green.

4.1.1:I/O techniques

- Digital on/off

  -For an explanation in the digital I/O pins of an Arduino the Arduino website has a good explanation of the specification and some uses.

  -Examples of components using digital I/O are push buttons, switches and LEDs.

- Analog

  -Analog Works quite similarly the digital pins but with a continuous range of voltages instead of just on and off, though most Arduino boards can only read Analog signals. Again the Arduino website has a good explanation on their working and some uses.

  -Most basic sensors like photo, o2 level or potentiometer use an Analog signal. Though most boards can't output Analog signals, they can output PWM signals which are functionally mostly the same and are used by for example: motors or LEDs.

- Serial UART

  -This is an explanation on the full serial communication method. This shows how to convert from parallel to serial shifting-in and this for the other way around. One more explanation: https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html

  -Serial can only be connected with one other component/device so there isn't really any sensors or actuators that directly use it, there are a lot of IC's that use it which in turn can control actuators or read from sensors.

- I2C

-I found a couple useful videos explaining the working and usage of I2C:

  I2C and how to use it, What is I2C, Basics for Beginners, How I2C Communication Works and How To Use It with Arduino

  -I2C can talk with a lot of different devices at once, for example gas/colour/motion sensors, motors driver, display drivers or even other microcontrollers.

- SPI

  -Video about SPI, explanation and tutorial on Arduino website plus and explanation of the software library.

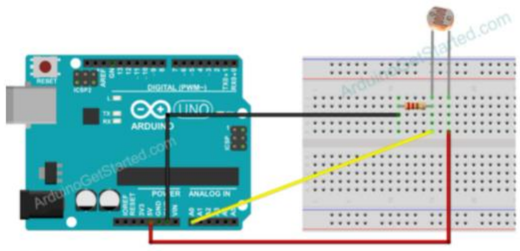
  -Used in the same type of components I2C is but is a lot faster while needing more cables for communicating with multiple devices.

Articles comparing UART,SPI,I2C:

- https://www.seeedstudio.com/blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses/
- https://www.makeuseof.com/how-uart-spi-i2c-serial-communications-work/

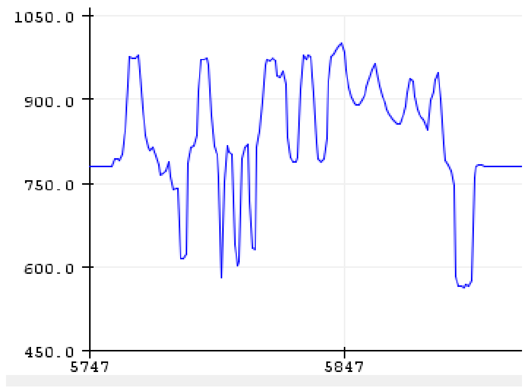
5.2.1/2:LDR

With this wiring diagram and small bit of code it was easy to set up and use the plotter of the Arduino IDE.

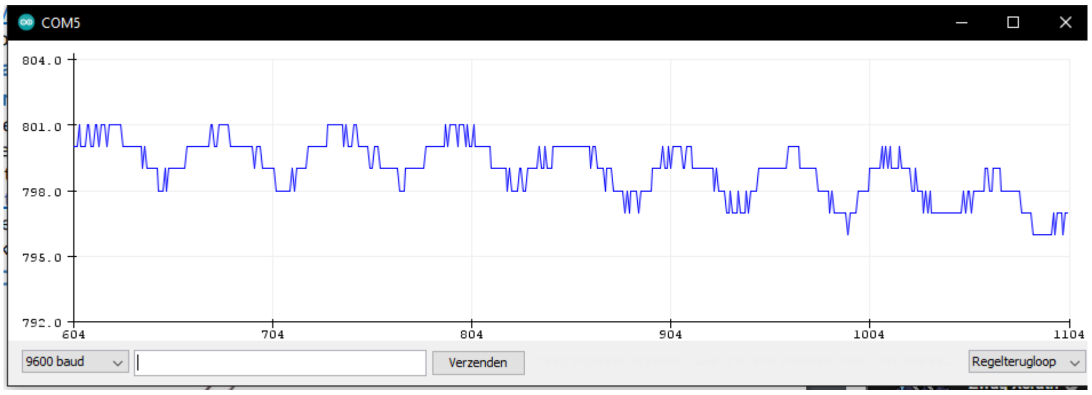


```
void setup() {
  Serial.begin(9600);
  pinMode(A0,INPUT);
}
void loop() {
  int lightLevel = analogRead(A0);
  Serial.println(lightLevel);
  delay(100);
}
```

This setup worked as expected, when shining a flashlight on the sensor it reads higher and when occluding it the reading becomes lower.



There was a weird thing that happens when left the sensor alone, there appeared this regular pattern of slightly varying values. When I searched online someone stated that it could be a voltage issue so I tried with 3.3v instead of 5 but got the same result. Another person recommended grounding the other Analog pins, this also didn't change anything. The only other things that I can't immediately test is fluctuation form USB power supply or just noise from the Analog pins themselves.



This thread gives some suggestions, one is to measure the LDR resistance and use a similar pull-up resistor or use a pot like the challenge suggests. First I measured the resistance if the LDR to be around 5k-10k. Then I used the pot to test some values, first a 100k pot which turned out to be too much. Then I used a 10k pot which needed to be maxed for the best performance. Eventually I realised that measuring a LDR isn't great because it changes from light so I looked it up, the resistance of my LDR is 10k-20k at 10 lux. So in the end the measuring was mostly correct and the original 10k I used was already good.

5.2.3:Scaling

This turned out to be a pretty simple thing to implement though it did take a little bit to understand how the map function works. When I understood how it just “maps” one range onto another one, which afterwards feels stupid how I didn’t immediately see that, it was as simple as plopping in the correct values into the function. One part of the challenge did still kind of stump me which was being able to set the amount of dimming. I don’t fully understand what “the amount of dimming means” so I took it as the maximum amount of dimming or setting a minimum brightness before turning off.

```
analogWrite(ledPin,map(lightLevel,dimmingEnd,dimmingStart,maxDimming,255));
```

Almost everything of this challenge happens on this line. ledPin and lightLevel are as normal. The map function takes dimmingEnd and dimmingStart as input range to, as the name suggests, start and stop dimming at a specific lightLevel. My solution for the amount of dimming was setting a minimum brightness at the output range, increasing this value will decrease the amount and rate of dimming because the output range will decrease.
The other parts of this code are turning off the light if the brightness becomes to high, or above the dimmingEnd value.

```
if(lightLevel > dimmingEnd){
        analogWrite(ledPin,LOW);
```

5.2.4: Running average

Taking the average of multiple values is something I had done before and with the given tutorial it wasn’t much problem to implement. With the way I did the starting and ending of the dimming I had also already done the second part of this challenge, easily changeable on/off threshold. I also took to creating a function for getting the average and making the amount of reading that are average and the rate easily changeable.

```
int readerAverage(){
    int total = 0;
    for (int i = 0; i < readingAmount; i++)
    {
        total += readings[i];
    }
    return int(total/readingAmount);
}
```

This is the function I made to get the average of the readings, it just sums up all the values and the divides that with the amount of values in the array.

```
for (int i = 0; i < readingAmount; i++)
    {
        readings[i]=initReading;
    }
}
```

In setup all the values in the array need to be initialized to a value, I made this initial value easily changeable with a variable so you can set it to a value that is close to an expected value. If you set the initial value to something that doesn’t make sense then before all of them are overwritten by actual readings you can get weird behaviour.
During this challenge I did have some trouble with filling the readings array, this turned out to luckily just be a because I forgot to replace a value with the correct variable somewhere so it wasn’t correctly going through the entire array.

5.2.5: Hysteresis

This challenge I had a little more trouble with because at first I didn't understand what the concept of hysteresis meant. When I finally did understand that is was kind of like a dead zone around the flipping point I was still a little confused as to how to actually implement it in the code I had. At first I didn't have a check for when the light should start turning on, just if it needed to be off and an else. Because of this I thought I would have to do it in the map function which didn't make sense. Eventually I did realize that I should just add the check and add the margin there.

```
if(readerAverage() >= dimmingEnd + hystMargin){
        analogWrite(ledPin,LOW);
    }
else if(readerAverage() < dimmingEnd - hystMargin){
        analogWrite(ledPin,map(readerAverage(),dimmingEnd,dimmingStart,0,255);
    }
```

This ended up being the way I did it, it's a lot simpler than what I originally though it would be. Now this still has a problem, if the program starts in this margin it wouldn't know what to do. To solve this problem I ended up copying the entire analogWrite line into setup but with the direct value of the LDR instead of the averaged value.

6.1.1: DHT library

The difference between resolution and accuracy is that resolution is the smallest difference the sensor can detect while the accuracy is the amount the sensor could be off from the true value. Because I don't have a DHT11 sensor to use for this challenge I couldn't try reading it myself but looking at some example code I do understand how to use it. If I read out one of these sensors for temperature I would expect it to close to the actual temperature but it will probably be off by 1 degree. According to the datasheet the DHT11 has a response time of between 6 and 30 seconds for temperature readings. For humidity the response time is between 6 and 15 seconds with a typical response time of 10 seconds.

Luckily with the DHT library getting the humidity or temperature from the sensor is quite easy after just a little bit of setting up.

For setup you need to create a DHT_Unified object with the data pin of the sensor and the sensor type(in this case the DHT11). The serial communication with the sensor also has to be initialized at setup.

```
DHT_Unified dht(DHTPIN, DHT11);
dht.begin();
```

To read the data it's a little bit more complicated than just reading it with one function. First you have to create a sensor event to get and store the data from the sensor. After getting the data with the event its as easy as reading the property of the event object.

```
sensors_event_t event;
dht.temperature().getEvent(&event);
Serial.print(event.temperature);
```

I also looked at some tutorials to better understand how these sensors work:
https://howtomechatronics.com/tutorials/arduino/dht11-dht22-sensors-temperature-and-humidity-tutorial-using-arduino/
https://lastminuteengineers.com/dht11-dht22-arduino-tutorial/
datasheet: https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf

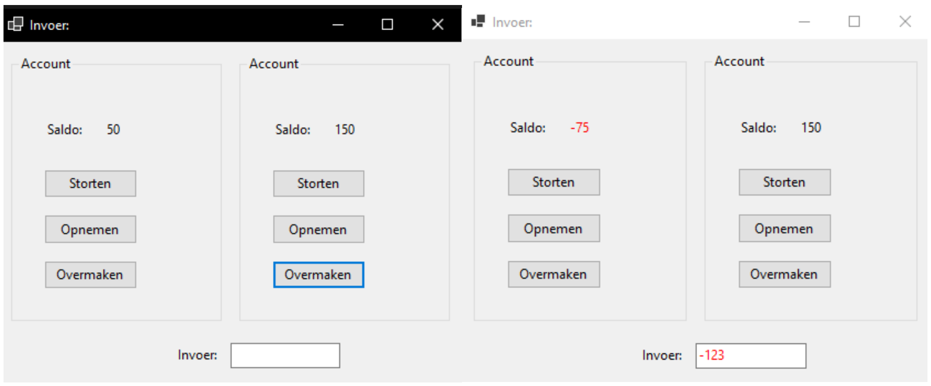
6.1.2: Ping sensor library

With this challenge I had the same problem as the previous one as I don’t have this sensor on hand either, though I will again do as much as I can without it. I could not find anything about the sensor having accuracy variance so using the library to do the data processing I would expect the distance reading to be quite accurate. Irregularly shaped object could interfere with distance measurements if too much of the sounds waves gets deflected away from the sensor. A sloped object will have a high chance of not working with an ultrasonic distance sensor, it’s very likely most, if not all, the sound waves will be deflected away from the sensor so no reading would be possible. Moving object would work like normal with a sensor that sends out only one pulse, though it would be measured only at one spot in space. Most ultrasonic sensors use multiple pulses however, usually 8, so if an object is moving each of the pulses will measure a slightly different distance. There is one very specific instance where a moving object should be able to be measured fine, that is if it’s moving perfectly perpendicular to the sensor and slow enough to stay in its 15° measuring angle during the 8 pulses. Lastly the case when there isn’t anything to measure in this case the sensor will just return its maximum measuring range as the distance measured.

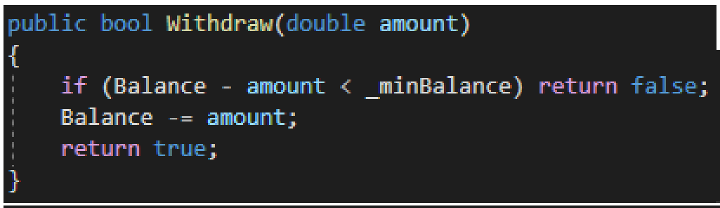
Datasheet: https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

7.1.3: Bank account

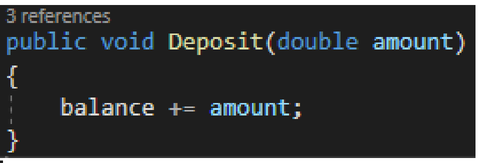
I hadn’t really used windows forms before this challenge so it took a little while to figure out how the visual ui designer was linked with the functioning code. When looking at the example interface of the challenge I immediately though of putting the account ui elements in a separate class to easily add more accounts later, sadly though I could not find an easy way of doing this so I ended up doing it manually.



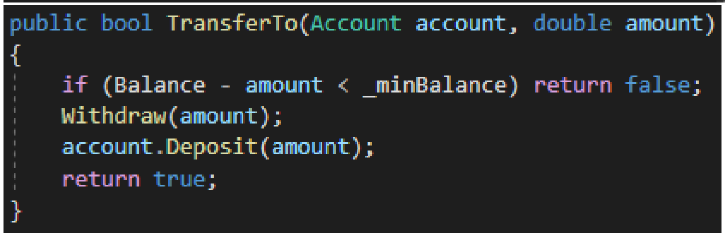
This challenge didn’t require much code, it’s pretty much only adding and subtracting a couple numbers. There were a couple criteria to keep in mind though, most made sense to me except the need for several parameters in the bank account constructor. The only constructor parameter I could think of was the starting balance, to get at least 2 I later decided to add a minimum balance too.



```
public bool Withdraw(double amount)
{
    if (Balance - amount < _minBalance) return false;
    Balance -= amount;
    return true;
}
```

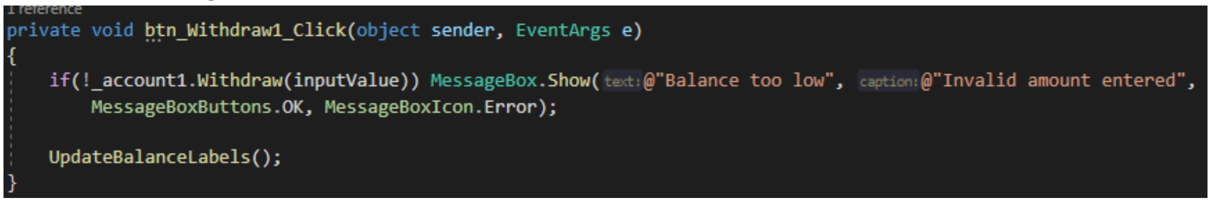


```
3 references
public void Deposit(double amount)
{
    balance += amount;
}
```



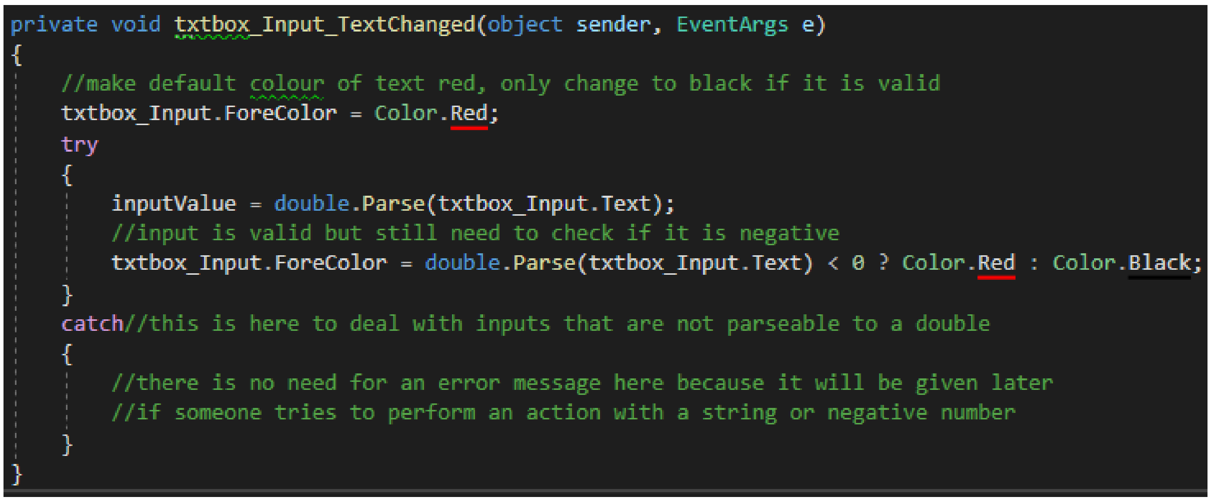
```
public bool TransferTo(Account account, double amount)
{
    if (Balance - amount < _minBalance) return false;
    Withdraw(amount);
    account.Deposit(amount);
    return true;
}
```

These are the functions that actually deal with changing an account’s balance and are located in the account class. Deposit is simple adding the inputted amount to the balance but both withdraw and transfer need to make sure to not get the balance too low. To give feedback for error messages I made both withdraw and transfer return a bool value indicating success or failure of transaction.



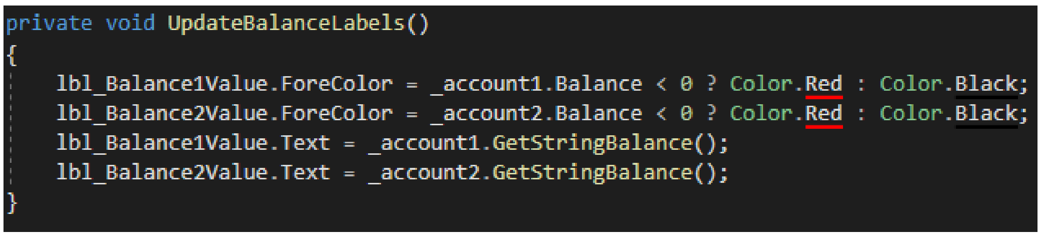
```
1 reference
private void btn_Withdraw1_Click(object sender, EventArgs e)
{
    if(!_account1.Withdraw(inputValue)) MessageBox.Show( text: @"Balance too low",  caption: @"Invalid amount entered",
        MessageBoxButtons.OK, MessageBoxIcon.Error);

    UpdateBalanceLabels();
}
```

The code for the buttons is mostly the same for all of them, they execute a function of the corresponding account. Then if the transaction is successful the balance gets updated otherwise an error message is shown.



```
private void txtbox_Input_TextChanged(object sender, EventArgs e)
{
    //make default colour of text red, only change to black if it is valid
    txtbox_Input.ForeColor = Color.Red;
    try
    {
        inputValue = double.Parse(txtbox_Input.Text);
        //input is valid but still need to check if it is negative
        txtbox_Input.ForeColor = double.Parse(txtbox_Input.Text) < 0 ? Color.Red : Color.Black;
    }
    catch//this is here to deal with inputs that are not parseable to a double
    {
        //there is no need for an error message here because it will be given later
        //if someone tries to perform an action with a string or negative number
    }
}
```

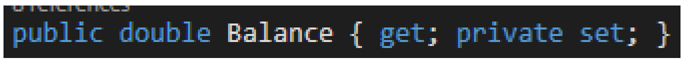
Getting the value from the input field seems easy but there were a couple things to keep in mind. Everything could be inputted so just parsing the input to a double isn't sufficient, this will fail on anything other that numbers and even when you start typing a negative number because of the first "-". To make sure the program doesn't crash every time something other than a number is typed I used a try catch clause. I also decided to change to colour of the text depending on if the input is valid or not.



```
private void UpdateBalanceLabels()
{
    lbl_Balance1Value.ForeColor = _account1.Balance < 0 ? Color.Red : Color.Black;
    lbl_Balance2Value.ForeColor = _account2.Balance < 0 ? Color.Red : Color.Black;
    lbl_Balance1Value.Text = _account1.GetStringBalance();
    lbl_Balance2Value.Text = _account2.GetStringBalance();
}
```

Lastly I made a function for updating the balance labels because this need to happen after every action. Since I added a minimum balance the balances can go negative so I coloured these accordingly too.

(It looks like I'm using a public variable for balance in the last function, this is only for getting the value, I made the setter private.)



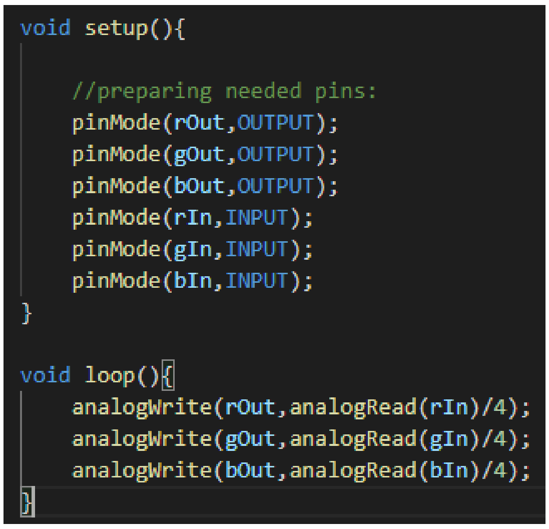
```
public double Balance { get; private set; }
```

7.1.4-7.1.7:

See chapter 3.

8.2.1 PWM LED:

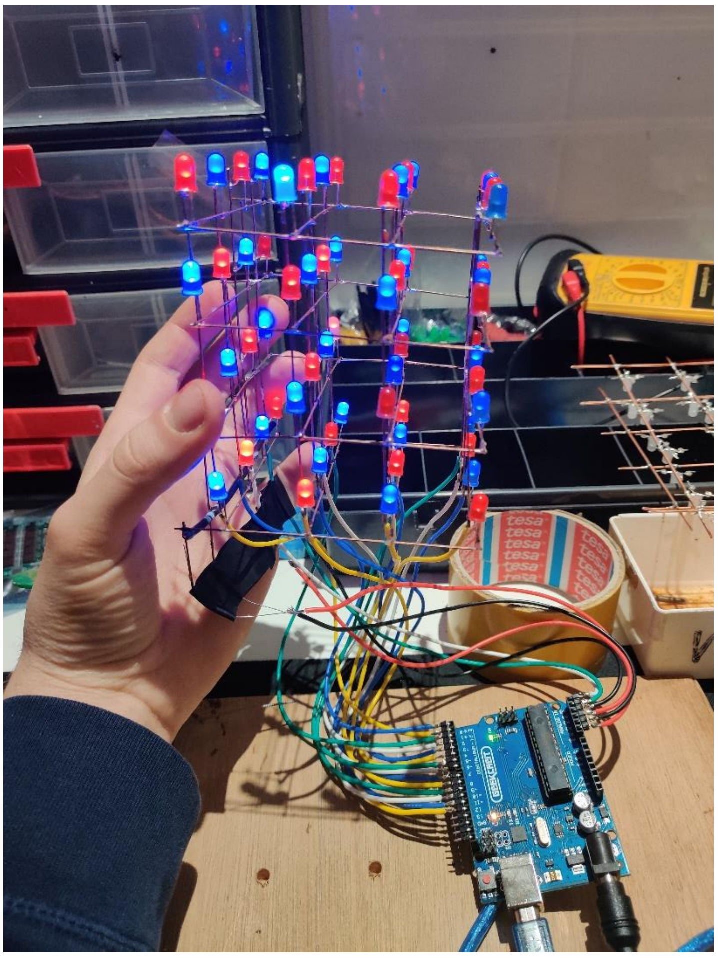
I previously made 2 projects akin to this challenge, an RGB LED where you can individually control the red green and blue brightness and an led cube.



```
void setup(){

    //preparing needed pins:
    pinMode(rOut,OUTPUT);
    pinMode(gOut,OUTPUT);
    pinMode(bOut,OUTPUT);
    pinMode(rIn,INPUT);
    pinMode(gIn,INPUT);
    pinMode(bIn,INPUT);
}

void loop(){
    analogWrite(rOut,analogRead(rIn)/4);
    analogWrite(gOut,analogRead(gIn)/4);
    analogWrite(bOut,analogRead(bIn)/4);
}
```

The RGB LED project was quite simple. There are 3 potentiometers, one for each colour. The inputs are directly fed into the analogwrite of the respective led output pins, though the value is divided by 4 to compensate for the resolution difference. I made this project before I knew about the map function so I just divided by 4. My solution works almost completely the same except that the range is ever so slightly different, 0-255.75 instead of 0-255 (because 1023/4=255.75).

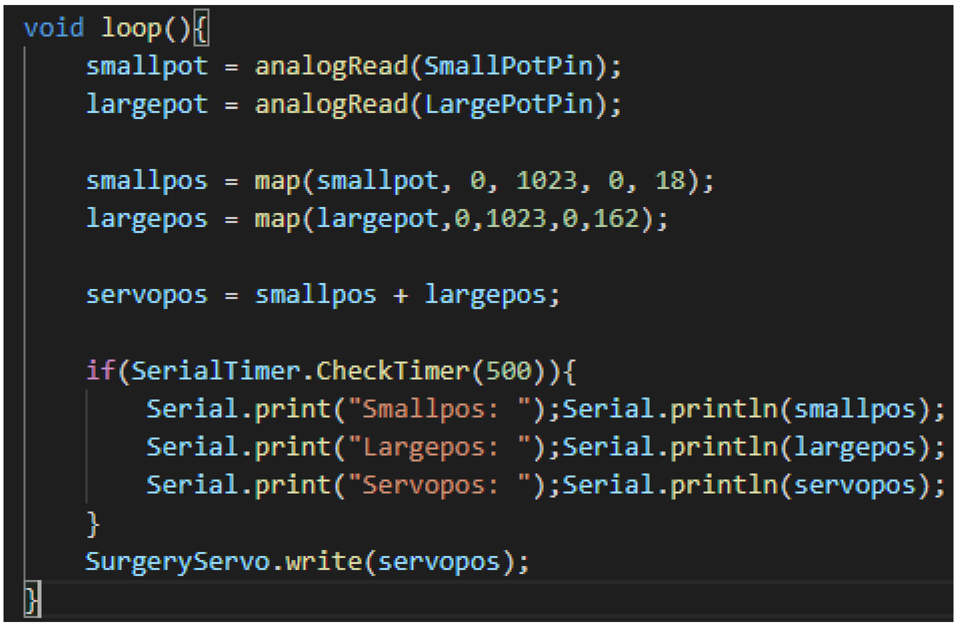
The led cube is a lot different. It is a 4x4x4 cube of LEDs controlled by 20 wire. This is done by connecting every LED to one row and one column. This way there is a sort of coordinate system for the LEDs so not every one of them has to be connected separately, but only allows for 1 specific led to turn on at a time. Trying to turn on more than 1 will also turn on other unintended LEDs. To still allow for specific control of multiple LEDs you *do* only turn one on at a time but in very quick succession. When the LED flicker as fast as the Arduino can handle it is imperceptible by a human that the LEDs are in fact flickering. This is a very basic version of what PWM does, by leaving a longer downtime between the LEDs turning off and on the brightness can be decreased.



(I put the code on the git but it is too long to show in this document)

8.2.2Servo control:

This challenge did not require much code for the basic workings. To give different resolutions per potentiometer but not extend the 180° range I used 2 map functions that together maximally add to 180°. This has the limitation that to get to 180° both potentiometer will have to be set to maximum. I did not think this to be a problem because it does not limit the servo functionally and the limitation is very slight.



```
void loop(){
    smallpot = analogRead(SmallPotPin);
    largepot = analogRead(LargePotPin);

    smallpos = map(smallpot, 0, 1023, 0, 18);
    largepos = map(largepot,0,1023,0,162);

    servopos = smallpos + largepos;

    if(SerialTimer.CheckTimer(500)){
        Serial.print("Smallpos: ");Serial.println(smallpos);
        Serial.print("Largepos: ");Serial.println(largepos);
        Serial.print("Servopos: ");Serial.println(servopos);
    }
    SurgeryServo.write(servopos);
}
```

Using this library the control of the servo was very easy so all I had to do was map the potentiometers to the correct range. I also added a periodic printout of all the position values to give some feedback and help with very precise control.

9.1.2Controlled testing:

I hadn't yet fully completed the first testing program when I started with this challenge so this will be my full completion of both challenges. I did have a slight problem with doing this challenge, Serial communication was the only thing I hadn't done yet but due to some unfortunate timing I only had the last weekend to do it. I had also forgotten to bring my Arduino home to my parents so I sadly had no way to test my code. I did the challenges out of order, I did the previous challenge 9.1.1 after this one as a backup that had a higher chance of the code actually working.

At first I started this challenge with the same idea I had in 2.5.3, connecting all the pins to 1 central one and testing through that one. After some thought however I realised that having one point of failure with the central pin is not a good idea, if the central pin is broken the entire test is unusable. To get rid of this one point of failure I decided to test every pin to the next one in line, this effectively tests every pin twice without adding extra work for the user. From my previous plan I also wanted to add the challenge of only using 1 set of components, or in my case only 2/3 wires and a resistor or potentiometer for digital or Analog testing respectively. It will require more input from the user during testing but in total it probably actually reduces setup time.

The main idea of this challenge is the serial communication between the Arduino and a program on a pc, so that is what I focussed on first. The previous challenge gave the hint of using the symbols # and % to signify the start and end of a command, this eliminates the problem of the Arduino constantly listening to Serial data when it doesn't concert it.(Though for this challenge it probably wouldn't have been a problem anyway) When I started making my code for the Arduino to receive the commands I quickly realised that using strings isn't great because Arduino doesn't really support strings like C# does. Instead it is easier to use a char* variable since this allows for easier manipulation of the serial input to a usable command.

Knowing when to read from the serial connection is luckily very easy using serialEvent() which gets called after loop whenever there is activity on the serial bus. My implementation of serialEvent is a modified version of what is in the Arduino documentation about serialEvent. I added a check for the # identifier as the first character and a bit that removes the identifiers from the command after it is fully read.

```
void serialEvent(){//get the serial input and converts to usable command string
    inputString = "";//reset inputstring
    while(Serial.available()){
        char inChar = Serial.read();
        if(sizeof(inputString) == 1 && inChar != '#') break;//check if first char
is # identifier
        inputString += inChar;
        if (inChar == '%'){
            commandComplete = true;
        }
    }
    if(sizeof(inputString)>0)
    {
        commandString = strtok(inputString,"#%");   //removes # and %
    }
}
```

(I now realise that the stuff I say here about strings and such isn't really true and I ended up using string in my other project anyway).

After the command is read the loop can execute upon it. The loop is quite short but it does do quite a bit. First the strtok function splits the incoming command into tokens split by ':' and ','. The command will be split into the first part which is the actual command and the 2 parameters which are the pins that should be used in the test. The Arduino sends back a response indicating to the computer if it received the correct command, this feedback to the computer is something I'm still unsure of if it actually works because I fear it the pc doesn't actually catch it at the correct time. C# doesn't seem to have an easy way of catching serial data when it gets sent, though the serial read and write do have built in timeouts but I couldn't find if they actually work as I'm expecting(waiting for serial data if the Arduino hasn't gotten around to sending it yet). After sending a response to the pc it actually executes the test using the specified pins in the command and sends the result back as well.

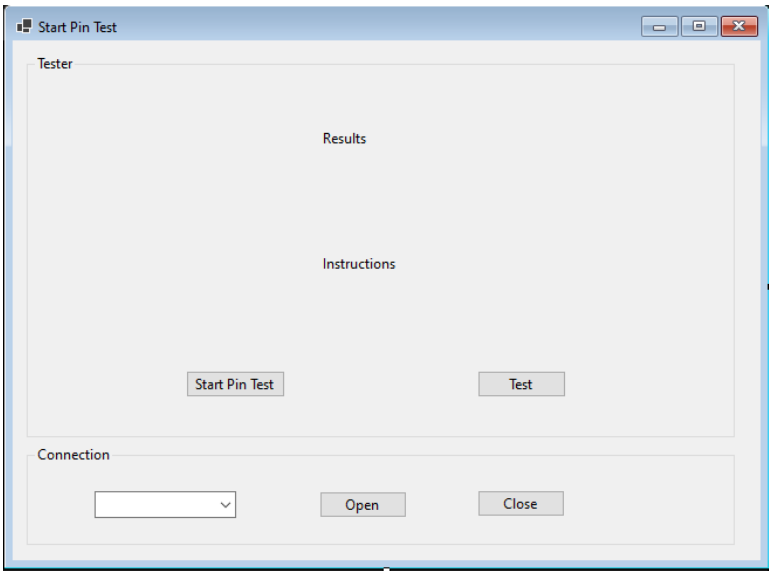
```
void loop(){
    if(commandComplete){//if there is a complete command execute the code
        commandComplete = false;
        commandTokens = strtok(commandString,":,");
        if(strcmp(commandTokens,"Test_Pins:")){//"Test_Pins:0,1"
            Serial.println("#Pin_Test_Starting%");
            testResult =
TestDigitalPin(atoi(strtok(NULL,",")),atoi(strtok(NULL,",")));
            Serial.println(testResult);//example #True,True%
        }
        if(strcmp(commandTokens,"Test_Analog_Pins:")){//"Test_Pins:A0,A1"
            Serial.println("#Analog_Pin_Test_Starting%");
            testResult =
TestAnalogPin(atoi(strtok(NULL,",")),atoi(strtok(NULL,",")));
            Serial.println(testResult);//example #True,True%
        }
        else Serial.println("Unkown_command_received");
    }
```

The digital test is very simple, it tests the testPin to the testerPin. First the input and then the output of the testPin gets tested and both these results get sent back as on char*.

```
char* TestDigitalPin(int testPin,int testerPin){
    char* result = "#"; //result, bool input, bool output
    //testpin INPUT
    pinMode(testPin,INPUT);
    pinMode(testerPin,OUTPUT);
    digitalWrite(testerPin,HIGH);
    strcat(result,digitalRead(testPin) == HIGH ? "True" : "False");
    strcat(result,",");
    //testpin OUTPUT
    pinMode(testPin,OUTPUT);
    pinMode(testerPin,INPUT);
    digitalWrite(testPin,HIGH);
    strcat(result,digitalRead(testerPin) == HIGH ? "True%" : "False%");
    return result;
}
```

There is also a function for the analog pins which is a bit more compacted. The output test of these 2 is exactly the same but the analog input is a bit harder because it needs the user to turn a potentiometer. I made my test by first instructing the user to set the potentiometer to its maximum value which I hope reads as below 16, again I sadly didn't get to test, as a minimum read value. The user should then turn the potentiometer to its lowest value, test will then pass read checkpoints that increment by 16 every pass. Once all the checkpoints have been passed the passedTest variable will have stayed as true which will be the final result. There is a problem with knowing when this test fails however, just an input isn't enough to pass the test because the pin should be able to read a range of inputs. I added a timeout delay to stop the test if it hasn't passed before the delay is reached. After writing this I realised I could have added an analog input test to this as well to at least know if the pin is reading some input.

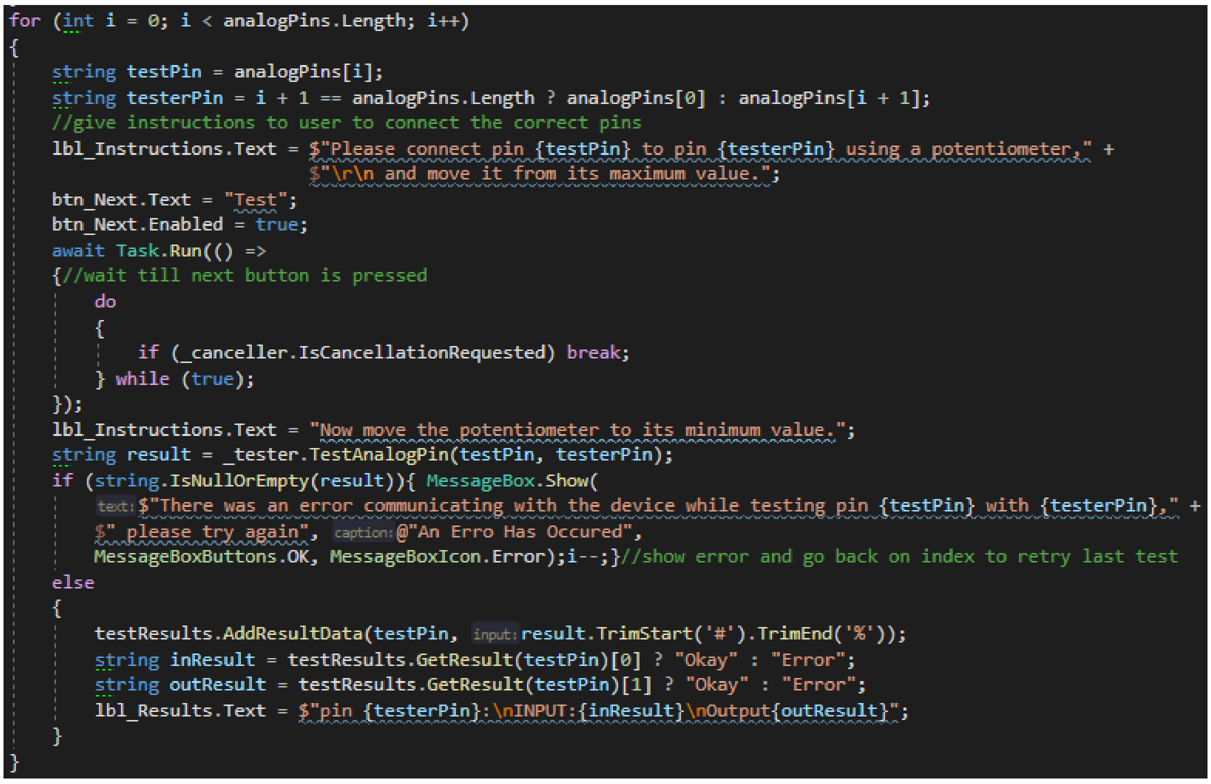
```
char* TestAnalogPin(int testPin,int testerPin){
     char* result = "#";
     timeoutTimer.Delay(120000);//start timoutTimer
     bool passedTest = true;
    pinMode(testPin,INPUT);
    pinMode(testerPin,OUTPUT);
    while(analogRead(testPin > 50)){//first set potentiometer to 0
       if(timeoutTimer.Delay(120000)){//timout of 2min if something is not working
               passedTest = false;
               break;
            };
        };
    for (size_t i = 16; i < 1024; i+16)
    {
      while(analogRead(testPin) <= i && passedTest){//skips this if test has
failed and so quickly ending for loop
        if(timeoutTimer.Delay(120000)){//timout of 2min if something is not
working
             passedTest = false;
             break;
          };
      }//only passes if it is able to increase from 0-1024 in increments of 16
    }
    timeoutTimer.ResetDelay();
    strcat(result,passedTest? "True":"False");//concatonate True if "passedTest"
is still true, otherwise False
    //testpin digital OUTPUT
    pinMode(testPin,OUTPUT);
    pinMode(testerPin,INPUT);
    digitalWrite(testPin,HIGH);
    strcat(result,digitalRead(testerPin) == HIGH ? "True%" : "False%");
    return result;
}
```



The C# part of this project is pretty straight forward but was quite tricky to implement, I still don't know if it actually works as I intend. I kept the interface simple with just a button to start the test and one to go to test the pin and go to the next one. I did add a dropdown to specify to what com port the Arduino is connected because I saw it in a tutorial I used and thought it interesting to add.

In short the program should open the connection to the Arduino, then iterate over all the pins while waiting in between for the user to change the setup or move the potentiometer and then save and show the result. This initially seemed like a large but not very difficult task, this turned out to definitely not be the case. Getting the program to wait while not freezing it entirely and keeping it synchronized with the Arduino were 2 problems I had hard time finding solutions to, the second part I never found a good answer to.

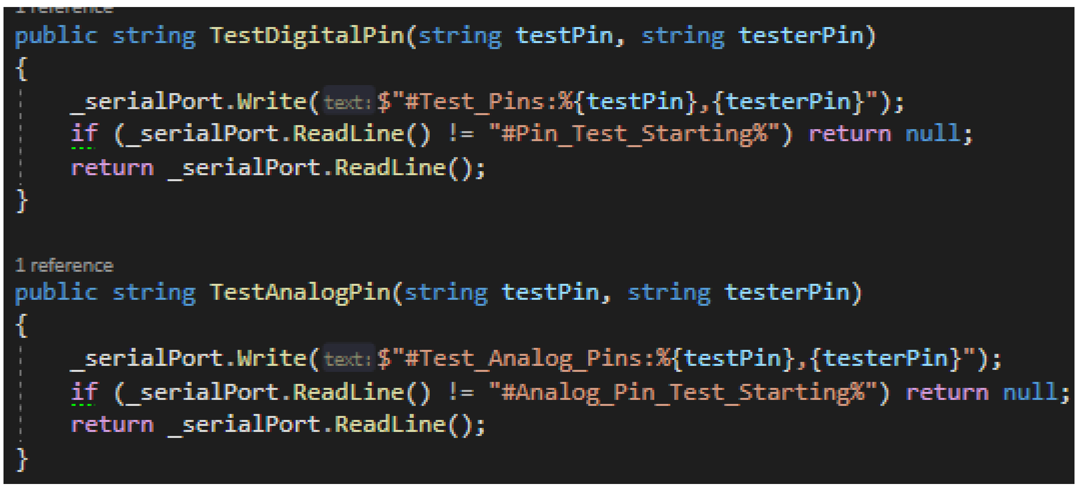
The code is quite long so I won't touch on every part but I will explain it in broad terms. I made 2 classes to help with sending the commands to the Arduino and saving the result. The TestResults class saves the results of the tests in Boolean arrays and is able to return the result of a specific pin or all the test results in a formatted string. I also made a Tester class that handles all the serial communication. The Form class is where all the logic for the test happens.



```
for (int i = 0; i < analogPins.Length; i++)
{
    string testPin = analogPins[i];
    string testerPin = i + 1 == analogPins.Length ? analogPins[0] : analogPins[i + 1];
    //give instructions to user to connect the correct pins
    lbl_Instructions.Text = $"Please connect pin {testPin} to pin {testerPin} using a potentiometer," +
                            $"\r\n and move it from its maximum value.";
    btn_Next.Text = "Test";
    btn_Next.Enabled = true;
    await Task.Run(() =>
    {//wait till next button is pressed
        do
        {
            if (_canceller.IsCancellationRequested) break;
        } while (true);
    });
    lbl_Instructions.Text = "Now move the potentiometer to its minimum value.";
    string result = _tester.TestAnalogPin(testPin, testerPin);
    if (string.IsNullOrEmpty(result)){ MessageBox.Show(
        text:$"There was an error communicating with the device while testing pin {testPin} with {testerPin}," +
        $" please try again", caption:@"An Erro Has Occured",
        MessageBoxButtons.OK, MessageBoxIcon.Error);i--;}//show error and go back on index to retry last test
    else
    {
        testResults.AddResultData(testPin, input:result.TrimStart('#').TrimEnd('%'));
        string inResult = testResults.GetResult(testPin)[0] ? "Okay" : "Error";
        string outResult = testResults.GetResult(testPin)[1] ? "Okay" : "Error";
        lbl_Results.Text = $"pin {testerPin}:\nINPUT:{inResult}\nOutput{outResult}";
    }
}
```

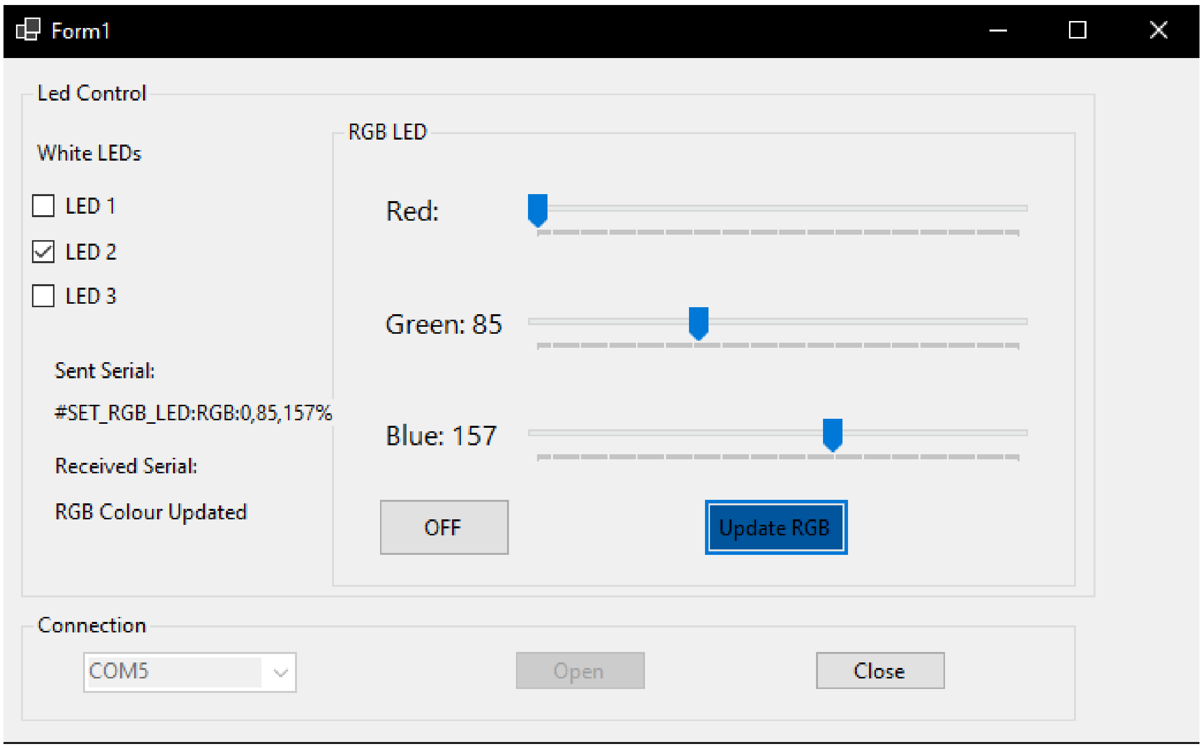
The above code is the analog pin part of the test, it is almost exactly the same as the digital part but of course with the different pins and with different instructions. The pins get iterated over, the pin to be tested and the next one in line get selected to use for the test. If it is at the end of the pins it will use the first pin instead. The user then gets instructions on how to set up the test and the program waits till the test button is pressed. The program will then perform the test and in this case show more instructions. If the test fails an error message will be shown and the program will retry. If the test is successful the results will be saved and shown to the user.

The Tester class has 2 main functions that send the commands to the Arduino. The function first constructs the command and writes it to the serial bus. After the command is sent it immediately waits for a response and fails is it doesn't to try again. If the correct acknowledge response is received it will read for the result and send that back.



```
public string TestDigitalPin(string testPin, string testerPin)
{
    _serialPort.Write(text:$"#Test_Pins:%{testPin},{testerPin}");
    if (_serialPort.ReadLine() != "#Pin_Test_Starting%") return null;
    return _serialPort.ReadLine();
}

1 reference
public string TestAnalogPin(string testPin, string testerPin)
{
    _serialPort.Write(text:$"#Test_Analog_Pins:%{testPin},{testerPin}");
    if (_serialPort.ReadLine() != "#Analog_Pin_Test_Starting%") return null;
    return _serialPort.ReadLine();
}
```

9.1.1Remote control:



- [ ] LED 1
- [✓] LED 2
- [ ] LED 3

Red:

Green: 85

Blue: 157

Sent Serial:

#SET_RGB_LED:RGB:0,85,157%

Received Serial:

RGB Colour Updated

COM5

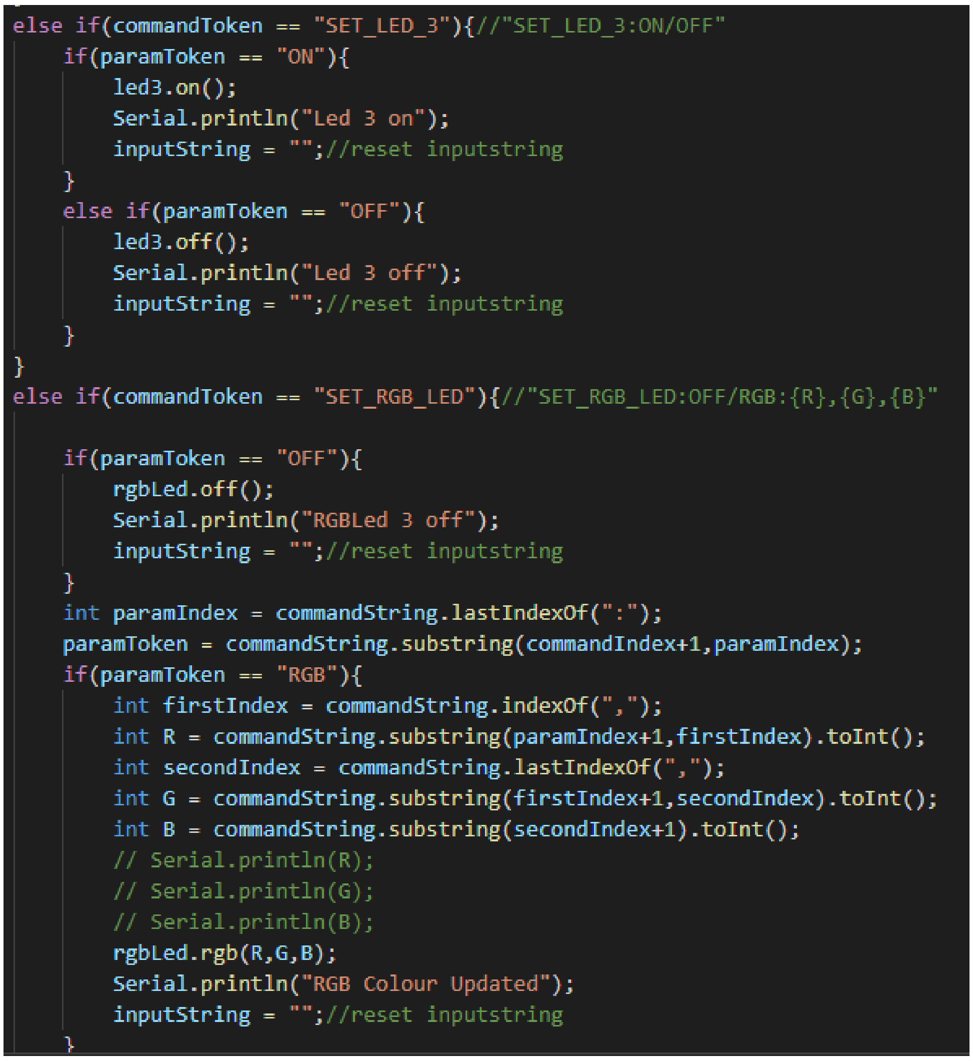
I wanted to kind of combine this challenge with the next one because I sadly didn't have enough time to fully test challenge 9.1.2 so added the interface to this challenge too.

The challenge recommends to use the symbols # and % at the start and end of a command respectively and asks what the use of this is. Using an identifier at the start and end of a command helps with determining when a command starts and when it ends. Having these identifiers helps to prevent the Arduino executing something it shouldn't and it trying to execute serial data that aren't actually commands.

The part that took the longest in this challenge was actually disassembling the commands coming in through serial into usable commands. Like I showed in the previous challenge I first tried to do it with char * variables because these seemed to have more useful functions related to the task. Eventually I figured out though that using char * in if statement to check what the command is isn't really possible so I had to go back to the drawing board. Eventually I just went back to using strings which was a bit more work getting the substring indexes to line up but it did eventually work.

Once I got the proper command outputs for the loop if statements the rest was quite straight forward, it was just a matter of setting the LEDs according to what the command said.

In the loop I wanted to use a switch case to determine the proper command but they sadly don’t support the use of strings so I had to resort to a lot of if statements.



```
else if(commandToken == "SET_LED_3"){//"SET_LED_3:ON/OFF"
    if(paramToken == "ON"){
        led3.on();
        Serial.println("Led 3 on");
        inputString = "";//reset inputstring
    }
    else if(paramToken == "OFF"){
        led3.off();
        Serial.println("Led 3 off");
        inputString = "";//reset inputstring
    }
}
else if(commandToken == "SET_RGB_LED"){//"SET_RGB_LED:OFF/RGB:{R},{G},{B}"

    if(paramToken == "OFF"){
        rgbLed.off();
        Serial.println("RGBLed 3 off");
        inputString = "";//reset inputstring
    }
    int paramIndex = commandString.lastIndexOf(":");
    paramToken = commandString.substring(commandIndex+1,paramIndex);
    if(paramToken == "RGB"){
        int firstIndex = commandString.indexOf(",");
        int R = commandString.substring(paramIndex+1,firstIndex).toInt();
        int secondIndex = commandString.lastIndexOf(",");
        int G = commandString.substring(firstIndex+1,secondIndex).toInt();
        int B = commandString.substring(secondIndex+1).toInt();
        // Serial.println(R);
        // Serial.println(G);
        // Serial.println(B);
        rgbLed.rgb(R,G,B);
        Serial.println("RGB Colour Updated");
        inputString = "";//reset inputstring
    }
```

This is the last part of the if statements, for the 3rd led and the RGB led. The normal led just switches either on when “ON” is sent and off when “OFF” is sent as you would expect. For the RGB led it was a bit more difficult because I had to isolate all the colour values.

The code on the interface side was very straight forward, make a button or checkbox send a command over the serial port when it is pressed. Because the interface was quite simple I decided to add some extra things like a preview for the RGB colour and a dropdown for connecting to the correct com port.