Technology Verdieping

Major switch preperation

mees van leusden

2022

Inhoud

[2.2:Board Markings 2](#_Toc92762843)

[2.5.1/2:Led Fade 3](#_Toc92762844)

[2.5.3:Board Testing 3](#_Toc92762845)

[3.2.1:Blinking 4](#_Toc92762846)

[3.2.2:Debounce 5](#_Toc92762847)

[3.2.3:debounce function 5](#_Toc92762848)

[Extra Classes: 6](#_Toc92762849)

[3.3.1:states 7](#_Toc92762850)

[4.1.1:I/O techniques 8](#_Toc92762851)

[5.2.1/2:LDR 9](#_Toc92762852)

[5.2.3:Scaling 10](#_Toc92762853)

[5.2.4: Running average 10](#_Toc92762854)

[5.2.5: Hysteresis 11](#_Toc92762855)

[6.1.1: DHT library 11](#_Toc92762856)

[6.1.2: Ping sensor library 12](#_Toc92762857)

[7.1.2: C# Traffic light 12](#_Toc92762858)

[7.1.3: Bank account 13](#_Toc92762859)

[7.1.4-7.1.7: 15](#_Toc92762860)

[7.1.8: 15](#_Toc92762861)

[8.2.1 PWM LED: 16](#_Toc92762862)

[8.2.2Servo control: 17](#_Toc92762863)

[9.1.1: 18](#_Toc92762864)

[9.1.2: 18](#_Toc92762865)

[9.1.3: 18](#_Toc92762866)

[9.1.4: 18](#_Toc92762867)

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

2.2:Board MarkingsAfbeelding met tekst, elektronica, circuit

Automatisch gegenereerde beschrijving

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.

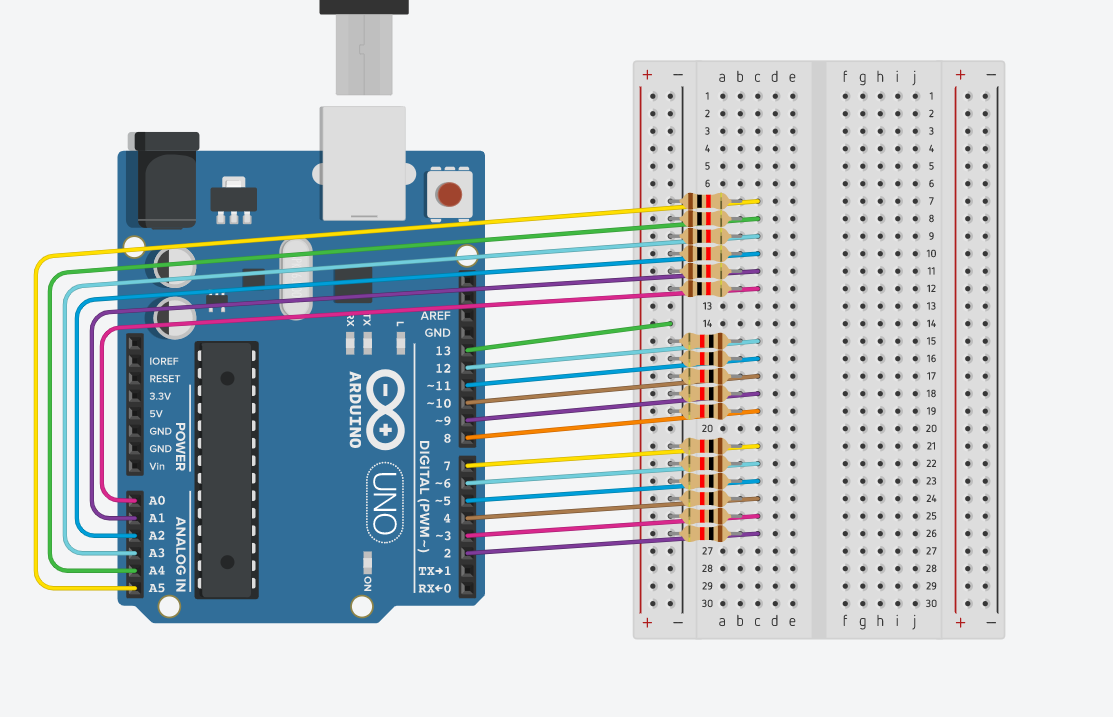
### 2.5.1/2:Led Fade

Afbeelding met tekst, elektronica

Automatisch gegenereerde beschrijving

### 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](https://github.com/rlogiacco/CircularBuffer?utm_source=platformio&utm_medium=piohome). 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.

Afbeelding met tekst, scherm

Automatisch gegenereerde beschrijving Afbeelding met tekst

Automatisch gegenereerde beschrijving Afbeelding met tekst

Automatisch gegenereerde beschrijving

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](https://www.arduino.cc/en/Tutorial/Foundations/DigitalPins) 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](https://www.arduino.cc/en/Tutorial/Foundations/AnalogInputPins) 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](https://learn.sparkfun.com/tutorials/serial-communication/all) is an explanation on the full serial communication method. [This](https://www.arduino.cc/en/Tutorial/Foundations/ShiftIn) shows how to convert from parallel to serial shifting-in and [this](https://www.arduino.cc/en/Tutorial/Foundations/ShiftOut) 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](https://www.youtube.com/watch?v=_fgWQ3TIhyE), [What is I2C, Basics for Beginners](https://www.youtube.com/watch?v=jFtr0Ha5f-c), [How I2C Communication Works and How To Use It with Arduino](https://www.youtube.com/watch?v=6IAkYpmA1DQ)

-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](https://www.youtube.com/watch?v=ba0SQwjTQfw), explanation and tutorial on [Arduino website](https://www.arduino.cc/en/Tutorial/SPIEEPROM) plus and explanation of the [software library](https://www.arduino.cc/en/reference/SPI).

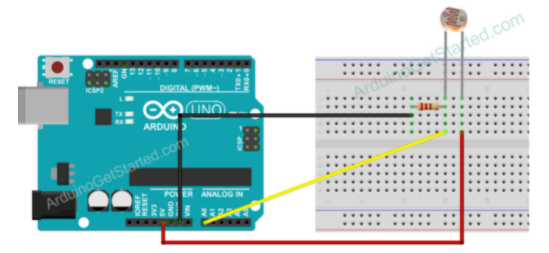
-Used in the same type of components I2C is but is a lot faster while needing more cables for communicating with multiple devices.

Articles comparing UART,SPI,I2C:

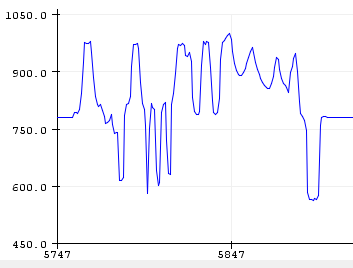
* <https://www.seeedstudio.com/blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses/>
* <https://www.makeuseof.com/how-uart-spi-i2c-serial-communications-work/>

### 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.

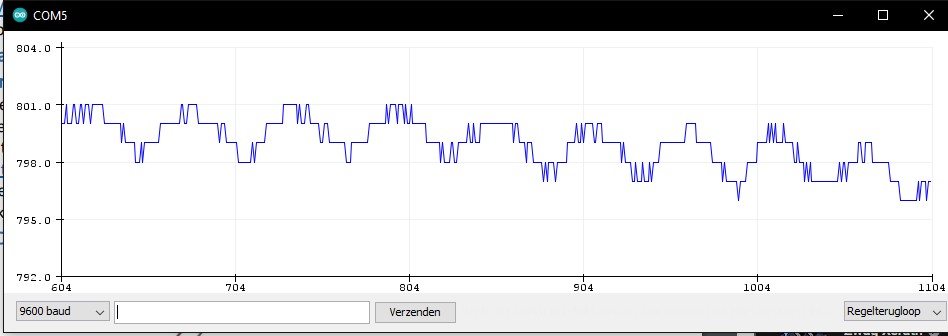
 Afbeelding met tekst

Automatisch gegenereerde beschrijving



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](https://forum.arduino.cc/t/ldr-which-resistor/546658/2) 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

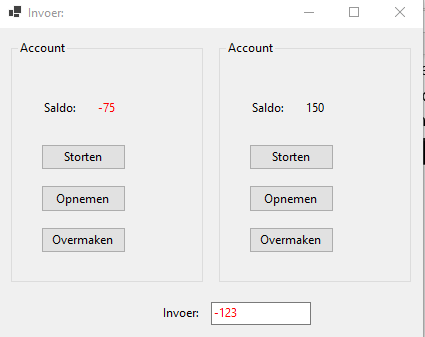
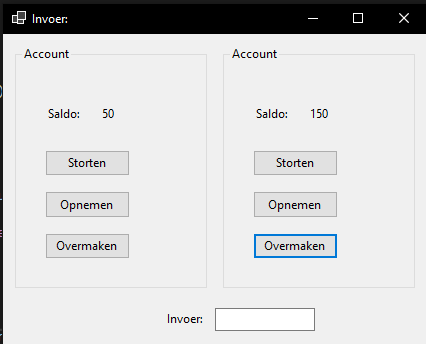
With this challenge I had the same problem as the previous one as I don’t have this sensor on hand either, though I will again do as much as I can without it. I could not find anything about the sensor having accuracy variance so using the library to do the data processing I would expect the distance 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>

### 7.1.2: C# Traffic light

### 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.

Afbeelding met tekst

Automatisch gegenereerde beschrijvingAfbeelding met tekst

Automatisch gegenereerde beschrijving

Afbeelding met tekst

Automatisch gegenereerde beschrijving

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.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

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.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

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.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

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.)



### 7.1.4-7.1.7:

I already did these earlier because I started with making objects earlier.

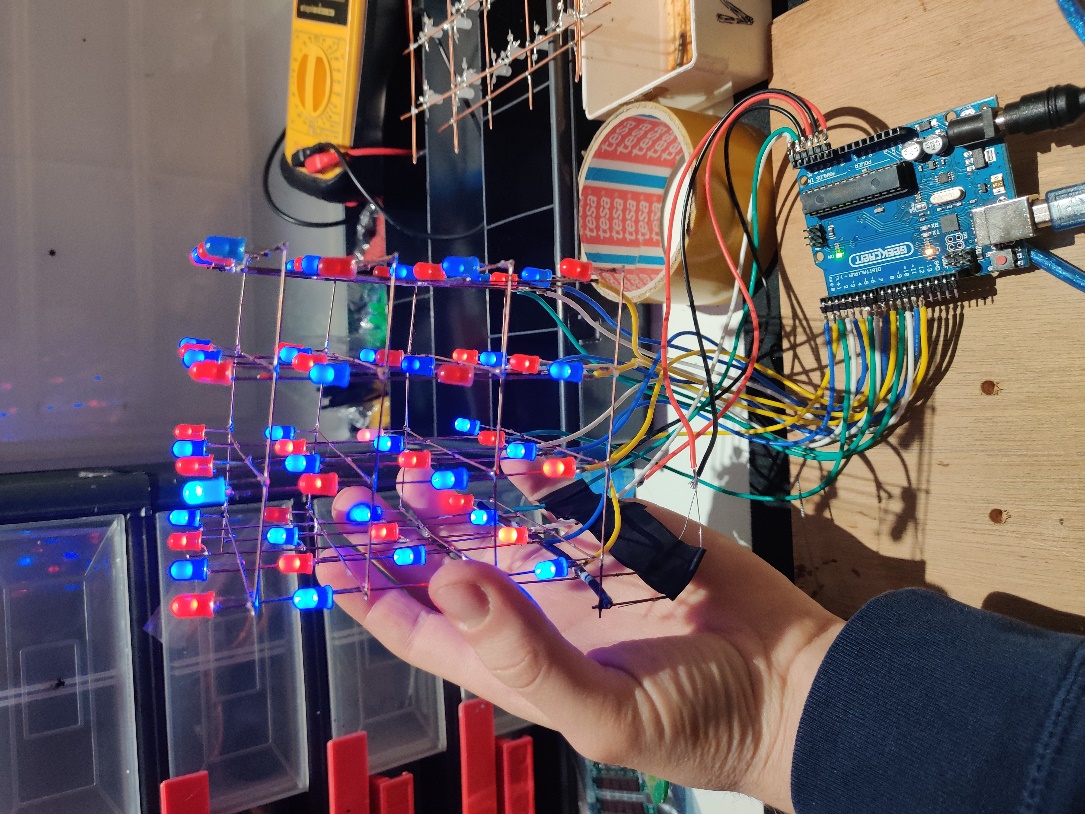
### 7.1.8:

### 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.

Afbeelding met tekst

Automatisch gegenereerde beschrijvingThe 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.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Using [this](https://www.arduino.cc/reference/en/libraries/servo/?utm_source=platformio&utm_medium=piohome) 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.

### 9.1.1:

### 9.1.2:

### 9.1.3:

### 9.1.4: