Arduino Lab 2

Knut Ola Nøsen

***Abstract*—The project explores how to control DC motors and servos. For the DC motor we are using the L293D chip, and for the servos we use the builtin Servo.h library. The reason for using the L293D chip is that motors not only require a lot more power than the Arduino can provide, but also more fine control than a relay. In this lab we will implement several ways of controlling motor speed, motor acceleration and servo position.**

***Index Terms*—Arduino, L293D, Servo, DC Motor, RAMP, PWM**

I. Theory

WE start with the motor. The L293D chip has two input pins (In1 and In2) for specifying directions, and one input pin (Enable) for setting speed with a PWM signal. The H-bridge derives its name from the H-shaped design, with the vertical lines representing a MOSFET, and the horizontal lines representing a motor. The mosfets are enabled/disabled in diagonal pairs, in order to switch the direction of the current. The directional MOSFETs are controlled by 2 transistors each. The transistors are connected inseries, where one transistor enables/disables the mosfet, and the other forwars the PWM signal. The following code shows how the L293D chip is used:

```

void updateMotor(
   bool direction,
   int speed,
   int in1Pin,
   int in2Pin,
   int enablePin
)
{
   digitalWrite(
      in1Pin,
      direction ? HIGH : LOW
   );
   digitalWrite(
      in2Pin,
      direction ? LOW : HIGH
   );
   analogWrite(enablePin, speed);
}
```

While this code clearly displays how the L293D chip is used, it still lacks some safeguards that should be in place on a DC-Motor. For example, brushed DC-Motors have a minimum voltage, before which they do not have suficient power to start moving. We will implement this safeguard later.

II. Methods

A. Hardware

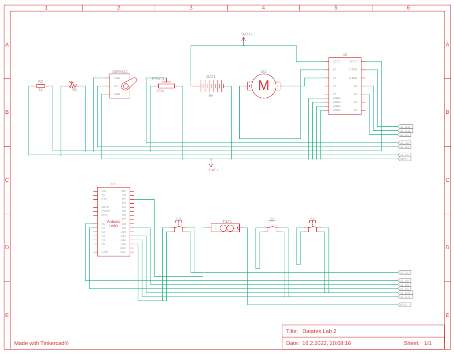

Fig. 1. Wiring Diagram

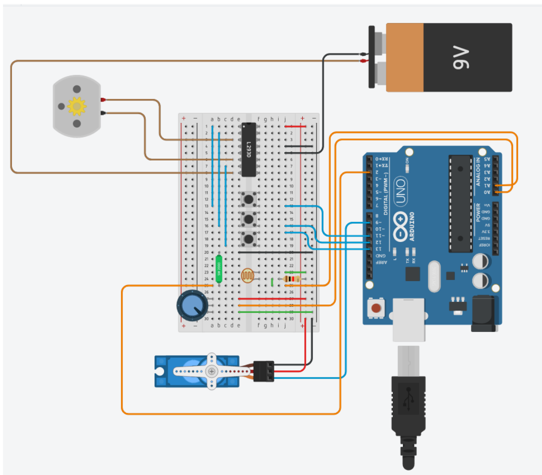

Fig. 2. Tinkercad

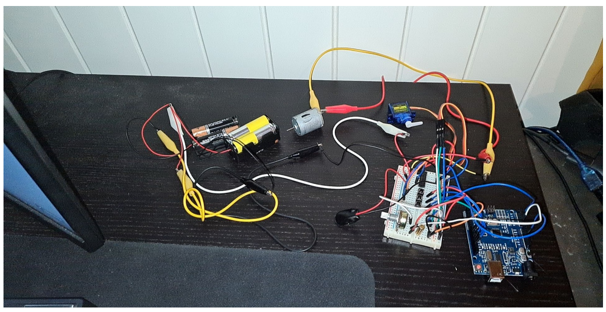

Fig. 3. Lab2 Circuit

B. Software

Lets start by creating some configs to keep our code free of ”magic constants”. By keeping these configs in separate files, we avoid cluttering our logic in the main.cpp file.

Listing 1. Range.h

```
#ifndef Range_h
#define Range_h

struct Range
{
    int minValue;
    int maxValue;
    Range(int minValue, int maxValue) :
        minValue(minValue),
        maxValue(maxValue)
    {
    }
};

#endif
```

Listing 2. ServoConfig.h

```
#ifndef ServoConfig_h
#define ServoConfig_h

#include <Range.h>

struct ServoConfig
{
    const int servoPin;
    const Range angleLimits;
};

#endif
```

Listing 3. MotorConfig.h

```
#ifndef MotorConfig_h
#define MotorConfig_h

#include <Range.h>

struct MotorConfig
{
    const int enablePin;
    const int in1Pin;
    const int in2Pin;
    const Range speedLimits;
};

#endif
```

Listing 4. ApplicationConfig.h

```
#ifndef ApplicationConfig_h
#define ApplicationConfig_h

#include <ServoConfig.h>
#include <MotorConfig.h>

struct ApplicationConfig
{
    const ServoConfig servoConfig = {
        .servoPin = 9,
        .angleLimits = Range(0, 180),
    };
    const MotorConfig motorConfig = {
        .enablePin = 11,
        .in1Pin = 12,
        .in2Pin = 13,
        // The motor is incapable of
            starting at speeds lower than
            140.
        .speedLimits = Range(140, 255),
    };
    const int potmeterPin = A0;
    const int photoresistorPin = A1;
    const int emergencyStopPin = 2;
    const int baudRate = 9600;
} const appConfig;

#endif
```

This gives us a global const appConfig when we include the ApplicationConfig.h file in our main sketch.

We also define a class for the Photoresistor so we can get rid of those pesky globas.

Listing 5. Photoresistor.h

```
#ifndef Photoresistor_h
#define Photoresistor_h

#include <Arduino.h>
#include <Range.h>

class Photoresistor
{
   // Start with oposite values
   Range limits = Range(1023, 0);
   const int pin;

public:
   Photoresistor(const int pin) : pin(pin)
   {
   }

   int mapWithinLimits(const int value)
   {
      return map(value, limits.minValue, limits.maxValue, 0, 1023);
   }
   int read()
   {
      const int value = analogRead(pin);
      // Always update the limits
      if (value < limits.minValue)
      {
         limits.minValue = value;
      }
      if (value > limits.maxValue)
      {
         limits.maxValue = value;
      }
      return mapWithinLimits(value);
   }

   void setup()
   {
      pinMode(pin, INPUT);
   }
};

#endif
```

Now that all our external files are set up, we can start looking at the main.cpp file.

Listing 6. main.cpp - imports

```
#include <Arduino.h>
#include <Ramp.h>
#include <Servo.h>
#include <Range.h>
#include <ApplicationConfig.h>
#include <Photoresistor.h>
```

Listing 7. main.cpp - CLAMPING

```
const int clampAngle(const int value, const Range range)
{
   return min(max(value, range.minValue), range.maxValue);
}

const int clampSpeed(const int value, const Range range)
{
   if (value < range.minValue)
   {
      return 0;
   }
   if (value > range.maxValue)
   {
      return range.maxValue;
   }
   return value;
}
```

We add some clamping functions to restrict servo position and motor power to set ranges. This way, we can avoid pushing our hardware beyone what it can do, without risking human error while programming.

Listing 8. main.cpp - Servo Control

```
void setServoPosition(const ServoConfig servoConfig, Servo servo, const int
    position)
{
   const int targetPosition = clampAngle(position, servoConfig.angleLimits);
   servo.write(targetPosition);
}
```

Here we pipe our servo positioning through a guarded function that keeps us within the hardware limits, similar to how we will with our DC motor.

Listing 9. main.cpp - Motor Control

```cpp
enum class MotorDirection
{
   CLOCKWISE,
   COUNTERCLOCKWISE,
};

void stopMotor(const MotorConfig motorConfig)
{
   digitalWrite(motorConfig.in1Pin, LOW);
   digitalWrite(motorConfig.in2Pin, LOW);
}

void setMotorDirection(const MotorConfig motorConfig, const MotorDirection
    direction)
{
   const int in1Pin = motorConfig.in1Pin;
   const int in2Pin = motorConfig.in2Pin;

   if (direction == MotorDirection::CLOCKWISE)
   {
      digitalWrite(in1Pin, HIGH);
      digitalWrite(in2Pin, LOW);
   }
   else
   {
      digitalWrite(in1Pin, LOW);
      digitalWrite(in2Pin, HIGH);
   }
}

/**
   @param speed 0 to 255
*/
void setMotorSpeed(const MotorConfig motorConfig, const int speed, const
    MotorDirection direction)
{
   const int limitedSpeed = clampSpeed(abs(speed), motorConfig.speedLimits);
   setMotorDirection(motorConfig, direction);
   analogWrite(motorConfig.enablePin, limitedSpeed);
}

/**
   @param speed -255 to 255
*/
void setMotorSpeed(const MotorConfig motorConfig, const int speed)
{
   setMotorSpeed(
      motorConfig,
      abs(speed),
      speed > 0 ? MotorDirection::CLOCKWISE : MotorDirection::COUNTERCLOCKWISE);
}

void setupMotor(const MotorConfig motorConfig)
{
   pinMode(motorConfig.enablePin, OUTPUT);
   pinMode(motorConfig.in1Pin, OUTPUT);
   pinMode(motorConfig.in2Pin, OUTPUT);
}
```

We create an enum for motor direction to be crystal clear about what is going on. We also add an overload to setMotorSpeed that automatically sets direction based on a range that can accept negative speed. This is useful for the centered potmeter controls, while it would pose problems during our ramp functions.

Listing 10. main.cpp - Emergency Stop

```
// Exercise 5
void emergencyStopInterrupt()
{
   while (digitalRead(appConfig.emergencyStopPin) == HIGH)
   {
      stopMotor(appConfig.motorConfig);
   }
}

// Exercise 5
void setupEmergencyStop()
{
   const int pin = appConfig.emergencyStopPin;
   pinMode(pin, INPUT_PULLUP);
   // Emergency stops should be normally closed (NC), so that cutting the wire
   // Causes the motor to stop.
   // Because the pinMode is PULLUP, that means that the pin should be LOW when
   // the emergancy stop is not activated.
   attachInterrupt(digitalPinToInterrupt(pin), emergencyStopInterrupt, RISING);
}
```

For the emergency stop we continously keep killing the motors power by setting its direction to none. This way, we are absolutely sure that nothing moves until the Emergency Stop is released.

Listing 11. main.cpp - Setup

```
ramp servoRamp;
Servo servo;
Photoresistor photoresistor(appConfig.photoresistorPin);

void setup()
{
   Serial.begin(appConfig.baudRate);
   setupEmergencyStop();
   photoresistor.setup();
   servo.attach(appConfig.servoConfig.servoPin);
   setupMotor(appConfig.motorConfig);
   pinMode(appConfig.potmeterPin, INPUT);
}
```

Listing 12. main.cpp - Center Potmeter

```
const int centerAnalogInput(const int value)
{
   return map(value, 0, 1023, -255, 255);
}
```

In order to make our code more consise, we create a helper function for centering a potmeter reading.

Listing 13. main.cpp - Exercise 3

```

/*
   This does not guard the motor from directions switching during full speed
   Changing speed from 255 CW to 255 CCW could cause damage to the motor as
   directional change is instant
*/
void rampMotorTo(const MotorConfig motorConfig, const int rampTime, const int
    speed, const MotorDirection direction)
{
   servoRamp.go(speed, rampTime);
   while (servoRamp.isRunning())
   {
      servoRamp.update();
      setMotorSpeed(appConfig.motorConfig, servoRamp.getValue(), direction);
   }
}

// Exercise 3
void fadeLoop()
{
   const int rampTime = 2000;
   MotorConfig motorConfig = appConfig.motorConfig;
   rampMotorTo(motorConfig, rampTime, 255, MotorDirection::CLOCKWISE);
   rampMotorTo(motorConfig, rampTime, 0, MotorDirection::CLOCKWISE);
   rampMotorTo(motorConfig, rampTime, 255, MotorDirection::COUNTERCLOCKWISE);
   rampMotorTo(motorConfig, rampTime, 0, MotorDirection::COUNTERCLOCKWISE);
}
```

Because the RAMP.h library can not (to my knowledge) handle negative ranges, we must handle this ourselves by explicitly setting the direction, moving from 0 to 255, back to 0 and then to -255 etc.

Listing 14. main.cpp - Exercise 4

```
// Exercise 4
void centeredPotmeterMotorControlLoop()
{
   const int potmeterValue = analogRead(appConfig.potmeterPin);
   setMotorSpeed(appConfig.motorConfig, centerAnalogInput(potmeterValue));
}
```

Because we created some robust reusable code earlier, exercise 4 is now a lot easier.

Listing 15. main.cpp - Exercise 6

```
// Exercise 6
void potmeterFadeLoop()
{
   // 600ms/60deg
   // (180/60) * 600 = 1800ms
   const int rampTime = 1800;
   servoRamp.go(180, rampTime);
   while (servoRamp.isRunning())
   {
      servoRamp.update();
      setServoPosition(appConfig.servoConfig, servo, servoRamp.getValue());
   }
   servoRamp.go(0, rampTime);
   while (servoRamp.isRunning())
   {
      servoRamp.update();
      setServoPosition(appConfig.servoConfig, servo, servoRamp.getValue());
   }
}
```

Listing 16. main.cpp - Exercise 7
```
// Exercise 7
void directPotmeterServoControlLoop()
{
    const int potmeterValue = analogRead(appConfig.potmeterPin);
    const int servoPosition = map(potmeterValue, 0, 1023, 0, 180);
    setServoPosition(appConfig.servoConfig, servo, servoPosition);
}
```

Listing 17. main.cpp - Exercise 8
```
// Exercise 8
void speedIndicator()
{
    const int potmeterValue = analogRead(appConfig.potmeterPin);
    const int motorSpeed = centerAnalogInput(potmeterValue);
    const int servoPosition = map(abs(motorSpeed), 0, 255, 0, 180);
    setServoPosition(appConfig.servoConfig, servo, servoPosition);
    setMotorSpeed(appConfig.motorConfig, motorSpeed);
}
```

Listing 18. main.cpp - Exercise 9
```
// Exercise 9
void photoresistorSpeedControl()
{
    const int photoresistorValue = photoresistor.read();
    const int motorSpeed = centerAnalogInput(photoresistorValue);
    const int servoPosition = map(abs(motorSpeed), 0, 255, 0, 180);
    setServoPosition(appConfig.servoConfig, servo, servoPosition);
    setMotorSpeed(appConfig.motorConfig, motorSpeed);
}
```

Listing 19. main.cpp - loop
```
void loop()
{
    // Uncomment code to run an exercise
    // NB! These procedures are not designed for concurrent runs

    // Exercise 3
    // fadeLoop();

    // Exercise 4
    // centeredPotmeterMotorControlLoop();

    // Exercise 6
    // potmeterFadeLoop();

    // Exercise 7
    // directPotmeterServoControlLoop();

    // Exercise 8
    // speedIndicator();

    // Exercise 9
    // photoresistorSpeedControl();
}
```

III. Discussion

The current code works excellently, however it does have a weakness. While the nested nature of this code makes for very few instances of ”state” and globals, it does prevent us from running continous updates on anything while a piece of code is executing. In a larger project, i believe it would be beneficial to avoid while and for-loops, in favour of a more flat architecture with state machines. That way, the project remains scalable, and we can easily add continous checks without risking spagheti code and human errors due to forgetting to call an updater during a special loop.

IV. Large Images

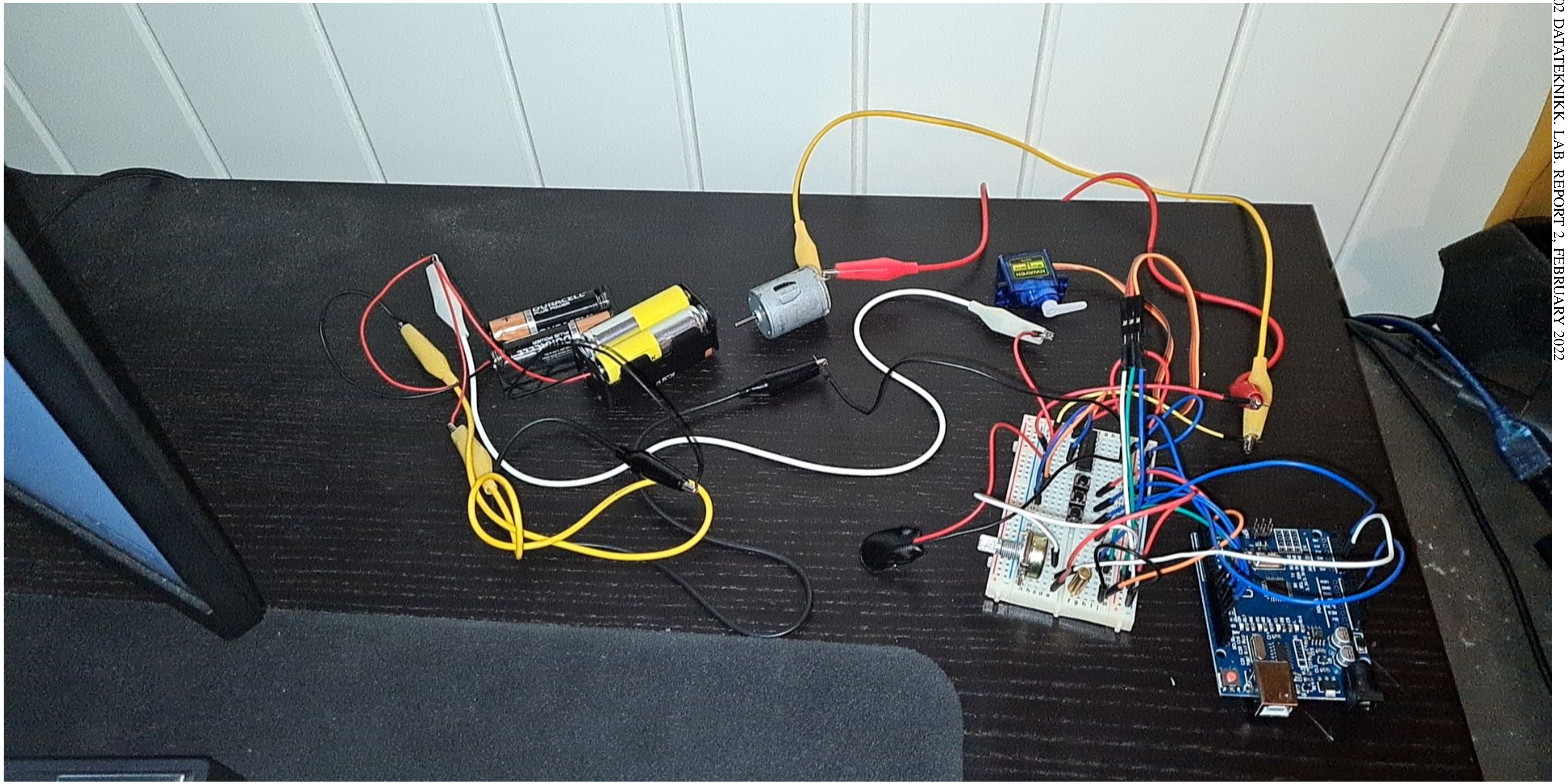

Fig. 4. Large Lab2 Circuit

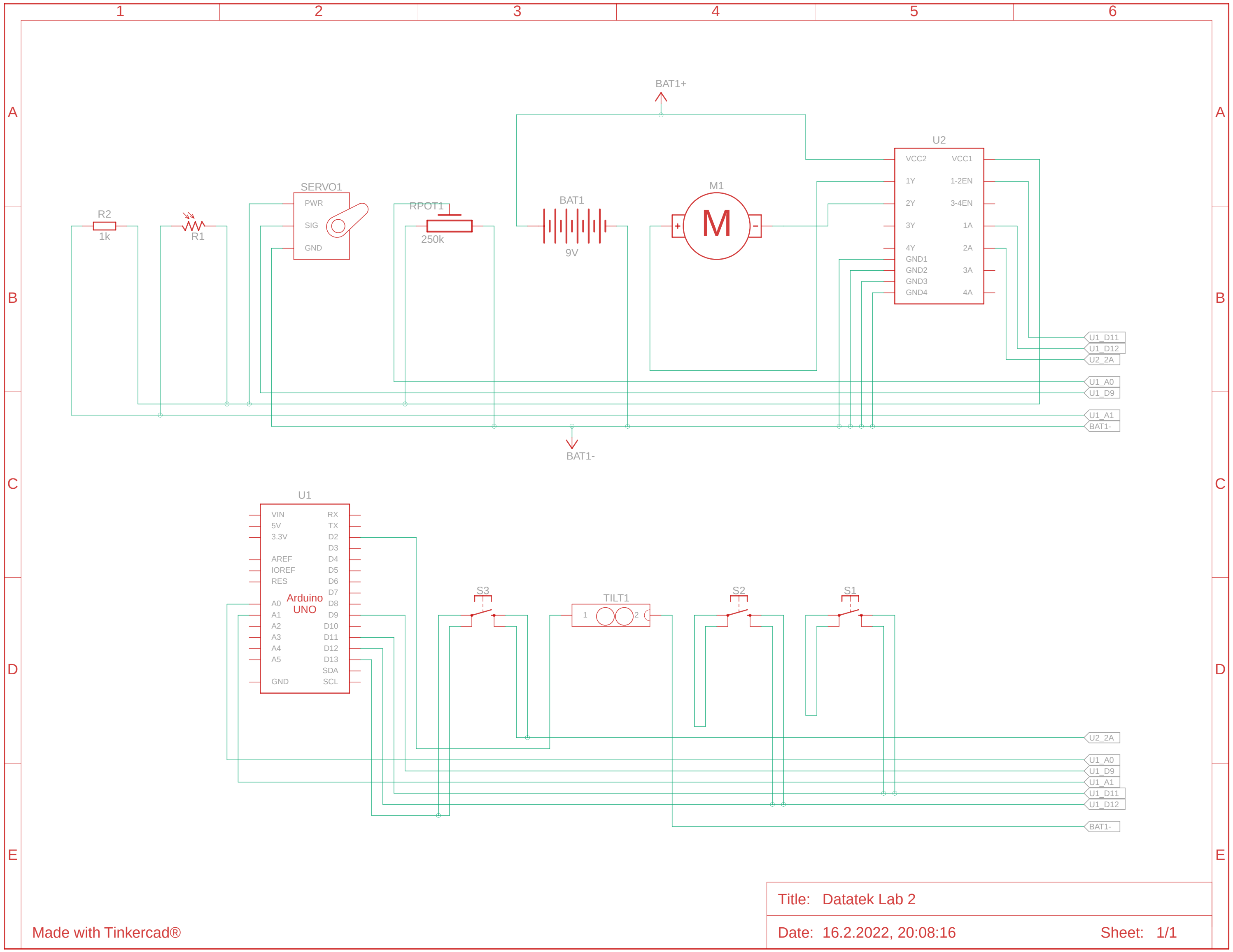

Fig. 5. Large Wiring Diagram

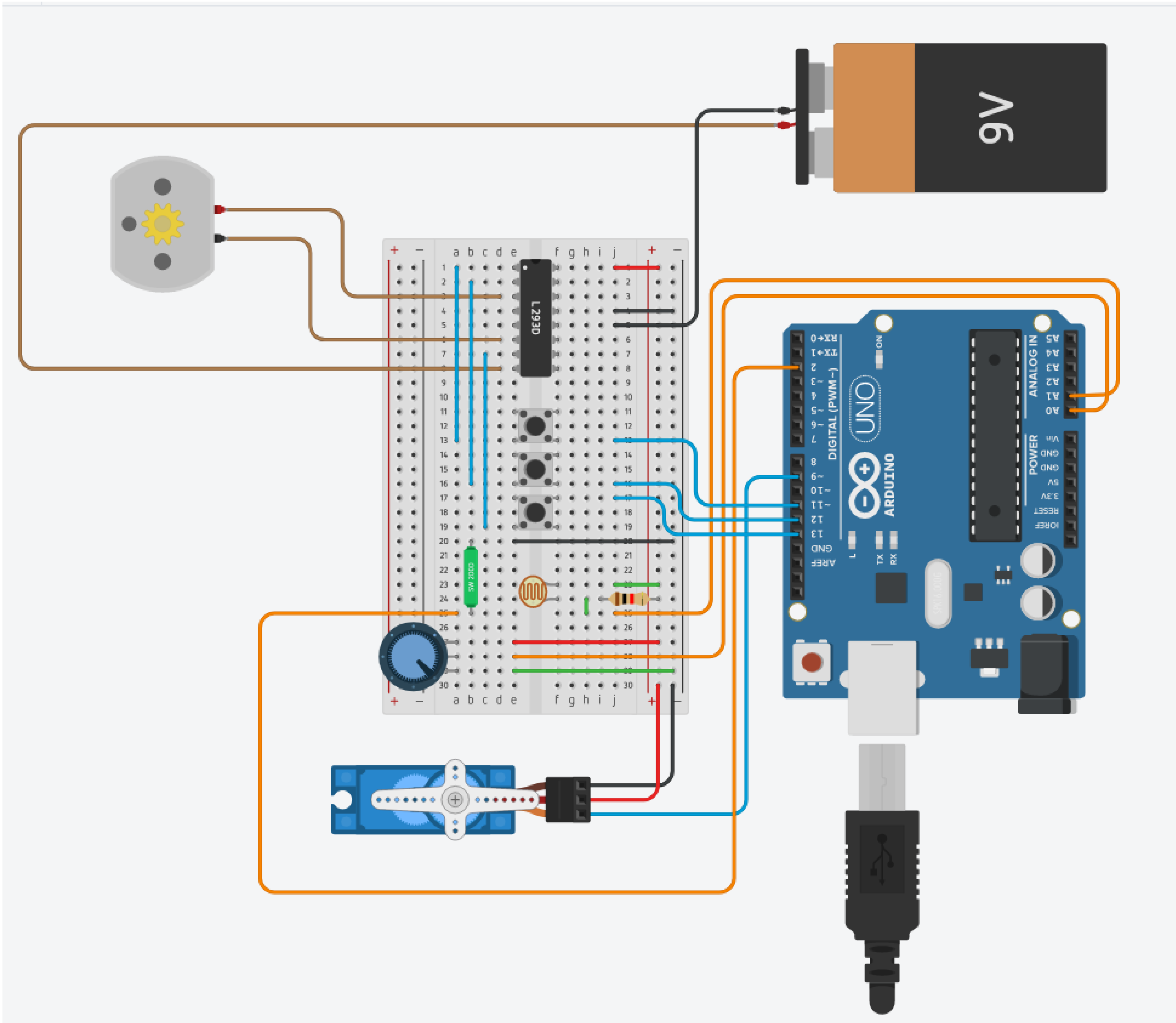

Fig. 6. Large Tinkercad