Christopher David McCarver
5/9/2025
Dr. Bashria Akter Anima
CPE 301.1001 Embedded Systems

FINAL PROJECT REPORT

CPE 301 UNR Final Project Christopher McCarver - YouTube

This project is a real-time monitoring system that tracks water level and temperature to control a fan and stepper motor for cooling and venting. The system uses a DHT11 sensor for temperature and humidity, and a water level sensor for fluid detection. Based on these inputs, it shifts between five modes: ERROR, IDLE, ACTIVE, COMPLETE, and DISABLED. Each mode triggers specific behaviors like LED indicators, fan control, LCD updates, and stepper motor activation. The goal was to create a fully reactive setup using direct port manipulation and sensor feedback, with no blocking delays.

There were a lot of challenges in this project for me. First off, I don't enjoy working with other people in my major. I'm 26, I've been out in the world, and I've spent over a year working in actual rocket science as the ultimate intern — I don't want to be micromanaged by your average university student. Most of the comp sci kids don't know how to communicate or collaborate, and I honestly don't have the patience for it anymore. In over a year in the industry, I've never dealt with the kind of unprofessionalism I've seen here. A lot of these kids have spent their whole lives sitting at computers without developing any other skills, and it's just not getting me anywhere. So, I did the entire project myself. And yeah — that was challenging.

The hardest part of the "single-player mode" for this project was recording the demonstration video. My camera wouldn't record audio no matter what I tried, which was extremely frustrating, so I ended up writing narration separately. I couldn't hold the water level sensor, the fan, and the cold source for the DHT11 sensor all at once, so it took me multiple takes to get a clean shot. Every time I fumbled, wires came loose, and I had to fix connections before trying again.

Hardware issues made things worse. The water sensors kept corroding, and I had to buy a 12-pack just to keep going. Something's wrong with the water in my building — the

sensor's range kept shrinking, and I could actually see buildup on the rails. I also had to replace my entire pack of 9V batteries because the fan drained them during testing. My potentiometers wouldn't stay seated in the breadboard, so I kept having to jam them back in every time I wanted to adjust the vent angle or LCD contrast.

Power distribution was another headache. The stepper motor driver wouldn't get enough current while the fan was running, so I had to make the fan turn off every time the vent rotated — not ideal, but I had no choice. It just wouldn't run otherwise. On top of that, the DHT11 sensor kept disconnecting if the temperature changed too quickly — at least, that's what it seemed like. Maybe it was something else, but either way, it killed a lot of takes.

Trying to capture the transition between ACTIVE and COMPLETE was brutal. The system had to hover in just the right state, and if the sensor disconnected or glitched, I had to start the whole recording over. It took me around 8 hours to finally get a usable video.

- Components and Materials:
  Arduino Mega 2560
- **DHT11 Temperature & Humidity Sensor**
- **Analog Water Level Sensor**
- **16x2 LCD Display** (parallel, not I2C)
- **RTC Module – DS1307**
- **ULN2003 Stepper Motor Driver Module**
- **28BYJ-48 Stepper Motor**
- **5V DC Fan** (powered separately)
- **MB V2 Power Module** (for breadboard, powered by 9V battery)
- **9V Battery** (power for fan and MB V2)
- **Potentiometers (x2)**
  - One for adjusting vent position
  - One possibly used for LCD contrast (from description)
- **Pushbuttons (x3)**
  - Red, Green, Blue (state transitions)
- **LEDs (x4)**
  - Green = IDLE
  - Yellow = DISABLED
  - Blue = ACTIVE
  - Red = ERROR
- **330 Ohm Resistors (x4)** – For LEDs
- **Pull-down or pull-up resistors** (assumed internal or coded for buttons)

- **NPN Transistor** (to drive fan, implied in report)
- **Jumper Wires** – Male and Female
- **Two Breadboards**
    - Large: sensors, buttons, potentiometers
    - Small: MB V2 module, fan, some LEDs

Miscellaneous

- **Duct tape** – used as a visual aid on the stepper motor
- **Frozen food bag** – to cool the DHT11 for demonstration
- **Bowl of water** – to test water sensor levels
- **USB cable** – Arduino to PC
- **Computer with Arduino IDE & Serial Monitor**

The **DHT11 sensor** is used to read temperature and humidity and is connected to digital pin 7, 5V and GRND to the arduino. For water detection, we're using a basic analog water level sensor connected to analog pin A0, and 5v/GRND to the Arduino. These two sensors provide the environmental inputs that the rest of the system reacts to.

The **LCD screen** uses pins 12, 11, 5, 4, 3, and 2, while also having multiple power and ground connections, which are passed into the LiquidCrystal constructor to show real-time information like temperature, humidity, and time. The time data comes from an **RTC module (DS1307)** connected to pins 20, 21, 5v, and GRND of the Arduino, using the Wire.h and RTClib.h libraries.

For output and control, we use four **LEDs (green, yellow, blue, red)** wired to PORTA pins 25 to 28. These indicate the current system mode (IDLE, DISABLED, ACTIVE, ERROR, or COMPLETE). Each LED has a 330 Ohm pulldown resistor. Three **pushbuttons** are also wired to PORTA pins 22 to 24 and are used to transition between modes or trigger shutdowns. Each button has its resistor coded into the program.

The **fan** is connected to pin 6 and is toggled using direct port manipulation. It turns on in ACTIVE mode and off in others. The fan is directly attached to a transistor. The **stepper motor** is connected through a ULN2003 driver module using IN1–IN4 on pins 36 to 29 respectively. These are driven using an 8-step sequence defined in stepSequence[], and the motor runs when the potentiometer changes significantly.
There's also an analog **potentiometer** wired to analog pin A2. We read its value using direct ADC configuration, then map it to step positions to rotate the motor when the value changes enough.

There are two breadboards in this circuit, one whose rails are powered by the Arduino, and the other, smaller board has the MBV2 Power Module attached to the rails and gets power from a 9V battery. The fan gets power from this source; everything else gets power from the Arduino. Though on this same board, in the middle, I put the LEDs because the large board ran out of space. I tried to say a lot of this in my video, but the audio never got picked up. Like I said, I tried all day. Regardless, the LCD screen is attached directly to the large board without wires in between because I only have so many female connector port wires — I had to use them all for the other peripherals that I had to move around. Both potentiometers and all three buttons are attached directly to the large breadboard. Peripherals such as the water level sensor, the real-time clock, the stepper driver module and stepper motor, the temp and humidity sensor, the battery, and the fan are all attached by wires hanging around loosely.

In my video, I start by pointing out all the different components, then show the ERROR state message on the LCD and place the water level sensor into the bowl. Since the water was high enough, I pressed the button to move the system into IDLE mode. Because the sensor was sitting too deep, it immediately went from IDLE to ACTIVE mode and started running the fan. As I interpret the instructions, this behavior is correct given the water reading. Then I readjust the angle of the sensor to bring the system back down to IDLE mode at a medium water level.

While in IDLE mode, I show the IDLE message on the LCD screen and demonstrate the process of changing states using the buttons. First, I show how the screen changes without the buttons in frame, and then I zoom out so you can see the button presses, LCD Screen, and LED changes more clearly.

Next, I demonstrate the stepper motor spinning by attaching a piece of duct tape to make the slow movement more visible. After that, I dip the water sensor deeper into the bowl again to trigger ACTIVE mode. I had a little trouble here since I was bumping things with my hands during the demo. At one point, I had to switch water sensors because the one I was using wasn’t picking up the full range of signals anymore. I struggled for a bit to get both the water level sensor and the temperature sensor working, but I got it eventually. I had to dry the sensor twice, but if you skip ahead to the 5-minute mark, you’ll see ACTIVE mode finally trigger.

At that point, I place the temp sensor on a frozen food bag from my freezer to cool it gradually while holding the water sensor at a steady depth. As the temperature drops, the green and blue LEDs start flickering, and once they both stay on solid (no flickering), that means COMPLETE mode has been reached. You'll also see me use the mouse to highlight the serial monitor output showing the temperature readings during that time.

Once COMPLETE mode kicks in, I show the message on the LCD, and then demonstrate how to exit COMPLETE mode by removing the water sensor from the bowl entirely, which puts the system back into ERROR mode. During both ERROR mode and DISABLED mode, nothing is active except the corresponding LED and button behavior.

Now heres the breakdown of the code line by line, (well I only have so much time so I willdo block by block and get detailed on the important stuff)

```
#include <DHT.h>
#include <LiquidCrystal.h>
#include <Wire.h>
#include <RTClib.h>
```

These are the library includes for all the modules we used. DHT.h is for the DHT11 temperature/humidity sensor, LiquidCrystal.h is for controlling the 16x2 LCD screen, and RTClib.h handles the DS1307 real-time clock.

```
#define DHTPIN 7
#define DHTTYPE DHT11
#define WATER_SENSOR_PIN A0
DHT dht(DHTPIN, DHTTYPE);
```

This is the setup for the DHT11 temp and humidity sensor. It's assigned to digital pin 7 and configured for type DHT11. I also define the analog water level sensor on A0.

```
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
```

The LCD is initialized using the standard 6-pin parallel interface. I didn't use I2C here, sticking with direct wiring to pins 2 through 5 and 11–12.

```
RTC_DS1307 rtc;
```

This initializes the real-time clock object. This shows accurate time updates on the LCD and for serial logs, with pins 20 and 21 being used.

```
volatile unsigned char* port_a = (unsigned char*) 0x22;
volatile unsigned char* ddr_a  = (unsigned char*) 0x21;
volatile unsigned char* pin_a  = (unsigned char*) 0x20;
```

This is the start of the direct memory-mapped I/O setup. These three lines give me access to PORTA for controlling LEDs and reading button states without using digitalWrite or digitalRead.

```
volatile unsigned char* port_h = (unsigned char*) 0x102;
volatile unsigned char* ddr_h  = (unsigned char*) 0x101;
volatile unsigned char* pin_h  = (unsigned char*) 0x100;
```

These registers control PORT H. Specifically, I use PH3 (pin 6) to drive the fan through a transistor.

```
volatile unsigned char* port_c = (unsigned char*) 0x28;
volatile unsigned char* ddr_c  = (unsigned char*) 0x27;
volatile unsigned char* port_d = (unsigned char*) 0x2B;
volatile unsigned char* ddr_d  = (unsigned char*) 0x2A;
volatile unsigned char* port_g = (unsigned char*) 0x34;
volatile unsigned char* ddr_g  = (unsigned char*) 0x33;
```

These are for the stepper motor's ULN2003 driver module. Each input pin (IN1 to IN4) is connected to a separate port.

```
#define LED_GREEN_BIT  4  // PA4=Pin26
#define LED_YELLOW_BIT 5  // PA5=Pin27
#define LED_BLUE_BIT   3  // PA3=Pin25
#define LED_RED_BIT    6  // PA6=Pin28
#define BTN_BLUE_BIT   2  // PA2=Pin24
#define BTN_GREEN_BIT  1  // PA1=Pin23
#define BTN_RED_BIT    0  // PA0=Pin22
#define IN1_BIT        1  // PC1=Pin36
#define IN2_BIT        0  // PC0=Pin37
#define IN3_BIT        7  // PD7=Pin38
#define IN4_BIT        2  // PG2=Pin39
#define FAN_BIT        3  // PH3=Pin6
```

This section defines every bit used in the system and labels it for clarity. Most of these are tied to PORTA for buttons, LEDs, and stepper motor driver.

```
unsigned long lastIdle = 0;
```

```
unsigned long lastBlink = 0;
unsigned long lastLCD = 0;
int lcdScreenIndex = 0;
bool gLED = true;
```

These variables help us track non-blocking time intervals and screen states. Everything here is managed through millis()-based timing.

Now into the Setup Function:

```
volatile bool resumeFlag = false;

void resumeISR() {
  resumeFlag = true;
}
```

This flag and ISR are used to support the DISABLED mode recovery system. When the interrupt triggers on pin 23 (green button), we set the resumeFlag to true so the system can detect it in the main loop.

```
void setup() {
  dht.begin();
  lcd.begin(16, 2);
  Serial.begin(9600);
```

I initialize the DHT11 sensor, the LCD (which uses a standard 16x2 display in 4-bit mode), and the serial connection for logging.

```
  *ddr_a |= (1 << LED_GREEN_BIT) | (1 << LED_YELLOW_BIT) | (1 << LED_BLUE_BIT) | (1 << LED_RED_BIT);
  *ddr_h |= (1 << FAN_BIT);
  *ddr_c |= (1 << IN1_BIT) | (1 << IN2_BIT);
  *ddr_d |= (1 << IN3_BIT);
  *ddr_g |= (1 << IN4_BIT);
```

This is where I define which pins are outputs. These are all direct port manipulations, so no pinMode() anywhere. PORTA is used for LEDs, PORTH for the fan, and PORTC, D, G for the stepper motor inputs.

```
  *port_h &= ~(1 << FAN_BIT);
```

This line turns off the fan at startup. We don’t want the fan turning on until ACTIVE mode is reached.

```
 *ddr_a &= ~((1 << BTN_BLUE_BIT) | (1 << BTN_GREEN_BIT) | (1 << BTN_RED_BIT));
 *port_a |= (1 << BTN_BLUE_BIT) | (1 << BTN_GREEN_BIT) | (1 << BTN_RED_BIT);
```

Here I define the three button pins as inputs with internal pull-ups enabled. This means the buttons read HIGH when unpressed and LOW when pressed, which we check for in the loop().

```
 if (!rtc.begin()) {
  Serial.println("RTC not found");
  while (1);
 }

 if (!rtc.isrunning()) {
  rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
 }
```

This part handles initialization of the DS1307 real-time clock. If the RTC isn't connected, I halt the program with an infinite loop. If it's not running, I sync it to the compile-time date and time, just as a default. This makes sure the serial logs and LCD clock always show something meaningful and match the real world time on my desktop.

```
 currentState = ERROR_MODE;
 setLEDs(false, false, false, true);
```

I start the system in ERROR_MODE to make sure nothing moves or turns on until valid water and temperature readings are detected. The red LED comes on right away.

```
 *port_c |= (1 << IN4_BIT);
 delay(1000);
 *port_c &= ~(1 << IN4_BIT);
```

This is just a quick test pulse to the stepper motor driver to verify power. The delay is one of the only intentional blocking delays, and it's done right after setup before entering loop(), so it doesn't interfere with anything.

```
 attachInterrupt(digitalPinToInterrupt(23), resumeISR, FALLING);
}
```

Finally, we attach the external interrupt for the green button (pin 23) so we can recover from DISABLED mode. It triggers on a FALLING edge, which matches how the internal pull-up logic works. The pin goes from HIGH to LOW when pressed.

LOOP FUNCTION:

```
void loop() {
 unsigned long now = millis();
```

The main loop stores the current time using millis(), which I use to manage all timing without using delay(). It's how I handle time-based updates for blinking LEDs, the LCD screen, and the sensor checks.

```
 if (currentState == DISABLED_MODE && resumeFlag) {
  noInterrupts();
  resumeFlag = false;
  interrupts();
  resumeFromDisabled();
  delay(500);
 }
```

This block handles the interrupt-based resume. I use a flag to safely switch modes without calling functions from inside the ISR itself. The noInterrupts()/interrupts() sandwich protects that global flag from getting changed. This lets the system return to IDLE_MODE cleanly when the green button is pressed in DISABLED_MODE.

```
 if (currentState == ACTIVE_MODE) {
  float temp = dht.readTemperature();
  int waterReading = readWaterSensor();

  Serial.print("Temp = "); Serial.print(temp);
  Serial.print(" | Water = "); Serial.println(waterReading);
```

In ACTIVE_MODE, I immediately check both temperature and water level. I didn't want ACTIVE_MODE to be permanent just because it triggered once. If either sensor changes conditions, we move out of it.

```
  if (waterReading < 400) {
    currentState = ERROR_MODE;
    setLEDs(false, false, false, true);
    *port_h &= ~(1 << FAN_BIT);
    lcd.clear();
    lcd.setCursor(0, 0); lcd.print("ERROR: LOW WATER");
    lcd.setCursor(0, 1); lcd.print("Fan OFF");
    Serial.println("Water LOW, Moving to ERROR_MODE.");
```

```
  }
```

This handles a drop in water level mid operation. If the water level drops too low, I immediately switch to ERROR_MODE, turn off the fan, and notify through both the LCD and serial monitor.

```
  else if (waterReading >= 600 && temp < 25.0) {
    currentState = COMPLETE_MODE;
    setLEDs(true, false, true, false); // GREEN + BLUE
    *port_h &= ~(1 << FAN_BIT);
    lcd.clear();
    lcd.setCursor(0, 0); lcd.print("COMPLETE MODE");
    lcd.setCursor(0, 1); lcd.print("Stable & Cool");
```

When conditions hit right (cool enough temperature and full water level) I consider the operation complete. I light up green and blue LEDs at the same time to signal COMPLETE_MODE and stop the fan with LCD + serial messaging to make sure the state is clear. In the tempurature decent as I was explaining earlier, somehow my code flickers between active mode and idle mode, theres like a range between that 25 degrees and alter 22 degrees that triggers active state sometimes if the water level isn't high enough. So if I had more time I would go in there and fix that but as long as the sensor is held straight I have no problem going into complete mode.

```
    DateTime nowTime = rtc.now();
    Serial.print("Entered COMPLETE_MODE at ");
    Serial.print(nowTime.hour()); Serial.print(":");
    Serial.print(nowTime.minute()); Serial.print(":");
    Serial.println(nowTime.second());
  }
```

I also log the timestamp when COMPLETE_MODE begins.

```
  else if (waterReading <= 580) {
    currentState = IDLE_MODE;
    setLEDs(true, false, false, false);
    *port_h &= ~(1 << FAN_BIT);
    lcd.clear();
    Serial.println("Water MEDIUM, Moving to IDLE_MODE");
  }
}
```

If the water level is no longer considered full, I downgrade back to IDLE_MODE even if the temp is still high. This ensures the system only stays in ACTIVE or COMPLETE when both conditions are fully met. It's reactive and makes the transitions feel like they're really based on sensor data, not timers. This might be the water level discrepancy I was speaking of, yet the flickering between active and idle mode only increased when I edited this block. That does point to this block being the issue yet I do no currently know what I would do about it. I just finally settled on 580 water level because it actually went to complete mode.

```
if (currentState != ERROR_MODE && currentState != DISABLED_MODE && currentState != COMPLETE_MODE) {
  int potVal = readPotentiometer();
  int newStepIndex = map(potVal, 0, 1023, 0, 7);

  Serial.print("Potentiometer ADC: ");
  Serial.print(potVal);
  Serial.print(" → newStepIndex = ");
  Serial.print(newStepIndex);
  Serial.print(" | stepIndex = ");
  Serial.println(stepIndex);
```

This block checks if we're in a state that allows stepper motor interaction. If so, I read the potentiometer using direct ADC configuration, which is mapped from 0–1023 to 0–7 to match the index positions in my 8-step sequence array.

```
  if (abs(newStepIndex - stepIndex) >= 1) {
    bool fanWasOn = false;

    if (currentState == ACTIVE_MODE) {
      fanWasOn = true;
      *port_h &= ~(1 << FAN_BIT);
      delay(20);
    }
```

If the new potentiometer value differs significantly from the previous, I prepare for a rotation. I included logic to momentarily disable the fan during motor movement in ACTIVE mode. This was a workaround for a hardware current limitation, the ULN2003 wouldn't run the motor properly if the fan stayed powered on at the same time. I understand that part of the assignment was to turn the stepper motor while the fan was going, but I just could not get that to happen. No matter where I got the power form on the breadboards, the current

wasn’t enough run both motors from either the 9v battery from the power module or the Arduino’s own 5v output. There is too much happening in the circuit, too many parallel paths, to run both the fan and the vent motor at the same time.

```
  int direction = (newStepIndex > stepIndex) ? 1 : -1;
  int stepsToMove = 1024;
  int stepDelay = 2;

  for (int i = 0; i < stepsToMove; i++) {
   stepIndex = (stepIndex + direction + 8) % 8;
   stepMotor(stepIndex);
   delay(stepDelay);
```

At this point, I figure out whether the motor needs to spin forward or backward by comparing the new potentiometer position to the last one. I then go through 1024 small steps to simulate a full rotation, adding a short delay between each one so the movement looks smooth. Since my motor sequence has 8 steps, I use modulo math to loop the index around as it steps through the pattern. This I thought would keep it moving the right direction, but the potentiometers do not stay in the holes of the breadboard well, and I think when they come out the data coming out from it gets reset somehow because the stepper motor would move back the other direction sometimes, though it moved when it was supposed to.

```
    if (i % 64 == 0) {
     DateTime nowTime = rtc.now();
     Serial.print("StepPos: "); Serial.print(stepIndex);
     Serial.print(" @ ");
     if (nowTime.hour() < 10) Serial.print('0'); Serial.print(nowTime.hour()); Serial.print(":");
     if (nowTime.minute() < 10) Serial.print('0'); Serial.print(nowTime.minute());
Serial.print(":");
     if (nowTime.second() < 10) Serial.print('0'); Serial.print(nowTime.second());
     Serial.println();
    }
  }

  stepIndex = newStepIndex;
```

To avoid flooding the Serial Monitor, I only log motor positions every 64 steps. This provided just enough feedback for debugging without overwhelming the screen. At the end

of the loop, I store the new position so future movement comparisons have a good reference point.

```
  if (fanWasOn) {
    delay(20);
    *port_h |= (1 << FAN_BIT);
  }

  DateTime nowTime = rtc.now();
  Serial.print("Vent demo spin at ");
  if (nowTime.hour() < 10) Serial.print('0'); Serial.print(nowTime.hour()); Serial.print(":");
  if (nowTime.minute() < 10) Serial.print('0'); Serial.print(nowTime.minute());
Serial.print(":");
  if (nowTime.second() < 10) Serial.print('0'); Serial.print(nowTime.second());
  Serial.println();
 }
}
```

Once the motor finishes rotating, if the fan had been temporarily turned off, I bring it back online. I log this moment with the time from the RTC so I can connect fan behavior with mechanical activity.

```
if (currentState == COMPLETE_MODE) {
 float temp = dht.readTemperature();
 int water = readWaterSensor();

 if (temp >= 25.0 || water < 600) {
  currentState = IDLE_MODE;
  setLEDs(true, false, false, false);
  lcd.clear();
  lcd.setCursor(0, 0); lcd.print("COMPLETE → IDLE");

  DateTime nowTime = rtc.now();
  Serial.print("Exited COMPLETE_MODE at ");
  Serial.print(nowTime.hour()); Serial.print(":");
  Serial.print(nowTime.minute()); Serial.print(":");
  Serial.print(nowTime.second());
  Serial.println();
 }
```

```
}
```

Finally, if the system is in COMPLETE mode and the temperature rises again or the water level drops, I automatically bring everything back to IDLE. This makes the COMPLETE state truly dynamic and responsive — the system doesn't just enter COMPLETE once and stay there. I use the same RTC logging format here to capture when the shift occurs, making it easy to identify transitions in both the video and the serial logs. If the water level sensor is completely removed from the water, whether its in Active, Idle, or Complete state, the program will drop the state back to Error state.

More Helper Functions:

```
int readPotentiometer() {
  if (currentState != ERROR_MODE && currentState != DISABLED_MODE) {
    ADMUX = (1 << REFS0) | (1 << MUX1);  // MUX1 = 1 → ADC2
    ADCSRA = (1 << ADEN) | (1 << ADSC) |
          (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);

    while (ADCSRA & (1 << ADSC));
    return ADC;
  }
  return 0;
}
```

This function directly configures and reads from analog pin A2, which is used for the potentiometer that controls the stepper motor (vent position). I don’t use analogRead() because it’s bloated and hides the ADC internals. Instead, I manually configure the ADC using the ADMUX and ADCSRA registers.

Setting MUX1 = 1 selects ADC2 (pin A2), and REFS0 = 1 chooses AVcc (5V) as the voltage reference. The ADC is then enabled (ADEN), and a conversion is started (ADSC). I also set the prescaler to 128 so the ADC clock stays within the recommended range. After waiting for the conversion to finish, I return the result from the ADC register directly.

I also put a check at the top to ignore the read entirely if the system is in ERROR or DISABLED mode. This keeps the motor from responding when something’s gone wrong or the system is shut off.

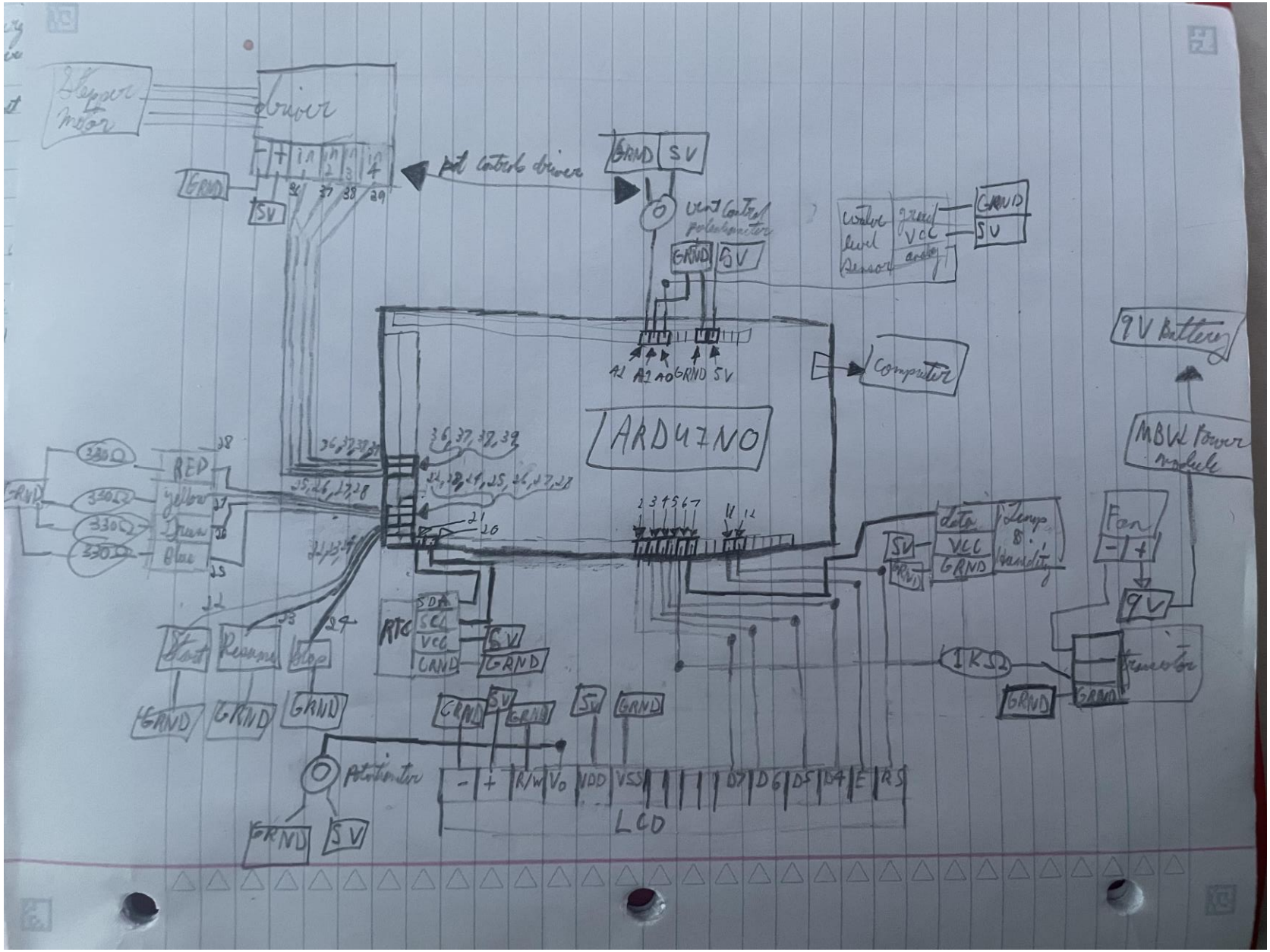
```
int readWaterSensor() {
  ADMUX = (1 << REFS0);
```

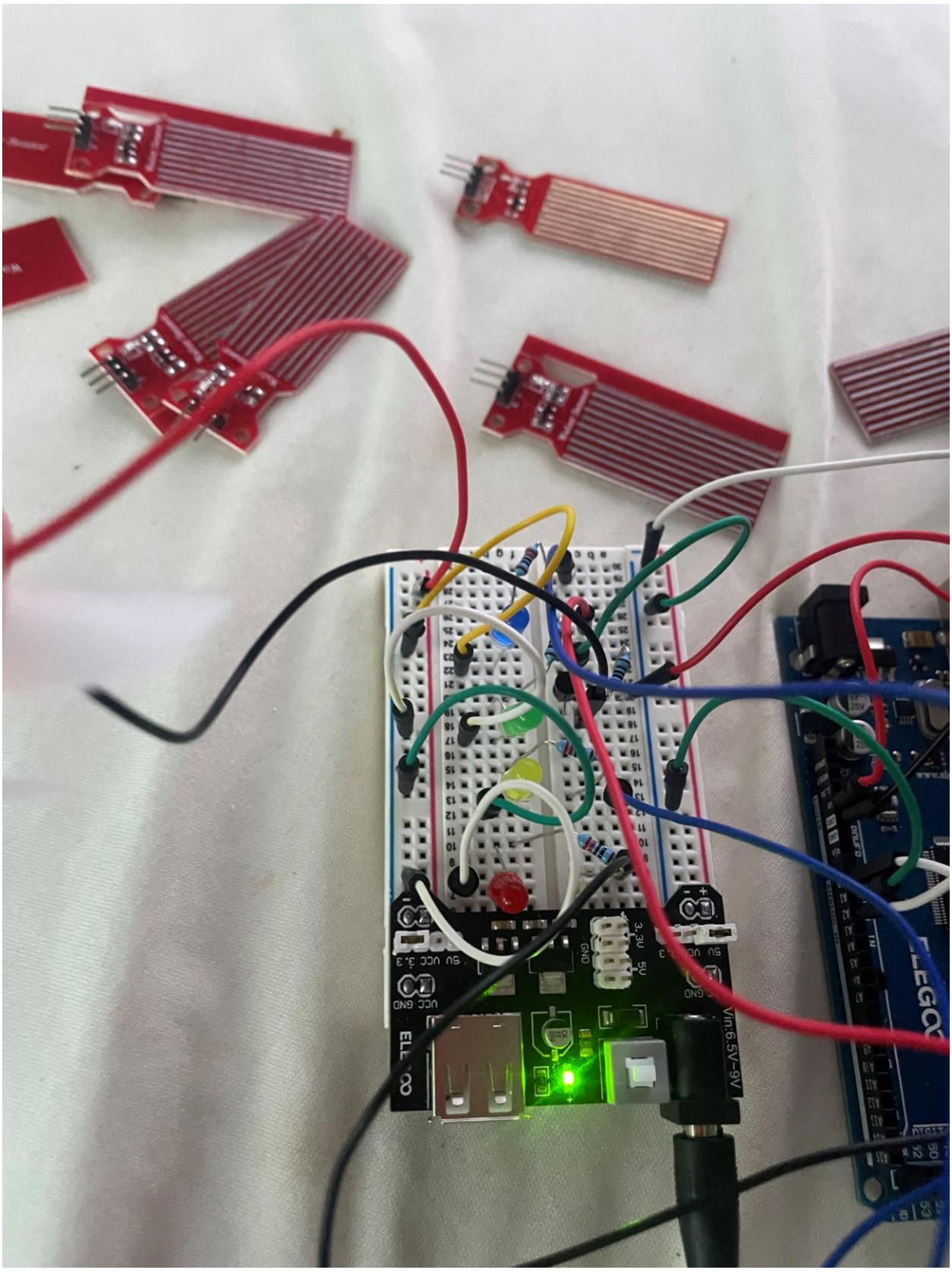
```
  ADCSRA = (1 << ADEN) | (1 << ADSC) |
        (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
  while (ADCSRA & (1 << ADSC));
  return ADC;
}
```

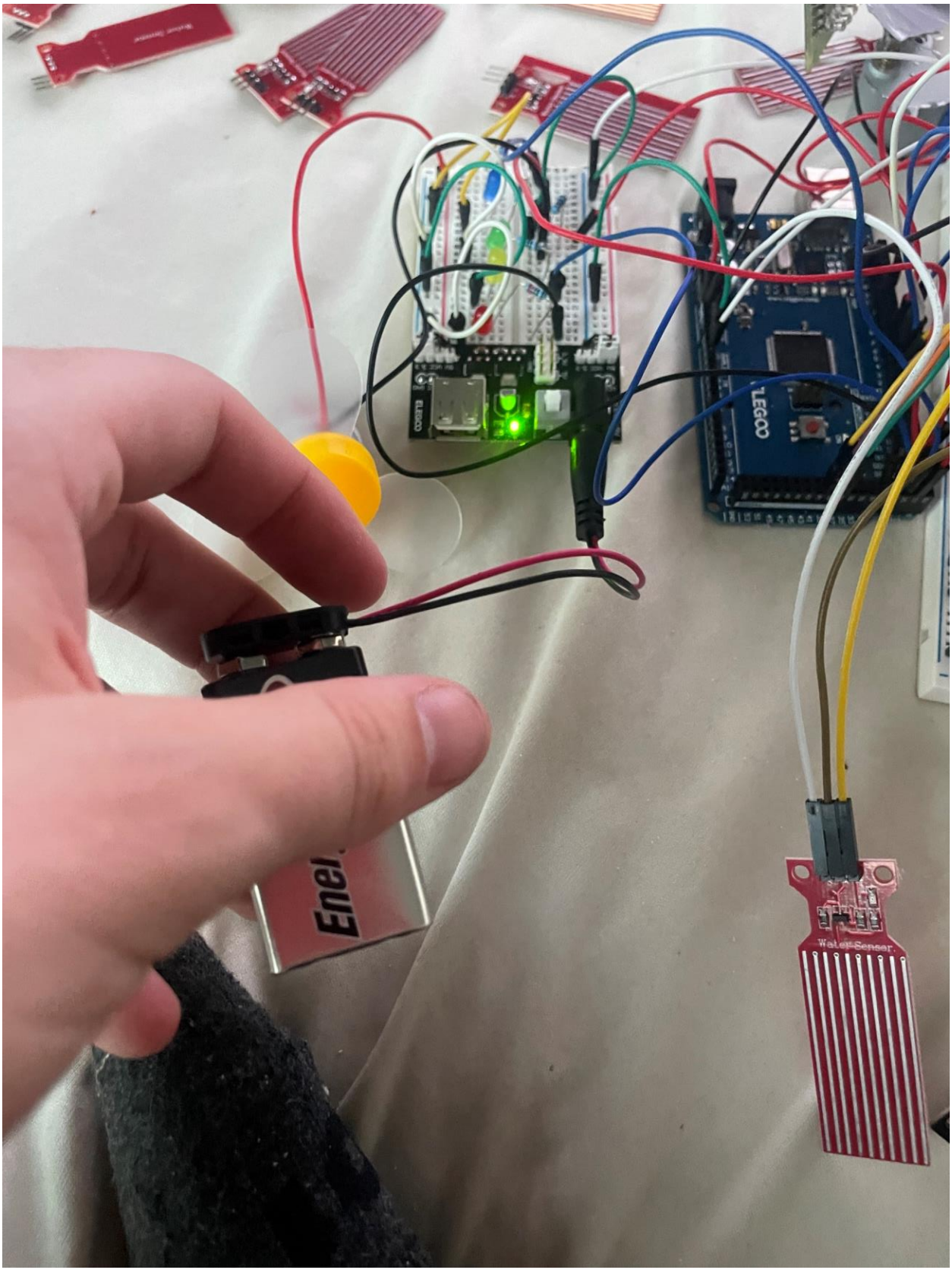
This one works almost the same as readPotentiometer() but reads from ADC0 (analog pin A0), which is where I plugged in the water level sensor. Since MUX bits are all 0 by default, I don't need to set anything for the channel. I just make sure the voltage reference is AVcc (REFS0 = 1) and start the conversion.

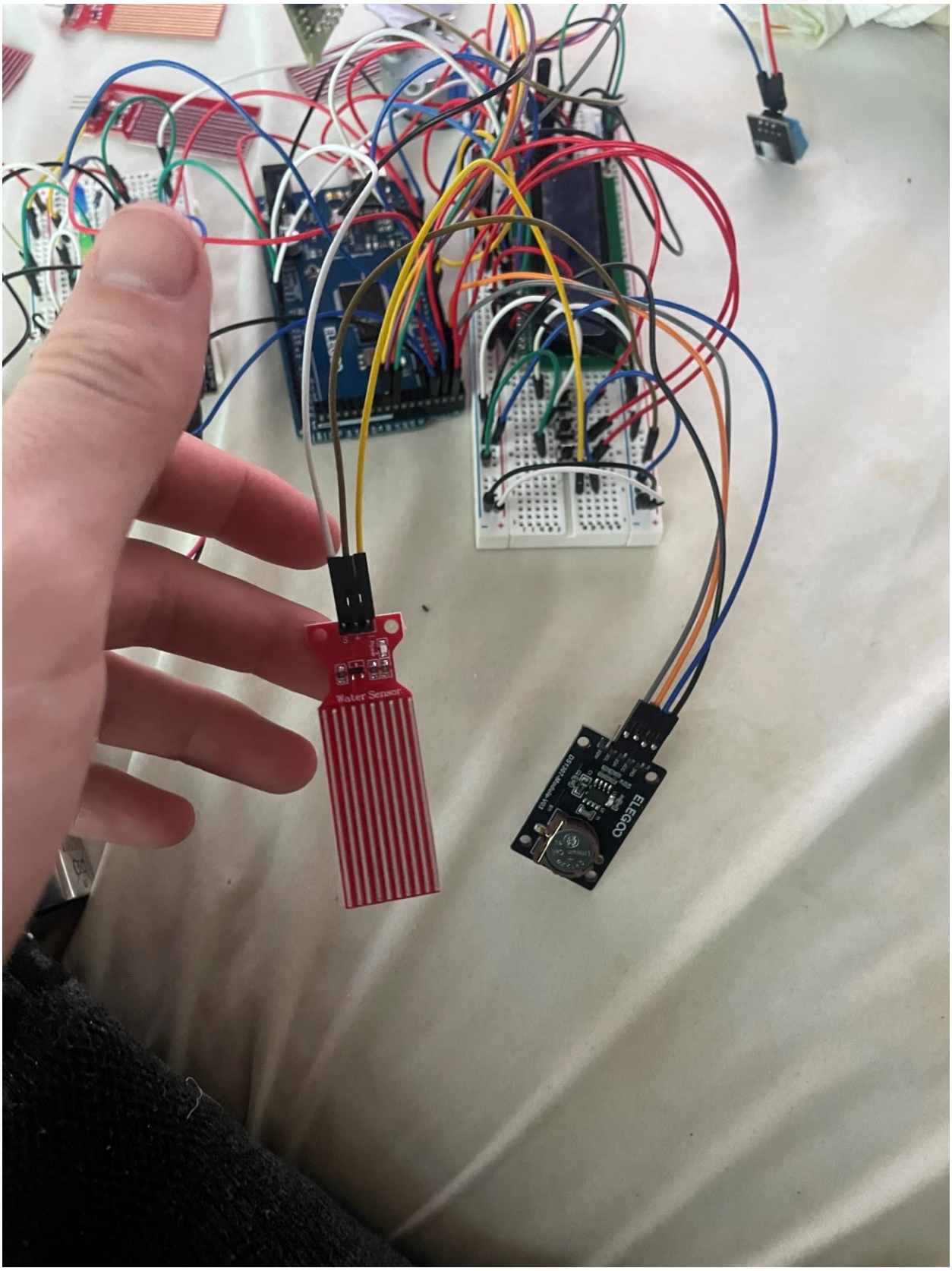
This project was fun. I enjoyed this very much. It can always get meticulous with the small leads and holes, and there's so many wires in this one, but challenge is good, I like challenge. I stated earlier all the hardware troubles I had, definitely made this drag on longer than it should have. I hope my video is good enough for you guys, I tried so many times to get a good take of it, but its hard on my own because I only have so many hands. I cannot hold the fan, water level sensor, rotate the potentiometer, and keep the temp sensor on something cold while the wires are pulling it back. I think that's the only part of this project that was harder with out other students working with me.

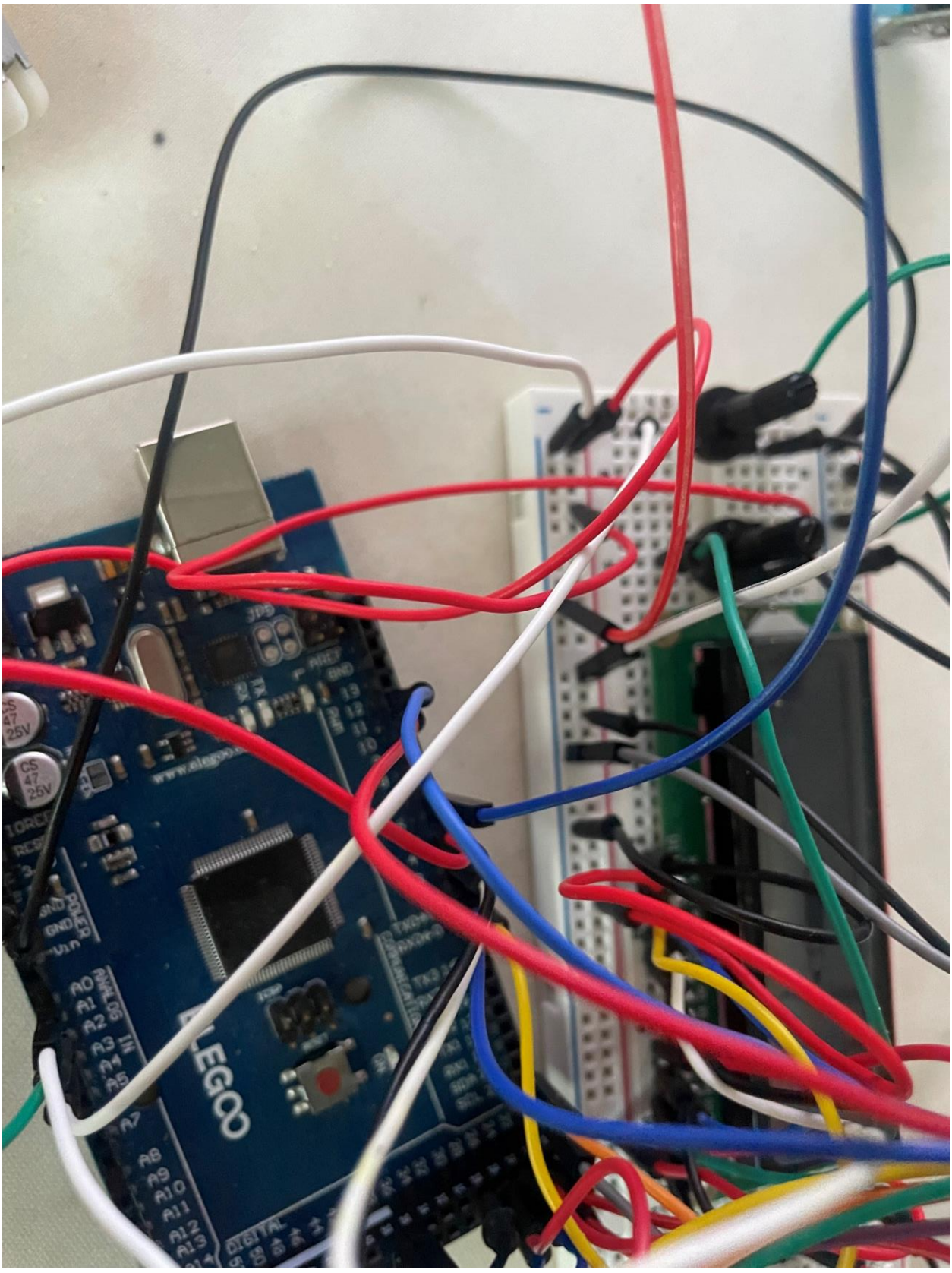
Overall, this project tested my embedded systems knowledge across GPIO, ADC, timers, and real-time constraints, and just about everything we've learned in the class this semester. I met all the core requirements using low-level register programming, managed multiple sensors in real time, and handled state transitions dynamically. Despite hardware limitations, I demonstrated all major features reliably. I'm confident this system shows mastery of the embedded techniques we learned this semester and represents a working real-world control application. I will say though that, the hardware aspect is usually my least favorite part of the degree, at least the earlier classes were because the material was on such a small scale that it felt meaningless and it's a lot of manual labor. This time I got to see something real and compare it to a real world version of these exact builds. We have swamp coolers at work, this is basically the same thing. I could redo this on a larger scale and have one for myself now. Now I have high confidence in my ability to build and rebuild machines. This was cool. Thank you.

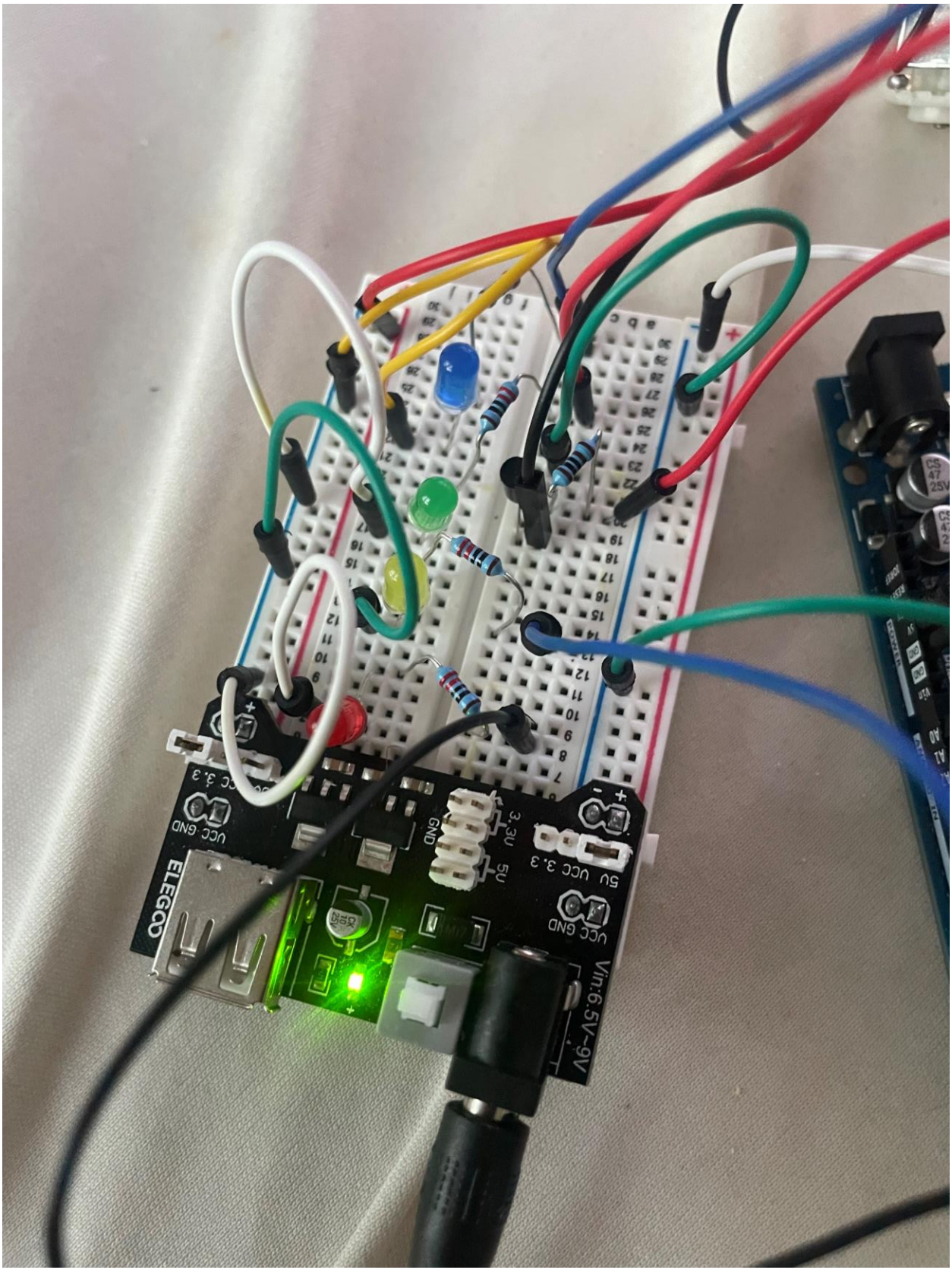

Stepper motor
driver
GRND
5V
Pot controls driver
Water level Sensor
Computer
ARDUINO
9V Battery
MBW Power module
RED
yellow
Green
Blue
Start
Reverse
Stop
RTC
SDA
SCL
VCC
GRND
Data
Temp & Humidity
Fan
IR Sensor
Actuator
LCD
R/W
VDD
VSS
RS

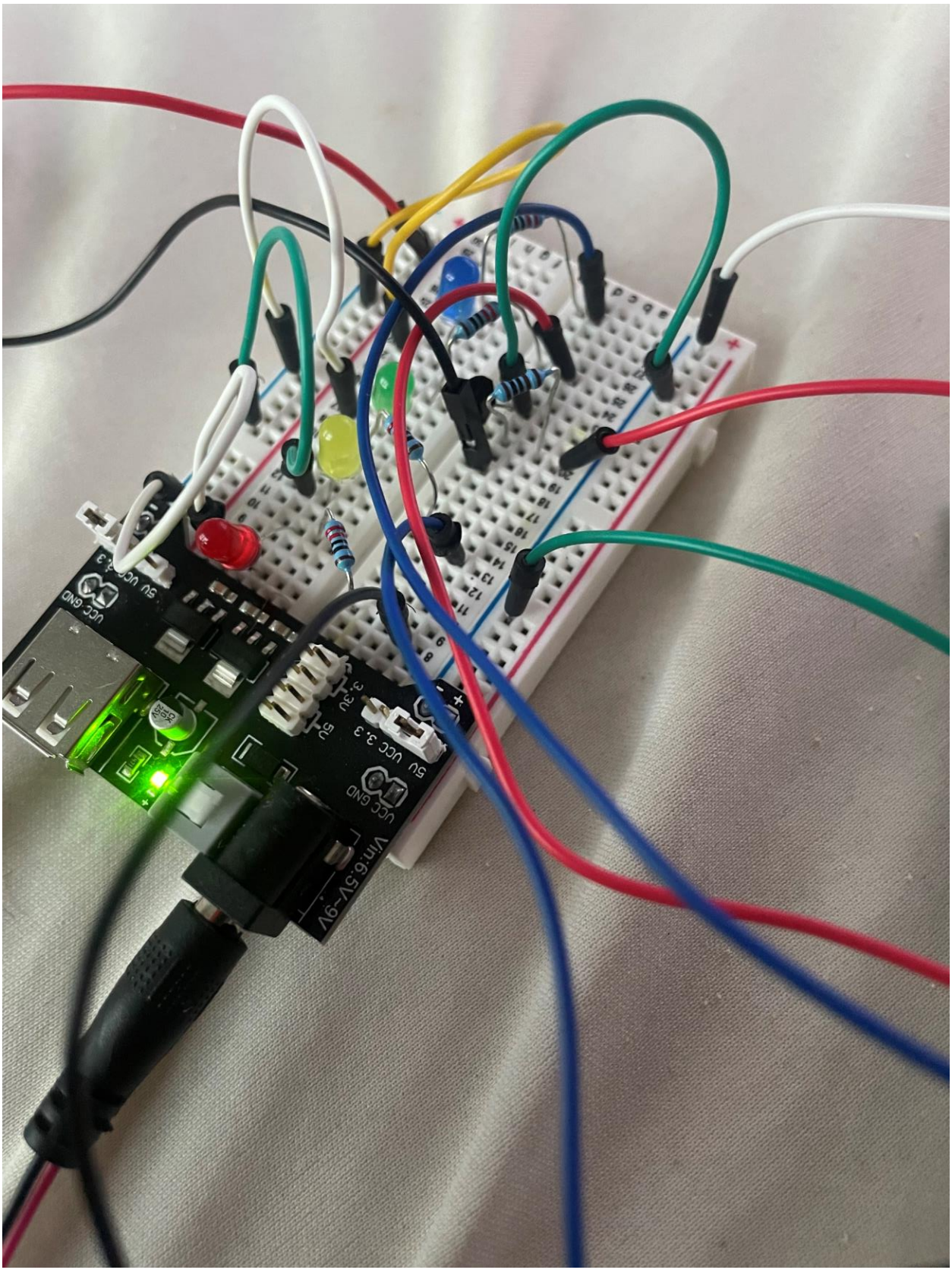

5V VCC 3.3
GND
3.3V 5V
Vin:6.5V~9V

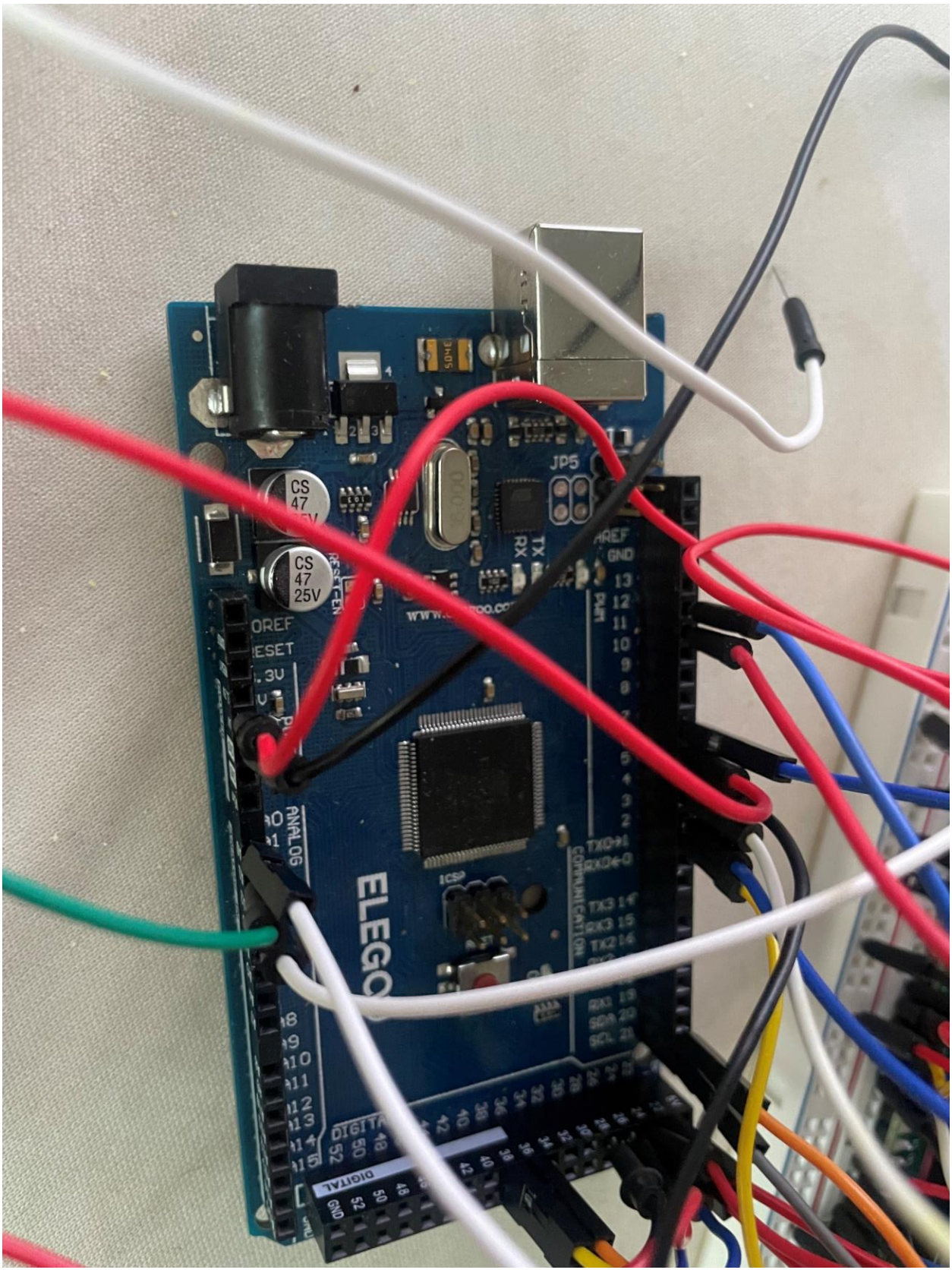

LEGO
Water Sensor

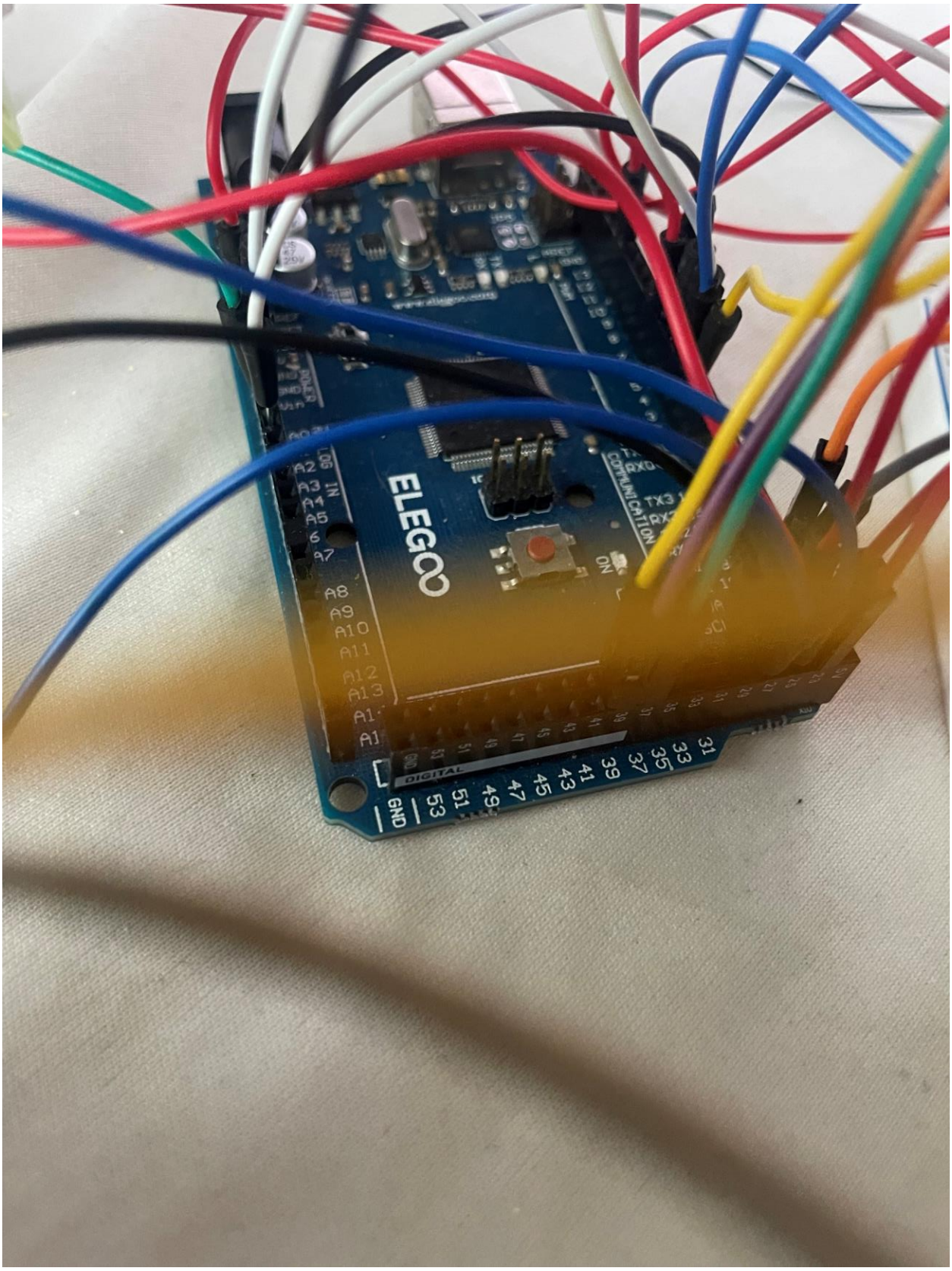

Water Sensor
ELEGOO

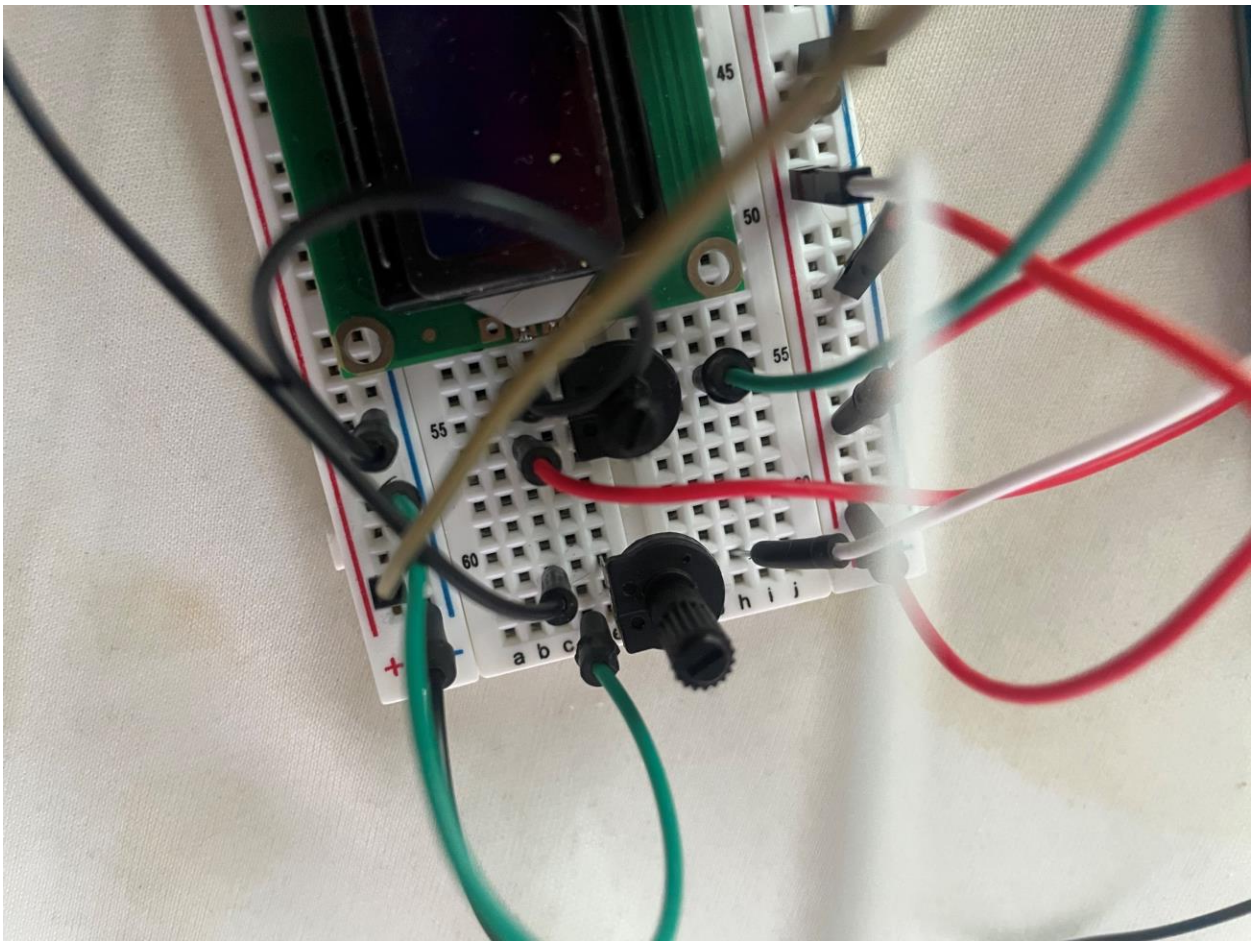

MEGA
DIGITAL
ANALOG IN
POWER

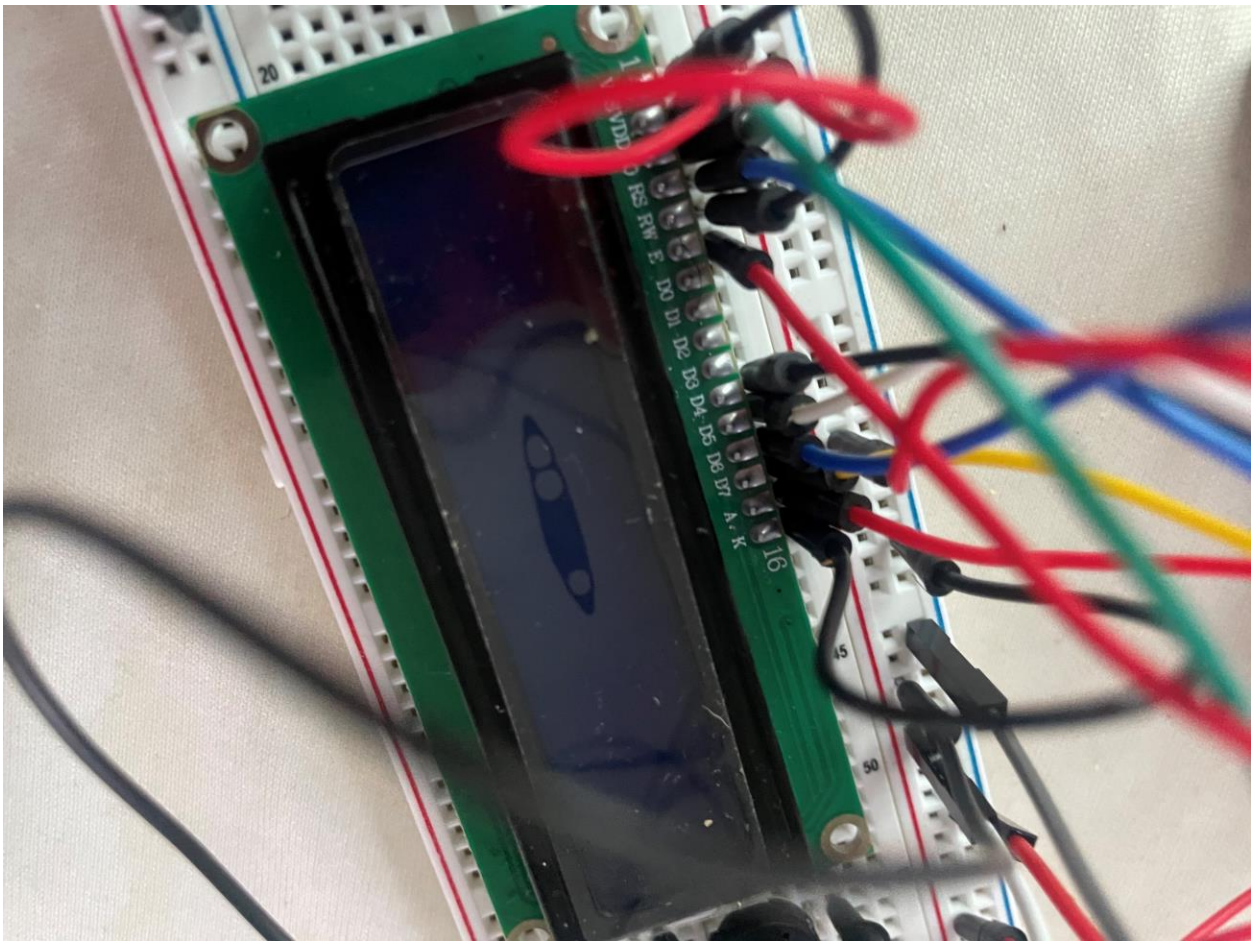

ELEGOO
VCC GND
GND
3.3V
5V
5V VCC 3.3
VCC GND
Vin:6.5V~9V

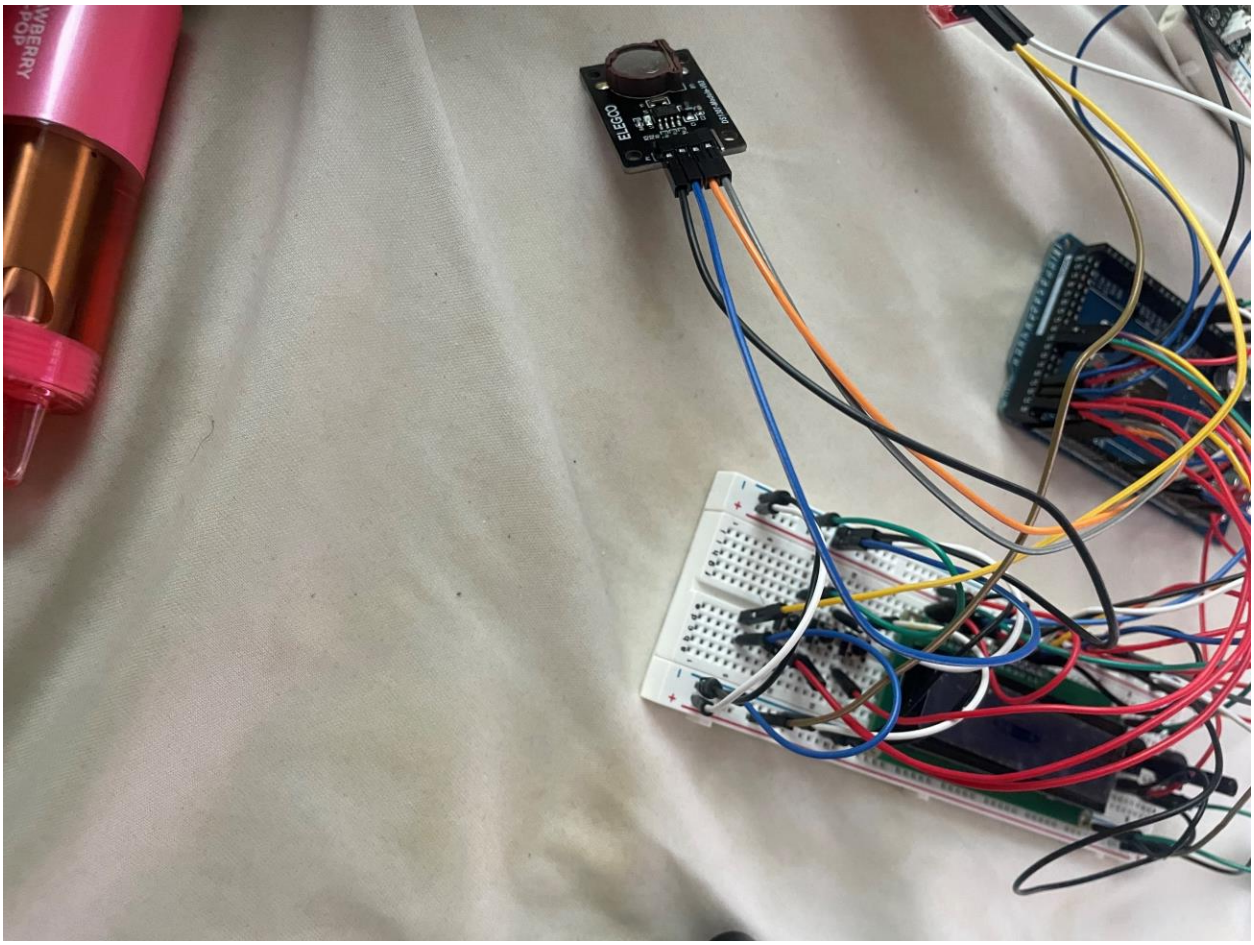

5V VCC 3.3
GND VCC
Vin:6.5V-9V

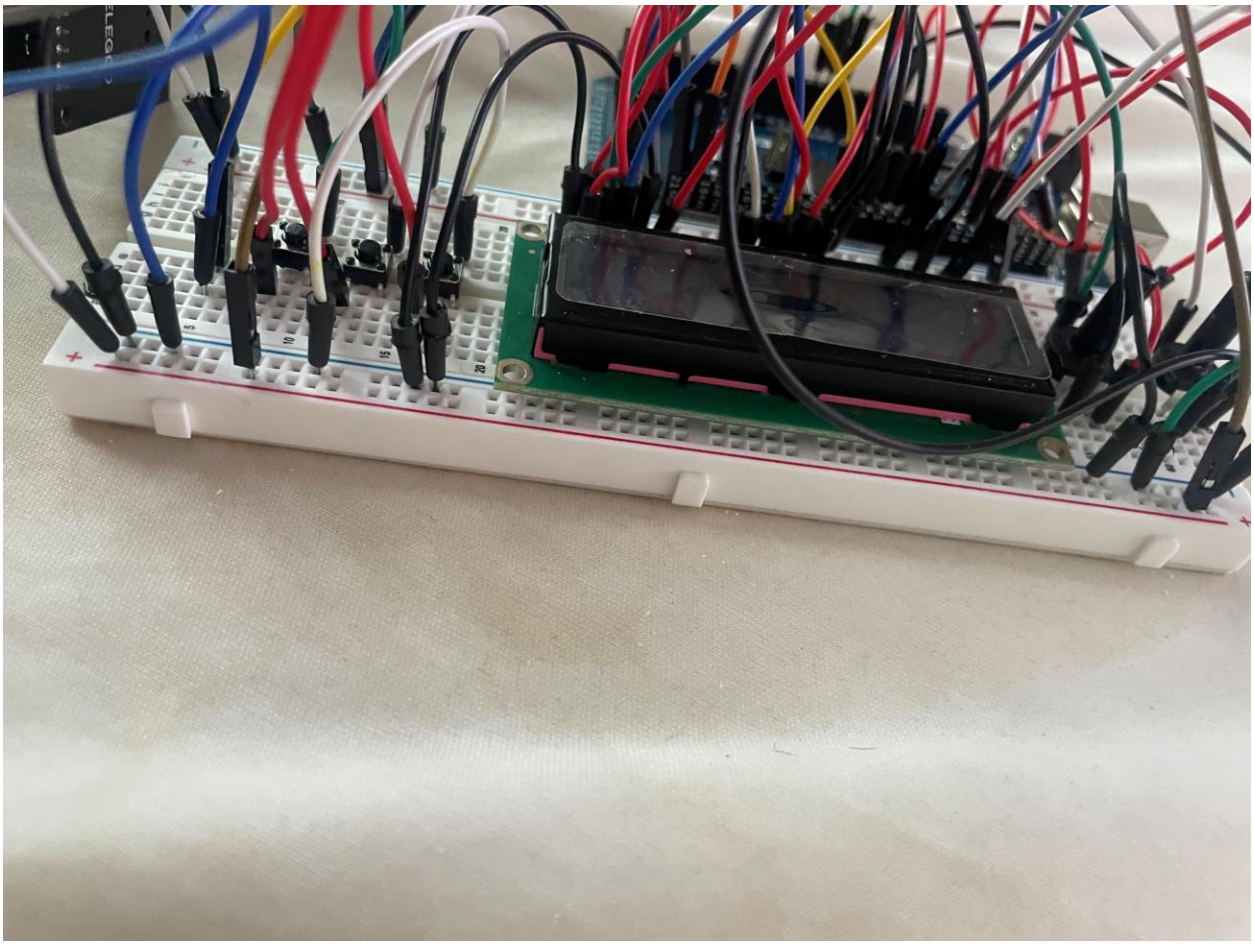

CS
47
25V
JP5
TX
RX
AREF
GND
ANALOG
DIGITAL
ICSP
COMMUNICATION

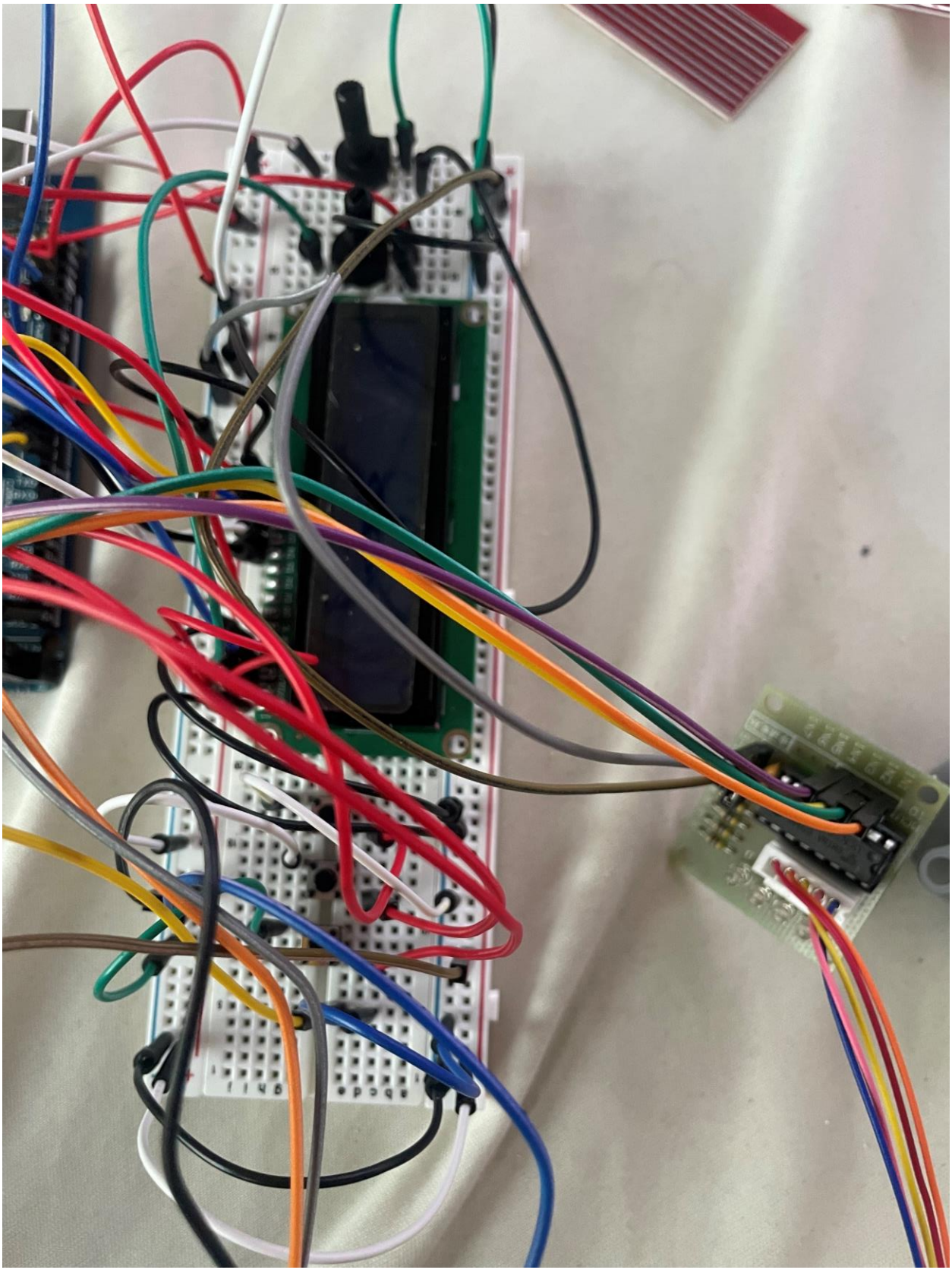

ELEGOO
COMMUNICATION
ON
DIGITAL
GND

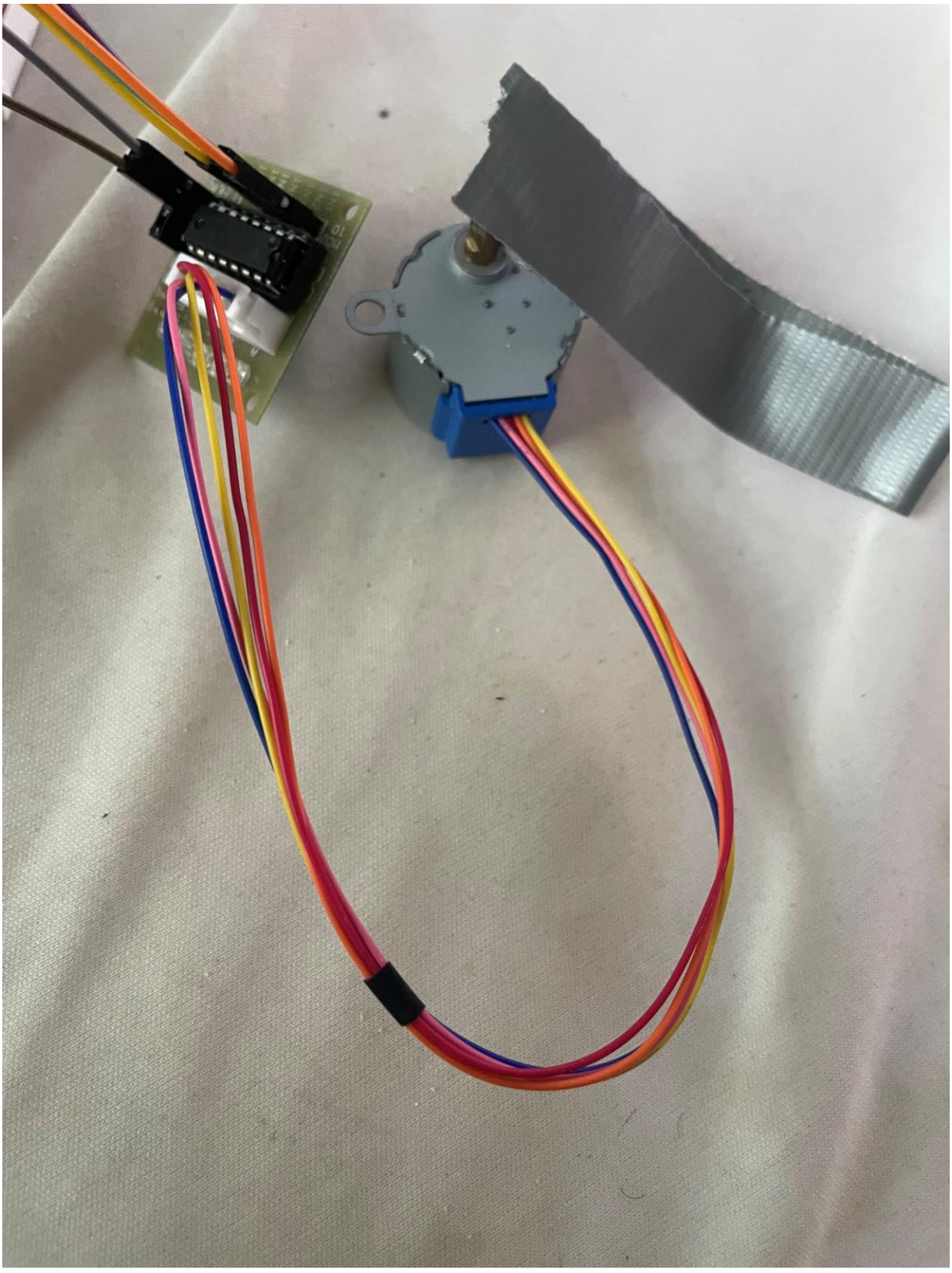

45
50
55
55
60
h i j
a b c

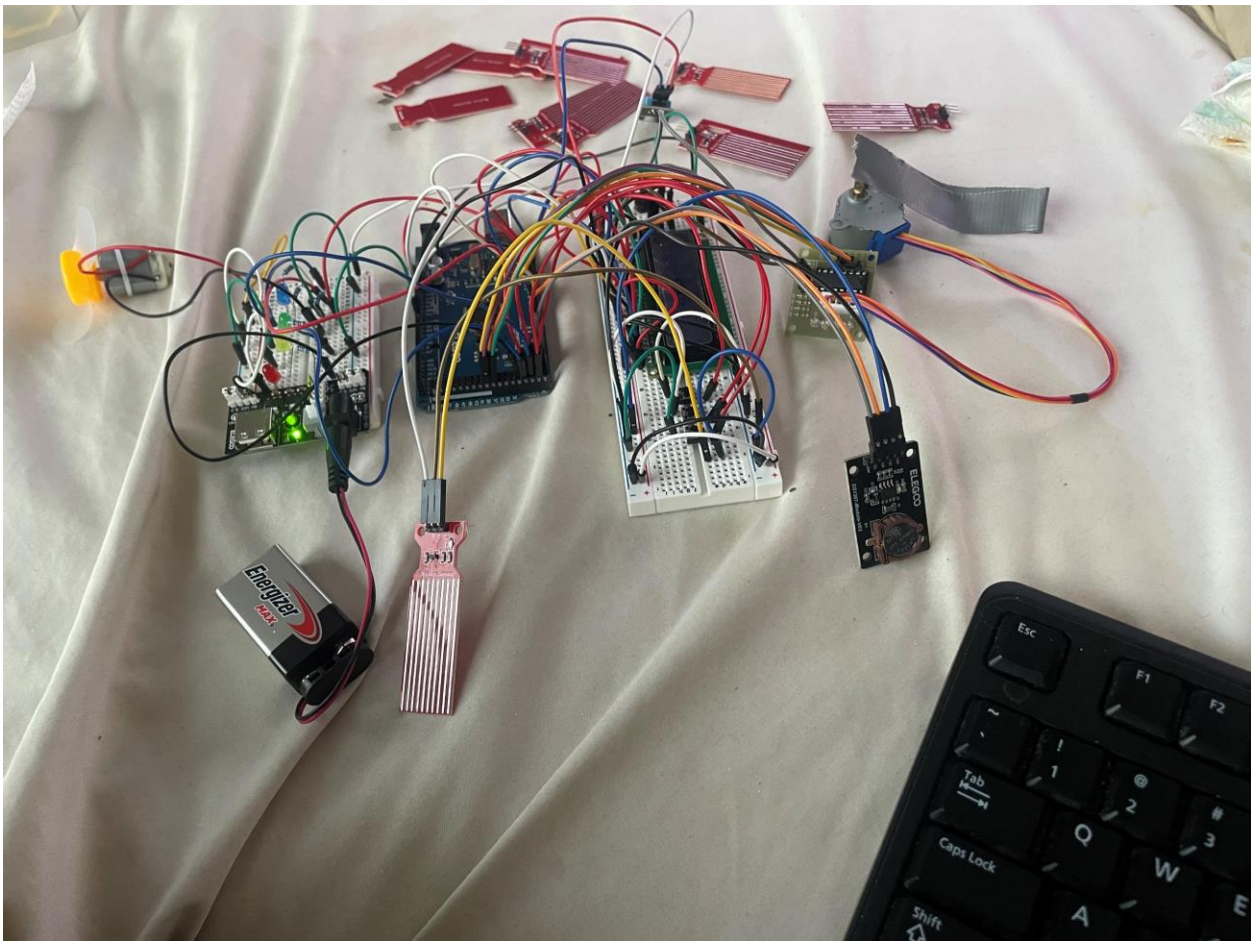

20
RS RW E D0 D1 D2 D3 D4 D5 D6 D7 A K
16
45
50


ELEGOO








Energizer
MAX
Esc
F1
F2
Tab
Caps Lock
Q
W
A
Shift