Sous Vide Immersion Circulator

Ian Kidder & Michal Kalita

ikidder2 / mkal3

December 12th, 2020

ECE 395: Advanced Digital Projects Lab

University of Illinois at Urbana-Champaign

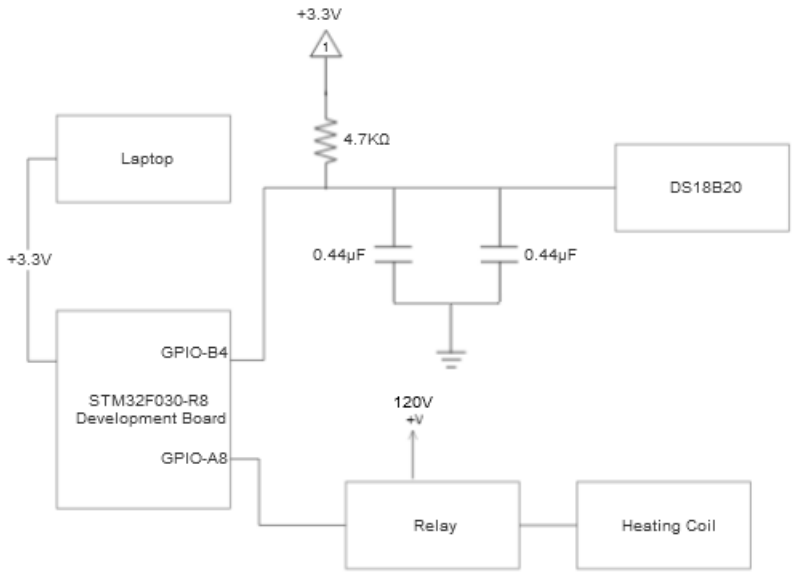
Abstract

Our goal this semester was to build a Sous Vide Immersion Circulator for less money than the cost of an equivalent commercial device. Sous Vide is French for 'Under vacuum' referring to the method by which the device cooks food. A Sous Vide heats water to a user-specified temperature and circulates it around food in a vacuum sealed bag until the food is cooked. We aimed to create our own budget Sous Vide using a temperature sensor, a propellor, a relay, and a water heating element. In the end, the bulk of our project came down to rewriting HAL library functions to support communication between our ARM Cortex-M0 microcontroller and a DS18B20 1-Wire temperature sensor.

1: A commercially available Sous Vide in action (5)

Operation Overview

In order to regulate water temperature, we decided to simply toggle on the heating element whenever the current temperature is below the target cooking temperature and shut it off once reached. We achieve this by requesting new temperature data once per second and toggling the GPIO port high or low depending on the result. The development board we used does not natively support 1-Wire devices, so we used a GPIO port to communicate with the temperature sensor. Establishing communication with the temperature sensor ended up taking so long that we ran out of time to integrate a propellor and a user interface but were able to control the heating element with data from the temperature sensor as planned. When combined with a propellor to circulate the water this should provide a relatively stable temperature.

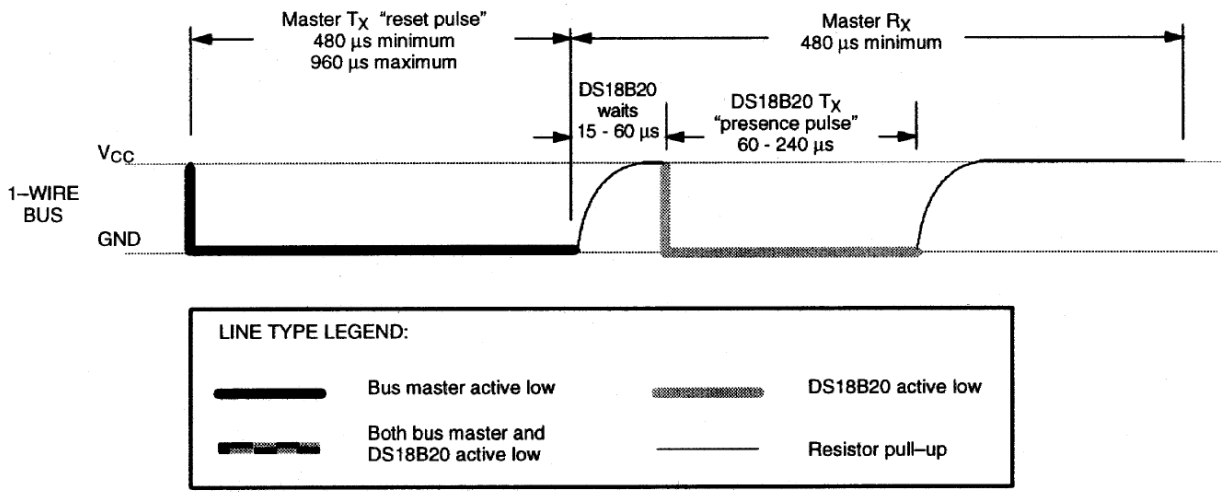


2: Block diagram

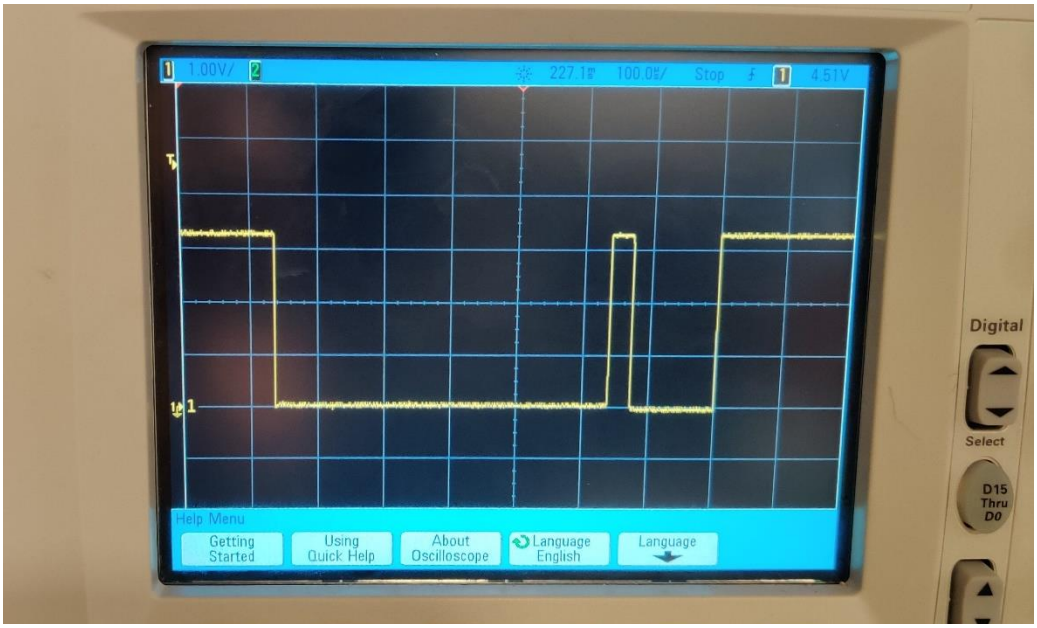
At the core of our design is an ARM Cortex-M0 microcontroller embedded on a STM32F030-R8 development board. The temperature sensor is connected to one of the board's GPIO pins via a single data line supported by a 4.7kΩ pull-up resistor and two 0.4µF capacitors for noise reduction. A second GPIO port is connected to the relay which controls the heating element for temperature regulation.

1-Wire Communication

1-Wire communication is established with a single data line between one master device and multiple slave devices. The data line is connected to a pull-up resistor, allowing Slave devices to hold the line at logic low (≤ 0.8V) or release it to high (≥ 2.2V) as needed. 1-Wire communication always starts with a reset pulse from the master followed by a presence pulse for each device on the bus.



3: 1-Wire reset and presence pulse (1)

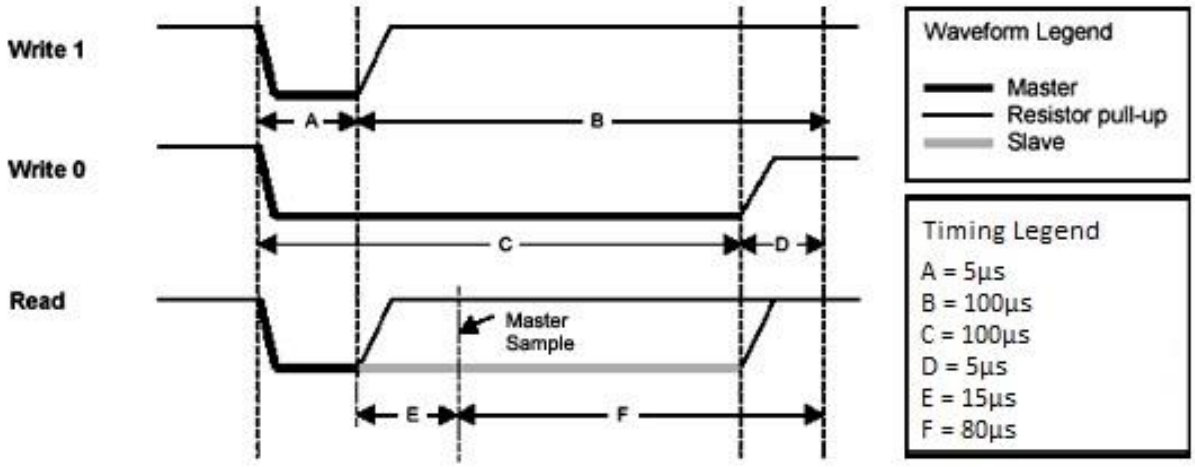


4: Our reset and presence pulses, 100µs / division

We first configure GPIOB-4 into output mode then send low for 500µs for the reset pulse. We were surprised to find that the GPIO port had trouble switching from output to input mode unless we set the output high first. Therefore, after the reset pulse we set the master output high and switch the port to input (high-Z) mode for 500µs to receive the presence pulse from the temperature sensor. Once initialization has been completed and the presence pulse received, the master must specify which device on the bus it is addressing with a ROM command. Since we only have a single 1-Wire device, we use the Skip ROM command to instruct all connected devices to 'listen' to the master.

Writing Commands

Writing commands to the temperature sensor proved to be quite time consuming as we had to create the framework from the ground up. The first hurdle in sending commands to the temperature sensor was getting delays accurate to at least 5µs as necessitated by the 1-Wire timing for reading and writing.

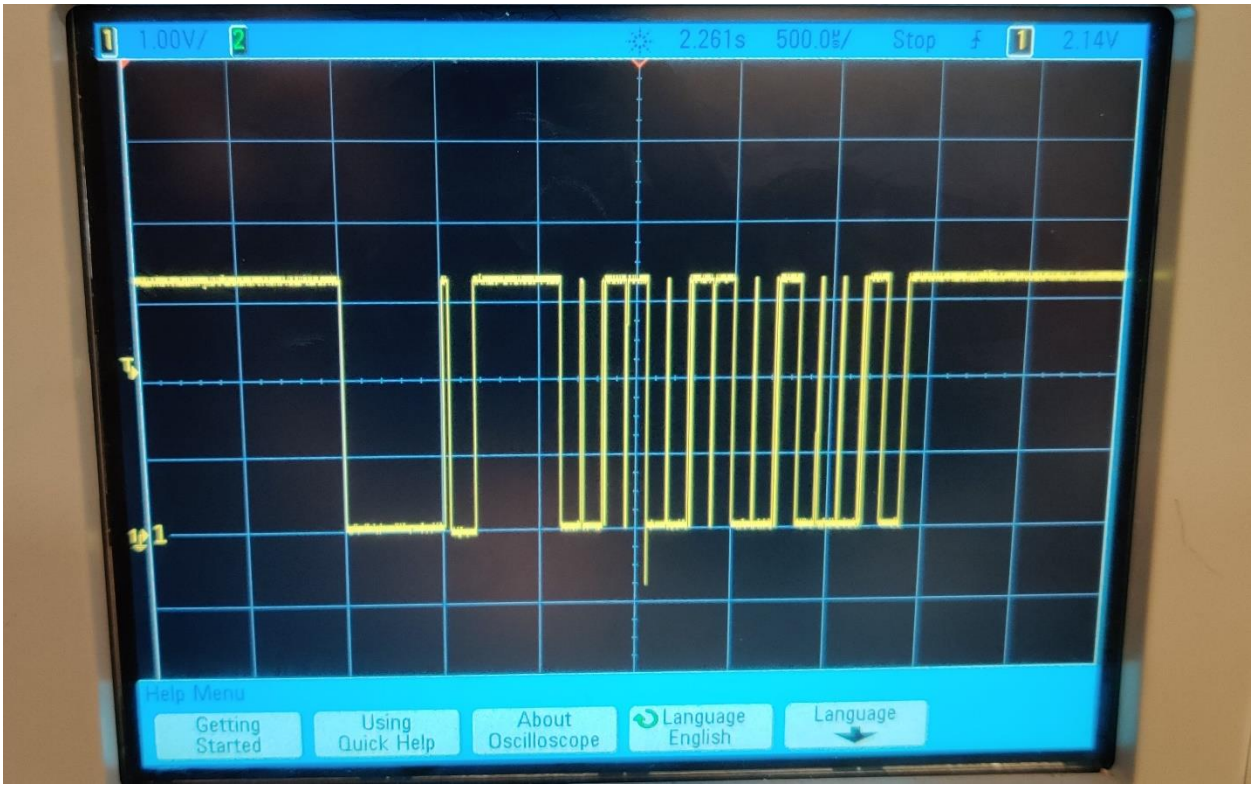


5: 1-Wire timing diagram modified from Maxim Integrated (2)

The smallest delay supported by the HAL library is 1ms, so we wrote our own 1µs accurate delay function using an onboard general-purpose timer. We then wrote code to replicate the write 1 and write 0 time slots for any byte, allowing us to send instructions to the sensor.

```
static void  SendByte(uint8_t  val)
{
    uint8_t n;
    for (n=0; n<8; n++)
    {
        ONEWIRE_OUTPUT_LOW;
        ONEWIRE_CONFIG_OUTPUT;
        delay_us2(5);       // output low for 5 microseconds
        if (val & 1)  ONEWIRE_OUTPUT_HIGH;      // output high for 1, low for 0
        delay_us2(95);      //60 - 120 us delay
        ONEWIRE_OUTPUT_HIGH;
        delay_us2(5);
        val = val >> 1;     // one bit at a time, LSB first.
    }
    ONEWIRE_CONFIG_INPUT;
}
```

6: SendByte function



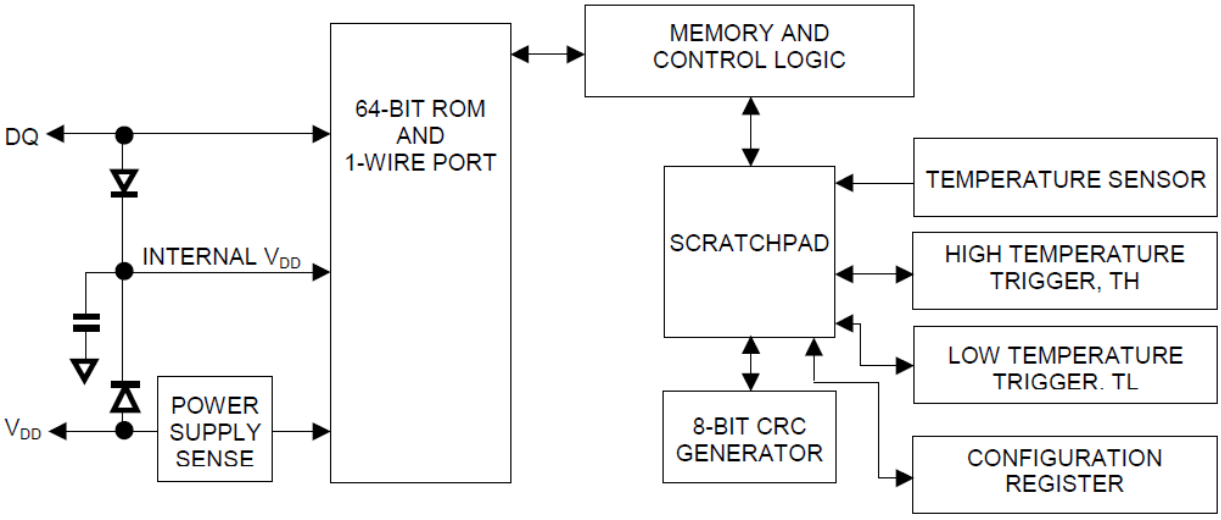
7: Initialization pulse followed by instructions to the sensor

Figure 7 shows the master sending a reset pulse, receiving the DS18B20 presence pulse, then writing 0xCC (Skip ROM) immediately followed by 0x44 (Convert Temperature) to the sensor. Note that instructions are written to the slave least significant bit first.

The process of reading and writing bits first requires proper configuration of the appropriate GPIO pin. Each GPIO bank on the board has a corresponding GPIO mode register which must be configured to switch ports between input and output mode. Input mode means the master releases the data line to high and lets the slave device send data, while in output mode only the master has control over the line. To make this process a bit smoother we defined the macros ONEWIRE_CONFIG_INPUT and ONEWIRE_CONFIG_OUTPUT to access the desired pin and change its mode. We initially used GPIOB-4 for the data line and GPIOB-5 for the relay port but found that we were unable to configure pin 5 into output mode. After much troubleshooting we gave up and swapped GPIOB-5 for GPIOA-8 and the problems were solved. We speculate that we might have been accidentally changing the IO mode of the entire bank B somehow since the swap to bank A fixed our problems, but this does not really make sense since each GPIO port is individually configured. Perhaps GPIOB-5 was just faulty on our board.

Reading from the temperature sensor

Once writing commands to the sensor was completed, we had to receive data from the sensor and convert it into digital values for processing. To understand how receiving data works, we first need to discuss how the temperature sensor operates.



8: DS18B20 block diagram (1)

The sensor stores a new temperature value into the scratchpad as a 16-bit 2's complement reading whenever the Convert Temperature (0x44) command is sent. Each conversion takes a

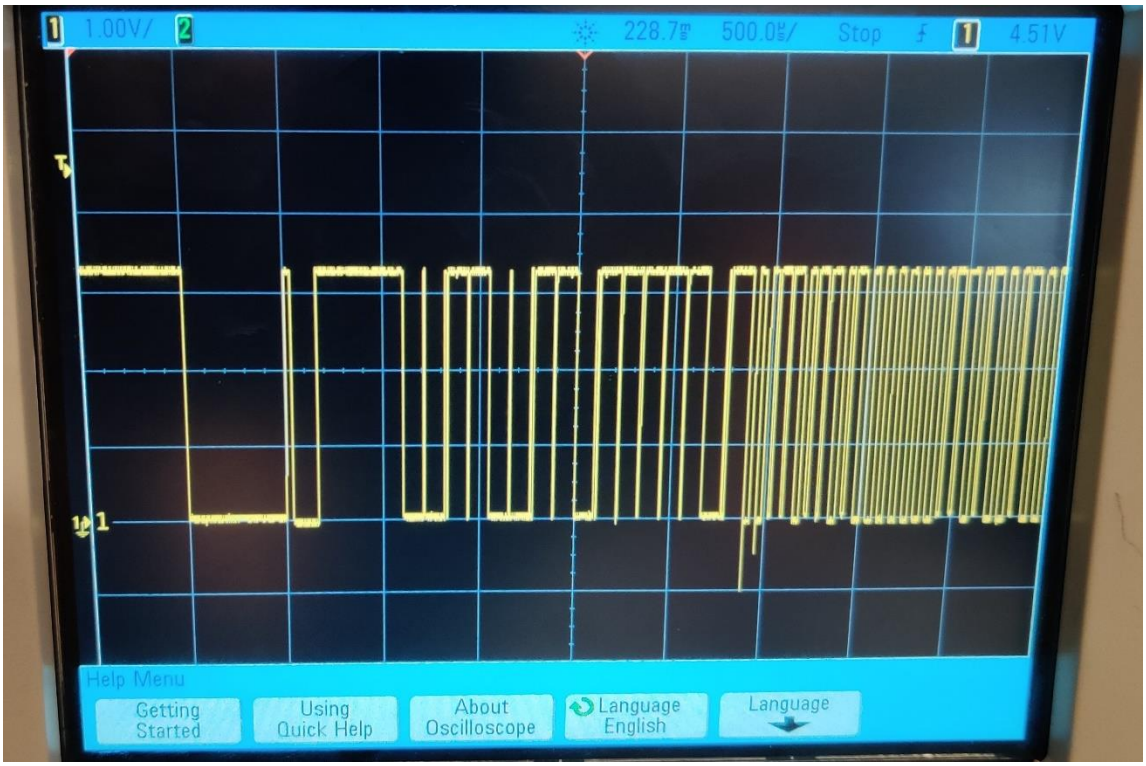
maximum of 750ms to complete, after which the temperature data can be read from the scratchpad by issuing the Read Scratchpad (0xBE) command. After receiving 0xBE, the sensor sends out the scratchpad data LSB first on the data line to the master. The default temperature resolution is 12-bits, which we stuck with despite lower resolutions taking less time to convert (93.75ms is the minimum time when configured to 9-bit resolution) since we did not want to have to write to the configuration register. The scratchpad contains 9 bytes of data, all of which are sent out after Read Scratchpad is issued. Bytes 0 and 1 give the least and most significant bytes of temperature data respectively while the other 7 give configuration information that we did not use. Once temperature data is put onto the data line, it is up to the master to capture the data according to the Read slots as shown in Figure 5. We wrote a ReadByte function to capture all 9 bytes of data and send it as an 8-bit words to our temperature conversion function.

```
static  uint8_t  ReadByte(void)
{
    uint8_t m;
    uint8_t val;

    val = 0;
    for (m=0; m<8; m++)
    {
        val = val >> 1;
        ONEWIRE_OUTPUT_LOW;
        ONEWIRE_CONFIG_OUTPUT;
        delay_us2(10);
        //ONEWIRE_OUTPUT_HIGH;
        ONEWIRE_CONFIG_INPUT;   // Adjusted high-Z from 10 to 15us
        delay_us2(15);
        if (ONEWIRE_INPUT_READ)  val = val | 0x80;
        delay_us2(35);
    }

    return  val;
}
```

9: ReadByte function



10: Receiving Data from the sensor

Figure 11 captures the following interaction:

1) Master sends Initialization / Reset pulse followed by DS18B20 presence pulse
2) Master sends 0x44 (Skip ROM)
3) Master sends 0xBE (Read Scratchpad)
4) Sensor sends 9 bytes of scratchpad data LSB first

We were stuck reading only 0's from the sensor for a long time until realizing that we were not meeting the required 480µs minimum Master RX delay as shown in figure 3. We also had some corrupted data readings until adjusting the Master sample window from the manufacturer recommended 10µs up to 15µs. With reliable data coming into the microcontroller, the last step was simply to convert these bits to floats and compare with the user specified cooking temperature to control the relay and heating element.

S	S	S	S	S	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

11: 2-Byte, 12-bit temperature format in degrees Celsius

```
static float  ReadTemperature(void) // will not work for negative temperatures.
{
    uint16_t raw_data;  // used to be unsigned, leading 0s are getting truncated.
    uint16_t data_1;
    uint8_t  decimal;
    uint8_t  integer;
    uint8_t  pad[9];
    float result;

    temp_initialize();
    delay_us2(100);
    SendByte(SKIP_ROM);
    SendByte(READ_SCRATCHPAD);

    for (int n=0; n<9; n++)
    {
        pad[n] = ReadByte();
    }
    data_1 = (pad[1] << 8);
    raw_data = data_1 | (pad[0]);           // extracts the 2-byte temperature data in big-endian
    integer = ((raw_data & INTEGER_MASK) >> 4); // takes the integer data from the raw data
    decimal = raw_data & DECIMAL_MASK;          // takes the decimal data from the raw data
    result = integer + (float)decimal*0.0625;   // adds the integer + (0.0625/lsb) * (decimal in binary)
    return result;
}
```

12: ReadTemperature Function

```
while (1)
{
    ConvertTemperature();
    Current_Temperature = ReadTemperature();

    if(Current_Temperature < COOK_TEMPERATURE){
        RELAY_OUTPUT_HIGH_A8;
    }

    else{
        RELAY_OUTPUT_LOW_A8;
    }
    HAL_Delay(1000);
}
```

13: Extremely simple control loop

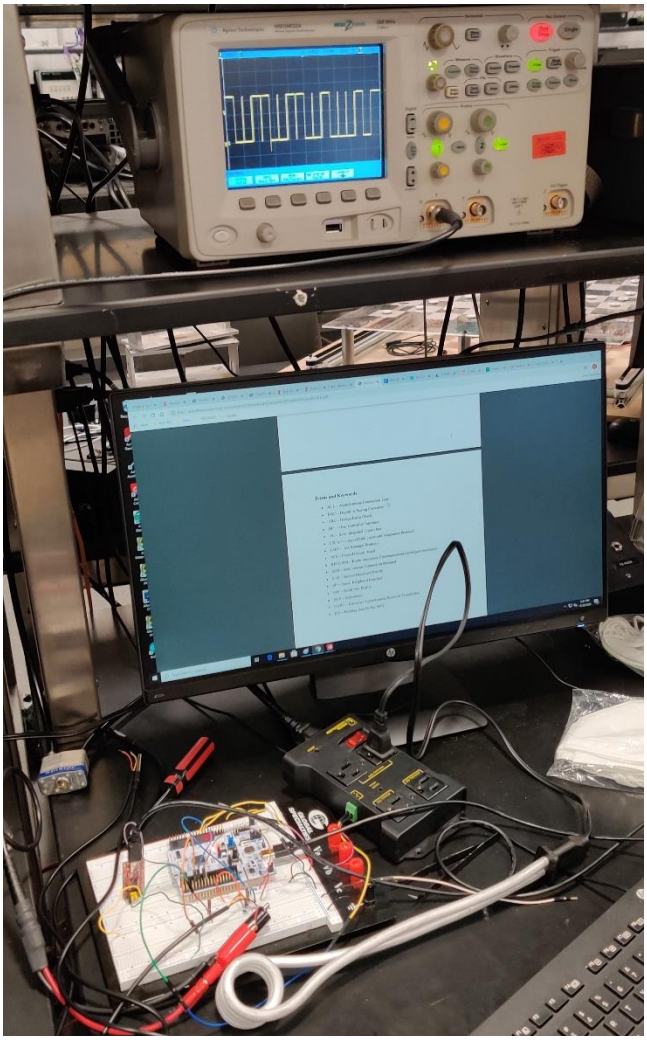
Future Work

The only thing we are missing for a fully functional (albeit clunky) Sous Vide is a way to circulate water around food. This would be easily solved by purchasing a small propellor and dropping it into the cooking pot. There are also a few improvements that would greatly improve the user experience, such as a digital display of the current temperature and buttons to more easily adjust the cooking temperature. If we were to continue this project, we would also design a PCB and design a waterproof housing for the entire device.

Other

It felt wrong to submit this report without mentioning the most exciting part of our project: the heating coil melting. We suspect that the 1000-watt heating coil used in this project was a fake replica of another brand given that it melted within 5 seconds of plugging it into the relay, leaving residue all over the floor of the lab.

14: Melted heating coil with exposed wiring



15: Lab station with the heating coil moments before disaster

References

1) https://cdn.sparkfun.com/datasheets/Sensors/Temp/DS18B20.pdf
2) https://www.maximintegrated.com/en/design/technical-documents/app-notes/1/126.html
3) https://www.st.com/resource/en/user_manual/dm00105928-getting-started-with-stm32-nucleo-board-software-development-tools-stmicroelectronics.pdf
4) https://www.st.com/content/ccc/resource/technical/document/user_manual/98/2e/fa/4b/e0/82/43/b7/DM00105823.pdf/files/DM00105823.pdf/jcr:content/translations/en.DM00105823.pdf
5) https://www.williams-sonoma.com/products/anova-pro-precision-cooker/