Project 3 - Embedded System

Reverse Engineering

Elaborated by:

Daniel Baptista Andrade - 93313

José Luís Rodrigues Costa - 92996

June 13, 2022

1. Introduction

1.1. Objectives

The objective of this project is to thoroughly document the greatest and most detailed amount of information possible by using online information, tools (hardware and software) and knowledge of a provided embedded system used to control a thermal camera.

1.2. Context

This project revolves around a thermal chamber with two cavities, both intertwined by a perforated wall. A microcontroller is responsible for controlling 4 main elements:

- Two outputs that can increase or decrease the chamber temperature (OC).
- Two thermal sensors that can measure the current temperature in each chamber.

For each chamber a set-point can be set for the desired temperature in each chamber.

The controllers main elements include:

- An EEPROM memory
- An RS232C port
- Both temperature sensors
- Both OCs
- LED’s

1.3. Summary

By using the multiple techniques ranging from using Pulseview with a logic analyser, to reading components documentation and using the provided documentation for this project, it was possible to analyze the full behavior of the microcontroller, including the code reversing process to demonstrate most of the analyzed behaviors.

The reverse code is available in the example_reversed_code.c, which is only an example of the code, it’s not executable.

This report is organized by communication protocol (or by component) and in each section all relevant information is explained. Starting in the mapping pins section to identify the board pins, the data structure when needed (RS232C), the full behavior (that can be multiple) for each component and the mode association, where the behaviors are associated with the board modes (normal and set point mode) and the time sequence is analyzed.

In some components behavior it’s also possible to verify the reverse code analysis that supports the observed behavior.

In the last section possible component interactions are presented.

2. Microcontroller Analysis

2.1. RS232C

2.1.1. Mapping Pins

The RS232C communication protocol uses 2 pins to allow information to be Received (Rx), via OC3 and Transmitted (Tx) via OC4.

2.1.2. Configuration Parameters

To calculate the baudrate, 1 bit duration is about 52 microseconds which, by looking at datasheets online, is equivalent to a baudrate of about 19200.

Given the baudrate, checking the bit sequence, it was possible to confirm that there are 8 data bits, followed by an odd parity bit and finalizing with one stop bit. Using this configuration the RS232 sequences show no framing and parity errors.

2.1.3. Reversing Behavior

2.1.3.1. Initial Behavior

When starting the system while measuring the signals coming from the OC4 and OC3 pins and defining the previous parameters for an RS232C read, a message, in ASCII, appears saying: “Reverse Engineering”, as shown in Figure 1, only during the system startup.

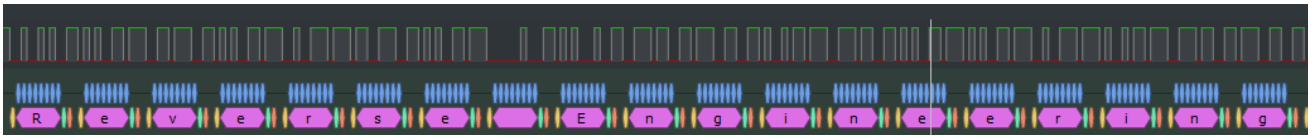

Figure 1 - RS232C Initial Message

2.1.3.2. Debug Message

The debug message sent from the RS232C bus has the following structure:

First, there was a message saying “Temperatures:'' that was delimited by “\n”; next, 30 temperature measurements were received and, after another “\n” character, another 30. Finally, there was one last message: “Set Points: \n” followed by another 3 measures, as we can see in the following Figures.

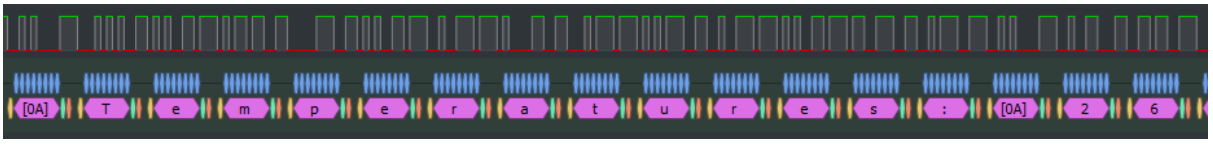

Figure 2 - RS232C Temperatures Message

Figure 3 - RS232C Set Points Message

Due to size limits and to make the Figures readable, it was not possible to place the full message in this report.

After this observation, it was possible to see that, each time there is a “Temperature” message, the next 30 values correspond to one of the sensor's temperature history and after the “\n” the other 30 values correspond to the other sensor temperature history. This makes sense as in subsection 2.2.4.3. Reading Temperature History the temperature history is read from memory and in the subsection 2.2.4.4. Writing Temperature History the temperature history is written to memory.

The remaining 3 values are all of the set point values, first the right side set point, secondly the left side and lastly the average set-point.

2.1.3.3. Mode association

The behavior presented by the RS232C is always the same in any mode.

This includes the initialization where it shows one initial message() and the 2.1.3.2. Debug Message**.**

It is also possible to verify that, between every debug message 2.1.3.2. Debug Message, there is a 7 second interval, as presented in Figure 4.

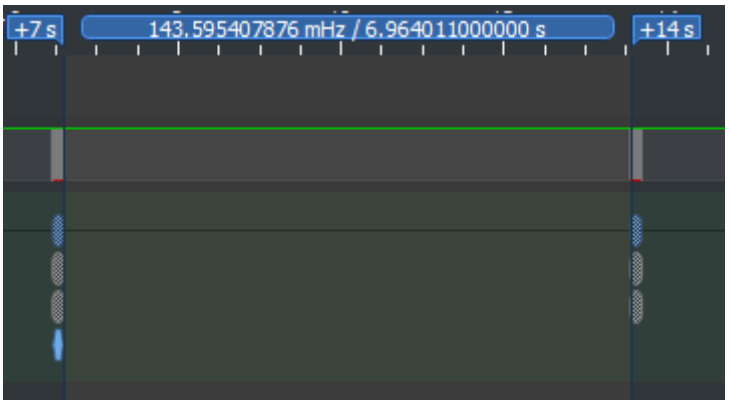

Figure 4 - RS232C Interval

2.1.4. Reversing code (Real behavior)

2.1.4.1. Debug Message

To verify the behavior explained in 2.1.3.2. Debug Message, the reverse code snippet in figure 5 show all the following steps:

1) The temperature history is read from memory.

2) After the “\nTemperatures:\n” is printed

3) The historical measurements are printed from each temperature sensor, separated by a “\n”.

4) Read the set points from the memory addresses referenced in subsections 2.2.5. Memory Address Table (0x43, 0x44 and 0x45).

```
                    */
if (iGpffff802c == 1) {
  iGpffff802c = 0;
  i = 0;
  len_temp_hist = Read_temperature_history(temp_hist);
  printRS232("\nTemperatures:\n");
  temp_hist1 = temp_hist;
  if (0 < len_temp_hist) {
    do {
      i = i + 2;
      print_rs232_numTOascii(*temp_hist1);
                /* print_space
                   */
      print_rs232_1(0x20);
      temp_hist1 = temp_hist1 + 2;
    } while (i < len_temp_hist);
  }
  i = 0;
  printRS232("\n");
  temp_hist1 = temp_hist;
  if (0 < len_temp_hist) {
    do {
      i = i + 2;
      print_rs232_numTOascii(temp_hist1[1]);
      print_rs232_1(0x20);
      temp_hist1 = temp_hist1 + 2;
    } while (i < len_temp_hist);
  }
  len_temp_hist = 0;
  printRS232("\nSet Points:\n");
  do {
    uVar1 = read_memory(len_temp_hist + 0x43);
    print_rs232_numTOascii(uVar1);
    len_temp_hist = len_temp_hist + 1;
    print_rs232_1(0x20);
  } while (len_temp_hist < 3);
```

Figure 5 - RS232C Set Points Message

2.2. SPI (EEPROM)

2.2.1. Mapping Pins

The SPI communication protocol uses 4 pins to allow the master to talk to the slave. The MISO, MOSI, CLK and CS pins.

The correspondence of each pin can be observed in figure 6.

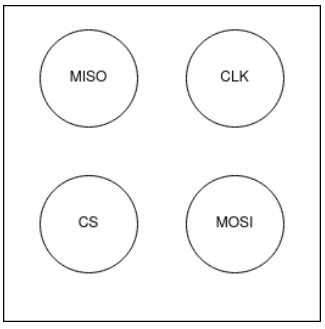

Figure 6 - SPI Pin Composition

To identify the pins, information from both the signal analysis and EEPROM documentation was used.

By analyzing the signal in figure 7, it can easily be observed that the D3 pin is connected to the clk, as the signal shows a square wave with a 50% duty cycle when the communication is occuring.

The D1 pin is the CS, as during the data transfer it has the value 0, and when the transfer ends, it returns to the default 1 value.

This leaves both the D2 and D0 pins being either MOSI or MISO. To determine which one is which, the EEPROM documentation (figure 8) was used to determine which kind of operation was being conducted on the figure 7. This documentation states that the read status register operation starts by sending 0x05 to the MOSI pin, then the response will be the status register information.

In this case, as the 0x05 appears in the D0 pin, this indicates that this is the MOSI pin and the response on the D2 line indicates it is the MISO pin.

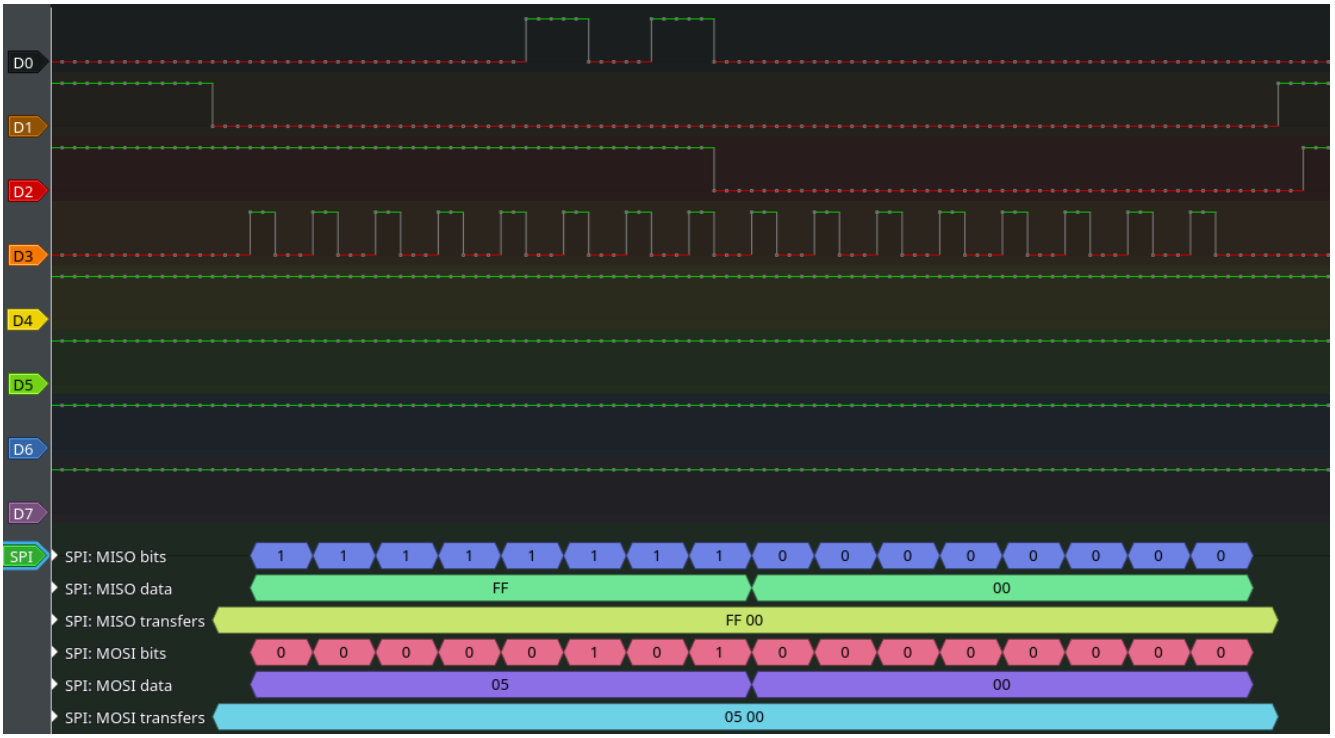

Figure 7 - SPI Signal

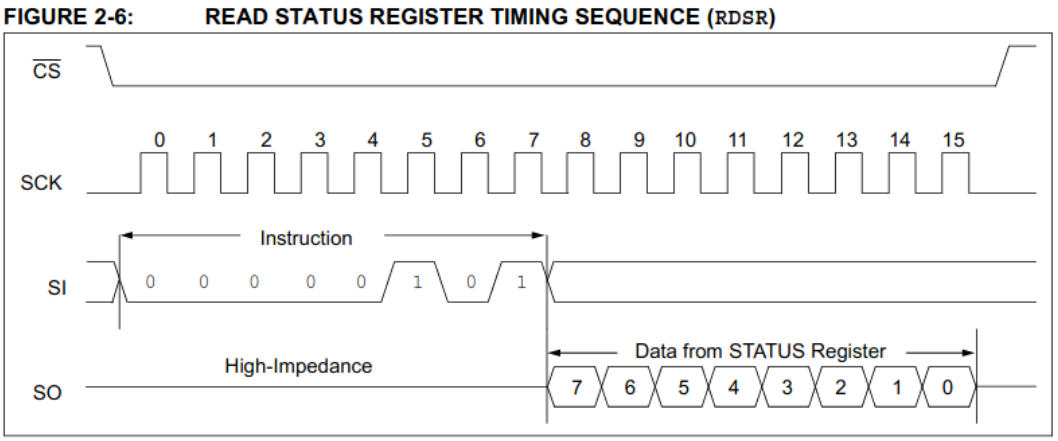

Figure 8- EEPROM Documentation

2.2.2. General Parameters

The general parameters for the SPI behavior is the frequency at which it operates, the clock polarity and phase.

In this case, the frequency when data is being transmitted / received is calculated using the following formula:

$$f = \frac{1}{Full\ Time\ Period}$$

$$f = 200\ kHz$$

The clock polarity is low as the SCK pin is LOW when idle and toggles to HIGH during active state (during a transfer).

The clock phase is low as data is transmitted on the falling edge.

2.2.3. EEPROM documentation brief

To better understand the reversing behavior section (2.2.4. Reversing Behavior), a summary of the most relevant information regarding the EEPROM documentation is necessary.

The EEPROM works by first providing instructions via the MOSI line. Figure 9 illustrates all possible instructions.

The most relevant instructions for our scope are the READ, WRITE, RDSR (read status register) and WREN (set write latch enabled).

For the read instruction, first the master send the instruction code (0x3), the address of the data and waits for the reply from the EEPROM (slave) (figure 10).

For the write instruction, the master sends the write instruction code (0x2), the address followed by the information to be stored in the address (figure 11).

For the RDST instruction, the master sends the RDST instruction and the EEPROM responds with the status of the device. When this response is 0, the device is not busy with a write operation.

The WREN instruction is sent before a write, to enable writing on the EEPROM, in this case, the code is 0x6.

Instruction Name	Instruction Format
READ	0000 $A_8$011
WRITE	0000 $A_8$010
WRDI	0000 x100
WREN	0000 x110
RDSR	0000 x101
WRSR	0000 x001

Figure 9 - EEPROM Instructions Documentation

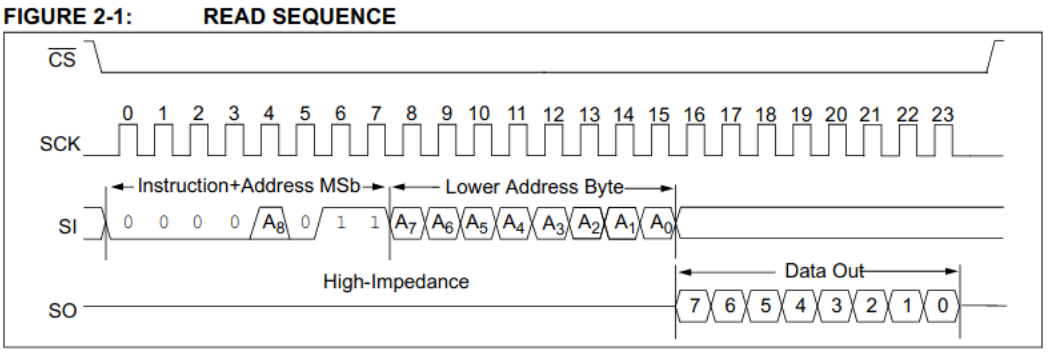

Figure 10 - EEPROM Read Documentation

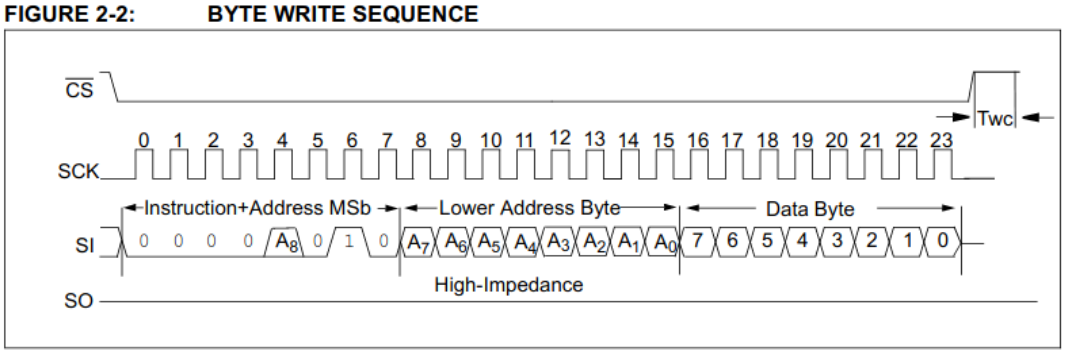

Figure 11 - EEPROM Write Documentation

2.2.4. Reversing Behavior

2.2.4.1. Reading Set Points

The behavior illustrated in figure 12 shows three different registers that are read from the EEPROM approximately every 7s and contain the set point values.

The values from three different addresses are being requested, the 0x43 address saves the right side set point, the 0x44 saves the left side set point, and the 0x45 saves the average set point with an interval of 1.875ms between the individual readings.

This statement has been achieved by verifying if the output of these registers always show the set-point information, by changing them to different values, associating them with the different sides and checking if these addresses keep the correct values.

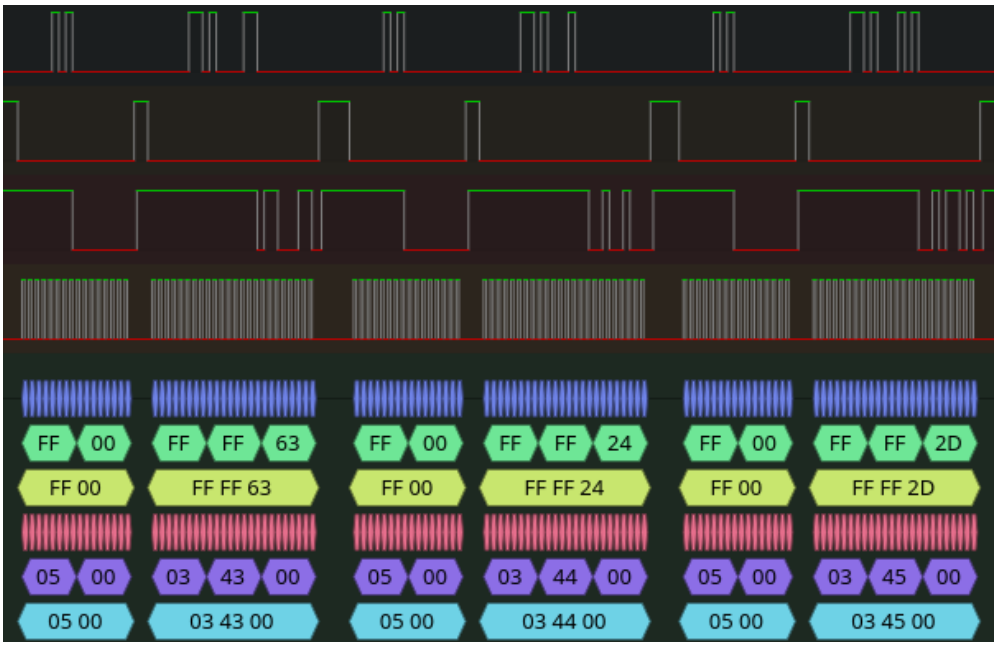

Figure 12 - SPI Signal 2

2.2.4.2. Writing Set Points

This behavior is directly associated with the one mentioned in the previous subsection (2.2.4.1. Reading Set Points), as the set point readings must be written into memory first.

The set points are stores into memory when the user presses the save button on the microcontroller following the approach illustrated in figure 12:

- First the read status register operation is executed to check if the microcontroller is available.
- After the writing latch is enabled (instruction 0x06) and the write operation is executed with the value from the potentiometer (from 35 to 99) to the address of the corresponding set point.
- The MOSI pin output stays 1 until the next operation.

In figure 13, the value of the right side set point was updated to 73 (0x49).

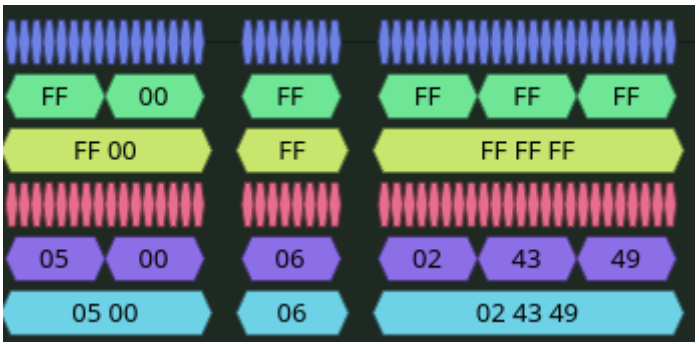

Figure 13 - SPI Signal 3

2.2.4.3. Reading Temperature History

The EEPROM stores the 30 temperature history readings for each of the temperature sensors (60 measurements in total).

This functionality is responsible for reading the full temperature history, every 7.2s and works in a circular buffer manner where the addresses range from 0x05 to 0x40 have the temperature readings. All of the steps required are the following:

- First the 0x42 and 0x41 addresses are read (figure 14 first 2 reads). The 0x42 address stores the value 60, which is the total length of the circular buffer and remains unchanged. The 0x41 address stores the next position of the circular buffer out of the 60 possible.
- After, as the offset between the next position (read from 0x41) and the address of next index of the buffer is 0x05 (as the buffer address range is from 0x05 to 0x40), all 60 addresses are read starting in the value read from 0x41 + offset:

$$Buffer\ address\ range\colon\ 0x05..\ 0x40$$
$$V(0x41)\ =\ 0..\ 59\ \%\ 60$$
$$Buffer\ head\ =\ V(0x41)\ +\ 0x05$$

In the case of figure 14, V(0x41) = 0x1C, by adding the 0x05 offset, the head of the buffer is 0x1C + 0x05 = 0x21, which is the first address that is read.

The last address to be read is 0x20, which makes sense, as:

$$(V(0x41) + 59) \,\%\, 60 + offset = \\ = (0x1C + 59) \,\%\, 60 + 0x05 \\ = 0x20$$

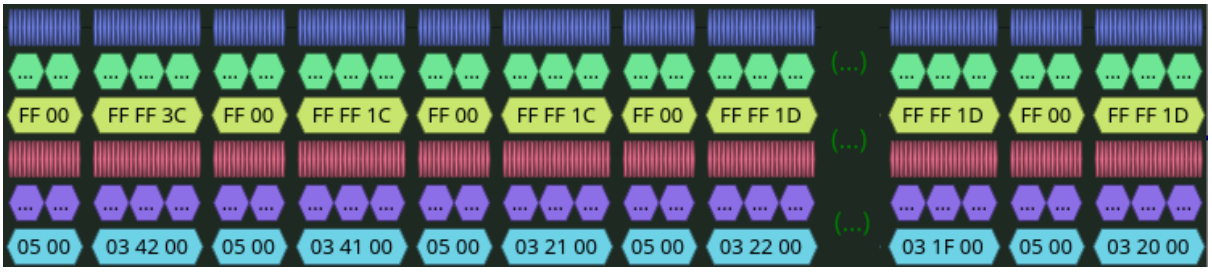

Figure 14 - SPI Addresses

2.2.4.4. Writing Temperature History

This behavior is directly associated with the one mentioned in the previous subsection (2.2.4.3. Reading Temperature History), as the temperate history readings must be written into memory first.

The temperature readings are stored in memory using the same circular buffer mentioned in the previous subsection every 3.6s following a similar approach as illustrated by figure 16:

- First the 0x42 and 0x41 addresses are read. As previously mentioned, the 0x42 address stores the buffer size (60) and the 0x41 address stores the next index of the buffer.
- After the enable write latch operation is executed (0x6) and the read status register instruction is executed until it returns a 0x0 response, meaning the write operation finished.
- After the temperature of the right side is stored, the same procedure is executed for the left side when the latter finishes.
- Lastly, the value of the 0x41 register is added with its original value + 2, as 2 measurements were added to the history.

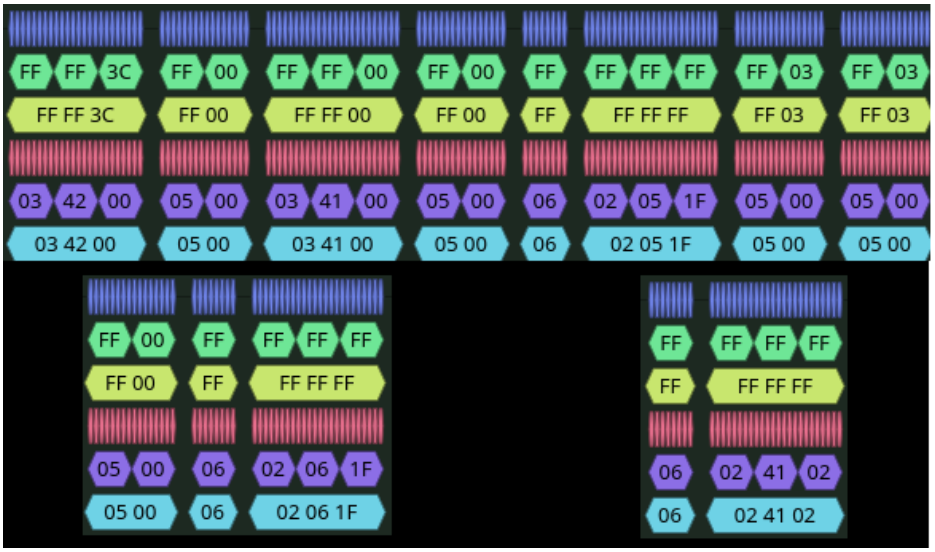

Figure 16 - SPI Addresses 2

2.2.4.5. Mode association

In normal mode (when either Dip3, Dip4 or both are on) these mentioned behaviors can be observed in the following repeating sequence:

1. Reading Temperature History (2.2.4.3. Reading Temperature History)
2. Wait 124 ms
3. Reading Set Points (2.2.4.1. Reading Set Points)
4. Wait 2756 ms
5. Writing Temperature History (2.2.4.4. Writing Temperature History)
6. Wait 1392 ms

In set point mode (when either Dip1, Dip2 or both are on) the behavior is observed in the following repeating sequence, when the save button is not pressed:

1. Reading Temperature History (2.2.4.3. Reading Temperature History)
2. Wait 7 s

When in set point mode, the button is pressed, the Writing Set Points (2.2.4.2. Writing Set Points) behavior occurs during the repeating sequence mentioned previously.

2.2.5. Memory Address Table

Address (Range)	Description
0x43	Right Side Set Point
0x44	Left Side Set Point
0x45	Average Set Point
0x05-0x40	Circular buffer
0x41	Circular buffer next index
0x42	Circular buffer size

2.3. I2C (Temperature Sensor)

2.3.1. Mapping Pins

The I2C communication protocol uses 2 pins to allow data to be read or written to the slaves by the microcontroler. The SCL (Serial clock Line) is responsible for transmitting the clock signal and SDA (Serial Data Line) is the pin responsible for transmitting the data between the master (microcontroller) and the slaves (the temperature sensors).

By analyzing the produced signal, it was possible to assume that, due to its signal frequency and behavior, INT4 is the SCL, represented in the next figures as D2, and OC5 is the SDA, represented as D0.

Figure 17 - I2C signal

2.3.2. Configuration Parameters

Given the analysis made by the PulseView program, the minimal frequency found for the clock cycle was 83.28 kHz, as shown in the figure 18.
This means that the baudrate is also 83.28k.

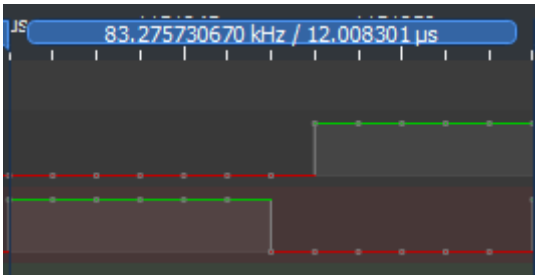

Figure 18 -I2C clock

2.3.3. Reversing Behavior

2.3.3.1. Read temperatures from sensors

Starting from the premise that D0 corresponds to the SDA and D2 is the SCL, both signals start at '1' value. When the SDA signal wants to read or write information, its value changes to '0', making it the "Start Condition". After that, the SCL signal starts to work as intended, making clock pulses. In the next 7 clock cycles, SDA transmits the address bits in which it wants to execute the action (Read/Write) and the 8th bit is the instruction, if it is the Read or Write action.

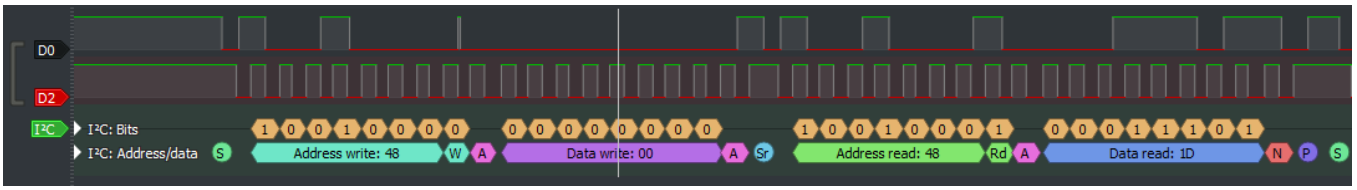

Figure 18 -I2C frame

According to the datasheet of the temperature sensors and to the previous figure, this flow corresponds to the Read Byte Format, as shown below.

Read Byte Format

S	Address	WR	ACK	Command	ACK	S	Address	RD	ACK	Data	NACK	P
	7 Bits			8 Bits			7 Bits			8 Bits		

Slave Address

Command Byte: selects which register you are reading from.

Slave Address: repeated due to change in data-flow direction.

Data Byte: reads from the register set by the command byte.

Figure 19 -Read byte Format

In order to read information from the sensors, firstly, it issues a Command Byte, corresponding to the first Data Write visible in Figure 18. Looking through the datasheet, the '00' as Data Write means that it wants to read values from the temperature sensor, as explained in Figure 20.

Command	Code	Function
RTR	00h	Read Temperature (TEMP)

Figure 20 - Command Code

After all this, it is possible to state that the master is issuing an order to read the temperature sensor (0x48) and, sequentially, an exact same request is made to the next temperature sensor (0x4C).

Figure 21 - I2C frame 2

This order is issued every ~1 second, as measured in the Figure below.

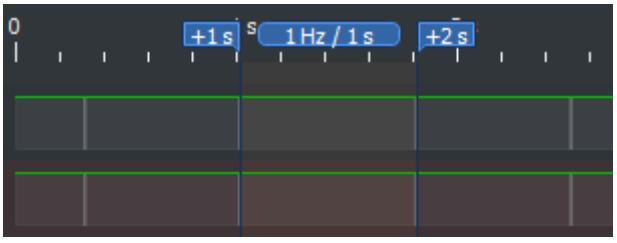

Figure 22 - I2C interval

According to the datasheet, the sensor TC74A0-3.3VCT can be associated with the bit sequence ‘1001000’ or ‘0x48’ in hexadecimal. The next sensor, TC74A5-3.3V has the value ‘1001101’ in its sheet corresponding to ‘0x4D’, not matching with the Figure 21. There are two possible alternatives, either the provided specs are misguided and instead of the TC74A5-3.3V sensor, it is the TC74A4-3.3V, or, it is possible to change this address since there is an ‘*’ in front of its address that states “Default Address”.

SOT-23 (V)	Address
TC74A0-3.3VCT	1001 000
TC74A1-3.3VCT	1001 001
TC74A2-3.3VCT	1001 010
TC74A3-3.3VCT	1001 011
TC74A4-3.3VCT	1001 100
TC74A5-3.3VCT	1001 101*
TC74A6-3.3VCT	1001 110
TC74A7-3.3VCT	1001 111

Note: * Default Address

Figure 23 - Sensor addresses

2.3.3.2. Mode association

When analyzing in which modes does the I2C create any type of signal, it was possible to observe that only when DIP3, DIP4 or both are turned ON, it was possible to see a signal being transmitted.

In the set point mode, no output from the I2C was observed.

2.4. LEDs

2.4.1. Reversing Behavior

2.4.1.1. Initial behavior

The first observation of the LED behavior was the correlation with the Dip switches. that resulted in the following behavior:

- With the Dip1 switch on the LED0 turns on.
- With the Dip2 switch on the LED7 turns on.
- With the Dip1 and Dip2 switches on, both LED7 and LED0 turn on.
- With the Dip3 switch on the LED3 turns on.

- With the Dip4 switch on the LED4 turns on.
- With the Dip3 and Dip4 switches on, both LED3 and LED4 turn on.
- With every Dip off, the LED5 and LED2 turn on.

The next step was to generate different offsets between the temperature and the corresponding set point, to verify its behavior. When the Dip3 is on (right side selected), by decreasing the offset, LED3 up to LED0 start to lighting up in a sequential manner. When this offset was increased, they started to turn off in the same sequential manner, leaving only LED3 on.

This behavior was also noticed when the left side was selected, but instead of lighting up LED3 up to LED0, the sequence was LED4 up to LED7, with the LED4 always on.

When both Dip3 and Dip4 are on, both behaviors explained for the left and right side occur at the same time, but this time both offsets were in relation to the average set point.

This indicates that there might be a correlation between the offset and the leds that light up. This behavior is further explained in the next subsection.

2.4.1.2. Temperature Offset and LEDs correlation

Considering the behavior experienced in the last subsection 2.4.1.1. Initial behavior, the relation (or mathematical expression) must be discovered for each LED.

Considering that the left or right chamber is active, the steps used were the following:

- First some data points were collected in order to possibly identify a pattern (Table 1). The temperature interval is the interval between the LEDs lighting up, one by one.
- These values show that the temperature interval between the LEDs lighting up depends on the set point. Starting on the lowest set point (35) this interval starts at 0 and increases by 1 every 4 sequential set points. This correlates to the equation:

$$temp_{interval} = round((set\ point - 35)/4)$$

- Another correlation can be observed, when the temperature is higher than the sum of the temperature interval with the base set point (35) the first LED lights up (2 or 5):

$$if\ (round((setPoint - 35)\ /\ 4\ ))\ +\ 35\ <=\ temp:$$
$$led\_2\ on$$

- For the next LED's (1,0 or 7,6) as the temperature offset is the same for all LED's, by simply multiplying the temperature interval by 2 or 3, the previous formula can be reformulated for the third and fourth LEDs:

$$if\ (round((setPoint - 35)\ /\ 4\ ))\ *\ 2\ +\ 35\ <=\ temp:$$
$$led\_3\ on$$
$$if\ (round((setPoint - 35)\ /\ 4\ ))\ *\ 3\ +\ 35\ <=\ temp:$$
$$led\_4\ on$$

When both chambers are active (Dip3 and Dip4) the same behavior was observed with the exact same process as explained for each chamber, but at the same time and in relation to the average set point.

Temperature	Set point	Number LEDs On	Temperature Interval
36 36 36 37 37 37	35 35 35 36 36 36	4 3 2 4 3 2	36 - 36 = 0 36 - 36 = 0 37 - 37 = 0 37 - 37 = 0

38	37	4	38 - 37 = 1
37	37	3	37 - 36 = 1
36	37	2	
53	59	4	53 - 47 = 6
47	59	3	47 - 41 = 6
41	59	2	
53	60	4	53 - 47 = 6
47	60	3	47 - 41 = 6
41	60	2	
56	61	4	56 - 49 = 7
49	61	3	49 - 42 = 7
42	61	2	
56	62	4	56 - 49 = 7
49	62	3	49 - 42 = 7
42	62	2	
56	63	4	56 - 49 = 7
49	63	3	49 - 42 = 7
42	63	2	
56	64	4	56 - 49 = 7
49	64	3	49 - 42 = 7
42	64	2	
59	65	4	59 - 51 = 8
51	65	3	51 - 43 = 8
43	65	2	
59	66	4	59 - 51 = 8
51	66	3	51 - 43 = 8
43	66	2	

Table 1 - Measurements

2.4.2. Reversing code (Real behavior)

2.4.2.1. Initial behavior

The initial behavior explained in 2.4.1.1. Initial behavior can be proven with the reversed code on figure 24:

```
if   DIP_SWITCH  == 0) {
  _LEDS_LAT = _LEDS_LAT & 0xff00 | 0x24;
}
else {
  len_temp_hist = get_potenciometer();
  LEDS_LAT_1 = (len_temp_hist * 0x40 + 0x1ff) / 0x3ff + 0x23;
             /* Value for set point */
  uGpffff8040 = display(LEDS_LAT_1 & 0xff);
             /* DIP 1,2 our both */
  if ( DIP_SWITCH  == 1) {
    _LEDS_LAT = _LEDS_LAT & 0xff00 | 1;
    uGpffff8018 = LEDS_LAT_1;
  }
  if ( DIP_SWITCH  == 2) {
    _LEDS_LAT = _LEDS_LAT & 0xff00 | 0x80;
    uGpffff8014 = LEDS_LAT_1;
  }
  if ( DIP_SWITCH  == 3) {
    _LEDS_LAT = _LEDS_LAT & 0xff00 | 0x81;
    uGpffff8010 = LEDS_LAT_1;
  }
}
```

Figure 24 - Code snippet

This board has only 7 LEDs, which means that the _LEDS_LAT register first 7 bits, when set to 1, light up the corresponding LED.

In case the DIP_SWITCH is 0 (all Dip's off), first the first 7 bits are cleared with the "& 0xff00" and the value 0x24 (0x24 = 00100100) is set using the or operation (| 0x24).

In case the DIP_SWITCH is 1 (Dip1 is on), the process repeats, but with the 1 (1 = 00000001) set.

In case the DIP_SWITCH is 2 (Dip2 is on), the process repeats, but with the 0x80 (0x80 = 10000000) set.

In case the DIP_SWITCH is 3 (Dip1 and Dip2 are on), the process repeats, but with the 0x81 (0x81 = 10000001) set.

The situations when either the DIP3 or DIP4 are on will be covered in the next subsection.

2.4.2.2. Temperature Offset and LEDs correlation

The behavior explained in 2.4.1.2. Temperature Offset and LEDs correlation is an accurate representation of the correlation behavior, but when reversing the microcontroller elf, the real mathematical formula used is:

$$offset = floor((setPoint - 35) * 2 + 4) / 8)$$
$$led = floor((temp - 35) / offset + 1)$$

If the led result is 2 the first and second LED will light up, if the result is 3 the first, second and third LED will light up and if the result is 4 the first, second, third and fourth LED will light up.

It's important to note that this microcontroller performs integer divisions with remainder. In this case only the quotient is taken into account, which is the same thing as rounding down (floor).

To convert the led value to the value that is going to be set in the register, for the right side and for the left side:

$$right_{side} = (1 << led) - 1$$
$$left_{side} = 16 - (16 >> led)$$

This formula is present on the following figure of reversed code:

```
if (temp < 0x23) {
  temp = 0x23;
}
                    /* 15 */
var1 = (1 << (4 & 0x1f)) + -1;
if (temp <= set_point) {
  var1 = (int)((set_point - 0x23) * 2 + 4) / (int)(4 << 1);
  if (4 << 1 == 0) {
    trap(7);
  }
  16 = 1 << (4 & 0x1f);
  if (var1 == 0) {
    var1 = 1;
  }
  if (var1 == 0) {
    trap(7);
  }
  uVar1 = (temp - 0x23) / var1 + 1;
  if ((int)uVar1 <= (int)4) {
    4 = uVar1;
  }
  var1 = (1 << (4 & 0x1f)) + -1;
  if (chamber(0,1) != 1) {
    var1 = 16 - (16 >> (4 & 0x1f));
  }
}
return var1;
```

Figure 25 - Code snippet 2

2.5. OCs

2.5.1. Mapping Chambers

To match each OC pin with the corresponding chamber side, it's relatively simple. As explained in the original documentation of this project, when either Dip4 (left side) or Dip3 (right side) is on, one of the OC's is used. By checking for any output on the OC pins with the different Dip combinations, it's possible to verify the following correspondence:

Left side -> OC2

Right side -> OC1

2.5.2. General Parameters

The only important parameter for the OC's behavior is the frequency at which it operates.

By measuring a full time period (On time + Off time), the frequency can be calculated using the following formula:

$$f = \frac{1}{Full\ Time\ Period}$$

$$f = 140.84\ Hz$$

2.5.3. Reversing Behavior

2.5.3.1. OCs, Set-point and temperature relation

As explained in the original project documentation, the OCs are used to increase or decrease the temperature emission for each side of the chamber.

This means that a relation between the OCs behavior and the distance from the current temperature to the corresponding set-point might exist.

The first step of this process was to obtain a few data points from PulseView:

Temperature	Set Point	Distance	PWM
69	70	1	4
68	70	2	9
67	70	3	13
66	70	4	18
65	70	5	22
64	70	6	27
63	70	7	31
62	70	8	36
61	70	9	40
60	70	10	45

Table 2 - Measurements 2

Based on these data points, a correlation is evident:

- When the distance between the current temperature and the set point is even, the PWM is a multiple of 9.
- When this distance is odd, a similar behavior can be observed, the PWM of the previous odd distance is increased by 9.

By using a mathematical sequence analyser for each case, the PWM signals can be calculated with the following formulas:

$$if\ (temp\ -\ setPoint)\ \%\ 2\ ==\ 0:$$
$$PWM\ (\%)\ =\ \frac{9\ *(temp - setPoint)}{2}$$
$$else:$$
$$PWM\ (\%)\ =\ \frac{9\ *(temp - setPoint) - 1}{2}$$

When the PWM % is greater than 100, the OCs signal will remain a 1 (full throttle), in other words, when the distance is greater than 22, the OCs will output a signal always at 1.

This information is in agreement with the original project documentation as the OCs control a heating element that tries to reach the set point temperature and the PWM will increase and decrease accordingly.

2.5.3.2. Mode association

As mentioned in the original documentation and confirmed by analyzing the OCs pins, the behavior stated in 2.5.3.1. OCs, Set-point and temperature relation is observed in the normal mode, when either the Dip3, Dip4 or both switches are turned on.

In the set point mode, no output from the OCs was observed.

2.5.4. Reversing code (Real behavior)

2.5.4.1. OCs, Set-point and temperature relation

The OCs, Set-point and temperature relation behavior explained in the subsection 2.5.3.1. OCs, Set-point and temperature relation can be proven with the reversed code on figure 26, 27 and 28.

```
if (DIP_SWITCH == 1) {
  uGpffff8040 = display(temp & 0xff);
  OCs((int)((Set_point_right - temp) * 0x2d) / 10,0);
```

Figure 26 - Code snippet 3

```
else if ( DIP_SWITCH == 2) {
  uGpffff8040 = display(temp2 & 0xff);
  OCs(0,(int)(( Set_point_left - temp2) * 0x2d) / 10);
```

Figure 27 - Code snippet 4

```
else if ( DIP_SWITCH == 3) {
  uGpffff8040 = display(LEDS_LAT_1 & 0xff);
  OCs((int)((Set_point_avg - LEDS_LAT_1) * 0x2d) / 10,
      (int)((Set_point_avg - LEDS_LAT_1) * 0x2d) / 10);
```

Figure 28 - Code snippet 5

The OCs function is used to set the PWM value for each of the OCs. The first argument of the function is the right side PWM and the second argument the left side PWM.

According to the DIP_SWITCH variable which indicates the state of the Dip3 and Dip4 switches, either the right or left side OC will output according to the following formula:

$$PWM\ (\%) = floor((SetPoint - temp) * 0x2d / 10)$$
$$= floor((SetPoint - temp) / 4.5)$$

This formula can be seen in figure 26, 27 and 28.

When both Dip3 and Dip4 switches are on, the DIP_SWITCH value is 3 and the calculation is performed of each OC in relation to the average setpoint (figure 28).

It's important to note that this microcontroller performs integer divisions with remainder. In this case only the quotient is taken into account, which is the same thing as rounding down (floor).

```
void OCs(int percent_side_1,int percent_side_2)

{
  if (100 < percent_side_1) {
    percent_side_1 = 100;
  }
  if (percent_side_1 < 0) {
    percent_side_1 = 0;
  }
  if (100 < percent_side_2) {
    percent_side_2 = 100;
  }
  if (percent_side_2 < 0) {
    percent_side_2 = 0;
  }
  _DAT_bf803020 = (uint)(_DAT_bf800820 * percent_side_1 + percent_side_1)
  _DAT_bf803220 = (uint)(_DAT_bf800820 * percent_side_2 + percent_side_2)
  return;
}
```

Figure 29 - Code snippet 6

3. Microcontroller Components Interaction

<table>
<tr><th colspan="3">Interaction #1 - OCs</th></tr>
<tr><td colspan="3">Components/Protocols:
• OCs • SPI(EEPROM) • I2C (Temperature Sensor)</td></tr>
<tr><td colspan="3">Related Behavior:

<table>
<tr><th>OCs</th><th>SPI (EEPROM)</th><th>I2C (Temperature Sensor)</th></tr>
<tr><td>• OCs, Set-point and temperature relation (2.5.3.1. OCs, Set-point and temperature relation)</td><td>• Reading Set Points (2.2.4.1. Reading Set Points)</td><td>• Read temperatures from sensors (2.3.3.1. Read temperatures from sensors)</td></tr>
</table>
</td></tr>
<tr><td colspan="3">Possible Interaction:

As explained in the OCs behavior section, both the corresponding set point and the temperatures of the same chamber are needed in order to calculate the PWM % output. This information is probably provided by the reading set point behavior of the EEPROM and by the temperature sensor readings.</td></tr>
</table>

Interaction #2 - LEDs

Components/Protocols:

- LEDs
- SPI(EEPROM)
- I2C (Temperature Sensor)

Related Behavior:

LEDs	SPI (EEPROM)	I2C (Temperature Sensor)
• Temperature Offset and LEDs correlation (2.4.1.2. Temperature Offset and LEDs correlation)	• Reading Set Points (2.2.4.1. Reading Set Points)	• Read temperatures from sensors (2.3.3.1. Read temperatures from sensors)

Possible Interaction:

As explained in the LEDs behavior section, both the corresponding set point and the temperatures of the same chamber are needed in order to calculate which LEDs should light up and when. This information is probably provided by the reading set point behavior of the EEPROM and by the temperature sensor readings.

Interaction #3 - SPI (EEPROM)
Components/Protocols: ● SPI(EEPROM) ● I2C (Temperature Sensor)
Related Behavior:

SPI (EEPROM)	I2C (Temperature Sensor)
● Writing Temperature History (2.2.4.4. Writing Temperature History)	● Read temperatures from sensors (2.3.3.1. Read temperatures from sensors)

Possible Interaction:

When the temperature history is written into memory, the readings from the temperature sensors are used, as they are written into memory.

Interaction #4 - RS232C

Components/Protocols:

- SPI(EEPROM)
- RS232C (UART)

Related Behavior:

SPI (EEPROM)	RS232C (UART)
• Writing Temperature History (2.2.4.4. Writing Temperature History) • Reading Temperature History (2.2.4.3. Reading Temperature History)	• Debug Message (2.1.3.2. Debug Message)

Possible Interaction:

As the temperature history is written into memory, the history readings are being transmitted from the Tx pin of the RS232C.