POLITECNICO DI BARI

DEI

LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

Embedded Control

Prof. Ing. Luca De Cicco

Progetto:

Peltier Cooler with tempearature control

Studenti:
Marcello Bari
Alessandro Mitola

ANNO ACCADEMICO 2023-2024

1 Introduction

The aim of this project is the creation of a Peltier Cooler, that must manage to maintain a constant cold temperature inside a small box with the use of a temperature controller.
As the name suggests, the cooling action of this system comes from a Peltier cell, which is a solid-state device that creates an electrical voltage from a temperature gradient (Seebeck effect) or converts an electrical voltage into a temperature gradient (Peltier effect).

Figure 1: The TEC1-12706 cell

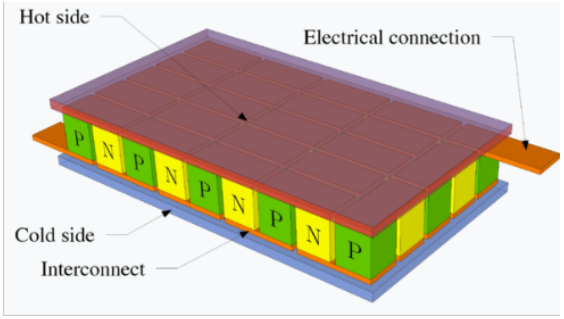


Figure 2: Peltier cell scheme

It can behave like this because of the metal-semiconductor junctions (both n-type and p-type pellets are used to increase heat exchange) positioned in series with respect to one another, as is shown in the following image.

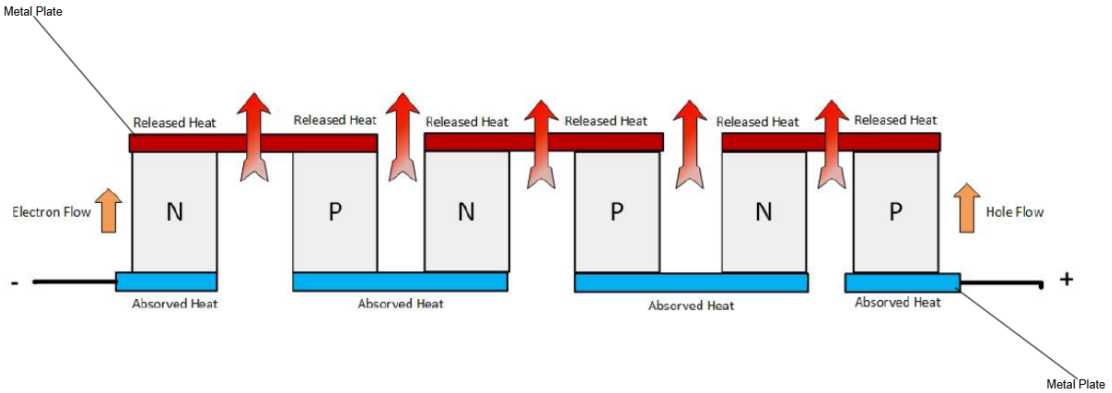


Figure 3: Internal structure of a Peltier cell

This type of structure causes one of the sides of the cell to become colder, as the other progressively heats up. This means that a Peltier cell, as it is, cannot be employed as a refrigerator. It needs to be paired with a heat dissipation system, in order for it to be used for our purposes. We adopted a heat sink connected to a DC fan, as most vendors sell them all in one package.
Typically, Peltier cells can operate with a 12-16 V supply voltage and can draw up 4 to 5 A. Therefore, one of the main challenges we faced was coming up with ways to interface the STM32 board with the device, without causing significant damage to the circuit. We will return to this problem later in this text.
The creation of the circuitry and the firmware implementation of the thermal regulator was the other major aspect of this project, where we especially focused on allowing the user to manually

set the reference temperature that the system should reach, with the use of a potentiometer, and to see directly the thermal conditions of the box with the help of a lcd screen.
During the course of this report, we will describe all the phases of our work in chronological order. Each stage will be detailed to provide a clear understanding of the progression, from the initial planning to the final implementation and evaluation, as well as all our failed attempts, if any.
To start, the very first thing we did was to create a reference schematic outline of all the elements we believed were necessary to carry out the project.

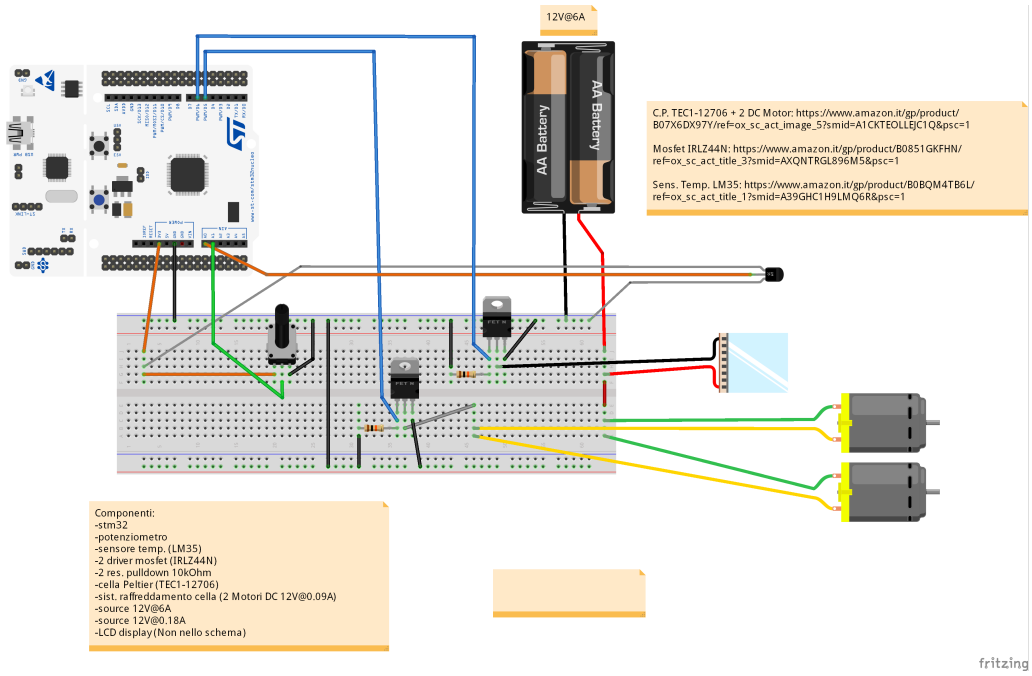


Figure 4: Starting circuit

This, of course, is not the final circuit diagram, as we changed it to better fit our objectives. The complete list of hardware we ultimately purchased and used is the following:

- 1 × STM32F446RET6 board
- 3 × Solderless boards
- 1 × package of jumper wires
- 1 × package of resistors
- 1 × package of inductors
- 1 × package of capacitors
- 1 × 12V 7Ah battery
- × Switching power source
- 1 × DS18B20 Digital temperature sensor
- 1 × 10 kΩ potentiometer

- 3 × IRLZ44N Mosfet transistors
- 1 × 2N2222A-1726 BJT transistor
- 1 × array of screw-terminals
- 1 × WiMas cooling kit
- 1 × lcd display with PCF8547T I2C expansion module

2 Sensor: Configuration and Implementation

The DS18B20 is a digital temperature sensor manufactured by Maxim Integrated and widely used in various applications due to its high precision (it can measure temperatures ranging from -55°C to +125°C with an accuracy of ±0.5°C over the range of -10°C to +85°C), ease of use, and cost-effectiveness. However, we chose this sensor mainly because it was the one that could ship to us the fastest, making it the most convenient option for our project timeline.

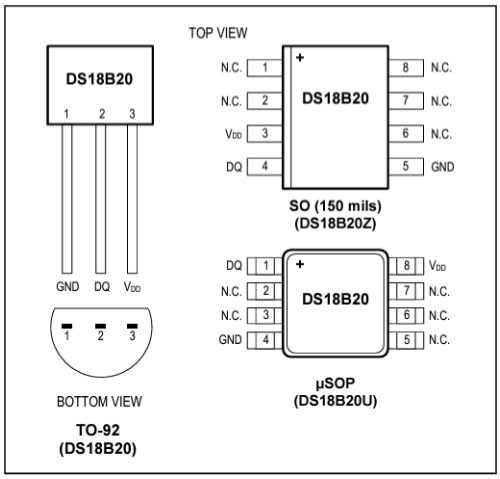


Figure 5: DS18B20 Sensor pin configuration

It can send data to the external world with the 1-Wire communication protocol; a low-speed, low-power protocol, which allows for simple connectivity and reduces the need for extensive wiring, as it only needs one wire to operate. The only problem is that the STM32F446RET6 board does not natively support this protocol (none of the 64 pins can be configured as 1-Wire channels, unlike other protocols like I2C, SPI, etc.). Therefore, to overcome this limitation, we searched for another communication framework that can work with just one wire and that can be easily interfaced with the MCU. We ended up choosing the USART in "Single Wire (Half Duplex Mode)", that meets all the needed specifications. From a register-level perspective, activating this mode means:

- Setting the HDSEL bit in the USART_CR3 register.
- Clearing the LINEN and CLKEN bits in the USART_CR2 register.
- Clearing the SCEN and IREN bits in the USART_CR3 register.

As soon as HDSEL is written to 1:

- The TX and RX lines are internally connected.

- The RX pin is no longer used.
- The TX pin acts as a standard I/O in idle or in reception. It means that the I/O must be configured so that TX is configured as floating input (or output high open-drain) when not driven by the USART.

So, through the CubeMX tool we activated the USART1 peripheral (PA9 pin), configured its initial settings as shown in figure 6, and set up the clock to be the High Speed External (HSE) Crystal/Ceramic resonator clock. In the "Clock configuration" tab we made sure that the SYSCLK is at 180 MHz and the APB2 peripheral clock is at its maximum value of 90 MHz (APB2 is the bus connected to the USART1 peripheral).

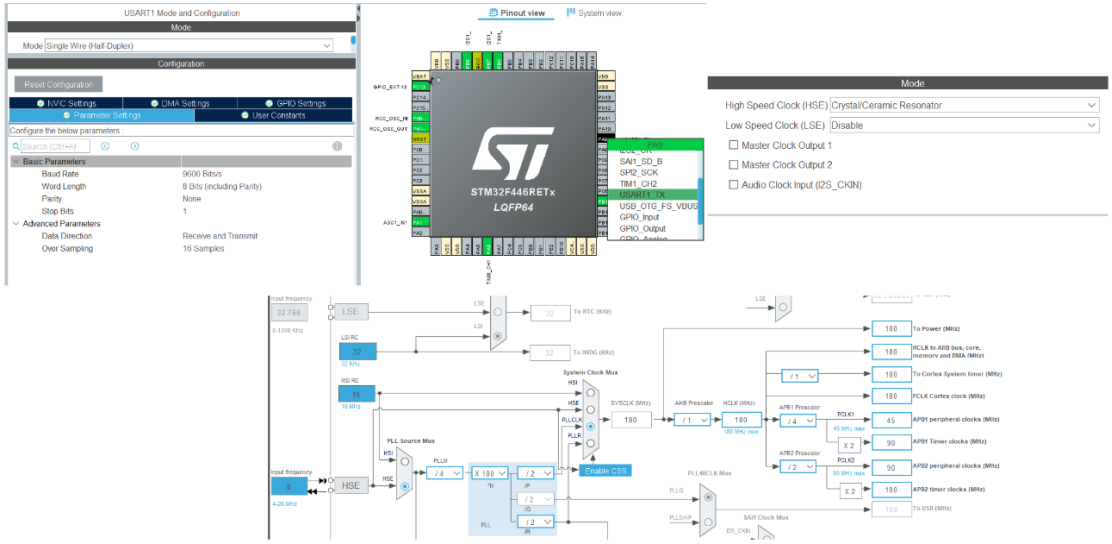


Figure 6: USART1 CubeMX configuration

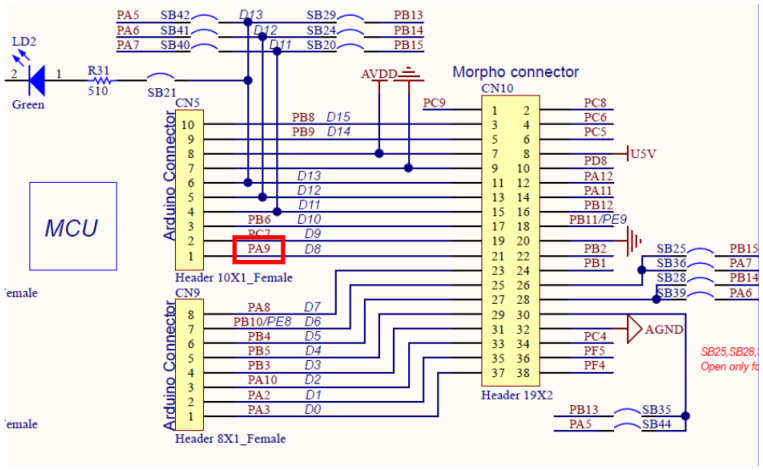


Figure 7: PA9 pin location

In order to understand our choice for the parameter settings of USART1, we first need to introduce the timings and the functioning of the 1-Wire protocol.

2.1 The 1-Wire protocol

This framework demands that there must always be a single bus master to control one or more slave devices (the DS18B20 is always a slave). Each device (master or slave) interfaces the data line via an open-drain or 3-state port (that is why we set the GPIO Mode of the PA9 pin is Alternate Function open drain). This allows each device to 'release' the data line when not transmitting data so that the bus is available for use by another device (the idle state of the 1-Wire bus is high).

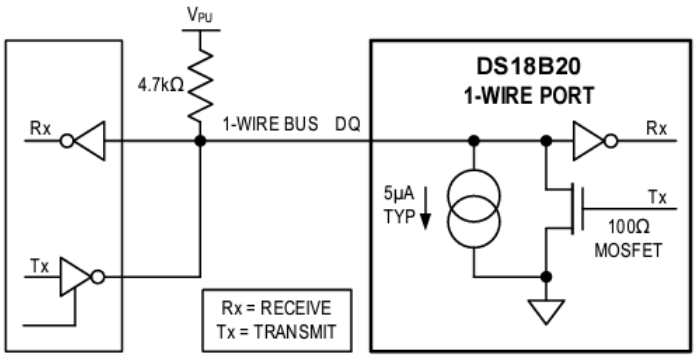


Figure 8: DS18B20 Sensor bus scheme

The DS18B20 datasheet states that the pull-up resistor's value must be approximately 5 kΩ; nevertheless, we also tried using the internal pull-up resistor of the MCU (40 kΩ) and it works just fine. The circuit we used for the sensor is basically the same one shown in figure 4 and is placed inside the box to cool.
Each time one of the slave needs to be accessed by the master, a 3-phased transaction must take place in the following order:

- Reset Sequence
- ROM CMD Sequence
- Function CMD Sequence

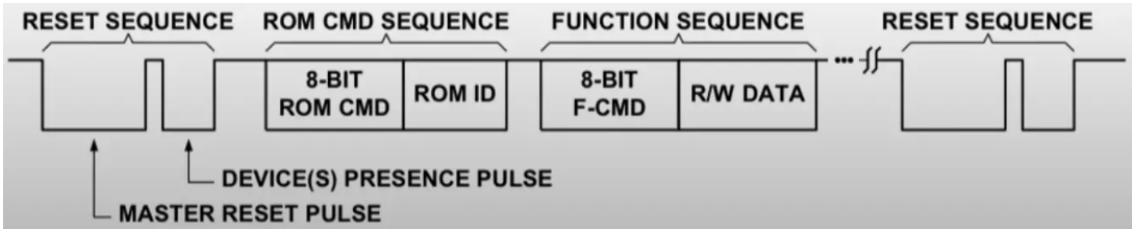


Figure 9: DS18B20 transaction scheme

The reset sequence is initiated by the master device, that pulls the data line low for a minimum of 480 μs before releasing it again. This is the Master Reset Pulse. If a device on the bus detects this pulse, it will respond by pulling the line low for 60-240 μs after the host releases it. This is called the Device Presence Pulse.

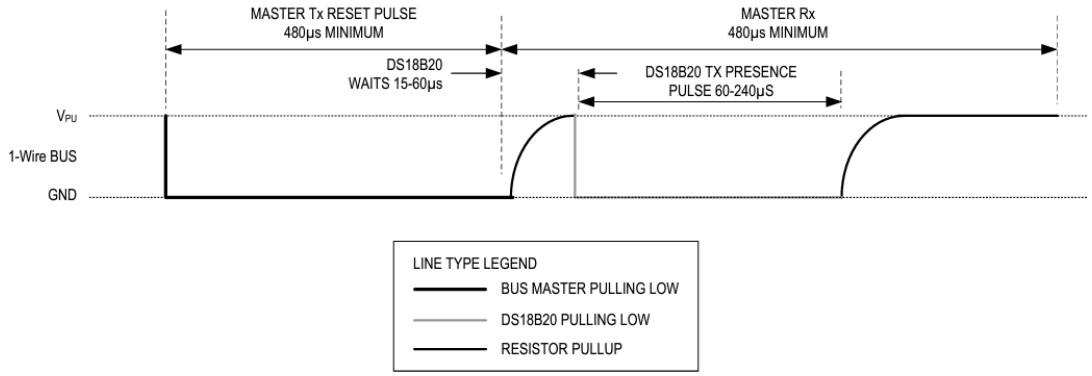


Figure 10: DS18B20 reset sequence

ROM commands are used to address devices on the bus. In our case, there is only one slave device, so we can issue a Skip ROM command (0xCC), which tells the master to address all slaves and go directly to the function sequence phase.
The Function command sequence is device-dependent and is used to issue specific commands across the bus. For the DS18B20 we will use just two of these types of codes: the F-CMD called "Convert T" (0x44) which is the instruction that makes the temperature conversion start, and the "Read Scratchpad" (0xBE) command which instructs the DS18B20 to read its scratchpad memory, that contains the temperature conversion results, and publish its contents on the bus. The 1-Wire protocol also establishes strict timing rules both for the reading and writing operations, which are summarized by the following schemes:

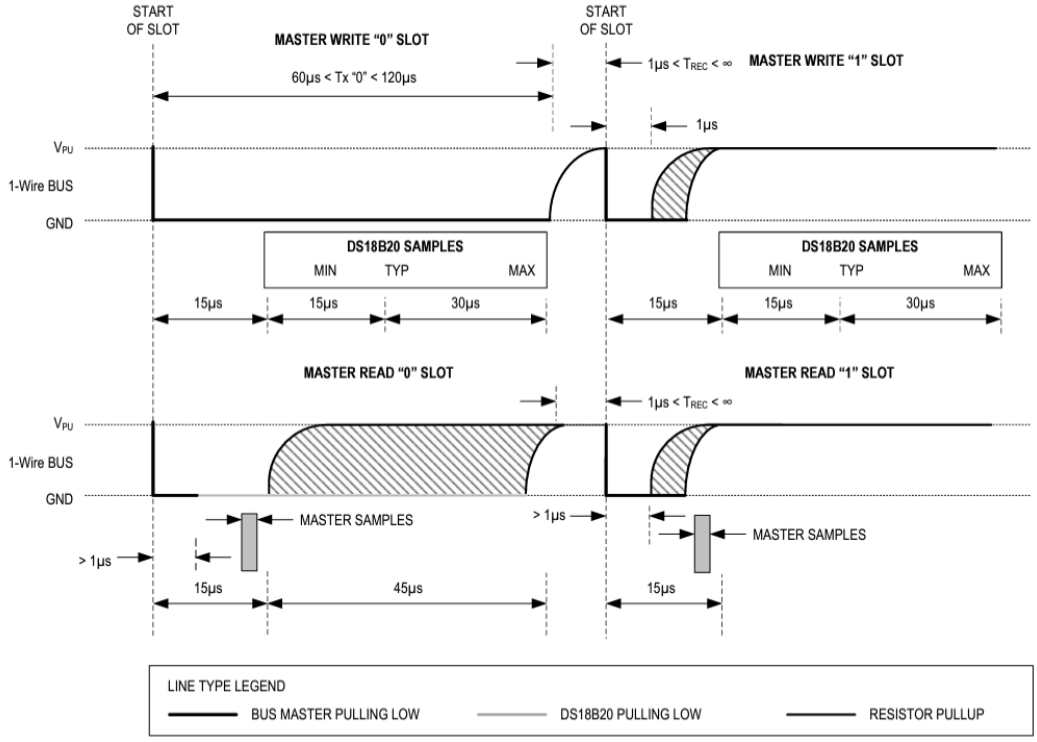


Figure 11: DS18B20 R/W operations

All write and read time slots must last a minimum of 60 μs with at least 1 μs of recovery time between each one of them. To write a "0" bit, the bus must transition from HIGH to LOW, stay LOW for at least 60 μs, and then be pulled up to HIGH again by the pull-up resistor. To write a "1" bit, the bus must transition from HIGH to LOW and be released back to HIGH within 15 μs. The DS18B20 always samples the data line during a window that lasts from 15 μs to 60 μs after the master initiates the write time slot. If the bus is high during the sampling window, a "1" is written. Consequently, if the line is low, a "0" is written.
The DS18B20 can only transmit data to the master when the master requests data. It does so by pulling the bus low for a minimum of 1 μs and then releasing it. The sensor responds by either pulling the line LOW ("0" bit) or leaving it HIGH ("1" bit). The response is valid for 15 μs after the falling edge that initiated the read time slot. Therefore, the master must release the bus and then sample the bus state within 15 μs from the start of the slot.

2.2 Code

Of course, the standard STM32 UART hardware in single-wire (half-duplex) mode is not ready to do 1-Wire transactions out of the box. We started coding the functions for writing and reading by considering a baud rate of 115200, so that each bit period is $\frac{1}{baudrate} \approx 8.68$ μs. This means that 60 μs, which is the amount of time that the bus needs to be LOW in order to write a "0", corresponds to around 7 UART bit periods. By setting the serial channel at 8 bits per frame, without parity, and by sending a 0x00 byte, we ensure that a "0" gets correctly transmitted to the sensor. On the other hand, when sending a 0xFF byte, the start bit keeps the line LOW for at least 8.68 μs, that is significantly higher than 1 μs but still acceptable as it falls under the 15 μs mark (the time at which the sensor starts sampling), and the subsequent bits pull the line HIGH and allow the sensor to correctly receive a "1" bit.
All the commands we use to address the DS18B20 are one byte long, so the function we ultimately wrote to handle bit transmission is the following:

```
1  void Sensor_WriteByte(UART_HandleTypeDef *huart, uint8_t data)
2  {
3      uint8_t TxBuffer[8];
4      for (int i=0; i<8; i++)
5      {
6        if ((data & (1<<i)) != 0){
7            TxBuffer[i] = 0xFF;
8        }
9        else{
10           TxBuffer[i] = 0;
11       }
12     }
13     HAL_UART_Transmit(huart, TxBuffer, 8 , 1);
14 }
```

It is a basic for loop that checks the individual bits of the byte that needs to be sent with the use of the left shift and bit wise and operator. The following image should clarify how each bit is isolated:

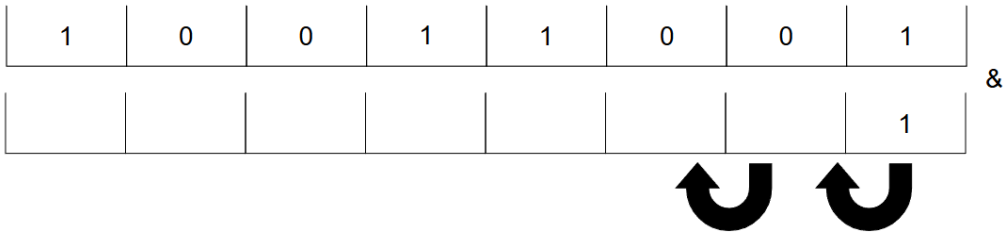


Figure 12: Bit checking of a byte

It then updates the values of the array "TxBuffer" accordingly, and sends it thanks to the HAL_UART_Transmit function. We use the USART in polling mode since strict timings are required, especially in reading operations as we will show soon.
To issue a reading, things are more convoluted. We have just 15 μs, after a LOW state to read the sensor's output. This means that ideally, we would want to send only the start bit of a UART frame and then start receiving. But frames are the smallest units of data that we can send through the bus, so we need to at least send 8 more bits after the initial one. Our first approach was to start the transmission in interrupt mode, send a 0xFF (after the start bit the line is kept HIGH) and immediately call a receive command in the following way:

```
uint8_t Sensor_ReadByte(UART_HandleTypeDef *huart)
{
    uint8_t RxBuffer[8];
    uint8_t HighBuffer[8];
    uint8_t RxByte = 0;
    for (uint8_t i = 0; i < 8; i++)
    {
        HighBuffer[i]=0xFF;
    }
    HAL_UART_Transmit_IT(huart, HighBuffer, 8);
    HAL_UART_Receive_IT(huart, RxBuffer, 8);
    while (isRead==0){};
    for (uint8_t i = 0; i < 8; i++)
    {
        if(RxBuffer[i] == 0xFF)
        {
            RxByte = (1<<i);
        }
    }
    isRead=0;
    return RxByte;
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart -> Instance  == USART1)
    {
           isRead = 1;
    }
}
```

The idea behind this is to make the reception of bytes start immediately after having sent the "HighBuffer" array and to check whether the bytes received are the same ones sent. If that's the case the sensor is writing a "1", otherwise a "0" (The RxByte $=(1 << i)$ line is needed beacuse

the 1-Wire protocol sends the LSB first).Unfortunately, this approach did not work, as the sensor produced unusual readings, although it did appear to respond to temperature changes. We also tried to use UART in DMA mode but the results were the same, as the cause of the issue seems to be timing related.
To find a solution, we searched for the code of the HAL_UART_Receive function in the stm32f4xx_hal_uart.c file. It all comes down to this line:

```
1  *pdata8bits = (uint8_t)(huart->Instance->DR & (uint8_t)0x00FF);
```

It writes the contents of the DR register, which holds transmitted or received bits, in the location referenced by the pdata8bits pointer (the second argument passed to this function).

25.6.2 Data register (USART_DR)

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	DR[8:0]								
							rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 13: The DR register

The STM32F446RE reference manual states that : "The Data register performs a double function (read and write) since it is composed of two registers, one for transmission (TDR) and one for reception (RDR)."

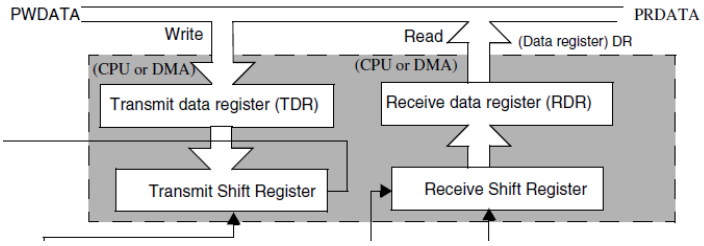


Figure 14: TDR and RDR registers

In our case, both the reading and transmission ports are connected. This means that, the contents of the RDR register will mirror the state of the bus during transmission. By exploiting this behavior we changed the code to be:

```
1  uint8_t Sensor_ReadBit(UART_HandleTypeDef *huart)
2  {
3      const uint8_t ReadBitCMD = 0xFF;
4      uint8_t RxByte;
5
6      // Manda sempre 1
7      HAL_UART_Transmit(huart, &ReadBitCMD, 1,1);
8      // Ricevi
```

```
9       HAL_UART_Receive(huart, &RxBit, 1 , 1);
10
11      return (RxByte & 0x01);
12  }
13
14  uint8_t Sensor_ReadByte(UART_HandleTypeDef *huart)
15  {
16      uint8_t RxByte = 0;
17      for (uint8_t i = 0; i < 8; i++)
18      {
19          RxByte >> = 1;
20          if (Sensor_ReadBit(huart))
21          {
22              RxByte |= 0x80;
23          }
24      }
25      return RxByte;
26  }
```

Now after sending a 0xFF byte in polling mode, the HAL_UART_Receive function is called and stores in RxByte the contents the DR register. The sensor's output value expires approximately when the second bit of the frame is sent, so we isolate only the first bit of the receive byte with a bitwise and. The Sensor_ReadByte function repeats this process for eight times. The if statement checks whether the bit written by the DS18B20 is a "1" and sets the first bit (from left) of the RxByte to "1". All the bits are shifted to the right at the beginning of each loop to adhere to the LSB logic. These two functions ended up working, making the use of DMA and interrupt mode unnecessary.
The only thing that remains to be coded to make the 1-Wire protocol work is the initialization sequence. As figure 10 shows, it requires to set the line LOW; something we have already coded when writing a "0". However, this time the problem lies in the amount of time that the line needs to remain in a "0" state, as 480 μs of low-level signaling is hard to achieve with a baud rate of 115200 (the stop bit of a frame always pulls up the line). It becomes much easier with a baud rate of 9600, as $\frac{1}{9600} \approx 104$ μs, and with just 5 "0" bits we achieve a time of 520 μs (the byte that must be sent is 0x$F0$). So, we need to ensure that the baud rate of the bus is constantly switching between 9600 and 115200 (the starting baud rate is 9600 as shown in figure 6) . In order to do that we need to change the USARTDIV encoded in the BRR register.

25.6.3 Baud rate register (USART_BRR)

Note: *The baud counters stop counting if the TE or RE bits are disabled respectively.*

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 15: BRR register

The value of USARTDIV, considering an oversampling rate of 16, is calculated as follows:
If $baudrate = 9600$

$$\frac{f_{pclk2}}{16 \times baudrate} = \frac{90 \times 10^6}{16 \times 9600} = 585.9375$$

If $baudrate = 115200$

$$\frac{f_{pclk2}}{16 \times baudrate} = \frac{90 \times 10^6}{16 \times 115200} = 48.828125$$

In order to format these numbers in a way that is acceptable for the BRR register, we used four macros defined in the stm32f4xx_hal_uart.h file:

```
1 #define UART_DIV_SAMPLING16(_PCLK_, _BAUD_)            ...
    ((uint32_t)((((uint64_t)(_PCLK_))*25U)/(4U*((uint64_t)(_BAUD_)))))
2 #define UART_DIVMANT_SAMPLING16(_PCLK_, _BAUD_)        ...
    (UART_DIV_SAMPLING16((_PCLK_), (_BAUD_))/100U)
3 #define UART_DIVFRAQ_SAMPLING16(_PCLK_, _BAUD_)        ...
    ((((UART_DIV_SAMPLING16((_PCLK_), (_BAUD_)) - ...
    (UART_DIVMANT_SAMPLING16((_PCLK_), (_BAUD_)) * 100U)) * 16U)\
4                                                           + 50U) / 100U)
5 /* UART BRR = mantissa + overflow + fraction
6             = (UART DIVMANT << 4) + (UART DIVFRAQ & 0xF0) + (UART ...
                DIVFRAQ & 0x0FU) */
7 #define UART_BRR_SAMPLING16(_PCLK_, _BAUD_)            ...
    ((UART_DIVMANT_SAMPLING16((_PCLK_), (_BAUD_)) << 4U) + ...
    \(UART_DIVFRAQ_SAMPLING16((_PCLK_), (_BAUD_)) & 0xF0U) + \
8    (UART_DIVFRAQ_SAMPLING16((_PCLK_), (_BAUD_)) & 0x0FU))
```

The one that will be called by our function is UART_BRR_SAMPLING16, and its parameters are the frequency of the APB2 clock and the baud rate. These macros are the equivalent of the examples reported in the STM32 documentation:

Example 3:

To program USARTDIV = 0d50.99

This leads to:

DIV_Fraction = 16*0d0.99 = 0d15.84

The nearest real number is 0d16 = 0x10 => overflow of DIV_frac[3:0] => carry must be added up to the mantissa

DIV_Mantissa = mantissa (0d50.990 + carry) = 0d51 = 0x33

Then, USART_BRR = 0x330 hence USARTDIV = 0d51.000

Figure 16: USARTDIV example

To better visualize the actual result of these codes, we wrote a small program in Programiz that mimics all of their operations:

```c
// Online C compiler to run C program online
#include <stdio.h>
#include <stdint.h>

int main() {
    // Write C code here
    uint64_t f=90000000;
    uint64_t b=115200;
    uint32_t DIV =(uint32_t) (f*25U)/(4U*b);
    uint32_t DIV_MANT=DIV/100U;
    uint32_t DIV_FRAQ = (((DIV-DIV_MANT*100U)*16)+50U)/100U;
    uint32_t BRR = (DIV_MANT<<4U) + (DIV_FRAQ & 0xF0U) + (DIV_FRAQ &
        0x0FU);
    printf ("DIV:%b\n",DIV);
    printf ("DIV:%u\n",DIV);
    printf ("DIV_MANT:%u\n",DIV_MANT);
    printf ("DIV_MANT:%b\n",DIV_MANT);
    printf ("DIV_FRAQ:%u\n",DIV_FRAQ);
    printf ("DIV_FRAQ:%b\n",DIV_FRAQ);
    printf ("BRR:%u\n",BRR);
    printf ("BRR:%b\n",BRR);
    return 0;
}
```

```
DIV:1001100010010
DIV:4882
DIV_MANT:48
DIV_MANT:110000
DIV_FRAQ:13
DIV_FRAQ:1101
BRR:781
BRR:1100001101


=== Code Execution Successful ===
```

Figure 17: Macros in Programiz

The actual initialization function we wrote in the main.c file is:

```c
uint8_t Sensor_Init(UART_HandleTypeDef *huart)
{
    const uint8_t ResetByte = 0xF0;
    uint8_t PresenceByte;
    huart1.Instance->BRR = ...
        UART_BRR_SAMPLING16(HAL_RCC_GetPCLK2Freq(), 9600);
    // Mantenere la linea bassa per almeno 480 us (0xF0)
    HAL_UART_Transmit(huart, &ResetByte, 1, 1);
    // Risposta del sensore
    HAL_UART_Receive(huart, &PresenceByte, 1, 1);
    huart1.Instance->BRR = ...
        UART_BRR_SAMPLING16(HAL_RCC_GetPCLK2Freq(), 115200);
    if (PresenceByte != ResetByte){
        return 1; // Presence pulse detected
    }
    else{
        return 0; // No presence pulse detected
    }
}
```

It simply changes baud rates, sends the reset pulse and receives a byte. If the byte received is different from the one sent, the sensor is correctly initialized and ready to communicate. What remains to be done is to put everything together and call the right instructions:

```c
float Sensor_ReadTemp(UART_HandleTypeDef *huart)
{
    uint8_t Temp_LSB, Temp_MSB;
    uint16_t Temp;
```

```
5       float Temperature;
6
7       Sensor_Init();
8       Sensor_WriteByte(huart,0xCC);  // Skip ROM
9       Sensor_WriteByte(huart,0xBE);  // Lettura memoria sensore
10      Temp_LSB = Sensor_ReadByte(huart);
11      Temp_MSB = Sensor_ReadByte(huart);
12      Temp = ((Temp_MSB<<8))|Temp_LSB;
13      Temperature = (float)Temp/16;
14
15      return Temperature;
16  }
17  void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
18  {
19      if (htim == &htim6)
20      {
21          Sensor_Init(&huart1);
22          Sensor_WriteByte(&huart1, 0xCC);  // Skip ROM
23          Sensor_WriteByte(&huart1, 0x44);  // Comincia la conversione ...
                in gradi
24          Sensor_Temp = Sensor_ReadTemp(&huart1); // Temperatura in ...
                centigradi
25      }
26  }
```

The only noteworthy thing to point out in the Sensor_ReadTemp function is that the DS18B20 temperature reading in the scratchpad memory is two bytes in size, but the resolution of the sensor is 12 bit. The correct temperature value is obtained by dividing the 16-bit reading by 16 (or right-shifting by 4).
As is also clear from the code above, we also started the TIM6 basic timer (in interrupt mode with preemption priority equal to 1) with a period of 1000 ms and the following parameters:

$$1000 \times 10^{-3} = \frac{(ARR+1) \times (PSC+1)}{f_{APB1}} = \frac{(9999+1) \times (8999+1)}{90 \times 10^6}$$

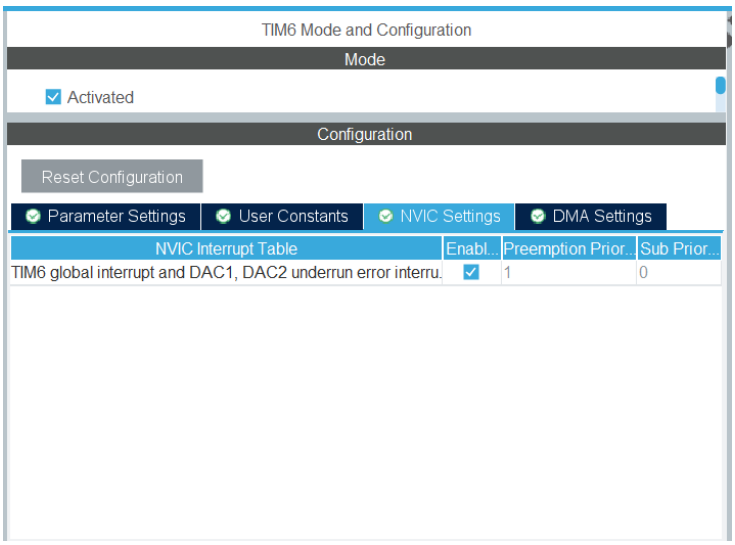


Figure 18: Interrupt configuration of TIM6

Setting the interrupt mode in the CubeMx tool generates the code needed to call the HAL_TIM _Base_MspInit function defined in the tim.c file. It does the following:

```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* tim_baseHandle)
{

  if(tim_baseHandle->Instance==TIM6)
  {

    /* TIM6 clock enable */
    __HAL_RCC_TIM6_CLK_ENABLE();

    /* TIM6 interrupt Init */
    HAL_NVIC_SetPriority(TIM6_DAC_IRQn, 1, 0);
    HAL_NVIC_EnableIRQ(TIM6_DAC_IRQn);
  }
}
```

The HAL_NVIC_SetPriority function accepts the Interrupt Request number, the preemption priority, and the sub-priority as inputs, and formally sets these parameters for the specified interrupt. In this case, the IRQn is "TIM6_DAC_IRQn" which is an alias for the number 54, while the other two values are the ones we choose in the CubeMx tool. The HAL_NVIC_EnableIRQ then enables this interrupt. The function that is tasked to handle it (ISR) is HAL_TIM_IRQHandler(&htim6), and it is called within the stm32f4xx_it.c file.

```
void TIM6_DAC_IRQHandler(void)
{

  HAL_TIM_IRQHandler(&htim6);
  /* USER CODE BEGIN TIM6_DAC_IRQn 1 */

}
```

It is a relatively long function, but we are only concerned with this specific part of it:

```
  /* TIM Update event */
  if ((itflag & (TIM_FLAG_UPDATE)) == (TIM_FLAG_UPDATE))
  {
    if ((itsource & (TIM_IT_UPDATE)) == (TIM_IT_UPDATE))
    {
      __HAL_TIM_CLEAR_FLAG(htim, TIM_FLAG_UPDATE);
#if (USE_HAL_TIM_REGISTER_CALLBACKS == 1)
      htim->PeriodElapsedCallback(htim);
#else
      HAL_TIM_PeriodElapsedCallback(htim);
#endif /* USE_HAL_TIM_REGISTER_CALLBACKS */
    }
  }
```

It calls the weak function defined in the stm32f4xx_hal_tim.c file named HAL_TIM_ PeriodElapsedCallback. It is overridden by our code for sensor data reception in the main.c file, as shown in the previously attached code. This function also serves as the callback from which the control loop instructions will be executed. It will be expanded further when addressing the Peltier cell code.

3 Potentiometer

A potentiometer is a simple component that is used to vary resistance and, consequently, the voltage drop across its terminals. It is, of course, an analog device, often employed as an input system in micro-controller projects.

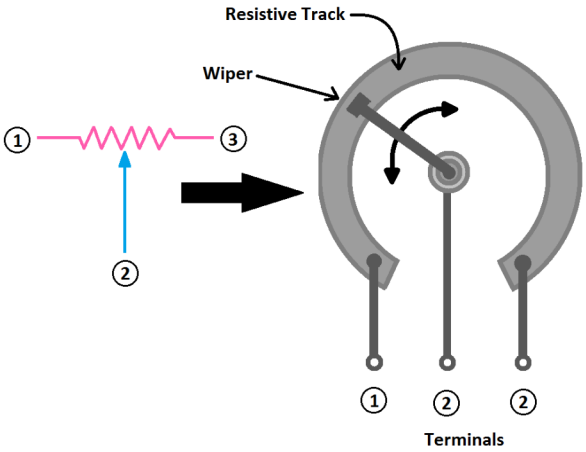


Figure 19: Potentiometer scheme

In our case, we employed a potentiometer with a maximum variable resistance of 10 kΩ, to allow the user to manually set the reference temperature of the cooler. The goal is to read the potentiometer's analog output, convert it into a digital value using the MCU's ADC peripheral, and then process it with the STM32, in order to use it in the control loop.
The STM32F446RET6 has 3 ADC ports (ADC1, ADC2 and ADC3) which are tightly coupled and share some external channels. On its own, the ADC is a successive approximation analog-to-digital converter and has up to 19 multiplexed channels allowing it to measure signals from 16 external sources. It has a configurable resolution of 12, 10, 8 or 6 bits and the scheme provided by the documentation is the following:

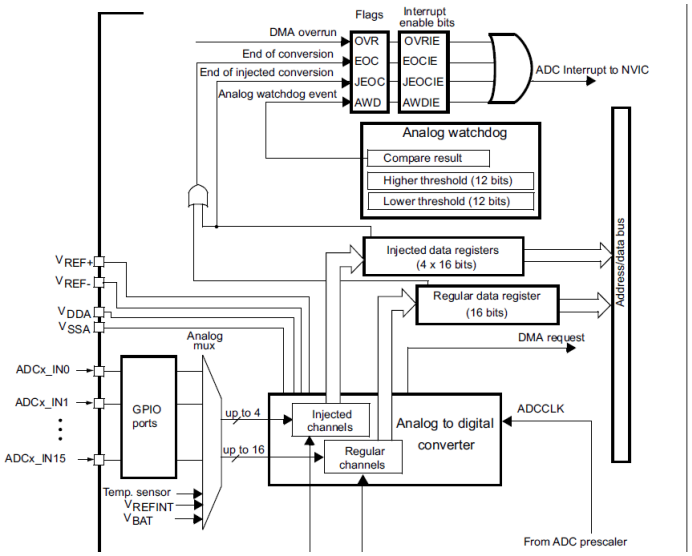


Figure 20: STM32 ADC scheme

Starting from the reference voltages, the ADC pin can operate within the subsequent input range:

$$V_{REF-} \leq V_{IN} \leq V_{REF+}$$

Where V_{REF+} is the higher/positive reference voltage for the ADC ($1.8 \leq V_{REF+} \leq V_{DDA}$) and V_{REF-} is the lower/negative reference voltage for the ADC ($V_{REF-} = V_{SSA}$). Now, since V_{DDA} is the analog power supply equal to V_{DD} or 3.3 V and V_{SSA} is the ground for the analog power supply equal to V_{SS}, we can assume that also V_{REF+} is close to 3.3 V. That is why we chose to power the potentiometer with 3.3 V as well, as can be seen in figure 4.
The ADCCLK is another important parameter to set when dealing with ADCs, as it is related to their conversion time. ADC1, ADC2, and ADC3 are all connected to the APB2 Bus (PCLK2 frequency). However, the ADCCLK frequency is derived through an additional division with another prescaler, which can be configured using the CubeMX tool. The ADC conversion time can be expressed with this equation:

$$T_{conv} = \frac{Sampling\_Cycles + Conversion\_Cycles}{f_{pclk2} * ADCCLK\_prescaler}$$

Where:

- Sampling_Cycles is the time taken, in terms of ADC cycles, by a particular channel to sample the data. This parameter is configurable in the CubeMX and it can be configured separately for each channel.
- Conversion_Cycles is the time taken, in terms of ADC cycles, by the ADC to convert the sampled data. This parameter is not configurable but depends on the ADC resolution that we set in the CubeMX. The higher the resolution, the higher this parameter will be. For 12 bit resolution, which is the starting resolution we set for ADC1 (pin A1), $Conversion\_Cycles = 12 + 3$.

The task involved with our potentiometer is relatively brief and does not require continuous monitoring. Therefore, keeping the ADC active at all times or optimizing its parameters to achieve the fastest possible performance would be unnecessary. In fact, we ultimately decided to split our system operation in two distinct modes: in the first one, the user can freely adjust the reference temperature by turning the potentiometer's knob; in the second mode, the control loop is active, and the Peltier cell is cooling the box. Another factor we considered when selecting the appropriate parameters is that we configured the ADC channel in interrupt mode, where having longer intervals between conversions is actually beneficial in preventing an excessive number of interrupts from overwhelming the MCU. The values we finally established are:

$$Sampling\_Cycles = 480 \qquad ADCCLK\_prescaler = 8$$

Both of these are the maximum quantities that can be set, resulting in a conversion time of approximately 44 μs (we left the resolution at 12 bits as this amount of time worked for us).
Another fundamental setting is the one related to the ADC's conversion mode. We activated the single conversion mode (in CubeMX it is set by disabling all the other conversion modes). There is no clear advantage or disadvantage to using this type of conversion over the continuous mode, except that in continuous conversion mode, we must stop and restart the ADC each time we switch between the two states of the system, previously discussed. In single conversion mode, we only need to "rearm" the ADC operation when required. From a register-level perspective, configuring and using the single conversion mode for regular channels means that:

- ADON bit (first bit of the ADC_CR2 register) is set to 1
- CONT bit (second bit of the ADC_CR2 register) is set to 0.

- SWSTART bit in the ADC_CR2 register is set to 1
- At the end of conversion, the converted data are stored into the 16-bit ADC_DR register
- The EOC bit (second bit of ADC_SR register) is set to 1, and cleared if data is read from the ADC_DR register.
- An interrupt is generated if the EOCIE bit (sixth bit of the ADC_CR1 register) is set (in our case is 1)

The final CubeMx configuration page for the ADC1 looks like this:

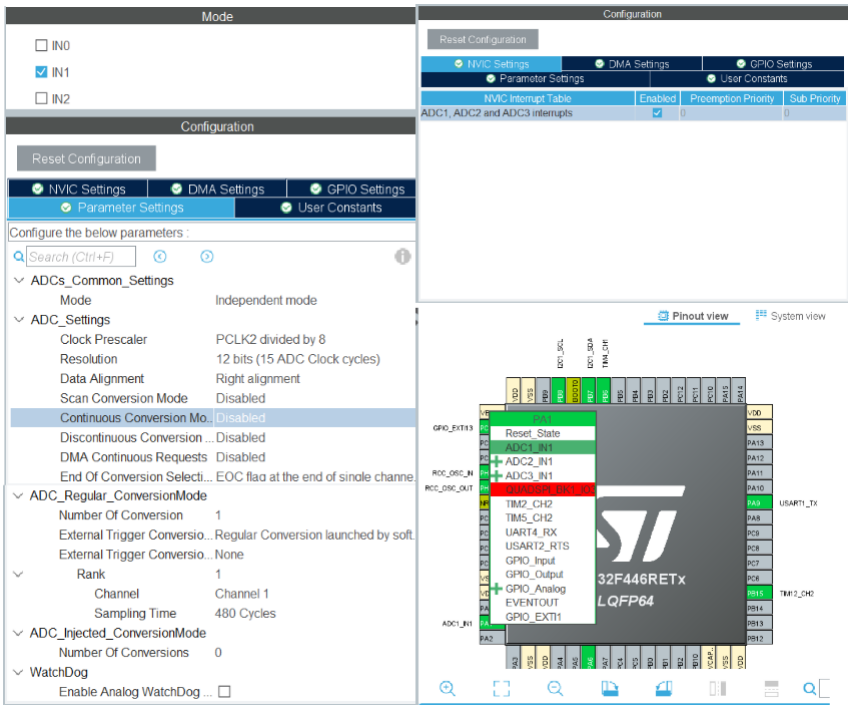


Figure 21: CubeMx ADC configuration

The code written by us for it is simply this:

```
1  void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
2  {
3      if(hadc -> Instance == ADC1)
4      {
5          if (start == 0)
6          {
7              ref_value=(int)HAL_ADC_GetValue(hadc)/256 + ...
                   (Extern_Temp-15);
8              HAL_ADC_Start_IT(hadc);
9          }
10
11     }
12 }
```

The HAL_ADC_ConvCpltCallback callback function is fired at the end of each ADC conversion. It is defined in a similar way to what has already been discussed when dealing with the TIM6

timer. The differences are just the file in which it is defined (stm32f4xx_hal_adc.c), the IRQ number (here its alias is ADC_IRQn and it stands for the number 18) and the handler function which now is HAL_ADC_IRQHandler. As for the code itself, it simply reads the contents of the ADC_DR register with the HAL_ADC_GetValue function (it is just one line of code that reads the DR register by accessing the Instance field of the ADC_HandleTypedef pointer), divides it by the value by 512 and adds a term dependent on the ambient temperature, which is measured using an additional sensor we installed on the outside of the box. The division is necessary to define the range between the maximum and minimum settable values (15 in this case), as the ADC has a 12-bit resolution and can represent a maximum integer value of $2^{12} = 4096$. Meanwhile, the additional sensor and its readings prevent the user from setting a reference temperature higher than the ambient temperature. As for where in the code this measurement takes place, we will explain it briefly.

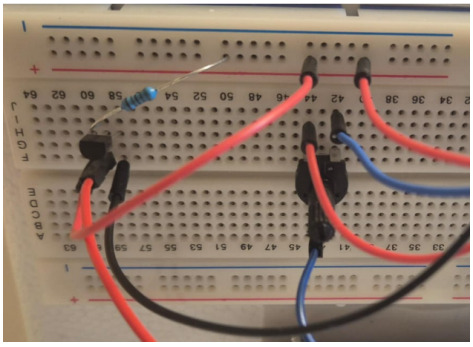


Figure 22: Potentiometer and external sensor on the side of our box

The main thing the HAL_ADC_Start_IT (it is, of course, called after MX_ADC1_Init()) does for us is call the HAl_ADC_Enable macro, which sets the ADON bit to "1", each time a conversion has ended to start a new ADC operation:

```
1  ((hadc)->Instance->CR2 |=  (0x1UL << (0U)))
```

The uint8_t variable "start" represents the current mode of operation of our system. When set to "0," the ADC is actively converting the potentiometer's output. When set to "1", an ADC conversion is no longer performed, and the system operates on the basis of the previously obtained value. We configured the B1 user button (PC13 pin) on the board to toggle this variable with each press.

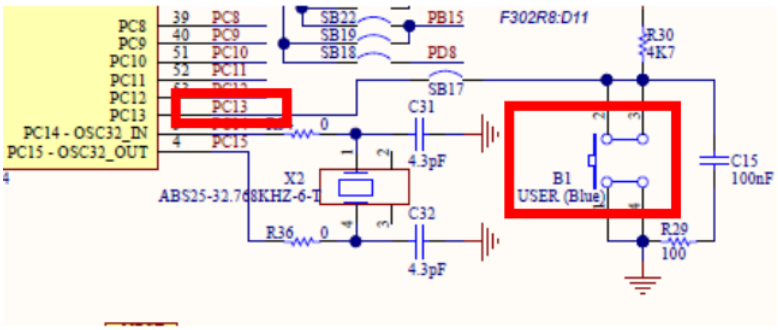


Figure 23: B1 Button schematic

The configuration of the related GPIO port in CubeMx is:

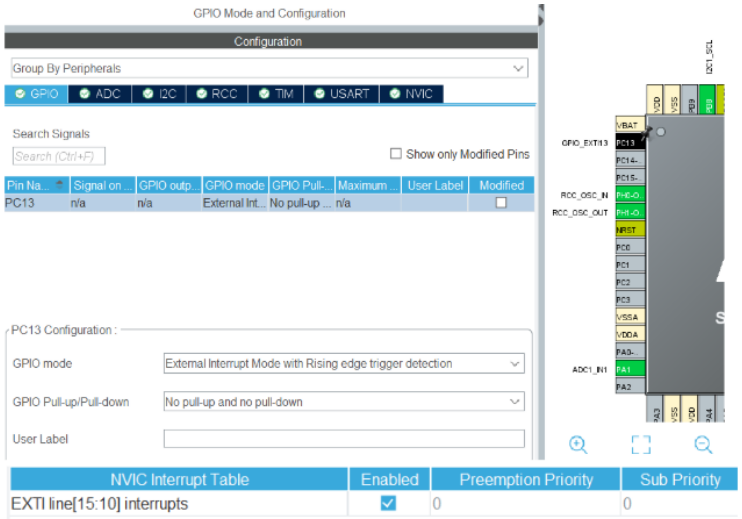


Figure 24: B1 Button schematic

The related code we wrote is:

```
  MX_GPIO_Init();
  void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == GPIO_PIN_13)
    {
        start ^=1;
        if (start==0)
        {
            Sensor_Init(&huart3);
            Sensor_WriteByte(&huart3,0xCC);
            Sensor_WriteByte(&huart3,0x44);
            Extern_Temp= (int) Sensor_ReadTemp(&huart3);
            HAL_TIM_Base_Stop_IT(&htim6);
            htim6.Instance -> CNT = 0x0000;
            HAL_ADC_Start_IT(&hadc1);
        }
        else
        {
            HAL_TIM_Base_Start_IT(&htim6);
        }
    }
}
```

The rising edge of the B1 user triggers an external interrupt, calling the HAL_GPIO_EXTI_Callback function. Here, the NVIC functions are called directly in the GPIO initialization function MX_ GPIO_Init and the IRQ number of the EXTI line 13 is 40 (EXTI lines 10 to 15 are associated with the same IRQn). Within the callback, the start variable is toggled between 0 and 1 using the XOR operation (start ^= 1). When start is set to 1, timer 6 is started using HAL_TIM_Base_Start_IT, which sets the CEN bit of the TIM6_CR1 register and UIE bit of the TIM6_DIER register with the HAL_TIM_ENABLE and HAL_TIM_ENABLE_IT macros. When start is set to 0, the timer is stopped by clearing the same bits mentioned above with

HAL_TIM_Base_Stop_IT and the external temperature is measured before the ADC operation is re-initiated. Additionally, the bits of the CNT register are cleared to ensure that the timer's counter starts again from 0. This callback will be completed in the following chapter.

4 Peltier cell and fans

As mentioned in the introductory chapter, we chose the WiMas kit as the actuator for our system. This kit includes two DC fans and a TEC1-12706 Peltier cell, each of which must be supplied independently with a 12 V voltage. The main difference lies in the current drawn from the power source: the Peltier cell requires up to 6 A, while the smaller fan draws 0.12 A and the larger fan 0.24 A.

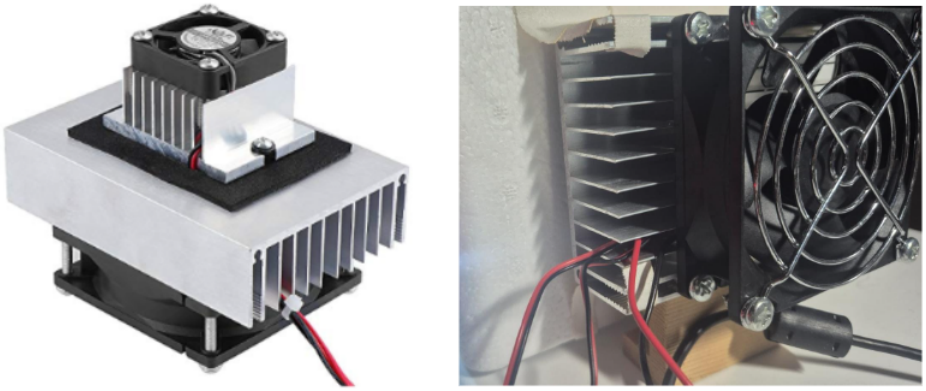


Figure 25: WiMas kit with wires

This time, we will begin by discussing the controller code, and save the circuit part for later in this chapter.
The initial design of our system was centered around using individual controllers for each of the three elements, with each one tuned independently from the other. To achieve this, we followed the approach of manipulating structures' fields, as commonly done in STM32CubeIDE files, and created a C structure for a PID controller. Starting from the pid_controller.h file:

```
1  #ifndef INC_PID_CONTROLLER_H_
2  #define INC_PID_CONTROLLER_H_
3  #endif /* INC_PID_CONTROLLER_H_ */
4
5  #include "stdint.h"
6  typedef struct{
7      uint16_t Kp;
8      uint16_t Kd;
9      uint16_t Ki;
10     uint16_t max_integral;
11     uint16_t max_output;
12     uint16_t min_output;
13     float prev_err;
14     float integral_error;
15 }pid_struct;
16 void set_gains(pid_struct* pid ,uint16_t Kp, uint16_t Kd, uint16_t Ki);
```

```
17 void set_limits(pid_struct* pid, uint16_t max_int, uint16_t max_out, ...
       uint16_t min_out);
18 float pid_action(pid_struct* pid, float error, float Ts);
```

The first three lines of the file are automatically generated by the IDE when creating the .h file. The remainder of the file consists of the declaration of the pid_struct structure and the function prototypes, whose implementations are defined in the pid_controller.c file. The set_gains function allows us to modify the first three fields of the pid_struct, which correspond to the proportional, derivative, and integral gains of the PID controller. The set_limits function modifies the next three fields of the pid_struct, which define the saturation limits for the integral term and the overall output range of the system. Lastly, the pid_action function computes and returns the control output that will be used to drive the actuator.

```
1  #include "pid_controller.h"
2
3  void set_gains(pid_struct* pid, uint16_t Kp, uint16_t Kd, uint16_t Ki)
4  {
5      pid -> Kp = Kp;
6      pid -> Kd = Kd;
7      pid -> Ki = Ki;
8  }
9  void set_limits(pid_struct* pid, uint16_t max_int, uint16_t max_out, ...
       uint16_t min_out)
10 {
11     pid -> max_integral = max_int;
12     pid -> max_output = max_out;
13     pid -> min_output = min_out;
14 }
15 float pid_action(pid_struct* pid, float error, float Ts)
16 {
17     float output;
18     pid -> integral_error +=  error;
19     if (pid -> integral_error > pid -> max_integral)
20     {
21         pid -> integral_error = pid -> max_integral;
22     }
23     if (pid -> integral_error < - (pid -> max_integral))
24     {
25         pid -> integral_error = - (pid -> max_integral);
26     }
27     output = (pid -> Kp*error) + (pid -> Ki * pid -> integral_error ...
           * Ts) +
28              (pid -> Kd *(error - pid ->prev_err)/Ts);
29     if (output > (pid -> max_output))
30     {
31         output = pid ->max_output;
32     }
33     if (output < (pid -> min_output))
34     {
35         output = pid -> min_output;
36     }
37     pid -> prev_err = error;
38
39     return output;
40 }
```

Now, defining PID controllers inside the main.c file becomes straightforward. This is done by declaring variables of type pid_struct and using the appropriate functions to set their parameters, as demonstrated in the following example:

```
#include "pid_controller.h"
pid_struct pid_peltier;
set_gains(&pid_peltier,18,3.5,2.3);
set_limits(&pid_peltier,15,100, 0);
outputP = pid_action(&pid_peltier, error, 1);
```

To translate the control action into actual actuation, we used the output values of the controllers to generate a PWM signal, which allows us to modulate the power delivered to the Peltier cell and the fans. For this purpose, we configured three general-purpose timers TIM3 (Peltier cell, PIN A6), TIM4 (big fan), and TIM12 (small fan) to operate in PWM mode, each assigned to one of the system's actuators. To calculate the ARR and PSC values for the TIM3 we considered the documentation provided by Tellurex (Manufacturer of the cell) which states:

38. Can I use pulse-width modulation to control my Peltier device if I keep the voltage at VMax or below?

Yes, and this is one of the most electrically-efficient ways to control voltage to your device—although you must observe some precautions. As long as you keep the voltage at VMax or below, you will effectively pump heat whenever the duty cycle applies voltage to your system; when the power is turned off, the heat pumping will stop. By pulse-width modulating a suitable voltage, you can easily control the extent of heat pumping by simply varying the duty cycle of the pulses. The great thing about this approach, is that it allows you to minimize power dissipation in your control circuit—especially if you use power MOSFET's for switching (a subject which goes beyond this particular question).

Significant precautions must be employed with PWM, however. First of all, the PWM should be at a high enough frequency to minimize thermal stresses to the TE devices. The "Rule Of Thumb" recommended by Tellurex, is that thermoelectric devices should be pulsed at a frequency of 2000 Hz or higher. Another important issue is the potential for generating electro-magnetic interference (EMI) in the wiring to the TE device. If you are using PWM, always shield your power wiring and keep it away from any sensitive electrical signals.

Figure 26: Tellurex suggestion

So, by considering PSC=1 we calculated that:

$$\frac{f_{APB1t}}{f_{PWM}} = (1 + ARR) \times (1 + PSC) \rightarrow ARR = \frac{90 \times 10^6}{2 \times 2000} - 1 = 22499$$

As for the fans, we discovered that their manufacturer CHAOJINGYIN Electronic, LTD. is actually a ghost company, so there is no official documentation available for their products. To determine an appropriate PWM frequency, we relied on the specifications provided by another Chinese company, CCHV, which suggests a frequency of 25 kHz for similar fan models. Following our previous approach, we set the PSC (prescaler) value to 1, so:

$$\frac{f_{APB1t}}{f_{PWM}} = (1 + ARR) \times (1 + PSC) \rightarrow ARR = \frac{90 \times 10^6}{2 \times 25000} - 1 = 1799$$

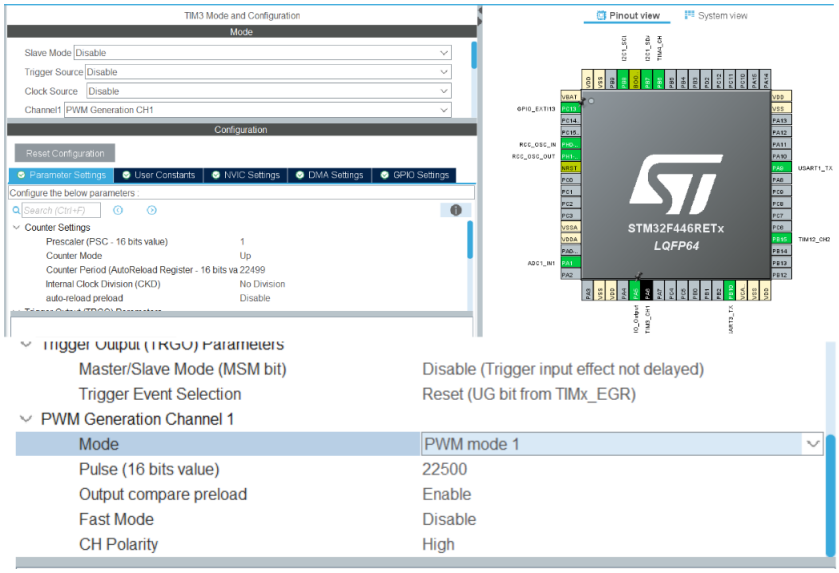


Figure 27: CubeMX TIM3 PWM configuration

After conducting several tests, we observed that independently controlling the two fans did not significantly improve the system's performance; actually, it introduced only disadvantages, some of which are:

- Peltier cells tend to generate more heat than they can dissipate through cooling. If the external fan is turned off, the hot side of the cell can reach temperatures high enough to nullify the cooling effect on the cold side.
- When the external fan starts spinning again after being off, it can introduces hot air into the box. This raises the internal temperature, further destabilizing the control system and counteracting the cooling effect of the Peltier cell.
- Temperature control systems are highly susceptible to disturbances. Factors such as ambient temperature variations and airflow fluctuations significantly impact performance. A constantly changing airflow inside the box introduces instability (large overshoots and undershoots).
- Fine-tuning three separate PID controllers is extremely challenging. Given the slow thermal dynamics of the system, each adjustment requires a long settling time, making the process impractical.

We also tried with an on-off control for the fans but the same instability problems continued to show up. So we ultimately decided to leave the fans continuously running when the system is in its control state. This helped in reducing the temperature fluctuations due to airflow changes. We deactivated then the two PWM channels that weren't needed anymore and set the two pins as simple GPIO outputs.
To better understand the Pulse value shown in Figure 27 and some of the reasoning behind our design choices so far, we will now examine the circuit that connects the power source, the STM32 board, and the Peltier cell. The first configuration we tested is illustrated in Figure 6, with the corresponding schematic shown below.

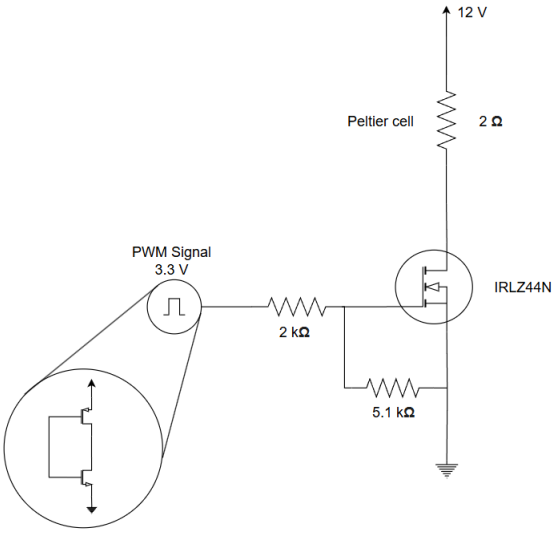


Figure 28: First design of the Peltier's driving circuit

This setup was designed to directly connect the STM32 PWM outputs to the actuator inputs through a basic driver circuits. The issue with this configuration lies in the fact that the PWM signal generated by the STM32 has an amplitude of only 3.3 V. However, the IRLZ44N MOSFET requires a Gate-to-Source voltage of approximately 5 V (logic gate level transistor) to fully turn on, at which point it behaves like a 0.028 Ω resistor. Since the gate voltage is insufficient, the MOSFET does not switch completely, leading to increased resistance, higher power dissipation, which in turn causes it to overheat significantly. In our case, this resulted in the MOSFET becoming extremely hot to the point of smoking. Another important aspect shown in Figure 28 is that the Peltier cell is modeled as a pure resistor, as its capacitance and inductance are negligible. The reason we relied on the PWM frequency recommended in the datasheet for our calculations is that these reactive components do not significantly influence the system's behavior. This makes it challenging to estimate an appropriate PWM frequency based on the cut-off frequency, as would typically be done for components with noticeable inductance or capacitance. We also attempted to design a filtering circuit using the inductors and capacitors we had available (as will be shown soon). However, this approach failed as the components, particularly the inductors, could not withstand the high current demand of 5-6 A from the Peltier cell. As a result, they overheated and burned out.
The next logical step in our circuit design was to configure the PA6 GPIO pin as an open drain output and pull it up with a suitable resistor to a 5 V source. Choosing the right resistor value for each circuit directly impacts the MOSFET's switching behavior. Specifically, we had to consider the gate-to-source capacitance of the transistor and its turn-on and turn-off delay times.

Total gate charge	Q_g	V_{GS} = 5.0 V	I_D = 51 A, V_{DS} = 48 V, see fig. 6 and 13 [b]	-	-	66	nC
Gate-source charge	Q_{gs}			-	-	12	
Gate-drain charge	Q_{gd}			-	-	43	
Turn-on delay time	$t_{d(on)}$	V_{DD} = 30 V, I_D = 51 A, R_g = 4.6 Ω, R_D = 0.56 Ω, see fig. 10 [b]		-	17	-	ns
Rise time	t_r			-	230	-	
Turn-off delay time	$t_{d(off)}$			-	42	-	
Fall time	t_f			-	110	-	

Figure 29: IRLZ44N gate charge and delay times

A higher resistor value increases these delays because the gate capacitance requires more time to

charge and discharge, slowing the transitions between states. Conversely, a lower resistor value allows for faster switching but increases power dissipation in the pull-up resistor. In our configuration, the transistor is driven at a frequency of 2000 Hz, corresponding to a period of 500 μs, while its transition delays are in the ns range. Given this relatively low switching frequency, selecting a higher resistance value for the pull-up resistor is not a concern, as it does not significantly impact switching performance of the Mosfet. Moreover, using a higher resistance value helps limit the current drawn or sunk directly by the microcontroller's GPIO pins. This is particularly important because the STM32 board has a maximum current rating of approximately 25 mA per pin and a total of 120 mA across all pins. The new configuration is the following:

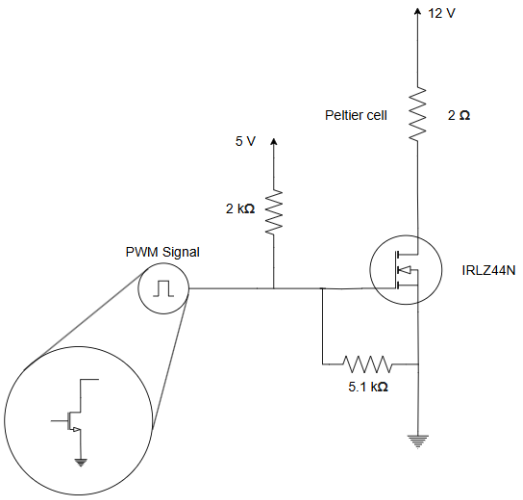


Figure 30: Second design of the Peltier's driving circuit

With this setup, the issue was actually related to the wires. Initially, the gate-to-source voltage (V_gs) is close to 5 V, allowing the MOSFET to switch properly. However, as time passes, the high current flowing through the wires causes them to heat up rapidly, increasing their resistance. Thus, the resistance between the MOSFET's source and ground is no longer negligible, leading to a gradual drop in V_{gs}. As the voltage decreases, the MOSFET gradually increases its resistance and causes additional power dissipation in the form of heat. Although this heating effect is not as extreme as in the previous case, it still leads to a significant temperature rise in the transistor, affecting long-term reliability. However, this circuit is suitable for driving the two DC fans. They do not need as much current and, as previously described, they just need to be turned on or off when switching the cooler's state. So we configured the PC5 and PC8 pins of the board to be open drain digital outputs through the CubeMx tool and followed the schematic to create the circuit on the breadboard.

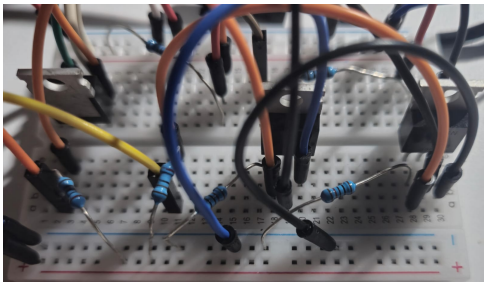


Figure 31: Breadboard circuit of the fans' drivers

The final configuration we ultimately choose to use for this project requires two transistors: an IRLZ44N, as usual, and the 2N2222A BJT transistor provided by the Arduino kit (it is not necessary for it to be a BJT, it just was our only other alternative).

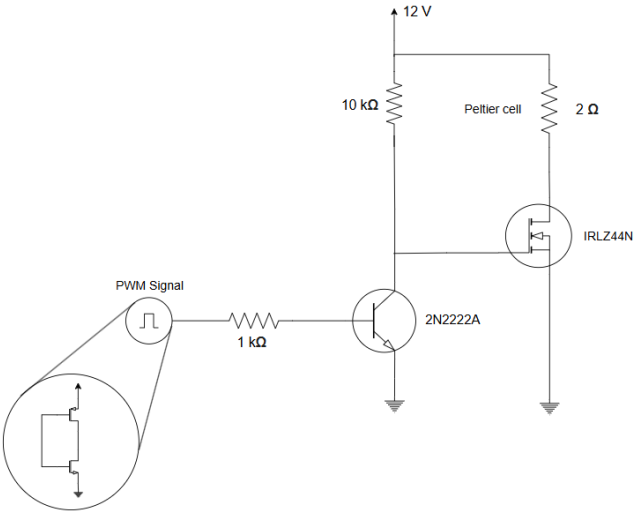


Figure 32: Final schematic for the driver of the Peltier cell

In this configuration, the V_{gs} applied to the MOSFET is limited to a maximum of 12 V. Even when the wires heat up and cause voltage drops, V_{gs} can decrease significantly without falling below the 5 V threshold. While, the BJT is responsible for controlling the switching of the IRLZ44N MOSFET. When the BJT is "closed," it allows the MOSFET to turn on, placing it in its "open" state. Conversely, when the BJT is "open," it causes the MOSFET to turn off, effectively cutting off the current flow through the circuit. (this is why the initial PULSE parameter is set to 22500). To ensure proper operation, the BJT must be driven into its saturation region. Achieving this requires selecting the appropriate resistor values in the circuit. We used a multimeter to verify the correct o peration of the components, which was also the primary instrument for testing all the other configurations proposed in this document. In the "closed" state, the collector-emitter voltage (V_{ce}) is measured to be only 0.03 V, and the collector current (I_c) is effectively zero. This confirms that the BJT is indeed operating in its saturation region, ensuring proper switching behavior for the MOSFET.
Regarding the 12 V power source, we initially used a 12 V 7 Ah rechargeable battery to power the circuit. However, due to the high current demands of the circuit, the battery would frequently discharge. As a result, we ultimately used a 12 V 10 A switching power supply, which can be directly connected to the house power grid. Both the breadboard configuration of the Peltier driver circuit and the switching power supply are shown here:

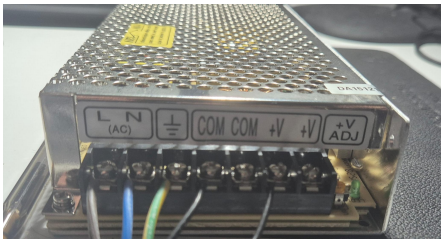


Figure 33: Switching power supply

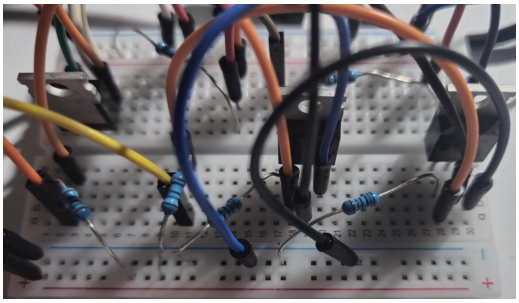


Figure 34: Switching power supply

At this stage, with all the hardware components correctly configured, we can proceed to complete the necessary callback functions to execute the control logic, starting from the main control loop:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim == &htim6)
    {
        if (ref_value>Extern_Temp){Error_Handler();}
        Sensor_Init(&huart1);
        Sensor_WriteByte(&huart1,0xCC);  // Skip ROM
        Sensor_WriteByte(&huart1,0x44);  // Comincia la conversione ...
            in gradi
        Sensor_Temp = Sensor_ReadTemp(&huart1); // Temperatura in ...
            centigradi
        error = (float) -(ref_value - Sensor_Temp);
        outputP = pid_action(&pid_peltier, error, 1);
        dutyP = (float) outputP/100;
/*      uint32_t*/ccrP =(uint32_t) (1+arrP)*(1-dutyP);
        __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, ccrP);
        htim3.Instance->EGR = TIM_EGR_UG;
    }
}
```

Since the system is designed for cooling, the control error is computed with an additional negative sign to ensure that a positive error corresponds to the box's temperature being higher than the desired setpoint. This ensures the PID controller reacts appropriately by increasing the cooling effect when needed. The duty cycle is obtained by dividing the PID output by 100, ensuring it always remains within the [0,1] range. This is necessary because the PID controller's output limits were previously set between 0 and 100. The compare register value (TIM3_CCR1 register), which determines the PWM signal's duty cycle, is computed using the formula:

$$ccr_P = (1 + arr_P) \times (1 - duty_P)$$

The (1 - duty$_P$) factor is needed because when the PWM signal is high, the MOSFET is actually open, meaning that the Peltier cell is inactive. The arrP value is obtained in the main function with this line of code:

```
arrP = htim3.Instance -> ARR;
```

The HAL_TIM_SET_COMPARE macro is what actually writes in the CCR1 register the value that we calculated as ccrP.

```
#define __HAL_TIM_SET_COMPARE(__HANDLE__, __CHANNEL__, __COMPARE__) \
  (((__CHANNEL__) == TIM_CHANNEL_1) ? ((__HANDLE__)->Instance->CCR1 ...
     = (__COMPARE__)) :\
   ((__CHANNEL__) == TIM_CHANNEL_2) ? ((__HANDLE__)->Instance->CCR2 ...
     = (__COMPARE__)) :\
   ((__CHANNEL__) == TIM_CHANNEL_3) ? ((__HANDLE__)->Instance->CCR3 ...
     = (__COMPARE__)) :\
   ((__HANDLE__)->Instance->CCR4 = (__COMPARE__)))
```

It checks the channel of the timer that we pass it and writes in the corresponding register. The last line of code sets the first bit of the EGR register to 1 in order to re-initialize the timer's counter and generates an update of the registers. Moreover, if for any reason the desired reference temperature is higher than the external ambient temperature, the error handler is invoked. It simply displays on the screen the message: "ERROR PRESS Restart".

```
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error ...
     return state */
  Lcd_Put_Cur(0, 0);
  Lcd_Send_String("ERROR");
  Lcd_Put_Cur(1, 0);
  Lcd_Send_String("Press RESET");
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}
```

The functions that it calls will be explained in the next chapter.
The callback fired when the B1 button is pressed now is this:

```
void Error_Handler(void)
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == GPIO_PIN_13)
    {
        start ^=1;
        if (start==0)
        {
            htim3.Instance -> CCR1 = 45000;
            htim3.Instance->EGR = TIM_EGR_UG;
            GPIOA -> ODR &= ~(1<<5);
            GPIOC -> ODR &= ~(1<<5);
            GPIOC -> ODR &= ~(1<<8);
            HAL_TIM_Base_Stop_IT(&htim6);
            htim6.Instance -> CNT = 0x0000;
            Sensor_Init(&huart3);
            Sensor_WriteByte(&huart3,0xCC);  // Skip ROM
            Sensor_WriteByte(&huart3,0x44);  // Comincia la ...
               conversione in gradi
            Extern_Temp= (int) Sensor_ReadTemp(&huart3);
```

```
20                HAL_ADC_Start_IT(&hadc1);
21            }
22            else
23            {
24                GPIOA -> ODR |= (1<<5);
25                HAL_TIM_Base_Start_IT(&htim6);
26                GPIOC -> ODR |= (1<<5);
27                GPIOC -> ODR |= (1<<8);
28            }
29        }
30    }
```

Now the callback function also sets or resets the appropriate bits in the Output Data Register (ODR) to turn the fans on or off as needed. Furthermore, we configured the LD2 LED (pin A5) to serve as an indicator of the control loop's activity. When the control loop is actively running, the LED lights up, providing a visual confirmation that the system is executing temperature regulation. Lastly, to fine tune the gains of the controller we used the "STM Studio" application to better visualize the response of the system. The final values of the gains are $K_p = 18$, $K_d = 3.5$ and $K_i = 2.3$. A typical response with these values looks like this:

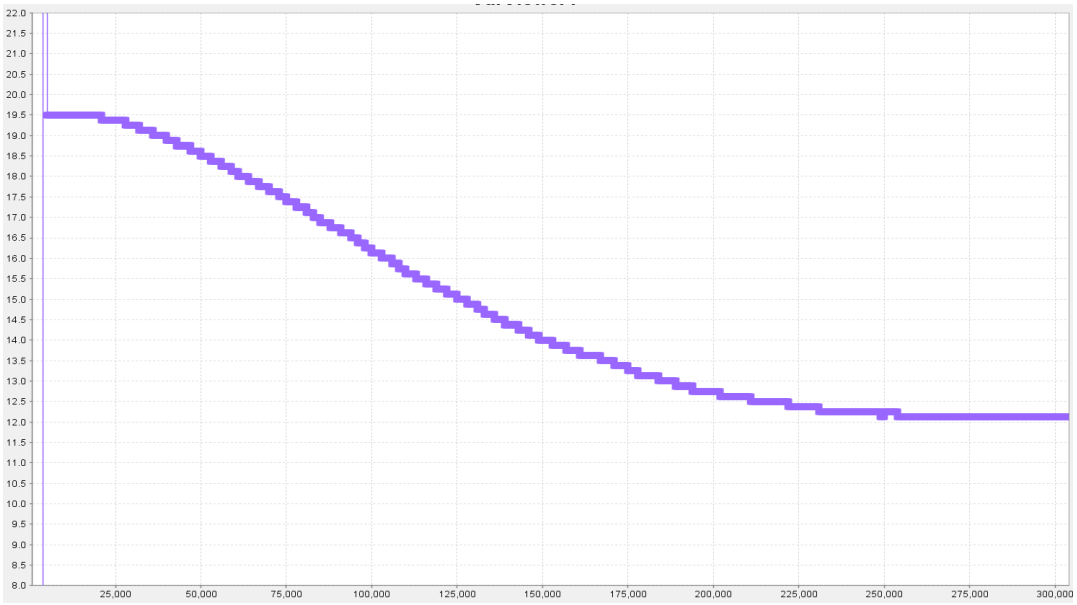


Figure 35: Response of the system

5 LCD screen

The LCD screen we selected is the LCD HD44780 with a 16x2 display, which means that it can display two rows of 16 characters each. It comes paired with the PCF8574, an 8-bit I/O expander for the I^2C-bus, which translates the I^2C signals into the parallel signals required by the LCD.

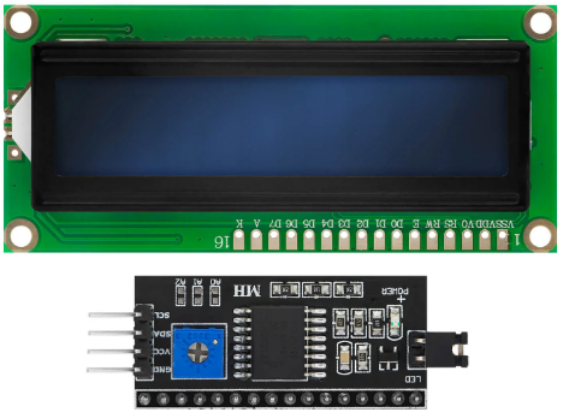


Figure 36: Lcd display and PCF8574 module

LCDs typically require multiple pins for proper interfacing with an MCU, including data lines, control signals, and enable lines. By integrating an I^2C module, the number of connections is reduced to just 4 pins:

- A ground pin GND.
- A supply voltage pin Vcc (5V for full brightness).
- Serial Clock Line pin (SCL). Connected to the SCL bus that distributes the clock signal.
- Serial Data Line (SDA). Connected to the SDA bus that sends data clocked by the SCL.

This module also includes 16 output pins, which must be soldered to the corresponding holes on the LCD display to establish a proper connection. The final result we obtained is the following:

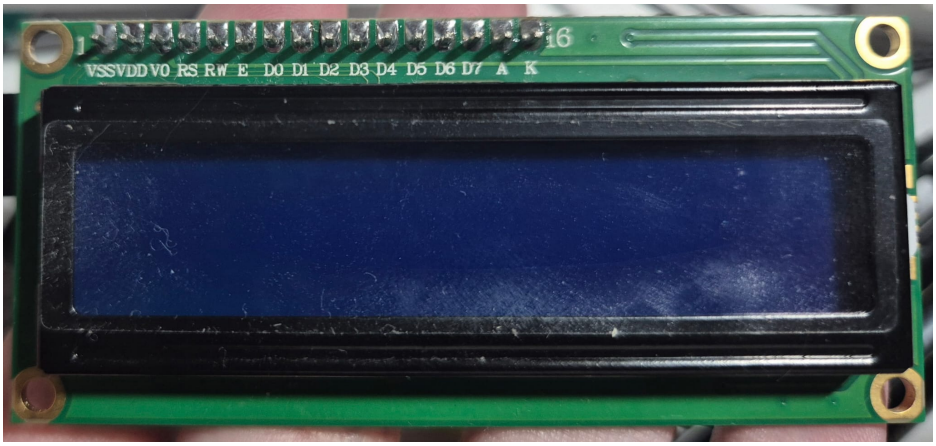
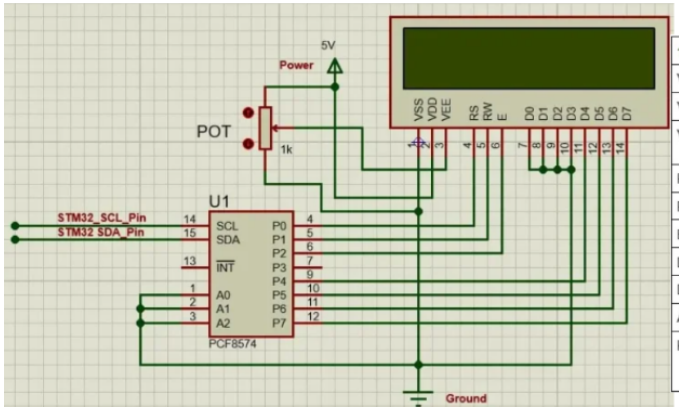


Figure 37: PCF8574 soldered with the display

To understand the role of each of those pins, the LCD and PCF8574 provide the following schematic and information:



16x2 LCD	PCF8574 Module
VSS	GND
VCC	5V
V0 / VEE	1K Potentiometer for control the brightness of the LCD
RS	P0
RW	P1
E	P2
D0 to D3	GND
D4 to D7	P4 to P7
A	5V
K	Connected to P3 through a transistor for ON/OFF the LCD backlight using command of MCU

Figure 38: LCD display schematic

Here is also some additional information about what is shown in Figure 38:

- Vcc and Vdd both stand for 5 V.
- The 1 kΩ potentiometer is integrated in the same PCB as the PCF8574 chip.
- P0 is connected to the pin RS of the LCD, which defines whether the transmitted byte is a command (0) or data (1).
- P1 is connected to the R/W pin of the LCD. This pin should be "0" when writing the data to the LCD and "1" when reading the data from the LCD.
- This pin is used to latch the data present on the data pins. A high to low pulse with a minimum width of 450 ns is required to latch the data to the display.
- P3 is connected to K and turns on/off the backlight.
- P4 – P7 are connected to the data pins D4 – D7. Since only 4 data pins are available for the PCF8574, D0 - D3 are unused, and the LCD must be configured in 4-bit mode.
- A0 - A2 are the pins that can change the I^2C address of the PCF8574. Normally these three pins are HIGH and the default address is 0100111 (0x27). This option is useful in case multiple displays are on the same bus. In our project, we left these three bits all at "1".

In order to set up the SDA and SCL lines on the MCU side we must again use the CubeMx tool. Four I^2C bus interfaces can operate in multimaster and slave modes on the STM32F446RET6. Three of them (I2C1, I2C2, I2C3) can support the standard (up to 100 KHz) and fast (up to 400 KHz) modes, while the other (FMPI2C1) supports the standard (up to 100 KHz), fast (up to 400 KHz) and fast mode plus (up to 1MHz) modes. The PCF8547 can operate on a 100 kHz I^2C-bus interface (as stated in its datasheet), so we chose to activate the I2C1. Its SDA line is associated with the PB7 pin and its SCL line with the PB8 pin.
The I^2C protocol requires both of these lines to operate in an open-drain configuration, a trait it shares with the data line of the 1-Wire protocol discussed earlier. Therefore, the GPIO mode of the two pins must be set to "Alternate Function open drain". The I^2C module connected to the LCD display provides the necessary pull-up resistors of 4.7 kΩ allowing to disable both pull up and pull down resistors of the two MCU ports.
The resulting CubeMx configuration page looks like this:

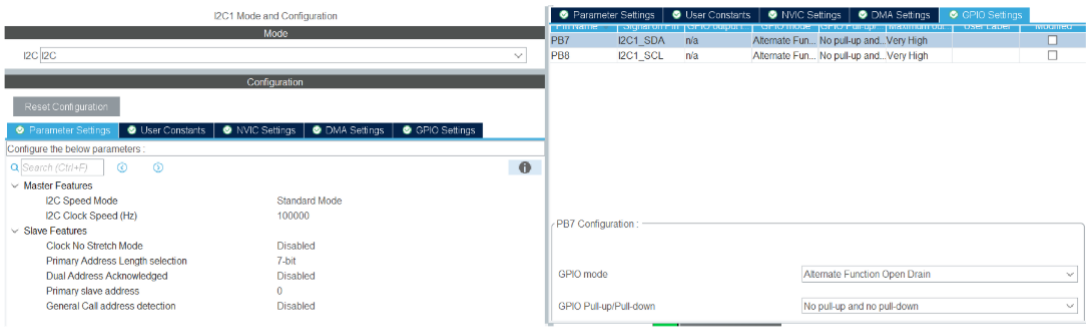


Figure 39: I2C1 CubeMx configuration

By default the I^2C interface operates in Slave mode. The first bit of the I^2C_CR1 register (PE) i set to "1" when the protocol is enabled. The peripheral input clock must be programmed in the first six bits of the I^2C_CR2 register to generate correct timings and must have a minimum value of 2 kHz or 0b000010 and cannot exceed 50 kHz or 0b110010 (I2C1 is connected to the APB1 bus which in our case operates at 45 kHz). Setting the START bit, which is the ninth bit of the I2C1_CR1 register, triggers the interface to generate a start condition and transition to Master mode. This is confirmed by the first bit of the I2C1_SR2 register being set to "1". However, this operation can only occur when the BUSY bit (the second bit of the I2C1_SR2 register) is "0", indicating that the bus is idle and no communication is currently taking place. Once the Start condition is sent, the SB bit (first bit of I2C_SR1 register) is set by hardware. The master then waits for a read of the SR1 register and writes the slave address (the address we will send is always 0b010001110) in the DR register (it holds the byte received or the byte to be transmitted to the bus). We will use the I^2C interface exclusively for writing operations, meaning the sequence of events will consistently follow this structure:

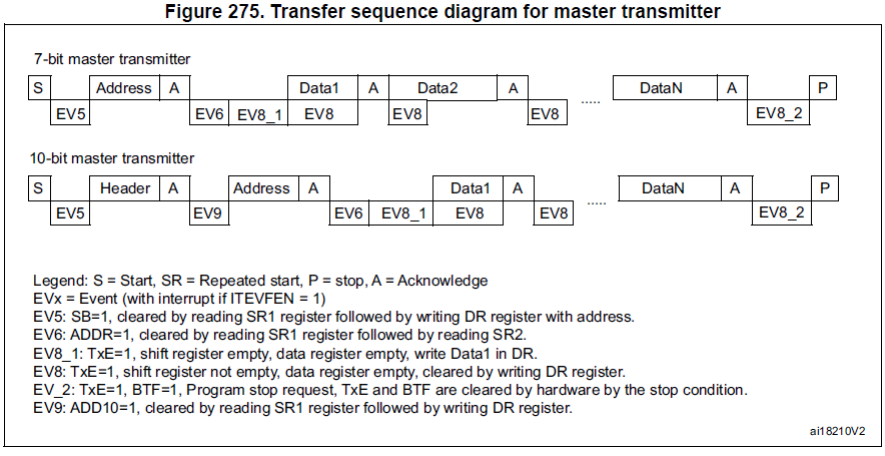


Figure 40: I^2C interface transmission sequence

The ADDR and TxE bits are located in the I^2C_SR1 register. The ADDR bit is set at the end of the slave address transmission, signaling that the address phase has been successfully finished. Meanwhile, the TxE bit is set when the DR register is empty, indicating that it is ready to accept

new data for transmission.
All these register-level operations are handled by the initialization code and the HAL functions automatically generated by the CubeMX tool. The additional logic required to send commands and data in the specific format expected by the LCD module had to be written by us. There are many libraries available that implement this kind of logic; however, they often include a wide range of functions that go beyond what we actually need. To keep things simple and efficient, we decided to write only the necessary parts tailored to our specific requirements. Some auxiliary functions that we wrote are the following:

```
void delay_time(uint16_t time)
{
    uint32_t i;
    for (i = 0; i < 1000*time; i++) {
        asm("nop");
    }
}

void intToStr(int N, char *str)
{
    uint8_t i =0;
    int temp = N;
    if (N==0){str[i++]='0';}
    else{
        while (N > 0)
        {
            str[i++] = (char) N % 10 + '0';
            N /= 10;
        }
        if (temp < 0) {
            str[i++] = '-';
            N=-N;
        }
        for (int j = 0, k = i - 1; j < k; j++, k--)
        {
            char temp = str[j];
            str[j] = str[k];
            str[k] = temp;
        }
    }
    str[i++] = '\0';
}
```

The delay_time function is a simple alternative to the HAL_delay function used to implement a pause in program execution for a specified amount of time. The assembly instruction asm("nop") is executed during each iteration of the for loop and basically tells the processor to do nothing.
The intToStr function converts an integer into its string representation and stores the result in the provided char array (we use a pointer to directly modify the argument passed to a function, without the need to return it from the function). It first checks whether it is 0. If that is the case it directly stores a '0' string in the array; otherwise, a while loop extracts each digit of N from right to left using the modulus operator and converts it to its corresponding ASCII character by adding '0'. In C, characters are represented by their ASCII values. When adding an int to a char, the corresponding ASCII value (integer) of the char is used in the calculation (e.g., 48 for '0'), and the result is then cast back to a char. The array is finally reversed to put everything back in order and a null terminator is added.

As mentioned earlier, we need to configure the LCD in 4-bit mode (it must receive 4 data bits at a time). The initialization procedure is provided by the datasheet and it is:

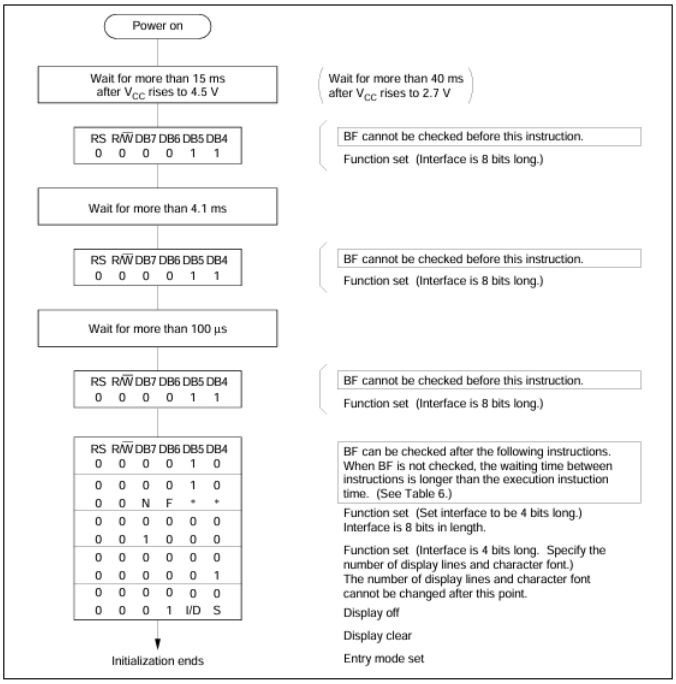


Figure 41: LCD 4-bit Interface initialization

We followed the same approach used for the sensor code, first writing the code to send data on the bus:

```
const uint8_t Lcd_Address= 0x4E;
const uint8_t RS_BIT= 0;
const uint8_t EN_BIT= 2;
const uint8_t BL_BIT= 3;
const uint8_t D4_BIT= 4;
void Lcd_write(uint8_t value, uint8_t rs) {
  uint8_t data = value << D4_BIT;
  data |= rs << RS_BIT;
  data |= 1 << BL_BIT;
  data |= 1 << EN_BIT;
  HAL_I2C_Master_Transmit(&hi2c1, Lcd_Address, &data, 1, 100);
  delay_time(180);
  data &= ¬(1 << EN_BIT);
  HAL_I2C_Master_Transmit(&hi2c1, Lcd_Address, &data, 1, 100);
}
void Lcd_Send_Cmd (char cmd)
{
      uint8_t upper_nibble = cmd >> 4;
      uint8_t lower_nibble = cmd & 0x0F;
      Lcd_write(upper_nibble, 0);
      Lcd_write(lower_nibble, 0);
}
void Lcd_Send_Data (char data)
```

```
{
      uint8_t upper_nibble = data >> 4;
      uint8_t lower_nibble = data & 0x0F;
      Lcd_write(upper_nibble, 1);
      Lcd_write(lower_nibble, 1);
}
void Lcd_Send_String (char *str)
{
    while (*str)
    {
        Lcd_Send_Data (*str++);
    }
}
```

The first five constant values represent the LCD address (it is not 0x27 only because it also contains the write bit) and the positions of the RS, K, E, and D4 bits. These variables are used to correctly format the information to be sent, as the data follows the LSB logic.

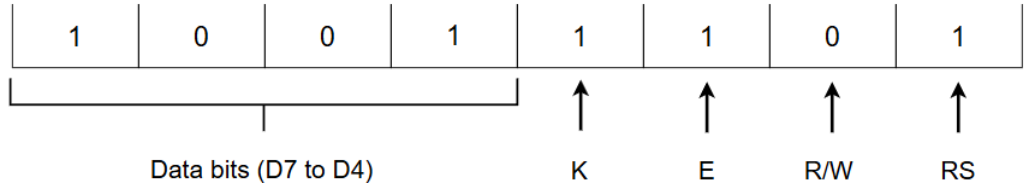


Figure 42: Bits of the data variable

The Lcd_write function formats the data to be sent as shown in Figure 42 and sends it with the HAL_I2C_Master_Transmit. The data must be sent twice because the E bit needs to transition from HIGH to LOW to latch the data onto the display. The argument of the delay_time function is obtained by considering that each SYSCLK period is roughly 5.5 ns. To set 1 ms the formula is: $\frac{1\text{ ms}}{5.5\text{ ns}} \times 1000 \approx 180$. (Of course, to be sure of this timing further measurements are needed, but these values worked for us)
The only difference between the functions Lcd_Send_Cmd and Lcd_Send_Data is that the former calls the Lcd_write function with the RS bit cleared, while the latter does not. Both functions divide the input byte into two nibbles using bitwise operations. The last function Lcs_Send_String will send each character of the string passed as an argument to the LCD one by one.
The initialization function, which follows the scheme shown in Figure 41 is:

```
void Lcd_Init(void){
    delay_time(2880);  // Fermo per 16 ms
    Lcd_write(0x03, 0);
    delay_time(900);  // Fermo per 5 ms
    Lcd_write(0x03, 0);
    delay_time(180);  // Fermo per 1 ms
    Lcd_write(0x03, 0);
    delay_time(180);
    Lcd_write(0x02, 0);
  // dislay initialisation
    Lcd_Send_Cmd (0x0C);
    Lcd_Send_Cmd (0x06);
    Lcd_Send_Cmd (0x01);
    delay_time(360);
}
```

The first three calls are not true commands, as they are needed just for initialization purposes. The first command sets the 4-bit mode. The second sets the rows that must be displayed (N=1 means 2 rows) and the character font (F=1 means 5×8 character font). The next command is actually the last one shown in figure 41. For some reasons our display does not follow the indicated order and requires the last two commands to be switched.
The last function we wrote allows us to change the cursor of the display, that is, the position it will consider when we write a new character:

```
void Lcd_Put_Cur(uint8_t row, uint8_t col)
{
    uint8_t address;
    switch (row) {
        case 0:
            address = 0x00;
            break;
        case 1:
            address = 0x40;
            break;
        default:
            address = 0x00;
    }
    address += col;
    Lcd_Send_Cmd(0x80 | address);
}
```

The upper nibble of the Set Cursor command controls the row, while the lower nibble specifies the column of the screen. For the first row, the upper nibble must be set to 0b1100 and for the second row, it is 0b1000. The column nibble corresponds to the number of the column to set.
All these functions are called from the endless loop inside the main function, in order to keep the values of the screen constantly updated.

```
Lcd_Init();
  Lcd_Put_Cur(0, 0);
  Lcd_Send_String("Goal:");
  Lcd_Put_Cur(1, 0);
  Lcd_Send_String("Temp:");
  while (1)
  {
          if (start==0)
      {
          intToStr(ref_value, num);
          if (ref_value<10)
          {
              Lcd_Put_Cur(0, 5);
              Lcd_Send_String("0");
              Lcd_Put_Cur(0, 6);
              Lcd_Send_String(num);
              Lcd_Put_Cur(0, 7);
              Lcd_Send_String(".0");
          }else
          {
              Lcd_Put_Cur(0, 5);
              Lcd_Send_String(num);
          }
```

```
24          }else
25          {
26              int intpart = (int) Sensor_Temp;
27              float temp =(Sensor_Temp-intpart)*100;
28              int decpart = (int) temp;
29              intToStr(intpart, numI);
30              if (intpart<10)
31              {
32                  Lcd_Put_Cur(1, 5);
33                  Lcd_Send_String("0");
34                  Lcd_Put_Cur(1, 6);
35                  Lcd_Send_String(numI);
36                  Lcd_Put_Cur(1, 7);
37              }else
38              {
39                  Lcd_Put_Cur(1, 5);
40                  Lcd_Send_String(numI);
41                  Lcd_Put_Cur(1, 7);
42              }
43              Lcd_Send_String(".");
44              if (decpart==0)
45              {
46                  Lcd_Put_Cur(1, 8);
47                  Lcd_Send_String("00");
48              }
49              intToStr(decpart, numD);
50              Lcd_Put_Cur(1, 8);
51              Lcd_Send_String(numD);
52          }
53          delay_time(5000);
54      }
55      }
```

The code is quite simple; the only important aspect is the conversion of a float number into a string. First, the integer part of the Sensor_Temp variable is isolated by casting it to an integer. Then, the fractional part of the original float number is extracted and processed similarly. The two resulting integers are then displayed separated by a comma.
The final result is the following:

Figure 43: Final result

6 Conclusion

In this project, we successfully designed and implemented a temperature control system using a Peltier cell, PWM control, and a PID algorithm. However, there are still some aspects that could be further improved:

- We initially attempted to create a low-pass filter using capacitors and inductors to smooth the PWM output, but our components were not rated for the high current. Using properly sized inductors in a future iteration could help in creating a better performing system. Here how such circuit could look like:

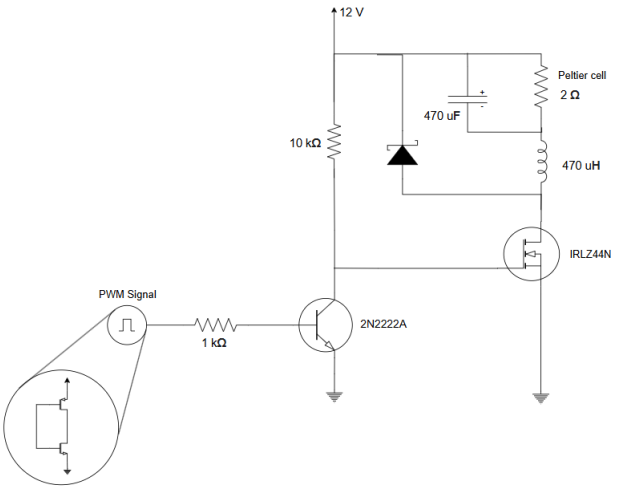


Figure 44: Better performing driving circuit

- Our system currently only cools the environment. By implementing a heating mode, the system could operate within a wider temperature range, making it more versatile. Both reversing the Peltier cell's polarity and integrating an additional heating element, such as a resistive heater could be valid option to consider.
- The current PID controller uses fixed Kp, Ki, and Kd values, which were manually tuned. Implementing an adaptive tuning method, such as auto-tuning algorithms, could allow the system to dynamically adjust these parameters based on external conditions.

The final system we created look like this:

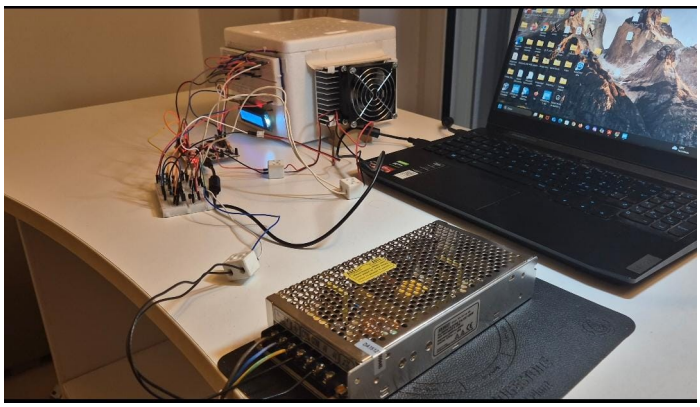


Figure 45: Better performing driving circuit