**PROJECT REPORT**

**WEIGHT SCALE USING LOAD CELL**

**&**

**BOOTLOADER**

Name:

Supervisor:

Ho Chi Minh City, 2024

# TABLE OF CONTENTS

[TABLE OF CONTENTS ii](#_Toc175316345)

[LIST OF FIGURES iv](#_Toc175316346)

[LIST OF TABLES vi](#_Toc175316347)

[CHAPTER 1 : WEIGHT SCALE 7](#_Toc175316348)

[1.1 . Strain-gauge loadcell 7](#_Toc175316349)

[1.2 . 24-Bit Analog to Digital Converter 9](#_Toc175316350)

[1.2.1 . 24-Bit ADC HX712 9](#_Toc175316351)

[1.2.2 . Comparison between HX712 and HX711 12](#_Toc175316352)

[1.2.3 . HX712 Data Acquisition 13](#_Toc175316353)

[1.3 . Communication protocols 17](#_Toc175316354)

[1.3.1 . Implementing UART for Host computer and Arduino communication 17](#_Toc175316355)

[1.3.2 . Implementing I2C for Arduino and STM32 communication 19](#_Toc175316356)

[1.4 . Data Transmission Flowcharts 22](#_Toc175316357)

[1.5 . Experiment and Evaluation 24](#_Toc175316358)

[1.5.1 . Experiment results 24](#_Toc175316359)

[1.5.2 . Results Evaluation 24](#_Toc175316360)

[CHAPTER 2 : BOOTLOADER 27](#_Toc175316361)

[2.1 . Introduction 27](#_Toc175316362)

[2.2 . ARM boot sequence 28](#_Toc175316363)

[2.3 . Project Memory Map Structure 29](#_Toc175316364)

[2.4 . Bootloader sequence 31](#_Toc175316365)

[2.4.1 . Bootloader flowchart 31](#_Toc175316366)

[2.4.2 . Jumping/Updating application 32](#_Toc175316367)

[2.5 . Data Transmission Process 34](#_Toc175316368)

[2.5.1 . Main Application Flowchart 34](#_Toc175316369)

[2.5.2 . Intel HEX file structure 35](#_Toc175316370)

[2.5.3 . Data confirmation 38](#_Toc175316371)

[2.5.4 . Data transmission flowchart 39](#_Toc175316372)

[2.6 . Bootloader result 41](#_Toc175316373)

[APPENDIX A: ADC types 43](#_Toc175316374)

[APPENDIX B: USING STM32 IN VSCODE VIA MAKEFILE AND STLINK 44](#_Toc175316375)

[APPENDIX C: OCCUPIED FLASH SIZE 59](#_Toc175316376)

[REFERENCES 61](#_Toc175316377)

# LIST OF FIGURES

Figure 1.1. Strain-gauge loadcell 7

Figure 1.2. Strain-gauge loadcell principle [4] 7

Figure 1.3. Wheatstone Bridge circuit [4] 8

Figure 1.4. HX712 Typical weigh scale application block diagram 10

Figure 1.5. HX711 input offset drift with different gains 11

Figure 1.6. Sigma\delta ADC block diagrams 11

Figure 1.7. HX711 Typical weigh scale application block diagram 12

Figure 1.8. HX712 data acquisition communication sequence 13

Figure 1.9. HX712 data acquisition communication sequence 15

Figure 1.10. UART communication sequence 18

Figure 1.11. I2C communication sequence 19

Figure 1.12. Host computer, Master and Slave MCUs data acquisition flowchart 23

Figure 1.13. Co 24

Figure 1.14. Measurement results with different objects 24

Figure 1.15. UART communication sequence 25

Figure 1.16. UART communication sequence 26

Figure 2.1. ARM Memory Maps [2] 27

Figure 2.2. STM32F103C8T6 Boot configuration pins 28

Figure 2.3. STM32F103C8T6 Boot mode 29

Figure 2.4. Project Flash Memory Map 29

Figure 2.5. Bootloader flowchart 31

Figure 2.6. Weight scale and Upload Application in Main Application flowchart 35

Figure 2.7. Intel HEX file structure examples 36

Figure 2.8. Data transmission flowcharts 40

Figure 2.9. Uploading Terminal 41

Figure 2.10. Fail data transmission 41

Figure 2.11. Successful data transmission 42

# LIST OF TABLES

**Table 1.1. *Key characteristic comparison between HX711 and HX712*** 12

**Table 1.2. *Record type used this the project*** 13

**Table 1.3. *Inputs, Rate and Gain selection*** 17

**Table 1.4. *Record type used this the project*** 21

**Table 2.1. *Some utilized Record type*** 36

# : WEIGHT SCALE

A weight scale using a load cell is an instrument that measures weight by detecting the strain produced by an applied load. The load cell converts this mechanical strain into an electrical signal, which is then processed to determine the precise weight. In this chapter, components of the system will be discussed, as well as the implementation of the system.

## . Strain-gauge loadcell

A load cell, shown in Fig. 1. 1 is an electro-mechanical sensor used to measure force or weight. It has a simple yet effective design which relies upon the well-known transference between an applied force, material deformation and the flow of electricity.

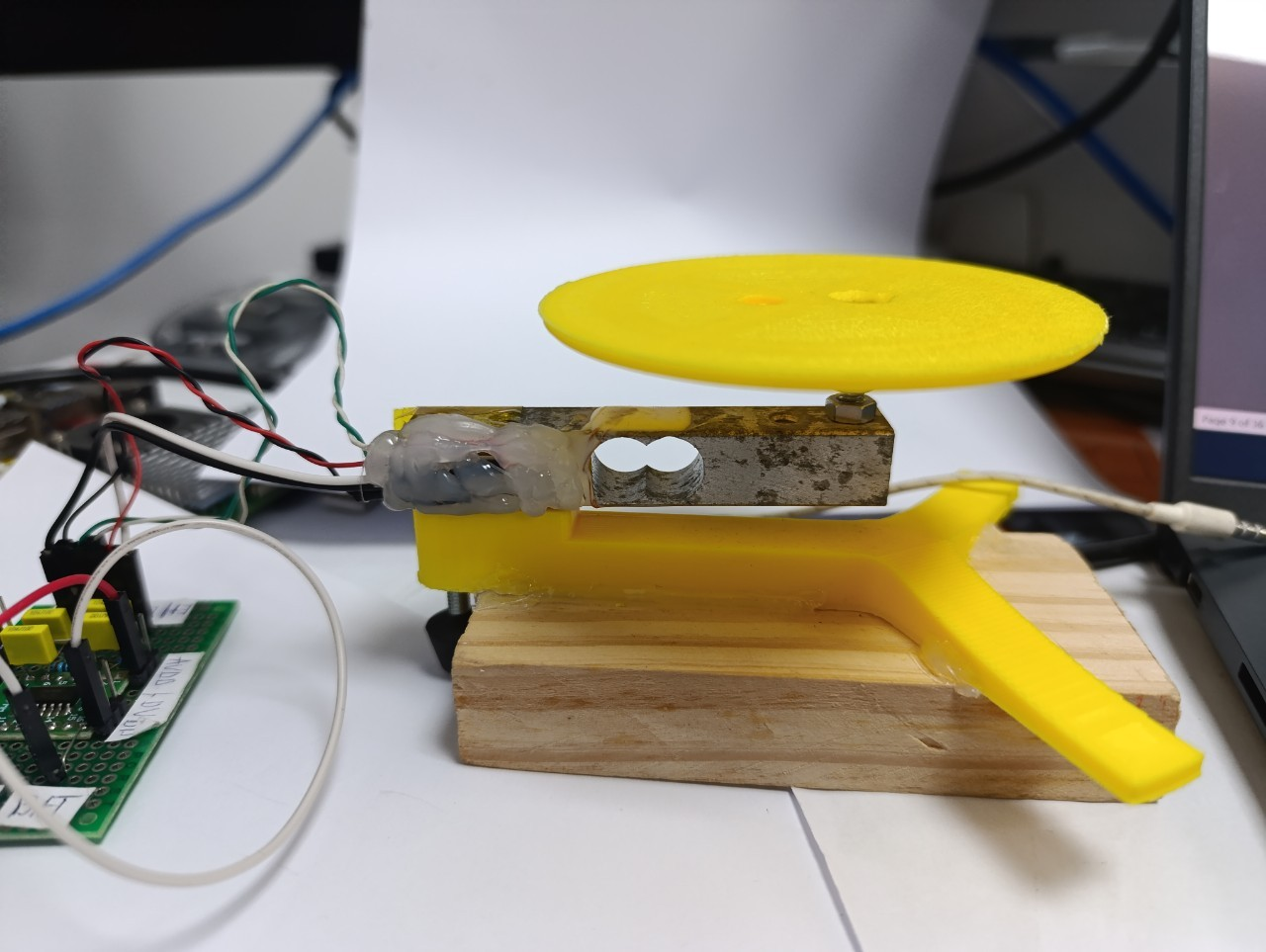


Figure 1.1. Strain-gauge loadcell

In this project, strain gauge loadcell is used. Strain gauges are thin, flexible sensors made of metal wire that are bonded to a load cell's surface. The load cell structure, typically made from materials like aluminum, steel, or other alloys, deforms slightly under load, causing the strain gauges to deform as well.

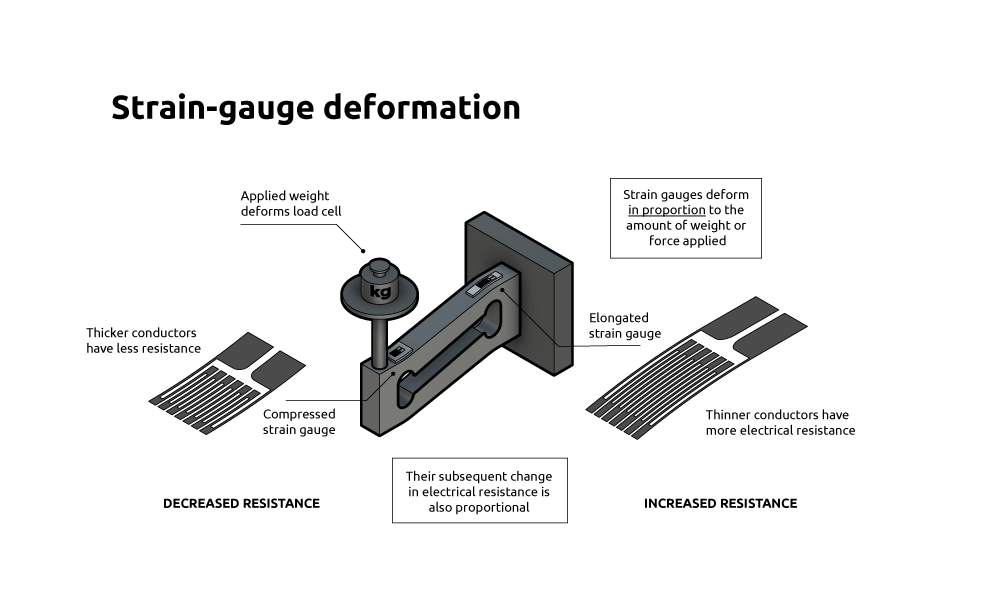


Figure 1.2. Strain-gauge loadcell principle [4]

Figure 2.2 illustrates the **working principle of loadcell, which can be described as**:

* **Force Application:** When a force or weight is applied to the load cell, it causes a slight deformation (strain) in the load cell’s structure. The strain gauges bonded to the load cell also deform. This deformation changes their electrical resistance because of the changes in wire diameter and length.
* **Wheatstone Bridge Circuit:** The strain gauges are typically arranged in a Wheatstone bridge configuration. In a balanced bridge (without load), the output voltage is zero.

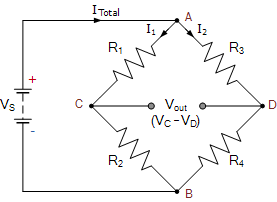


Figure 1.3. Wheatstone Bridge circuit [4]

According to reference [1], the output voltage can be determined as the following formula:

When strain occurs due to the applied load, the resistance changes, unbalancing the bridge and creating a small output voltage proportional to the strain. However, for the output voltage to be readable by the microcontroller, the signal must be amplified.

## . 24-Bit Analog to Digital Converter

The output of the loadcell is small analog signal. For the system to receive the change in load, the signal must be amplified and convert to digital signal. Hence, an Analog to Digital Converter (ADC) is needed to be implement is this project.

### . 24-Bit ADC HX712

The HX712 is specifically designed for applications requiring precise weight measurement, such as load cell-based weighing scales. It amplifies and converts the small analog signals from load cells into digital data that can be processed by microcontrollers or other digital systems.

**According to Fig. 1. 4, the workflow of the HX712 can be listed as follows:**

* **Signal Input**: The HX712 receives analog voltage signals from the load cell. The load cell typically outputs a differential signal, which provides two signals that are opposite in polarity.
* **Amplification**: The HX712 includes an internal programmable gain amplifier (PGA) of 128 that amplifies the small input signals from the load cell.
* **Analog-to-Digital Conversion**: After amplification, the HX712 converts the amplified analog signal into a digital value using a 24-bit ADC.
* **Digital Output**: The HX712 outputs the digital data via a serial interface. The output data rate can be adjusted (10 Hz, 40 Hz) allowing the system to balance between speed and precision based on the requirements of the application.

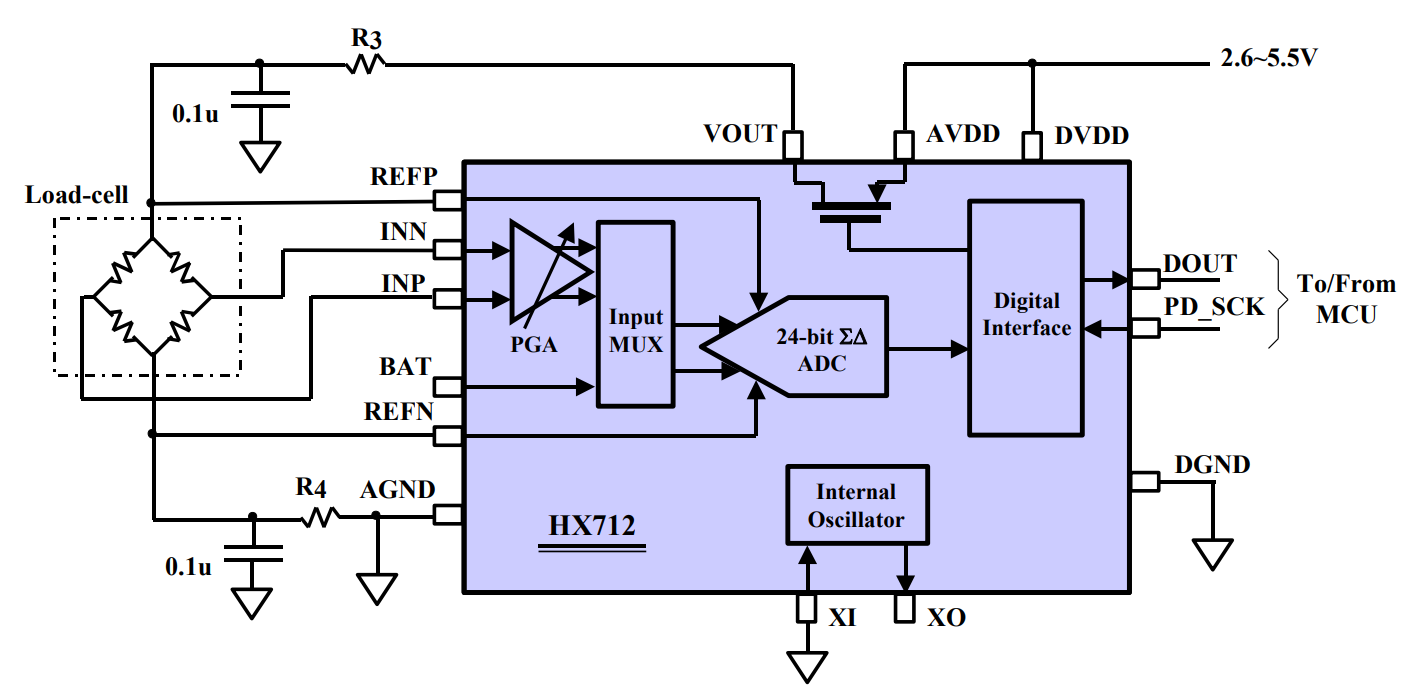


Figure 1.4. HX712 Typical weigh scale application block diagram

Some characteristics of the ADC are considered:

**a) Gain**: When the input signal is small (in, the higher the gain is, the better measured value, this is due to several reasons:

* **Low Signal Strength**: These small signals can be difficult for the ADC to detect accurately. ADCs have a fixed resolution, expressed in bits (24-bit). To make full use of this resolution, the input signal should span the ADC's input range as fully as possible.   
   The full-scale input range of ADC is, with choosing gain :

The loadcell rated output range of a 5kg loadcell is, with :

* **Improving Signal-to-Noise Ratio (SNR)**: Amplifying the signal increases the amplitude of the useful signal without increasing the noise as much. This leads to a better signal-to-noise ratio, which improves the accuracy of the ADC conversion and the quality of the data read by the MCU. This can be proven with the datasheet of HX711, shown in Fig. 1. 5, which has selectable gain of 32, 64 and 128.

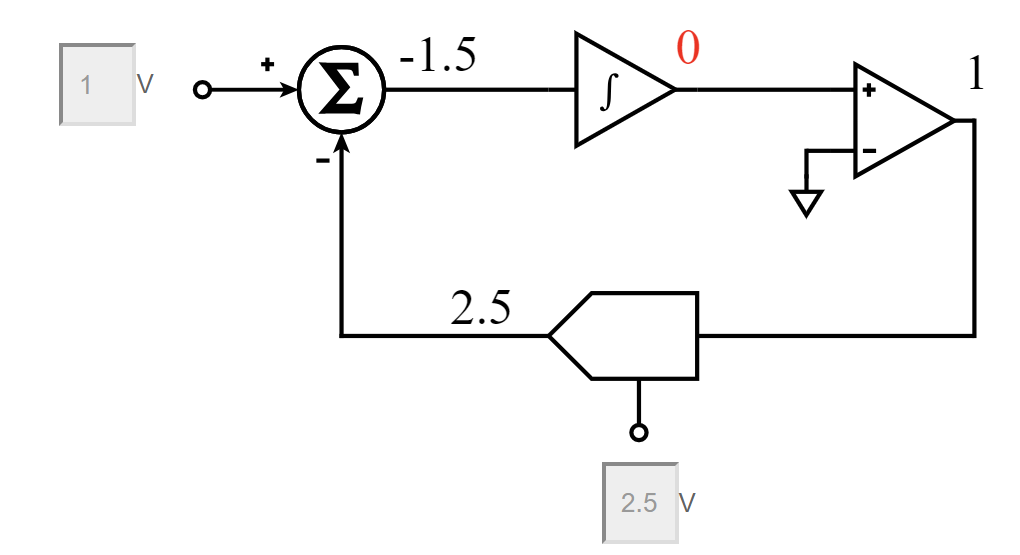


Figure 1.5. HX711 input offset drift with different gains

b) 24-bit Delta/Sigma (**)** ADC:

The HX712 utilizes an 24 bit Delta/Sigma ADC, which processing flow (demonstrated in Fig. 1.6) can be describe as follows:

The input voltage VIN is first summed with the output of a feedback DAC. This summing can be accomplished by means of a switched capacitor circuit which accumulates charge onto a capacitor summing node. An integrator then adds the output of this summing node to a value it has stored from the previous integration step. A comparator outputs a logic 1 if the integrator output is greater than or equal to zero volts and a logic 0 otherwise. A 1-bit DAC feeds the output of the comparator back to the summing node: +VREF for logic 1 and -VREF for logic 0. This feedback tries to keep the integrator output at zero by making the ones and zeros output of the comparator equal to the analog input.



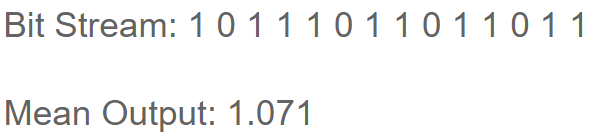


Figure .. Sigma\delta ADC block diagrams

The stream of 1's and 0's is subsequently digitally filtered (not shown) to produce a slower stream of multi-bit samples. The sigma-delta modulator loop typically runs at a much higher frequency than the final output rate of the digital filter. For example, a converter with a 2kHz output data rate may have a modulator loop frequency of over 2.5MHz.

### . Comparison between HX712 and HX711

In the market, HX711 is much more widely use and easier to implement than HX712. However, because the precision requirement for weigh scale, HX711 must be reconsidered for an alternative solution, which is the HX712. The main difference between the HX712 and HX711 is the absence of a bandgap reference in the HX711 circuitry, as shown in Fig. 1.4 and Fig. 1.7. In contrast, the HX712 obtains its reference directly from the power source. By avoiding the need for additional wiring and pass-through components, the direct reference can avoid hardware noise, whereas the HX711 reference must pass through integrated components, affected by hardware efficiency.

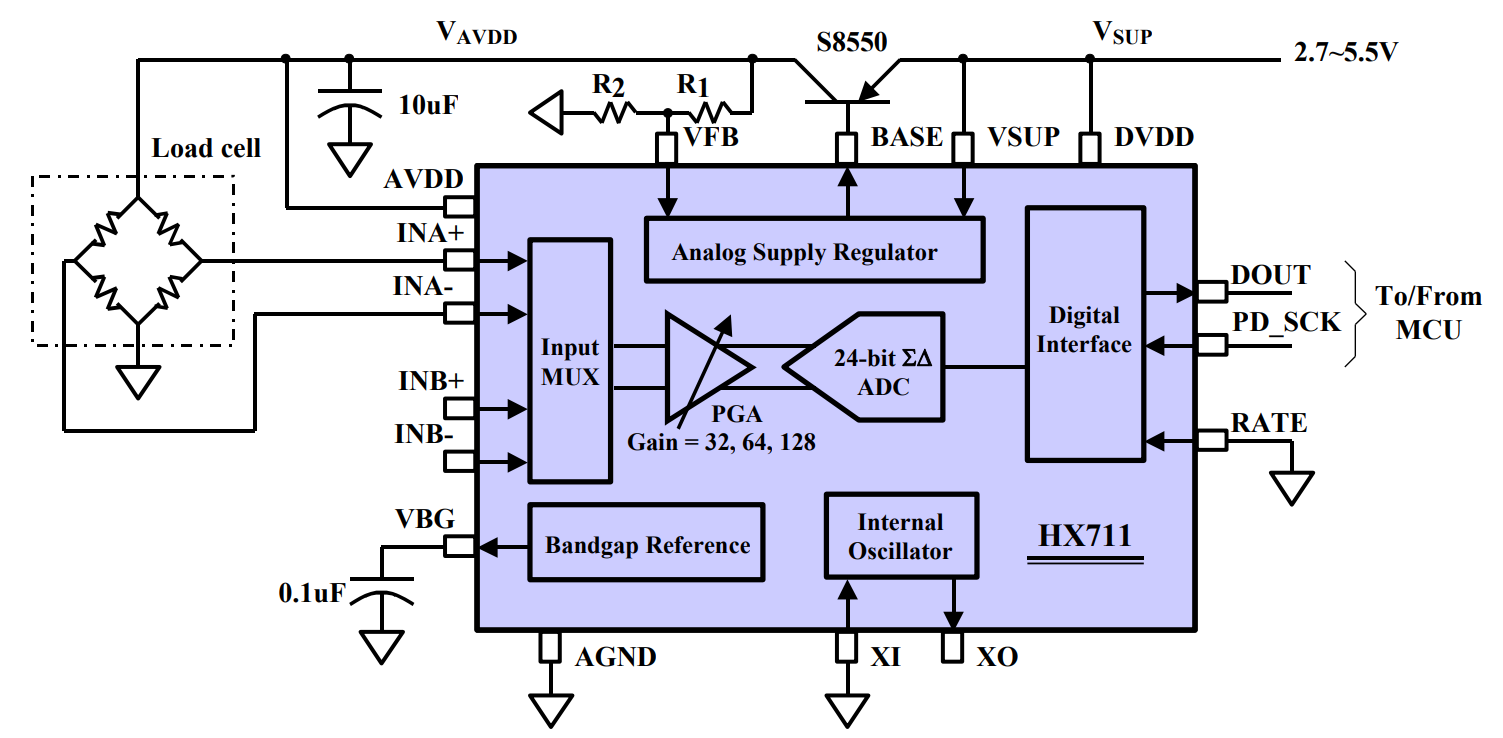


Figure 1.7. HX711 Typical weigh scale application block diagram

**Table 1.1. *Key characteristic comparison between HX711 and HX712***

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Models** | |
| **HX711** | **HX712** |
| Gain |  |  |
| Sampling rate |  |  |
| Power |  |  |
| Noise | Lower resistance | Higher resistance |

From the comparison in Table 2.1, the HX712 provides more flexibility in gain settings and sampling rates, better noise performance, and lower power consumption compared to the HX711, making it a better choice for more demanding or power-sensitive applications.

### . HX712 Data Acquisition

Based on HX712 datasheet, Pin PD\_SCK and DOUT are used for data retrieval, input selection, gain selection and power down controls. Figure 2.6 and Table 2.2 indicates the communication sequence between MCU and HX712.

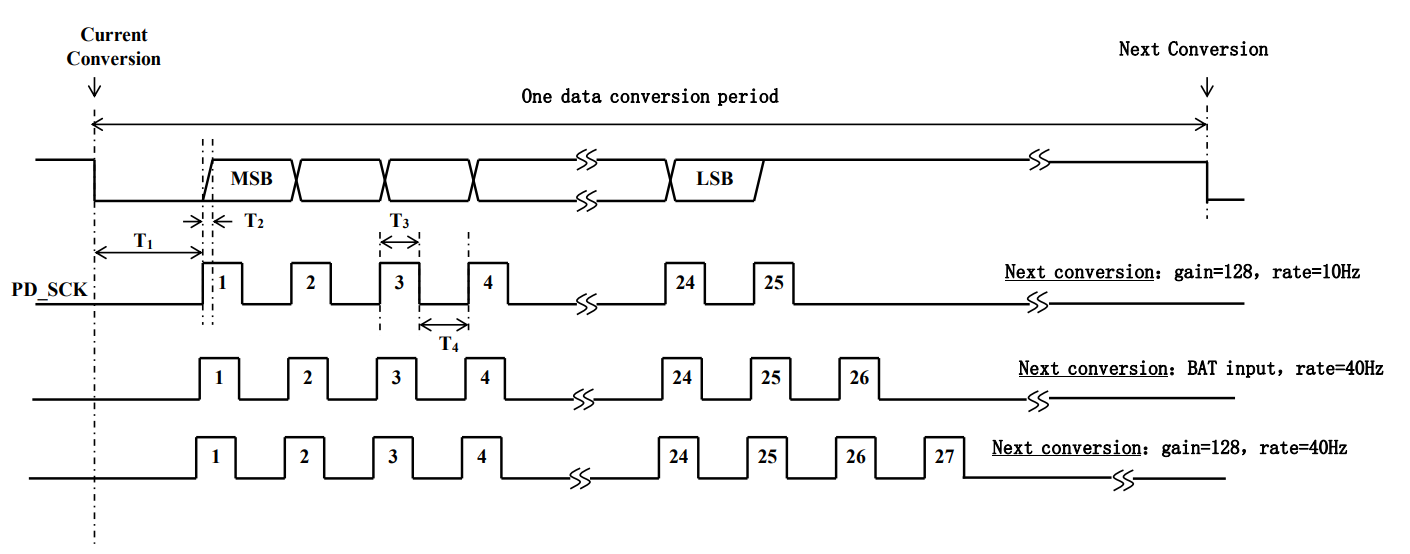
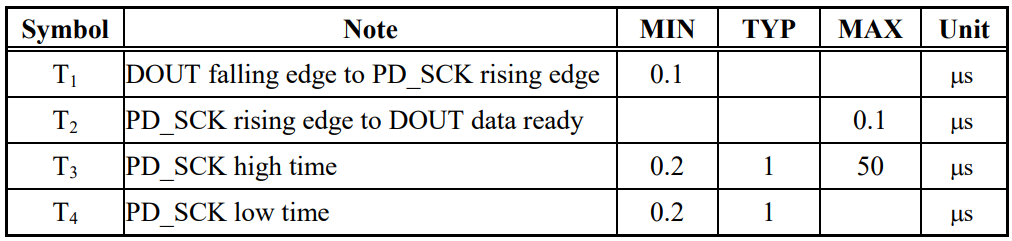


Figure 1.8. HX712 data acquisition communication sequence

**Table 1.2. *Record type used this the project***



According to Fig. 1.6, the data acquisition process can be divided into three main steps: waiting for the data to be ready, reading the data bit by bit for 24 bits, and then performing data conversion and setting up for the next read attempt. Fig. 1.7 shows the flowchart for reading data and can be described as follows:



Figure 1.9. HX712 data acquisition communication sequence

1. Attempt to read data: Pin PD\_SCK and DOUT are used for data retrieval, input selection, gain selection and power down controls. When output data is not ready for retrieval, digital output pin DOUT is high. Serial clock input PD\_SCK should be low. When DOUT goes to low, it indicates data is ready for retrieval.

**while**(**HAL\_GPIO\_ReadPin**(GPIOB, GPIO\_PIN\_8) == *GPIO\_PIN\_SET*)

{

**if** (tick < 100000) tick++;

**else** **return** 0;

}

*Implement code explain: Using a while loop to read DOUT, if it is HIGH (GPIO\_PIN\_SET), system will wait. If reach time outs, loop will be break to execute other applications.*

2. Read bits by bits: By applying 24 positive clock pulses at the PD\_SCK pin, data is shifted out from the DOUT output pin. Each PD\_SCK pulse shifts out one bit, starting with the MSB bit first, until all 24 bits are shifted out.

**for**(**int8\_t** len=0; len<24 ; len++)

{

**HAL\_GPIO\_WritePin**(GPIOB, GPIO\_PIN\_9, *GPIO\_PIN\_SET*);

**microDelay**(1);

data = data << 1;

**HAL\_GPIO\_WritePin**(GPIOB, GPIO\_PIN\_9, *GPIO\_PIN\_RESET*);

**microDelay**(1);

**if**(**HAL\_GPIO\_ReadPin**(GPIOB, GPIO\_PIN\_8) == *GPIO\_PIN\_SET*)

data ++;

}

*Implement code explain: loop for 24 times, each time will read data from ADC using PD\_SCK pin and read response in DOUT pin. Data will be shifted bits by bits until 24 bits are read.*

3. Set sampling rate: The 25th pulse at PD\_SCK input will pull DOUT pin back to high. Input and gain selection is controlled by the number of the input PD\_SCK pulses (Table 3). PD\_SCK clock pulses should not be less than 25 or more than 27 within one conversion period, to avoid causing serial communication error. Additionally, the output 24 bits of data is in 2’s complement format. When input differential signal goes out of the 24-bit range, the output data will be saturated at 800000h (MIN) or 7FFFFFh (MAX), until the input signal comes back to the input range.

**for** (**int** i = 0; i<speed; i++){

**HAL\_GPIO\_WritePin**(GPIOB, GPIO\_PIN\_9, *GPIO\_PIN\_SET*);

**microDelay**(1);

**HAL\_GPIO\_WritePin**(GPIOB, GPIO\_PIN\_9, *GPIO\_PIN\_RESET*);

**microDelay**(1);

}

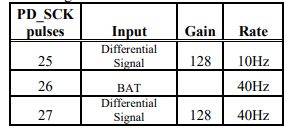
data = data ^ 0x800000;

**return** data;

}

*Implement code explain: loop for 1-3 times, setting gain and sampling rate selection for next sample. Data will be decoded from 2’s complement.*

**Table 1.3. *Inputs, Rate and Gain selection***



After the data is received, it is sent to Master MCU to be processed, calibrated and output to monitor:

double int\_to\_gram() {

  int32\_t total = 0;

  int32\_t samples = 2;

  long wo\_load\_coef = 7718930;

  float milligram;

  float coef = 1;

  coef = -0.00131;

  for (uint16\_t i = 0; i < samples; i++) {

    total += get\_value();

  }

  double average = (float)(total / samples);

  milligram = (average - wo\_load\_coef) \* coef;

  return milligram;

}

*Implement code explain: value is read and get averaged (ex. 2 times), then convert to more meaningful value (milligram) by adding offset and coefficient. The constants are found by trials and errors.*

## . Communication protocols

### . Implementing UART for Host computer and Arduino communication

For communication between the host computer and the master microcontroller Arduino, UART (Universal Asynchronous Receiver-Transmitter) is chosen because it requires no additional hardware to connect the two devices.

In definition, UART is a hardware communication protocol used for asynchronous serial communication between devices. It converts parallel data from a computer into a serial form for transmission to MCU and vice versa. Some key features can be listed as follows:

1. Asynchronous Communication: UART does not use a clock signal to synchronize the transmitter and receiver. Instead, it uses start and stop bits to signify the beginning and end of data packets.

2. Data Format: Data is sent in packets that include a start bit, a data frame (8 bits without parity bit) as shown in Fig. 2.1.

3. Baud Rate: UART communication speed is defined by the baud rate, which is the number of signal changes or symbols per second. Both the transmitter and receiver must be set to the same baud rate. For this application, the highest possible speed is desired. Considering Arduino Nano has the maximum theorical baud rate of . In reality, the maximum value hardly returns consistent results. Hence, the baud rate selected for this application is .

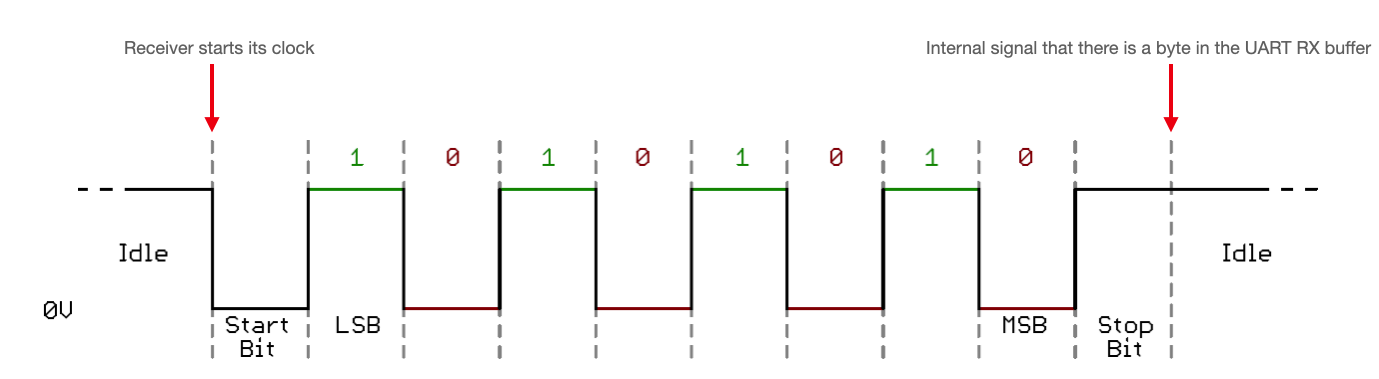


Figure 1.10. UART communication sequence

**Implemented code:**

* Host Computer (Python):

def send\_data(COM, data\_to\_send):

    global arduino\_res

    COM.write(bytes([data\_to\_send])) # Write to Arduino

    ticks = 0

    while (ticks < 30000):  # > 30000 when plugged in

        ticks += 1

arduino\_res = int(COM.readline().decode()) # read Arduino message

Note: “ticks loop” is to wait for the Arduino receive and process the data before sending new value. The value (ex. 30000) is determined not by calculation due to the nature of Windows OS and non-constant CPU clock speed. Hence, it is chosen by trials and errors.

* Arduino (C):

void setup() {

  Serial.begin(76800, SERIAL\_8N1);  // start serial for output 234000/5

}

void loop() {

    while (Serial.available() > 0) {

      InBytes = Serial.read();  // read byte from PC

}

Serial.print(404); // write byte to PC

}

### . Implementing I2C for Arduino and STM32 communication

1. **I2C communication overview**

I2C, or Inter-Integrated Circuit, is a synchronous, multi-master, multi-slave, packet-switched, single-ended, serial communication bus commonly used in embedded systems and microcontroller applications. I2C communication sequence, illustrated in Fig. 1.11: The data transfer rate is determined by SCL signal line, which is controlled by the Master. For each pulse in the SCL, the Master/Slave will response in the SDA data line. The sequence can be described as follows:

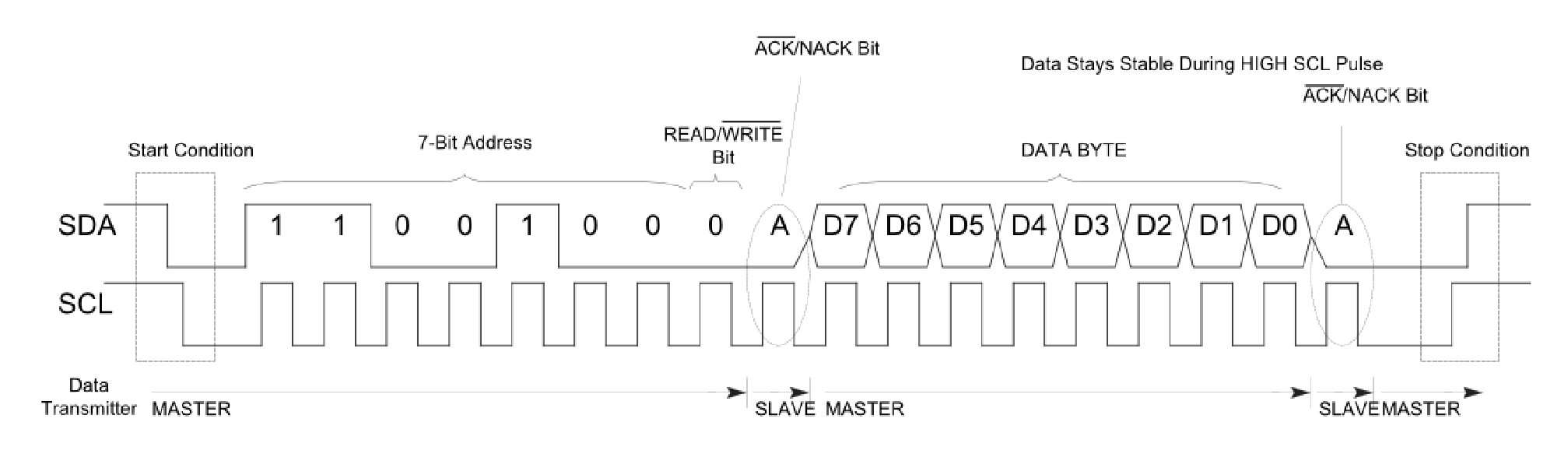


Figure 1.11. I2C communication sequence

Step 1. Start Condition: The master initiates communication by pulling the SDA line low while keeping the SCL line high.

Step 2. Address Frame: After the start condition, the master sends a 7-bit or 10-bit address frame. This specifies the slave device the master wants to communicate with. This is followed by a read/write bit, indicating the direction of data transfer.

Step 3. Acknowledge (ACK) Bit: After the address frame, the addressed slave device must acknowledge receipt by pulling the SDA line low during the acknowledge clock pulse.

Step 4. Data Transfer: Data is transferred in bytes, which are followed by an acknowledgment bit. The master can send multiple bytes to the slave (in write mode), or the slave can send multiple bytes to the master (in read mode).

Step 5. Stop Condition: To end the communication, the master generates a stop condition by releasing the SDA line to high while the SCL line is high.

Some notable features used in this application are:

* Addressing: For this application, the STM32’s slave address is set as 0x08.
* Data Transfer Rate: STM32F103C8T6 I2C supports various data transfer rates, including standard mode (up to ), fast mode (up to ). Arduino Nano by default uses fast mode (). However, because the utilized board is defected, which bring all clock speed down by times, as the same as to standard mode. Hence why the STM32F103C8T6 is set at standard mode.

1. I2C pull up resistors

According to [3], the pull up resistor can be calculated as follows:

The maximum pullup resistance is a function of the maximum rise time ():

The minimum pullup resistance is a function of and :

The parameters used in (3.1) and (3.2) is listed in Table 3.1:

**Table 1.4. *Record type used this the project***





Comment: The higher the transfer speed, the lower the resistors value. This is to ensure that the internal capacitors are charged fast enough.

1. **Implementing code:**

* STM32 I2C Settings

**static** **void** **MX\_I2C1\_Init**(**void**)

{

hi2c1.Instance = I2C1;

hi2c1.Init.ClockSpeed = 100000; //Clockspeed

hi2c1.Init.DutyCycle = I2C\_DUTYCYCLE\_2;

hi2c1.Init.OwnAddress1 = 16; //Slave address

hi2c1.Init.AddressingMode = I2C\_ADDRESSINGMODE\_7BIT;

hi2c1.Init.DualAddressMode = I2C\_DUALADDRESS\_DISABLE;

hi2c1.Init.OwnAddress2 = 0;

hi2c1.Init.GeneralCallMode = I2C\_GENERALCALL\_DISABLE;

hi2c1.Init.NoStretchMode = I2C\_NOSTRETCH\_ENABLE;

}

* STM32 I2C responses ACK

**void** **HAL\_I2C\_ListenCpltCallback**(**I2C\_HandleTypeDef** \*hi2c){

**HAL\_I2C\_EnableListen\_IT**(hi2c);

}

* STM32 transmits/receives data to Master

**void** **HAL\_I2C\_AddrCallback**(**I2C\_HandleTypeDef** \*hi2c, **uint8\_t** TransferDirection, **uint16\_t** AddrMatchCode)

{

**if** (TransferDirection == I2C\_DIRECTION\_RECEIVE){

**HAL\_I2C\_Slave\_Seq\_Transmit\_IT**(&hi2c1, TxData, TXDATA\_SIZE, I2C\_FIRST\_FRAME);

} **else** **if** (TransferDirection == I2C\_DIRECTION\_TRANSMIT){

**HAL\_I2C\_Slave\_Seq\_Receive\_IT**(&hi2c1, receive\_data, 1, I2C\_FIRST\_AND\_LAST\_FRAME);

receive\_signal\_command = 1;

}

}

Note: Transfer Data size is 10 bytes, while the size of received byte is 1 byte.

## . Data Transmission Flowcharts

After the data is collected from the slave MCU, it will store the data, which will be sent when the Master request the data. The data is updated continuously regardless of data request from Master. When the Host computer command weigh value request, Master MCU will retrieve data from Slave, calibrate and send back the data to Computer



Figure 1.12. Host computer, Master and Slave MCUs data acquisition flowchart

## . Experiment and Evaluation

### . Experiment results

First, the Host computer will send read data request to the Master, goes into weight scale reading mode.

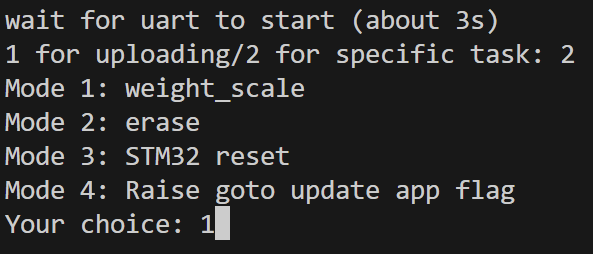


Figure 1.13. Co

The experiment involves measuring the weight of 6 random objects (examples shown in Fig. 1.14), including when the scale is under load. The sampling rate of the HX712 is set at 10Hz, and averaging rate of the algorithms is two samples per output, which. Note that the goal of this experiment is to evaluate the noise in each measurement. The accuracy of object load and the ability to reduce the measured value to zero when there is no load will be addressed in future developments.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| a) No load | b) Load with | c) Load with |

Figure 1.14. Measurement results with different objects

### . Results Evaluation

A precise measurement has two requirements: a fine-tuned calibration and a measured value that must not change dramatically. For this assignment, the first requirement is assumingly met, the later requirement evaluation is carried out in this project. To evaluate performance, a sample space of 50 is taken for each object. From the dataset, the mean (), standard deviation (), and variance are determined. The relative errorcan then be calculated as follows:

To meet the requirement, the relative error must be less than .

Figure 1.15 shows the dataset of weights corresponding to the relative error. From this, it can be concluded that objects heavier than approximately 55 grams meet the requirements, with the error decreasing as weight increases. The highest recorded error in the dataset is 8.325%, observed when there is almost no load. The lowest error, 0.007%, was recorded when the load was approximately 769.9 grams.



Figure 1.15. UART communication sequence

To further evaluation, standard deviation of each measurement is also considered. From the Fig. 1.16, the value is ranging from 0.05 to 0.1g. Consider the maximum record value is the baseline that the load measurement to be acceptable. The minimum load value within acceptable error is:



Figure 1.16. UART communication sequence

If reducing the minimum value is necessary, the standard deviation must be lower by increasing more sample for averaging, which lowers the sampling rate.

# : BOOTLOADER

## . Introduction

A bootloader is a piece of code which allows user application code to be updated. The new code can be obtained using alternative download channels, such as a USB stick or a network port. After the boot ROM's execution, the bootloader is executed and will do the update when required and then execute the end-user application.

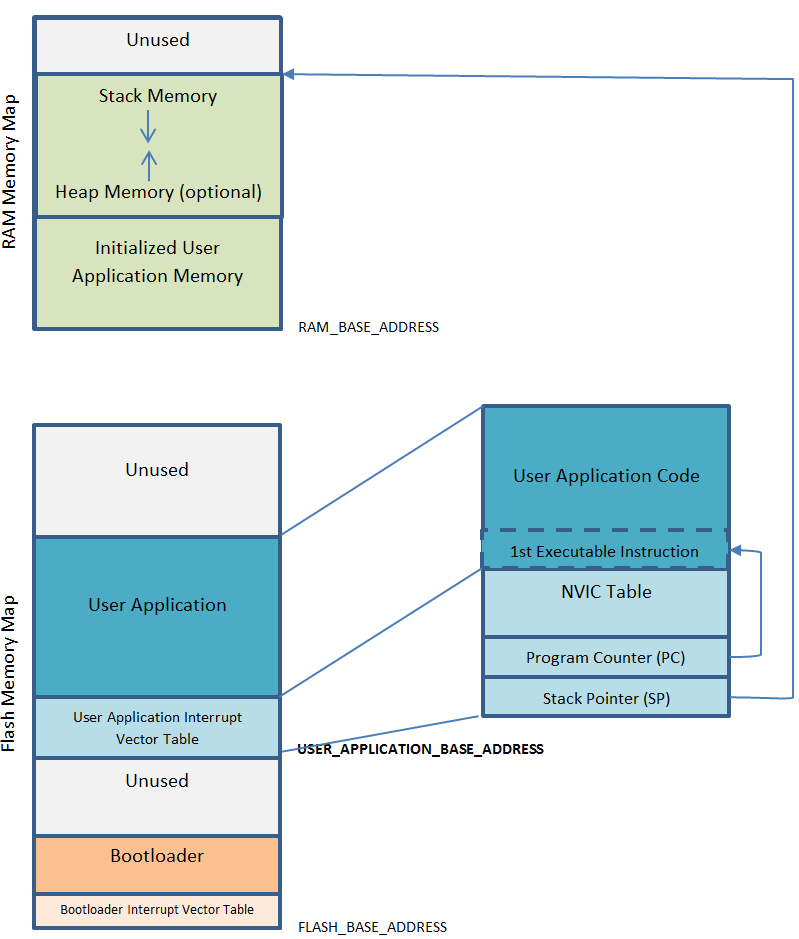


Figure 2.1. ARM Memory Maps [2]

The bootloader and the user application should be written and built as two separate projects or targets, resulting in two separate and executable images/applications. The main tasks of the bootloader are to reprogram/replace the user application, if necessary, and to jump to the user application to execute it. The user application doesn't necessarily need to know the existence of the bootloader.

The bootloader is usually placed at the chips flash base address (demonstrated in Fig. 2.1), so that it will be executed by the CPU after reset. The following figure demonstrates a typical code placement of the user application and the bootloader.

## . ARM boot sequence

The boot sequence can be divided into four parts:

* Power on Reset (Hardware Process)
* Memory Aliasing (Remapping) and Architecture (Hardware Process)
* Firmware Booting (Hardware and Firmware Process)
* Reset\_Handler function execution (Firmware Process)

1. **Power on Reset**

The boot process starts with the Power-On Reset (POR). When an MCU is initially powered on, it performs predefined hardware checks to properly initialize the system.

1. **Memory Aliasing and Architecture**

After the POR Stage MCU comes out of the reset state. The processor starts pointing from the 0x00000000 address. This zero address can also be remapped to any other address in the address space with the help of Remapper hardware. This boot setting for Remapper hardware is typically configured from the Boot Mode selection pin of the MCU. Processor internal circuitry reads this boot mode pin configuration to set the Remapper address as per the selected configuration. In STM32, the boot setting is shown in Figs. 2.2-2.3:

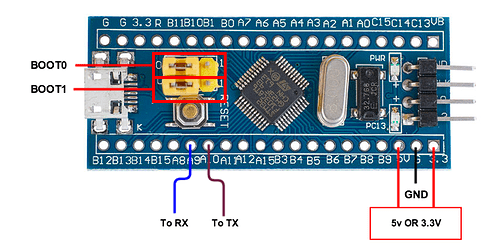


Figure 2.2. STM32F103C8T6 Boot configuration pins

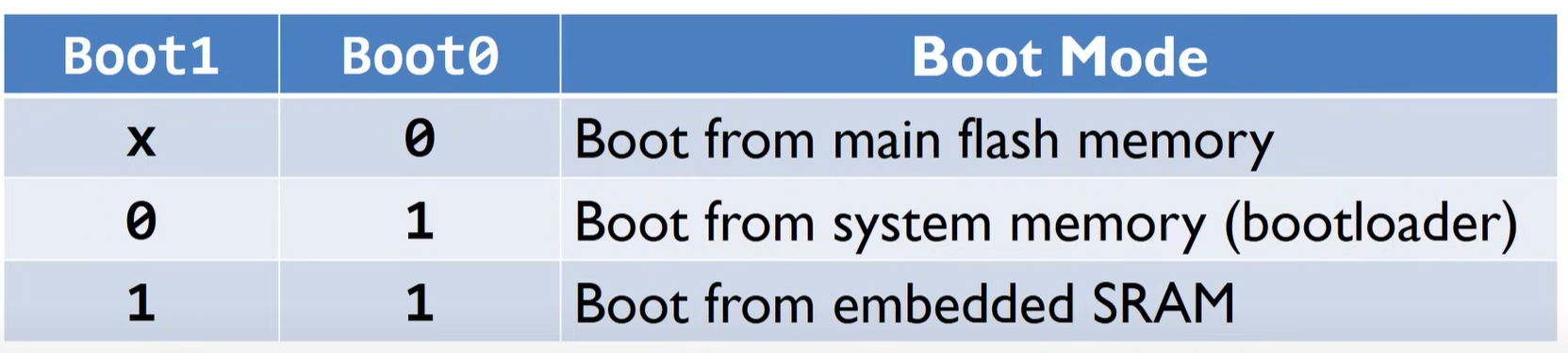


Figure 2.3. STM32F103C8T6 Boot mode

1. **Firmware Booting**

The firmware boot process begins by fetching a 32-bit memory address to load into the Program Counter (PC). Initially, the PC is set to the value at address 0x00000000. In an ARM-based MCU, 0x00000000 typically corresponds to the top of the stack. The PC then points to the next address, 0x00000004, which contains the Reset Handler, the first piece of code to execute.

1. **Reset\_Handler function execution**

The Reset Handler defined in the startup file performs the following tasks:

1. Initializes the stacks and CPU registers.

2. Copies data segments from FLASH to RAM and sets BSS segment.

3. Configures peripherals to their default states.

4. Jumps to the `main` function to begin program execution.

## . Project Memory Map Structure



Figure 2.4. Project Flash Memory Map

In this project, Flash memory is divided into four sections (as shown in Fig. 2.4):

* Bootloader: placed at the start of the Flash, at address 0x08000000, with size of 32KB. This section contains the bootloader code responsible for initializing the system and determining whether to load the main application or update the application based on specific indicators.
* Main Application: placed after the bootloader, at address 0x08008000, with size of 32KB. This section holds the primary software application that runs on the MCU during normal operation.
* Update Application: placed after the main application, at address 0x08010000, with size of 32KB. This section stores the update version of the main application, which can be loaded directly into or to replace the existing main application when an update is required.
* Indicators: placed after the update application, at address 0x08018000, with size of 32KB. This section contains specific memory addresses used by the bootloader to determine the next steps, such as whether to load the update application or the main application, and whether to erase and replace the main application.

The memory section is defined in the linker script:

Memory allocation:

MEMORY

{

RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 20K

FLASH (rx) : ORIGIN = 0x8000000, LENGTH = 32K

}

Main Application:

MEMORY

{

RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 20K

FLASH (rx) : ORIGIN = 0x8008000, LENGTH = 32K

MY\_MEMORY (rx) : ORIGIN = 0x8018000, LENGTH =32K

}

Update Application:

MEMORY

{

RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 20K

FLASH (rx) : ORIGIN = 0x8010000, LENGTH = 32K

}

## . Bootloader sequence

### . Bootloader flowchart

`

Figure 2.5. Bootloader flowchart

All control commands are entered into the computer and sent back to the MCUs. The Slave MCUs only need to read the indicators written in memory. The bootloader of the Slave MCU reads indicators from specific memory addresses: 0x08001F00 and 0x08001F01.

* If the value at memory address 0x08001F00 is 0x01, the program counter points to the update application; otherwise, the bootloader loads the main application by default.
* If the value at memory address 0x08001F01 is 0x01, the main application image is erased and replaced by the update application, which is also deleted after successful data migration.

### . Jumping/Updating application

1. **Jump to Main/Update Application:**

The process starts at checking if the value at the application address is valid. The mask 0x2FFE0000 is applied, and the result is compared to 0x20000000. This check is used to ensure that the memory contains a valid stack pointer.

If the code is valid, the function disables all interrupts. JumpAddress is set to the value stored at the application address + 4, which is typically the reset vector (the entry point) of the application.

JumpToApplication is then cast to a function pointer pointing to this JumpAddress. The main stack pointer (MSP) is set to the value at application address. The code then jumps to the update application by calling the function pointer JumpToApplication, effectively transferring control to the new application.

**void** **go\_to\_update\_app**(**void**){

**uint32\_t** JumpAddress;

**pfunction** Jump\_To\_Application;

//check for code

**if** (((\*(**uint32\_t**\*) FLASH\_UPDATE\_APP\_ADDR) & 0x2FFE0000) == 0x20000000){

**\_\_disable\_irq**();

JumpAddress = \*(**uint32\_t** \*) (FLASH\_UPDATE\_APP\_ADDR + 4);

Jump\_To\_Application = (**pfunction**) JumpAddress;

**\_\_set\_MSP**(\*(**uint32\_t**\*) FLASH\_UPDATE\_APP\_ADDR);

Jump\_To\_Application();

}

**HAL\_GPIO\_WritePin**(GPIOB, GPIO\_PIN\_1, *GPIO\_PIN\_SET*);

}

}

1. **Immigrating Update Application Image to Main Application Image:**

First, the indicators are reset after the data is read successfully

Secondly, erase the main memory section: This step requires the Flash memory to be unlocked for writing and erasing. Then the while loop erases the main application in Flash memory page by page. Each page is 1 KB (0x400 bytes) in size. The PER (Page Erase) bit in the Flash control register is cleared to complete the erase operation. Finally, the Flash memory is locked again to prevent unintended writes. SHOW DATASHEET

Finally, copy the update application and write the main application: Same as previously, the Flash memory is unlocked for programming. Inside the loop, data is read byte by byte from READ\_ADDRESS and then combined into a 32-bit word, which writes this 32-bit word to Flash memory. The WRITE\_ADDRESS is incremented by 4 bytes after each write operation. The Flash memory is locked again after programming.

**void** **update\_main\_app**(**void**){

**uint32\_t** READ\_ADDRESS = 0x08010000;

**uint32\_t** WRITE\_ADDRESS = 0x08008000;

**uint32\_t** result = 0;

**\_\_disable\_irq** ();

// clear flag

**HAL\_FLASH\_Unlock**();

**FLASH\_PageErase**(UPDMAIN\_FLAG\_ADDR);

CLEAR\_BIT(FLASH->CR, (FLASH\_CR\_PER));

**HAL\_FLASH\_Lock**();

// ERASE MAIN APPLICATION

**HAL\_FLASH\_Unlock**();

**while**(WRITE\_ADDRESS < 0x08010000){

**FLASH\_PageErase**(WRITE\_ADDRESS);

CLEAR\_BIT (FLASH->CR, (FLASH\_CR\_PER));

WRITE\_ADDRESS += 0x00000400; //1 PAGE

}

**HAL\_FLASH\_Lock**();

**HAL\_GPIO\_WritePin**(GPIOA, GPIO\_PIN\_7, *GPIO\_PIN\_SET*);

// WRITE MAIN APPLICATION

**HAL\_FLASH\_Unlock**();

WRITE\_ADDRESS = 0x08008000;

**while**(WRITE\_ADDRESS < 0x08010000){

**for** (**int** i = 0; i<4; i++){

read\_data[i] = \*(**uint8\_t** \*) READ\_ADDRESS;

READ\_ADDRESS++;

}

result = (read\_data[3] << 24) | (read\_data[2] << 16) | (read\_data[1] << 8) | read\_data[0];

**HAL\_FLASH\_Program**(FLASH\_TYPEPROGRAM\_WORD, WRITE\_ADDRESS, result);

CLEAR\_BIT (FLASH->CR, (FLASH\_CR\_PG));

WRITE\_ADDRESS+= 0x04;

result = 0;

}

**HAL\_GPIO\_WritePin**(GPIOA, GPIO\_PIN\_7, *GPIO\_PIN\_RESET*);

**HAL\_FLASH\_Lock**();

**enable\_irq**();

}

## . Data Transmission Process

### . Main Application Flowchart

The flowchart shown in Fig. 2.6 outlines the process flow for a microcontroller's main application, which normally reading weight scale value or receiving firmware updates when is called. Below is a step-by-step explanation of the flowchart:

The system first checks for any update indicators to determine if a firmware update process should be initiated. If an upload signal is received, it reads data from the master and begins the update process. If no upload signal is received, the process continues with the load cell scale application. The MCU check for verification by matching security key. If the data passes verification, the system proceeds to write the data to Flash memory. If at any point the verification fails, the system sends a "Verification failed" message back to the master device.



Figure 2.6. Weight scale and Upload Application in Main Application flowchart

### . Intel HEX file structure

The input of the application file is intended to be in Intel HEX file structure, which include 6 parts (examples shown in Fig. 2.7):

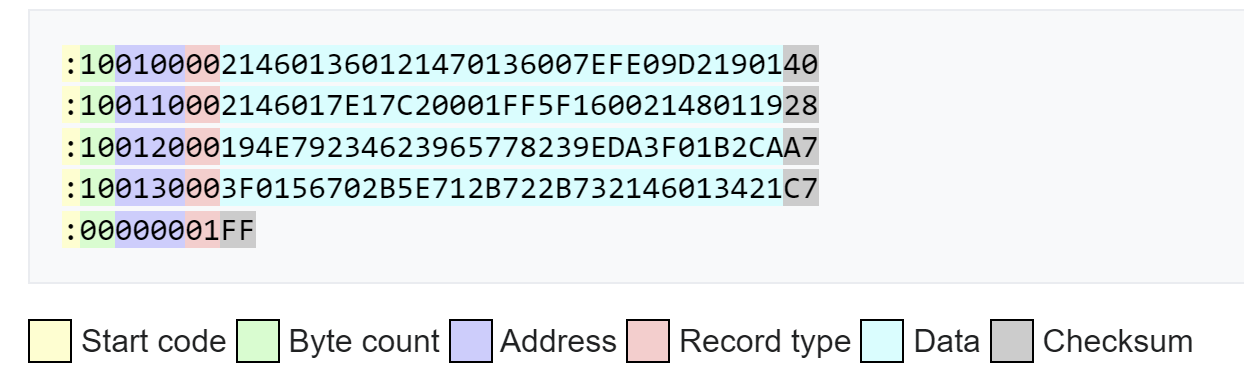


Figure 2.7. Intel HEX file structure examples

* Start code: one character, always an ASCII colon ":", marking the start of the data.
* Byte count: one hex pair, indicates the number of data in bytes.
* Address: one hex pair, indicates the beginning address of which the data should be recorded. The used values in listed in the Table 3.1:
* Record type: one hex pair, ranging from “00” to “05”, indicates the meaning of data field.
* Data: a sequence of bytes of data.
* Checksum: two hex digits, a computed value used to verify that the record has no errors. A record's checksum byte is the two's complement of the least significant byte (LSB) of the sum of all decoded byte values in the record preceding the checksum (S). The calculated checksum, which will be used to compare with the given checksum, is determined as follows:

**Table 2.1. *Some utilized Record type***

|  |  |  |
| --- | --- | --- |
| **Hex code** | **Record type** | **Description** |
| 00 | Data | The byte count specifies number of data bytes in the record |
| 01 | End Of File | Indicate the last line of hex file |
| 04 | Extended Linear Address | Allows for 32 bit addressing, specify the upper 16 bits. |

**Hex file read code:**

def read\_and\_send\_file(line, file, lines, line\_count):

    global period

    file.seek(-len(line), 1)  # return cursor to start of line

    print(line\_count, "/", len(lines), "|", f'{round(line\_count\*100/len(lines),2):.2f}' + "%", end ="")

    read\_data = file.read(1).decode()

    if read\_data != ":":  # check validation

        print("error")

        return IOError

    else:

        data\_startcode = read\_data               # read ":"

        data\_bytecount = file.read(2).decode()   # read byte count

        data\_address = file.read(4).decode()     # read Address

        data\_recordtype = file.read(2).decode()  # read Recordtype

        data\_datapart = file.read(

            len(line) - 13).decode()             # read data

        data\_checksum = file.read(2).decode()    # read checksum

        data\_EOL = file.read(2).decode()         # bypass /r/n

    # prepare for checksum

    data\_string = data\_bytecount + data\_address + data\_recordtype + data\_datapart

    verify\_checksum(data\_string, data\_checksum)

    # prepare to send data

    data\_string = data\_string + data\_checksum

    data = 0

    j = 0

    for \_ in range(int(len(data\_string)/2)):

        data = int(data\_string[j:j+2], base=16)

        j += 2

        send\_data(COM, data)

**Hex file verification code:**

def verify\_checksum(data\_string, pref\_cs):

    j = 0

    total\_checksum = 0

    # half of len because read 2 chars at a time

    for \_ in range(int(len(data\_string)/2)):

        total\_checksum += int(data\_string[j:j+2], base=16)

        j += 2

    # %256 for return to 0 if reach 256

    calc\_checksum = (256-total\_checksum % 256) % 256

    if (calc\_checksum == int(pref\_cs, base=16)):

        # if success, do nothing, else exit the application

        pass

    else:

        print(data\_string, total\_checksum, (256-total\_checksum %

                                            256) % 256, int(pref\_cs, base=16))

        print("checksum not pass, recheck the HEX file")

        exit()

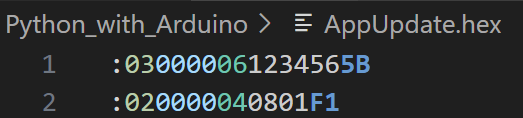
### . Data confirmation

To ensure the integrity and security of data being uploaded to Flash memory, a verification mechanism is essential. In this project, a validation process is implemented using a matching verification key. This method involves several key steps:

* **Predefined Verification Key:** A unique verification key is pre-stored in a secure section of the Flash memory. Also, each Hex file intended for upload is prefixed with a specific verification key. This key is written at the beginning of the file. When a data upload is initiated, the MCU first reads the initial portion of the Hex file.
* **Verification Process:** The MCU compares the extracted key from the Hex file with the predefined key stored in the Flash memory.
* **If Keys Match:** The MCU proceeds with the upload process, transferring the data to the Flash memory in a secure and reliable manner.
* **If Keys Do Not Match:** the MCU identifies the data as untrusted or potentially corrupted. In this case, the upload process is halted, and the data is rejected. This prevents any compromised or unauthorized data from being written to the Flash memory.

**Implementing code:**

**Verification key at the beginning of Hex file**



**Data comparison in Main Application:**

**#define** **pref\_key** 0x12345600

**---------------------------------------------------------------------------**

**case** read\_recordtype:

data[i] = receive\_data[0];

**//...**

**else** **if** (data[i] == 6){

data\_type = 6;

}

**//...**

**break**;

---------------------------------------------------------------------------

**if** (data\_type == 6){

key += data[i];

key = key << 8;

}

---------------------------------------------------------------------------

**if** (key == pref\_key) {

correct\_key = 1;

}

### . Data transmission flowchart

As shown in Fig. 2.8, the data will first be read line by line from the Host computer. Initially, each line will be divided into five parts, and the data will be verified using a checksum. After a successful verification, the data will be sent to the Arduino microcontroller, and the system will wait for data upload responses.

The data will be verified again before and after transmission to the STM32 to ensure the uploaded data is correct. Once a data upload is complete, without any verification failures, a success signal will be sent back to the host computer, which will then proceed to the next line in the dataset. If any verification fails, a signal will be sent back to the host computer to reread the line and resend it to the processing units. This loop continues until the host computer reaches the end of the file (EOF).



Figure 2.8. Data transmission flowcharts

## . Bootloader result

First, the uploading mode is selected and upload hex file is chosen from terminal, a successful file read and confirmation from the MCUs will return no error and sending lines gradually (shown in Fig. 2.9).

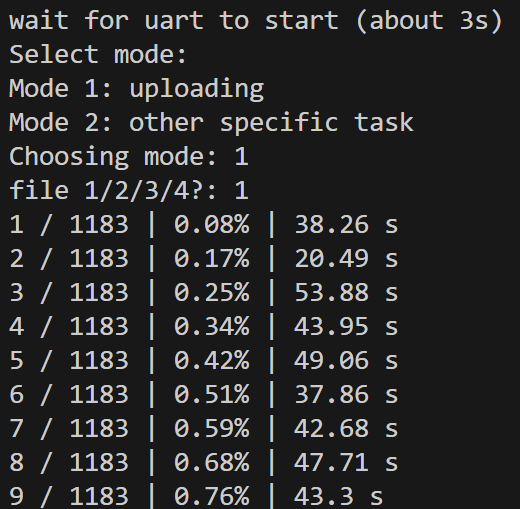


Figure 2.9. Uploading Terminal

In case of any errors, such as checksum fail or no confirmation from slave MCU, master MCU will return error of 404 (from Fig. 2.10), the Host computer will try to resend the data line.

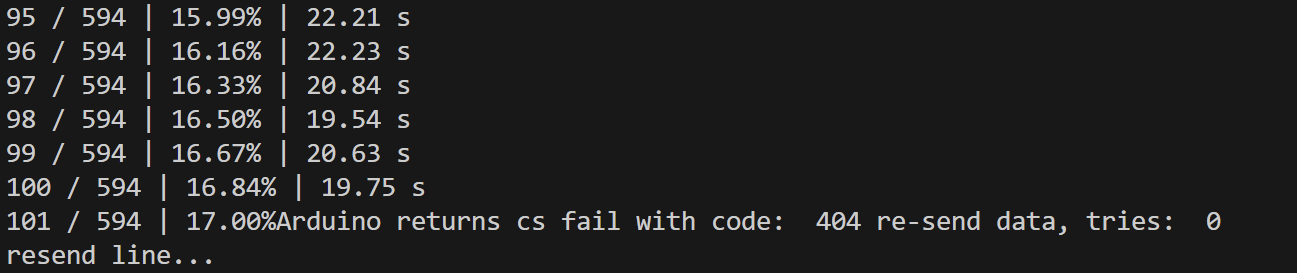


Figure 2.10. Fail data transmission

Figure 2.11 indicates that if the data transmission is complete successfully, Host Computer will print out in terminal “Upload complete”. The devices can be loaded into new image after reset.

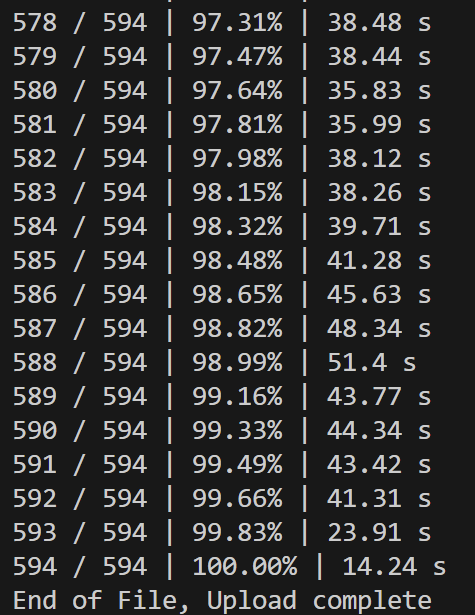
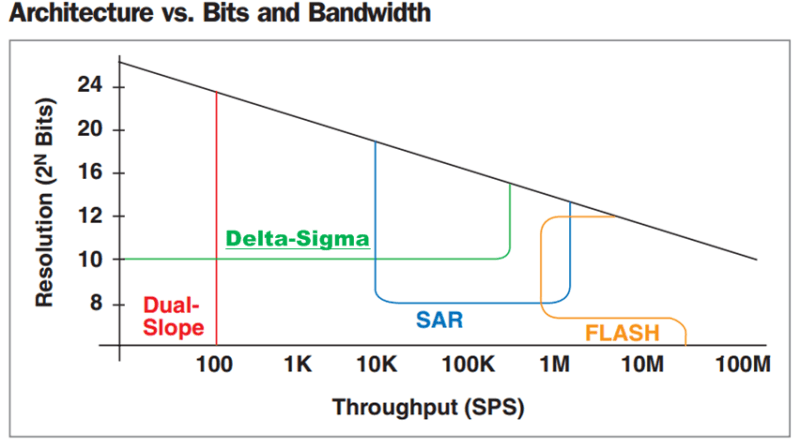


Figure 2.11. Successful data transmission

# APPENDIX A: ADC types



**A.1. Flash**

**A.2. SAR**

**A.3. Sigma/Delta ADC**

**A.4. Dual-Slope**

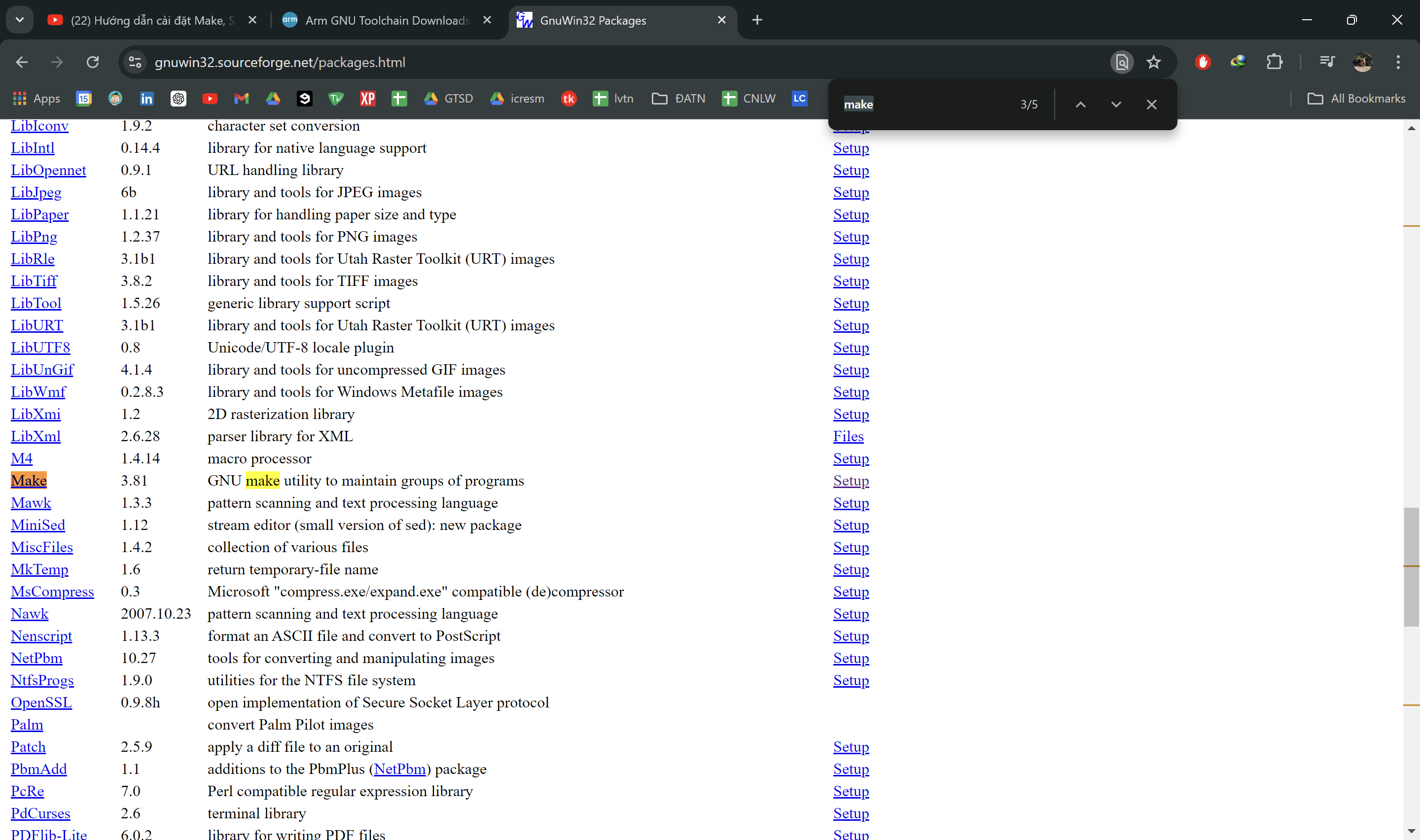
# APPENDIX B: USING STM32 IN VSCODE VIA MAKEFILE AND STLINK

**B.1. Download Make:**

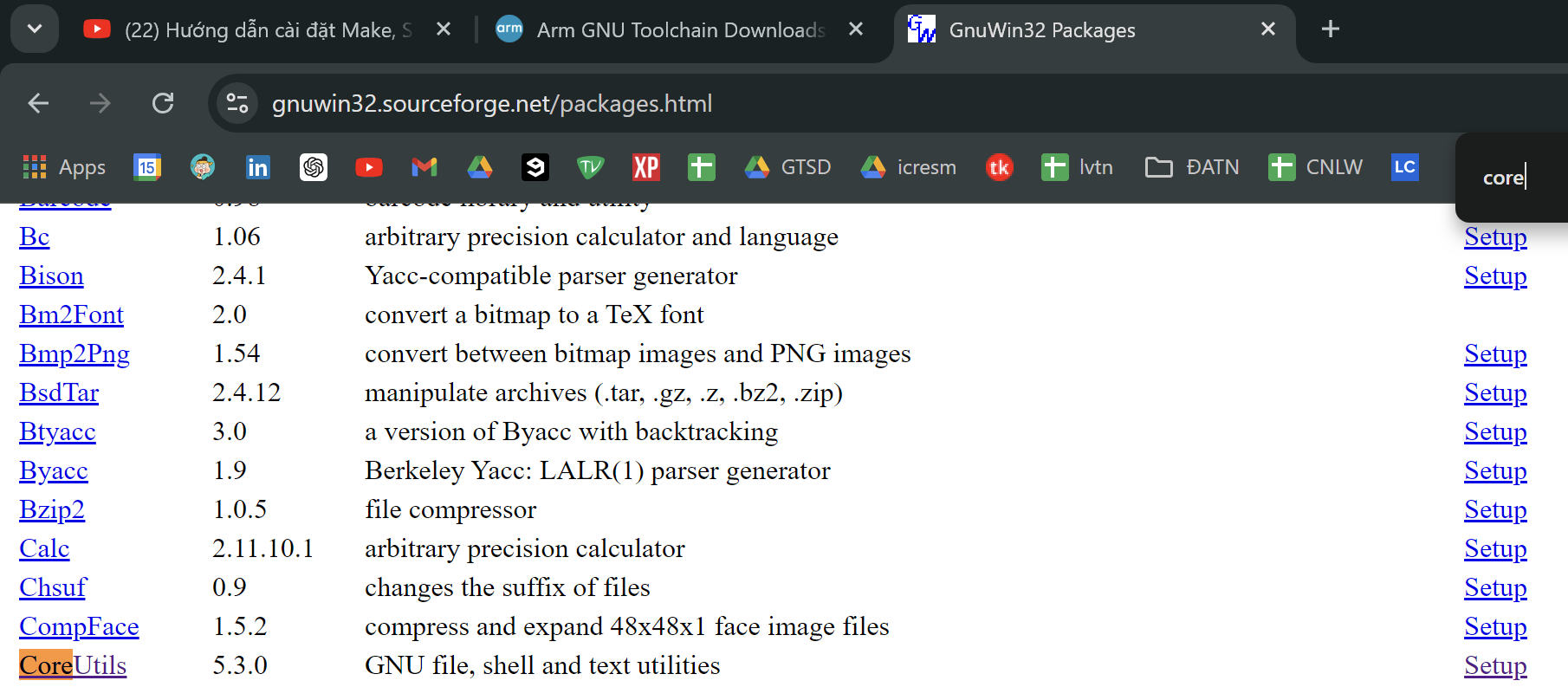
(Based on this video: <https://www.youtube.com/watch?v=NY6pW61U9uA> – watch this if needed further assist)

1. **Download GnuWin32 Make:**

<https://gnuwin32.sourceforge.net/packages.html>



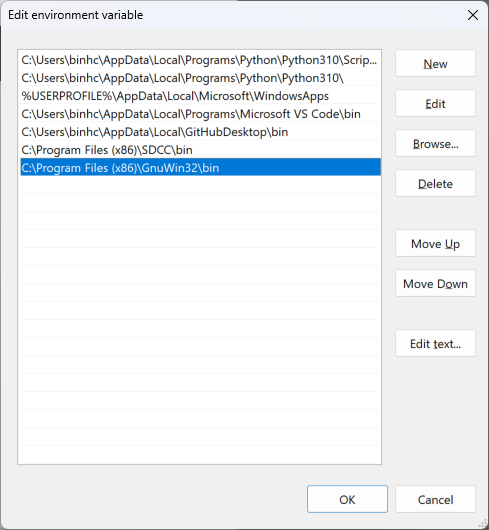
1. **Download coreutils:**



1. **Install the downloaded file, leave all by default:**

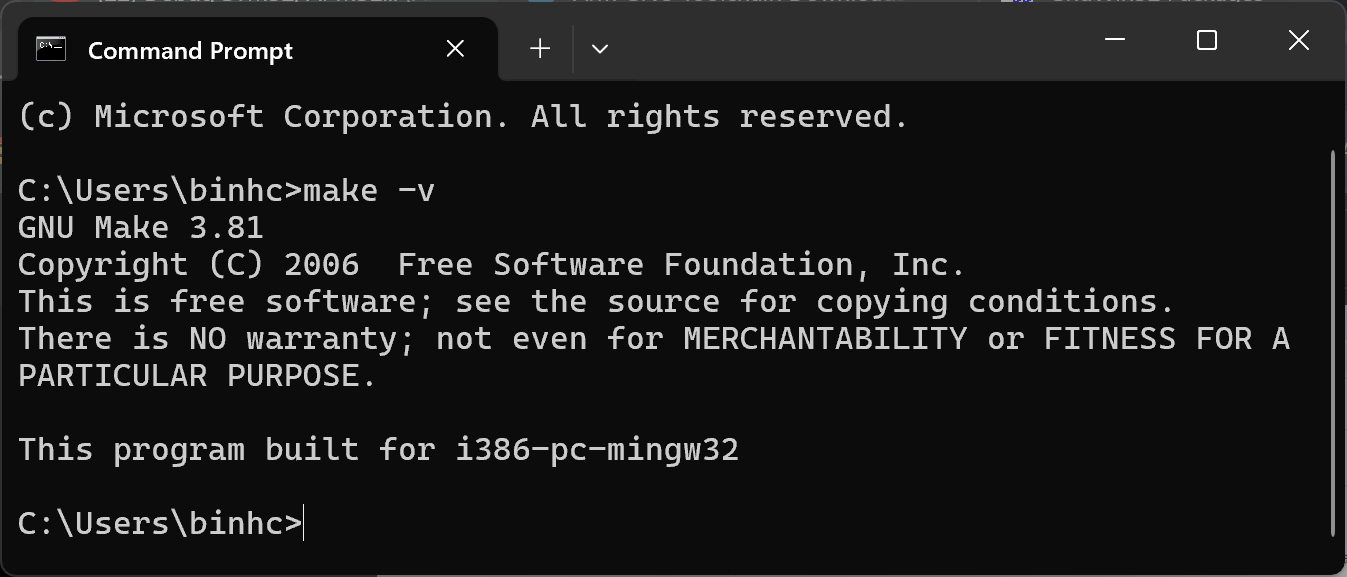


1. **Add Make to Windows Environment Variable:**



1. **Check if installation is successful: access cmd.exe, type:**

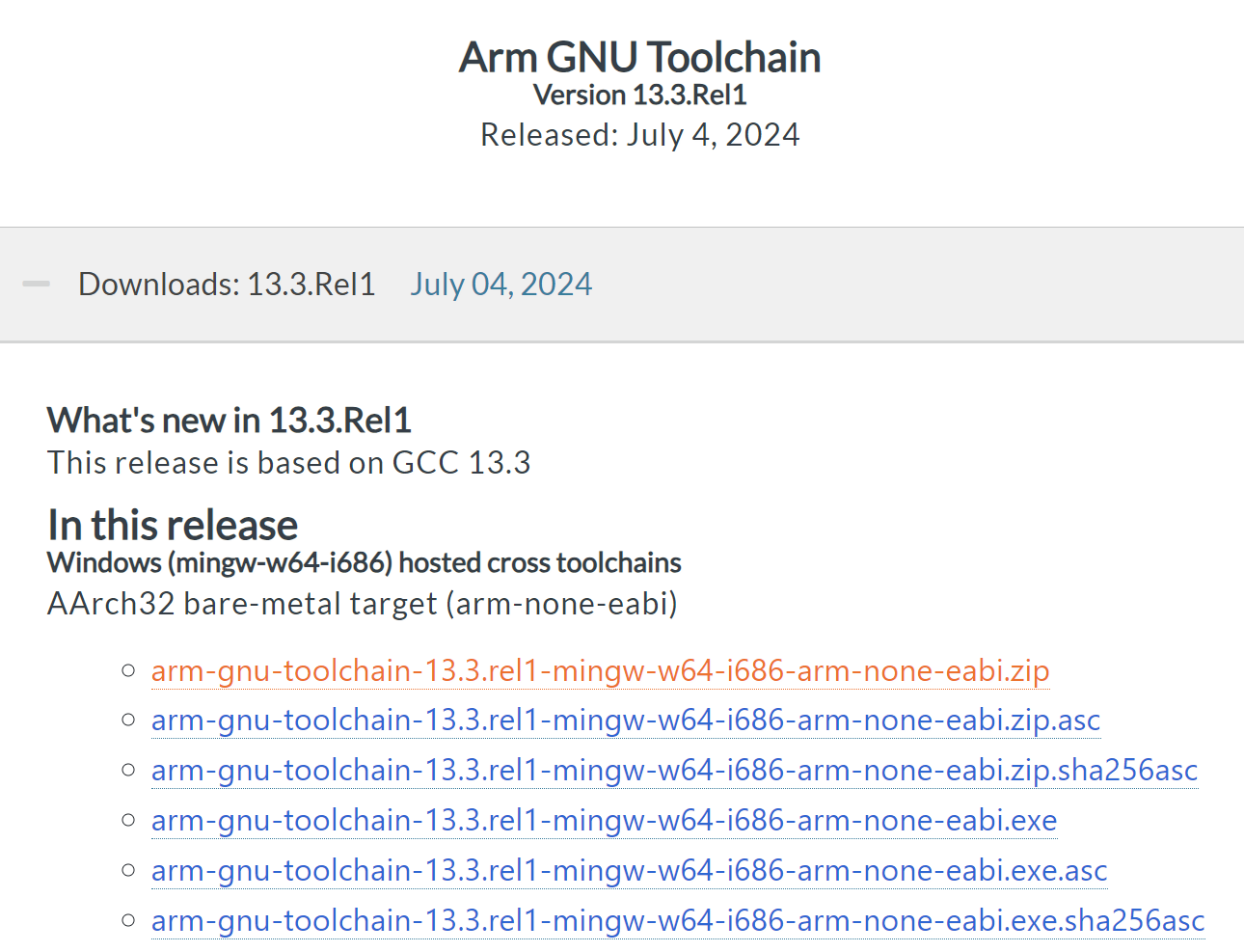
make -v



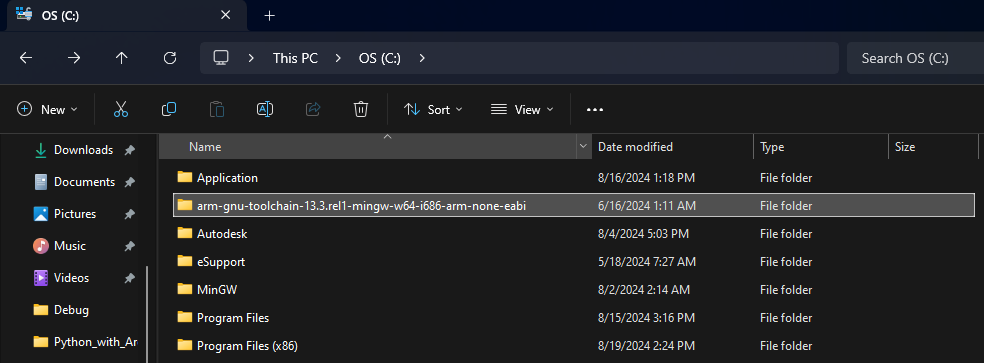
Result indicates successful installation

**B.2. Download ARM GCC**

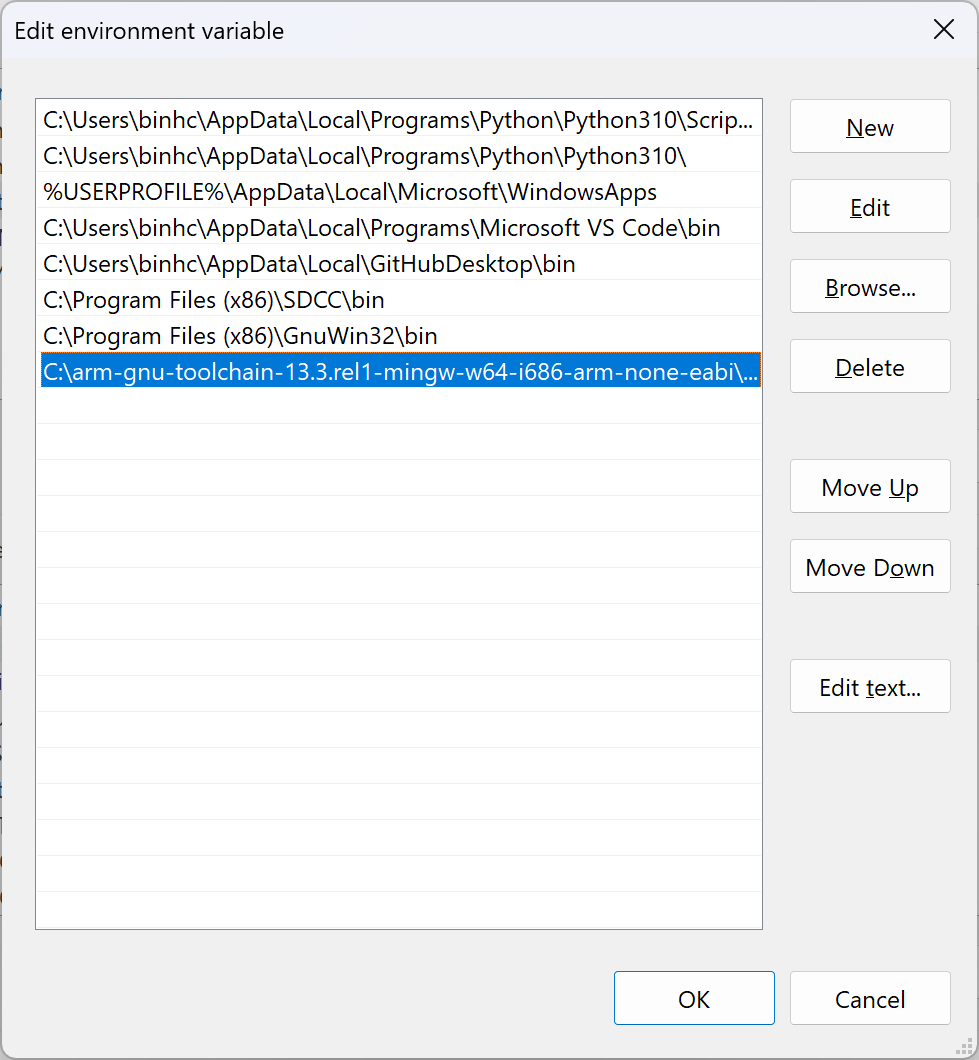
1. **Download GNU Arm toolchain**: <https://developer.arm.com/downloads/-/gnu-rm>



1. **Extract the file:**



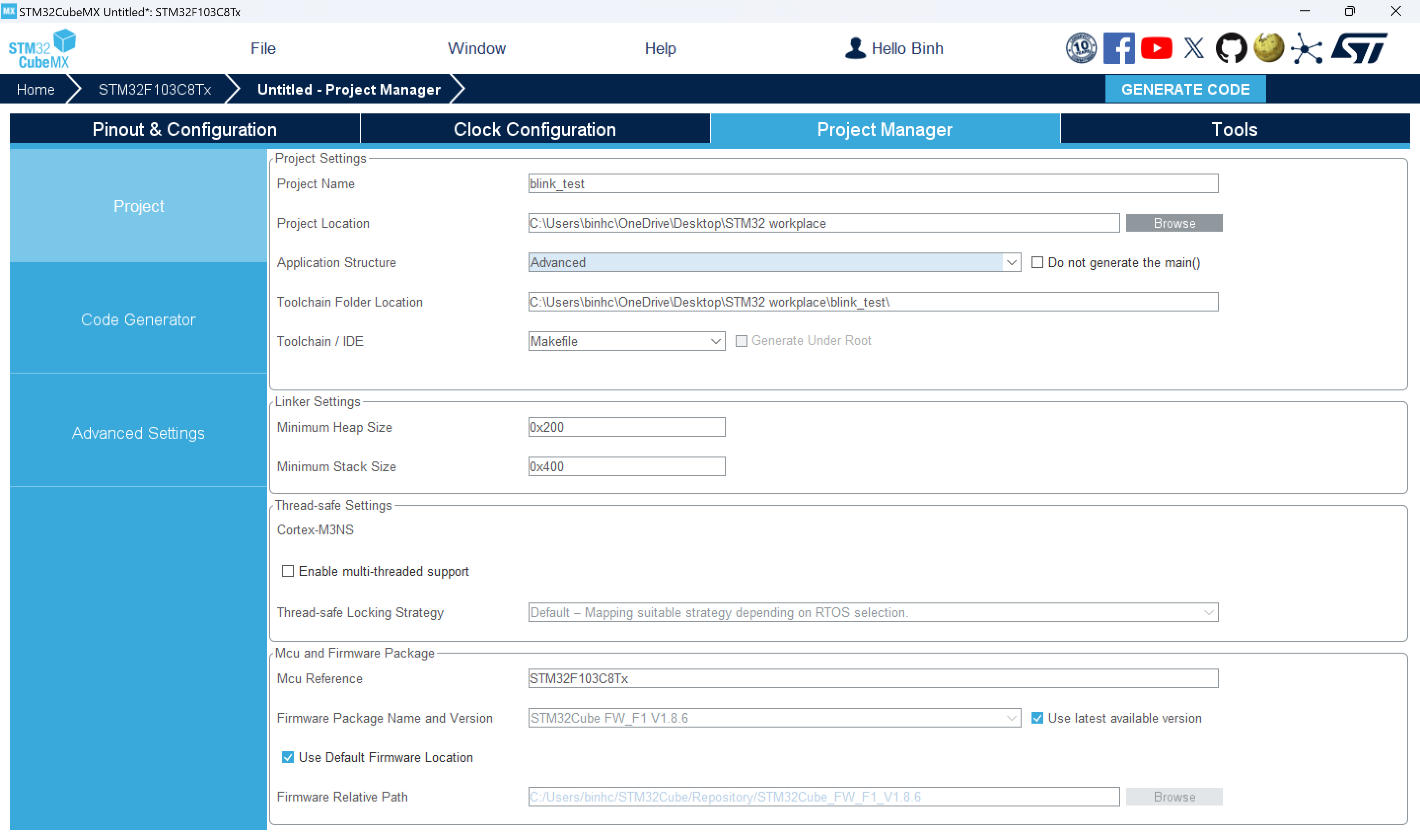
1. **Add ARM GCC to Environment Variable:**



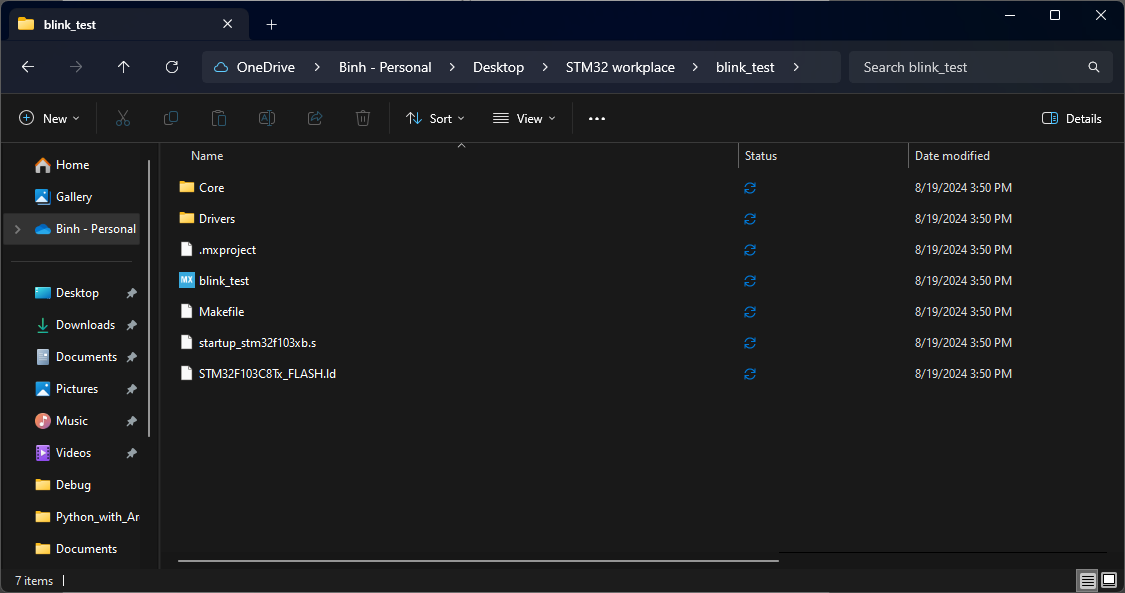
**B.3. Build project with Make**

Based on this vid: https://www.youtube.com/watch?v=FkqQpBqkSns&t=1081s

1. **Make a STM32 project with CubeMX: (MUST turn on serial wire debug in SYS; otherwise the MCU is bricked)**

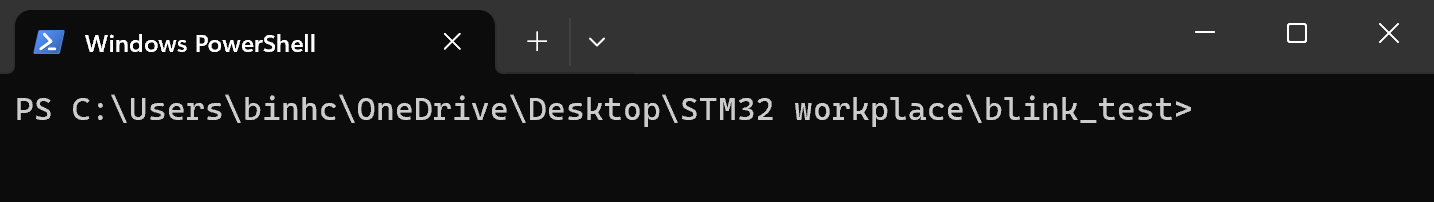


Check project installation in Windows explore:



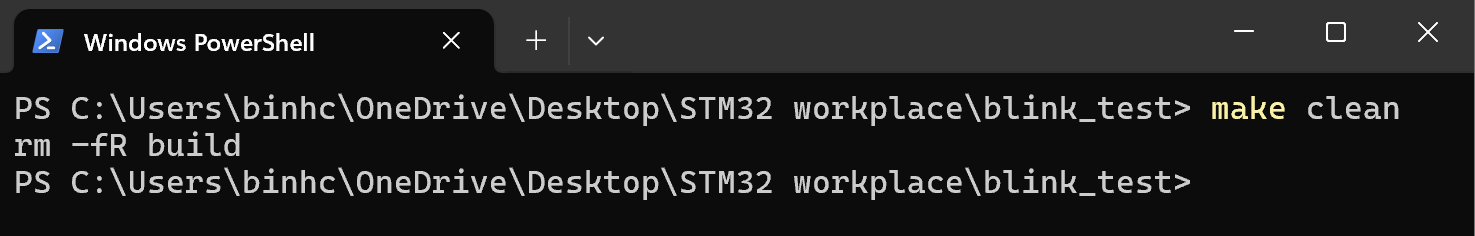
1. **Build project using windows cmd (or PowerShell):**

Change directory to project



Input command:

make clean

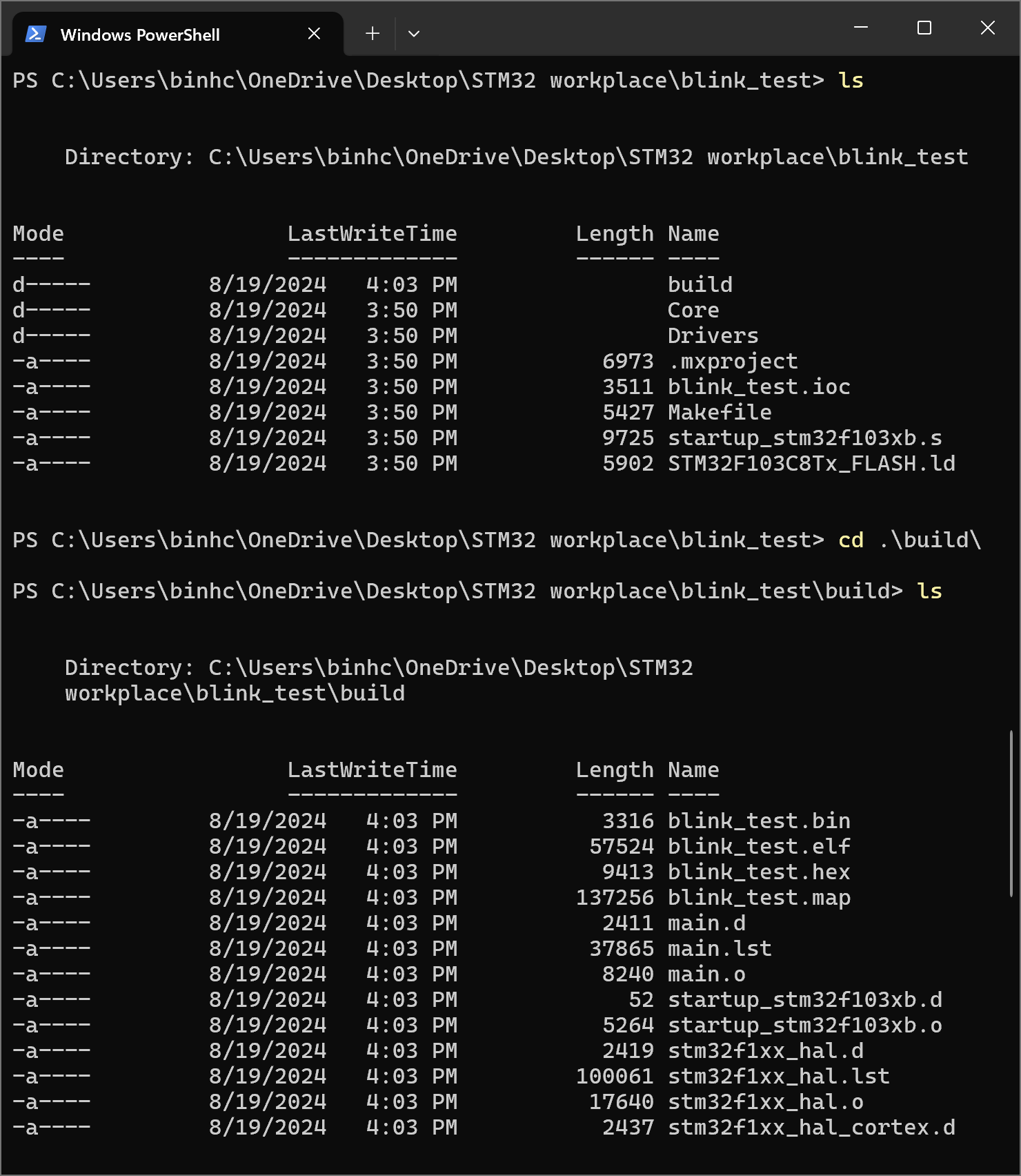


Input command: (for -j12: 12 or any is the number of CPU threads you want to use to build the project)

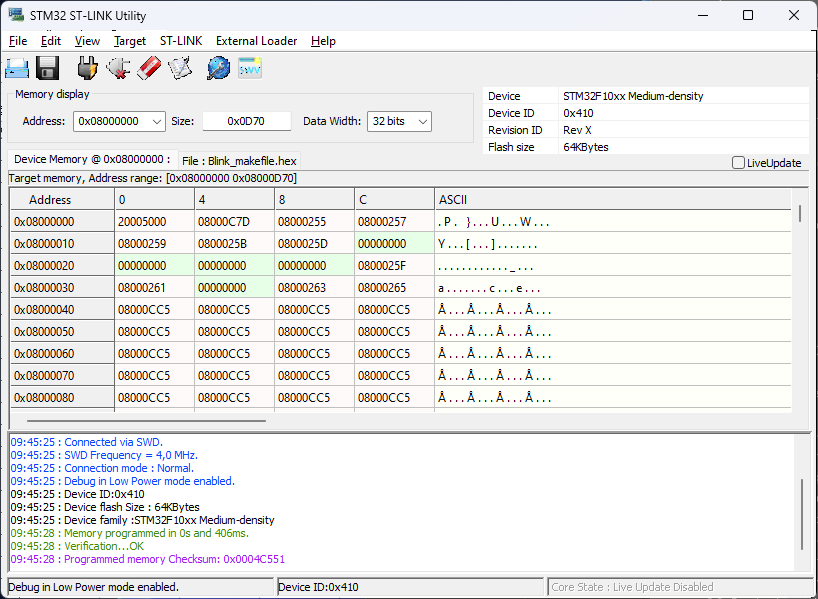
make -j12



1. **Confirm that the installation is completed: build folder is created with bin, hex, elf file inside**



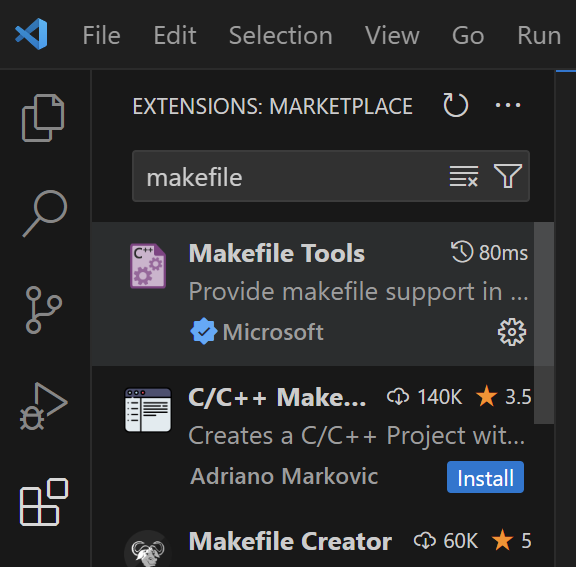
1. **Flash built hex/bin file to MCU, using stm32 st-link utility:**



**B.4. Configurate VScode for STM32 project**

Based on vid: https://www.youtube.com/watch?v=jcy5TpbXfAY&t=3s

1. **Open stm32 project in vscode**
2. **Install extensions C/C++ and Makefile tools:**



1. **Overwrite Settings.json with these lines:**

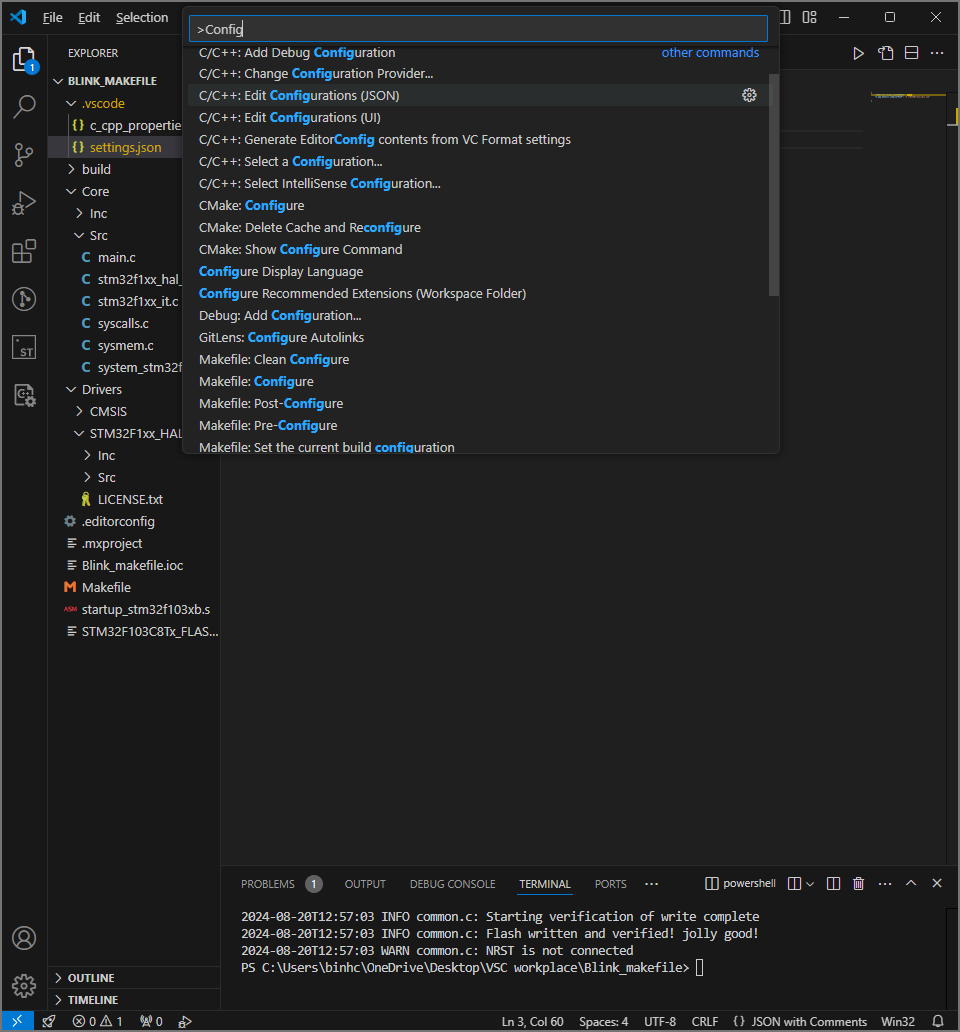
{

    "stm32-for-vscode.openOCDPath": false,

    "C\_Cpp.default.compilerPath": "C:\\MinGW\\bin\\gcc.exe"

}

1. **Overwrite c\_cpp\_properties.json with these lines:**



{

    "configurations": [

        {

            "name": "Win32",

            "includePath": [

                "${workspaceFolder}/\*\*"

            ],

            "defines": [

                "\_DEBUG",

                "UNICODE",

                "\_UNICODE",

                "USE\_HAL\_DRIVER",

                "STM32F103xB"

            ]

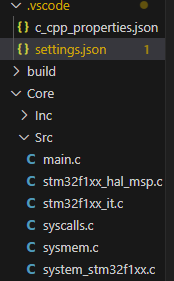
        }

    ],

    "version": 4

}

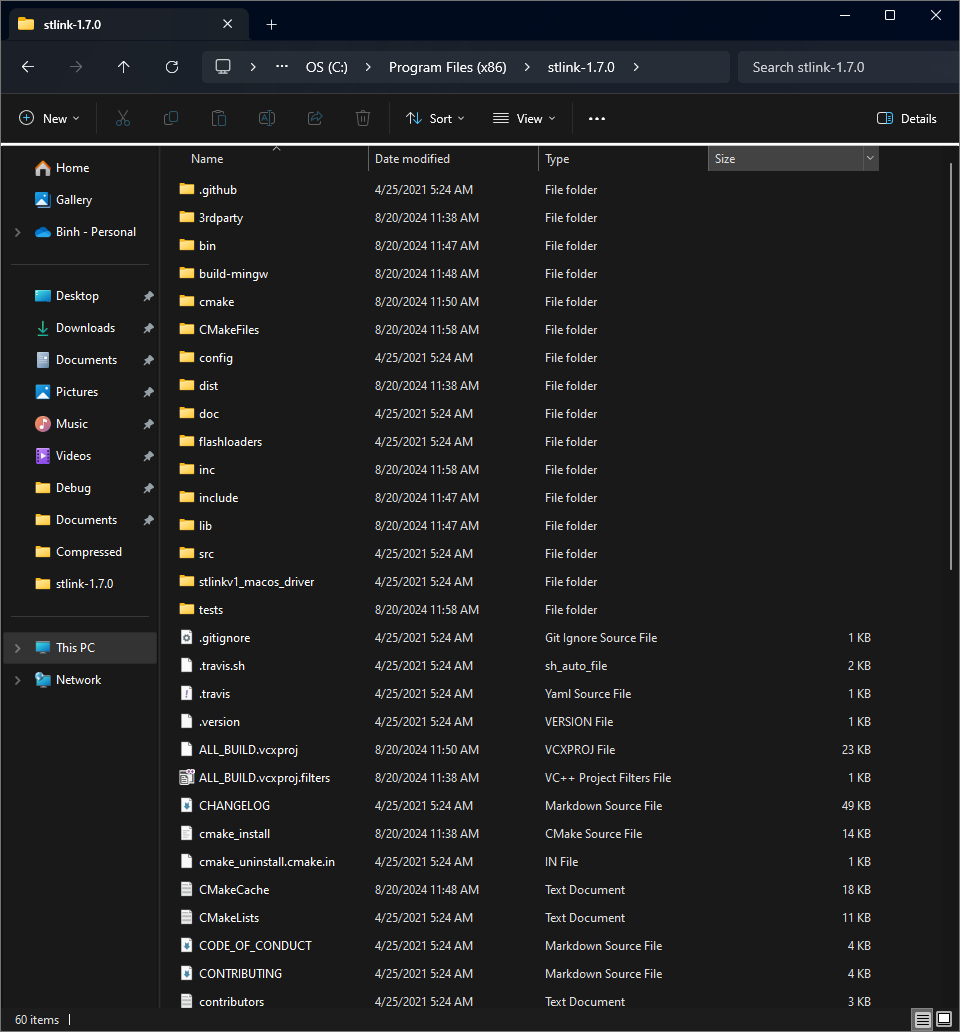
1. **If setup is successful, no critical errors will be shown in the main.c:**



**B.5. Flashing STM32 with VScode and STLINK**

Based on vid: <https://www.youtube.com/watch?v=1cleO3mHjWw&t=188s>

1. **Download stlink from git (get version 1.7.0 or 1.8.0, don’t get testing version):** <https://github.com/stlink-org/stlink/releases/tag/v1.7.0>
2. **Extract it into Program Files (x86):**

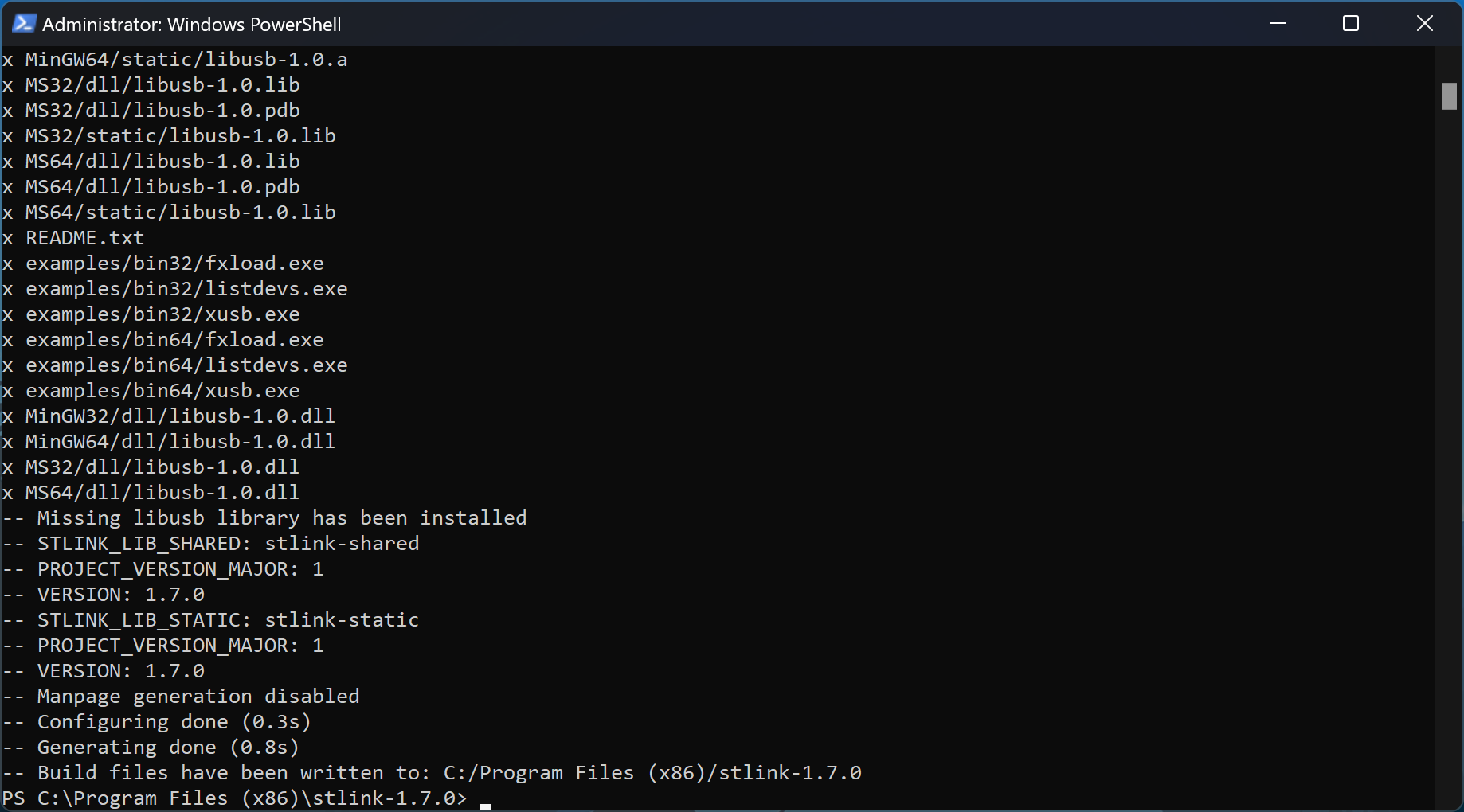


1. **Go into powershell (cmd) as admin and change directory to the extract folder**



1. **Run command:**

cmake .

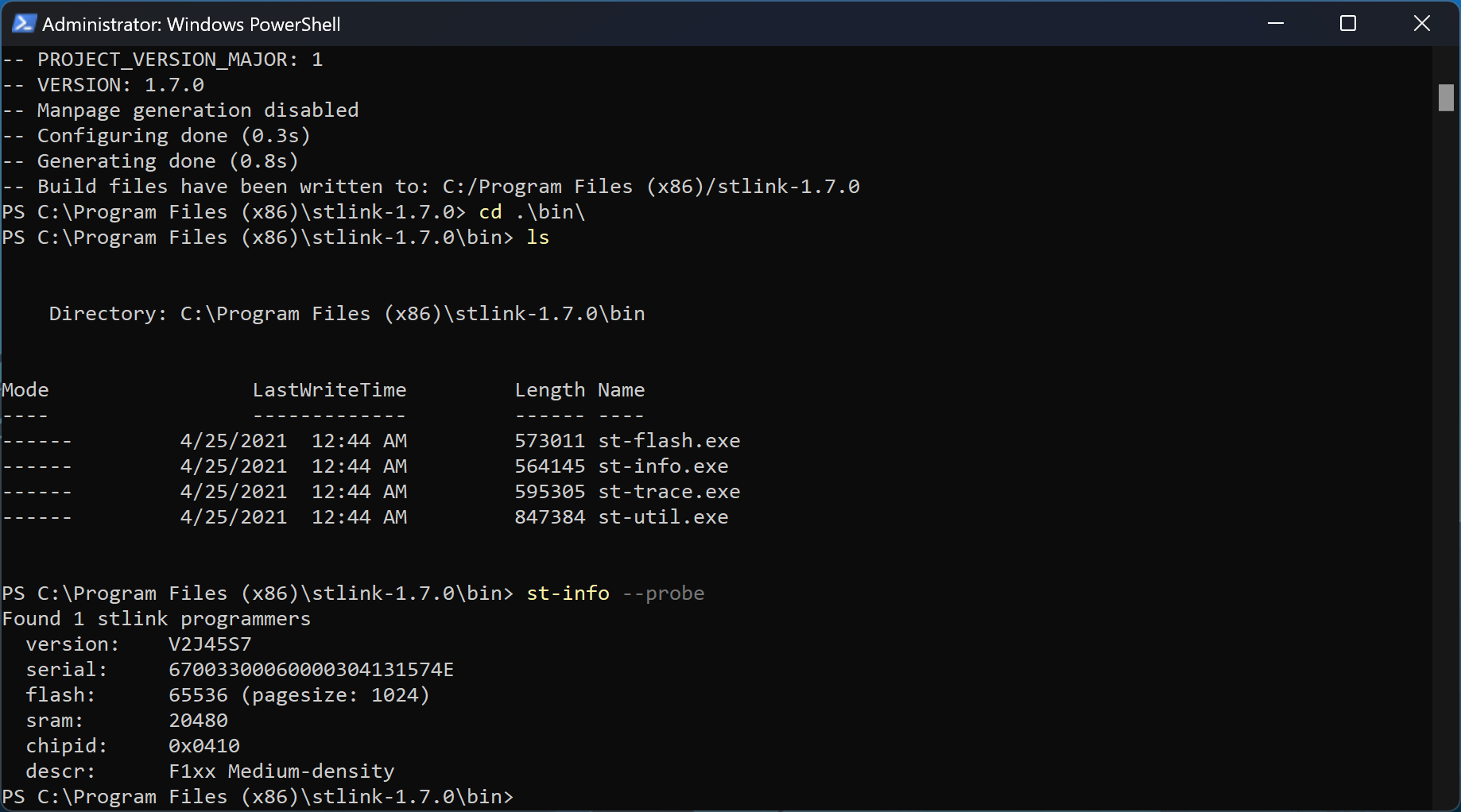


1. **Check if stlink is installed successfully:**

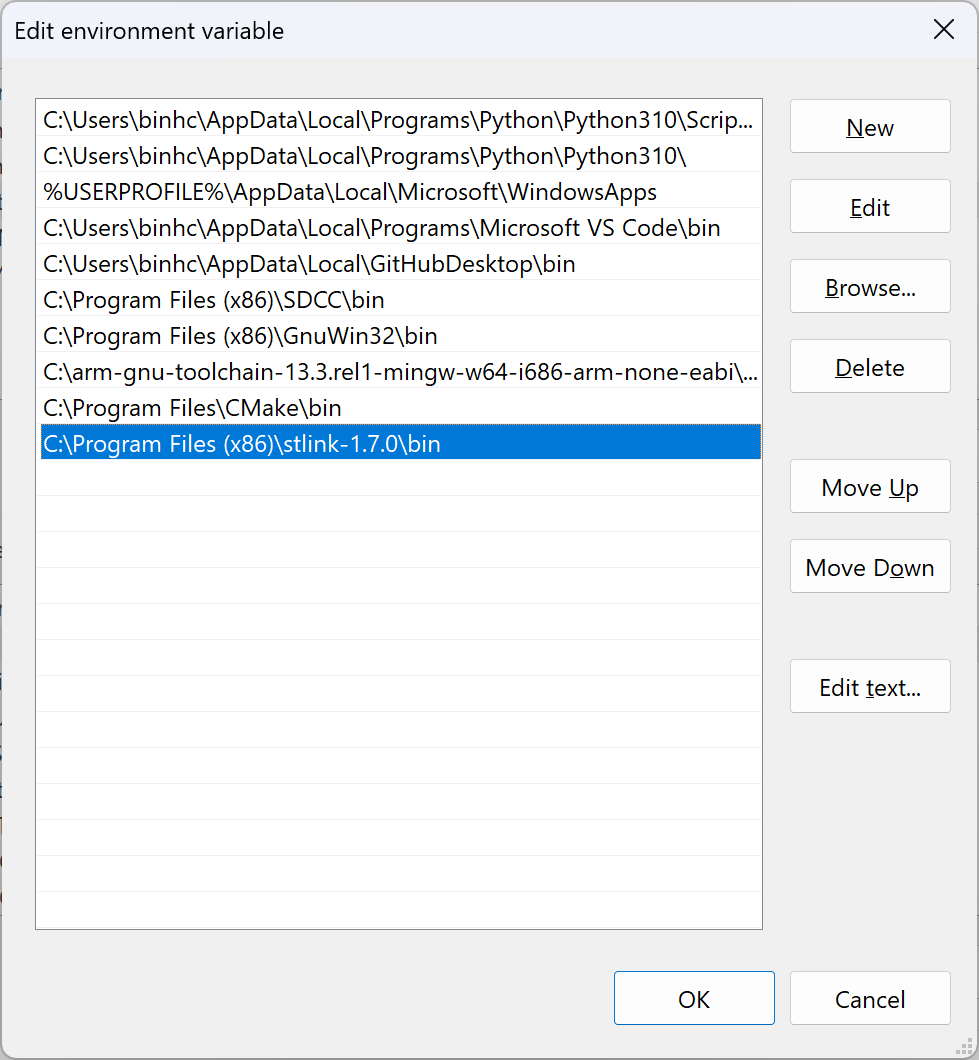
Run command:

cd ./bin/

st-info --probe



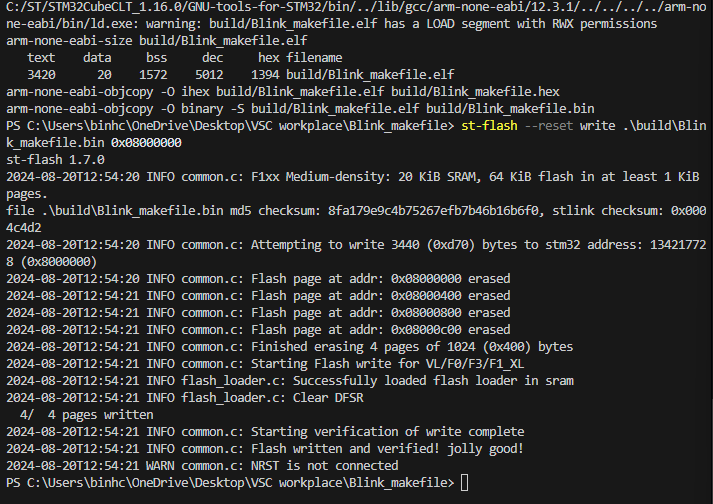
1. **Add bin to Environment Variable:**



1. **Flashing code to STM32:**

Run command:

st-flash --reset write .\build\Blink\_makefile.bin 0x08000000



**B.6. Debugging using st-util**

1. **Config settings.json:**

{

    //"stm32-for-vscode.openOCDPath": true,

    "C\_Cpp.default.compilerPath": "C:\\MinGW\\bin\\gcc.exe",

    "cortex-debug.gdbPath": "C:\\arm-gnu-toolchain-13.3.rel1-mingw-w64-i686-arm-none-eabi\\bin\\arm-none-eabi-gdb",

    "cortex-debug.stutilPath": "C:\\Program Files (x86)\\stlink-1.7.0\\bin\\st-util",

    "stm32-for-vscode.openOCDPath": false,

}

1. **Config launch.json:**

{

    "version": "0.2.0",

    "projectName": "Blink\_makefile",

    "configurations": [

        {

            "name": "STlink launch",

            "cwd": "${workspaceRoot}",

            "executable": "${workspaceRoot}/build/Blink\_makefile.elf",

            "request": "launch",

            "type": "cortex-debug",

            "servertype": "stutil",

            "device": "STM32F103C8",

            "interface": "swd",

            //"runToMain": true, // else it starts at reset handler - not interested

            "preLaunchTask": "Build all", // configured in tasks.json

            // "preLaunchCommands": ["Build all"], // you can execute command instead of task

            "svdFile": "", // Include svd to watch device peripherals

            "swoConfig": {} // currently (v1.7.0) not supported

        },

        {

            "name": "STlink attach",

            "cwd": "${workspaceRoot}",

            "executable": "${workspaceRoot}/build/Blink\_makefile.elf",

            "request": "attach",

            "type": "cortex-debug",

            "servertype": "stutil",

            "device": "STM32F103C8",

            "interface": "swd",

            //"runToMain": true, // else it starts at reset handler - not interested

            "preLaunchTask": "Build all", // configured in tasks.json

            // "preLaunchCommands": ["Build all"], // you can execute command instead of task

            "svdFile": "", // Include svd to watch device peripherals

            "swoConfig": {} // currently (v1.7.0) not supported

        },

    ]

}

1. **Config tasks.json:**

{

    // See https://go.microsoft.com/fwlink/?LinkId=733558

    // for the documentation about the tasks.json format

    "version": "2.0.0",

    "tasks": [

        {

            "label": "Build all",

            "group": "build",

            "type": "shell",

            "command": "make",

            "args": ["all", "-j4"]

        },

        {

            "label": "Build clean",

            "group": "build",

            "type": "shell",

            "command": "make",

            "args": ["clean"]

        },

        {

            "label": "JFlash",

            "group": "build",

            "type": "shell",

            "command": "make",

            "args": ["-j4","j-flash"]

        },

        {

            "label": "STflash",

            "group": "build",

            "type": "shell",

            "command": "make",

            "args": ["-j4","st-flash"]

        },

        {

            "label": "UARTFlash",

            "group": "build",

            "type": "shell",

            "command": "make",

            // Replace PORT= with your UART port

            // Linux has ttyUSBx or ttyACMx, windows has COMx

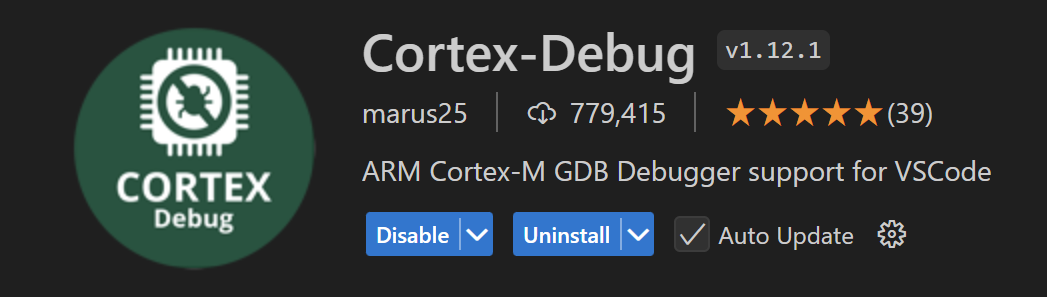
            "args": ["-j4","u-flash", "PORT=/dev/ttyUSB1"]

        }

    ]

}

1. **Install Cortex-debug:**

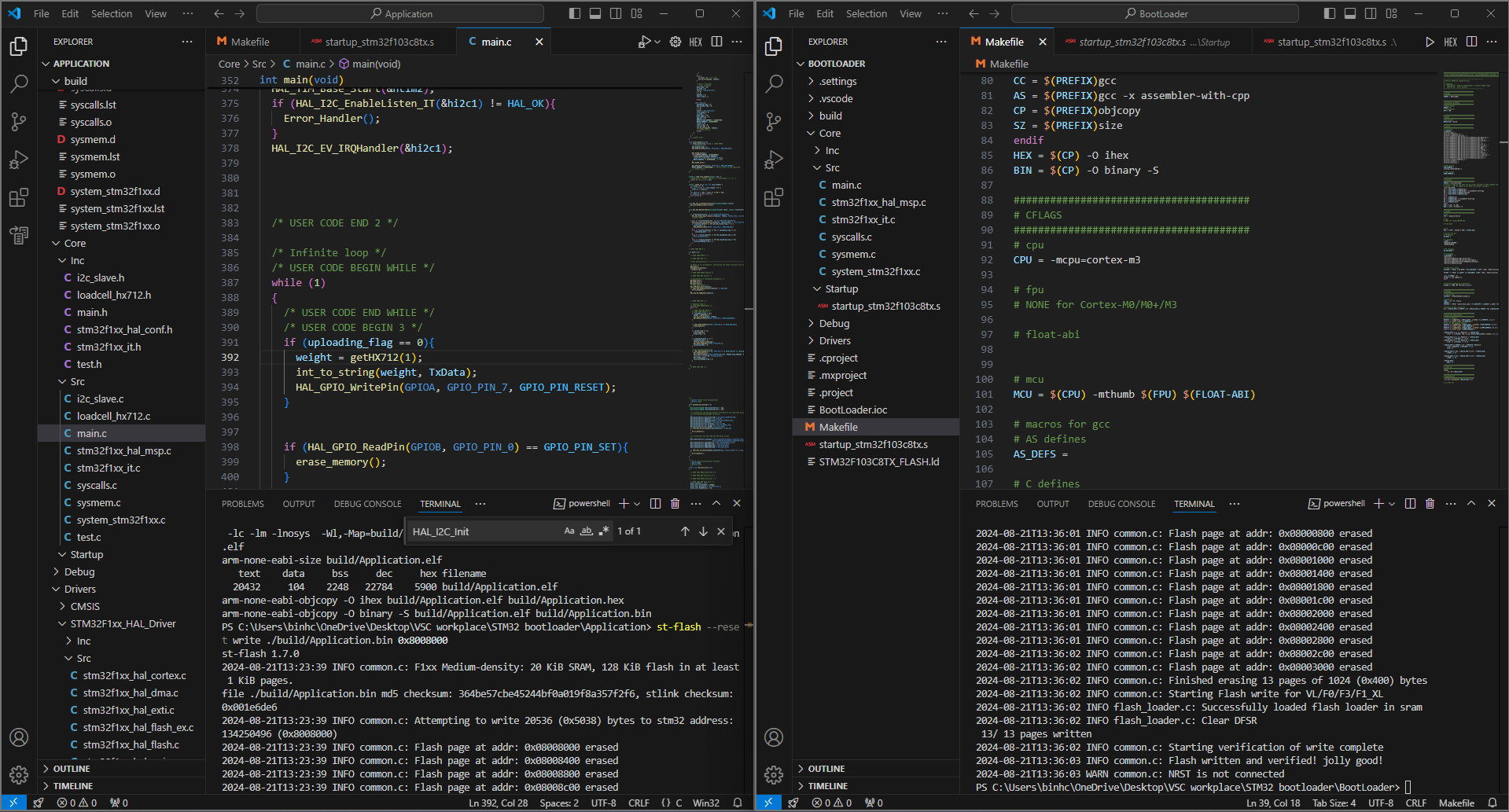


1. **Run:**



Result:

Apply Makefile and stlink to Bootloader project:



# APPENDIX C: OCCUPIED FLASH SIZE

**C.1. Arduino Uno**

Sketch uses 6634 bytes (21%) of program storage space. Maximum is 30720 bytes.

Global variables use 626 bytes (30%) of dynamic memory, leaving 1422 bytes for local variables. Maximum is 2048 bytes.

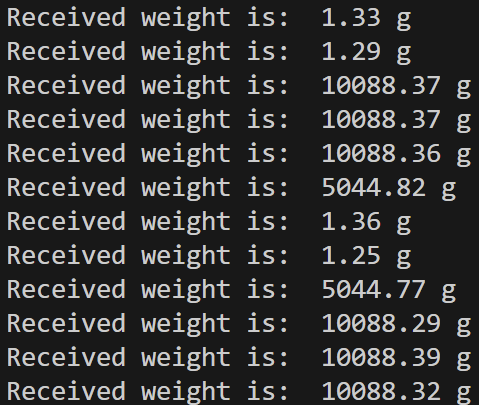
**C.2. STM32F103**

**Application image, with build optimization -O1:**



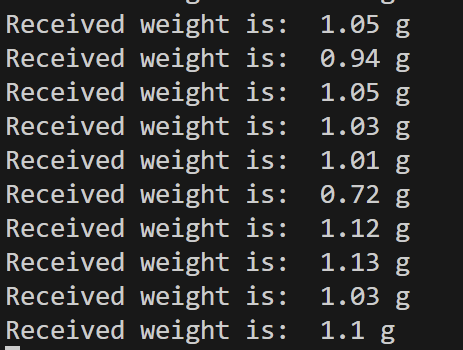
Note: -O1 affects the weight scale application:

For 10Hz sampling time, the application does not work as intended:



In which, values that are larger than 2500g are error (when check with the values read directly from ADC, two most significant digits are missing)

For 40Hz sampling time, the application works as intended:



To neglect the problem, optimization level must be set to -O0 or those with the same level, which brings the occupied size to 18kB.



**Bootloader image, with build optimization -O1:**



|  |  |  |  |
| --- | --- | --- | --- |
| Version | Year of release | Max transmission range | Max range |
| Bluetooth 1.0 | 1999 | 732.2 kbit/s | 10 m |
| Bluetooth 1.1 | 2001 | 732.2 kbit/s | 10 m |
| Bluetooth 1.2 | 2003 | 1 Mbps | 10 m |
| Bluetooth 2.0 | 2004 | 2.1 Mbps | 30 m |
| Bluetooth 2.1 | 2007 | 2.1 Mbps | 30 m |
| Bluetooth 3.0 | 2009 | 24 Mbps | 30 m |
| Bluetooth 4.0 | 2009 | 1 Mbps (LE)  3 Mbps (EDR) | 60 m |
| Bluetooth 4.1 | 2013 | 1 Mbps (LE)  3 Mbps (EDR) | 60 m |
| Bluetooth 4.2 | 2014 | 1 Mbps (LE)  3 Mbps (EDR) | 60 m |
| Bluetooth 5.0 | 2016 | 2 Mbps (LE)  50 Mbps (EDR) | 240 m |
| Bluetooth 5.1 | 2019 | 2 Mbps (LE)  50 Mbps (EDR) | 240 m |
| Bluetooth 5.2 | 2020 | 2 Mbps (LE)  50 Mbps (EDR) | 240 m |
| Bluetooth 5.3 | 2021 | 2 Mbps (LE)  50 Mbps (EDR) | 240 m |
| Bluetooth 5.4 | 2023 | 2 Mbps (LE)  50 Mbps (EDR) | 240 m |

# REFERENCES

[1] https://www.grc.nasa.gov/www/k-12/airplane/tunwheat.html

[2] https://developer.arm.com/documentation/ka002218/latest/

[3] I2C Bus Pullup Resistor Calculation, Texas Instruments Incorporated, February 2015, [C:\Users\binhc\OneDrive\Desktop\slva689.pdf](file:///C:\Users\binhc\OneDrive\Desktop\slva689.pdf)

[4] https://www.flintec.com/weight-sensors/load-cells/what-is-a-load-cell