DCT Internship: Weekly Report

Contents

1.1 MON - JAN 19 2026

TODO: ELABORATE ON BELOW

- 8085 Microprocessor.
- LED Interfacing with 8085.
- Programmable Peripheral Interface.
- Interrupts of 8085.
- Direct Memory Access.
- Operation of Stack.

1.2 TUE - JAN 20 2026

TODO: ELABORATE ON BELOW

- Address Mapping.
- Introduction to STM32H755ZI.

1.3 WED - JAN 21 2026

TODO: ELABORATE ON BELOW

- Introduction to Linux.
- Command Line.
- Exercise: Send a byte to a com port from Linux using C.

1.4 THU - JAN 22 2026

TODO: ELABORATE ON BELOW

- Dynamic and Static Libraries.
- Steps in creating them.
- Compiler, linker, and loader.
- Linux boot process.
- Linux file system.
- Symbol table.

1.5 FRI - JAN 23 2026

TODO: ELABORATE ON BELOW

- Resistors.
- Caps.
- Diode.
- Inductor.
- Transistor.
- Current limiting.
- C rating of battery.
- R, C, L, standards.
- V_{OH}, V_{OL}, V_{IH}, and V_{IL}.
- Power sequencing.

DCT Internship: Weekly Report

Contents

2.1 MON - JAN 26 2026

TODO: NEED TO UPDATE

- Simulation of RC circuit.
- Simulation of RL circuit.
- Simulation of RLC circuit.
- Pi filter.
- High side and low side switching.
- Power supplies.
- Voltage regulators.
- Buck, boost, buck-boost.
- SMBus.
- UART.
- I2C.

2.2 TUE - JAN 27 2026

TODO: NEED TO UPDATE

- Exam.
- Rough intro to STM32 nucleo board.

2.3 WED - JAN 28 2026

TODO: NEED TO UPDATE

- Familiarization of STM32 Board and IDE.
- LED blinking, interrupt, uart.

2.4 THU - JAN 29 2026

2.4.1 Architecture of Nucleo-H755ZI-Q

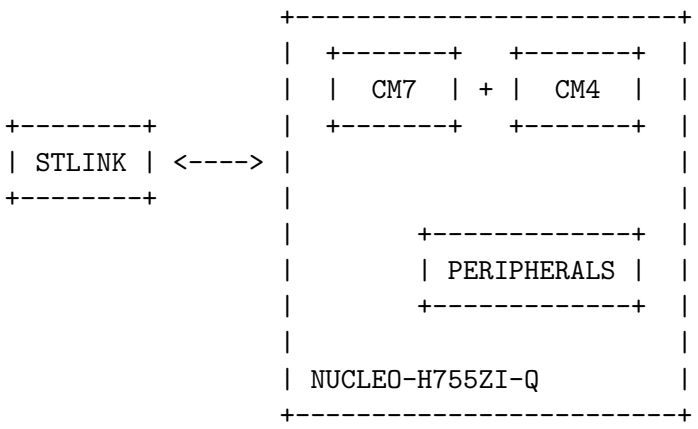

Figure 2.1: Rough architecture of Nucleo-H755ZI-Q.

Nucleo-H755ZI-Q has a asymmetric dual core architecture, having two cores Arm Cortex M7 and Arm Cortex M4.

Note Asymmetric Multi-processing: One Master and One or More Slaves. Master does all the configuration and task scheduling.

Note Symmetric Multi-processing: All have equal access to memory and resources.

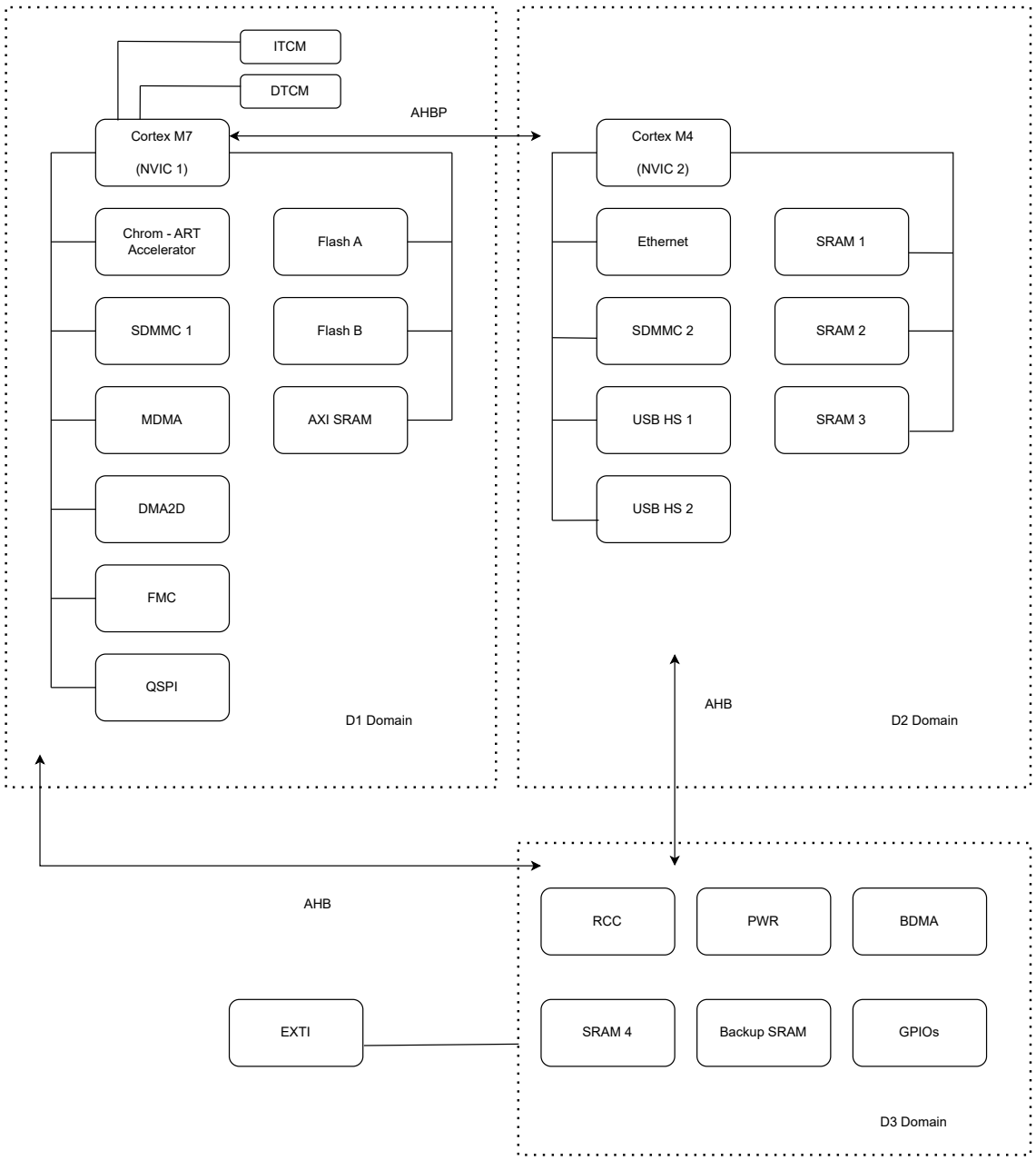

Figure 2.2: STM32H755ZI Architecture.

Sub-systems

The entire MCU is divided into 3 major power domains.

- D1: Where the M7 lies.
- D2: Where the M4 lies.
- D3: Where the RCC and PWR lies.
- BACKUP Domain: Where the EXTI lies.

Important From figure 2.2, we can see that the `EXTI`[a] is outside of the 3 domains. This enables to power down all 3 of these domains at the same time. Then `EXTI`'s domain can be kept active, and `EXTI` will be responsible for powering up the required domains to serve interrupts.

[a] External Interrupt Controller

D1 sub-system

D1 domain is intended for high performance applications. Has a high bandwidth AXI bus matrix. The Cortex M7 core is located here running at up to 480MHz with 16-Kbyte I-cache and 16-Kbyte D-cache.

Note **Internal Memory Resources:**

- AXI SRAM (512-Kbyte) mapped at address `0x2400 0000`.
- ITCM[a] accessible through 64-bit ITCM bus.
- DTCM[b] accessible through 2x32-bit DTCM bus.
- Flash memory: Two 1MB independent flash banks.

[a] Instruction Tightly Coupled Memory.
[b] Data Tightly Coupled Memory.

D2 sub-system

The D2 domain is intended for generic peripheral usage, communication and data gathering that can be processed later in D1 domain. It contains Cortex M4 core running at up to 240MHz.

Note **Internal Memory Resources:**

- AHB SRAM1 (128 KByte) mapped at address `0x3000 0000` and `0x1000 0000`.
- AHB SRAM2 (128 KByte) mapped at address `0x3002 0000` and `0x1002 0000`.
- AHB SRAM3 (32 KByte) mapped at address `0x3004 0000` and `0x1004 0000`.

D3 sub-system

The D3 domain provides system management and low power operating modes feature. It contains peripherals like PWR, RCC, ADCs, GPIOs. It has a AHB bus matrix, and a BDMA[1] It also contains the communication peripherals like I2C, SPI, LPUART.

[1] Basic DMA.

Note **Internal Memory Resources:**

- AHB SRAM4 (128 KByte) mapped at address `0x3000 0000`.

Peripherals

From figure 2.2 we can see that different peripherals are in different power domains. Each domain has it's own bus matrix. There are inter domain buses for communication across domains. In D1, 64-bit AXI[2] bus is used. D2 and D3 domains use 32-bit AHB[3] buses.

Note Most of the communication peripherals are located in the D2 domain. For example, USB, Ethernet, FDCAN, UART, SPI, and SD/MMC are located here. Most of the memories dedicated to I/O processing are also located here.

2.5 FRI - JAN 30 2026

2.5.1 Boot Process of STM32H755ZI

There are three power domains. CM7 and CM4 lies in the D1 and D2 domains respectively. Circuits that are responsible for power up and clock generation lies in the D3 domain. RCC[4] is responsible for processing reset and generating clock to the CM7 and CM4. In essence, RCC boots up the cores.

BOOT0 Pin and BOOT_ADDx Registers

Boot0 pin enables the user to boot from a factory configured bootloader[5] if it is set to 1. Otherwise BOOT_ADDx registers chooses the boot location of each of the cores.

Note There are two system memory banks. Each one is of size 128 K.

- Bank 1: Located at `0x1FF0 0000`.
- Bank 2: Located at `0x1FF4 0000`.

[2] Advanced eXtensible Interface.
[3] Advanced High Performance
[4] Reset and Clock Controller
[5] Non user programmable.

DCT Internship: Weekly Report

Contents

3.1 MON - FEB 02 2026

TODO: NEED TO UPDATE

- Interrupts.
- NVIC.
- EXIT.

3.2 TUE - FEB 03 2026

TODO: NEED TO UPDATE

- Ethernet.
- UART.

3.3 WED - FEB 04 2026

3.3.1 UART Interfacing Exercise using STM32

Exercise Question: Configure three UART interfaces on the development board. One UART is used as the ST-LINK virtual COM port and acts as the console interface. The application prompts the user through the console to enter a numeric value between 100 and 10000 and to select a UART interface (UART2 or UART3).

Based on the selected UART:

- If UART2 is selected, the console displays the data received from UART3 multiplied by 2.
- If UART3 is selected, the console displays the data received from UART2 multiplied by 2.

UART2 and UART3 are physically interconnected using jumper wires between their TX and RX pins to enable bidirectional communication.

Solution Overview

The USART3 is connected to the ST-LINK's UART. We can use USART1 and USART2 as the other two UARTs. All we need to do is to initialize these UARTs[1]. Then we need to set up the ISR[2] to process the received data. While the main task will handle the user input.

[1] Spoiler Alert: This can be done through STM32CubeMX, and all we need to do is to start using them.
[2] Interrupt Service Routine.

Connection Block Diagram

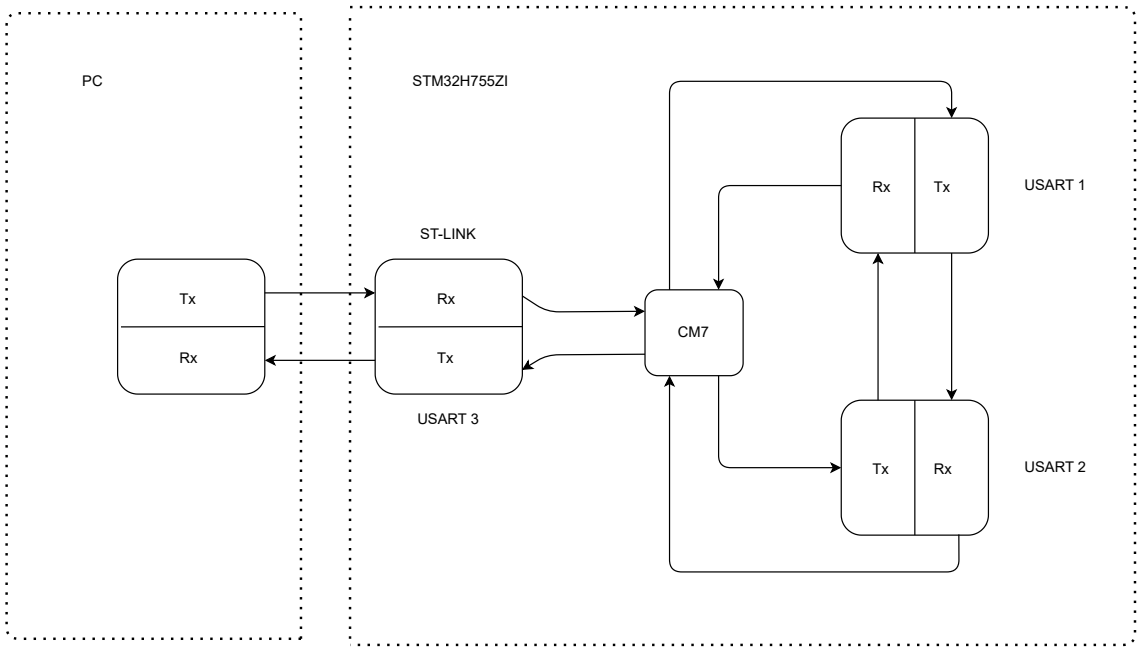

Figure 3.1: Connection block diagram.

Main Task Flow

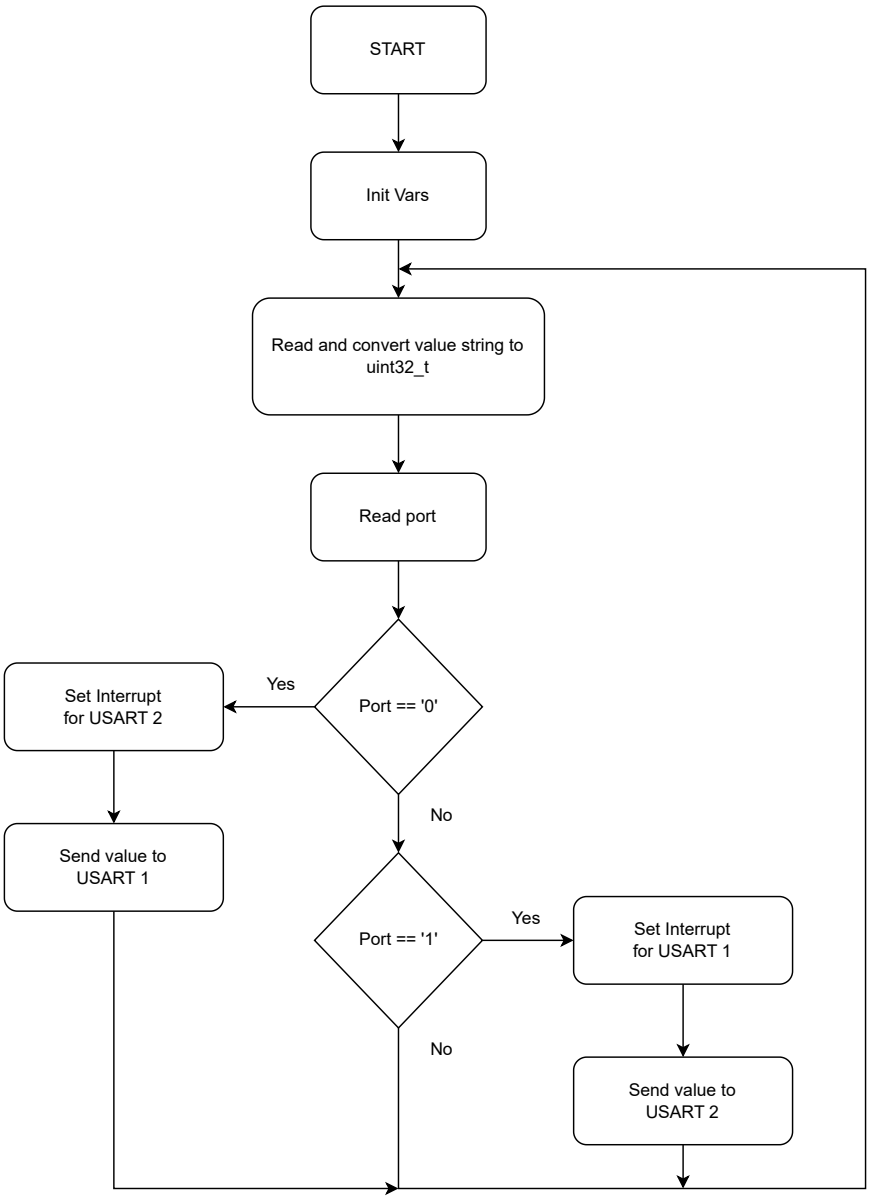

Figure 3.2: Flow of main task / thread.

ISR Task Flow

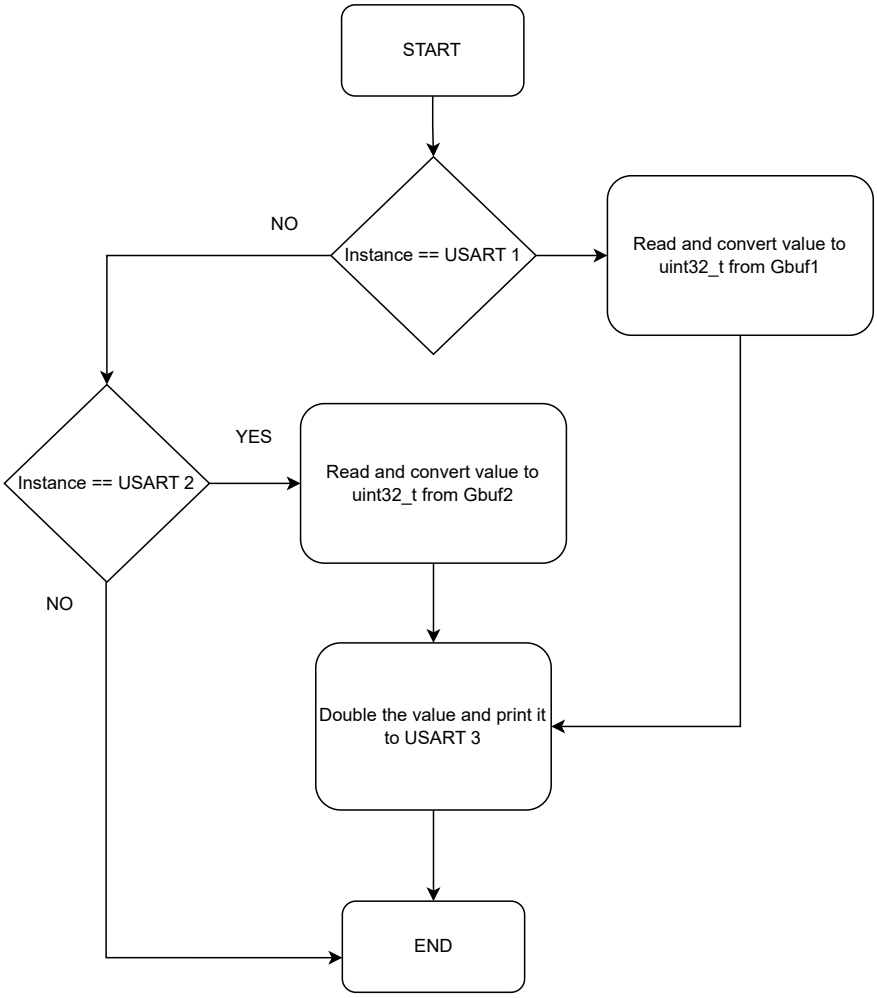

Figure 3.3: Flow of ISR task / thread.

3.4 THU - FEB 05 2026

3.4.1 UART Interfacing Exercise using STM32 (Continued)

C Implementation

The execution flow can be divided into two, one that interacts with the user and the other that deals with the interrupt.

main()

```
#include <string.h>

#define VALUE_READ_BUF_SIZE 128
```

```c
char Gbuf1[4];
char Gbuf2[4];

void print_doubled(int val);
void send_int(UART_HandleTypeDef *dst_uart, uint32_t val);
uint32_t recv_int(char *buf);
void read_value(char *dst, uint16_t size);

int main(void) {
    /*
     * After initialization code
     */

    char ask_value_str[] = "Enter the value: ";
    char ask_port_str[] = "Enter the port (0 == usart1 or 1 == usart2): ";

    int value = 0;

    char value_buf[VALUE_READ_BUF_SIZE];
    char port_buf[2];

    while (1) {
        HAL_UART_Transmit(&huart3, (const uint8_t *) ask_value_str,
        ↪  sizeof(ask_value_str), HAL_MAX_DELAY);
        read_value(value_buf, VALUE_READ_BUF_SIZE);
        value = atoi(value_buf);

        HAL_UART_Transmit(&huart3, (const uint8_t *) ask_port_str,
        ↪  sizeof(ask_port_str), HAL_MAX_DELAY);
        read_value(port_buf, 2);

        if (*port_buf == '0') {
            HAL_UART_Receive_IT(&huart2, (uint8_t *) Gbuf2, sizeof(Gbuf2));
            send_int(&huart1, value);
        } else if (*port_buf == '1') {
            HAL_UART_Receive_IT(&huart1, (uint8_t *) Gbuf1, sizeof(Gbuf1));
            send_int(&huart2, value);
        }
    }
}
```

As we can see, the main function constantly reads the value and port to send. Then depending on the port, sets the interrupt and sends the data.

read_value()

```c
void read_value(char *dst, uint16_t size) {

    char tmp;
```

```
    char *ptr = dst;

    while (1) {
        HAL_UART_Receive(&huart3, (uint8_t *) &tmp, sizeof(tmp),
        ↪  HAL_MAX_DELAY);

        if (tmp == '\r') {
            HAL_UART_Transmit(&huart3, (const uint8_t *) "\n\r", 2,
            ↪  HAL_MAX_DELAY);
            *ptr = '\0';
            break;
        } else if (tmp >= '0' && tmp <= '9') {
            HAL_UART_Transmit(&huart3, (const uint8_t *) &tmp, sizeof(tmp),
            ↪  HAL_MAX_DELAY);
            *ptr = tmp;
            ptr++;
        }

    }

}
```

send_int()

```
void send_int(UART_HandleTypeDef *dst_uart, uint32_t val) {

    int i;
    uint8_t src;

    for (i = 0; i < 4; i++) {
        val = (val >> (i * 8));
        src = val & 0xff;
        HAL_UART_Transmit(dst_uart, (const uint8_t *) &src, sizeof(src),
        ↪  HAL_MAX_DELAY);
    }

}
```

recv_int()

```
uint32_t recv_int(char *buf) {

    int i;
    uint32_t val = 0;

    for (i = 0; i < 4; i++) {
        val = val | (((uint32_t) *(buf + i)) << (i * 8));
    }

    return val;
}
```

print_doubled()

```
void print_doubled(int val) {

    char value_buf[VALUE_READ_BUF_SIZE * 4];
    int len;

    char got[] = "Got: ";
    char processed[] = "Processed: ";

    HAL_UART_Transmit(&huart3, (const uint8_t *) got, sizeof(got),
    ↪  HAL_MAX_DELAY);

    sprintf(value_buf, "%d", val);
    len = strlen(value_buf);
    HAL_UART_Transmit(&huart3, (const uint8_t *) value_buf, len,
    ↪  HAL_MAX_DELAY);

    HAL_UART_Transmit(&huart3, (const uint8_t *) " ", 1, HAL_MAX_DELAY);

    HAL_UART_Transmit(&huart3, (const uint8_t *) processed, sizeof(processed),
    ↪  HAL_MAX_DELAY);

    sprintf(value_buf, "%d", val * 2);
    len = strlen(value_buf);
    HAL_UART_Transmit(&huart3, (const uint8_t *) value_buf, len,
    ↪  HAL_MAX_DELAY);

    HAL_UART_Transmit(&huart3, (const uint8_t *) "\n\r", 2, HAL_MAX_DELAY);
}
```

HAL_UART_RxCpltCallback(): The Callback Function

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {

    uint32_t tmp;

    if (huart->Instance == USART1) {
        tmp = recv_int(Gbuf1);
    } else if (huart->Instance == USART2) {
        tmp = recv_int(Gbuf2);
    }

    print_doubled(tmp);

    return;

}
```

3.4.2 USB On The Go

STM32 Nucleo boards supports USB OTG in full speed mode via a USB **Micro-AB** connector (CN13). The USB power switch (U18) is connected to V_{BUS}.

Note USB OTG (On The Go) means the electronics can act either as a host or a device. This is done via HNP[a], and SRP[b]. The board is also capable of ADP[c] to determine the attachment of devices.

[a] Host Negotiation Protocol
[b] Session Request Protocol
[c] Attach Detection Protocol

Warning Do not connect Micro-AB connector before properly powering the Nucleo-144 board. As the Micro-AB connector cannot power the board. There is a chance of current injection.

Important STM32H755ZI is capable of USB HS[a] but the Nucleo board only supports upto USB FS[b]. In order to use USB HS, we need to use an external PHY through the ULPI[c] interface.

[a] High Speed
[b] Full Speed
[c] UTMI+ Low Pin Interface

USB 2.0: An Overview

USB 2.0 uses 4 pins to enable communication. These pins are:

Pins	5V	DM	DP	GND
Function	Power	Differential Pairs		Ground

The data is send as differential signal using DP and DM pins. This improves noise immunity.

Note USB 3.0 and above uses extra pins to enable higher speeds. Also note that the STM32H755ZI is only capable of USB 2.0.

Different USB Speed Modes

USB Modes	Low Speed (LS)	Full Speed (FS)	High Speed (HS)
Speed	1.5 Mbit/s	12 Mbit/s	480 Mbit/s

Different Modes

USB features can be divided into three categories:

- **General:**
- **Host-mode:** To be used as a host / master.
- **Device-mode:** To be used as a peripheral.

Endpoints

- **Control:**

  Commonly used for configuring the devices, retrieving data, sending commands and retrieving status. Usually in small size. And it is guaranteed to have reserved bandwidth.

- **Interrupt:**

  For sending small amount of data at a fixed interval. Used for keyboards and mice. Have reserved bandwidth.

- **Bulk:**

  For sending large amount of data. Can transfer huge amount of data without any data lose. Does not guaranteed to make it through in a specific amount of time. There won't be enough room on the bus to transfer huge amount of data, so the packet is split in multiple smaller packets.

- **Isochronous:**

  For sending large amount of data. For devices that need continuous stream of data but can handle data loss. It is periodic.

Note Interrupt and Isochronous are periodic and have reserved bandwidth. While Control and Bulk are asynchronous. Bulk does not have a reserved bandwidth.

3.5 FRI - FEB 06 2026

3.5.1 USB On The Go (Continued)

Hardware Side Specific to STM32H755ZI

The device is capable of USB HS through an external PHY. It already has an embedded USB FS PHY. The nucleo board is capable of USB FS as it does not have an external USB HS PHY. In STM32H7, there are two instances of OTG_HS, namely OTG_HS1 and OTG_HS2. Only the OTG_HS1 instance is capable of achieving USB HS through an external PHY.

The device has 4Kb of USB data RAM with advanced FIFO[3] control. This memory can be partitioned by the user for each of the endpoints.

[3]First in first out

OTG_HS Core

OTG_HS receives the 48 MHz from the RCC[4]. There are different registers associated with the OTG_HS core that helps to control the core.

3.5.2 I2C: Inter-Integrated Circuit Interface of STM32H755ZI

The I2C bus interface handles communication between the microcontroller and the serial I2C bus. The interface provides multi-master capability, and controls all the I2C bus-specific sequencing, protocol, arbitration and timing. It supports the following modes:

Modes	Standard Mode	Fast Mode	Fast Mode Plus
Speed	Up to 100 kHz	Up to 400 kHz	Up to 1 MHz

Note Note that the I2C bus interface is SMBus[a] and PMBus[b] compatible. Also DMA can be used to reduce CPU overload.

[a]System Management Bus
[b]Power Management Bus

I2C: Hardware Overview

Hardware Interface: Hardware bus consists of two lines:

Pin	SDA	SCL
Function	Serial Data (Data Line)	Serial Clock (Clock Line)

Line Logic: These lines are open collector or open drain. Meaning the devices can only drive them low. Default / Idle state is high.

Note We need to use a pull up resistor to pull these lines high. Usually we use 1 kOhm to 10 kOhm. Then the pull up current will be about 1mA or less.

Note Also the driving device need to sink at least 10mA or more current.

Pin Connection

[4]Reset and Clock Controller

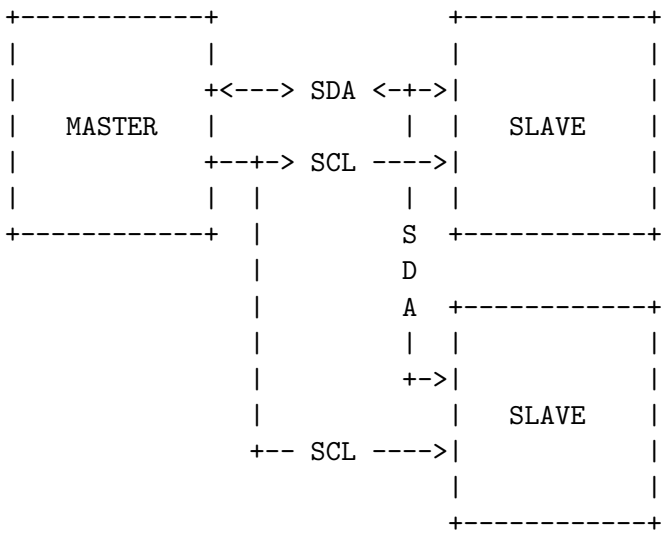

Figure 3.4: Typical connection diagram of SDA and SCL pins in a single master application.

SDA Line: This line is used to send the address from the master to slave. And to send and receive data to and from the master and slaves.

SCL Line: This line is used to provide clock.

Voltage Levels: Supply Voltage: Ranging from 1.2V to 5.5V

I2C: Communication Protocol

Every data frame or the transaction is started with a `START` condition and ends with `STOP` condition.

Clock Stretching: In I2C it is possible to stretch[5] the SCL line. If any of the device is slow to process the data, it can pull the SCL line low.

Arbitration: It's a mechanism to decide which master should take control of the I2C bus.

`TODO: FRAME`

More about STM32H755ZI's I2C Interface: Implementation

There are four I2C channels implemented in STM32H755ZI. They are: `I2C1`, `I2C2`, `I2C3`, and `I2C4`. All of these interfaces can be used in the 3 modes. These interfaces can also generate interrupts. And they can be enabled in the software.

Important In order to use these interfaces in Fast-mode plus, we need to enable 20 mA output current in the control bit present in SYSCFG register.

[5] Analogues to pausing the clock.

Modes for these Interfaces: These interfaces can operate in one of the four modes:

Modes	Slave Transmitter	Slave Receiver	Master Transmitter	Master Receiver

By default, the interface will operate in slave mode. Once it generates[6] a `START` condition, it will automatically switches to master mode. If the arbitration losses or a `STOP` condition is generated, it will automatically switches back to slave mode.

I2C Initialization

- First of all we need to configure and enable I2C peripheral clock in the clock controller[7].
- Then the interface can be enabled by setting PE bit in I2C_CR1 register.

[6] From the software
[7] In RCC

DCT Internship: Weekly Report

Contents

4.1 MON - FEB 09 2026

4.1.1 UART - I2C Exercise

Exercise Question: Configure two UART interfaces, one I2C interfaces and an onboard LED.

Based on the command received location:

- If command received from ST-LINK's UART, then relay that to the other UART.
- If command received from the other UART, then relay that to the I2C.
- If command received from I2C, do as such to the LED.

Then connect the other UART and I2C to another STM32 and test.

Solution Overview

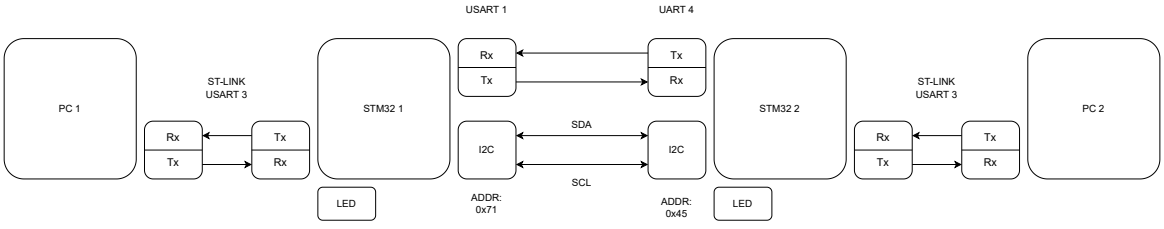

Figure 4.1: Connection block diagram of UART - I2C Exercise.

Some Important Registers Related with I2C

`TODO: MOVE THIS TO SOMEWHERE ELSE`

Registers:

- I2C_CR1
- I2C_TIMINGR
  - PRESC[3:0]
  - SCLDEL[3:0]
  - SDADEL[3:0]
- I2C_CR2
- I2C_ISR
- I2C_RXDR
- RXNE bit should be zero to receive.
- I2C_TXDR
- TXE bit should be zero to send.
- NBYTES[7:0] in I2C_CR2
- I2C_OAR1
- I2C_OAR2

4.2 TUE - FEB 10 2026

4.2.1 UART - I2C Exercise (Continued)

4.3 WED - FEB 11 2026

4.3.1 SPI: Serial Peripheral Interface

Serial peripheral interface is used for synchronous serial communication in embedded systems for short distance wired communication. The standard SPI supports full duplex[1]. It follows a master-slave architecture. The master orchestrates the communication to and from one or more slave devices.

Pin Description and Connection

The standard SPI makes use of 4 pins in total. They are:

[1] Two way communication at the same time

Pin	$\overline{\texttt{CS}}$	SCLK	MOSI[2]	MISO[3]
Function	Chip Select	Serial Clock	Data to Slave from Master	Data to Master from Slave

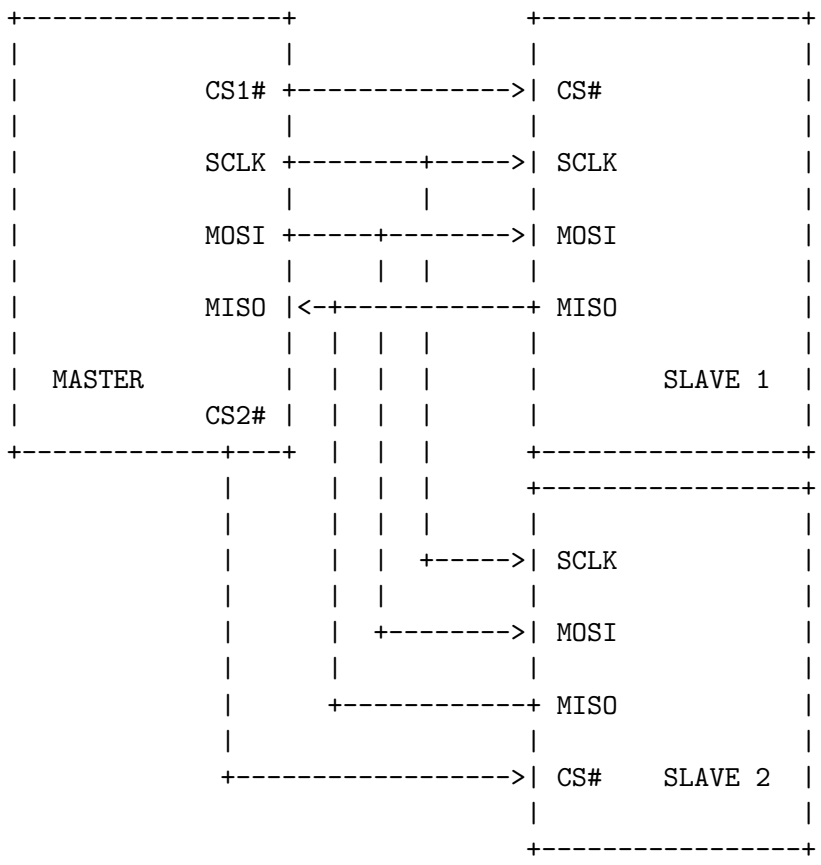

Figure 4.2: Typical connection diagram of MOSI, MISO, SCLK, and $\overline{\text{CSx}}$ pins in a single master SPI application.

From figure 4.2 we can see that the SCLK, MOSI, and MISO lines are common to all the slaves. Each of the slave device will have it's corresponding $\overline{\texttt{CS}}$ line.

Tip

Figure 4.2 shows the regular mode setup. As the number of devices increases the number of $\overline{\texttt{CS}}$ lines needed also increases. That might become a problem. One way to solve this issue is to use a decoder to generate the $\overline{\texttt{CS}}$ lines. There is another way to solve this issue by using **daisy-chain**[a] method.

[a] Look up daisy-chaining SPI for more info.

[2] Master Out Slave In.
[3] Master In Slave Out.

Note Native SPI does not support multi-master configuration. In order to use multi-master we need some way to perform master arbitration. STM32H755ZI's reference manual also does not mention about any in-built way to use the SPI in mutli-master mode.

Important MISO line should be tri-stated, as it needs to be driven from different slaves at different times depending on the state of the corresponding chip select pin.

Note MISO and MOSI are also called SDI[a] and SDO[b] respectively.

[a] Serial Data In.
[b] Serial Data Out.

Data Transmission

The master initiates the transaction by asserting the corresponding $\overline{\texttt{CS}}$ line. The clock is also provided by the master. Full duplex communication is carried out by the master and the selected slave. The data is sampled or pushed out at the edge[4] of the `SCLK`.

SPI Modes: Clock Polarity and Clock Phase

SPI has four modes, the master can choose these modes using CPOL[5] and CPHA[6] bits. See table 4.1 for the corresponding polarity and phase.

SPI Mode	CPOL	CPHA	Clock Polarity in Idle State	Clock Phase Used to Sample Data	Clock Phase Used to Shift the Data
0	0	0	Logic Low	Rising Edge	Falling Edge
1	0	1	Logic Low	Falling Edge	Rising Edge
2	1	0	Logic High	Falling Edge	Rising Edge
3	1	1	Logic High	Rising Edge	Falling Edge

Table 4.1: CPOL and CPHA states and corresponding clock polarity and edge selections for sampling and shifting data.

Data Frame

The neat thing about SPI is that it does not define any standard data frame. This allows the users to set the frame as we wish. This also means that interfacing[7] will be different to each slave devices.

[4] Can be configured to rising or falling edge, depending on the device.
[5] Clock Polarity.
[6] Clock Phase.
[7] From the data frame's perspective.

4.3.2 SPI: Serial Peripheral Interface of STM32H755ZI

STM32H755ZI's has 6 SPI instances. And they supports full-duplex, half-duplex and simplex transactions. It has multi-master mode and multi-slave mode capabilities. The SS[8] can be managed by both the hardware and software. Can configure SS signal polarity and timing. MISO and MOSI swappable. Configurable polarity and phase using CPOL and CPHA bits. Configurable data order (choosing whether to MSB or LSB first). SPI Motorola and TI formats supported. Hardware CRC feature.

Relevant registers:

- SPI_CFG2.
- COMM[1:0] bits.

`TODO: REST OF SPI`

4.3.3 QUADSPI of STM32H755ZI

Along with SPI, STM32H755ZI features a QUADSPI.

Note This QUADSPI can interface single-, dual-, or quad-SPI devices.

QUADSPI in STM32H755ZI have 3 modes of operation:

Mode	Description
Indirect Mode	The operations are done using the QUADSPI registers.
Automatic Static-Polling Mode	External memory registers are periodically read, and can generated an interrupt in case of flag change.
Memory-Mapped Mode	External memory is mapped to the device address space and can be seen by the system as if it was the internal memory.

Table 4.2: STM32H755ZI's QUADSPI modes.

QUADSPI: Main Features

- 3 Functional modes: indirect, automatic static-polling, and memory-mapped modes.
- Dual-flash mode: 8 bits can be sent / received simultaneously by accessing two flash memories in parallel.
- SDR[9] and DDR[10] support.
- Fully programmable opcodes for both indirect and memory-mapped modes.
- Fully programmable frame format for both indirect and memory-mapped modes.
- 8-, 16-, 32-bit data access.

[8] Slave Select.
[9] Single Data Read.
[10] Double Data Read.

QUADSPI Command Sequence: While

TODO: REST OF QUADSPI