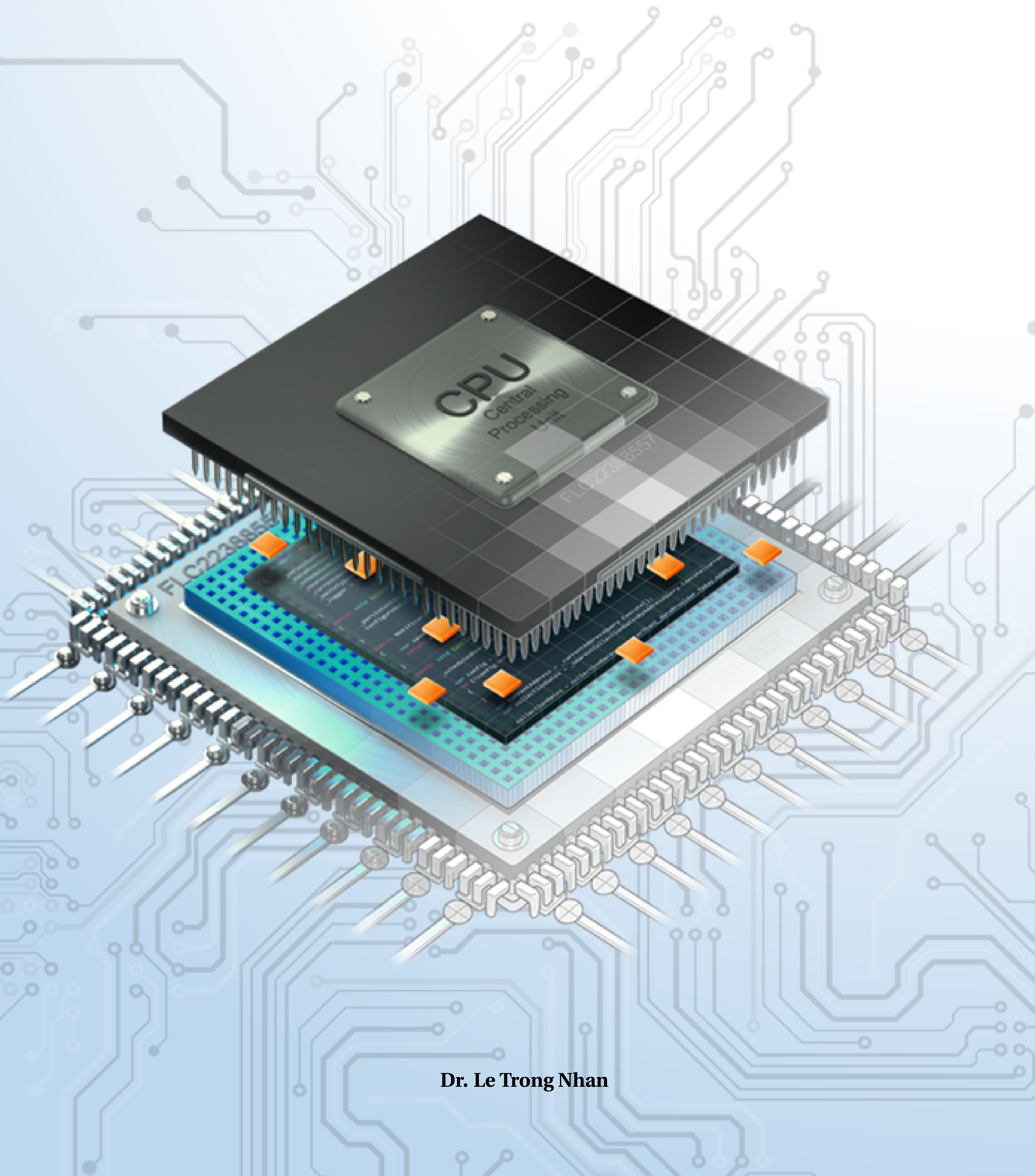
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
COMPUTER ENGINEERING

Microcontroller



Dr. Le Trong Nhan

Contents

CHAPTER 1

Flow and Error Control in Communication

1.1 Introduction

Flow control and Error control are the two main responsibilities of the data link layer, which is a communication channel for node-to-node delivery of the data. The functions of the flow and error control are explained as follows.

Flow control mainly coordinates with the amount of data that can be sent before receiving an acknowledgment from the receiver and it is one of the major duties of the data link layer. For most of the communications, flow control is a set of procedures that mainly tells the sender how much data the sender can send before it must wait for an acknowledgment from the receiver.

A critical issue, but not really frequently occurred, in the flow control is that the processing rate is slower than the transmission rate. Due to this reason each receiving device has a block of memory that is commonly known as buffer, that is used to store the incoming data until this data will be processed. In case the buffer begins to fill-up then the receiver must be able to tell the sender to halt the transmission until once again the receiver become able to receive.

Meanwhile, error control contains both error detection and error correction. It mainly allows the receiver to inform the sender about any damaged or lost frames during the transmission and then it coordinates with the re-transmission of those frames by the sender.

The term Error control in the communications mainly refers to the methods of error detection and re-transmission. Error control is mainly implemented in a simple way and that is whenever there is an error detected during the exchange, then specified frames are re-transmitted and this process is also referred to as Automatic Repeat request(ARQ).

The target in this lab is to implement a UART communication between the STM32 and a simulated terminal. A data request is sent from the terminal to the STM32. Afterward, computations are performed at the STM32 before a data packet is sent to the terminal. The terminal is supposed to reply an ACK to confirm the communication successfully or not.

1.2 Proteus simulation platform

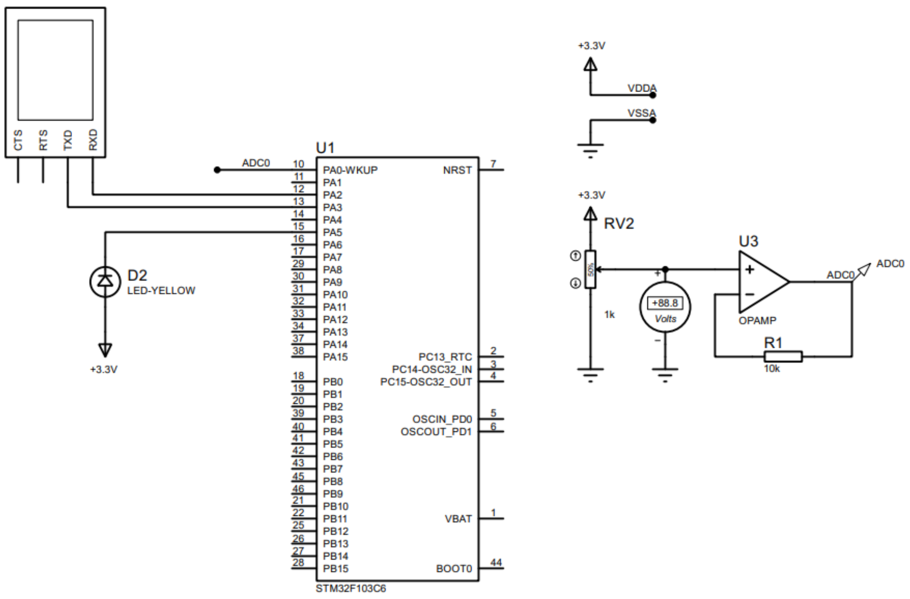


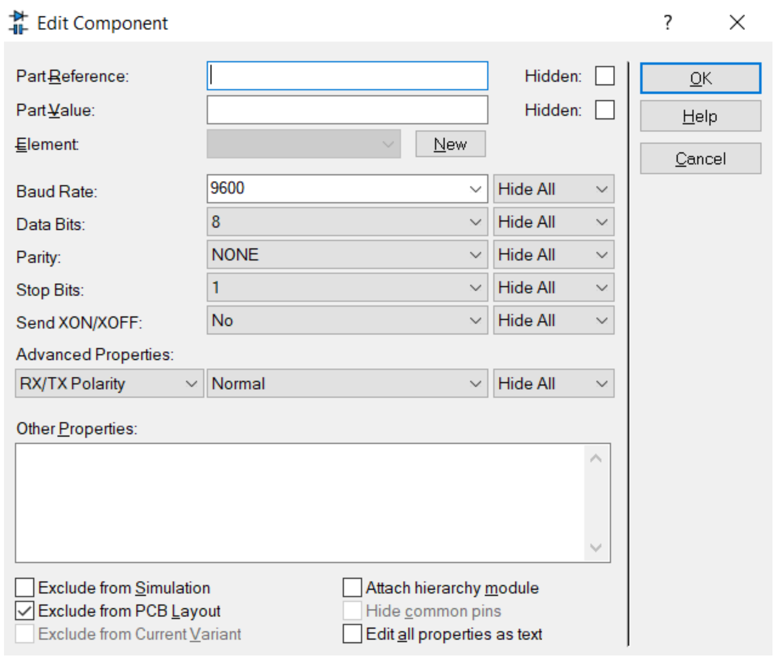
Figure 1.1: *Simulation circuit on Proteus*

Some new components are listed bellow:

- Terminal: Right click, choose Place, Virtual Instrument, then select VIRTUAL TERMINAL.
- Variable resistor (RV2): Right click, choose Place, From Library, and search for the POT-HG device. The value of this device is set to the default 1k.
- Volt meter (for debug): Right click, choose Place, Virtual Instrument, the select DC VOLTMETER.
- OPAMP (U3): Right click, choose Place, From Library, and search for the OPAMP device.

The opamp is used to design a voltage follower circuit, which is one of the most popular applications for opamp. In this case, it is used to design an adc input signal, which is connected to pin PA0 of the MCU.

Double click on the virtual terminal and set its baudrate to 9600, 8 data bits, no parity and 1 stop bit, as follows:


Edit Component

Part Reference:		Hidden: ☐
Part Value:		Hidden: ☐
Element:		New
Baud Rate:	9600	Hide All
Data Bits:	8	Hide All
Parity:	NONE	Hide All
Stop Bits:	1	Hide All
Send XON/XOFF:	No	Hide All

Advanced Properties:

RX/TX Polarity	Normal	Hide All

Other Properties:

- ☐ Exclude from Simulation
- ☑ Exclude from PCB Layout
- ☐ Exclude from Current Variant
- ☐ Attach hierarchy module
- ☐ Hide common pins
- ☐ Edit all properties as text

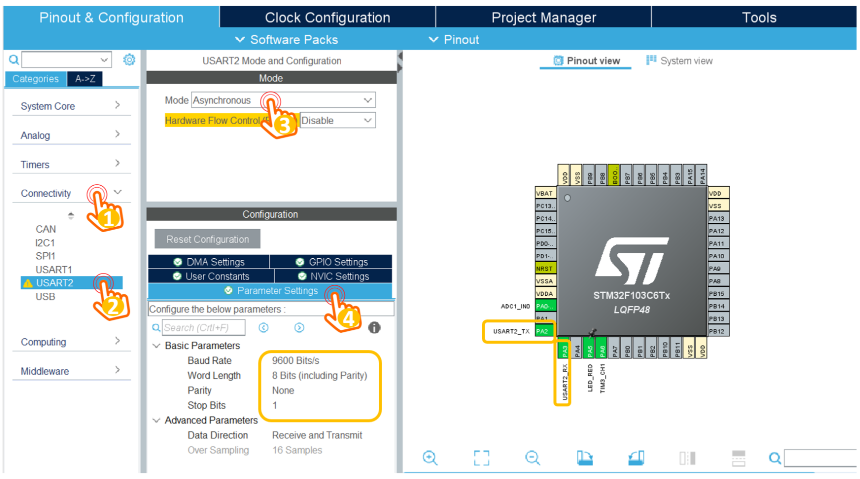
Figure 1.2: *Terminal configuration*

1.3 Project configurations

A new project is created with following configurations, concerning the UART for communications and ADC input for sensor reading. The pin PA5 should be an GPIO output, for LED blinky.

1.3.1 UART Configuration

From the ioc file, select **Connectivity**, and then select the **USART2**. The parameter settings for UART channel 2 (USART2) module are depicted as follows:


USART2 Mode and Configuration

Mode: Asynchronous
Hardware Flow Control (RS232): Disable

Configure the below parameters :

Basic Parameters

Baud Rate	9600 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1

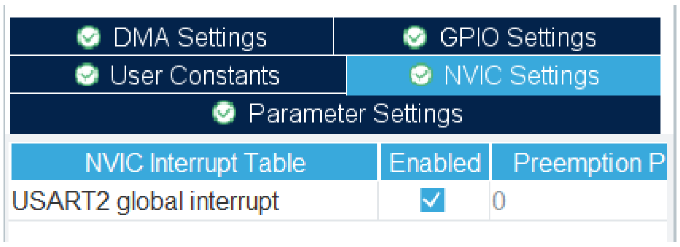
Advanced Parameters

Data Direction	Receive and Transmit
Over Sampling	16 Samples

Figure 1.3: *UART configuration in STMCube*

The UART channel in this lab is the Asynchronous mode, 9600 bits/s with no Parity and 1 stop bit. After the uart is configured, the pins PA2 (Tx) and PA3(Rx) are enabled.

Finally, the NVIC settings are checked to enable the UART interrupt, as follows:

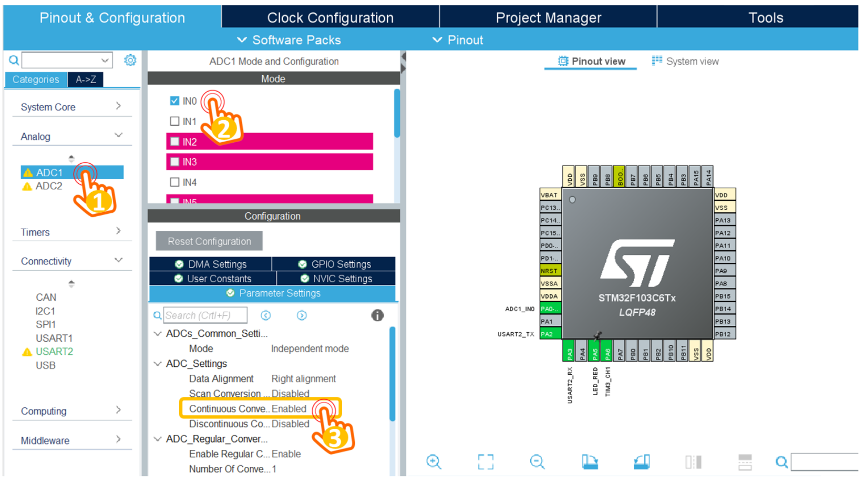


NVIC Interrupt Table	Enabled	Preemption P
USART2 global interrupt	✓	0

Figure 1.4: *Enable UART interrupt*

1.3.2 ADC Input

In order to read a voltage signal from a simulated sensor, this module is required. By selecting on **Analog**, then **ADC1**, following configurations are required:



ADCs_Common_Setti...	
Mode	Independent mode
ADC_Settings	
Data Alignment	Right alignment
Scan Conversion ...	Disabled
Continuous Conve...	Enabled
Discontinuous Co...	Disabled
ADC_Regular_Conver...	
Enable Regular C...	Enable
Number Of Conve...	1

Figure 1.5: *ADC configuration in STMCube*

The ADC pin is configured to PA0 of the STM32, which is shown in the pinout view dialog.

Finally, the PA5 is configured as a GPIO output, connected to a blinky LED.

1.4 UART loop-back communication

This source is required to add in the main.c file, to verify the UART communication channel: sending back any character received from the terminal, which is well-known as the loop-back communication.

```
/* USER CODE BEGIN 0 */
uint8_t temp = 0;
```

```

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
  if(huart->Instance == USART2){
    HAL_UART_Transmit(&huart2, &temp, 1, 50);
    HAL_UART_Receive_IT(&huart2, &temp, 1);
  }
}
/* USER CODE END 0 */
```

Program 1.1: Implement the UART interrupt service routine

When a character (or a byte) is received, this interrupt service routine is invoked. After the character is sent to the terminal, the interrupt is activated again. This source code should be placed in a user-defined section.

Finally, in the main function, the proposed source code is presented as follows:

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();

  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_ADC1_Init();

  HAL_UART_Receive_IT(&huart2, &temp, 1);

  while (1)
  {
    HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
    HAL_Delay(500);
  }

}
```

Program 1.2: Implement the main function

1.5 Sensor reading

A simple source code to read adc value from PA0 is presented as follows:

```
uint32_t ADC_value = 0;
while (1)
{
  HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
  ADC_value = HAL_ADC_GetValue(&hadc1);
HAL_UART_Transmit(&huart2, (void *)str, sprintf(str, "%d\n"
   , ADC_value), 1000);
  HAL_Delay(500);
}
```

Program 1.3: ADC reading from AN0

Every half of second, the ADC value is read and its value is sent to the console. It is worth noticing that the number ADC_value is convert to ascii character by using the sprintf function.

The default ADC in STM32 is 13 bits, meaning that 5V is converted to 4096 decimal value. If the input is 2.5V, ADC_value is 2048.

1.6 Project description

In this lab, a simple communication protocol is implemented as follows:

- From the console, user types **!RST#** to ask for a sensory data.
- The STM32 response the ADC_value, following a format **!ADC=1234#**, where 1234 presents for the value of ADC_value variable.
- The user ends the communication by sending **!OK#**

The timeout for waiting the **!OK#** at STM32 is 3 seconds. After this period, its packet is sent again. **The value is kept as the previous packet**.

1.6.1 Command parser

This module is used to received a command from the console. As the reception process is implement by an interrupt, the complexity is considered seriously. The proposed implementation is given as follows.

Firstly, the received character is added into a buffer, and a flag is set to indicate that there is a new data.

```c
#define MAX_BUFFER_SIZE  30
uint8_t temp = 0;
uint8_t buffer[MAX_BUFFER_SIZE];
uint8_t index_buffer = 0;
uint8_t buffer_flag = 0;
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
  if(huart->Instance == USART2){

    //HAL_UART_Transmit(&huart2, &temp, 1, 50);
    buffer[index_buffer++] = temp;
    if(index_buffer == 30) index_buffer = 0;

    buffer_flag = 1;
    HAL_UART_Receive_IT(&huart2, &temp, 1);
  }
}
```

Program 1.4: Add the received character into a buffer

A state machine to extract a command is implemented in the while(1) of the main function, as follows:

```c
while (1){
    if(buffer_flag == 1){
        command_parser_fsm();
        buffer_flag = 0;
    }
}
```

Program 1.5: State machine to extract the command

The output of the command parser is to set **command_flag** and **command_data**. In this project, there are two commands, **RTS** and **OK**. The program skeleton is proposed as follows:

```c
while (1){
    if(buffer_flag == 1){
        command_parser_fsm();
        buffer_flag = 0;
    }
    uart_communiation_fsm();
}
```

Program 1.6: Program structure

1.6.2 Project implementation

Students are proposed to implement 2 FSM in seperated modules. Students are asked to design the FSM before their implementations in STM32Cube.

- **Finite State Machine design:**

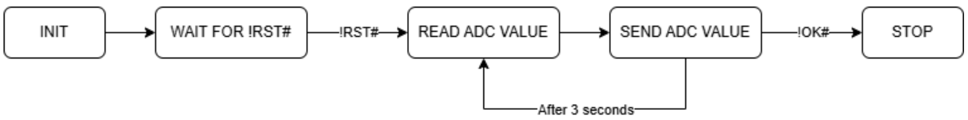


Figure 1.6: *Finite State Machine diagram*

- **Command parser FSM:**

```
#define MAX_BUFFER_SIZE 30

// INITIAL
uint8_t buffer[MAX_BUFFER_SIZE];
uint8_t buffer_index = 0;
uint8_t command_flag = 0;
uint8_t command_data = 0;


// COMMAND PARSER FSM
void command_parser_init(void)
{
  buffer_index = 0;
  command_flag = 0;
};

void command_parser_input(uint8_t input_data)
{
  buffer[buffer_index++] = input_data;
  if(buffer_index >= MAX_BUFFER_SIZE) buffer_index = 0;
};
void command_parser_fsm(void)
{
  // Identify commands with syntax "!---#"

  if(buffer[buffer_index - 1] == '#')
  {
    if(strstr((char*)buffer, "!RST#") )
    {
      command_flag = 1; // command detected
      command_data = CMD_RST; // "!RST#"
    }
    else if(strstr((char*)buffer, "!OK#") )
    {
      command_flag = 1; // command detected
      command_data = CMD_OK; // "!OK#"
    }

    // Reset buffer
    buffer_index = 0;
    memset(buffer, 0, MAX_BUFFER_SIZE);
  }
};
```

Program 1.7: Command parser FSM

- **UART communication FSM:**

```
// INITIAL
```

```c
uint8_t uart_state = 0; // UART_STATE_IDLE -> idle,
   UART_STATE_WAIT -> wait ACK
uint32_t last_tick = 0;
char str[50];


// UART COMMUNICATION FSM
void uart_communication_init(void)
{
  uart_state = 0;
  last_tick = HAL_GetTick();
};
void uart_communication_fsm(void)
{
  uint32_t ADC_value = 0;

  switch(uart_state)
  {
  case 0: // Idle
    if(command_flag && command_data == CMD_RST)
    {
      // Read ADC value
      ADC_value = HAL_ADC_GetValue(&hadc1);

      // Display ADC value on terminal
      sprintf(str, "!ADC=%lu#\r\n", ADC_value);
      HAL_UART_Transmit(&huart2, (uint8_t*)str, strlen(
   str), 1000);

      // Reset UART state
      uart_state = UART_STATE_WAIT;

      // Get tick to check timeout
      last_tick = HAL_GetTick();

      // Reset command flag
      command_flag = 0;
    }
    break;

  case 1: // Wait ACK
    if(command_flag && command_data == CMD_OK)
    {
      // Reset UART state
      uart_state = UART_STATE_IDLE;

      // Reset command flag
      command_flag = 0;
    }
    if(HAL_GetTick() - last_tick >= 3000)
    {
      // Resend ADC value
      HAL_UART_Transmit(&huart2, (uint8_t*)str, strlen(
  str), 1000);

      // Get tick to check timeout
      last_tick = HAL_GetTick();
    }
    break;

  default:
    uart_state = UART_STATE_WAIT;
    break;
  }
};
```

Program 1.8: UART communication FSM

- **Github link:**

  STM32CubeIDE source code and Proteus schematic of Lab5 Exercise