



HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
COMPUTER ENGINEERING

Microcontroller



Dr. Le Trong Nhan



Microprocessor - Microcontroller (CO3009)

Laboratory Report

LAB 5

Teacher: Le Trong Nhan
Huynh Phuc Nghi
Class: L03
Student: Pham Van Nhat Vu (2110676)

HO CHI MINH CITY, 2023

Contents

Chapter 1. Flow and Error Control in Communication	5
1 Introduction	6
2 Proteus simulation platform	7
3 Project configurations	8
3.1 UART Configuration	8
3.2 ADC Input	9
4 UART loop-back communication	9
5 Sensor reading	10
6 Project description	11
6.1 Command parser	11
6.2 Project implementation	12
6.3 Answer	13
6.3.1 FSM Design	13
6.3.2 Simulation schematic	14
6.3.3 Implementation	14

CHAPTER 1

Flow and Error Control in Communication



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.

2 Proteus simulation platform

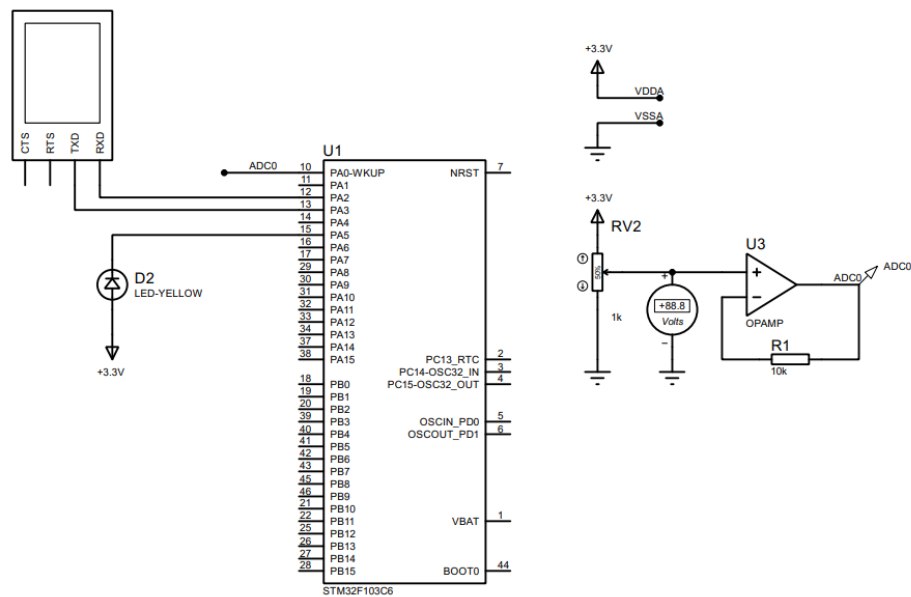


Figure 1.1: Simulation circuit on Proteus

Some new components are listed below:

- 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:

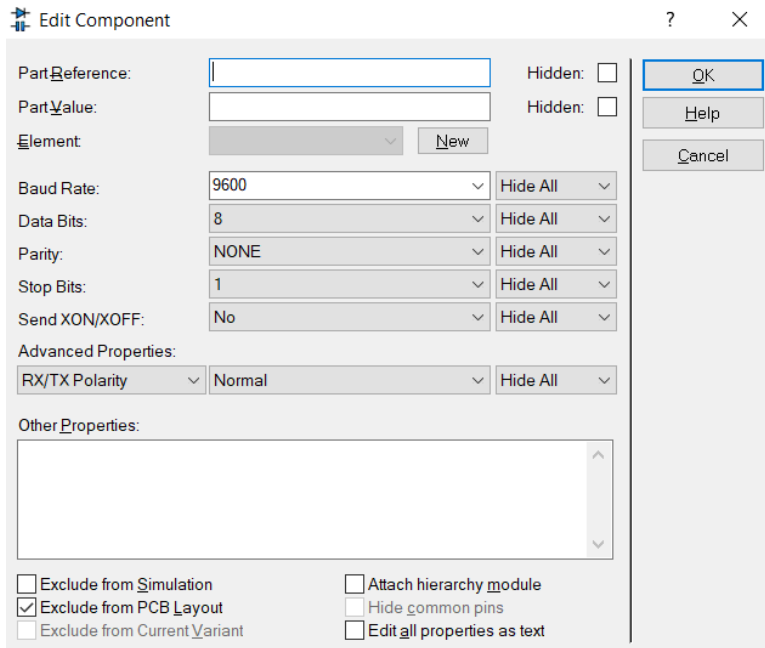


Figure 1.2: Terminal configuration

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.

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:

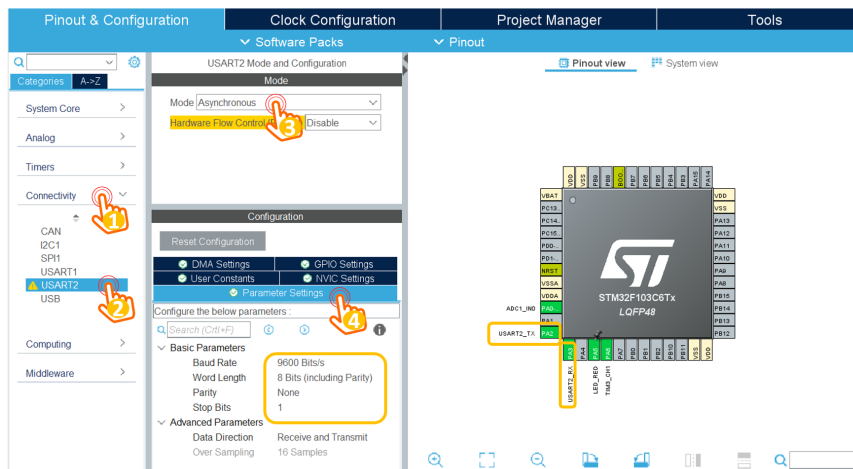


Figure 1.3: UART configuration in STMCube

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:







 DMA Settings	 GPIO Settings	
 User Constants	 NVIC Settings	
 Parameter Settings		
NVIC Interrupt Table	Enabled	Preemption P
USART2 global interrupt		0

Figure 1.4: Enable UART interrupt

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:

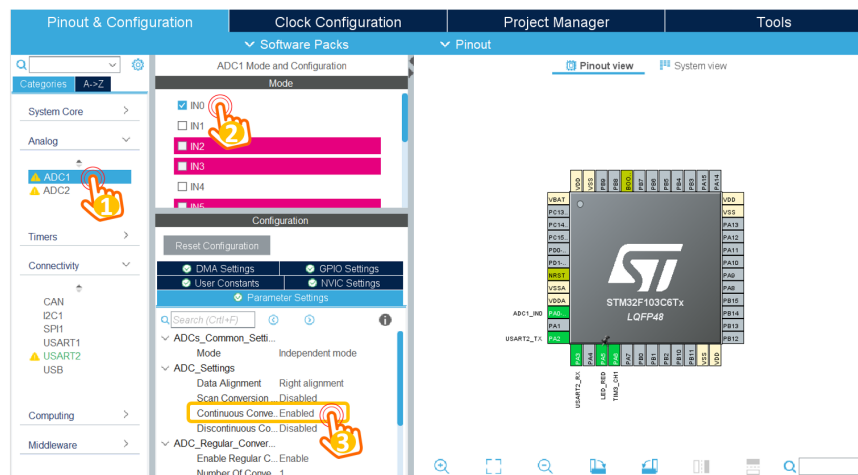


Figure 1.5: Enable UART interrupt

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.

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.

```
1 /* USER CODE BEGIN 0 */
2 uint8_t temp = 0;
3
```

```

4 void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
5     if(huart->Instance == USART2){
6         HAL_UART_Transmit(&huart2, &temp, 1, 50);
7         HAL_UART_Receive_IT(&huart2, &temp, 1);
8     }
9 }
10 /* 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:

```

1 int main(void)
2 {
3     HAL_Init();
4     SystemClock_Config();
5
6     MX_GPIO_Init();
7     MX_USART2_UART_Init();
8     MX_ADC1_Init();
9
10    HAL_UART_Receive_IT(&huart2, &temp, 1);
11
12    while (1)
13    {
14        HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
15        HAL_Delay(500);
16    }
17
18 }

```

Program 1.2: Implement the main function

5 Sensor reading

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

```

1 uint32_t ADC_value = 0;
2 while (1)
3 {
4     HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
5     ADC_value = HAL_ADC_GetValue(&hadc1);
6     HAL_UART_Transmit(&huart2, (void *)str, sprintf(str, "%d\n",
7     , ADC_value), 1000);
7     HAL_Delay(500);

```

8 }

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.

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.**

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.

```
1 #define MAX_BUFFER_SIZE 30
2 uint8_t temp = 0;
3 uint8_t buffer[MAX_BUFFER_SIZE];
4 uint8_t index_buffer = 0;
5 uint8_t buffer_flag = 0;
6 void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
7     if(huart->Instance == USART2){
8
9         //HAL_UART_Transmit(&huart2, &temp, 1, 50);
10        buffer[index_buffer++] = temp;
11        if(index_buffer == 30) index_buffer = 0;
12
13        buffer_flag = 1;
14        HAL_UART_Receive_IT(&huart2, &temp, 1);
15    }
```

```
16 }
```

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:

```
1 while (1){
2     if(buffer_flag == 1){
3         command_parser_fsm();
4         buffer_flag = 0;
5     }
6 }
```

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:

```
1 while (1){
2     if(buffer_flag == 1){
3         command_parser_fsm();
4         buffer_flag = 0;
5     }
6     uart_communiation_fsm();
7 }
```

Program 1.6: Program structure

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.

6.3 Answer

6.3.1 FSM Design

UART communication FSM

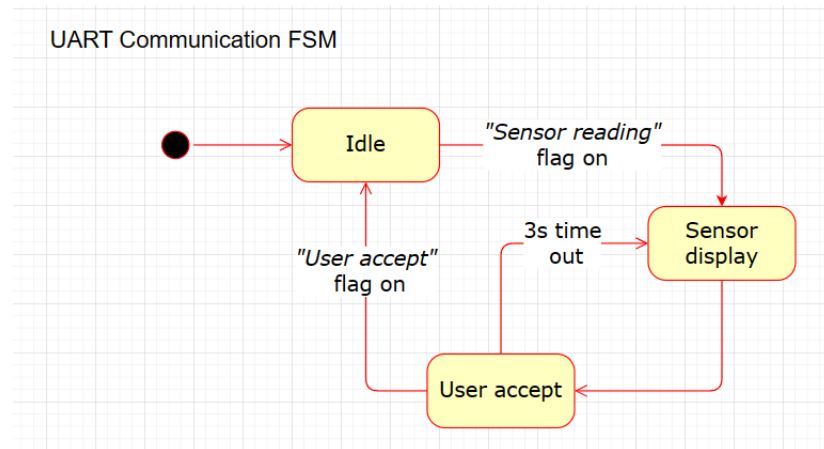


Figure 1.6: UART communication FSM

This FSM has three states:

- *Idle*: The STM32 **does not** send anything to the terminal.
- *Sensor display*: The STM32 send the **voltage** at the output of the sensor to the terminal.
- *User accept*: The STM32 waits for the user to accept the sensor value that is being displayed (by typing **!OK#** to the virtual terminal).

Command parser FSM

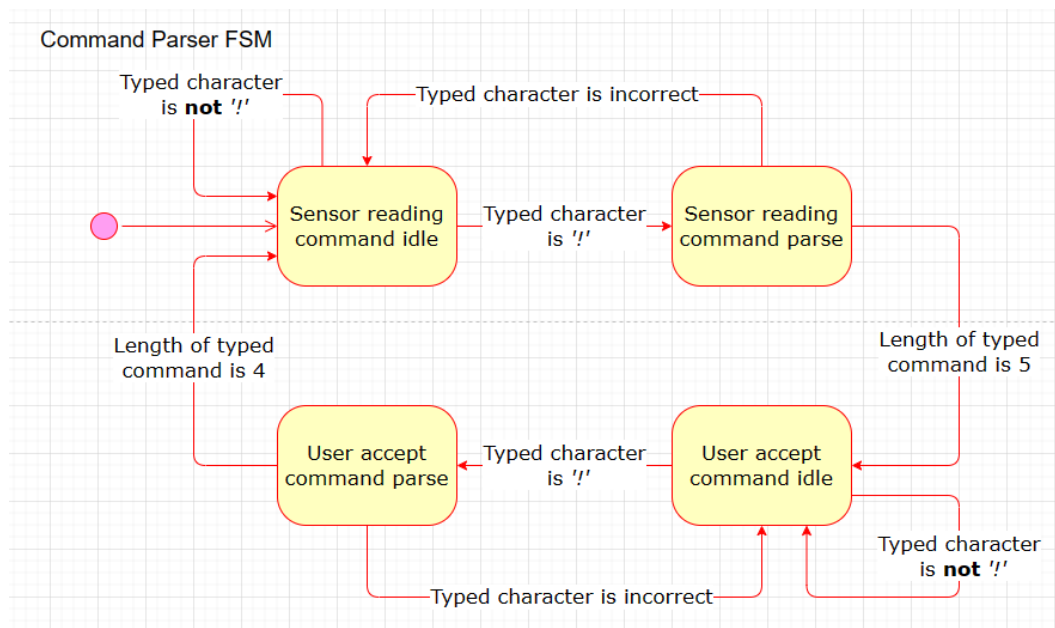


Figure 1.7: Command parser FSM

This FSM has four states:

- *Sensor reading command idle*: The STM32 waits for the user to start asking for sensory data.
- *Sensor reading command parse*: The STM32 checks whether the command the user is typing is indeed **!RST#** (used to request sensory data).
- *User accept command idle*: The STM32 waits for the user to start accepting the sensor value that is being displayed.
- *User accept command parse*: The STM32 checks whether the command the user is typing is indeed **!OK#** (used to accept the sensor value that is being displayed).

6.3.2 Simulation schematic

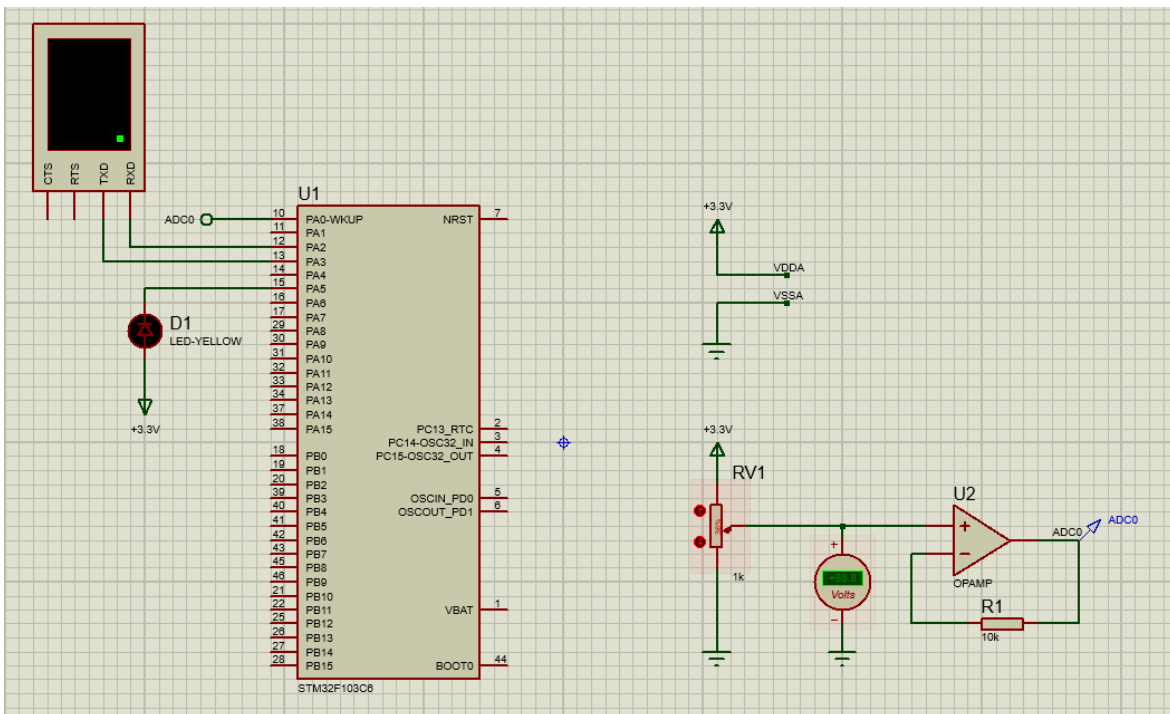


Figure 1.8: The schematic of the Proteus simulation project, which is store [here](#).

6.3.3 Implementation

The structure of the program is implemted in the **main()** function, located at the **main.c** file as follows:

```

1  ...
2  int main(void)
3  {
4      ...
5      uart_reading_init();
6      command_parser_init();
7      adc_reading_init();

```



```

8   timer_init();
9   uart_communication_init();
10
11  while (1)
12  {
13      if (buffer_flag == 1)
14      {
15          command_parser_fsm();
16          buffer_flag = 0;
17      }
18      uart_communication_fsm();
19  }
20 }
21 ...

```

Program 1.7: Program structure

The init functions shown above are used to activate the UART (for receiving), the ADC, a timer, and to set up the initial state of the FSMs. In addition to the init function, the **uart_reading** module implements UART receiving by interrupt, as shown in program 1.4 and a **get_last_character()** function to retrieve the newest character of the buffer.

```

1  uint8_t temp = 0;
2  uint8_t buffer[MAX_BUFFER_SIZE];
3  uint8_t index_buffer = 0;
4  uint8_t buffer_flag = 0;
5
6  void uart_reading_init()
7  {
8      HAL_UART_Receive_IT(&huart2, &temp, 1);
9  }
10
11  uint8_t get_last_character()
12  {
13      if (index_buffer == 0) return buffer[MAX_BUFFER_SIZE -
14      1];
15      return buffer[index_buffer - 1];
16  }
17
18  void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
19  {
20      if (huart->Instance == USART2)
21      {
22          // Loop-back communication to observe system
23          // behavior conveniently
24          HAL_UART_Transmit(&huart2, &temp, 1, 50);
25          buffer[index_buffer] = temp;
26          ++index_buffer;
27          if (index_buffer == MAX_BUFFER_SIZE)
28          {
29              index_buffer = 0;
30          }
31      }
32  }

```

```

28     }
29     buffer_flag = 1;
30     HAL_UART_Receive_IT(&huart2, &temp, 1);
31 }
32 }

```

Program 1.8: The **uart_reading** module

The **adc_reading** module contains the **read_sensor_value()** functions, used to read sensory data ranging from 0 to 4095. Additionally, it includes the **transmit_sensor_value()** function to send that value to the terminal after converting it to a voltage value in the range of 0 to 5V.

```

1  #define VCC_VALUE 4095
2  #define GND_VALUE 0
3  #define VCC_VOLTAGE 5
4
5  #include "adc_reading.h"
6
7  uint32_t ADC_value = 0;
8
9  void adc_reading_init()
10 {
11     HAL_ADC_Start(&hadc1);
12 }
13
14 void read_sensor_value()
15 {
16     ADC_value = HAL_ADC_GetValue(&hadc1);
17 }
18
19 void transmit_sensor_value()
20 {
21     HAL_GPIO_TogglePin(LED_TEST_GPIO_Port, LED_TEST_Pin);
22     float sensor_voltage = 1.0 * (ADC_value - GND_VALUE) /
23         (VCC_VALUE - GND_VALUE) * VCC_VOLTAGE;
24     char str[10];
25     uint8_t str_len = sprintf(str, "%4.3f\r\n",
26         sensor_voltage);
27     for (uint8_t i = 0; i < str_len; ++i)
28     {
29         HAL_UART_Transmit(&huart2, (uint8_t*)&str[i], 1, 50);
30     }
31 }

```

Program 1.9: The **adc_reading** module

The **timer** module creates a software timer to send sensory data to the terminal every 3 seconds, starting from the moment the user types the request command (!RST#) until they type the accept command (!OK#).

```

1  int sensor_timer_flag = 0;

```

```

2 int sensor_timer_counter = 0;
3
4 void timer_init()
5 {
6     HAL_TIM_Base_Start_IT(&htim2);
7 }
8
9 void set_sensor_timer(int duration)
10 {
11     sensor_timer_counter = duration / TIME_UNIT;
12     sensor_timer_flag = 0;
13 }
14
15 int is_sensor_timer_flagged()
16 {
17     return sensor_timer_flag;
18 }
19
20 void sensor_timer_run()
21 {
22     --sensor_timer_counter;
23     if (sensor_timer_counter <= 0)
24     {
25         sensor_timer_flag = 1;
26     }
27 }
28
29 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
30 {
31     if (htim->Instance == TIM2)
32     {
33         sensor_timer_run();
34     }
35 }

```

Program 1.10: The **timer** module

The **uart_communication_fsm** module is implemented based on the Figure 1.6 in the following manner:

```

1 #include "uart_communication_fsm.h"
2 #define USER_ACCEPT_TIMEOUT 3000
3
4 UARTState uartState = IDLE;
5
6 void uart_communication_init()
7 {
8     uartState = IDLE;
9 }
10
11 void uart_communication_fsm()

```

```

12 {
13     switch (uartState)
14     {
15     case IDLE:
16         if (sensor_reading_flag == 1)
17         {
18             uartState = SENSOR_DISPLAY;
19             sensor_reading_flag = 0;
20         }
21         break;
22     case SENSOR_DISPLAY:
23         set_sensor_timer(USER_ACCEPT_TIMEOUT);
24         transmit_sensor_value();
25         uartState = USER_ACCEPT;
26         break;
27     case USER_ACCEPT:
28         if (user_accept_flag == 1)
29         {
30             uartState = IDLE;
31             user_accept_flag = 0;
32         }
33         if (is_sensor_timer_flagged())
34         {
35             uartState = SENSOR_DISPLAY;
36         }
37         break;
38     default:
39         break;
40     }
41 }

```

Program 1.11: The **uart_communication_fsm** module

The **sensor_reading_flag** and **user_accept_flag** are used to indicate that the user has typed the request and accept commands at appropriate times, respectively.

Similarly, the **command_parser_fsm** module is implemented based on the Figure 1.7:

```

1 #include "command_parser_fsm.h"
2
3 ParserState parserState = SENSOR_READING_PARSE;
4
5 uint8_t sensor_reading_command[] = {'!', 'R', 'S', 'T', '#'};
6
7 uint8_t sensor_reading_command_len = 5;
8 uint8_t user_accept_command[] = {'!', 'O', 'K', '#'};
9 uint8_t user_accept_command_len = 4;
10 uint8_t command_index = 0;
11
12 uint8_t sensor_reading_flag = 0;
13 uint8_t user_accept_flag = 0;

```

```

14 void command_parser_init()
15 {
16     parserState = SENSOR_READING_IDLE;
17 }
18
19 void command_parser_fsm()
20 {
21     switch (parserState)
22     {
23     case SENSOR_READING_IDLE:
24         if (get_last_character() == '!')
25         {
26             parserState = SENSOR_READING_PARSE;
27             read_sensor_value();
28             command_index = 1;
29         }
30         break;
31     case SENSOR_READING_PARSE:
32         if (get_last_character() != sensor_reading_command[
33             command_index])
34         {
35             parserState = SENSOR_READING_IDLE;
36         }
37         ++command_index;
38         if (command_index == sensor_reading_command_len)
39         {
40             sensor_reading_flag = 1;
41             parserState = USER_ACCEPT_IDLE;
42         }
43         break;
44     case USER_ACCEPT_IDLE:
45         if (get_last_character() == '!')
46         {
47             parserState = USER_ACCEPT_PARSE;
48             command_index = 1;
49         }
50         break;
51     case USER_ACCEPT_PARSE:
52         if (get_last_character() != user_accept_command[
53             command_index])
54         {
55             parserState = USER_ACCEPT_IDLE;
56         }
57         ++command_index;
58         if (command_index == user_accept_command_len)
59         {
60             user_accept_flag = 1;
61             parserState = SENSOR_READING_IDLE;
62         }
63     }
64 }

```

```

61     break;
62 default:
63     break;
64 }
65 }

```

Program 1.12: The **command_parser_init()** function

During the period the system is at **SENSOR_READING_PARSE** state, if the *i*-th typed character (where the character '!' is indexed as 0) is not equal to the *i*-th character of the desired command (**IRST#**), the currently typing command is clearly incorrect. Consequently, the FSM returns to the **SENSOR_READING_IDLE** state. This mechanism is also applied to the **USER_ACCEPT_PARSE** state.

The entire STM32CubeIDE Project is stored [here](#).