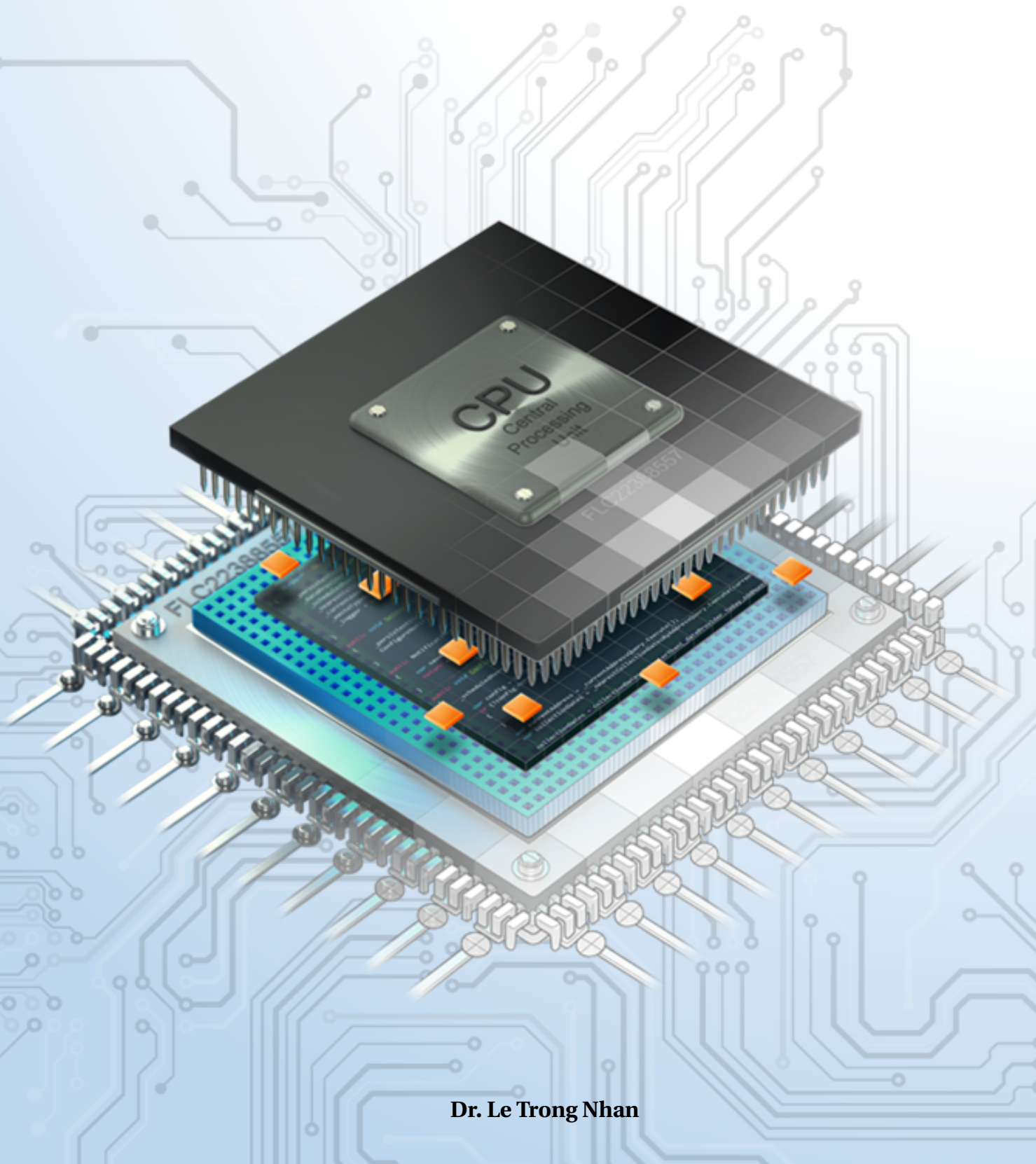




HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
COMPUTER ENGINEERING

Microcontroller



Dr. Le Trong Nhan

Mục lục

Chapter 1. Flow and Error Control in Communication	7
1 Introduction	8
2 Proteus simulation platform	9
3 Project configurations	10
3.1 UART Configuration	10
3.2 ADC Input	11
4 UART loop-back communication	11
5 Sensor reading	12
6 Project description	13
6.1 Command parser	13
6.2 Project implementation	14
6.2.1 Thiết kế máy trạng thái	14
6.2.2 Thực hiện máy trạng thái command_parser_fsm và uart_communiation_fsm	16
6.2.3 Kết quả	19

CHƯƠNG 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.

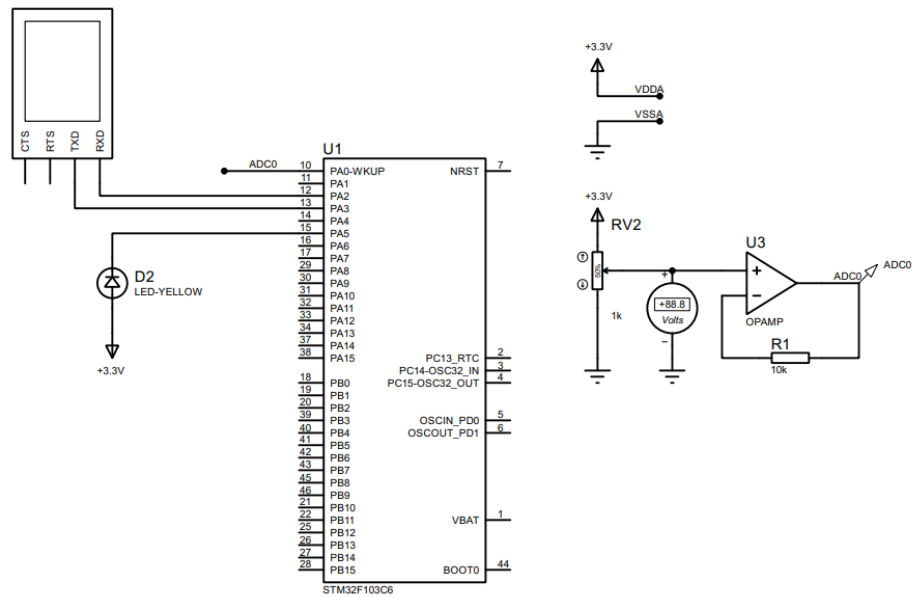
A critical issue, but not really frequently occurred, in the flow control is that the processing rate is slower than the transmission rate. Due to this reason each receiving device has a block of memory that is commonly known as buffer, that is used to store the incoming data until this data will be 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



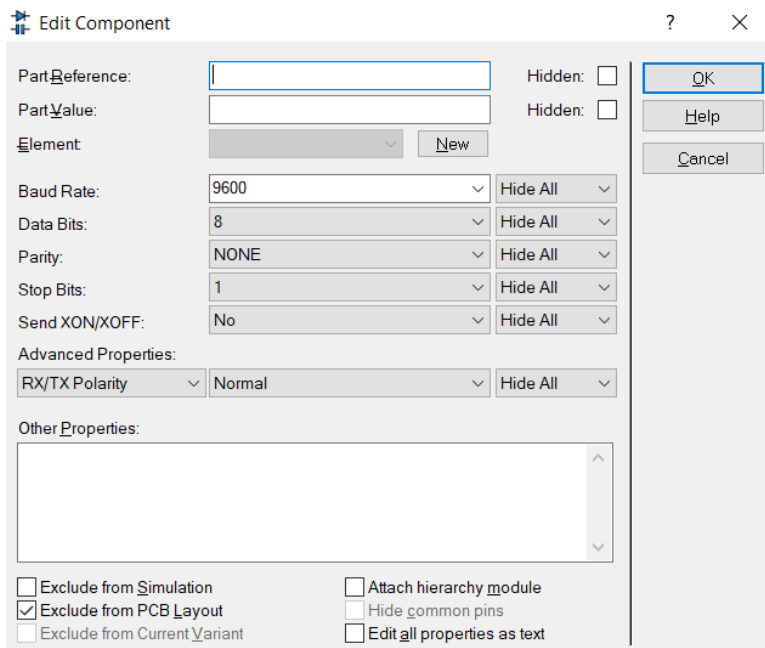
Hình 1.1: Simulation circuit on Proteus

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:



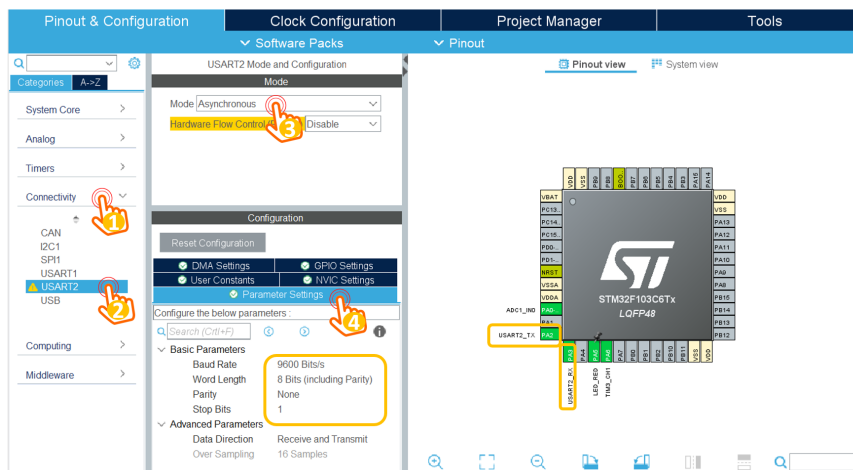
Hình 1.2: Terminal configuration

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:









Hình 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.

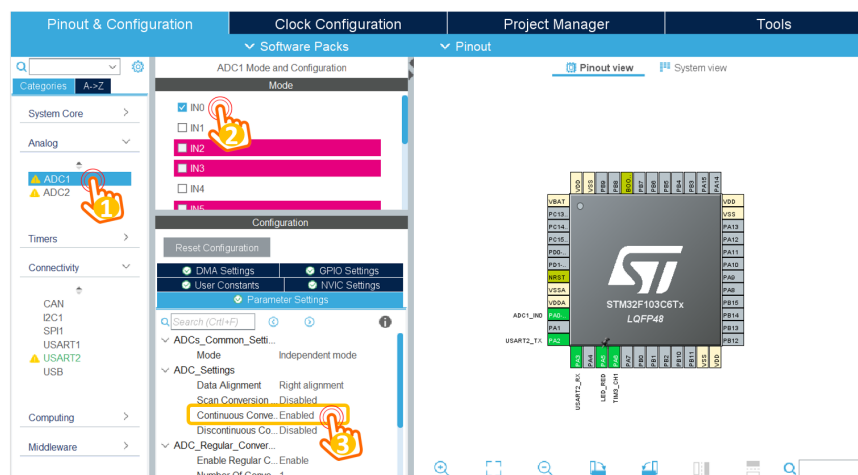
Finally, the NVIC settings are checked to enable the UART interrupt, as follows:

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

Hình 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:



Hình 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);
8     HAL_Delay(500);
9 }

```

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

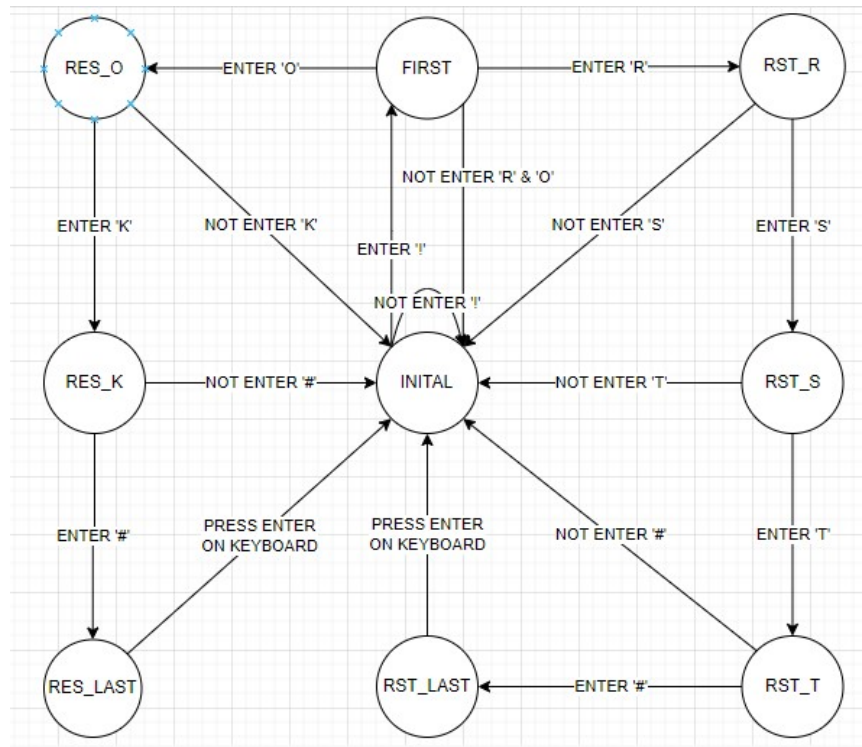
Students are proposed to implement 2 FSM in seperated modules. Students are asked to design the FSM before their implementations in STM32Cube.

6.2.1 Thiết kế máy trạng thái

Hình 1.6 biểu diễn các trạng thái khi người dùng nhập các lệnh !RST# để yêu cầu dữ liệu hay !OK# để xá nhận đã nhận dữ liệu.

Có 9 trạng thái tất cả, các trạng thái này để đánh dấu dòng lệnh có đang đúng cú pháp hay không:

- Các trạng thái RST_R, RST_S, RST_T, RST là trạng thái của Request, còn RES_O, RES_K là trạng thái của Responce. INITAL, FIRST là của chung.
- INITAL: trạng thái khởi tạo, cũng là trạng thái trở về của bất kỳ trạng thái nào khác nếu đủ điều kiện.
- FIRST: nếu đang ở INITAL mà nhấn '!' sẽ chuyển sang FIRST, các trường hợp còn lại sẽ chuyển về INITAL. Từ đây, nếu nhập 'R' sẽ chuyển trạng thái sang RST_R, nếu nhập 'O' sẽ chuyển trạng thái sang RES_O.
- RST_R: nếu đang ở FIRST mà nhấn 'R' sẽ chuyển sang trạng thái này, từ trạng thái này, nhấn 'S' sẽ chuyển sang RST_S
- Các trạng thái tiếp theo có cái nhìn tương tự.
- Khi hệ thống ở trạng thái RST (request) nghĩa là người dùng đã nhập thành công lệnh '!RST#', bây giờ hệ thống có thể gửi dữ liệu cảm biến. Còn nếu ở RES (responce) tức là lệnh '!OK#' là người dùng đã phản hồi việc nhận dữ



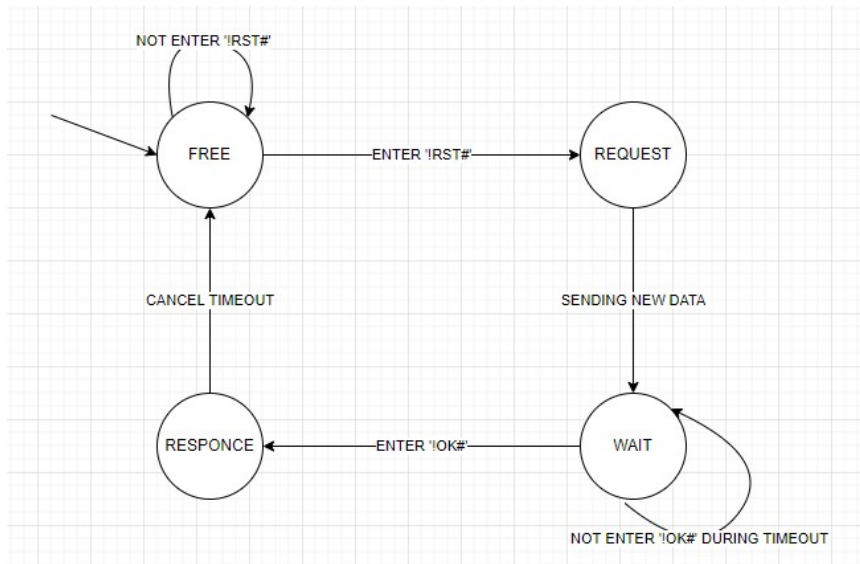
Hình 1.6: Máy trạng thái cho *command_parser_fsm*

liệu thành công. Tuy nhiên máy trạng thái này em thêm một thao tác nữa là nhấn ENTER, khi nhấn ENTER (13) trên bàn phím, hệ thống mới xác nhận đã nhập lệnh và gửi dữ liệu, đồng thời chuyển trạng thái về INITIAL chờ lệnh tiếp theo.

- Ta có thể bỏ qua thao tác nhấn ENTER cũng không ảnh hưởng gì đến hệ thống. Nhưng như vậy mỗi lần người dùng nhập xong sẽ lập tức thực hiện tác vụ. Giả sử có thêm 1 tính năng nữa được thực hiện bởi lệnh '!RS' thì khi người dùng muốn nhập '!RST#' để lấy dữ liệu mà không có thao tác ENTER để xác nhận hoàn thành câu lệnh thì không bao giờ thực hiện được lệnh '!RST#' này.

Hình 1.7 biểu diễn các trạng thái của việc giao tiếp truyền nhận giữ liệu với UART, có 4 trạng thái tất cả:

- FREE: Trạng thái khi không có tác vụ nào cần thực hiện (người dùng không nhập lệnh request hoặc đã nhập lệnh response). Có thể nói là trạng thái CHỜ REQUEST.
- REQUEST: là trạng thái sau khi người dùng nhập lệnh '!RST' và nhấn ENTER, hệ thống chuyển từ FREE sang REQUEST. Tại đây, hệ thống gửi dữ liệu cảm biến đi và chuyển trạng thái sang WAIT chờ người dùng nhập lệnh '!OK#' để xác nhận nhận dữ liệu thành công, thiết lập TIMEOUT chờ phản hồi '!OK#' bằng timer interrupt.
- WAIT: tại đây, nếu nhận được cờ TIMEOUT (người dùng chưa phản hồi đã nhận) thì hệ thống gửi lại giá trị cũ, đồng thời thiết lập lại thời gian chờ phản hồi từ người dùng.



Hình 1.7: Máy trạng thái cho `uart_communiation_fsm`

- RESPONSE: là trạng thái khi ở WAIT, nếu người dùng nhập '!OK#' sẽ chuyển sang RESPONSE. Công việc ở trạng thái này là hủy TIMEOUT đi và chuyển trạng thái về FREE tiếp tục chờ lệnh.

6.2.2 Thực hiện máy trạng thái `command_parser_fsm` và `uart_communiation_fsm`

Như đã nói ở trên, máy trạng thái `command_parser_fsm` có thêm thao tác nhấn ENTER để xác nhận hoàn thành dòng lệnh. Chương trình được thực hiện như sau:

```

1 void command_parser_fsm() {
2     index = (index_buffer - 1 >= 0) ? index_buffer - 1 :
3         MAX_BUFFER_SIZE - 1;
4     switch (stateCommand) {
5     case INITAL:
6         if(buffer[index] == '!') {
7             stateCommand = FIRST;
8         }
9         break;
10    case FIRST:
11        if(buffer[index] == 'R') {
12            stateCommand = RST_R;
13        } else if(buffer[index] == 'O') {
14            stateCommand = RES_0;
15        } else {
16            stateCommand = INITAL;
17        }
18        break;
19    case RST_R:
20        if(buffer[index] == 'S') {
21            stateCommand = RST_S;
22        } else {
23            stateCommand = INITAL;
24        }
25    }
26 }
  
```



```

23     }
24     break;
25 case RST_S:
26     if(buffer[index] == 'T') {
27         stateCommand = RST_T;
28     } else {
29         stateCommand = INITAL;
30     }
31     break;
32 case RST_T:
33     if(buffer[index] == '#') {
34         stateCommand = RST;
35     } else {
36         stateCommand = INITAL;
37     }
38     break;
39 case RES_0:
40     if(buffer[index] == 'K') {
41         stateCommand = RES_K;
42     } else {
43         stateCommand = INITAL;
44     }
45     break;
46 case RES_K:
47     if(buffer[index] == '#') {
48         stateCommand = RES;
49     } else {
50         stateCommand = INITAL;
51     }
52     break;
53 case RST:
54     if(buffer[index] == ENTER) {
55         stateCommand = INITAL;
56         stateRequest = REQUEST;
57     }
58     break;
59 case RES:
60     if(buffer[index] == ENTER) {
61         stateCommand = INITAL;
62         stateRequest = RESPONCE;
63     }
64     break;
65 default: break;
66 }
67 }

```

Program 1.7: Thực hiện `command_parser_fsm`

Đối với `uart_communiation_fsm`, trạng thái được chuyển phụ thuộc vào **command_parser_fsm**.

```

1 void uart_communiation_fsm() {
2     switch (stateRequest) {
3     case FREE:
4         break;
5     case REQUEST:
6         sendData(NOT_RESEND);
7         setTimeout(TIMEOUT);
8         stateRequest = WAIT;
9         break;
10    case WAIT:
11        if(get_timeOut_flag()) {
12            sendData(RESEND);
13            setTimeout(TIMEOUT);
14        }
15        break;
16    case RESPONCE:
17        offTimeOut();
18        stateRequest = FREE;
19        break;
20    default: break;
21    }
22 }

```

Program 1.8: Thực hiện uart_communiation_fsm

Ngoài ra có thêm một hàm phụ nữa để đảm nhận truyền dữ liệu đến người dùng. Hàm đó như sau:

```

1 void sendData(uint8_t is_resend) {
2     if(is_resend == NOT_RESEND) {
3         uint8_t ADC_value = HAL_ADC_GetValue(&hadc1);
4         prev_data = ADC_value;
5     }
6     uint8_t msg[14];
7     uint8_t size = sprintf (msg , "!ADC=%d#\r\n", prev_data);
8     HAL_UART_Transmit (& huart2 , msg , size , 1000);
9 }

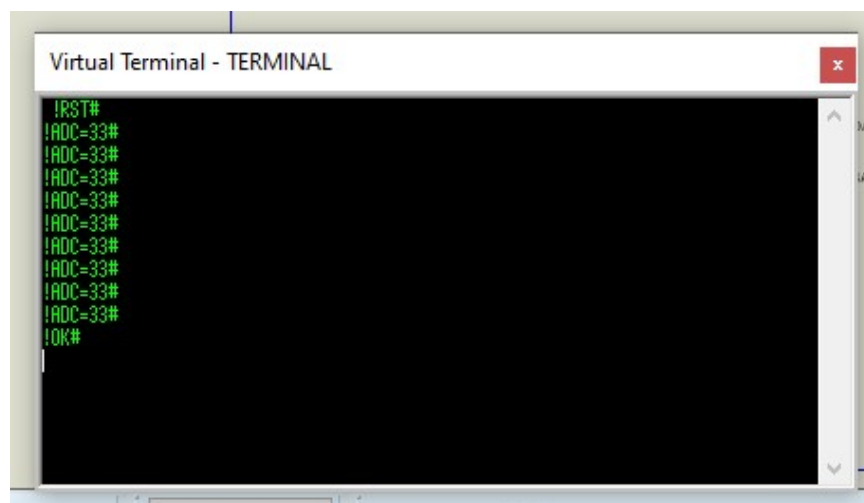
```

6.2.3 Kết quả

Hình 1.8 là kết quả của chương trình đọc giá trị cảm biến độ ẩm mô phỏng bằng Proteus. Để thấy rõ hơn, mời thầy truy cập đường link bên dưới chứa video demo, các file hex và file mô phỏng.

Link video demo: https://drive.google.com/file/d/1pX0Uj1I_Y9azBqBMYPAeaYJD_A687uNm/view?usp=sharing

Link git file mô phỏng: https://github.com/trongthuyen/vxl-vdk-full-lab/tree/main/git_lab5



Hình 1.8: Kết quả mô phỏng bằng Proteus