VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
Faculty of Computer Science and Engineering

CC02 — Lab Report

Microprocessor - Microcontroller
Lab 5

Supervisors: Nguyen Thien An
Students: Nguyen Minh Dang 2252154

Ho Chi Minh City, December 9, 2024

Contents

1 Exercise

The GitHub link for the lab schematics, source code is at here or in this link: https://github.com/dangalpha78/Workspace-for-Microprocessor---Microcontroller/tree/main/Lab5.

The schematic for this lab is located here:

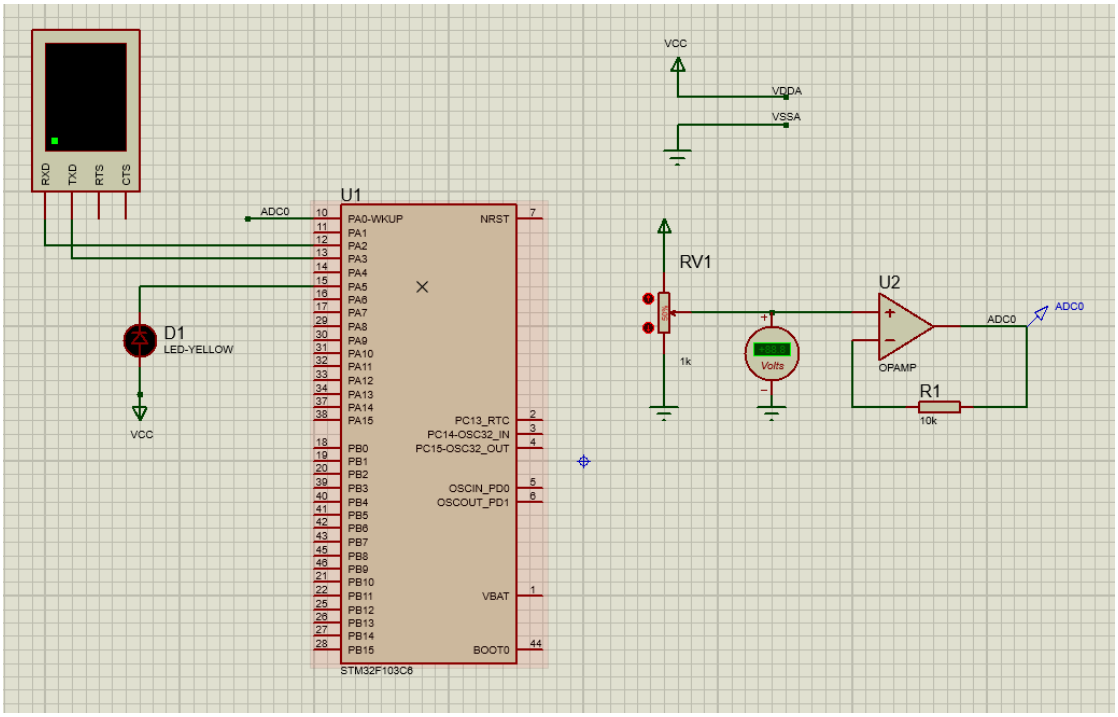

Figure 1: *The schematic.*

- Virtual Terminal:
  - Simulates serial communication via UART, allowing data exchange between the microcontroller and software such as Proteus.
  - Connected to the TXD (transmit) and RXD (receive) pins of the microcontroller.
  - Used for debugging or sending/receiving text commands during simulation.
- ADC and Operational Amplifier:
  - The analog-to-digital converter (ADC) converts the analog voltage from the op-amp into a digital value for processing.
  - The operational amplifier (U2) is configured as a buffer to prevent signal distortion by providing high input impedance and low output impedance.
  - Ensures that accurate voltage levels are fed to the ADC without affecting the source.
- Potentiometer:
  - A variable resistor used to adjust the voltage at its wiper, which serves as the input to the operational amplifier.

– Allows for manual control of the analog signal level fed into the op-amp and ADC.
– Typically used to test or calibrate the input signal range in the circuit.

1.1 Read ADC value

The `read_ADC` function starts the ADC conversion, waits for it to complete, retrieves the value, and stores it in `ADC_value`, then stops the ADC conversion.

```
void read_ADC(void) {
    HAL_ADC_Start(&hadc1);
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
    ADC_value = HAL_ADC_GetValue(&hadc1);
    HAL_ADC_Stop(&hadc1);
}
```

1.2 FSM

1.2.1 FSM for command parser

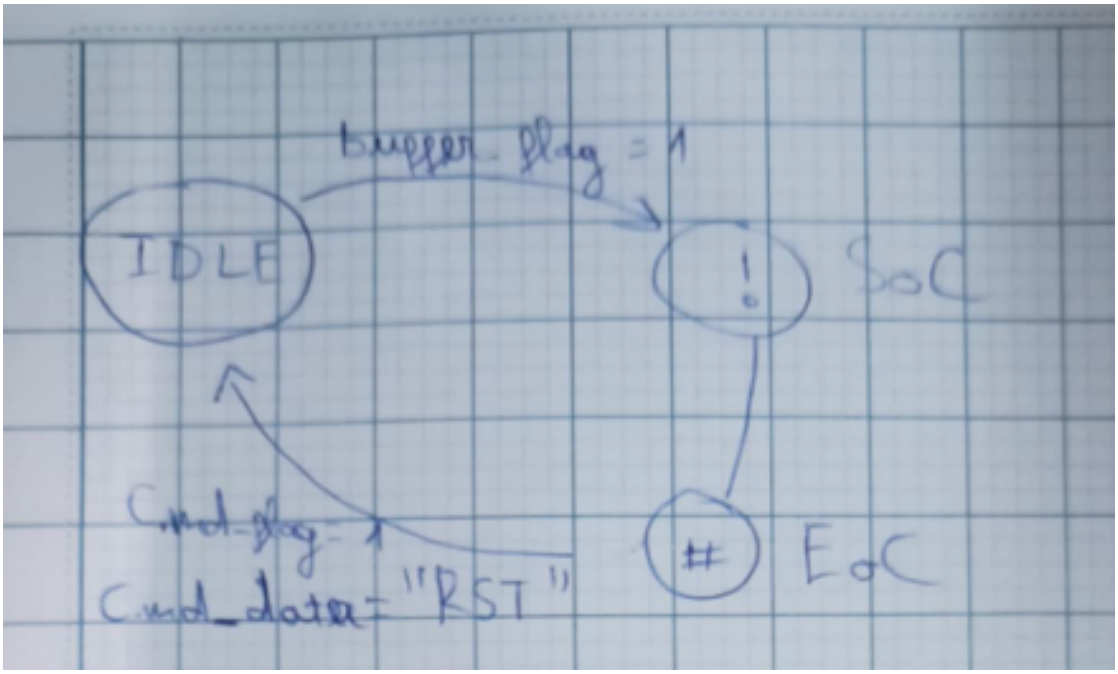

Figure 2: *FSM for command parser.*

The diagram represents a finite-state machine (FSM) for parsing commands entered via a virtual terminal through UART. The input characters are stored in a buffer.

- IDLE State:
  - The system waits for input from the virtual terminal.
  - Transitions to the SoC (Start of Command) state when buffer_flag = 1, indicating that the data is available in the buffer.
- SoC State (!):

- Detects the start of a command sequence.
- Stores the input character into cmd_data, if it is not '#' continue store characters.
- Transitions to the EoC (End of Command) state after receiving the '#' character, signaling the end of the command.

- EoC state (#):
  - Processes the command in the buffer. Also sets cmd_flag on after we get the character '#'.
  - Returns to the IDLE state after processing is complete.

1.2.2 FSM for command status

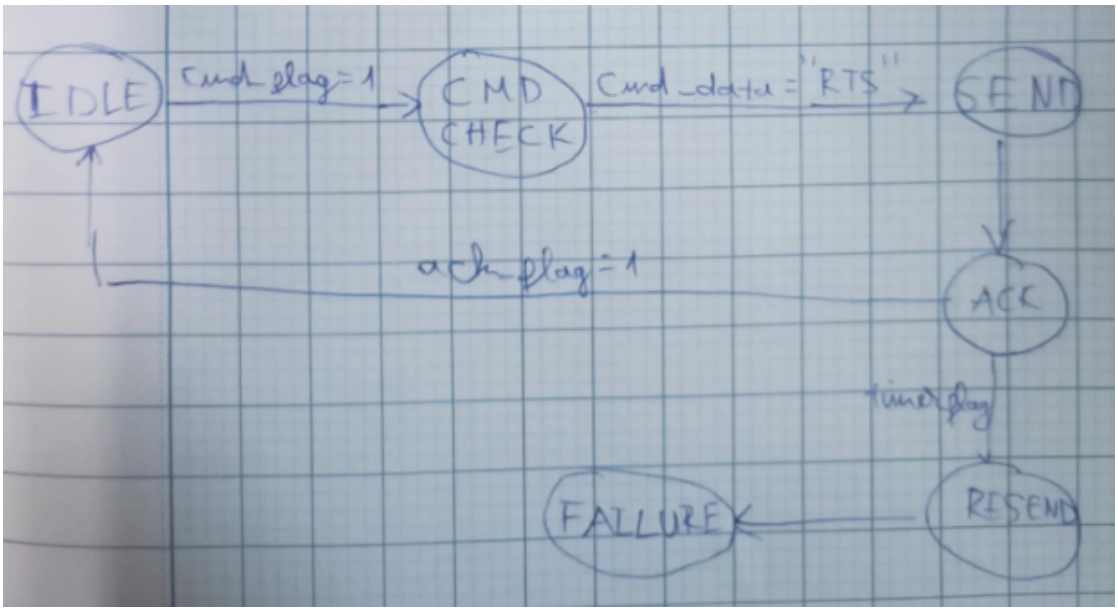

Figure 3: *FSM for command parser.*

The diagram represents a state machine for handling commands received via UART from a virtual terminal. The flow is as follows:

- IDLE State:
  - The system waits for the cmd_flag to be set, indicating a command is available in the buffer.
  - Transitions to the CMD CHECK state if cmd_flag = 1.
- CMD CHECK State:
  - Verifies the content of cmd_data.
  - If the command is valid (e.g., cmd_data = "RTS"), transitions to the SEND state to execute the request.

- SEND State:
  - Executes the requested action and waits for acknowledgment (ack_flag = 1).
  - If acknowledgment is received, transitions to the ACK state.
- ACK State:
  - Indicates successful execution and transitions back to the IDLE state.
- FAILURE State:
  - If no acknowledgment is received within a timeout period, transitions to the RESEND state to retry.
  - May eventually return to IDLE after several retries or report a failure.

1.3 UART Communication

1.3.1 Receive

The buffer stores each entered character sequentially. When a character is entered, it is placed in the next empty position in the `buffer`, and the program continues receiving characters. When the buffer reaches its maximum size (`MAX_BUFFER_SIZE`) or a complete command (`cmd_flag == 1`) is received, the `index_buffer` is reset to 0 to start receiving characters for a new command. Simultaneously, `buffer_flag` is set to 1 to indicate that a character has been entered.

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if (huart->Instance == USART2) {
        buffer[index_buffer++] = temp;

        if (index_buffer == MAX_BUFFER_SIZE || cmd_flag == 1) {
            //buffer[index_buffer] = '\0';
            index_buffer = 0;
        }

        buffer_flag = 1;

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

1.3.2 Transmit

When the `cmd_action` function is called, it transmits the value of `ADC_value` (the ADC result) to the virtual terminal using the UART communication protocol. The message is formatted as `!ADC=<value>#`. After sending the data, the function waits for 3 seconds (`HAL_Delay(3000)`) before proceeding to the next action.

```
void cmd_action(void){
    HAL_UART_Transmit(&huart2, (char*)str, sprintf(str, "!ADC=%d#", ADC_value),
    1000);
    HAL_Delay(3000);
}
```

1.4 Command processing

1.4.1 Command parser

The `command_parser_fsm` function is a finite state machine (FSM) that processes incoming characters from the buffer. It transitions between two states: SOC (Start of Command) and EOC (End of Command). When the SOC state is active and the received character is `!`, it indicates the start of a command, and the function prepares for command processing. When the EOC state is active and the received character is `#`, it marks the end of the command. The command data is stored in the `cmd_data` array, and a flag `cmd_flag` is set to signal that a command is complete.

```
void command_parser_fsm(void) {
    uint8_t received_char = buffer[index_buffer - 1];
    switch (current_read_state){
        case SOC:
            if (received_char == '!') {
                auto_action_flag = 0;
                current_read_state = EOC;
                cmd_index = 0;
                HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
            }
            break;

        case EOC:
            if (received_char == '#'){
            cmd_data[cmd_index] = '\0';
            current_read_state = SOC;
            cmd_flag = 1;
                }
            else if (cmd_index < MAX_CMD_LENGTH - 1){
            cmd_data[cmd_index++] = received_char;
            }
            break;

        default:
            current_read_state = SOC;
            break;
    }
}
```

1.4.2 Command status

The `uart_communication_fsm` function is a finite state machine (FSM) that handles different communication states. It starts in the `AUTO` state, where it checks if the `cmd_flag` is set to 1, which transitions the FSM to the `CMD_CHECK` state. If the `auto_action_flag` is set to 1, the FSM transitions to the `SEND` state. In the `CMD_CHECK` state, the function checks if the `cmd_data`

matches predefined commands (such as `"stopSen"` or `"askSen"`). If the `"stopSen"` command is received, it toggles a red LED and returns to the `AUTO` state. If `"askSen"` is received, it triggers an ADC read and transitions to the `SEND` state. After processing the command, the FSM returns to the `AUTO` state. The `SEND` state performs the command action via the `cmd_action` function and then transitions back to `AUTO`.

```
void uart_communiation_fsm(void){
    switch (current_state) {
    case AUTO:
        if (cmd_flag == 1) {
            current_state = CMD_CHECK;
            cmd_flag = 0;
        }
        else if (auto_action_flag == 1){
          current_state = SEND;
        }
        break;

    case CMD_CHECK:
        if (strncmp((char *)cmd_data, stopSen, 2) == 0) {
            current_state = AUTO;
            auto_action_flag = 0;
            HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
        }
        else if (strncmp((char *)cmd_data, askSen, 3) == 0) {
            read_ADC();
            current_state = SEND;
            auto_action_flag = 1;
        }
        current_state = AUTO;
        break;

    case SEND:
        cmd_action();
        current_state = AUTO;
        break;

    default:
        current_state = AUTO;
        break;
    }
}
```

1.5 Variables and main()

1.5.1 Variables

```
typedef enum {
    IDLE,
    SOC,
    EOC
} CommandRead;
CommandRead current_read_state = IDLE;

static enum {
    AUTO,
    CMD_CHECK,
    SEND,
    ACK,
    FAILURE,
    RESEND
} current_state = AUTO;
#define MAX_BUFFER_SIZE 30
const char askSen[3] = "RST";
const char stopSen[2] = "OK";
uint8_t temp = 0;
uint8_t buffer[MAX_BUFFER_SIZE];
uint8_t index_buffer = 0;
uint8_t buffer_flag = 1;

#define MAX_CMD_LENGTH 10

uint8_t cmd_data[MAX_CMD_LENGTH];
uint8_t cmd_flag = 0;
static uint8_t cmd_index = 0;

#define MAX_RETRIES 3
uint8_t auto_action_flag = -1;
```

1.5.2 main()

```
HAL_UART_RxCpltCallback(&huart2);
HAL_TIM_Base_Start_IT(&htim2);
uint32_t ADC_value = 0;
uint8_t str[20];
while (1)
{
    /* USER CODE END WHILE */
    if (buffer_flag == 1) {
      auto_action_flag = 0;
      command_parser_fsm();
      buffer_flag = 0;
```

```
12      }
13      uart_communiation_fsm();
14      /* USER CODE BEGIN 3 */
15  }
```

2 Problem about this Lab

In this lab, I encountered a problem where the `HAL_Delay(3000)` in the `cmd_action()` function causes a 3-second delay, during which no actions are recognized. As a result, any input entered during this period is not recorded, or it is only recognized after 3 seconds. For example, when entering `!OK#` too fast in 3 seconds, the system will only recognize the `#` character, which prevents the ADC value from stopping as expected.

I thought of a solution by combining it with a timer interrupt, as in previous labs, but due to time constraints, I was unable to complete it.

References