VIETNAM NATIONAL UNIVERSITY
HO CHI MINH UNIVERSITY OF TECHNOLOGY
COMPUTER SCIENCE & ENGINEERING FACULTY

MICROCONTROLLER MICROPROCESSOR (CO3010)

Lab Report

Lab5

FLOW AND ERROR CONTROL IN COMMUNICATION

Teacher: Huynh Phuc Nghi
Student: Nguyen Thanh Hien (2111203)

GitHub: Lab 5

HO CHI MINH CITY, DECEMBER 2023

Contents

List of Figures

List of Tables

1 REQUIREMENT ANALYSIS

1.1 General requirement

1. Construct a system reading sensory data, and collecting ADC data.

2. Implement a simple communication protocol, which:

   - Request sensory data: user types **!RST#** via the console.
   - Then, the STM32 would transmits the **ADC value**, following a format **!ADC=xxxx** (*xxxx* is the ADC value, in the range 0 - 4096, 13 bits). After every 3 seconds, the same data is retransmitted if the user didn't ask to close the communication line.
   - Cancel request sensory data: user types **!OK#** via the console.

3. Implement 2 FSM in separate modules: `command_parser_fsm()` and `uart_communication_fsm()`.

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

Program 1: Program structure need to implement

1.2 Finite state machine

1.2.1 Command parser

The **command parser** operates separately and reacts with every keyboard trigger. We have two command keywords, which are:

- **!RST#** - *Request to transmit ADC value via UART to the virtual terminal*: Always detect and treat it as a request for the ADC value at that current timestamp.

- **!OK#** - *Stop requesting and no longer sending data*: Detect only when user still requesting for ADC value.

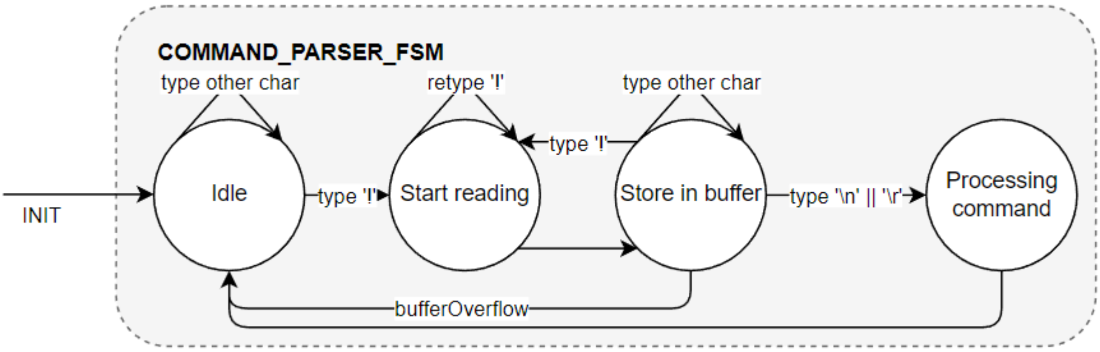


Figure 1: FSM: Command parser

1.2.2 UART communication

UART communication is used for:

- **Receive (RXD)** - *User keyboard input via console*: The key can be stored in a buffer for the command parser to process.
- **Tranfer (TXD)** - *ADC value to virtual terminal*: When communication line is established (user typed **!RST#**), the ADC value is sent.

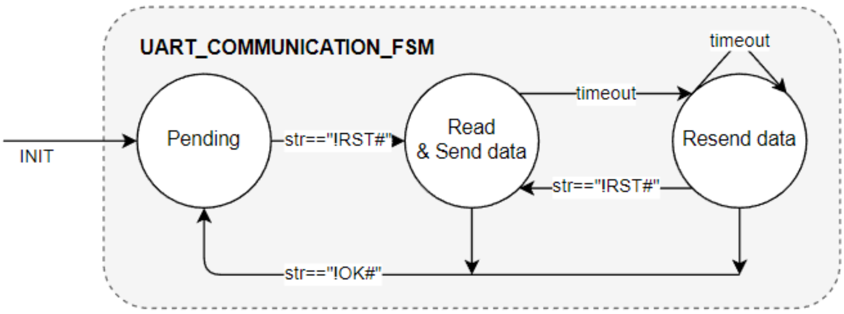
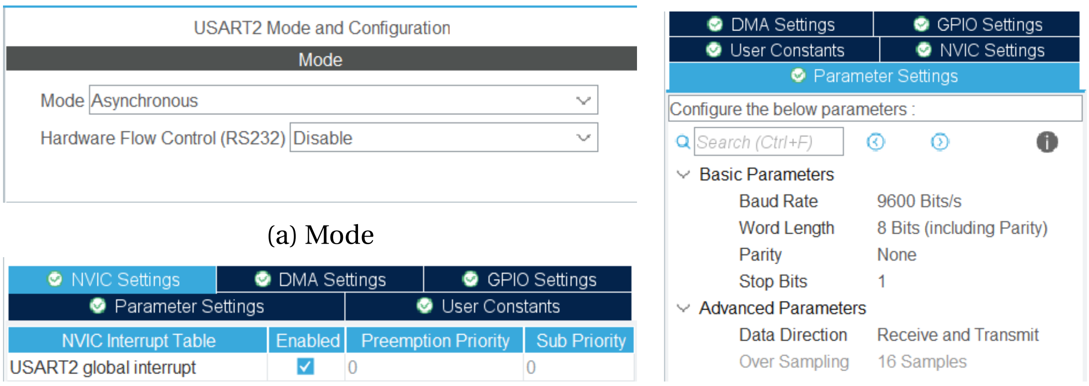


Figure 2: FSM: UART Communication

2 SYSTEM CONFIGURATION

2.1 Microcontroller configuration

2.1.1 USART configuration



(a) Mode

(b) NVIC Settings

(c) Parameter Settings

Figure 3: USART2 configuration

The UART is required to be in **Asynchronous** mode, with its **Global interrupt = Enabled** in NVIC settings.

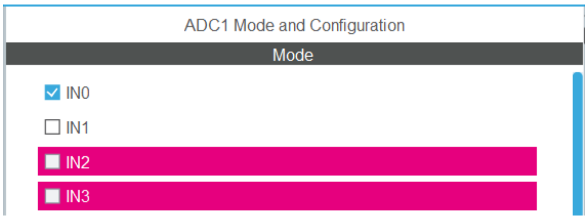
The parameter must be aligned with the settings in Proteus **virtual terminal** since both must have the same settings to establish UART communication successfully.

- Baud Rate: **9600 Bits/s**
- Word Length: **8 Bits**
- Parity: **None**
- Stop bits: **1**

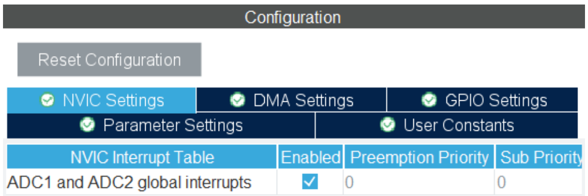
We then would have the pin configuration as:

- **UASRT_TX**: PA2, to transmit ADC value to virtual terminal.
- **UASRT_RX**: PA3, to receive user keyboard input via the console.

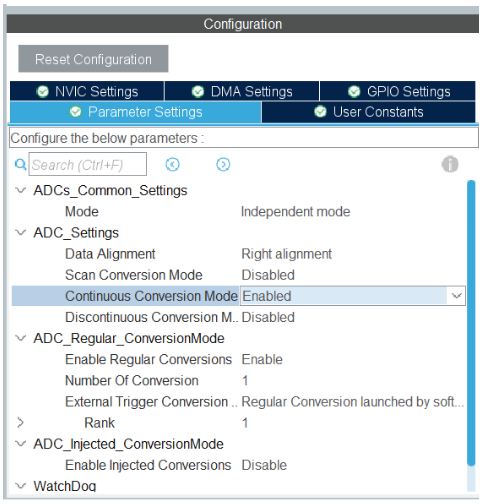
2.1.2 ADC configuration



(a) Mode



(b) NVIC Settings



(c) Parameter Settings

Figure 4: ADC1 configuration

We use **ADC1**, with **IN0** mode and the c**Continuous Coversion Mode = Enabled**. The corresponding pin is **PA0**.

2.1.3 TIMER configuration

Since I decided to implement a time-based system, which would have an interrupt period of $T_{TICK} = 10(ms)$

- Timer: 2
- Clock Source: Internal clock (8MHz)
- Prescaler: 1
- Counter Mode: Up
- Counter Period: 39999

We arrived with that conclusion based on the following calculation:

$$f_{TIM} = \frac{1}{T_{TICK}} = \frac{1}{10.10^{-3}} = 100(Hz) \tag{1}$$

$$\text{Auto Reload Register: } ARR_{before} = \frac{f_{TICK}}{f_{TICK} - 1} = \frac{8.10^{6}}{100} - 1 = 79999\text{, exceeded 16-bit.} \tag{2}$$

$$\text{Prescaler:} PSC = \frac{ARR}{Max_{16bit}} = \frac{79999}{65535} = 1 \tag{3}$$

$$ARR_{after} = \frac{f_{TIM}}{f_{TICK}.(PSC+1)} - 1 = \frac{8.10^6}{100.(1+1)-1} = 39999 \tag{4}$$

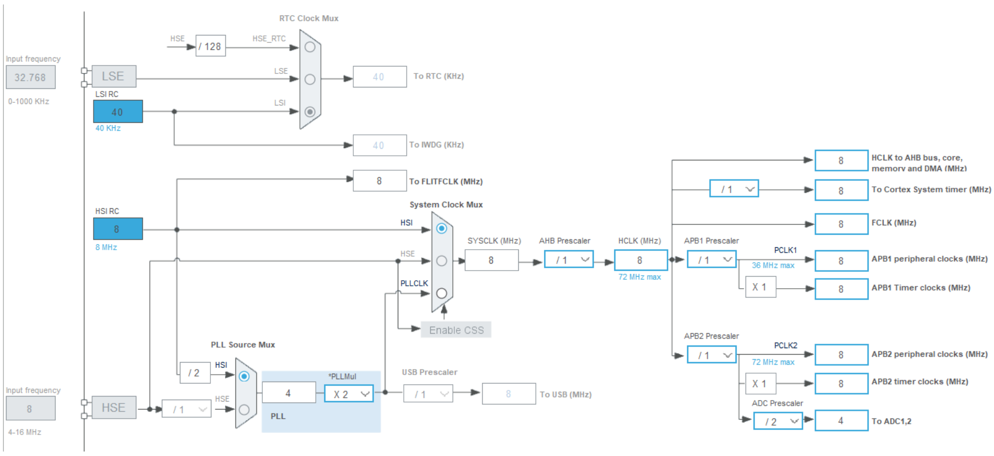
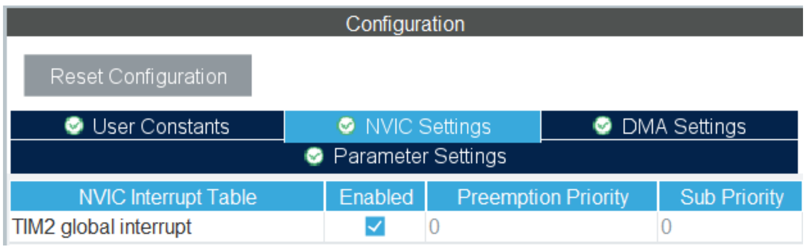
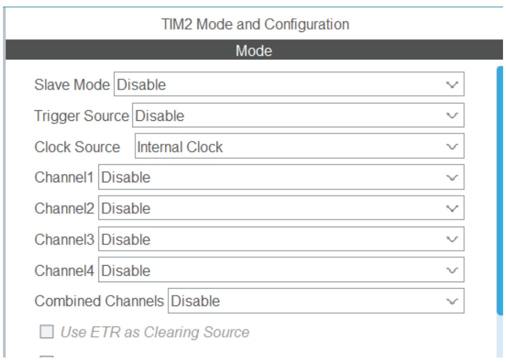


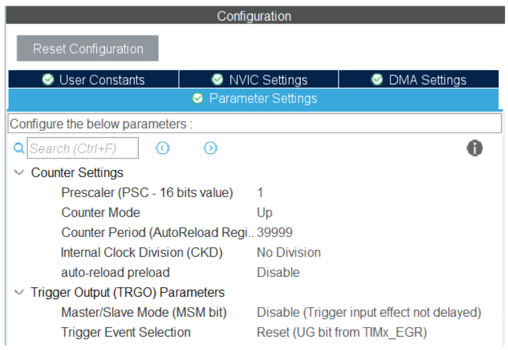
Figure 5: Clock configuration



(a) NVIC Settings



(b) Mode

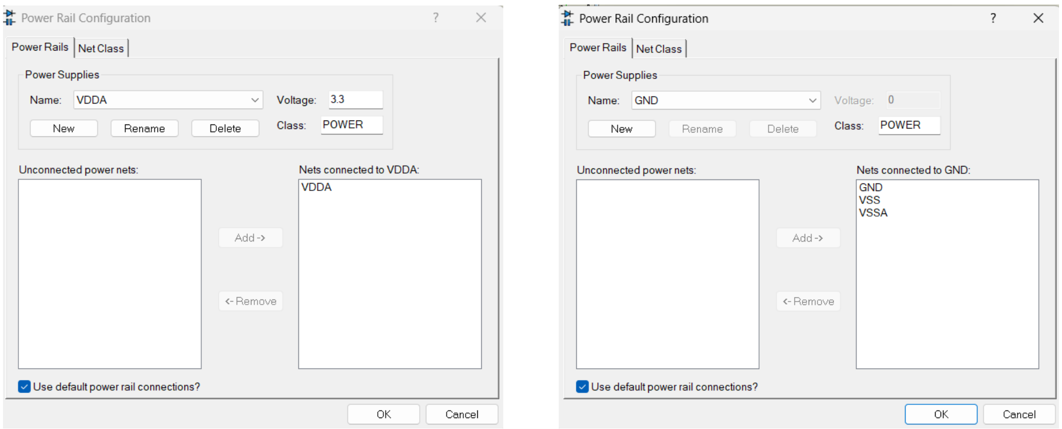


(c) Parameter Settings

Figure 6: Timer2 configuration

2.2 Schematic configuration

2.2.1 Power rail configuration



(a) VDDA (b) VSSA

Figure 7: Power rail configuration

With the power rails, we configured the power source as **VDDA = 3.3V** and the reference point as **VSSA = GND**.

2.2.2 Virtual terminal

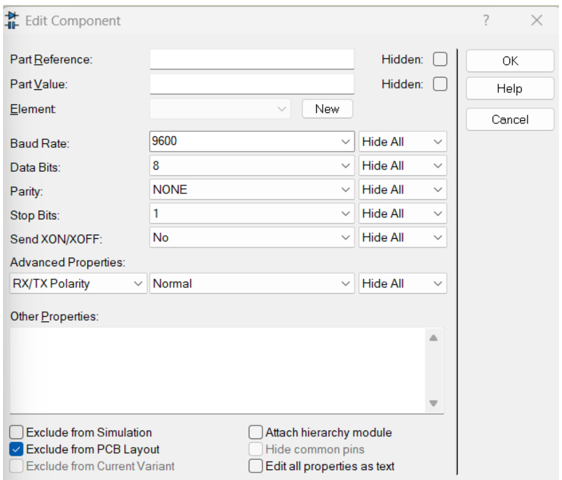


Figure 8: Edit component: Virtual terminal

The virtual terminal configuration is similar to UART configuration in the microcontroller mentioned above.

2.2.3 Schematic setup

STM32 Pin	Component connected
PA0	Opamp (POT-HG)ab
PA2	Virtual terminal RX
PA3	Virtual terminal TX
PA5	LED_RED

Table 1: Schematic pin configuration

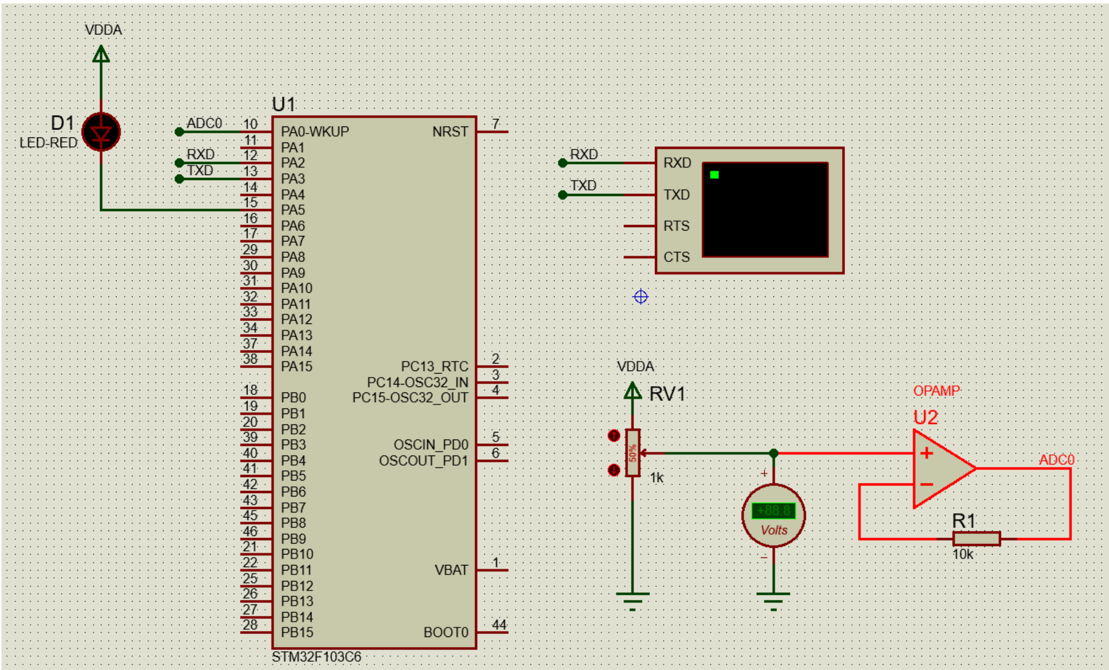


Figure 9: Schematic

3 PROJECT IMPLEMENTATION

3.1 Overall structure

```
HAL_UART_Receive_IT(&huart2, &parserByte, 1);
while (1)
{
    if(systemTickFlag == 1){
        HAL_GPIO_TogglePin(LED_RED_GPIO_Port,
   LED_RED_Pin);
        systemTickFlag = 0;
    }
    if(rxFlag == 1){
        fsmCommandParser();
        rxFlag = 0;
    }
    fsmUARTCommunication();
}
```

Program 2: Implement project in **main.c**

In the while loop, we have:

- 1 system tick **(every 1s)**: to simulate the system clock to verify the system time-based operation. The indicator LED would toggle independently after every 1s.
- 2 finite state machine for:
  1. **Command parser:** reads each character user input to respond.
  2. **UART communication:** implement the receiving and transmitting mechanism of the system as described above.

```
void fsmCommandParser(){
  if(rxFlag  == 1){
    switch(PARSER_FSM){
    case PARSER_IDLE:
      parserIdle();
      break;
    case PARSER_READING:
      parserReading();
      break;
```

```
    }
    rxFlag = 0;
  }
}

void fsmUARTCommunication(){
  switch(UART_FSM){
  case UART_WAIT_FOR_COMMAND:
    uartWaitForCommand();
    break;
  case UART_WAIT_FOR_ACK:
    uartWaitForCommand();
    uartWaitForAck();
    break;
  }
}
```

Program 3: 2 FSM structure in **fsm.c**

3.2 ADC reading

```
int ADC_value = 0;

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

Program 4: Read new ADC data

This function is called every time the user types in **!RST#** to read and update the ADC value, regardless of the state that the system is in.

3.3 Timer interrupt

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *
   htim){
  if(htim->Instance == htim2.Instance){
    if(timeoutFlag == TIMER_IDLE){
      timerReset();
```

```
    } else if(timeoutFlag == TIMER_COUNTING){
      resendCounter++;
      if(resendCounter >= RESEND_TIME_OUT_DURATION){
        timeoutFlag = TIMER_EXCEED;
        resendCounter = 0;
      }
    }
    if(systemTickCounter >= TICK_DURATION){
      systemTickFlag = 1;
      systemTickCounter = 0;
    } else{
      systemTickCounter++;
    }
  }
}
```

The timer interrupt is used for 2 purposes, which are to:

1. Implement transmitting timeout mechanism to resend the ADC value, once the user has entered the requesting session (**!RST#**) and hasn't decided to close the communication line (**!OK#**) yet.

2. Keeping track of the second tick, which would interrupt every 1s to toggle the indicator LED.

3.4 Command parser

3.4.1 Idle state

```
void parserIdle(){
  // Clear buffer
  parserIndex = 0;
  parserBuffer[parserIndex] = '\0';
  // Start reading if a command begins
  if(parserByte == '!'){
    parserBuffer[parserIndex++] = parserByte;
    PARSER_FSM = PARSER_READING;
  }
}
```

Program 5: Parser Idle state

Whilst typing on the console, if the user hasn't typed the **start signal** (character "!") the system wouldn't store the input in the buffer but simply read it. Once the keyword is entered, the system will:

1. Store the keyword at the beginning of the buffer and increase the indexing cursor by one.
2. Set the next state = **Reading state**.

3.4.2 Reading state

```
void parserReading(){
  if(parserByte == '!'){// Read as new command
    parserIndex = 0;
    parserBuffer[parserIndex++] = parserByte;
  }
  else if(parserByte == '\b'){//Delete character
    parserBuffer[parserIndex--] = '\0';
    if(parserIndex < 0){
      PARSER_FSM = PARSER_IDLE;
    }
  }
  else if(parserByte == '\r' || parserByte == '\n'){//
 Process input
    parserBuffer[parserIndex] = '\0';
    if(strcmp(parserBuffer, "!OK#") == MATCHED){
      uartFlag = FLAG_EXIT_REQUEST_ADC;
    }
    if(strcmp(parserBuffer, "!RST#") == MATCHED){
      uartFlag = FLAG_REQUEST_ADC;
    }
    PARSER_FSM = PARSER_IDLE;
  }
  else{// Store data
    parserBuffer[parserIndex++] = parserByte;
  }
  if(parserIndex >= PARSER_BUFFER_SIZE){ // Check buffer
    overflow
    PARSER_FSM = PARSER_IDLE;
  }
}
```

Program 6: Parser while reading and processing command

As complicated as it may seem, the mechanism can be broken down as a reaction to user keyboard input once have entered the **Reading state**:

- **Read as a new command**: Flush the buffer and read the following characters as a new command, once the user retyped the starting keyword **"!"**.
- **Delete a typed character:** Delete a character from the buffer if the user typed the **Backspace** character. In addition, we need to consider the case of deleting the entire buffer, which would reset the state = **Idle state**.
- **Process data:** The system use either the **new line** *(\n)* character or the **carriage return** *(\r)* as a signal to determined the typed string was: **!RST#** or **!OK#**. Once completed processing, the parser would reset to state = **Idle state**.
- **Simply storing:** Continuously store read character to buffer, then detect error once the buffer overflows, which indicated malfunction and needs to reset entire state = **Idle state**. The required command (!RST# and !OK#) wouldn't be able to exceed the buffer by all means.

3.5 UART communication

3.5.1 UART interrupt service routine

```
void  HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
  {
  if(huart->Instance == USART2){
     HAL_UART_Receive_IT(&huart2, &parserByte, 1);
     uint8_t temp = parserByte;
     HAL_UART_Transmit(&huart2, &temp, sizeof(temp), 10)
  ;
     rxFlag = 1;
  }
}
```

Program 7: UART Interrupt

This interrupt is mainly used to implement:

- Receiving each user keyboard character input and raise a flag to notify the system.
- Transmiting the typed character to the screen (loopback).

3.5.2 Wait for CMD

```
void uartWaitForCommand(){
  if(uartFlag == FLAG_REQUEST_ADC){//REQUESTED
    uartFlag = FLAG_NONE;
    timeoutFlag = TIMER_COUNTING;
    ADCRead();
    uartSendReponse();
    UART_FSM = UART_WAIT_FOR_ACK;
  }
}
```

3.5.3 Wait for ACK

```
void uartWaitForAck(){
  if(uartFlag == FLAG_EXIT_REQUEST_ADC){//ACKED
    uartFlag = FLAG_NONE;
    timeoutFlag = TIMER_IDLE;
    UART_FSM = UART_WAIT_FOR_COMMAND;
  } else if(timeoutFlag == TIMER_EXCEED){
    timerReset();
    timeoutFlag = TIMER_COUNTING;
    uartSendReponse();
  }
}
```

4 SIMULATION RESULT

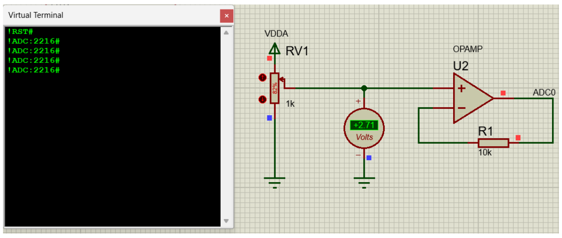


Figure 10: Request for ADC value

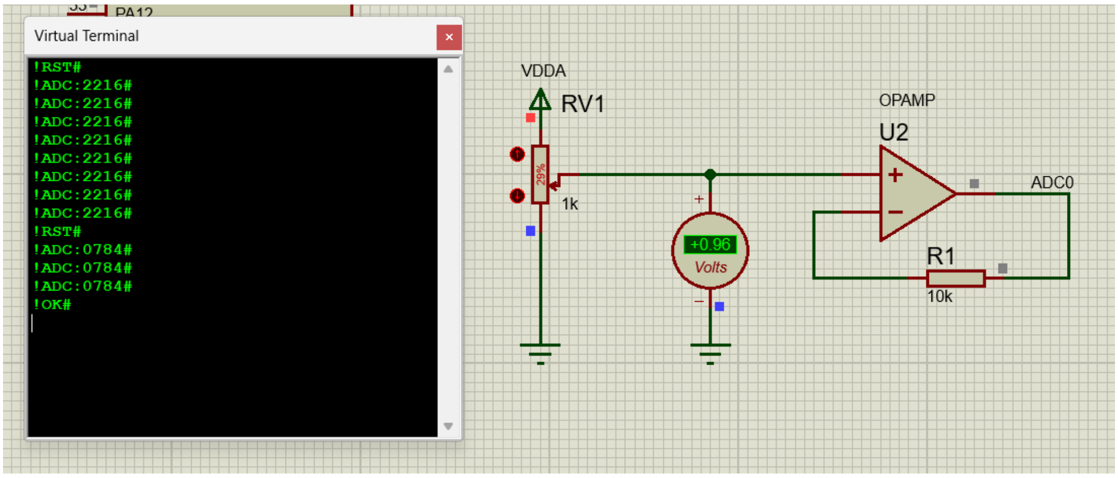


Figure 11: Request the ADC value again at a different value and close the line