ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

BÁO CÁO THỰC HÀNH

VI XỬ LÝ - VI ĐIỀU KHIỂN

LAB 5: FLOW AND ERROR CONTROL IN COMMUNICATION

Lớp – Nhóm: L03 – L05
Giảng viên hướng dẫn: Lê Trọng Nhân
Cao Tiến Đạt
Sinh viên thực hiện: Ngô Quang Tùng 2213869

Thành phố Hồ Chí Minh, 12/2024

Mục lục

Chương 5

Flow and Error Control in Communication

Link GitHub

5.1 Đặc tả yêu cầu

5.1.1 Yêu cầu chung

- Hiện thực một giao thức giao tiếp đơn giản thực hiện các nhiệm vụ:
  – Người dùng nhập !RST# để yêu cầu dữ liệu từ cảm biến.
  – Hệ thống điều khiển cảm biến đọc dữ liệu.
  – Phản hồi bằng dữ liệu thu được theo cú pháp !ADC=x#, trong đó x là dữ liệu thu được từ cảm biến. Tác vụ phản hồi được lặp lại sau mỗi 3 giây.
  – Nếu người dùng nhập OK#, kết thúc quá trình phản hồi.

5.1.2 Các ràng buộc bổ sung

- Để làm rõ quá trình giao tiếp, một số ràng buộc được bổ sung gồm:
  – Nếu người dùng nhập !OK# khi chưa có yêu cầu !RST#, câu lệnh sẽ bị bỏ qua.
  – Nếu người dùng nhập các lệnh !RST# liên tục nhau, hệ thống ngừng thực hiện lệnh !RST# cũ và yêu cầu dữ liệu mới từ cảm biến. Khi đó, giá trị phản hồi sẽ là giá trị mới nhất thu được từ cảm biến.

- Ngoài ra, hoàn thiện thêm và sử dụng bộ định thời (scheduler) được hiện thực ở Bài thực hành 4 để gọi thực thi các tác vụ.

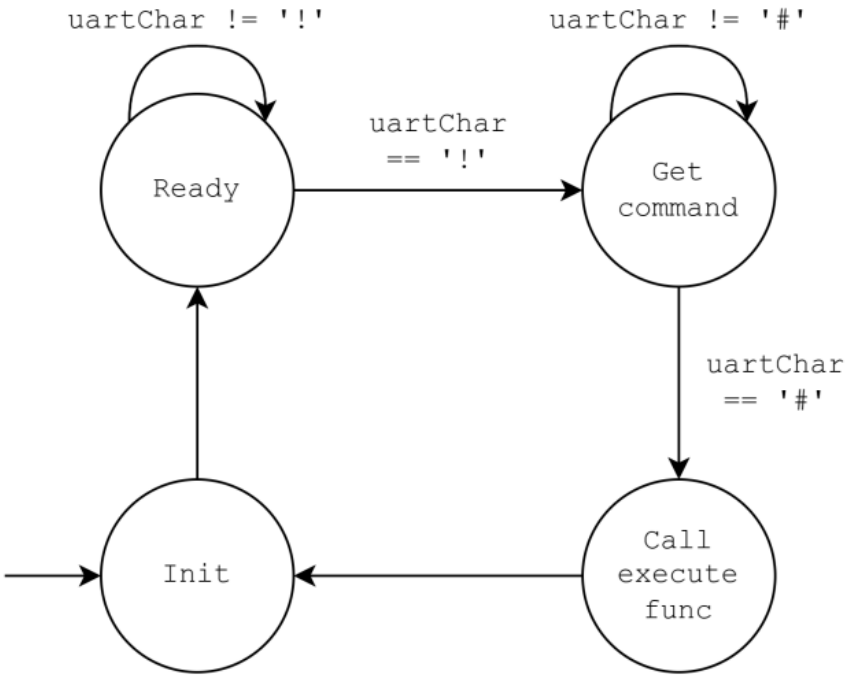
5.2 Thiết kế

5.2.1 Máy trạng thái

Trình phân tích lệnh

- Trạng thái khởi tạo (Init) thực hiện khởi tạo các giá trị cần thiết và tự chuyển sang trạng thái tiếp theo sau khi hoàn tất.

- Trạng thái sẵn sàng (Ready) chờ người dùng nhập ký tự khởi đầu !, các ký tự khác sẽ không được chấp nhận và trình phân tích lệnh bỏ qua.
- Trạng thái nhận lệnh (Get command) được bắt đầu sau khi ký tự khởi đầu được xác nhận. Trình phân tích lệnh nhận lệnh từ người dùng và lưu vào bộ đệm của mình. Trạng thái nhận lệnh sẽ được duy trì cho đến khi người dùng nhập ký tự kết thúc #.
- Trạng thái gọi lệnh thực thi (Call execute func) thực hiện gọi thực thi lệnh đã được xác nhận. Sau khi gọi lệnh thực thi, trình phân tích lệnh trở lại trạng thái khởi tạo để bắt đầu lại quy trình của mình mà không cần chờ việc thực thi lệnh hoàn tất.

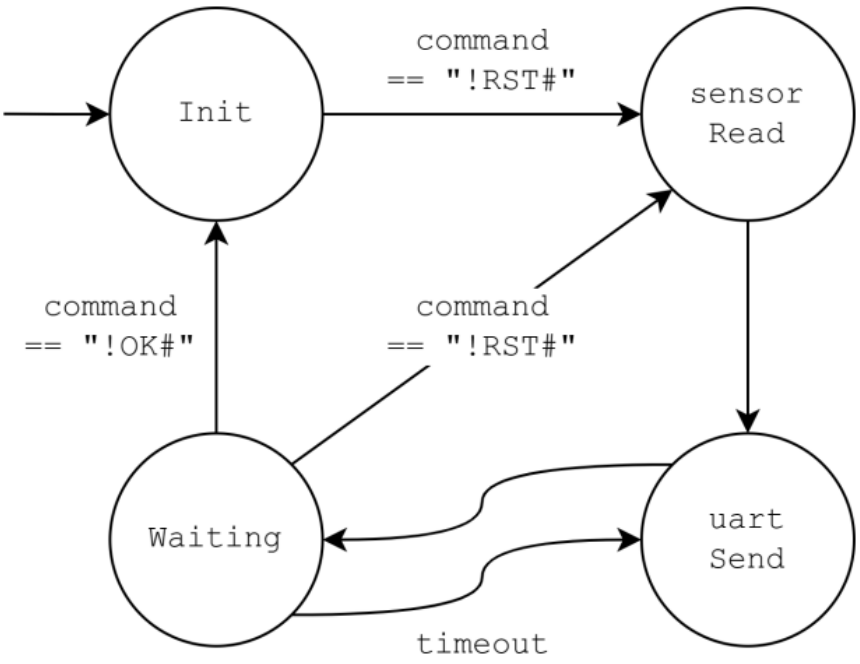


Hình 5.1: Máy trạng thái: Trình phân tích lệnh

Trình thực thi lệnh

- Trạng thái khởi tạo (Init) kiểm tra và đối chiếu lệnh, nếu lệnh được gọi là !RST#, trình thực thi lệnh chuyển sang trạng thái tiếp theo. Ngược lại, lệnh sẽ bị bỏ qua.
- Trạng thái nhận dữ liệu (sensorRead) gọi thực thi tác vụ nhận dữ liệu từ cảm biến. Sau khi có dữ liệu phù hợp, chuyển sang trạng thái tiếp theo.
- Trạng thái phản hồi (uartSend) gọi thực thi tác vụ truyền dữ liệu ra màn hình qua giao thức UART theo cú pháp !ADC=x#, trong đó x là dữ liệu thu được từ cảm biến.
- Trạng thái đợi (Waiting) thực hiện chờ phản hồi từ người dùng. Nếu hết thời gian đợi (timeout), phản hồi sẽ được lặp lại với bộ giá trị cũ. Trong khoảng thời gian này, nếu trình thực thi lệnh nhận được !OK#, quá trình phản hồi sẽ được ngừng lại.

- Ngoài ra, nếu trong trạng thái đợi, lệnh !RST# được gọi, quá trình phản hồi đang thực hiện sẽ bị hủy và trình phân tích lệnh trở lại trạng thái nhận dữ liệu để phản hồi với người dùng bằng dữ liệu mới nhất.

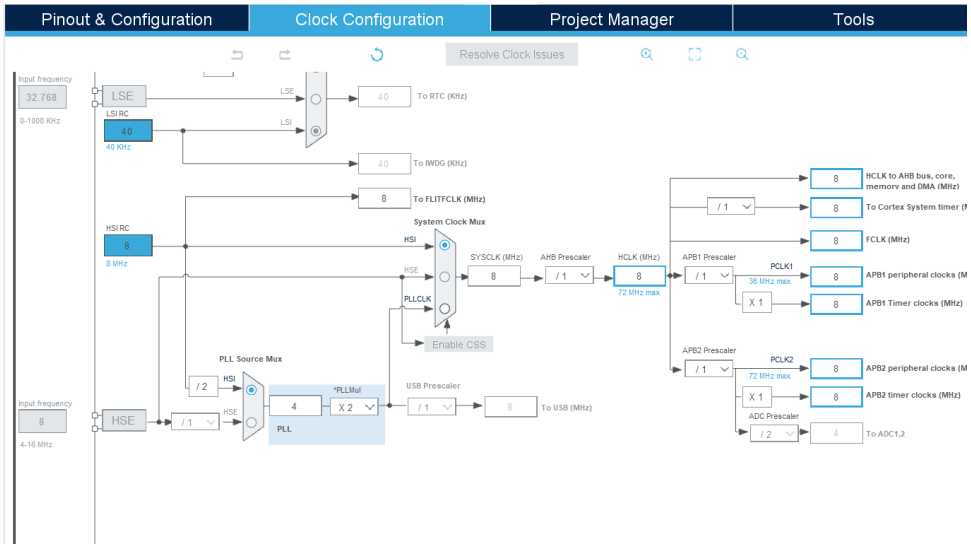


Hình 5.2: Máy trạng thái: Trình thực thi lệnh

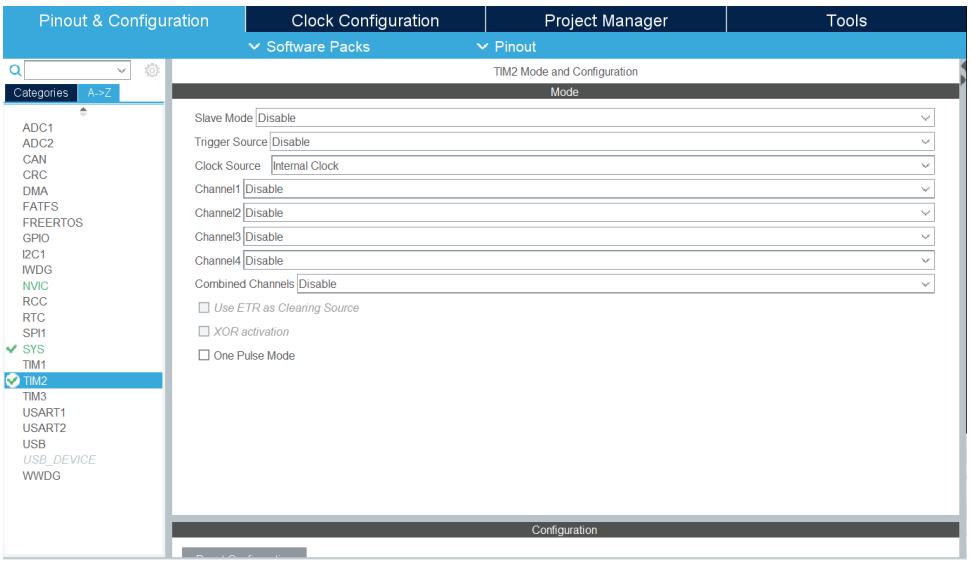
5.2.2 Cấu hình vi điều khiển

Cấu hình timer

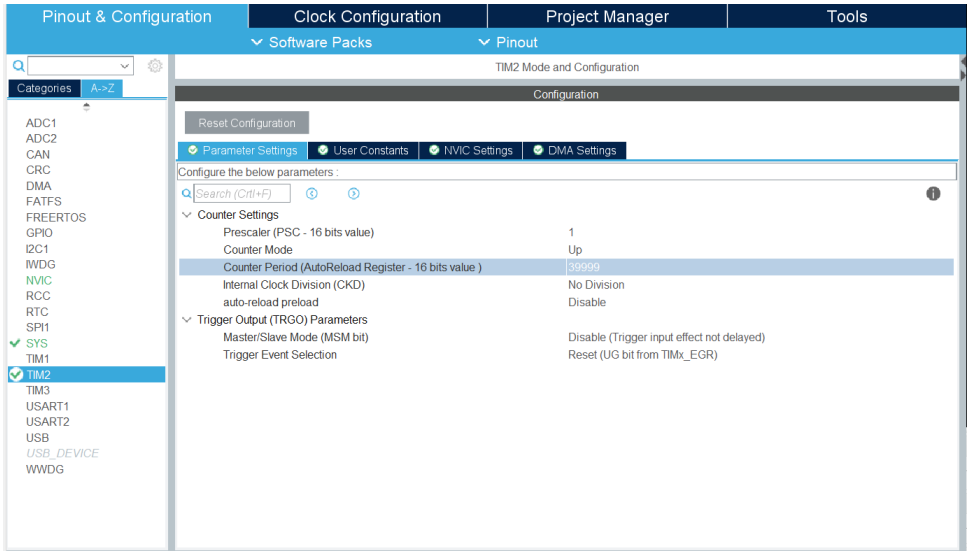
- Chọn sử dụng TIM2.
- Đặt giá trị tần số timer bằng 8MHz (Hình 5.3).
- Chọn Internal Clock làm Clock Source của TIM2 (Hình 5.4).



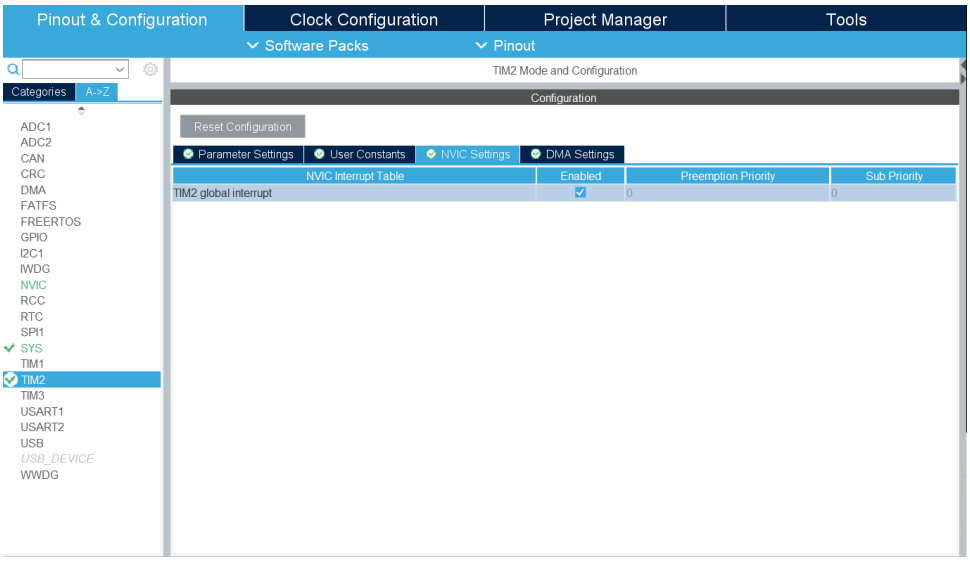
Hình 5.3: Cấu hình timer: Clock Configuration



Hình 5.4: Cấu hình timer: TIM2 NVIC Settings



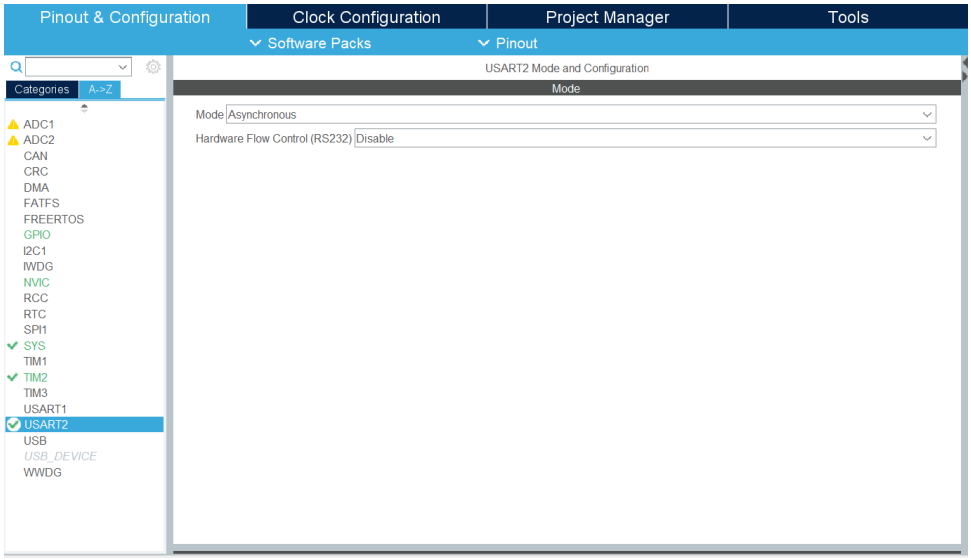
Hình 5.5: Cấu hình timer: TIM2 Parameter Settings



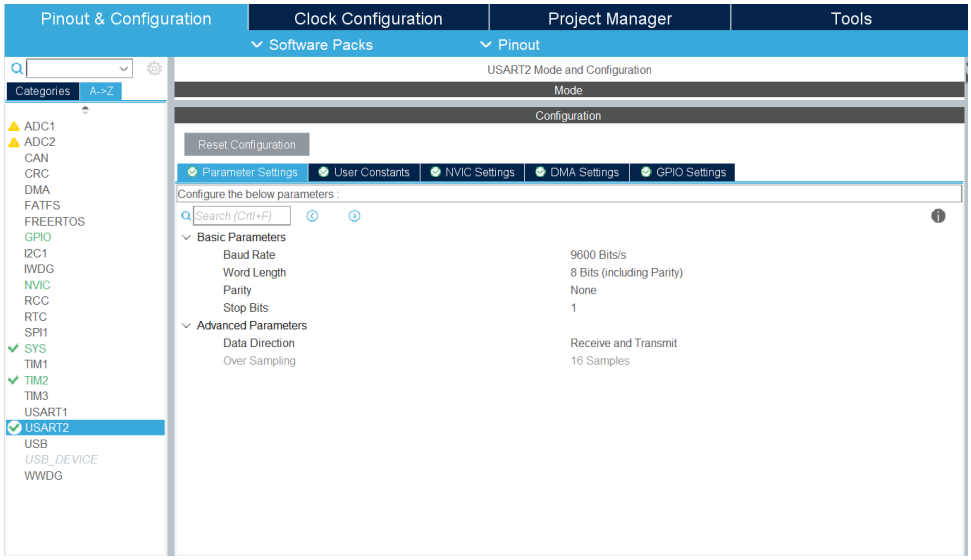
Hình 5.6: Cấu hình timer: TIM2 Parameter Settings

Cấu hình giao tiếp

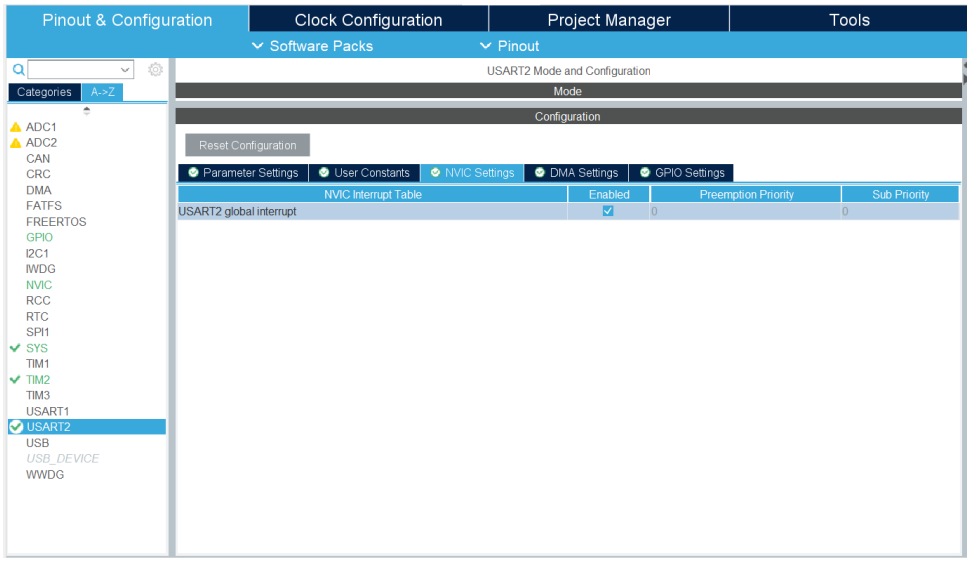
- Dự án chọn sử dụng USART2.
- Chọn Mode Asynchronous cho USART2 (Hình 5.7). c Đặt Baud Rate bằng 9600, Word Length bằng 8 Bits, Parity bằng None, Stop Bits bằng 1 (Hình 5.8).
- Enable USART2 global interupt trong cài đặt NVIC (Hình 4.7).



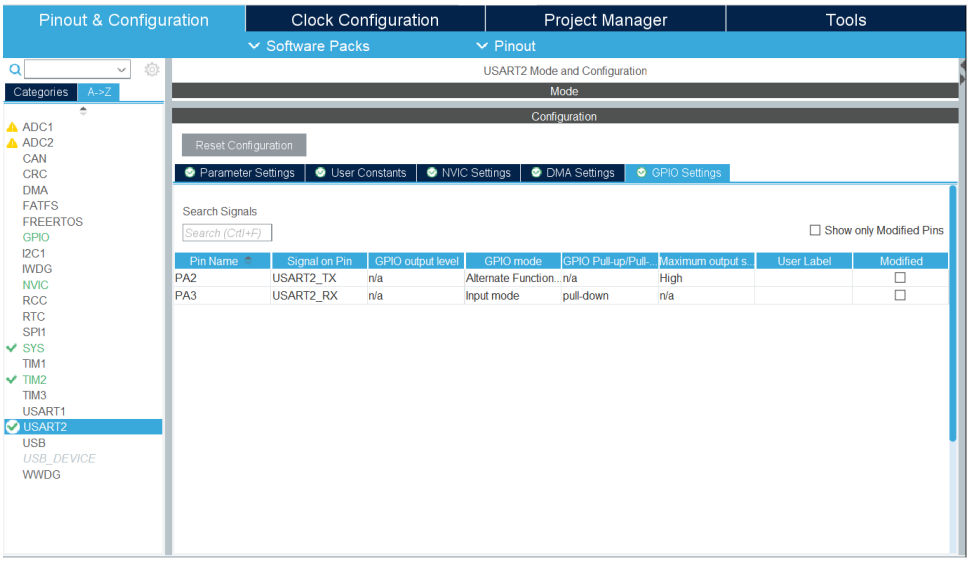
Hình 5.7: Cấu hình giao tiếp: USART2 Mode



Hình 5.8: Cấu hình giao tiếp: USART2 Parameter Settings



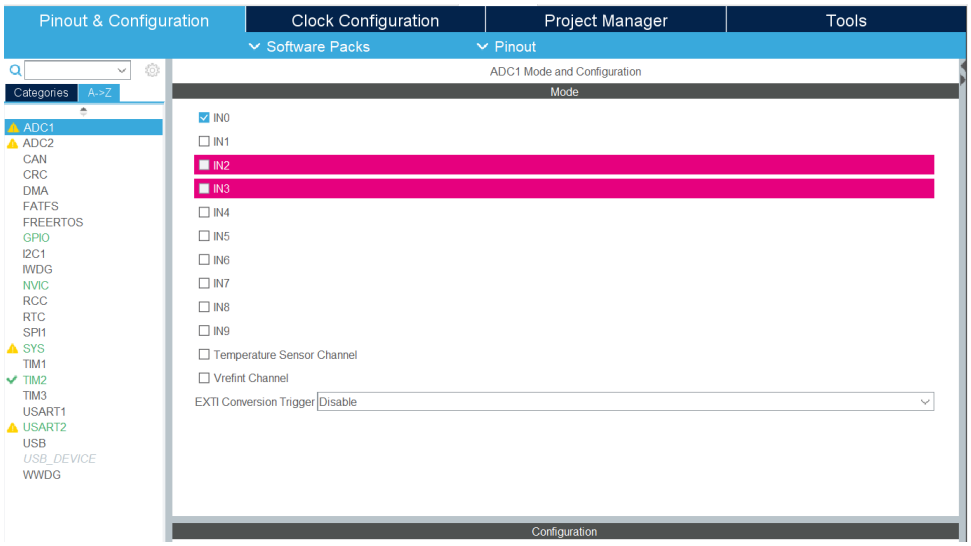
Hình 5.9: : Cấu hình giao tiếp: USART2 NVIC Settings



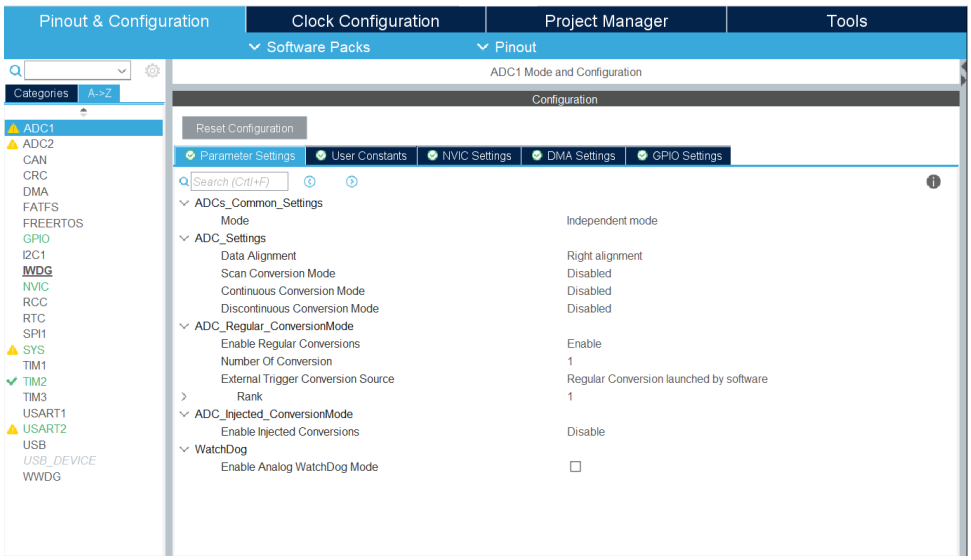
Hình 5.10: Cấu hình giao tiếp: USART2 GPIO Settings

Cấu hình ADC

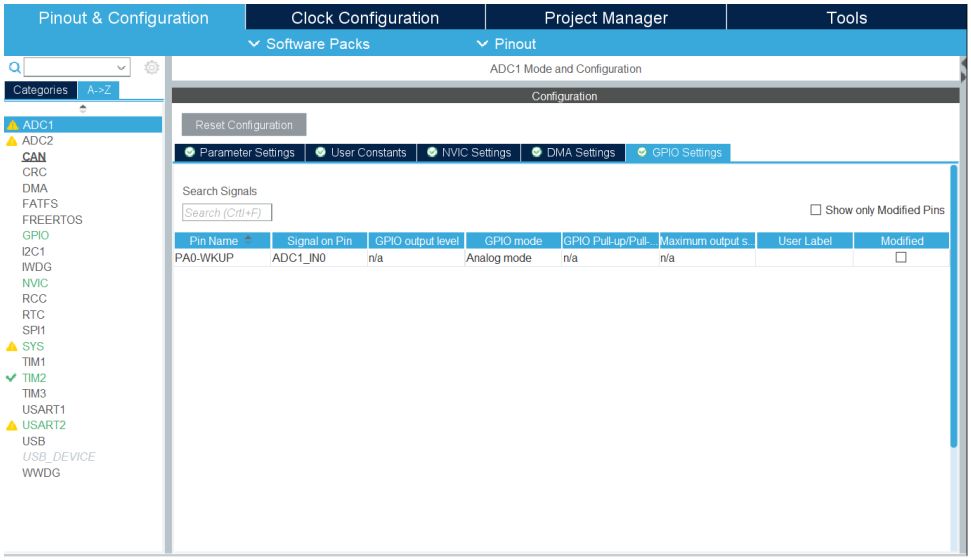
- Chọn sử dụng ADC1.
- Chọn mode IN0 cho ADC1 (Hình 5.11).
- Enable Continuous Conversion Mode trong cài đặt Parameter (Hình 5.12).
- Giữ nguyên giá trị mặc định cho các chân GPIO của ADC1 (Hình 5.13).



Hình 5.11: Cấu hình ADC: ADC1 Mode



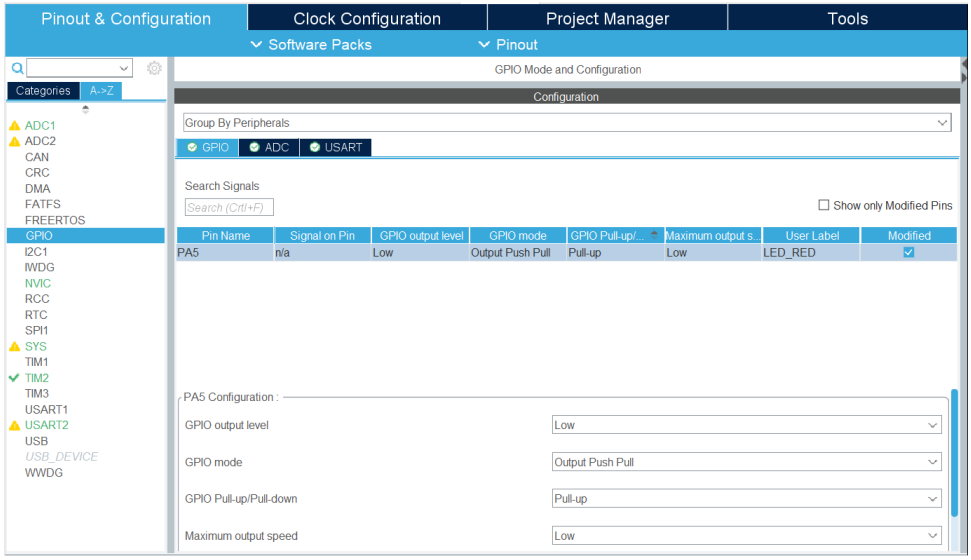
Hình 5.12: Cấu hình ADC: ADC1 Parameter Settings



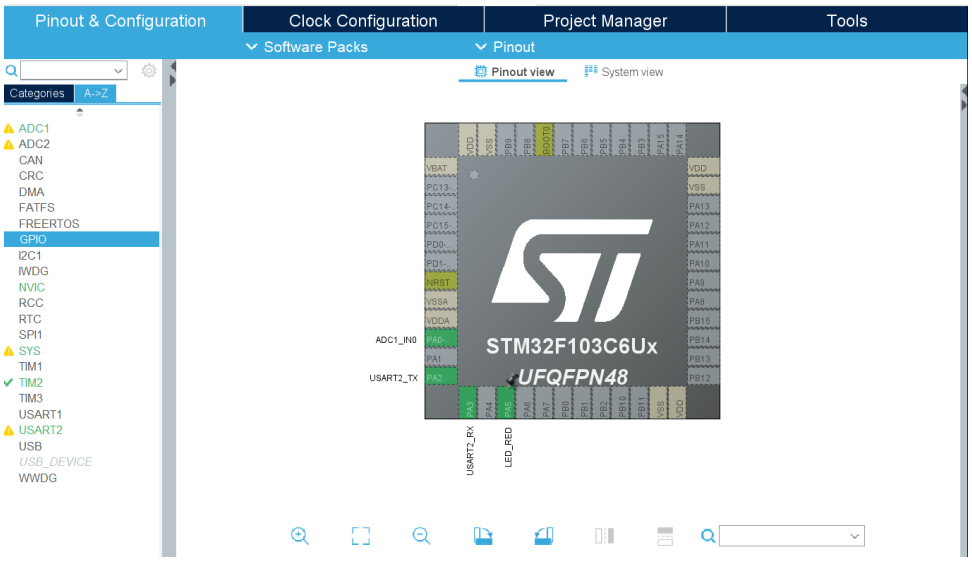
Hình 5.13: Cấu hình ADC: ADC1 GPIO Settings

Cấu hình chân tín hiệu

- Sử dụng chân tín hiệu PA5 điều khiển LED_RED.
- Thiết lập GPIO Mode Output Push Pull, GPIO Pull-up (Hình 5.14).
- Kết quả cuối cùng được thể hiện ở Hình 5.15.
- Giữ nguyên giá trị mặc định cho các chân GPIO của ADC1 (Hình 5.13).



Hình 5.14: Cấu hình chân tín hiệu: GPIO



Hình 5.15: Cấu hình chân tín hiệu: Pinout

5.2.3 Sơ đồ nguyên lý

Linh kiện sử dụng

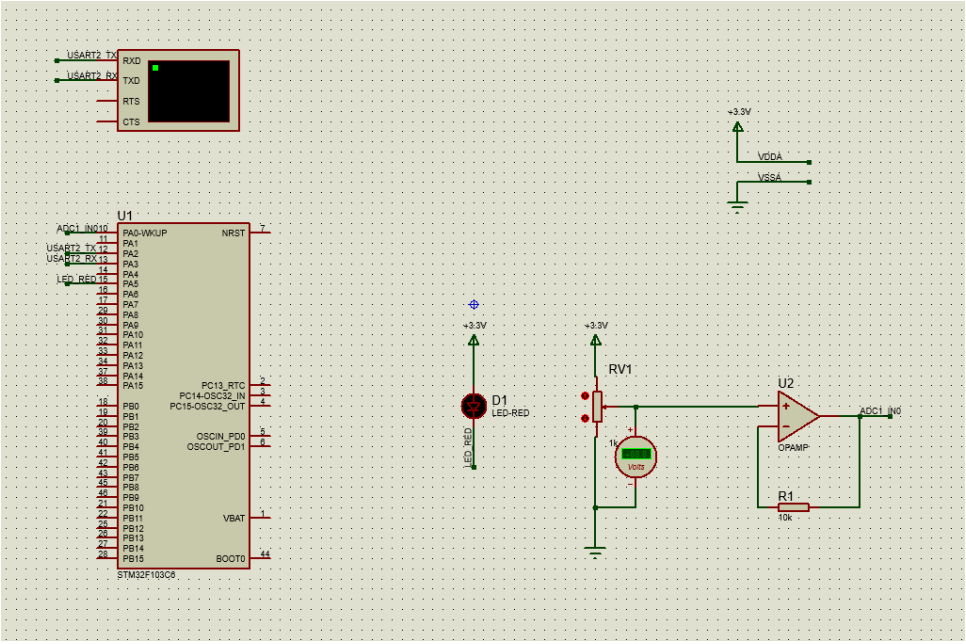
- STM32F103C6: Vi điều khiển trung tâm
- LED-RED: LED đơn.
- OPAMP: Opamp.
- POT: Biến trở..
- RES: Điện trở cố định.
- DC VOLTMETER: Vôn kế.
- VIRTUAL TERMINAL: Terminal hiển thị giá trị.

Sơ đồ nối dây

- Kết nối LED đơn với vi điều khiển trung tâm.
  – Cathode của các LED đơn nối đến các chân tương ứng từ PA5 của vi điều khiển trung tâm.
  – Anode của các LED đơn được nối với nguồn điện 3.3V.
  – Để đơn giản, giả sử dòng điện do vi điều khiển cung cấp thỏa mãn điều kiện hoạt động của LED và bỏ qua các điện trở hạn dòng.
- KKết nối Virtual Terminal với vi điều khiển trung tâm:
  – Nối chân RXD của Virtual Terminal với chân PA2 của vi điều khiển. – Nối chân TXD của Virtual Terminal với chân PA3 của vi điều khiển.
- Cấu hình ADC1 nhận dữ liệu từ Opamp:
  – Nối 2 chân cố định của biến trở lần lượt với nguồn 3.3V và GND.
  – Nối chân V+ của Opamp với chân điều khiển của biến trở, đồng thời nối thêm

với một vôn kế để thuận lợi trong việc gỡ lỗi.
– Lần lượt nối chân V- và Output của Opamp với điện trở cố định.
– Nối chân PA0 (ADC1_IN0) với chân Output của Opamp để hoàn tất quá trình cấu hình.

- Kết quả thu được thể hiện ở Hình 5.16.



Hình 5.16: Sơ đồ nối dây: Schematic

5.3 Hiện thực

5.3.1 Bộ định thời

Sử dụng header file scheduler.h và source file scheduler.c để quản lý trình lập lịch. Cấu trúc dữ liệu

- functionPointer: Con trỏ hàm của tác vụ (bắt buộc có kiểu hàm void(void)). Nếu functionPointer == 0 (trỏ đến null), tác vụ chưa được thiết lập và ngược lại.
- id: Mã số định danh của mỗi tác vụ. Trong suốt thời gian tồn tại, mỗi tác vụ có một mã số định danh duy nhất.
- delay: Khoảng thời gian (ms) cho đến lần thực thi kế tiếp của tác vụ.
- period: Khoảng thời gian (ms) giữa hai lần chạy liên tiếp của tác vụ. Nếu period == 0, tác vụ chỉ được thực hiện một lần duy nhất.
- flag: Cờ thông báo tác vụ có sẵn sàng được thi hay chưa. Nếu flag == 1, tác vụ đã sẵn sàng thực thi và ngược lại.

```
typedef struct {
    void (*functionPointer)(void);
    uint8_t id;
    uint32_t delay;
    uint32_t period;
    unsigned char flag;
} SCH_Task;
```

Chương trình 5.1: Bộ định thời: Cấu trúc dữ liệu

Các hàm sử dụng

- SCH_Init: Khởi tạo cơ sở dữ liệu nhằm lưu trữ các tác vụ. Các tác vụ được gán giá trị ID ngay khi khởi tạo, các giá trị ID này nằm trong đoạn [1, SCH_TASKNUMBER] (giá trị 0 không được sử dụng làm ID).
- SCH_Update: Cập nhật thời gian đợi còn lại của tác vụ trong hàng đợi. Đây là hàm được gọi trong ngắt timer, bộ định thời phải đảm bảo hàm được thực hiện với thời gian O(1).
- SCH_Dispatch: Thực thi các tác vụ đã sẵn sàng trong hàng đợi. Đồng thời đảm bảo hoạt động của các tác vụ thực thi định kỳ bằng cách gọi hàm SCH_RefreshTask.
- SCH_AddTask: Thêm tác vụ vào cơ sở dữ liệu và trả về ID của tác vụ nếu thành công, ngược lại trả về 0.
- SCH_DeleteTask: Xóa tác vụ khỏi cơ sở dữ liệu và trả về 0.
- SCH_RefreshTask: Làm mới tác vụ thay cho việc lần lượt xóa và thêm một tác vụ khi cập nhật trong hàm SCH_Dispatch nhằm đảm bảo ID của tác vụ không bị thay đổi.

```
void SCH_Init(void);
void SCH_Update(void);
void SCH_Dispatch(void);

uint8_t SCH_AddTask(void (*functionPointer)(void), uint32_t
   delay, uint32_t period);

unsigned char SCH_DeleteTask(uint8_t id);

unsigned char SCH_RefreshTask(void);
```

Chương trình 5.2: Bộ định thời: Các hàm sử dụng

```
void SCH_Init(void) {
    for (uint8_t i = 0; i < SCH_TASKNUMBER; i++) {
        tasks[i].functionPointer = NULL;
        tasks[i].id = SCH_TASKNUMBER - i - 1;
        tasks[i].delay = 0;
        tasks[i].period = 0;
        tasks[i].flag = 0;
    }
}
```

Chương trình 5.3: Hiện thực: Hàm SCH_Init

```
void SCH_Update(void) {
    if (tasks[0].functionPointer == 0) return;
    if (tasks[0].delay > 0) {
        if (tasks[0].delay > TIMER_TICK) {
            tasks[0].delay -= TIMER_TICK;
        } else {
            tasks[0].delay = 0;
        }
    }
    if (tasks[0].delay == 0) {
        tasks[0].flag = 1;
    }
}
```

Chương trình 5.4: Hiện thực: Hàm SCH_Update

```
void SCH_Dispatch(void) {
    if (tasks[0].flag == 0) return;
    (*tasks[0].functionPointer)();
    if (tasks[0].period > 0) {
        SCH_RefreshTask();
    } else {
        SCH_DeleteTask(tasks[0].id);
    }
}
```

Chương trình 5.5: Hiện thực: Hàm SCH_Dispatch

```
uint8_t SCH_AddTask(void (*functionPointer)(void), uint32_t
   delay, uint32_t period) {
    if (tasks[SCH_TASKNUMBER - 1].functionPointer != 0)
       return 0;
    uint8_t currentID = tasks[SCH_TASKNUMBER - 1].id;
```

```
    uint32_t currentDelay = 0;
    for (uint8_t i = 0; i < SCH_TASKNUMBER; i++) {
        currentDelay += tasks[i].delay;
        if (currentDelay > delay || tasks[i].functionPointer
           == 0) {
            for (uint8_t j = SCH_TASKNUMBER - 1; j > i; j--)
               {
                tasks[j] = tasks[j - 1];
            }
            tasks[i].functionPointer = functionPointer;
            tasks[i].id = currentID;
            tasks[i].period = period;
            tasks[i].flag = 0;
            if (currentDelay > delay) {
                int newDelay = currentDelay - delay;
                tasks[i].delay = tasks[i + 1].delay -
                   newDelay;
                if (tasks[i].delay == 0) {
                    tasks[i].flag = 1;
                }
                tasks[i + 1].delay = newDelay;
                if (tasks[i + 1].delay == 0) {
                    tasks[i + 1].flag = 1;
                }
            } else {
                tasks[i].delay = delay - currentDelay;
                if (tasks[i].delay == 0) {
                    tasks[i].flag = 1;
                }
            }
            return tasks[i].id;
        }
    }
    return 0;
}
```

Chương trình 5.6: Hiện thực: Hàm SCH_AddTask

```
unsigned char SCH_DeleteTask(uint8_t id) {
    for (uint8_t i = 0; i < SCH_TASKNUMBER; i++) {
        if (tasks[i].functionPointer == 0) return 0;

        if (tasks[i].id == id) {
            uint8_t currentID = tasks[i].id;

            if (tasks[i + 1].functionPointer != 0) {
                tasks[i + 1].delay += tasks[i].delay;
            }
```

```
11
12              for (uint8_t j = i; j < SCH_TASKNUMBER - 1; j++)
                    {
13                  tasks[j] = tasks[j + 1];
14              }
15
16              tasks[SCH_TASKNUMBER - 1].functionPointer = 0;
17              tasks[SCH_TASKNUMBER - 1].id = currentID;
18              tasks[SCH_TASKNUMBER - 1].delay = 0;
19              tasks[SCH_TASKNUMBER - 1].period = 0;
20              tasks[SCH_TASKNUMBER - 1].flag = 0;
21
22              return tasks[SCH_TASKNUMBER - 1].functionPointer
                    == 0;
23          }
24      }
25      return 0;
26  }
```

Chương trình 5.7: Hiện thực: Hàm SCH_DeleteTask

```
1   unsigned char SCH_RefreshTask(void) {
2       if (tasks[0].functionPointer == 0) return 0;
3       SCH_Task currentTask = tasks[0];
4       uint32_t currentDelay = 0;
5
6       for (uint8_t i = 0; i < SCH_TASKNUMBER; i++) {
7           if (i + 1 == SCH_TASKNUMBER || tasks[i + 1].
                functionPointer == NULL) {
8               tasks[i].functionPointer = currentTask.
                    functionPointer;
9               tasks[i].id = currentTask.id;
10              tasks[i].period = currentTask.period;
11              tasks[i].flag = 0;
12              tasks[i].delay = currentTask.period -
                    currentDelay;
13
14              if (tasks[i].delay == 0) {
15                  tasks[i].flag = 1;
16              }
17
18              return 1;
19          }
20
21          currentDelay += tasks[i + 1].delay;
22
23          if (currentDelay > currentTask.period) {
```

```
24              tasks[i].functionPointer = currentTask.
                   functionPointer;
25              tasks[i].id = currentTask.id;
26              tasks[i].period = currentTask.period;
27              tasks[i].flag = 0;
28
29              int newDelay = currentDelay - currentTask.period;
30              tasks[i].delay = tasks[i + 1].delay - newDelay;
31
32              if (tasks[i].delay == 0) {
33                  tasks[i].flag = 1;
34              }
35
36              tasks[i + 1].delay -= tasks[i].delay;
37
38              if (tasks[i + 1].delay == 0) {
39                  tasks[i + 1].flag = 1;
40              }
41
42              return 1;
43          } else {
44              tasks[i] = tasks[i + 1];
45          }
46      }
47
48      return 0;
49  }
```

Chương trình 5.8: Hiện thực: Hàm SCH_RefreshTask

5.3.2 LED

- Sử dụng header file led.h và source file led.c riêng biệt để quản lý LED nhằm thuận tiện cho việc mở rộng (nếu có).

```
1  #ifndef INC_LED_H_
2  #define INC_LED_H_
3
4  #include "main.h"
5
6  void ledBlink(void);
7
8  #endif /* INC_LED_H_ */
```

Chương trình .9: LED: Header file

```c
#include "led.h"

void ledBlink(void) {
    HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
}
```

Chương trình 5.10: LED: Source file

5.3.3 Cảm biến

- Sử dụng header file sensor.h và source file sensor.c để quản lý cảm biến và việc đọc dữ liệu ADC từ cảm biến.

```c
#ifndef INC_SENSOR_H_
#define INC_SENSOR_H_

#include "main.h"

extern uint32_t sensorValue;

void sensorRead(void);

#endif /* INC_SENSOR_H_ */
```

Chương trình 5.11: Cảm biến: Header file

```c
#include "sensor.h"

uint32_t sensorValue = 0;

void sensorRead(void) {
    sensorValue = HAL_ADC_GetValue(&hadc1);
}
```

Chương trình 5.12: Cảm biến: Source file

5.3.4 Giao tiếp UART

- Sử dụng header file uart.h và source file uart.c để quản lý việc đọc, ghi dữ liệu thông qua giao tiếp UART.
- Hàm uartRead thực hiện đọc dữ liệu người dùng nhập vào từ Virtual Terminal và bật cờ uartFlag. Sau đó, yêu cầu trình phân tích lệnh thực hiện nhiệm vụ.
- Hàm uartSend thực hiện truyền dữ liệu đến người dùng thông qua Virtual Terminal với dữ liệu và cú pháp đã quy định.

```c
#ifndef INC_UART_H_
#define INC_UART_H_

#include <stdio.h>
#include "command.h"
#include "main.h"
#include "sensor.h"

#define UART_SIZE 10

extern uint8_t uartChar;
extern uint8_t uartBuffer[UART_SIZE];
extern uint8_t uartIndex;
extern unsigned char uartFlag;

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart);
void uartRead(void);
void uartSend(void);

#endif /* INC_UART_H_ */
```

Chương trình 5.13: Giao tiếp UART: Header file

```c
#include "uart.h"

uint8_t uartChar = 0;
uint8_t uartBuffer[UART_SIZE];
uint8_t uartIndex = 0;
unsigned char uartFlag = 0;
char uartMessage[100];

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if (huart->Instance == USART2) {
        uartBuffer[uartIndex] = uartChar;
        uartFlag = 1;
        HAL_UART_Receive_IT(&huart2, &uartChar, 1);
    }
}

void uartRead(void) {
    if (uartFlag == 1) {
        uartFlag = 0;
        SCH_AddTask(commandParser, 0, 0);
    }
}

void uartSend(void) {
```

```
25      HAL_UART_Transmit(&huart2, (void *)uartMessage, sprintf(
            uartMessage, "\r\n! ADC = %ld#", sensorValue), 100);
26  }
```

Chương trình 5.14: Giao tiếp UART: Source file

5.3.5 Trình quản lý lệnh

- Sử dụng header file command.h và source file command.c để quản lý việc phân tích, thực thi các lệnh.

- HHàm commandExecute thực hiện đối chiếu lệnh hiện tại với các lệnh khả dụng và thực thi theo máy trạng thái đã được thể hiện ở Hình 5.2.
  – Nếu nhận lệnh !RST#, gọi thực thi tác vụ đọc dữ liệu từ cảm biến, sau đó xóa tác vụ phản hồi hiện tại (nếu có) và gọi thực thi tác vụ phản hồi mới.
  – Tác vụ phản hồi mới này được thực hiện lặp lại sau mỗi chu kỳ 3 giây. Ngoài ra, tác vụ cũng được đảm bảo truyền đúng số liệu bằng cách thêm một khoảng delay nhỏ.
  – Nếu nhận lệnh !OK#, thực hiện xóa tác vụ phản hồi hiện tại.
  – Vì bộ định thời đã kiểm soát chặt chẽ việc xóa tác vụ không tồn tại, do đó trình quản lý lệnh bỏ qua các ràng buộc và trực tiếp gọi lệnh xóa tác vụ.

- Hàm commandParser thực hiện nhận và phân tích từng ký tự trong lệnh của người dùng theo đúng máy trạng thái đã được thể hiện ở Hình 5.1.

  – Trong trạng thái sẵn sàng (khi uartIndex == 0), trình phân tích lệnh chỉ chấp nhận ký tự khởi đầu !.
  – Trong trạng thái nhận lệnh, trình phân tích lệnh ghi giá trị vào bộ đệm và tăng giá trị uartIndex không vượt quá giá trị tối đa UART_SIZE.
  – Khi nhận được ký tự kết thúc #, trình phân tích lệnh gọi hàm thực thi thông qua thêm tác vụ thực hiện một lần. Sau đó, các giá trị uartIndex và uartBuffer sẽ được làm mới để chuẩn bị nhận lệnh tiếp theo.

```
1  #ifndef INC_COMMAND_H_
2  #define INC_COMMAND_H_
3
4  #include <string.h>
5  #include "scheduler.h"
6  #include "uart.h"
7
8  void commandExecute(void);
9  void commandParser(void);
10
11 #endif /* INC_COMMAND_H_ */
```

Chương trình 5.15: Trình quản lý lệnh: Header file

```c
#include "command.h"

static uint8_t commandTask = 0;
char commandString[UART_SIZE];

void commandExecute(void) {
    if (strcmp(commandString, "!RST#") == 0) {
        SCH_AddTask(sensorRead, 0, 0);
        commandTask = SCH_DeleteTask(commandTask);
        commandTask = SCH_AddTask(uartSend, 100, 3000);
        return;
    }

    if (strcmp(commandString, "!OK#") == 0) {
        commandTask = SCH_DeleteTask(commandTask);
        return;
    }
}

void commandParser(void) {
    if (uartIndex == 0) {
        if (uartBuffer[uartIndex] == '!') {
            uartIndex++;
        }
    } else {
        if (uartBuffer[uartIndex] == '#') {
            sprintf(commandString, "%s", uartBuffer);
            SCH_AddTask(commandExecute, 0, 0);
            memset(uartBuffer, 0, UART_SIZE);
            uartIndex = 0;
        } else {
            if (uartIndex < UART_SIZE - 1) {
                uartIndex++;
            }
        }
    }
}
```

Chương trình 5.16: Trình quản lý lệnh: Source file

5.3.6 Tổng hợp

- Thực hiện định nghĩa bổ sung các giá trị cần thiết ở header file main.h
- Thực hiện #include các thư viện cần thiết ở source file main.c
- Tiến hành các khởi tạo cần thiết cho timer, giao tiếp, ADC và bộ định thời trước khi vào vòng lặp while(1).

- Khởi tạo ngắt timer và gọi hàm SCH_Update trong hàm ngắt timer.
- Gọi hàm SCH_Dispatch trong vòng lặp while(1).
- Khởi tạo các tác vụ cần thiết trước ngay trước vòng lặp while(1) gồm: – Thay đổi trạng thái của LED-RED sau mỗi 1 giây nhằm kiểm tra hệ thống. – Đọc tín hiệu từ người dùng qua giao tiếp UART sau mỗi TIMER_TICK.

```
/* USER CODE BEGIN Private defines */
#define TIMER_TICK 10
extern UART_HandleTypeDef huart2;
extern ADC_HandleTypeDef hadc1;
/* USER CODE END Private defines */
```

Chương trình 5.17: Tổng hợp: Private defines

```
/* USER CODE BEGIN Includes */
#include <stdint.h>
#include <stdio.h>
#include "led.h"
#include "sensor.h"
#include "scheduler.h"
#include "uart.h"
/* USER CODE END Includes */
```

Chương trình 5.18: Tổng hợp: Includes

```
/* USER CODE BEGIN 2 */
HAL_UART_Receive_IT(&huart2, &uartChar, 1);
HAL_ADC_Start(&hadc1);
HAL_TIM_Base_Start_IT(&htim2);
SCH_Init();
/* USER CODE END 2 */
```

Chương trình 5.19: Tổng hợp: Khởi tạo

```
/* USER CODE BEGIN WHILE */
SCH_AddTask(ledBlink, 0, 1000);
SCH_AddTask(uartRead, 0, TIMER_TICK);

while (1) {
/* USER CODE END WHILE */
```

Chương trình 5.20: Tổng hợp: Thêm tác vụ

```
/* USER CODE BEGIN 3 */
SCH_Dispatch();
/* USER CODE END 3 */
```

Chương trình 5.21: Tổng hợp: Thực thi tác vụ

```
/* USER CODE BEGIN 4 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    if (htim->Instance == TIM2) {
        SCH_Update();
    }
}
/* USER CODE END 4 */
```

Chương trình 5.22: Tổng hợp: Hàm ngắt timer