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 4: COOPERATIVE SCHEDULER

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, 11/2024

Mục lục

Chương 4

Cooperative Scheduler

Link GitHub

4.1 Đặc tả yêu cầu

4.1.1 Yêu cầu chung

- Hiện thực trình lập lịch quản lý tác vụ.
- Các tác vụ được gọi trong hàm ngắt timer phải có độ phức tạp O(1)

4.1.2 Các hàm cần hiện thực

- SSCH_Init: Khởi tạo cơ sở dữ liệu nhằm lưu trữ các tác vụ.
- SCH_Update: Cập nhật thời gian đợi còn lại của tác vụ trong hàng đợi.
- SCH_Dispatch: Thực thi các tác vụ đã sẵn sàng trong hàng đợi.
- SCH_AddTask: Thêm tác vụ vào cơ sở dữ liệu.
- SSCH_DeleteTask: Xóa tác vụ khỏi cơ sở dữ liệu.

4.1.3 Các yêu cầu mô phỏng

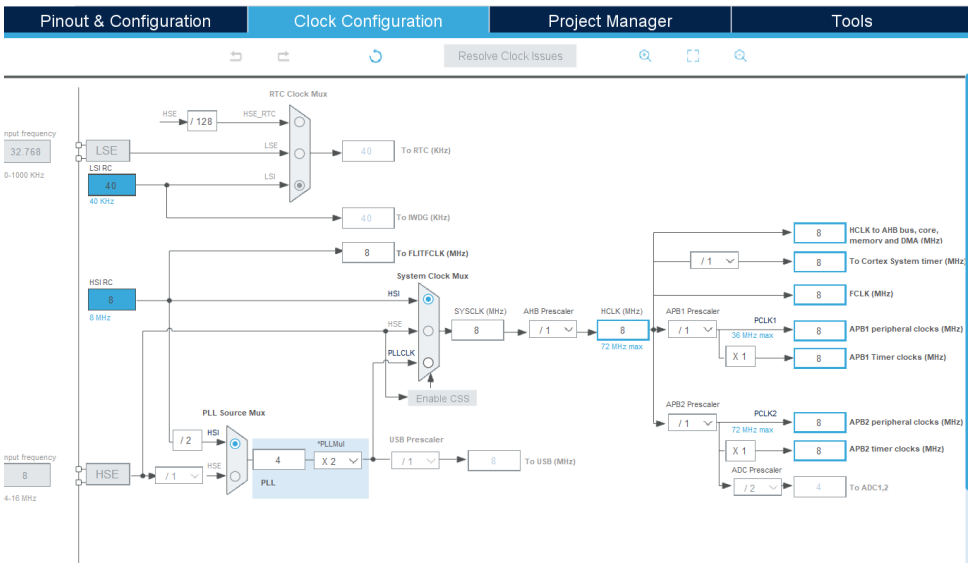
- Timer tick có giá trị 10ms, hàm ngắt timer được gọi mỗi 10ms.
- Có ít nhất 5 tác vụ thực hiện đồng thời với chu kỳ 0,5; 1; 1,5; 2 và 2,5 giây
- In giá trị thời gian ra màn hình mỗi khi hàm ngắt timer được gọi (mỗi 10ms).
- In giá trị thời gian ra màn hình mỗi 500ms, thực hiện đồng thời với lệnh in trước.

4.2 Cấu hình hệ thống

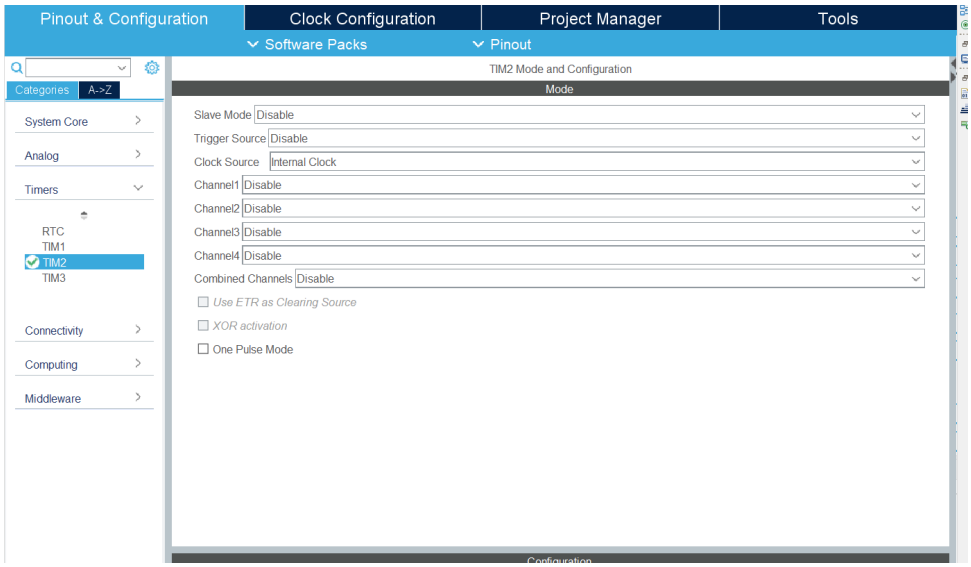
4.2.1 Cấu hình vi điều khiển

Cấu hình timer

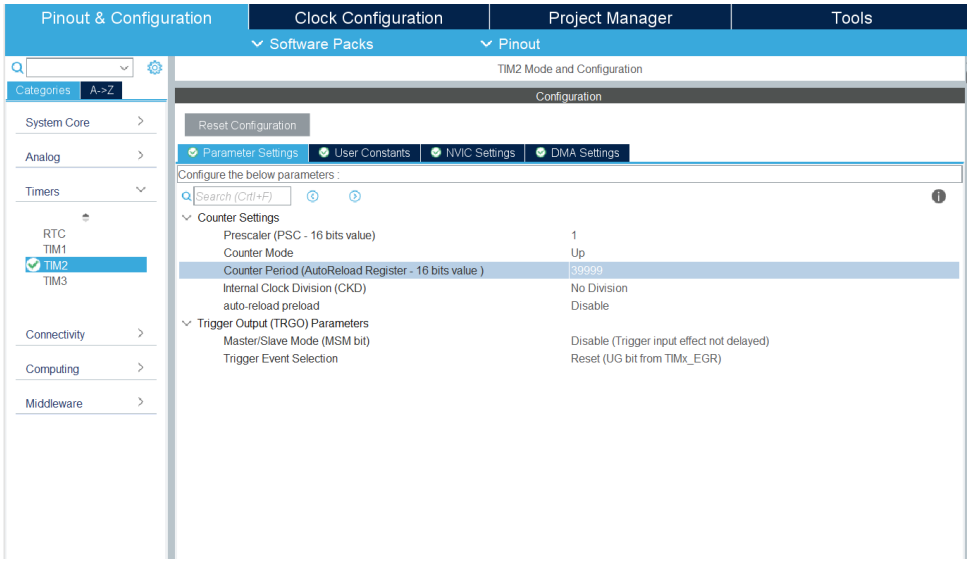
- Dự án chọn sử dụng TIM2.
- Đặt giá trị tần số timer bằng 8MHz (Hình 4.1).
- Chọn Internal Clock làm Clock Source của TIM2 (Hình 4.2).



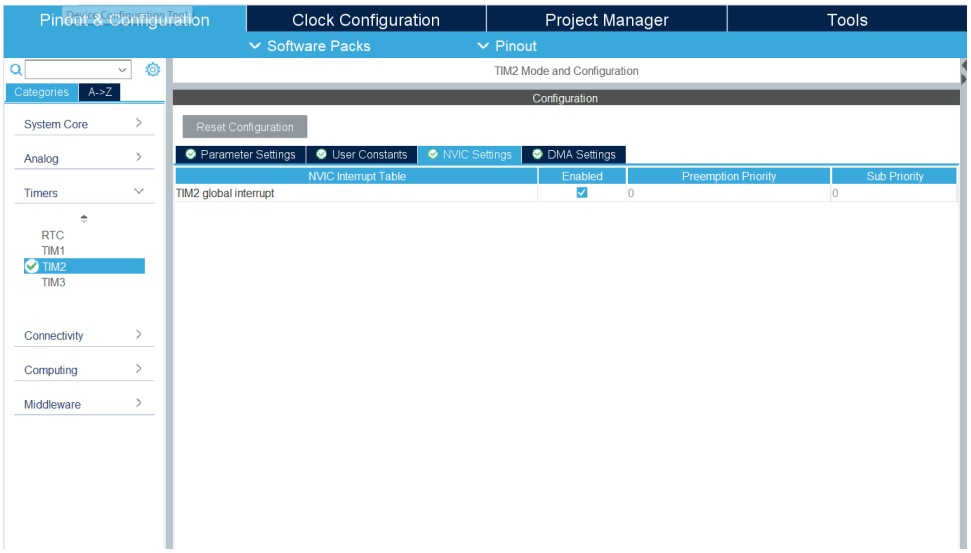
Hình 4.1: Cấu hình timer: Clock Configuration



Hình 4.2: Cấu hình timer: TIM2 Clock Source



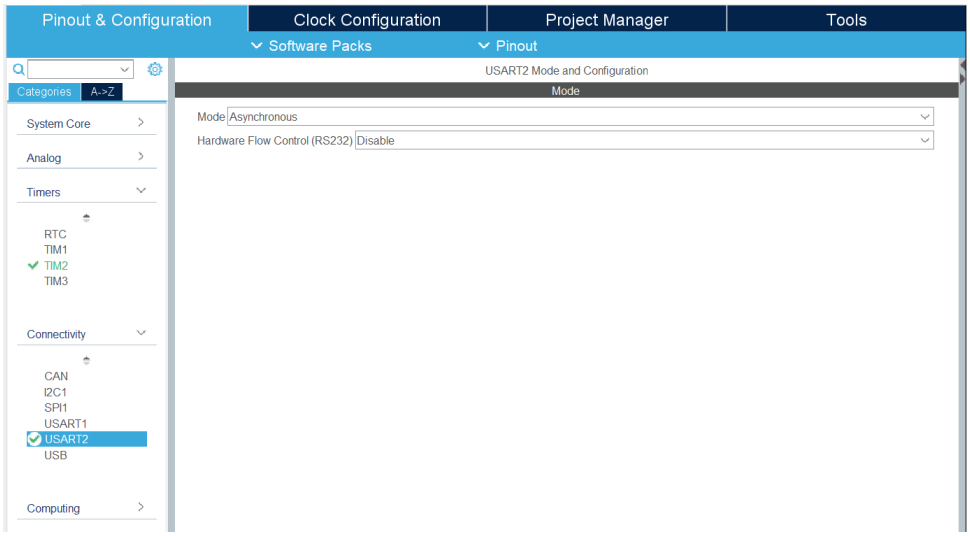
Hình 4.3: Cấu hình timer: TIM2 Parameter Settings



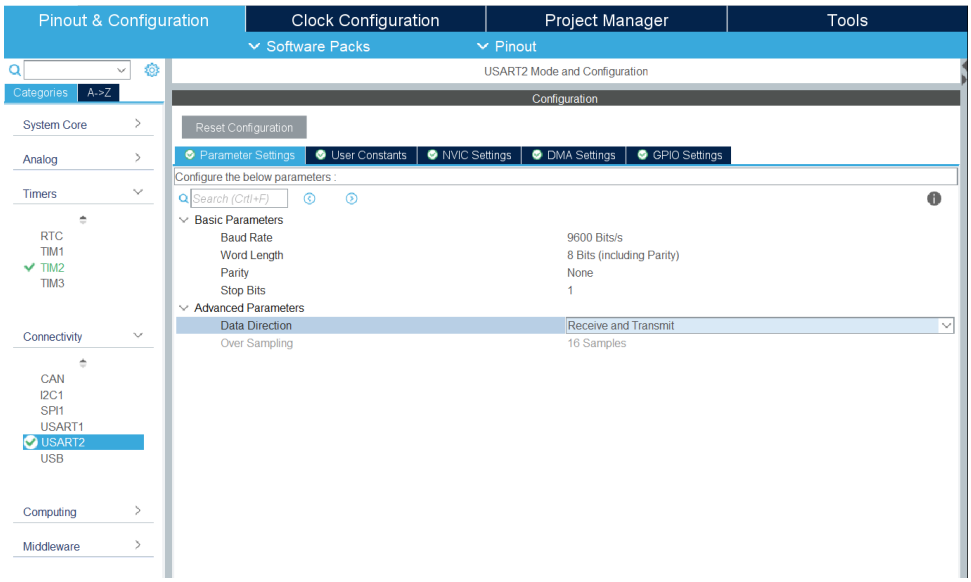
Hình 4.4: Cấu hình timer: TIM2 NVIC Settings

Cấu hình giao tiếp

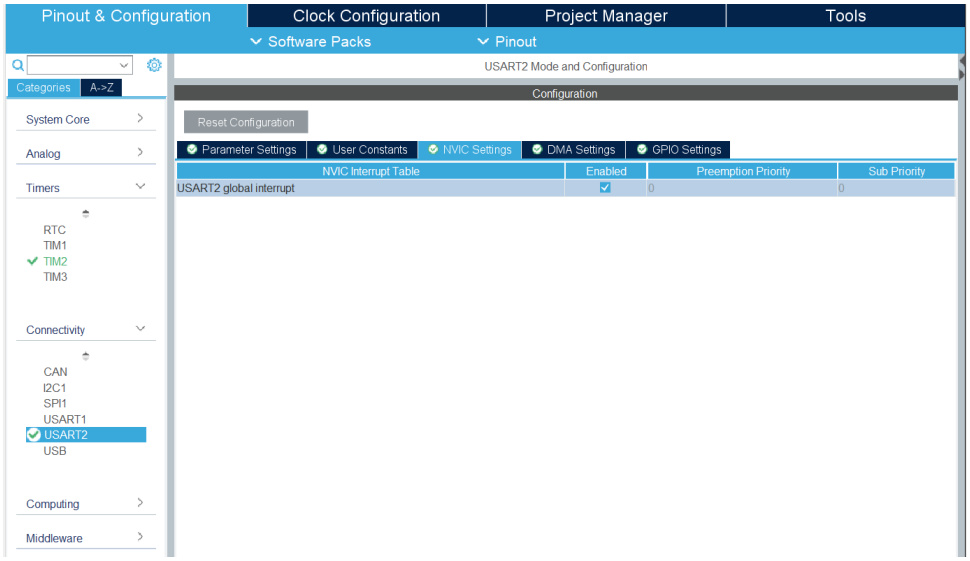
- Dự án chọn sử dụng USART2.
- Chọn Mode Asynchronous cho USART2 (Hình 4.5). c Đặt Baud Rate bằng 9600, Word Length bằng 8 Bits, Parity bằng None, Stop Bits bằng 1 (Hình 4.3).
- Enable USART2 global interupt trong cài đặt NVIC (Hình 4.7).



Hình 4.5: Cấu hình giao tiếp: USART2 Mode



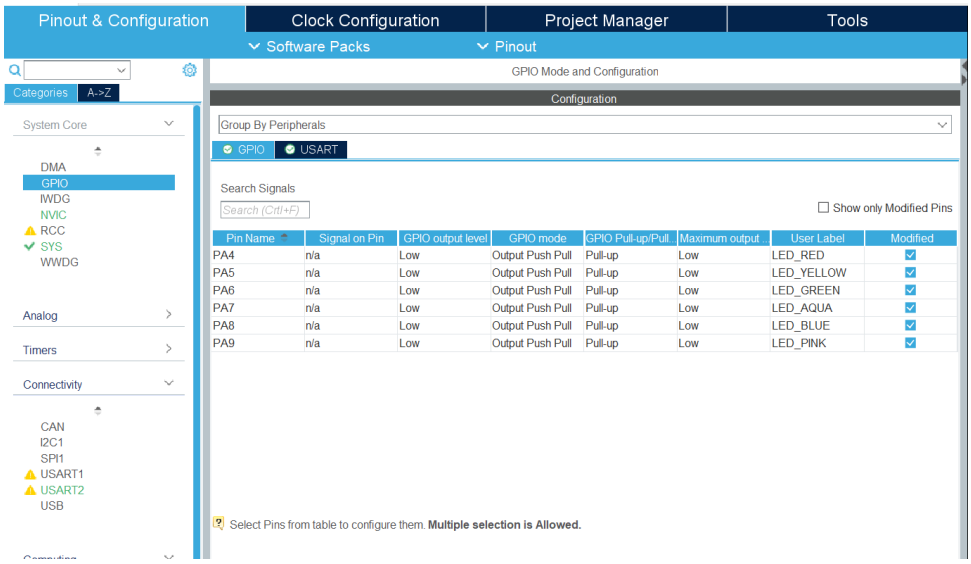
Hình 4.6: Cấu hình giao tiếp: USART2 Parameter Settings



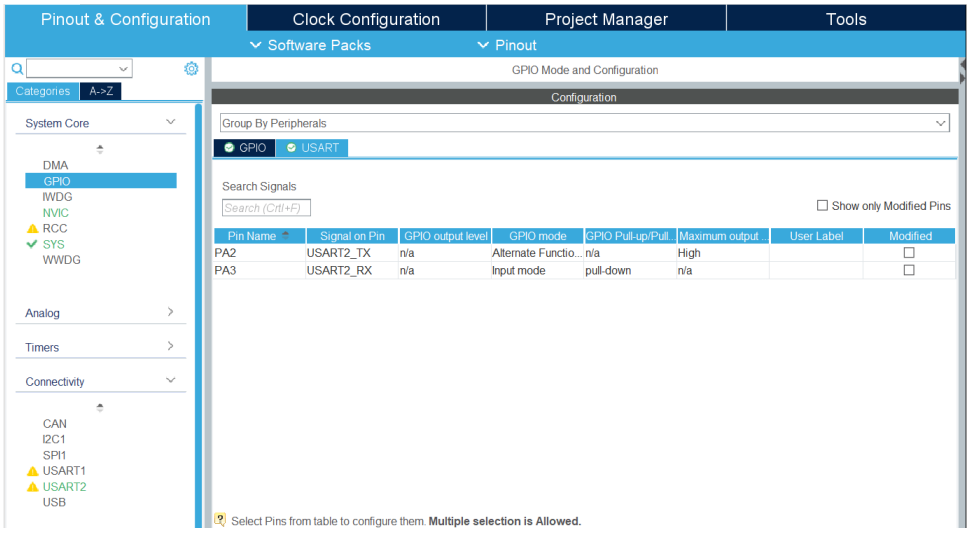
Hình 4.7: Cấu hình giao tiếp: USART2 NVIC Settings

Cấu hình chân tín hiệu

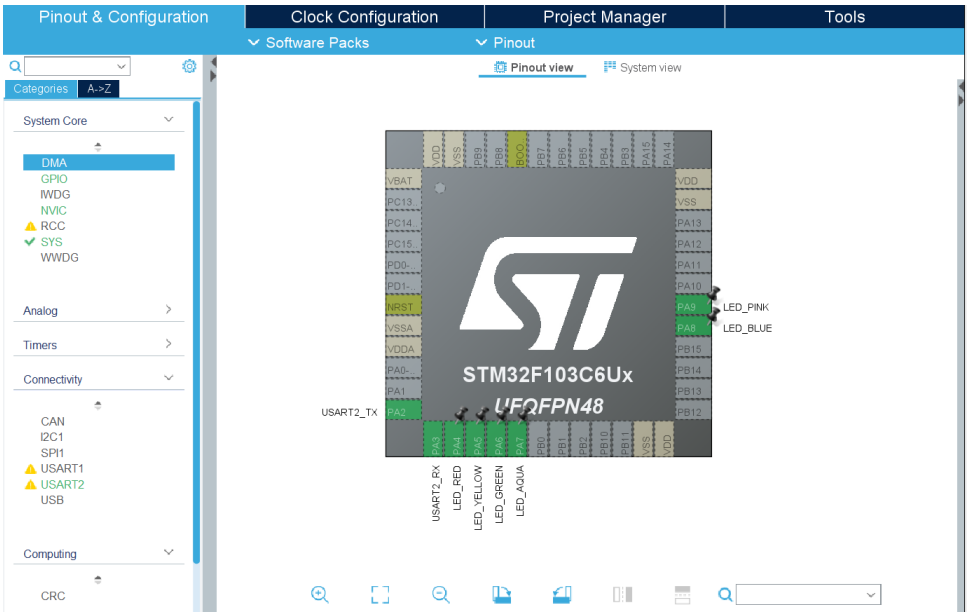
- Sử dụng 6 chân tín hiệu từ PA4 đến PA9 lần lượt điều khiển LED-RED, LEDYELLOW, LED-GREEN, LED-AQUA, LED-BLUE, LED-PINK.
- Thiết lập GPIO Mode Output Push Pull, GPIO Pull-up cho các chân output (Hình 4.8).
- Các chân USART sử dụng giá trị mặc định (Hình 4.9).
- Kết quả cuối cùng được thể hiện ở Hình 4.10.



Hình 4.8: Cấu hình chân tín hiệu: GPIO



Hình 4.9: Cấu hình chân tín hiệu: USART



Hình 4.10: Cấu hình chân tín hiệu: Pinout

4.2.2 Sơ đồ nguyên lý

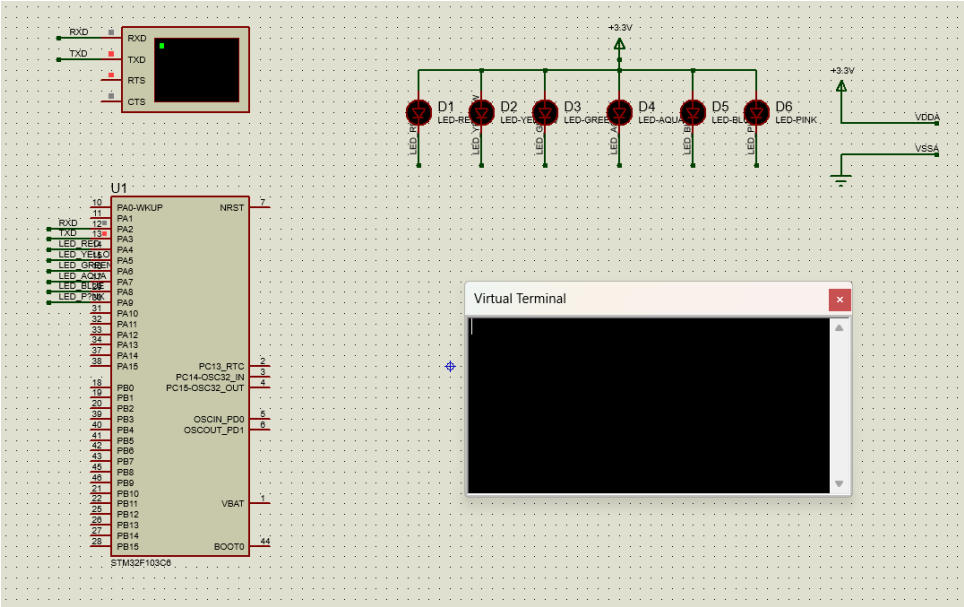
Linh kiện sử dụng

- STM32F103C6: Vi điều khiển trung tâm
- LED-RED: LED đơn (màu đỏ).
- LED-YELLOW: LED đơn (màu vàng).
- LED-GREEN: LED đơn (màu xanh lục).
- LED-AQUA: LED đơn (màu xanh lam).

- LED-BLUE: LED đơn (màu xanh dương).
- LED-PINK: LED đơn (màu hồng).
- VIRTUAL TERMINAL: Terminal hiển thị giá trị.

Sơ đồ nối dây

- Sắp xếp các linh kiện cần sử dụng ra không gian làm việc gồm 1 vi điều khiển trung tâm, 1 Virtual Terminal và 6 LED đơn.
- Kết nối các LED đơn với vi điều khiển trung tâm:
  – Cathode của các LED đơn nối đến các chân tương ứng từ PA4 đến PA9 của vi điều khiển trung tâm theo cấu hình ở Hình 4.10.
  – Anode của các LED đơn được nối với nguồn điện 5V.
  – 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.
- Kết 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.
- Kết quả thu được thể hiện ở Hình 4.11.



Hình 4.11: Sơ đồ nối dây: Schematic

4.3 Hiện thực

4.3.1 Tổng quan ý tưởng

- Sử dụng header file scheduler.h và source file scheduler.c để quản lý trình lập lịch.
- Sử dụng một mảng với số phần tử tối đa SCH_TASKNUMBER nhằm lưu trữ thông tin tác vụ.

- Định nghĩa SCH_TIMERTICK là khoảng thời gian timer tick nhằm dễ dàng điều chỉnh khi thay đổi timer tick. Ngoài ra, với SCH_TIMERTICK, người lập trình ở các lớp ứng dụng phía trên có thể sử dụng thời gian theo đơn vị ms mà không quan tâm đến timer tick thật sự của hệ thống.

```
1 SCH_Task tasks[SCH_TASKNUMBER];
```

Chương trình 4.1: Hiện thực: Mảng tasks lưu trữ thông tin tác vụ

```
1 # define SCH_TASKNUMBER 8
2 # define SCH_TIMERTICK 10
```

Chương trình 4.3: Hiện thực: Cấu trúc dữ liệu SCH_Task

4.3.2 Cơ sở dữ liệu

- Định nghĩa cấu trúc dữ liệu SCH_Task nhằm quản lý thông tin của tác vụ.
- 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.

```
1 typedef struct {
2     void (*functionPointer)(void);
3     uint8_t id;
4     uint32_t delay;
5     uint32_t period;
6     unsigned char flag;
7 } SCH_Task;
```

Chương trình 4.2: Hiện thực: Các define được khai báo

4.3.3 Hàm khởi tạo

- Hàm SCH_Init thực hiện khởi tạo giá trị cho một mảng với số phần tử xác định.
- Thông số id được gán giá trị cụ thể nhằm thuận tiện cho hoạt động của giải thuật thêm và xóa ở các phần sau.
- Tất cả các thông số còn lại được gán giá trị bằng 0.

```
1 void SCH_Init(void) {
2     for (uint8_t i = 0; i < SCH_TASKNUMBER; i++) {
3         tasks[i].functionPointer = NULL;
4         tasks[i].id = SCH_TASKNUMBER - i - 1;
5         tasks[i].delay = 0;
6         tasks[i].period = 0;
7         tasks[i].flag = 0;
8     }
9 }
```

Chương trình 4.4: Hiện thực: Hàm SCH_Init

- Hình 4.15 mô tả giá trị id của một cơ sở dữ liệu có SCH_TASKNUMBER bằng 8 sau khi được khởi tạo.
- Giá trị id chỉ cần đảm bảo tính duy nhất. Nó có thể được chọn các số ngẫu nhiên, sắp xếp theo bất kỳ cách nào.

Hình 4.12: Sơ đồ nối dây: Schematic

4.3.4 Hàm cập nhật

- Hàm SCH_Update được gọi trong hàm ngắt timer, thực hiện cập nhật thời gian đợi còn lại của tác vụ trong hàng đợi.
- Kiểm tra phần tử đầu tiên của cơ sở dữ liệu, nếu tác vụ chưa được khởi tạo, bỏ qua câu lệnh. Ngược lại, cập nhật giá trị delay và flag phù hợp cho tác vụ.
- Vì kiểu dữ liệu của delay được khai báo là uint32_t, khi cập nhật phải tự kiểm soát vấn đề tràn số.
- Vì chỉ thực hiện kiểm tra duy nhất phần tử đầu tiên của cơ sở dữ liệu, hàm SCH_Update đảm bảo có độ phức tạp O(1).

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

Chương trình 4.5: Hiện thực: Hàm SCH_Update

4.3.5 Hàm thực thi

- Hàm SCH_Dispatch được gọi trong vòng lặp while(1), thực thi tác vụ đã sẵn sàng trong cơ sở dữ liệu
- Kiểm tra phần tử đầu tiên của cơ sở dữ liệu, nếu tác vụ chưa được khởi tạo (functionPointer == 0) hoặc chưa sẵn sáng thực thi (flag == 0), bỏ qua câu lệnh. Ngược lại, gọi hàm thực thi thông qua con trỏ hàm được lưu trữ.
- Sau khi thực thi tác vụ, xóa tác vụ khỏi cơ sở dữ liệu. Đồng thời, kiểm tra giá trị period. Nếu period có giá trị khác 0 (hàm thực hiện định kỳ), bổ sung tác vụ vào cơ sở dữ liệu với giá trị delay = period. Ngược lại, xác định tác vụ chỉ thực hiện một lần duy nhất và kết thúc.

```
void SCH_Dispatch(void) {
    if (tasks[0].functionPointer == 0 || tasks[0].flag == 0)
       return;
    (*tasks[0].functionPointer)();
    tasks[0].flag = 0;
    SCH_Task newTask = tasks[0];
    SCH_DeleteTask(tasks[0].id);
    if (newTask.period > 0) {
        SCH_AddTask(newTask.functionPointer, newTask.period,
           newTask.period);
    }
}
```

Chương trình 4.6: Hiện thực: Hàm SCH_Dispatch

4.3.6 Hàm thêm tác vụ

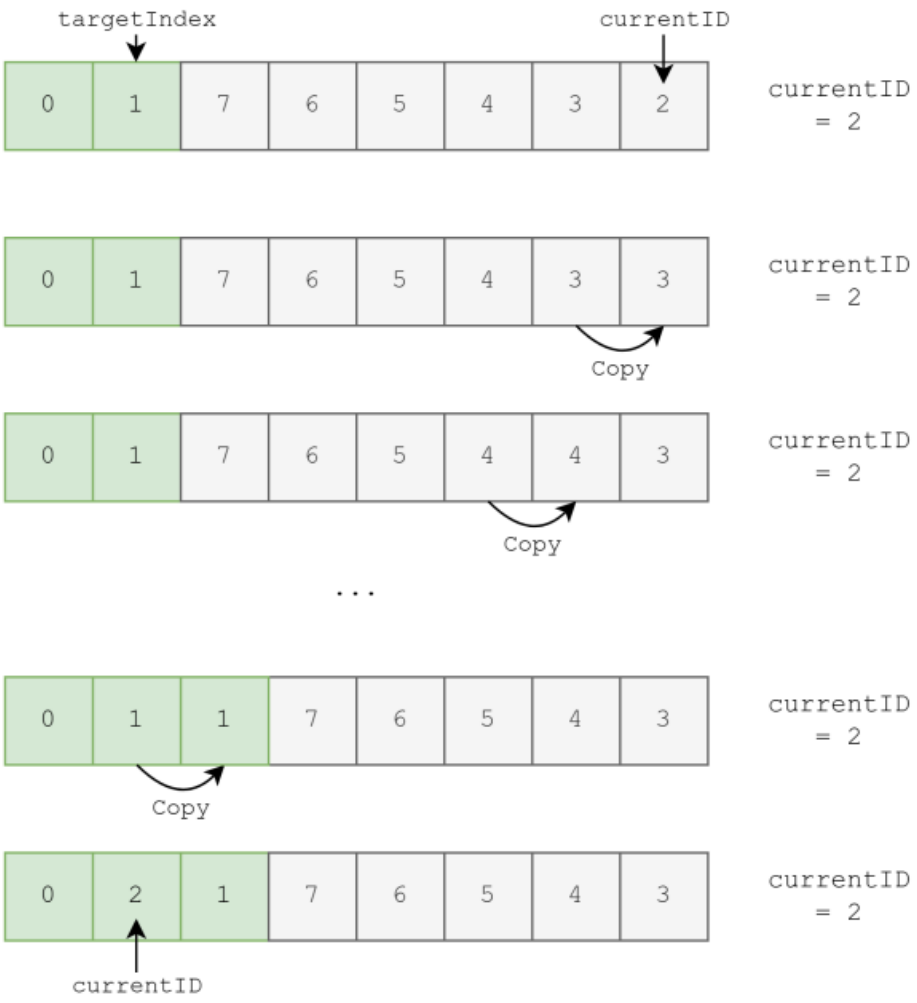
- Hàm SCH_AddTask thực hiện thêm tác vụ vào cơ sở dữ liệu.
- Kiểm tra phần tử cuối cùng của mảng nhằm xác định cơ sở dữ liệu đã đạt giới hạn tác vụ hay chưa. Nếu đã đạt giới hạn, trả về giá trị SCH_TASKNUMBER và kết thúc. Ngược lại, tiến hành tìm kiếm vị trí thích hợp và thêm tác vụ vào cơ sở dữ liệu.
- Quá trình tìm kiếm vị trí thích hợp và thêm tác vụ vào cơ sở dữ liệu được mô tả trực quan ở Hình 4.16, 4.17, 4.18.
- Hàm SCH_AddTask được hiện thực có độ phức tạp O(n), mỗi lần gọi lệnh đều bắt buộc thực hiện duyệt tất cả các phần tử của cơ sở dữ liệu và không có ngoại lệ.

```
1  uint8_t SCH_AddTask(void (*functionPointer)(void), uint32_t
       delay, uint32_t period) {
2      if (tasks[SCH_TASKNUMBER - 1].functionPointer != 0)
           return SCH_TASKNUMBER;
3      uint8_t currentID = tasks[SCH_TASKNUMBER - 1].id;
4      uint32_t currentDelay = 0;
5      for (uint8_t i = 0; i < SCH_TASKNUMBER; i++) {
6          currentDelay += tasks[i].delay;
7          if (currentDelay > delay || tasks[i].functionPointer
               == 0) {
8              for (uint8_t j = SCH_TASKNUMBER - 1; j > i; j--)
                   {
9                  tasks[j] = tasks[j - 1];
10             }
11             tasks[i].functionPointer = functionPointer;
12             tasks[i].id = currentID;
13             tasks[i].period = period;
14             tasks[i].flag = 0;
15             if (currentDelay > delay) {
16                 int newDelay = currentDelay - delay;
17                 tasks[i].delay = tasks[i + 1].delay -
                       newDelay;
18                 if (tasks[i].delay == 0) {
19                     tasks[i].flag = 1;
20                 }
21                 tasks[i + 1].delay = newDelay;
22                 if (tasks[i + 1].delay == 0) {
23                     tasks[i + 1].flag = 1;
24                 }
25             } else {
26                 tasks[i].delay = delay - currentDelay;
27                 if (tasks[i].delay == 0) {
28                     tasks[i].flag = 1;
29                 }
```

```
30                  }
31                  return tasks[i].id;
32              }
33          }
34          return SCH_TASKNUMBER;
35      }
```

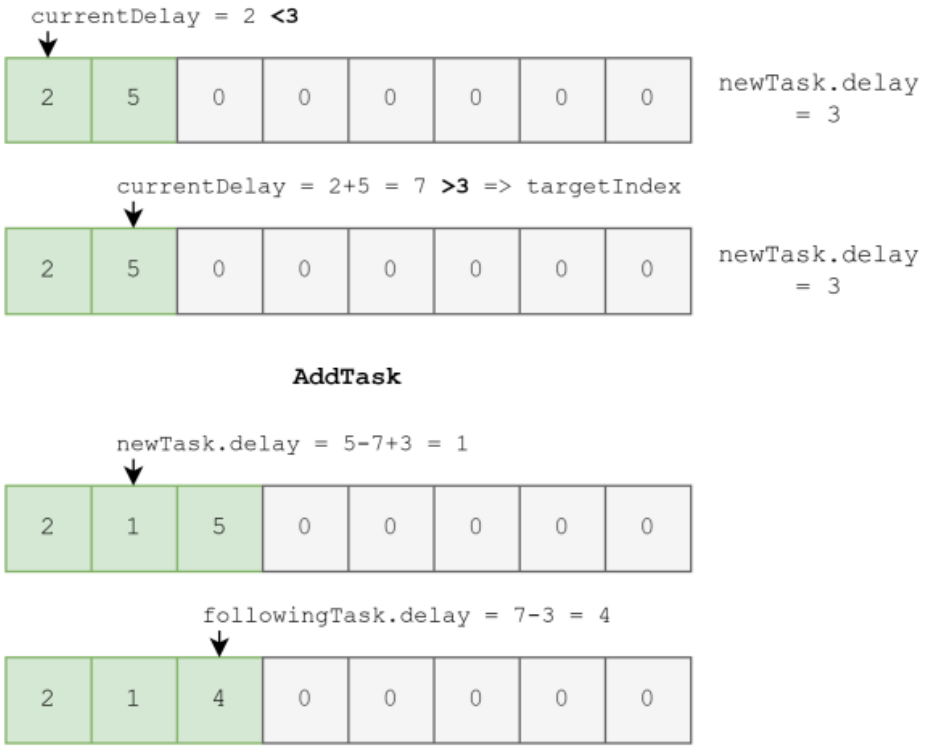
Chương trình 4.7: Hiện thực: Hàm SCH_AddTask

- Hình 4.16 thể hiện các bước của quá trình thêm tác vụ.
- Các phần tử màu xanh lục biểu thị tác vụ đã được khởi tạo; các phần tử màu trắng biểu thị tác vụ chưa được khởi tạo.
- Các ký tự số trên hình biểu thị giá trị id.
- Quá trình tìm kiếm vị trí thích hợp được thể hiện ở Hình 4.17 và 4.18.

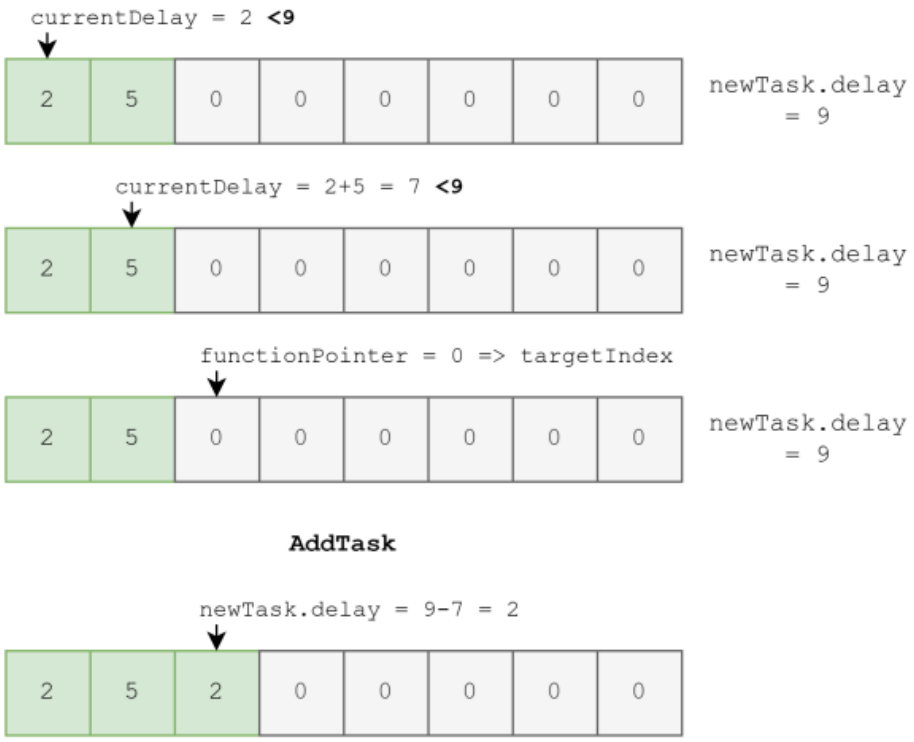


Hình 4.13: Hình 4.16: Hiện thực: Hàm thêm tác vụ

- Hình 4.17 và 4.18 mô tả 2 trường hợp cơ bản của quá trình tìm kiếm vị trí thích hợp cho tác vụ.
- Các phần tử màu xanh lục biểu thị tác vụ đã được khởi tạo; các phần tử màu trắng biểu thị tác vụ chưa được khởi tạo.
- Các ký tự số trên hình biểu thị giá trị delay.



Hình 4.14: Hình 4.17: Hiện thực: Hàm thêm tác vụ



Hình 4.15: Hình 4.18: Hiện thực: Hàm thêm tác vụ

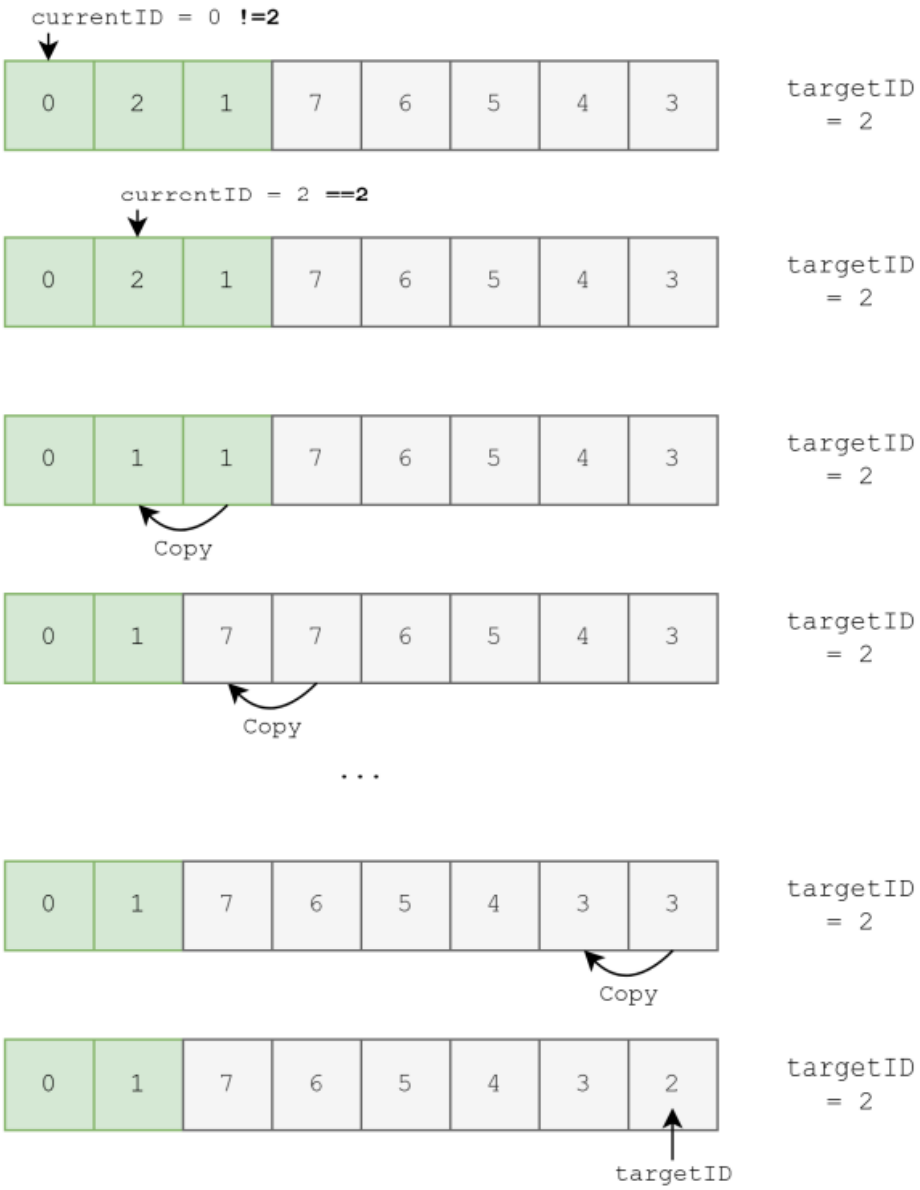
4.3.7 Hàm xóa tác vụ

- Hàm SCH_DeleteTask thực hiện xóa tác vụ khỏi cơ sở dữ liệu.
- Lần lượt duyệt qua các phần tử của cơ sở dữ liệu để tìm tác vụ có giá trị id tương ứng. Nếu có, thực hiện xóa tác vụ và trả về 1. Ngược lại, trả về 0 và kết thúc.
- Sau khi xóa, giá trị id được giữ lại để tái sử dụng cho những lần sau. Điều này vẫn đảm bảo được quy ước giá trị id là duy nhất ở một thời điểm xác định.
- Các giá trị còn lại đều được gán lại bằng 0 và sẵn sàn sử dụng ở những lần sau. Đặc biệt, giá trị functionPointer phải đảm bảo được đặt lại bằng 0, nếu không quá trình xóa tác vụ sẽ bị xem như thất bại.
- Quá trình tìm kiếm và xóa tác vụ được mô tả trực quan ở Hình 4.19.
- Hàm SCH_DeleteTask được hiện thực có độ phức tạp O(n), mỗi lần gọi lệnh đều bắt buộc thực hiện duyệt tất cả các phần tử của cơ sở dữ liệu và không có ngoại lệ.

```
1 unsigned char SCH_DeleteTask(uint8_t id) {
2     for (uint8_t i = 0; i < SCH_TASKNUMBER; i++) {
3         if (tasks[i].id == id) {
4             uint8_t currentID = tasks[i].id;
5             if (tasks[i + 1].functionPointer != 0) {
6                 tasks[i + 1].delay += tasks[i].delay;
7             }
```

```
8               for (uint8_t j = i; j < SCH_TASKNUMBER - 1; j++)
                    {
9                     tasks[j] = tasks[j + 1];
10              }
11              tasks[SCH_TASKNUMBER - 1].functionPointer = 0;
12              tasks[SCH_TASKNUMBER - 1].id = currentID;
13              tasks[SCH_TASKNUMBER - 1].delay = 0;
14              tasks[SCH_TASKNUMBER - 1].period = 0;
15              tasks[SCH_TASKNUMBER - 1].flag = 0;
16              return tasks[SCH_TASKNUMBER - 1].functionPointer
                    == 0;
17          }
18      }
19      return 0;
20  }
```

Chương trình 4.8: Hiện thực: Hàm SCH_DeleteTask



Hình 4.16: Hình 4.19: Hiện thực: Hàm xóa tác vụ

4.4 Mô phỏng

4.4.1 Điều khiển các LED đơn

- Sử dụng header file output.h và source file output.c để điều khiển các LED đơn.
- Vì mô phỏng chỉ yêu cầu các tác vụ thực hiện với chu kỳ xác định, chọn thực hiện nhấp nháy các LED đơn để dễ dàng quan sát.

```
1 #ifndef INC_OUTPUT_H_
2 #define INC_OUTPUT_H_
3
4 #include "main.h"
```

```
5
6  void ledRedToggle(void);
7  void ledYellowToggle(void);
8  void ledGreenToggle(void);
9  void ledAquaToggle(void);
10 void ledBlueToggle(void);
11 void ledPinkToggle(void);
12 #endif /* INC_OUTPUT_H_ */
```

Chương trình 4.9: Mô phỏng: Header file output.h

```
1  #include "output.h"
2
3  void ledRedToggle(void) {
4      HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
5  }
6
7  void ledYellowToggle(void) {
8      HAL_GPIO_TogglePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin);
9  }
10
11 void ledGreenToggle(void) {
12     HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
13 }
14
15 void ledAquaToggle(void) {
16     HAL_GPIO_TogglePin(LED_AQUA_GPIO_Port, LED_AQUA_Pin);
17 }
18
19 void ledBlueToggle(void) {
20     HAL_GPIO_TogglePin(LED_BLUE_GPIO_Port, LED_BLUE_Pin);
21 }
22
23 void ledPinkToggle(void) {
24     HAL_GPIO_TogglePin( LED_PINK_GPIO_Port , LED_PINK_Pin );
25 }
```

Chương trình 4.10: Mô phỏng: Source file output.c

4.4.2 Điều khiển Virtual Terminal

- Sử dụng chương trình mẫu được giới thiệu ở Lab5 để hiện thực chương trình đơn giản nhằm hiển thị giá trị ra Virtual Terminal thông qua giao tiếp UART..
- Các hàm liên quan được hiện thực trực tiếp trên source file main.c.

```c
/* USER CODE BEGIN 0 */
uint8_t temp = 0;

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if(huart->Instance == USART2) {
        HAL_UART_Receive_IT(&huart2, &temp, 1);
        HAL_UART_Transmit(&huart2, &temp, 1, 50);
    }
}

void timePrint(void) {
    char str[100];
    HAL_UART_Transmit(&huart2, (void*)str, sprintf(str, "%lu\
      r\n", HAL_GetTick()), 10);
}

/* USER CODE END 0 */
```

Chương trình 4.11: Mô phỏng: Điều khiển Virtual Terminal

4.4.3 Khởi tạo chương trình mô phỏng

- Thực hiện include các header file cần thiết vào source file main.c.
- Tiến hành các khởi tạo cần thiết cho timer, giao tiếp 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 1 giây, tác vụ thực hiện một lần.
  – Nhấp nháy LED-YELLOW sau mỗi 0.5 giây.
  – Nhấp nháy LED-GREEN sau mỗi 1 giây.
  – Nhấp nháy LED-AQUA sau mỗi 1.5 giây.
  – Nhấp nháy LED-BLUE sau mỗi 2 giây.
  – Nhấp nháy LED-PINK sau mỗi 2.5 giây.
  – In thời gian hiện tại ra Virtual Terminal sau mỗi 10ms.
  – In thời gian hiện tại ra Virtual Terminal sau mỗi 500ms.

```c
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include "scheduler.h"
#include "output.h"
/* USER CODE END Includes */
```

Chương trình 4.12: Mô phỏng: Các #include được khai báo

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

Chương trình 4.13: Mô phỏng: Các khởi tạo cần thực hiện

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

Chương trình 4.14: Mô phỏng: Hàm ngắt timer HAL_TIM_PeriodElapsedCallback

```
  while (1)
  {
    /* USER CODE END WHILE */
      SCH_Dispatch();
    /* USER CODE BEGIN 3 */
  }
```

Chương trình 4.15: Mô phỏng: Gọi hàm thực thi SCH_Dispatch

```
  /* USER CODE BEGIN WHILE */

  SCH_AddTask(ledRedToggle, 1000, 0);
  SCH_AddTask(ledYellowToggle, 0, 500);
  SCH_AddTask(ledGreenToggle, 0, 1000);
  SCH_AddTask(ledAquaToggle, 0, 1500);
  SCH_AddTask(ledBlueToggle, 0, 2000);
  SCH_AddTask(ledPinkToggle, 0, 2500);
  SCH_AddTask(timePrint, 0, 10);
  SCH_AddTask(timePrint, 0, 500);
```

Chương trình 4.16: Mô phỏng: Khởi tạo các tác vụ cần thiết

4.4.4 Kịch bản mô phỏng

- Thực hiện mô phỏng hành vi của các LED trong 5 giây kể từ lúc bắt đầu chương trình, ghi lại kết quả sau mỗi 0,5 giây. Kết quả mong muốn được thể hiện ở Bảng 4.1.

	LED-RED	LED-YELLOW	LED-GREEN	LED-AQUA	LED-BLUE	LED-PINK
0.0	Sáng	Tắt	Tắt	Tắt	Tắt	Tắt
0.5	Sáng	Sáng	Tắt	Tắt	Tắt	Tắt
1.0	Tắt	Tắt	Sáng	Tắt	Tắt	Tắt
1.5	Tắt	Sáng	Sáng	Sáng	Tắt	Tắt
2.0	Tắt	Tắt	Tắt	Sáng	Sáng	Tắt
2.5	Tắt	Sáng	Tắt	Sáng	Sáng	Sáng
3.0	Tắt	Tắt	Sáng	Tắt	Sáng	Sáng
3.5	Tắt	Sáng	Sáng	Tắt	Sáng	Sáng
4.0	Tắt	Tắt	Tắt	Tắt	Tắt	Sáng
4.5	Tắt	Sáng	Tắt	Sáng	Tắt	Sáng
5.0	Tắt	Tắt	Sáng	Sáng	Tắt	Tắt

Hình 4.17: Bảng 4.1: Kịch bản mô phỏng: Kết quả mong muốn

- Kiểm tra các kết quả in ta Virtual Terminal ứng với mỗi lần kiểm tra hành vi của đèn LED. Các giá trị phải cách nhau 10ms. Đồng thời, sau mỗi 500ms, phải có một dấu thời gian khác được in ra màn hình. 4.1.

4.5 Bài tập bổ sung cho LAB 3

- Thực hiện bổ sung thêm file scheduler.h và scheduler.c thực hiện ở LAB 4 vào LAB 3
- Sau đó thực hiện thay đổi đoạn mã trong hàm main.c như sau :

```
1 /* USER CODE BEGIN Includes */
2 #include "processing_fsm.h"
3 #include "software_timer.h"
4 #include "manual_fsm.h"
5 #include "scheduler.h"
6 /* USER CODE END Includes */
```

Chương trình 4.17: Thực hiệm thêm Includes

```
1   /* USER CODE BEGIN WHILE */
2   SCH_AddTask(led7segScanning,250,250);
3   SCH_AddTask(fsmProcessing,100,100);
4   SCH_AddTask(fsmManua,100,100);
5   while (1)
6   {
```

```
7       /* USER CODE END WHILE */
8         SCH_Dispatch();
9       /* USER CODE BEGIN 3 */
10    }
11    /* USER CODE END 3 */
12  }
```

Chương trình 4.18: Thực hiện trong hàm main

```
1  /* USER CODE BEGIN 4 */
2  void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
3      timerRun();
4      buttonReading();
5      SCH_Update();
6  }
7  /* USER CODE END 4 */
```

Chương trình 4.19: Thực hiện trong hàm ngắt