VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

REPORT
LAB 4

Class: Microprocessors - Microcontrollers – CC04

Lecturer: PHAN VĂN SỸ

No.	Full Name	ID Student
1.	Lê Mạnh Quân	2352993

Ho Chi Minh City, October 2025

Contents

1 Task Files: task.h và task.c

1.1 Code Implementation

```
// ===== task.h =====
#ifndef INC_TASK_H_
#define INC_TASK_H_
#include "main.h"
#include "stdint.h"
#include "schedule.h"
void task_white(void);
void task_red(void);
void task_yellow(void);
void task_green(void);
void task_pink(void
void task_led(void); //4 debug
#endif
```

Listing 1: File task.h

```
// ===== task.c =====
#include "main.h"
#include "tasks.h"
#include "string.h"
extern UART_HandleTypeDef huart1;
    int _write(int file, char *ptr, int len){
    HAL_UART_Transmit(&huart1, (uint8_t*)ptr, len, HAL_MAX_DELAY);
 return len;
}

void task_white(void){
    HAL_GPIO_TogglePin(led_white_GPIO_Port, led_white_Pin);
    printf("White running at: %lu ms\r\n", HAL_GetTick());
}
```

Listing 2: File task.c

1.2 Detailed Explanation of task.c

File task.c là nơi định nghĩa toàn bộ các tác vụ (task) được bộ lập lịch (scheduler) gọi và thực thi định kỳ. Mỗi task tương ứng với một chức năng cụ thể(điều khiển bật/tắt một LED hoặc gửi thông tin qua UART).

- 1. Hàm _write(): Đây là một hàm ghi đè (override) lại hàm xuất chuẩn trong thư viện C. Mục đích của nó là cho phép sử dụng printf() để gửi dữ liệu qua UART1.
  - Khi gọi printf(), dữ liệu sẽ được truyền đến HAL_UART_Transmit().
  - HAL_UART_Transmit() sẽ gửi dữ liệu đến máy tính thông qua cổng UART cấu hình ở chân PA9 (TX).

- 2. Các hàm task_white(), task_red(), task_yellow(), task_green,task_pink(): Mỗi task thực hiện việc bật/tắt LED tương ứng bằng hàm HAL_GPIO_TogglePin(), tức là đảo trạng thái LED mỗi khi task được gọi.
  - Ví dụ, task_red() sẽ đảo LED đỏ mỗi khi đến chu kỳ thực thi.
  - Sau khi đổi trạng thái LED, task còn gửi một chuỗi UART như "RED running at:
- 3. Biến HAL_GetTick(): Hàm HAL_GetTick() trả về số mili–giây kể từ khi chương trình bắt đầu chạy (tăng lên mỗi 1 ms nhờ SysTick timer). Giá trị này giúp ta biết chính xác thời điểm task được thực thi.
- 4. Tính chất định kỳ của task: Mỗi task được thêm vào scheduler với các thông số delay và period khác nhau:
  - Delay: thời gian chờ trước khi task chạy lần đầu tiên.
  - Period: chu kỳ lặp lại của task (tính bằng mili–giây theo tick timer).

  Khi đến thời điểm phù hợp, scheduler sẽ gọi các task này và đặt lại delay theo chu kỳ đã cấu hình.

2 Scheduler Files: schedule.h và schedule.c

```
#define SCH_MAX_TASKS 10
#define NO_ERROR 0
#define ERROR_SCH_INVALID_INDEX 101
#define ERROR_SCH_TOO_MANY_TASKS 102
#define ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK 103
#define ERROR_SCH_WAITING_FOR_START_CMD_MASTER 104
#define ERROR_SCH_ONE_OR_MORE_SLAVES_NOT_STARTED 105
#define ERROR_SCH_LOST_SLAVE 106
#define ERROR_SCH_CAN_BUS_ERROR 107
#define ERROR_I2C_WRITE_BYTE_AT24C64 108

typedef struct {
void (*funcPtr)(void);
uint32_t delay;
uint32_t period;
uint8_t runme;
bool isEmpty;
bool isOneShot;
} Tasks;

void SCH_Init(void);
uint32_t SCH_Add_Task(void (*task)(void), uint32_t delay, uint32_t
    period);
void SCH_Update(void);
void SCH_Dispatch_Tasks(void);
uint32_t SCH_Delete_Task(uint32_t taskID);
#endif
```

Listing 3: File schedule.h

2.1 Detailed Explanation of schedule.h

File schedule.h là phần khai báo (header file) cho bộ lập lịch (scheduler) — chịu trách nhiệm quản lý, lưu trữ và điều phối việc thực thi các tác vụ (task) trong hệ thống nhúng. Nó định nghĩa các hằng số, cấu trúc dữ liệu và nguyên mẫu hàm cần thiết cho toàn bộ cơ chế hoạt động của scheduler.

1. Các hằng số định nghĩa

   - #define SCH_MAX_TASKS 10: Giới hạn tối đa 10 tác vụ có thể hoạt động cùng lúc.
   - Các mã lỗi như:
     * ERROR_SCH_INVALID_INDEX: Truy cập sai chỉ số task.
     * ERROR_SCH_TOO_MANY_TASKS: Số lượng task vượt giới hạn.
     * Các lỗi khác (CAN, I2C...) được định nghĩa sẵn để mở rộng trong tương lai.

2. Cấu trúc dữ liệu Tasks

Giải thích các trường:

- funcPtr: Con trỏ tới hàm mà scheduler sẽ gọi để chạy task.
- delay: Khoảng thời gian (đơn vị tick) trước khi task chạy lần đầu.
- period: Nếu khác 0, task sẽ được chạy định kỳ; nếu bằng 0 thì chỉ chạy một lần.
- runme: Bộ đếm, khi tăng lên (do SCH_Update) thì task đã sẵn sàng thực thi.
- isOneShot: Đánh dấu task chỉ chạy một lần duy nhất (non-periodic).

3. Biến toàn cục

```
1 extern Tasks taskList[SCH_MAX_TASKS];
2 extern uint8_t currentTasks;
3 extern uint32_t ERROR_CODE_G;
```

Listing 4: Biến toàn cục chia sẻ giữa schedule.c và main.c

- taskList: Mảng lưu trữ toàn bộ các task đang hoạt động.
- currentTasks: Số lượng task hiện có.
- ERROR_CODE_G: Biến chứa mã lỗi toàn cục (nếu xảy ra).

4. Các nguyên mẫu hàm chính

   - SCH_Init(): Khởi tạo scheduler, dọn sạch danh sách task cũ.
   - SCH_Add_Task(): Thêm một task mới vào hệ thống.
   - SCH_Update(): Hàm được gọi định kỳ trong ngắt timer, giảm biến delay của từng task.
   - SCH_Dispatch_Tasks(): Gọi thực thi các task đã sẵn sàng (runme > 0).
   - SCH_Delete_Task(): Xóa một task khỏi danh sách.
   - SCH_Delete_All_Tasks(): Xóa toàn bộ các task đang hoạt động.

5. LOGIC

Scheduler vận hành theo nguyên lý đa nhiệm hợp tác (Cooperative Multitasking):

1. Mỗi task được thêm vào bằng SCH_Add_Task() với giá trị delay và period riêng.
2. Timer định kỳ gọi SCH_Update() để giảm thời gian delay.
3. Khi runme > 0, hàm SCH_Dispatch_Tasks() gọi task đó thực thi.
4. Sau khi chạy xong, nếu task là OneShot, nó bị xóa khỏi danh sách.

2.2 Detailed Explanation of schedule.c

File schedule.c chứa phần cài đặt chi tiết cho các hàm của bộ lập lịch (Scheduler), chịu trách nhiệm khởi tạo, thêm, cập nhật, thực thi và xóa các task trong chương trình. Phần dưới đây trình bày chi tiết từng hàm trong file này.

1.Hàm SCH_Init()

```
1  void SCH_Init(void) {
2      deleteAllTasks();
3      currentTasks = 0;
4
5      HAL_GPIO_WritePin(led_white_GPIO_Port, led_white_Pin,
           GPIO_PIN_SET);
6      HAL_GPIO_WritePin(led_red_GPIO_Port, led_red_Pin, GPIO_PIN_SET
           );
7      HAL_GPIO_WritePin(led_yellow_GPIO_Port, led_yellow_Pin,
           GPIO_PIN_SET);
8      HAL_GPIO_WritePin(led_green_GPIO_Port, led_green_Pin,
           GPIO_PIN_SET);
9      HAL_GPIO_WritePin(led_pink_GPIO_Port, led_pink_Pin,
           GPIO_PIN_SET);
10     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
```

```
}
```

Listing 5: Hàm SCH_Init()

Giải thích: Hàm này được gọi đầu tiên khi khởi động hệ thống. Nó xoá toàn bộ các task cũ (gọi hàm deleteAllTasks()), đặt lại bộ đếm currentTasks = 0, và đồng thời bật tất cả các LED ở trạng thái ban đầu (logic 1). Mục đích là giúp hệ thống bắt đầu từ một trạng thái sạch và dễ quan sát khi debug.

2.Hàm SCH_Add_Task()

```
uint32_t SCH_Add_Task(void (*task)(void), uint32_t delay, uint32_t period) {
    if (currentTasks == SCH_MAX_TASKS) {
        ERROR_CODE_G = ERROR_SCH_TOO_MANY_TASKS;
        return ERROR_CODE_G;
    }

    taskList[currentTasks].funcPtr = task;
    taskList[currentTasks].delay = delay;
    taskList[currentTasks].period = period;
    taskList[currentTasks].runme = 0;
    taskList[currentTasks].taskID = currentTasks;
    taskList[currentTasks].isEmpty = false;
    taskList[currentTasks].isOneShot = (period == 0);

    currentTasks++;
    return currentTasks;
}
```

Listing 6: Hàm SCH_Add_Task()

Giải thích: Hàm này thêm một task mới vào danh sách taskList. - Nếu số lượng task đã đạt giới hạn SCH_MAX_TASKS, hệ thống trả về mã lỗi. - Các tham số: - task: con trỏ đến hàm sẽ thực thi. - delay: thời gian trễ trước khi chạy lần đầu. - period: chu kỳ lặp lại (nếu bằng 0 → chỉ chạy một lần). Sau khi thêm thành công, bộ đếm currentTasks được tăng lên 1.

3.Hàm SCH_Update()

```
void SCH_Update(void) {
    for (int i = 0; i < SCH_MAX_TASKS; i++) {
        if (!taskList[i].isEmpty) {
            if (taskList[i].delay == 0) {
                taskList[i].runme++;
                taskList[i].delay = taskList[i].period;
            } else {
                taskList[i].delay--;
            }
        }
    }
```

```
}
```

Listing 7: Hàm SCH_Update()

Giải thích: Hàm này được gọi định kỳ bởi ngắt timer (TIM2). Nó duyệt qua toàn bộ danh sách task: - Nếu delay $> 0 \rightarrow$ giảm delay đi 1 mỗi chu kỳ. - Nếu delay $== 0 \rightarrow$ tăng cờ runme, báo hiệu task đã sẵn sàng thực thi. Đây là "bộ đếm thời gian" giúp lập lịch các task mà không cần dùng hàm delay().

4.Hàm SCH_Dispatch_Tasks()

```
void SCH_Dispatch_Tasks(void) {
    for (int i = 0; i < SCH_MAX_TASKS; i++) {
        if (!taskList[i].isEmpty && taskList[i].runme > 0) {
            taskList[i].runme--;
            (*taskList[i].funcPtr)();

            if (taskList[i].isOneShot) {
                SCH_Delete_Task(taskList[i].taskID);
            }
        }
    }
}
```

Listing 8: Hàm SCH_Dispatch_Tasks()

Giải thích: Hàm này được gọi trong vòng lặp while(1) của main.c. Nó duyệt toàn bộ danh sách task: - Nếu runme $> 0 \rightarrow$ gọi hàm được trỏ bởi funcPtr. - Sau khi chạy, giảm runme–. - Nếu task là loại isOneShot, task đó sẽ bị xoá khỏi danh sách.

5.Hàm SCH_Delete_Task()

```
uint32_t SCH_Delete_Task(uint32_t taskID) {
    if (taskID >= SCH_MAX_TASKS) {
        ERROR_CODE_G = ERROR_SCH_INVALID_INDEX;
        return ERROR_CODE_G;
    }

    if (taskList[taskID].isEmpty || taskList[taskID].funcPtr ==
        NULL) {
        ERROR_CODE_G = ERROR_SCH_INVALID_INDEX;
        return ERROR_CODE_G;
    }

    for (int i = taskID; i < SCH_MAX_TASKS - 1; i++) {
        taskList[i] = taskList[i + 1];
        taskList[i].taskID = i;
    }

    taskList[SCH_MAX_TASKS - 1].funcPtr = NULL;
    taskList[SCH_MAX_TASKS - 1].delay = 0;
    taskList[SCH_MAX_TASKS - 1].period = 0;
```

```
    taskList[SCH_MAX_TASKS - 1].runme = 0;
    taskList[SCH_MAX_TASKS - 1].taskID = 0;
    taskList[SCH_MAX_TASKS - 1].isEmpty = true;
    taskList[SCH_MAX_TASKS - 1].isOneShot = false;

    if (currentTasks > 0) currentTasks--;
    return taskID;
}
```

Listing 9: Hàm SCH_Delete_Task()

Giải thích: Hàm này xóa một task dựa theo taskID. Sau khi xoá, các task sau nó trong danh sách được "dịch trái" để lấp chỗ trống, và ID được cập nhật lại. Cuối cùng, slot cuối cùng được reset về trạng thái rỗng.

6.Hàm deleteAllTasks()

```
void deleteAllTasks(void) {
    for (int i = 0; i < SCH_MAX_TASKS; i++) {
        taskList[i].funcPtr = NULL;
        taskList[i].delay = 0;
        taskList[i].period = 0;
        taskList[i].runme = 0;
        taskList[i].taskID = 0;
        taskList[i].isEmpty = true;
        taskList[i].isOneShot = false;
    }
    currentTasks = 0;
}
```

Listing 10: Hàm deleteAllTasks()

Giải thích: Hàm này dùng để xóa toàn bộ các task trong scheduler. Mỗi phần tử trong mảng taskList[] được đặt về giá trị mặc định. Thường được gọi khi hệ thống reset hoặc khi cần khởi tạo lại toàn bộ tiến trình.

3 Main.c

3.1 Khởi tạo Scheduler và thêm các Task

```
MX_USART1_UART_Init();
HAL_TIM_Base_Start_IT(&htim2);
SCH_Init();
SCH_Add_Task(task_white, 20, 50);
SCH_Add_Task(task_red, 100, 100);
SCH_Add_Task(task_yellow, 50, 150);
SCH_Add_Task(task_green, 0, 200);
SCH_Add_Task(task_pink, 200, 250);
```

Listing 11: Khởi tạo bộ lập lịch và thêm các task

Giải thích:

- MX_USART1_UART_Init() khởi tạo UART1 để giao tiếp với Virtual Terminal trong Proteus.
- HAL_TIM_Base_Start_IT(htim2) khởi động Timer2 ở chế độ ngắt định kỳ (mỗi 10ms).
- SCH_Init() khởi tạo bộ lập lịch, xóa toàn bộ task cũ.
- SCH_Add_Task() thêm từng task vào danh sách với:
  * Tham số thứ nhất: con trỏ hàm (task).
  * Tham số thứ hai: delay — thời gian chờ trước lần chạy đầu tiên.
  * Tham số thứ ba: period — chu kỳ lặp lại của task (0 nếu chỉ chạy một lần).

3.2 Hàm ngắt Timer

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
SCH_Update();
}
```

Listing 12: Hàm xử lý ngắt Timer định kỳ

Giải thích:

- Hàm này được gọi tự động mỗi khi Timer2 tràn (mỗi 10ms).
- SCH_Update() giảm biến delay của từng task; khi delay = 0, task đó được đánh dấu sẵn sàng chạy bằng runme++.
- Cơ chế này giúp tất cả các task hoạt động đúng chu kỳ mà không cần dùng delay thủ công.

3.3 Vòng lặp chính (Main Loop)

```
while (1) {
SCH_Dispatch_Tasks();
}
```

Listing 13: Vòng lặp chính xử lý các task

Giải thích:

- SCH_Dispatch_Tasks() kiểm tra danh sách task — nếu task nào có runme > 0, nó sẽ được gọi để thực thi.
- Sau khi chạy xong, biến runme của task giảm đi 1.
- Nếu task là loại one-shot (chỉ chạy 1 lần), nó được xóa khỏi danh sách.
- Cơ chế này đảm bảo CPU luôn chạy theo lịch định trước, không bị treo do delay() hoặc HAL_Delay().

4 ẢNH

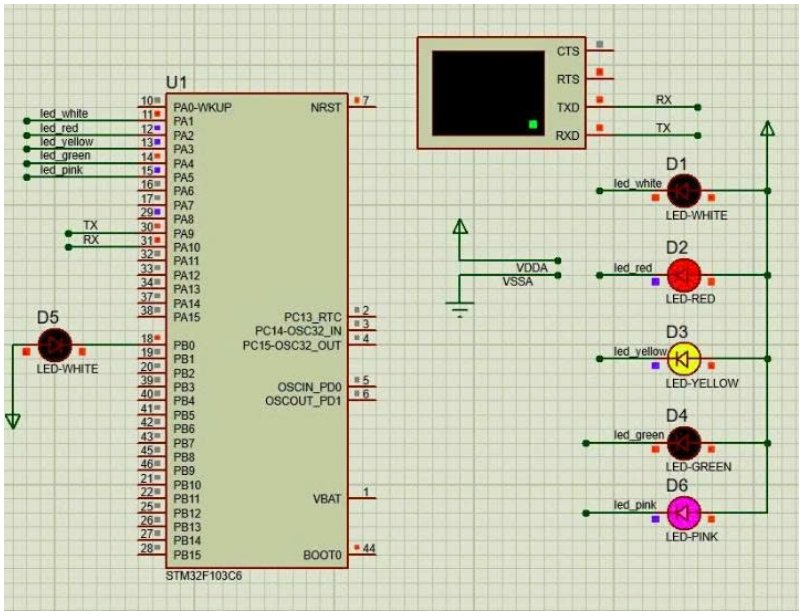

Figure 1: Mô phỏng mạch LED và UART trên Proteus

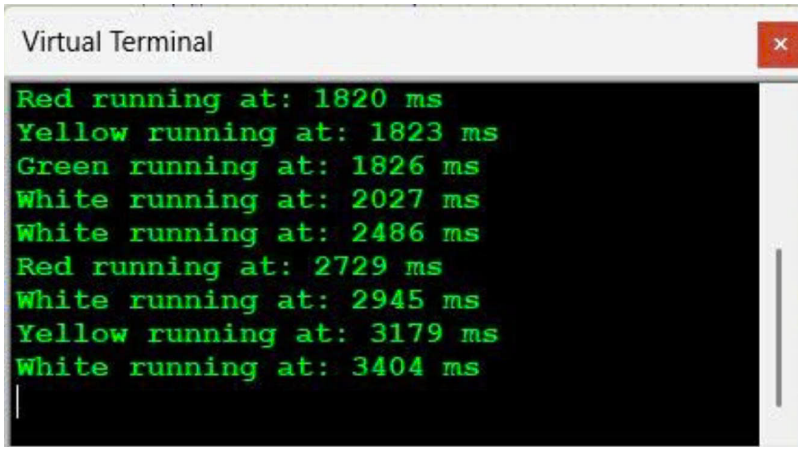

Figure 2: Kết quả hiển thị UART trên Virtual Terminal

5 Source

GG Drive Link: My Source Code Github Link: My Source Code