

Họ và tên	Phạm Quốc Tiến	Nguyễn Đức Trung	Bùi Thái Toàn
MSSV	22521472	22521564	22521485
Họ và tên	Trịnh Thành Trung	Trần Quốc Trinh	Nhật Tin
MSSV	22521570	22521542	22521479
Họ và tên	Nguyễn Hữu Bảo Trọng		
MSSV	22521545		

LAB 4: RTOS trên STM43F429

1 CHUẨN BỊ

1.1 Phần cứng

- KIT STM32F4 Discovery for STM32F429 MCU.

1.2 Phần mềm

- STM32CubeIDE: sử dụng để lập trình, build, nạp và debug code.

1.3 Kiến thức

- Biết cách tạo project và cấu hình project sử dụng STM32CubeIDE
- Có kiến thức vững về Hệ điều hành, đặc biệt là vấn đề lập lịch và đồng bộ các tiến trình



Hình 1.1. KIT STM32F4 Discovery

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	1
---	---

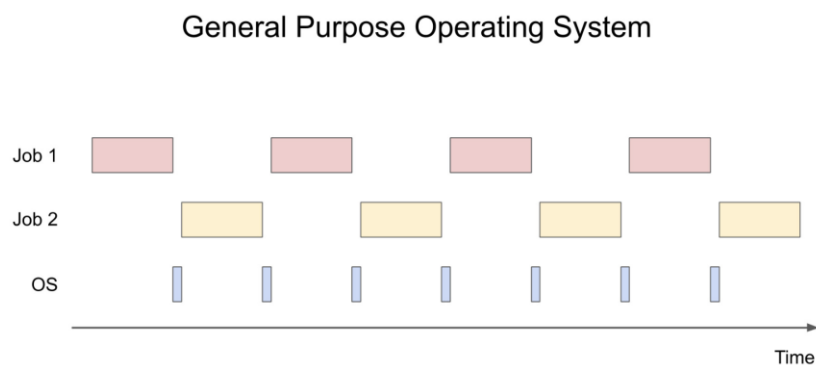
2 HƯỚNG DẪN

2.1 Real-time Operating System (RTOS) là gì?

Để có thể làm việc với RTOS, trước tiên chúng ta cần hiểu đây là gì và sự khác biệt giữa một hệ điều hành thông thường và hệ điều hành thời gian thực là như thế nào?

Hệ điều hành thời gian thực - RTOS, là một phần mềm giúp quản lý phần cứng và các phần mềm khác bên trong một hệ thống máy tính. Chúng ta đã khá quen thuộc với một số hệ điều hành đa dụng trên thế giới như Windows, Linux, OSX hay iOS và Android. Một hệ điều hành đa dụng (general purpose OS) sẽ thường được thiết kế tập trung vào trải nghiệm của người dùng.

Nhắc lại kiến thức về hệ đa dụng, ví dụ, khi chúng ta phát triển một phần mềm cho một hệ điều hành trên điện thoại, như Android hoặc iOS chẳng hạn. Người dùng có thể sẽ muốn stream một bộ phim, chúng ta có thể chia quá trình stream bộ phim này thành 2 công việc: (1) là tải từng đoạn video về từ Internet và (2) chiếu từng đoạn video đó cho người dùng xem. 2 công việc này là một phần của chương trình lớn và có thể được hiện thực đồng thời với nhau thông qua các tiểu trình (thread). Trong trường hợp như máy tính của chúng ta chỉ có 1 core CPU, như vậy, phần mềm stream phim sẽ phải luân phiên thực hiện liên tục công việc (1) và công việc (2), do tốc độ xử lý của CPU là rất cao vì vậy sẽ đem đến cho người dùng cảm giác như họ đang vừa load và vừa xem phim cùng lúc.



Hình 2.1. Hai công việc được hệ điều hành thực hiện luân phiên liên tục với nhau.

Hình 2.1 mô tả cách hệ điều hành chuyển đổi luân phiên liên tục giữa 2 công việc, hay còn có tên gọi khác là tiến trình hay tiểu trình. Lưu ý rằng, trong quá trình này còn xuất hiện một công việc thứ 3 chính là công việc của hệ điều hành thực hiện "chuyển

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	2

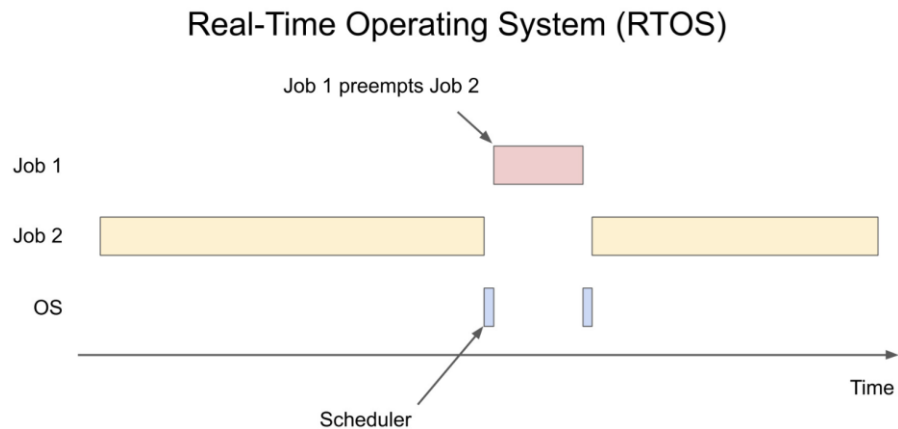
ngữ cảnh" (switch context). Chuyển ngữ cảnh là công việc lưu lại PCB - chứa thông tin trạng thái - của tiến trình cũ, và nạp PCB của tiến trình mới vào bộ nhớ chính cũng như các thanh ghi.

Trong quá trình thực hiện các công việc, có thể sẽ xảy ra độ trễ, một gói tin có thể mất trong quá trình tải video, một khung hình có thể bị rớt trong quá trình chiếu video để đảm bảo video có thể bắt kịp tốc độ trình chiếu. Trọng tâm của các hệ điều hành đa dụng sẽ tập trung vào trải nghiệm của người dùng, những vấn đề nhỏ nhất trên trong quá trình thực thi tiến trình thường sẽ không gây quá nhiều sự chú ý đối với người dùng, hay nói cách khác, người dùng có thể dễ dàng chấp nhận, hoặc không quan tâm đến những lỗi nho nhỏ đó.

Tuy nhiên, trọng tâm của hầu hết các chương trình trên các vi điều khiển lại khác. Việc đáp ứng các hạn định thời một cách nghiêm ngặt là một vấn đề hết sức quan trọng. Hãy nghĩ đến một bộ điều khiển động cơ hay một hệ thống phanh tự động trong một chiếc xe điện. Trong trường hợp này, nếu như việc xử lý định thời bị trễ dù chỉ vài mili giây thì tính mạng con người cũng có thể gặp nguy hiểm. Do đó, RTOS chính là sự lựa chọn tốt nhất để quản lý các công việc được thực thi đồng thời với nhau và phải đảm bảo tính thời gian thực.

Trong ví dụ trên, một bộ điều khiển điện tử (electronic control unit - ECU) được gắn trên xe có thể chịu trách nhiệm việc kiểm soát và hỗ trợ phanh xe. Quan sát Hình 2.2, công việc (2) thực hiện việc giám sát các input được ghi nhận từ tài xế, từ đó giúp đạp và bật đèn hậu. Tuy nhiên, giả sử rằng ECU của chúng ta nhận được thông báo từ các cảm biến phát hiện một vụ va chạm sắp xảy ra, khi đó, công việc (1) sẽ trung dụng CPU từ công việc (2) và điều khiển thắng xe. Tất nhiên, tất cả điều này giả định rằng đây là một chiếc xe được trang bị hỗ trợ phanh tự động.

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	3



Hình 2.2. Hai công việc được thực thi trong RTOS

Lưu ý rằng đây là một ví dụ hết sức đơn giản. Chúng ta cũng có thể sử dụng các ngắt phần cứng để trung dụng chương trình đang thực thi. Dù vậy, một RTOS có thể cho phép chúng ta tạo ra các tác vụ phần mềm thay vì phụ thuộc vào các ngắt phần cứng và gán độ ưu tiên cho các tác vụ này. Ngoài ra, hầu hết các RTOS đều có thể hoạt động như một lớp trừu tượng (abstraction layer), cho phép chúng ta viết code và có thể dễ dàng triển khai trên các vi điều khiển khác.

Tìm hiểu và trình bày những khác biệt giữa hệ điều hành đa dụng và hệ điều hành thời gian thực?

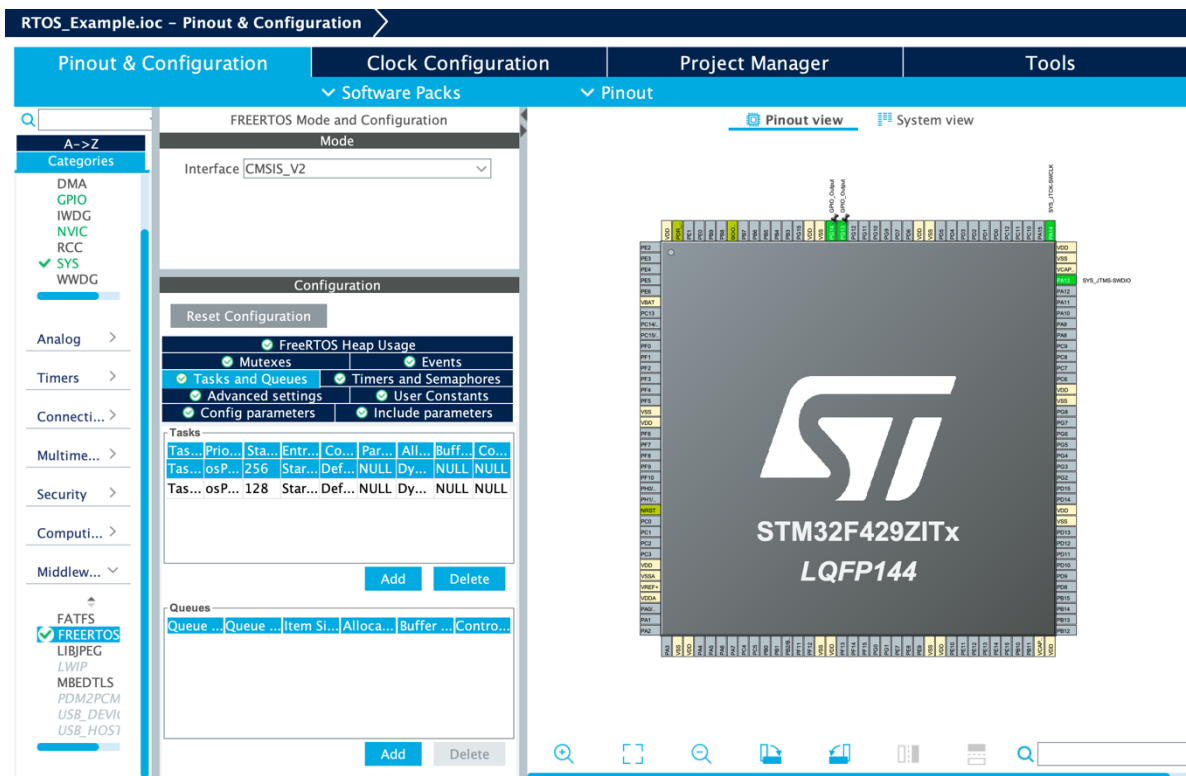
Đặc điểm	GPOS	RTOS
Độ chính xác thời gian	Không yêu cầu	Rất quan trọng
Lập lịch	Công bằng hoặc hiệu suất	Ưu tiên nghiêm ngặt
Thời gian đáp ứng	Không đảm bảo	Đảm bảo đúng thời gian
Ngành ứng dụng	Máy tính, smartphone	Hệ thống nhúng, IoT, công nghiệp

2.2 Lập trình với RTOS trên STM32F429

Trong phần này, chúng ta sẽ thực hiện lại việc lập trình cho LED 3 và LED 4 cùng nhấp nháy với chu kỳ khác nhau dựa trên việc tạo ra tác vụ thực thi đồng thời trong RTOS trên STM32F429.

Cấu hình Project:

Trước tiên, chúng ta cần tạo một project trên STM32CubeIDE và thực hiện cấu hình như sau:



Hình 2.3. Kích hoạt RTOS trên KIT

- Trong mục Middlewares, chọn FREERTOS, trong mục Interface chọn CMSIS_V2
- Trong phần Configurations cho FREERTOS, chọn tab Tasks and Queues:
 - Double-click vào task mặc định và thay đổi thông tin cho Task như bên dưới. Trong này, ta đặc biệt chú ý đến trường Priority sẽ cài đặt độ ưu tiên cho task, trường Stack Size cài đặt kích thước vùng Stack


LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	5
---	---

(dùng để chứa các biến cục bộ), trường Entry Function xác định tên hàm chứa code cụ thể cho task.

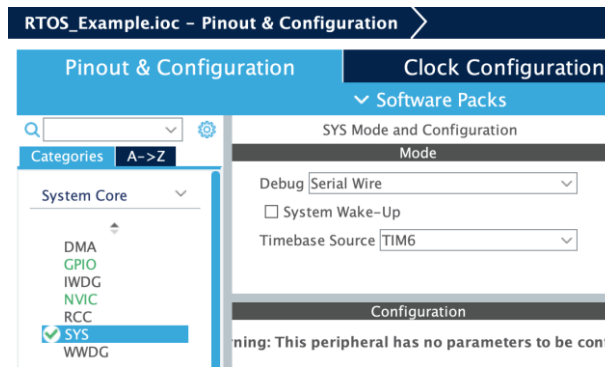
Hình 2.4. Cài đặt Task01

- Tiếp theo, cũng trong tab Tasks and Queues, bấm nút Add để tạo thêm một task mới có cấu hình như bên dưới.

Hình 2.5. Cài đặt Task02

Tiếp theo, chúng ta cần phải quan tâm đến SysTick. SysTick là một timer đặc biệt trong hầu hết các bộ xử lý ARM và thường được dùng cho các mục đích của hệ điều hành. Mặc định, SysTick trong một chip STM32 sẽ kích hoạt một ngắt mỗi 1ms. Nếu chúng ta đang sử dụng framework STM32 HAL, mặc định, SysTick sẽ được dùng cho các hàm như HAL_Delay() (đã đề cập ở LAB 01) và HAL_GetTick(). Do đó, STM32 HAL framework cài đặt độ ưu tiên rất cao cho SysTick. Tuy nhiên, FreeRTOS cần dùng SysTick cho bộ định thời của nó, và nó yêu cầu SysTick ở một mức độ ưu tiên thấp hơn nhiều. Chúng ta có vài cách để xử lý vấn đề xung đột này, nhưng dễ nhất chính là gán một timer khác, chưa được sử dụng để làm timebase source cho HAL. Timer 6 và 7 trong hầu hết các vi điều khiển STM32 thường rất là cơ bản, do đó nó có thể được dễ dàng sử dụng để làm timebase source cho HAL. Để thực hiện thao tác này, trong mục System Core  SYS, trong mục Timebase Source chọn TIM6.

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	6



Hình 2.6. Thay đổi Timebase Source cho HAL

Sau cùng của bước cấu hình project, chọn File Save để lưu và sinh code.

Viết các task/tiểu trình thực thi đồng thời:

Sau khi code được tự động sinh ra, quan sát file main.c, ta thấy trong phần Includes, thư viện "cmsis_os.h" được thêm vào để làm một lớp trừu tượng giúp chúng ta giao tiếp và làm việc với nhân RTOS được dễ dàng.

```
/* Includes ----- */
#include "main.h"
#include "cmsis_os.h"
```

Tiếp tục quan sát, trong phần định nghĩa hai task, ta có:

```
/* Definitions for Task01 */
osThreadId_t Task01Handle;
const osThreadAttr_t Task01_attributes = {
    .name = "Task01",
    .stack_size = 256 * 4,
    .priority = (osPriority_t) osPriorityNormal,
};
/* Definitions for Task02 */
osThreadId_t Task02Handle;
const osThreadAttr_t Task02_attributes = {
    .name = "Task02",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityNormal,
};
```

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	7

Dễ thấy, các thông số mà ta tạo ra cho tác vụ Task01 và Task02 trong giao diện cấu hình trước đó được định nghĩa tại đoạn code này. Tiếp tục, ta sẽ thấy các hàm sau lần lượt được gọi:

- **osKernelInitialize()**: Khởi tạo nhân OS
- **Task01Handle = osThreadNew(StartTask01, NULL, &Task01_attributes)**: Xác định công việc cho Task01
- **Task02Handle = osThreadNew(StartTask02, NULL, &Task02_attributes)**: Xác định công việc cho Task02
- **osKernelStart()**: Khởi chạy nhân OS. Lưu ý, khi hàm này được gọi, ngay lập tức, Task01 và Task02 ở trên sẽ được chạy. Công việc trong Task01 và Task02 thường là vòng lặp vô tận và được thực thi đồng thời với nhau. Do đó, ta sẽ không để bất kỳ đoạn code nào khác sau osKernelStart() trong hàm main(), bởi một cách lý tưởng, chương trình sẽ không còn return về main() sau khi osKernelStart() chạy mà chỉ thực thi Task01 và Task02 theo đúng thiết kế của chúng ta.

Lúc này, chương trình của chúng ta sẽ có 2 tác vụ được thi đồng thời là Task01 và Task02 với các công việc được xác định trong hàm StartTask01 và StartTask02. Các công việc này là liên tục và không bao giờ dừng. Bên cạnh đó, sẽ có một tác vụ định thời được chạy nền để thực hiện công việc chuyển ngữ cảnh giữa các task trên.

Như đã nói ở trên, mục đích của chúng ta là nhấp nháy LED 3 và LED 4, chúng ta sẽ thực hiện việc này lần lượt trong hàm StartTask01 và StartTask02:

```
void StartTask01(void *argument)
{
    /* USER CODE BEGIN 5 */
    /* Infinite loop */
    for(;;)
    {
        HAL_GPIO_TogglePin(GPIOG, GPIO_PIN_13);
        osDelay(1000);
    }
    /* USER CODE END 5 */
}
```

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	8


```

}

void StartTask02(void *argument)
{
    /* USER CODE BEGIN StartTask02 */
    /* Infinite loop */
    for(;;)
    {
        HAL_GPIO_TogglePin(GPIOG, GPIO_PIN_14);
        osDelay(2000);
    }
    /* USER CODE END StartTask02 */
}

```

Lưu ý, ở đây thay vì sử dụng hàm HAL_Delay() như trong LAB 01, chúng ta sẽ sử dụng hàm osDelay(). Lý do là vì hàm HAL_Delay() có độ ưu tiên rất cao và có thể sẽ can thiệp vào tiến trình, ngăn cản việc chuyển ngữ cảnh của hệ điều hành. Đồng thời nó cũng sẽ ngăn cản bộ định thời nghỉ ngơi, làm cho hệ thống không thể tiết kiệm năng lượng. Do đó, chúng ta sử dụng hàm osDelay() để báo với bộ định thời rằng nó có thể chuyển sang Task khác trong khi chúng ta chờ đợi.

3 BÀI TẬP

Bài tập 1: Không sử dụng FreeRTOS, hãy thiết kế một hệ thống đọc dữ liệu từ Gyroscope, sau đó gửi cho máy tính thông qua Virtual Com Port, đồng thời hiện hướng của Gyroscope lên màn hình LCD như mô tả trong Hình 3.1.

Nhóm sẽ setup như những lab trước đã thực hiện

Khai báo biến toàn cục

```

47
48 /* USER CODE BEGIN PV */
49 char text[50];
50 float pData[3];
51 /* USER CODE END PV */

```

Setup LCD, Gyroscope

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	9

```

96  BSP_GYRO_Init();
97  BSP_LCD_Init();
98  BSP_LCD_LayerDefaultInit(1, SDRAM_DEVICE_ADDR);
99  BSP_LCD_SelectLayer(1);
100 BSP_LCD_DisplayOn();
101 BSP_LCD_Clear(LCD_COLOR_BLUE);
102 BSP_LCD_SetBackColor(LCD_COLOR_BLUE);
103 BSP_LCD_SetTextColor(LCD_COLOR_WHITE);

```

Trong vòng while nhóm sẽ thực hiện đọc giá trị mà gyroscope trả về, chuyển đổi dữ liệu và sau đó sẽ gửi ba tham số XYZ đã chuyển đổi qua USB

```

112 while (1)
113 {
114     BSP_GYRO_GetXYZ(pData);
115     pData[0] = pData[0] * 17.5 * 0.001;
116     pData[1] = pData[1] * 17.5 * 0.001;
117     pData[2] = pData[2] * 17.5 * 0.001;
118     uint16_t data_len = sprintf(text, "%d %d %d\n", (int16_t)pData[0], (int16_t)pData[1], (int16_t)pData[2]);
119     CDC_Transmit_HS((uint8_t*)text, data_len);

```

Lý do mà nhóm thực hiện nhân $17.5 * 0.0001$ cho các giá trị XYZ là để đưa các giá trị về tốc độ góc. Theo datasheet thì mỗi đơn vị sẽ được quy đổi thành mdps (Millidegrees per Second) theo giá trị full scale (fs) trong sensitivity tương ứng và khi đọc trong hàm BSP_GYRO_Init() thì nhóm thấy thư viện sử dụng FS = 500 dps

So	Sensitivity	FS = 250 dps		8.75		mdps/digit
		FS = 500 dps		17.50		
		FS = 2000 dps		70		

```

98 Gyro_InitStructure.Endianness = L3GD20_BLE_LSB;
99 Gyro_InitStructure.Full_Scale = L3GD20_FULLSCALE_500;

```

Sau đó nhóm sẽ thực hiện so sánh các giá XYZ và xuất ra các trường hợp mũ tên tương ứng. Do gyroscope trên kit khá nhiều nên giá trị trong lệnh điều kiện nhóm chỉ lấy giá trị tương đối

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	10

```

120     if(pData[0] > 1000 && pData[1] > 1000)
121     {
122         BSP_LCD_Clear(LCD_COLOR_BLUE);
123         BSP_LCD_FillTriangle(238, 215, 215, 163, 140, 186); //phai
124         BSP_LCD_FillTriangle(121, 98, 144, 318, 295, 295); //duoi
125     }
126     else if(pData[0] < -1000 && pData[1] < -1000)
127     {
128         BSP_LCD_Clear(LCD_COLOR_BLUE);
129         BSP_LCD_FillTriangle(1, 24, 24, 163, 140, 186); //trai
130         BSP_LCD_FillTriangle(121, 98, 144, 1, 24, 24); //tren
131     }
132     else if(pData[0] < -1000 && pData[1] > 100)
133     {
134         BSP_LCD_Clear(LCD_COLOR_BLUE);
135         BSP_LCD_FillTriangle(238, 215, 215, 163, 140, 186); //phai
136         BSP_LCD_FillTriangle(121, 98, 144, 1, 24, 24); //tren
137     }
138     else if(pData[0] > 1000 && pData[1] < -100)
139     {
140         BSP_LCD_Clear(LCD_COLOR_BLUE);
141         BSP_LCD_FillTriangle(1, 24, 24, 163, 140, 186); //trai
142         BSP_LCD_FillTriangle(121, 98, 144, 318, 295, 295); //duoi
143     }

```

Sau đó nhóm sẽ thực hiện delay 100 ms

```

143     }
144     HAL_Delay(100);

```

Bài tập 2: Thực hiện lại bài tập trên nhưng có sử dụng FreeRTOS.

Với ý tưởng và giải thuật như trên bài tập 1, nhóm sẽ tách quá trình xử lý trong vòng while tại hàm main ra hai task độc lập. Với Task01 sẽ xử lý việc đọc, convert dữ liệu và gửi data qua USB và Task02 sẽ là giải thuật xuất lên LCD mỗi tên tương ứng. Do Task01 sẽ xử lý nhiều hơn nên nhóm sẽ cấp bộ nhớ stack nhiều hơn Task02

Khai báo biến và task

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	11

```

49 osThreadId_t Task01Handle;
50 const osThreadAttr_t Task01_attributes = {
51     .name = "Task01",
52     .stack_size = 256 * 4,
53     .priority = (osPriority_t) osPriorityNormal,
54 };
55 /* Definitions for Task02 */
56 osThreadId_t Task02Handle;
57 const osThreadAttr_t Task02_attributes = {
58     .name = "Task02",
59     .stack_size = 128 * 4,
60     .priority = (osPriority_t) osPriorityNormal,
61 };
62 /* USER CODE BEGIN PV */
63 char text[50];
64 float pData[3];
65 /* USER CODE END PV */

```

Code trong hàm main

```

108
109 /* Initialize all configured peripherals */
110 MX_GPIO_Init();
111 /* USER CODE BEGIN 2 */
112 BSP_GYRO_Init();
113 BSP_LCD_Init();
114 BSP_LCD_LayerDefaultInit(1, SDRAM_DEVICE_ADDR);
115 BSP_LCD_SelectLayer(1);
116 BSP_LCD_DisplayOn();
117 BSP_LCD_Clear(LCD_COLOR_BLUE);
118 BSP_LCD_SetBackColor(LCD_COLOR_BLUE);
119 BSP_LCD_SetTextColor(LCD_COLOR_WHITE);
120 /* USER CODE END 2 */

```

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	12

```

123  osKernelInitialize();
124
125  /* USER CODE BEGIN RTOS_MUTEX */
126  /* add mutexes, ... */
127  /* USER CODE END RTOS_MUTEX */
128
129  /* USER CODE BEGIN RTOS_SEMAPHORES */
130  /* add semaphores, ... */
131  /* USER CODE END RTOS_SEMAPHORES */
132
133  /* USER CODE BEGIN RTOS_TIMERS */
134  /* start timers, add new ones, ... */
135  /* USER CODE END RTOS_TIMERS */
136
137  /* USER CODE BEGIN RTOS_QUEUES */
138  /* add queues, ... */
139  /* USER CODE END RTOS_QUEUES */
140
141  /* Create the thread(s) */
142  /* creation of Task01 */
143  Task01Handle = osThreadNew(StartTask01, NULL, &Task01_attributes);
144
145  /* creation of Task02 */
146  Task02Handle = osThreadNew(StartTask02, NULL, &Task02_attributes);
147
148  /* USER CODE BEGIN RTOS_THREADS */
149  /* add threads, ... */
150  /* USER CODE END RTOS_THREADS */
151
152  /* USER CODE BEGIN RTOS_EVENTS */
153  /* add events, ... */
154  /* USER CODE END RTOS_EVENTS */
155
156  /* Start scheduler */
157  osKernelStart();
158
159  /* We should never get here as control is now taken by the scheduler */
160

```

Có thể thấy trong code main lại không gọi hàm MX_USB_DEVICE_Init() do IDE đã đưa phần này xuống Task01

Ở hai Task nhóm sẽ thực hiện delay 100 ms ở hai chỗ khác nhau, một ở đầu vòng lặp và một là ở cuối vòng lặp với mục đích là để tránh bị tranh chấp ở array pData[]

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	13

```

247 void StartTask01(void *argument)
248 {
249     /* init code for USB_DEVICE */
250     MX_USB_DEVICE_Init();
251     /* USER CODE BEGIN 5 */
252     /* Infinite loop */
253     for(;;)
254     {
255         BSP_GYRO_GetXYZ(pData);
256         pData[0] = pData[0] * 17.5 * 0.001;
257         pData[1] = pData[1] * 17.5 * 0.001;
258         pData[2] = pData[2] * 17.5 * 0.001;
259         uint16_t data_len = sprintf(text, "%d %d %d\n", (int16_t)pData[0], (int16_t)pData[1], (int16_t)pData[2]);
260         CDC_Transmit_HS((uint8_t*)text, data_len);
261         osDelay(100);
262     }
263     /* USER CODE END 5 */
264 }

```

```

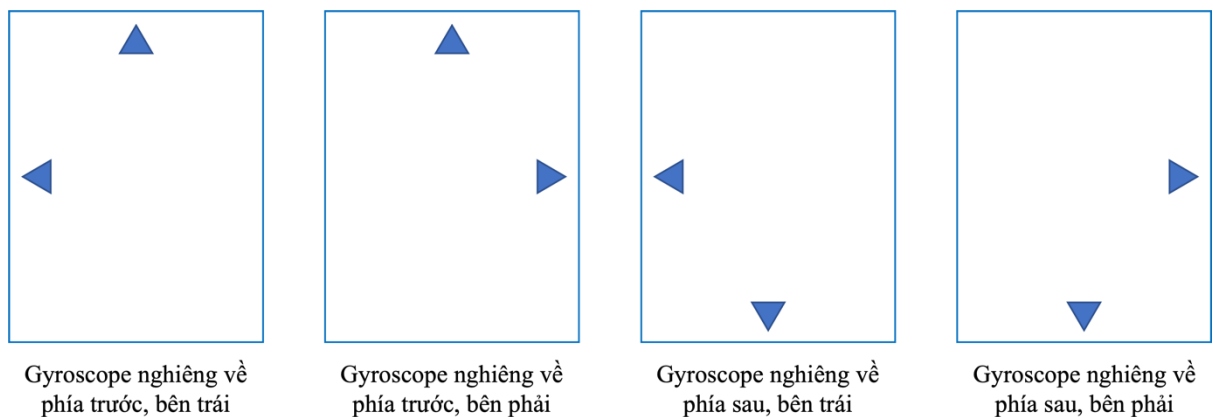
273 void StartTask02(void *argument)
274 {
275     /* USER CODE BEGIN StartTask02 */
276     /* Infinite loop */
277     for(;;)
278     {
279         osDelay(100);
280         if(pData[0] > 1000 && pData[1] > 1000 && pData[2])
281         {
282             BSP_LCD_Clear(LCD_COLOR_BLUE);
283             BSP_LCD_FillTriangle(238, 215, 215, 163, 140, 186); //phai
284             BSP_LCD_FillTriangle(121, 98, 144, 318, 295, 295); //duoi
285         }
286         else if(pData[0] < -1000 && pData[1] < -1000)
287         {
288             BSP_LCD_Clear(LCD_COLOR_BLUE);
289             BSP_LCD_FillTriangle(1, 24, 24, 163, 140, 186); //trai
290             BSP_LCD_FillTriangle(121, 98, 144, 1, 24, 24); //tren
291         }
292         else if(pData[0] < -1000 && pData[1] > 100)
293         {
294             BSP_LCD_Clear(LCD_COLOR_BLUE);
295             BSP_LCD_FillTriangle(238, 215, 215, 163, 140, 186); //phai
296             BSP_LCD_FillTriangle(121, 98, 144, 1, 24, 24); //tren
297         }
298         else if(pData[0] > 1000 && pData[1] < -100)
299         {
300             BSP_LCD_Clear(LCD_COLOR_BLUE);
301             BSP_LCD_FillTriangle(1, 24, 24, 163, 140, 186); //trai
302             BSP_LCD_FillTriangle(121, 98, 144, 318, 295, 295); //duoi
303         }
304     }
305     /* USER CODE END StartTask02 */
306 }

```

So sánh bài tập 1 và bài tập 2

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	14

Tiêu chí	Bài tập 1	Bài tập 2
Độ phức tạp	Đơn giản, dễ hiểu hơn cho người mới	Phức tạp hơn, yêu cầu hiểu rõ về FreeRTOS và cách chia task
Hiệu quả sử dụng CPU	Không tận dụng được CPU khi có các công việc không liên quan	Tận dụng CPU tốt hơn nhờ osDelay() và hệ thống lập lịch của FreeRTOS
Khả năng mở rộng	Khó mở rộng nếu thêm nhiều tác vụ	Dễ mở rộng bằng cách thêm các task mới với độ ưu tiên tương ứng
Xử lý đồng thời	Không thực hiện đồng thời thật sự, chỉ là tuần tự trong vòng lặp	Thực hiện đồng thời hai task nhờ lập lịch của FreeRTOS
Độ ổn định	Có thể gặp tranh chấp hoặc lỗi nếu vòng lặp phức tạp hơn	Tránh tranh chấp nhờ FreeRTOS quản lý tài nguyên tốt hơn



Hình 3.1. Hiển thị hướng của GyroScope lên LCD

Tham khảo:

<https://hocarm.org/rtos-co-ban-phan-1/>

<https://hocarm.org/rtos-co-ban-phan-2/>

<https://www.st.com/resource/en/datasheet/l3gd20.pdf>

<https://stm32f4-discovery.net/2014/08/library-28-l3gd20-3-axis-gyroscope/>

<https://itecnotes.com/electrical/converting-raw-gyro-l3gd20h-values-into-angles/>

LAB 1: LẬP TRÌNH NHÚNG TRÊN KIT STM32F4 DISCOVERY	15
---	----