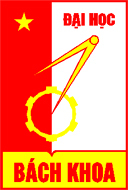
**ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**VIỆN ĐIỆN TỬ VIỄN THÔNG**

====o0o====



**BÁO CÁO**

**MÔN: HỆ ĐIỀU HÀNH**

***Đề tài*: Nghiên Cứu Hệ Điều Hành Nhúng Thời Gian Thực FREERTOS**

**GVHD**:

|  |  |
| --- | --- |
| **Bùi Văn Bao** |  |
| **Đinh Tôn Thép** | 20144242 |
| **Ngô Gia Tiến** |  |

**Lớp : KSTN ĐTVT K59**

Hà Nội, ngày 23 tháng 11 năm 2018

# LỜI MỞ ĐẦU

Ngày nay, cùng với sự phát triển chúng của công nghệ, lĩnh vực hệ điều hành đã có những bước tiền dài trong quá trình hình thành và phát triển của mình. Từ sự khởi đầu đơn giản có bản quyền trên máy tính cá nhân như hệ điều hành DOS, tiến lên các phiên bản Windows như Win 3.1, Win 95, Win 98, … Rồi tiến tới các hệ điều hành đa nhiệm thời gian thực Win 7, Win 10, bên cạnh đó phải kể đến các hệ điều hành mã nguồn mở như Mac OS, Linux, … Đặc biệt hơn nữa, trong lĩnh vực điện tử đã có những hệ điều hành chuyên biệt cho các chip xử lý như VxWork, uCLinux, FreeRTOS, OpenRTOS, SAFERTOS, … Trong đó cần phải nói đến FreeRTOS, với tính thời gian thực và các ưu điểm đã giải quyết nhiều yêu cầu trong công nghệ vi xử lý. Đề tài nãy sẽ đi sâu vào tìm hiểu hệ điều hành thời gian thực mã nguồn mở FreeRTOS và triển khai một chương trình đa tác vụ với hệ điều hành FreeRTOS trên vi điều khiển.

MỤC LỤC

[LỜI MỞ ĐẦU 2](#_Toc530710218)

[DANH MỤC HÌNH ẢNH 4](#_Toc530710219)

[DANH MỤC BẢNG BIỂU 5](#_Toc530710220)

[CHƯƠNG 1. TỔNG QUAN VỀ HỆ ĐIỀU HÀNH THỜI GIAN THỰC 6](#_Toc530710221)

[1.1 Tổng quan các loại hệ điều hành 6](#_Toc530710222)

[1.2 Tìm hiểu hệ điều hành thời gian thực 7](#_Toc530710223)

[1.2.1 Hệ điều hành thời gian thực (RTOS) 7](#_Toc530710224)

[1.2.2 Các loại hệ điều hành thời gian thực 10](#_Toc530710225)

[1.2.3 Tầm quan trọng hệ điều hành thời gian thực 11](#_Toc530710226)

[CHƯƠNG 2. TÌM HIỂU CHI TIẾT VỀ FREERTOS 13](#_Toc530710227)

[2.1 Tổng quan về FreeRTOS 13](#_Toc530710228)

[2.1.1 Khái niệm FreeRTOS 13](#_Toc530710229)

[2.1.2 Các đặc điểm của FreeRTOS 14](#_Toc530710230)

[2.1.3 Các vấn đề cơ bản trong FreeRTOS 16](#_Toc530710231)

[2.1.4 Cách phân phối tài nguyên của FreeRTOS 19](#_Toc530710232)

[2.2 Các file trong kernel của FreeRTOS 22](#_Toc530710233)

[2.2.1 Các file chính trong kernel 23](#_Toc530710234)

[2.2.2 Các file còn lại trong kernel của FreeRTOS 27](#_Toc530710235)

[CHƯƠNG 3. MÔ PHỎNG TRIỂN KHAI HỆ ĐIỀU HÀNH FREERTOS LÊN KIT STM32F103C8………………………………………………………………………………..30](#_Toc530710236)

[3.1 Mô phỏng triển khai FreeRTOS lên KIT STM32F103C8 30](#_Toc530710237)

[3.1.1 Phân tích bài toán mô phỏng 30](#_Toc530710238)

[3.1.2 Các file cần thiết để triển khai FreeRTOS lên vi điều khiển 30](#_Toc530710239)

[3.1.3 Triển khai bài toán và kết quả mô phỏng 33](#_Toc530710240)

[TÀI LIỆU THAM KHẢO 34](#_Toc530710241)

# DANH MỤC HÌNH ẢNH

[Hình 1.1 Sơ đồ hệ điều hành 5](#_Toc530501008)

[Hình 1.2 Hệ điều hành thời gian thực trong điều khiển 7](#_Toc530501009)

[Hình 1.3 Sơ đồ chức năng của hệ điều hành thời gian thực 7](#_Toc530501010)

[Hình 1.4 Trạng thái các tác vụ 8](#_Toc530501011)

[Hình 2.1 Sơ đồ phát triển của FreeRTOS 11](#_Toc530501012)

[Hình 2.2 Sơ đồ phân chia thời gian các tác vụ thực hiện 15](#_Toc530501013)

[Hình 2.3 Sơ đồ chuyển giao các tác vụ 15](#_Toc530501014)

[Hình 2.4 Sơ đồ phân chia các sự kiện theo thời gian 17](#_Toc530501015)

[Hình 2.5 Sơ đồ lập lịch của ví dụ về ưu tiên kế thừa 20](#_Toc530501016)

[Hình 2.6 Sơ đồ các file và thư mục trong gói FreeRTOS.zip tải về 21](#_Toc530501017)

[Hình 2.7 Ví dụ về đánh dấu hoạt động của kernel 23](#_Toc530501018)

# DANH MỤC BẢNG BIỂU

[Bảng 1.1 Bảng so sánh hai loại hệ điều hành thời gian thực 9](#_Toc530500972)

[Bảng 2.1 Các hỗ trợ khác nhau từ FreeRTOS và OpenRTOS 11](#_Toc530500973)

[Bảng 2.2 Bảng phân phối RAM của heap1 18](#_Toc530500974)

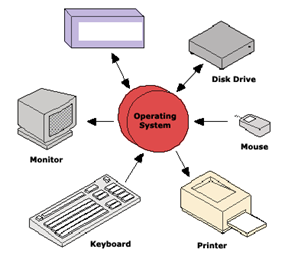
[Bảng 2.3 Bảng phân phối RAM của heap2 19](#_Toc530500975)

[Bảng 2.4 Bảng phân chi tiết các tiến trình 19](#_Toc530500976)

# TỔNG QUAN VỀ HỆ ĐIỀU HÀNH THỜI GIAN THỰC

## Tổng quan các loại hệ điều hành

Những hệ thống máy tính bao gồm một hay nhiều bộ vi xử lý, bộ nhớ chính, bàn phím và nhiều thiết bị vào ra. Tất cả trong một hệ thống phức tạp. Viết chương trình kiểm soát và sử dụng tất cả chương trình một cách chính xác, tối ưu là công việc khó vì thế máy tính được trang bị lớp phần mềm gọi là hệ điều hành [2], nhiệm vụ của nó là quản lý mọi thiết bị và cung cấp các chương trình ứng dụng với giao diện đơn giản hơn xuống phần cứng.



Hình . Sơ đồ hệ điều hành

Hệ điều hành có thể nhìn từ hai quan điểm [2]:

* Quản lý tài nguyên (resource manages)
* Máy mở rộng (extended machines).

Ở quan điểm quản lý tài nguyên, công việc của hệ điều hành là quản lý các phần khác nhau của hệ thống một cách hiệu quả. Tài nguyên của hệ thống (CPU, bộ nhớ, thiết bị ngoại vi, ...) vốn rất giới hạn, nhưng trong các hệ thống đa nhiệm, nhiều người sử dụng có thể đồng thời yêu cầu nhiều tài nguyên. Để thỏa mãn yêu cầu sử dụng chỉ với tài nguyên hữu hạn và nâng cao hiệu quả sử dụng tài nguyên, hệ điều hành cần phải có cơ chế và chiến lược thích hợp để quản lý việc phân phối tài nguyên. Ngoài yêu cầu dùng chung tài nguyên để tiết kiệm chi phí, người sử dụng còn cần phải chia sẻ thông tin (tài nguyên phần mềm), khi đó hệ điều hành cần đảm bảo việc truy xuất đến các tài nguyên này là hợp lệ, không xảy ra tranh chấp, mất đồng nhất.

Ở quan điểm là các máy mở rộng, công việc của hệ điều hành là cung cấp cho người sử dụng các máy ảo (virtural machine) sử dụng thuận tiện hơn các máy thực (actual machine). Hệ điều hành làm ẩn đi các chi tiết phần cứng, người sử dụng được cung cấp giao diện đơn giản, dễ hiểu và không phụ thuộc vào thiết bị cụ thể. Thực tế, ta có thể xem hệ điều hành như hệ thống bao gồm nhiều máy tính trừu tượng xếp thành nhiều lớp chồng lên nhau, máy tính mức dưới phục vụ cho máy tính mức trên. Lớp trên cùng là giao diện trực quan nhất để chúng ta điều khiển.

Ngoài ra hệ điều hành theo có 4 chức năng sau:

* Quản lý quá trình (process management): hệ điều hành quản lý các tiến trình, sắp xếp cho tiến trình nào chạy tiến trình nào dừng, phối hợp nhịp nhàng các tiến trình.
* Quản lý bộ nhớ (memory management): hệ điều hành quản lý phân phối tài nguyên nhớ cho các tiến trình chạy.
* Quản lý hệ thống lưu trữ (storage management): hệ điều hành quản lý lưu trữ trên các ổ đĩa, đĩa CD.
* Giao tiếp với người dùng (user interaction)

Có 7 loại hệ điều hành [2]:

* Hệ điều hành cho Mainframe
* Hệ điều hành cho Server
* Hệ điều hành đa vi xử lý
* Hệ điều hành cho máy tính cá nhân
* Hệ điều hành thời gian thực
* Hệ điều hành nhúng
* Hệ điều hành cho thẻ thông minh

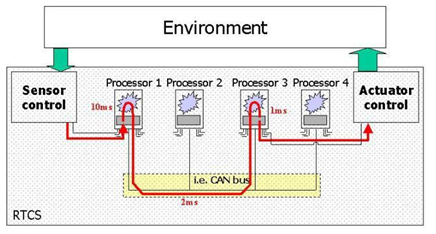
## Tìm hiểu hệ điều hành thời gian thực

Trong phần này ta sẽ đi sâu vào tìm hiểu hệ điều hành thời gian thực với khái niệm, các đặc điểm và tầm quan trọng của hệ điều hành thời gian thực trong thực tế. Từ đó liên hệ trình bày cụ thể về hệ điều hành thời gian thực mã nguồn mở FreeRTOS và các vấn đề cơ bản trong đó.

### Hệ điều hành thời gian thực (RTOS)

#### Khái niệm hệ điều hành thời gian thực

“A real-time system is one in which the correctness of the system depends not only on the logical results, but also on the time at which the results are produced” [5]. Có thể dịch: hệ thống thời gian thực là hệ thống mà sự hoạt động tin cậy của nó không chỉ phụ thuộc vào sự chính xác của kết quả, mà còn phụ thuộc vào thời điểm đưa ra kết quả, hệ thống có lỗi khi yêu cầu về thời gian không được thoả mãn.

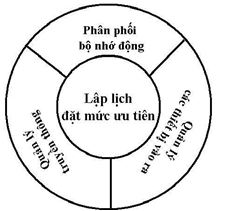


Hình . Hệ điều hành thời gian thực trong điều khiển [1]

Một RTOS thường có tính mềm dẻo và có tính cấu trúc. Nó cho phép tích hợp thêm các dịch vụ gia tăng theo vòng tròn đồng tâm. Vòng trong cùng hay nhân cung cấp những đặc tính quan trọng nhất của hệ điều hành thời gian thực. Các đặc điểm khác có thể được thêm vào như một vòng ngoài khi cần thiết. Nhân nhỏ của một RTOS thích hợp cho một ứng dụng bộ xử lý nhỏ, trong khi những vòng ngoài có thể giúp đỡ xây dựng hệ thống thời gian thực lớn. Các RTOS thường cung cấp các mức xử lý ưu tiên. Các công việc ưu tiên cao hơn sẽ được thực hiện trước.

Ngoài các chức năng của hệ điều hành như trên, hệ điều hành thời gian thực có thể hỗ trợ thêm các chức năng sau:

* Lập lịch phân chia thời gian sử dụng tài nguyên, đặt mức ưu tiên các tác vụ.
* Truyền thông và đồng bộ giữa các tác vụ hoặc giữa tác vụ và ngắt.
* Phân phối bộ nhớ động.
* Quản lý các thiết bị vào ra.



Hình . Sơ đồ chức năng của hệ điều hành thời gian thực [1]

Nhiều hệ điều hành không thời gian thực cũng cung cấp nhân của tương tự nhưng điểm khác biệt lớn nhất của hệ điều hành thời gian thực và hệ điều hành không thời gian thực nói chung là tính tiền định (deterministic). Thời gian tiền định tức là các dịch vụ của hệ điều hành thực hiện chỉ được yêu cầu một khoảng thời gian nhất định, muốn tiền định tức là phải tính toán chính xác theo toán học. Các đáp ứng về thời gian là nghiêm ngặt trong hệ điều hành thời gian thực, không thể có thành phần thời gian ngẫu nhiên. Với một phần thời gian ngẫu nhiên có thể tạo ra trễ ngẫu nhiên, từ đó gây ra các đáp ứng deadline không thoả mãn.

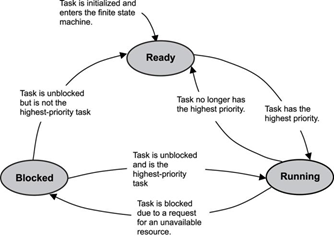
#### Các đặc điểm của RTOS

Một RTOS được ứng dụng thành công vào một nền vi điều khiển thường phải bao gồm 3 nhóm sau:

* RTOS Kernel: nhân của hệ điều hành, trong đó thực thi các nhiệm vụ cơ bản của RTOS. Kernel dùng chung cho tất cả các platform và được cập nhật theo version.
* Port: thường được phát triển bởi nhà sản xuất MCU, nó khác nhau đối với từng họ MCU.
* BSP (Board Support Package): chứa các hàm chức năng truy xuất đến các ngoại vi, thực chất đây chính là driver cho các ngoại vi của MCU.

Một RTOS tốt chỉ khi có nhân tốt, tài liệu tốt và được phân phát cùng các công cụ tốt để phát triển và vận hành ứng dụng. Vì vậy, các tính toán về khoảng thời gian ngắt và thời gian chuyển mạch ngữ cảnh là rất quan trọng, cùng với các thông số khác làm nên một RTOS tốt.

Xây dựng các khối cơ bản của phần mềm dưới RTOS là tác vụ - task. Việc tạo ra các tác vụ dưới RTOS là rất đơn giản. Một tác vụ đơn giản chỉ là một thủ tục con. Tại một số điểm trong chương trình, chúng ta thực hiện một hoặc nhiều lời gọi tới một hàm trong RTOS để bắt đầu các tác vụ. Mỗi tác vụ trong RTOS luôn luôn ở một trong ba trạng thái chính:



Hình . Trạng thái các tác vụ [1]

* **Running:** Với ý nghĩa bộ xử lý đang thực hiện tác vụ. Với một bộ xử lý thì chúng ta chỉ chạy một tác vụ tại một thời điểm nhất định.
* **Ready:** Với ý nghĩa một số tác vụ khác sẵn sàng chạy nếu bộ xử lý rỗi.
* **Blocked:** Với ý nghĩa tác vụ không sẵn sàng chạy kể cả khi bộ xử lý trong trạng thái nghỉ - Idle. Tác vụ ở trong trạng thái này vì chúng đợi một sự kiện bên ngoài tác động để kích hoạt nó trở lại trạng thái sẵn sàng.

### Các loại hệ điều hành thời gian thực

Hệ điều hành thời gian thực cứng (Hard Real Time Operating Systems): là hệ điều hành thời gian thực mà các tác vụ không chỉ đúng về thực thi mà còn phải đúng về thời gian, không cho phép sai lệch về thời gian. Nó thường được tìm thấy ở các lớp thấp với tầng vật lý. Ví dụ như hệ thống điều khiển năng lượng của ô tô là hệ điều hành thời gian thực vì chỉ cần trễ điều khiển một chút có thể gây ra lỗi năng lượng gây hỏng hóc phá huỷ. Một ví dụ khác là hệ điều hành thời gian thực cứng trong y học như máy điều hoà nhịp tim và điều khiển các quá trình công nghiệp.

Hệ điều hành thời gian thực mềm (Soft Real Time Operating Systems): là hệ điều hành thời gian thực cho phép sai lệch về thời gian và dung sai lỗi ở một mức độ nào đó. Nó thường được tìm thấy ở những chỗ xảy ra sự tranh chấp và cần giữ số kết nối hệ thống phản ứng với sự thay đổi hoàn cảnh. Nói chung so với hệ điều hành thời gian thực cứng, hệ điều hành thời gian thực mềm đặt sự thoả mãn yêu cầu cho hầu hết các tác vụ hơn là thoả mãn một deadline nào đó, nó cho phép một số deadline không được thoả mãn ở một dung sai nào đó. Ví dụ như phần mềm duy trì và cập nhậtkế hoạch bay cho hãng hàng không thương mại, hệ thống truyền hình và nghe nhạc trực tuyến, điều khiển máy giặt, …

Bảng . Bảng so sánh hai loại hệ điều hành thời gian thực

|  |  |  |
| --- | --- | --- |
| **Các đặc điểm** | **Hệ điều hành thời gian thực cứng** | **Hệ điều hành thời**  **gian thực mềm** |
| Thời gian đáp ứng | Thoả mãn nghiêm ngặt | Cho phép sai lệch trong giới hạn |
| Thực thi tại thời điểm tải nặng nhất | Có thể dự báo trước | Giảm sút |
| Độ an toàn | Thường là tranh chấp | Ít tranh chấp |
| Dung lượng dữ liệu | Nhỏ hoặc trung bình | Lớn |
| Điều khiển nhịp độ | Nhỏ hoặc trung bình | Phụ thuộc vi diều khiển |
| Đoạn dữ liệu | Đoạn ngắn | Đoạn dài |
| Bắt lỗi | Tự động | Phụ thuộc người dùng |

### Tầm quan trọng hệ điều hành thời gian thực

Do các yêu cầu khắt khe về thời gian, về việc sử dụng tài nguyên, và sự quan trọng của việc lập lịch, các hệ điều hành thời gian thực đóng vai trò rất quan trọng. Chúng giống như những thư viện, chúng ta có thể dùng, thêm bớt các dịch vụ cho phù hợp với ứng dụng thời gian thực để có thể phát triển ứng dụng thời gian thực một cách nhanh hơn, tin tưởng hơn. Vì vậy sự tồn tại của các hệ điều hành thời gian thực là rất cần thiết và quan trọng.

Có thể tìm thấy hệ điều hành thời gian thực ở bất kỳ nơi nào. Chúng cũng phổ biến như những hệ điều hành mà bạn đã quen thuộc như Windows, Mac OS và Unix. Chúng âm thầm làm việc bên trong các bộ định tuyến và chuyển mạch trên mạng, động cơ xe, máy nhắn tin, điện thoại di động, thiết bị y tế, thiết bị đo lường và điều khiển công nghiệp và các vô số ứng dụng khác.

Một thuộc tính quan trọng của RTOS là khả năng tách biệt với ứng dụng. Nếu có một chương trình bị "chết" hay hoạt động không hợp lệ, hệ điều hành thời gian thực có thể nhanh chóng cô lập chương trình này, kích hoạt cơ chế phục hồi và bảo vệ các chương trình khác hay chính bản thân hệ điều hành khỏi các hậu quả của các lệnh sai. Cơ chế bảo vệ tương tự cũng được áp dụng để tránh tình trạng tràn bộ nhớ do bất kỳ chương trình nào gây ra. RTOS xuất hiện ở hai dạng: cứng và mềm. Nếu tính năng xử lý ứng với một sự kiện nào đó không xảy ra hay xảy ra không đủ nhanh, RTOS cứng sẽ chấm dứt hoạt động này và giữ không gây ảnh hưởng đến độ tin cậy và tính sẵn sàng của phần còn lại của hệ thống.

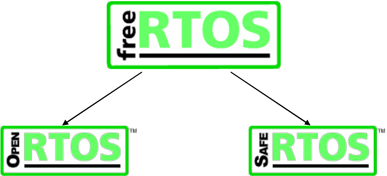
RTOS và máy tính nhúng trở nên phổ biến trong các ứng dụng quan trọng nên các nhà phát triển thương mại đang tạo nên những RTOS mới với tính sẵn sàng cao. Những sản phẩm này có một thành phần phần mềm chuyên dụng làm chức năng cảnh báo, chạy các chương trình chẩn đoán hệ thống để giúp xác định chính xác vấn đề trục trặc hay tự động chuyển đổi sang hệ thống dự phòng. Hiện thời RTOS sẵn sàng cao hỗ trợ bus Compact PCI của tổ chức PCI Industrial Computer Manufacturers Group, bus này dùng cho phần cứng có thể trao đổi nóng.

Trong nhiều năm, ứng dụng dựa trên RTOS chủ yếu là trong các hệ thống nhúng và mới gần đây thì chúng đã có mặt khắp nơi, từ thiết bị y tế được điều khiển bằng máy ảnh cho đến máy pha cà phê, những ứng dụng tính toán phân tán đang thúc đẩy các nhà phát triển hệ điều hành thực hiện nghiên cứu và phát triển chuẩn. Chính phủ Mỹ cũng có một số chương trình về lĩnh vực này như công nghệ quản lý tài nguyên thời gian thực, mạng, quản lý dữ liệu và phần mềm điều khiển trung gian. Mục đích của chương trình là làm cho các hệ thống cộng tác, phân tán có thể giao tiếp và chia sẻ tài nguyên với nhau. Một uỷ ban chuyên trách đang đẩy mạnh việc tạo ra khung công nghệ cho tính toán phân tán thời gian thực, áp dụng cho cả ứng dụng quân sự và thương mại.

# TÌM HIỂU CHI TIẾT VỀ FREERTOS

## Tổng quan về FreeRTOS

### Khái niệm FreeRTOS



Chuẩn IEC 61508

Thương mại hóa

Hình . Sơ đồ phát triển của FreeRTOS

FreeRTOS là lõi của hệ điều hành thời gian thực miễn phí. Hệ điều hành này được Richard Barry công bố rộng rãi từ năm 2003, phát triển mạnh đến nay và được cộng đồng mạng mã nguồn mở ủng hộ. FreeRTOS có tính khả chuyển, mã nguồn mở, lõi có thể down miễn phí và nó có thể dùng cho các ứng dụng thương mại. Nó phù hợp với những hệ nhúng thời gian thực nhỏ. Hầu hết các code được viết bằng ngôn ngữ C nên nó có tính phù hợp cao với nhiều nền khác nhau.

Ưu điểm của nó là dung lượng nhỏ và có thể chạy trên những nền mà nhiều hệ không chạy được. Có thể port cho nhiều kiến trúc vi điều khiển và những công cụ phát triển khác nhau. Mỗi port chính thức bao gồm những ứng dụng ví dụ tiền cấu hình biểu hiện sự riêng biệt của lõi, kiến thức mới và hướng phát triển. Những hỗ trợ miễn phí được cung cấp bởi cộng đồng mạng. Hỗ trợ thương mại với những dịch vụ phát triển đầy đủ cũng được cung cấp.

FreeRTOS được cấp giấy phép bởi bản đã được chỉnh sửa bởi GPL (General Public License) và có thể sử dụng trong ứng dụng thương mại với giấy phép này. Ngoài ra liên quan đến FreeRTOS có OpenRTOS và SafeRTOS. OpenRTOS là bản thương mại của FreeRTOS.org và không liên quan gì đến GPL. SafeRTOS là về cơ bản dựa trên FreeRTOS nhưng được phân tích, chứng minh bằng tài liệu và kiểm tra nghiêm ngặt với chuẩn IEC61508. Và chuẩn IEC61508 SIL3 đã được tạo ra và phát triển độc lập để hoàn thiện tài liệu cho SafeRTOS.

Cụ thể các hỗ trợ khác nhau của FreeRTOS và OpenRTOS:

Bảng . Các hỗ trợ khác nhau từ FreeRTOS và OpenRTOS

|  |  |  |
| --- | --- | --- |
|  | FreeRTOS | OpenRTOS |
| Miễn phí | Có | Không |
| Có thể sử dụng trong ứng dụng thương mại? | Có | Có |
| Miễn phí trong ứng dụng thương mại? | Có | Có |
| Phải đưa ra mã nguồn của code ứng dụng? | Không | Không |
| Phải đưa ra thay đổi mã nguồn của lõi? | Có | Không |
| Phải đưa vào báo cáo nếu sử dụng FreeRTOS.org? | Có, đường dẫn | Không |
| Phải cung cấp mã FreeRTOS cho người sử dụng? | Có | Không |
| Có thể nhận hỗ trợ thương mại? | Không | Có |

### Các đặc điểm của FreeRTOS

#### Các chức năng được cung cấp trong FreeRTOS

FreeRTOS.org là lõi của hệ điều hành thời gian thực được thiết kế cho các hệ thống nhúng nhỏ, với các chức năng chính sau [1]:

* FreeRTOS hỗ trợ cả preemptive, cooperative và các lựa chọn cấu hình lai giữa hai phần.
* SafeRTOS là sản phẩm dẫn xuất, cung cấp mã nguồn riêng ở mức độ cao.
* Được thiết kế nhỏ, đơn giản và dễ sử dụng.
* Cấu trúc mã nguồn rất linh động được viết bằng ngôn ngữ C.
* Hỗ trợ cả task và co-routine
* Mạnh về hàm theo vết.
* Có lựa chọn nhận biết tràn ngăn xếp.
* Không giới hạn số task có thể tạo ra, phụ thuộc vào tài nguyên của chip.
* Không giới hạn số mức ưu tiên được sử dụng.
* Không giới hạn số task cùng một mức ưu tiên.
* Hỗ trợ truyền thông và đồng bộ giữa các task hoặc giữa task và ngắt: queues, binary semaphores, counting semaphores and recursive mutexes.
* Mutexes với ưu tiên kế thừa.
* Các công cụ phát triển miễn phí, port cho Cortex-M3, ARM7, PIC, MSP430, H8/S, AMD, AVR, x86 và 8051.
* Miễn phí mã nguồn phần mềm nhúng.
* Miễn phí trong ứng dụng thương mại.
* Tiền cấu hình cho các ứng dụng demo, từ đó dễ dàng tìm hiểu và phát triển.

#### Tại sao chọn FreeRTOS?

Đây là một số nguyên nhân cho thấy tại sao lại chọn FreeRTOS cho ứng dụng thời gian thực [1]:

* Cung cấp một giải pháp cho rất nhiều kiến trúc và công cụ phát triển khác nhau.
* Được biết là đáng tin cậy. Sự tin cậy được đảm bảo bởi những hoạt động đảm nhận bởi SafeRTOS.
* Đang tiếp tục các hoạt động phát triển mở rộng.
* Sử dụng ít ROM, RAM và it bị quá tải.
* Mã nguồn được viết bằng C nên phù hợp với nhiều nền khác nhau.
* Rất đơn giản, lõi của hệ điều hành chỉ gồm 3 hoặc 4 file (phụ thuộc vào việc có sử dụng co-routine hay không). Phần lớn các file nằm trong file .zip được tải về, nó cung cấp hầu hết các ứng dụng.
* Có thể sử dụng miễn phí trong ứng dụng thương mại (xem kỹ các điều kiện bản quyền).
* Hỗ trợ nhiều: porting, nền phát triển, hoặc các dịch vụ phát triển ứng dụng mà nó yêu cầu.
* Là tổ chức tốt với số người sử dụng lớn và ngày càng tăng.
* Bao gồm nhiều ví dụ tiền cấu hình cho mỗi port. Không cần tính toán xem làm thế nào cài đặt, chỉ tải về và dịch!
* Có những hỗ trợ miễn phí từ cộng đồng mạng.

#### Các vi điều khiển và trình dịch đã được hỗ trợ port FreeRTOS

Vi điều khiển:

* Vi điều khiển ST STM32 Cortex-M3.
* ARM Cortex-M3 dựa trên vi điều khiển sử dụng ARM Keil (RVDS), IAR, Rowley và công cụ GCC.
* Atmel AVR32 AT32UC3A: vi điều khiển flash sử dụng GCC và IAR.
* Các vi điện tử ST: STR71x (ARM7), STR75x( ARM7), STR9 (ARM9) (STR711F, STR712F, … ).
* LPC2106, LPC2124 và LPC2129 (ARM7). Gồm mã nguồn cho I2C driver.
* H8S2329 (Hitachi H8/S) với EDK2329 demo.
* Atmel AT91SAM7 family (AT91SAM7X256, AT91SAM7X128, AT91SAM7S32, AT91SAM7S64, AT91SAM7S128, AT91SAM7S256). Bao gồm mã nguồn USB driver cho IAR Kickstart, uIP và lwIP nhúng vào Ethernet TCP/IP.
* AT91FR40008 với Embest ATEB40X demo.
* MSP430 với demo cho LCD driver. MSPGCC and Rowley CrossWorks được hỗ trợ.
* HCS12 (MC9S12C32 loại bộ nhớ nhỏ và MC9S12DP256B kiểu bank nhớ)
* Fujitsu MB91460 series (32bit) and MB96340 series (16FX 16bit) sử dụng trình dịch Softune và Euroscope debugger.
* Cygnal 8051 / 8052
* Microchip PICMicro PIC18 (8 bit), PIC24 (16bit MCU) và dsPIC (16bit DSC) và PIC32 (32bit)
* Atmel AVR (MegaAVR) với STK500 demo.
* Vi điều khiển RDC8822 với demo cho Flashlite 186 SBC.
* PC (chạy ở FreeDOS hoặc DOS khác)
* ColdFire, chú ý rằng port nàyko được hỗ trợ.
* Zilog Z80, chú ý rằng port nàyko được hỗ trợ.
* Xilinx Microblaze chạy trên Virtex4 FPGA.
* Xilinx PowerPC (PPC405) chạy trên Virtex4 FPGA.

Ngoài ra các trình dịch đã hỗ trợ port: Rowley CrossWorks, Keil, CodeWarrior, IAR, GNU GCC (nhiều loại), MPLAB, SDCC, Open Watcom, Paradigm và Borland.

### Các vấn đề cơ bản trong FreeRTOS

Các vấn đề cơ bản trong FreeRTOS [1] cũng nằm trong các vấn đề cơ bản của RTOS nói chung:

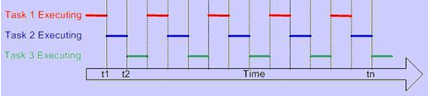
* Đa nhiệm (Mutiltasking)
* Bộ lịch trình (Scheduling)
* Chuyển đổi ngữ cảnh (Context Switching)
* Ứng dụng thời gian thực (Real Time Application)
* Lập lịch thời gian thực (Real Time Scheduling)

#### Đa nhiệm

Thuật ngữ kernel được dùng để chỉ đến một thành phần cốt lõi bên trong của một hệ điều hành. Các hệ điều hành như Linux sử dụng nhân kernel cho phép nhiều người dùng có thể truy cập vào máy tính dường như là liên tục về mặt thời gian. Nhiều người dùng có thể thi hành các chương trình nhìn bề ngoại có vẻ như là đồng thời với hệ điều hành. Thực ra, mỗi một chương trình đang thi hành là một nhiệm vụ được phân chia điều khiển bởi hệ điều hành. Nếu một hệ điều hành có khả năng thi hành nhiều tác vụ thì được gọi là đa nhiệm (multitasking). Sử dụng hệ điều hành đa nhiệm sẽ làm đơn giản quá trình thiết kế những bài toán mà nó sẽ là gánh nặng nếu chuyển hết cho phần mềm ứng dụng xử lý.

Đa nhiệm và tính năng liên lạc nội bộ giữa các tác vụ của hệ điều hành cho phép các ứng dụng phức tạp có thể phân chia ra thành các tác vụ nhỏ hơn, đơn giản hơn, dễ quản lý hơn. Các phần chia nhỏ này sẽ giúp chúng ta dễ dàng có kết quả trong quá trình kiểm tra phần mềm, sử dụng lại mã chương trình ... Những sự tính toán thời gian phức tạp và các quá trình tuần tự chi tiết được tách biệt ra khỏi chương trình ứng dụng và chuyển nhiệm vụ này cho hệ điều hành đảm trách.

Thường thì các bộ vi xử lý chỉ có thể thực hiện một tác vụ duy nhất trong một thời điểm nhưng với sự chuyển đổi một cách rất nhanh giữa các tác vụ của một hệ điều hành đa nhiệm làm cho chúng dường như được chạy đồng thời với nhau. Điều này được mô tả ở sơ đồ dưới đây với 3 tác vụ và giản đồ thời gian của chúng.

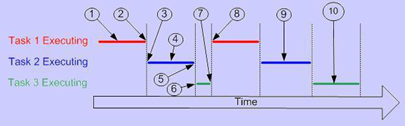


Hình . Sơ đồ phân chia thời gian các tác vụ thực hiện [2]

#### Lập lịch

Bộ lịch trình là một phần của nhân hệ điều hành chịu trách nhiệm quyết định nhiệm vụ nào sẽ được thi hành tại một thời điểm. Nhân kernel có thể cho dừng một tác vụ và phục hồi lại tác vụ sau đó nhiều lần trong suốt quá trình sống của tác vụ đó.

Scheduling policy - cơ chế lập lịch trình là thuật toán được sử dụng bởi bộ lịch trình để quyết định tác vụ nào được thi hành tại thời điểm được chỉ dịnh. Cơ chế của một hệ thống nhiều người dùng (không phải thời gian thực) gần như là cho phép mỗi nhiệm vụ chiếm lĩnh hoàn toàn thời gian của bộ vi xử lý. Còn cơ chế của hệ thời gian thực hay hệ nhúng sẽ được mô tả sau đây:



Hình . Sơ đồ chuyển giao các tác vụ [2]

Các task 1, 2, 3 có mức ưu tiên giảm dần cứ khi nào tác vụ ưu tiên cao yêu cầu thì các task ưu tiên thấp phải nhường.

* Tại (1), nhiệm vụ thứ nhất được thi hành
* Tại (2), nhân kernel dừng tác vụ 1
* Tại (3), phục hồi lại tác vụ 2
* Tại (4), trong khi tác vụ 2 đang thi hành, nó cấm các ngoại vi của vi xử lý chỉ cho phép chính bản thân tác vụ này truy cập vào.
* Tại (5), nhân kernel dừng tác vụ 2
* Tại (6) phục hồi lại tác vụ 3
* Tác vụ 3 cố thử truy cập vào cùng ngoại vi của vi xử lý và thấy rằng đã bị cấm, vì vậy nó không thể tiếp tục và tự dừng tại vị trí (7)
* Tại (8), nhân kernel phục hồi lại task 1.
* Tại (9), thời điểm tiếp theo tác vụ 2 đang thi hành và kết thúc với ngoại vi của vi xử lý và cho phép các tác vụ khác truy cập.
* Thời điểm tiếp là tác vụ 3 đang thi hành và thấy rằng có thể truy cập vào ngoại vi nên tiếp tục thực thi cho khi bị dừng bởi nhân kernel

#### Chuyển đổi ngữ cảnh

Khi một tác vụ đang thi hành, nó sẽ sử dụng các thanh ghi và truy cập vào ROM, RAM như các tác vụ khác. Những tài nguyên này bao gồm: thanh ghi, RAM, ROM, stack ... gọi là ngữ cảnh thực thi nhiệm vụ của một tác vụ.

Một tác vụ là một đoạn mã liên tục, nó sẽ không biết và không được báo trước nếu bị dừng hoặc được phục hồi bởi kernel. Phân tích trường hợp mà một tác vụ bị dừng một cách tức thì khi đang thực hiện một lệnh cộng hai thanh ghi của bộ vi xử lý.

* Khi tác vụ đó đã bị dừng, các tác vụ khác sẽ thi hành và có thể thay đổi các giá trị thanh ghi của bộ vi xử lý. Dựa trên sự phục hồi một tác vụ sẽ không nhận biết được rằng các thanh ghi của vi xử lý đã bị thay đổi - nếu sử dụng các giá trị đã bị thay đổi sẽ dẫn đến một kết quả sai.
* Để ngăn chặn kiểu lỗi này, yếu tố cần thiết là sự phục hồi một tác vụ phải có một ngữ cảnh đồng nhất. Nhân hệ điều hành sẽ chịu trách nhiệm xác định chắc chắn trường hợp nào cần chuyển ngữ cảnh và thực hiện nhiệm vụ đó khi tác vụ bị dừng. Khi tác vụ được phục hồi, ngữ cảnh đã được lưu lại sẽ được trao trả cho tác vụ đó thực hiện tiếp.

#### Các ứng dụng thời gian thực

Hệ điều hành thời gian thực thực hiện đa nhiệm cũng với nguyên lý trên nhưng các đối tượng của nó thì rất khác so với các đối tượng của hệ không phải thời gian thực. Sự khác biệt này được phản ánh bởi cơ chế lập lịch trình. Hệ thời gian thực/ hệ nhúng được thiết kế sao cho các đáp ứng về mặt thời gian là thực đối với các sự kiện xảy ra trên thế giới thật. Các sự kiện này xảy ra trên thế giới thực có thể có thời điểm kết thúc trước ngay cả hệ nhúng/hệ thời gian thực phải đáp ứng và cơ chế lập lịch của hệ RTOS phải xác định được thời điểm kết thúc mà nó phải gặp.

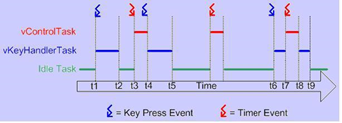
Để thực hiện những mục tiêu trên, kỹ sư lập trình phải gán quyền ưu tiên cho mỗi một tác vụ. Sau đó cơ chế lập lịch của hệ RTOS chỉ đơn giản là xác định tác vụ có quyền ưu tiên cao nhất được phép thi hành ở thời điểm đang xử lý. Điều này dẫn đến cần chia sẻ thời gian xử lý một cách công bằng giữa các tác vụ có cùng ưu tiên và sẵn sàng thực thi.

Tác vụ điều khiển có quyền ưu tiên cao nhất vì:

* Thời hạn cho tác vụ điều khiển có yêu cầu nghiêm ngặt hơn tác vụ xử lý bàn phím.
* Hậu quả của việc mất thời hạn kết thúc (dead line) của tác vụ điều khiển sẽ lớn hơn tác vụ xử lý phím.

#### Bộ lập lịch thời gian thực

Sơ đồ dưới đây trình bày các tác vụ được định nghĩa như thế nào trong phần trước sẽ được lịch trình bởi hệ thời gian thực. Hệ RTOS trước tiên tự tạo cho nó một tác vụ gọi là Idle Task, tác vụ này chỉ thực thi khi không có tác vụ nào có khả năng thực thi. Tác vụ Idle của hệ RTOS luôn ở trạng thái sẵn sàng hoạt động.



Hình . Sơ đồ phân chia các sự kiện theo thời gian [2]

Với giản đồ trên, ta thấy nếu tác vụ điều khiển yêu cầu hoạt động thì tác vụ bàn phím buộc phải nhường do trong lập trình ta luôn để tác vụ điều khiển có mức ưu tiên cao hơn, bộ lập lịch thời gian thực cho phép các tác vụ có ưu tiên cao hơn chiếm quyền chạy trước. Trong bộ lập lịch luôn có tác vụ Idle nhằm quản lý phân phối tài nguyên và nó luôn ở mức ưu tiên thấp nhất, chỉ được chạy khi không có tác vụ nào chạy.

### Cách phân phối tài nguyên của FreeRTOS

#### Phân phối RAM [1]

Kernel RTOS buộc phải sắp xếp, phối hợp RAM cho để tạo ra mỗi tác vụ, hàng đợi hoặc semaphore. Gọi hàm malloc() và free() thỉnh thoảng được sử dụng cho mục đích này nhưng có một số nhược điểm tồn tại:

* Không phải lúc nào nó cũng tồn tại trong các hệ nhúng
* Chiếm nhiều dung lượng code
* Đây không phải là một đoạn tin cậy
* Không thể tiên liệu được (sẽ mất những khoảng thời gian khác nhau khi chạy các hàm này khi gọi từ hàm này đền hàm khác).

nên những cách phân phối RAM là không nhiều. Một hệ thống nhúng hay thời gian thực sẽ có số lượng RAM và các đòi hỏi về thời gian khác nhưng hệ thông khác nên nếu chỉ có 1 cách phân phối RAM sẽ chỉ đáp ứng được một số ít ứng dụng.

Để giải quyết vấn đề này, phương pháp phân phối bộ nhớ API nằm trong lớp portable cung cấp các giải pháp thích đáng cho từng ứng dụng riêng biệt. Khi kernel yêu cầu RAM, thay vì gọi malloc() có thể gọi pvPortMalloc(). Khi RAM được giải phóng, thay vì gọi free() có thể gọi vPortFree().

Có ba kiểu cấp phát bộ nhớ API mã nguồn của FreeRTOS:

* **scheme1 – heap1.c:**

Đây là cách sắp xếp đơn giản nhất. Nó không cho phép giải phóng bộ nhớ khi chúng đã được phân phối nhưng mặc dù vậy nó vẫn thích hợp cho phần lớn các ứng dụng. Thuật toán đơn giản là chia các mảng đơn vào các khối khi có các yêu cầu về RAM. Dung lượng tổng của dãy được đặt bằng cách định nghĩa configTOTAL\_HEAP\_SIZE trong FreeRTOSconfig.h. Sự phân phối này:

* Có thể sử dụng trong các ứng dụng không bao giờ xóa task hoặc hàng đợi
* Luôn tiền định (luôn mất cùng một khoảng thời gian để trở về block)
* Được sử dụng trong PIC, AVR và 8051 vì không linh hoạt trong việc tạo ra và xóa task sau khi vTaskStartScheduler() được gọi.

Bảng . Bảng phân phối RAM của heap1

|  |  |
| --- | --- |
| **Mục** | **Lượng RAM sử dụng (bytes)** |
| Bộ lập lịch | 83 (có thể giảm khi sử dụng kiểu dữ liệu khác nhỏ hơn) |
| Mỗi task mới | 20 (TCB trong đó có 2 byte cho tên) + vùng cho ngăn xếp |
| Mỗi mức ưu tiên | 16 |
| Mỗi hàng đợi | 45 + vùng lưu trữ hàng đợi |
| Mỗi semaphore | 45 |

*\*heap 1 rất phù hợp với các ứng dụng thời gian thực nhỏ, tại đó các task và các hàng đợi được tạo ra trước khi kernel tạo ra.*

* **scheme 2 – heap 2:**

Sự sắp xếp này được coi là thuật toán tốt nhất, không giống 1, nó cho phép các khối nhớ trước được giải phóng. Nó không kết hợp các khối nhớ được giải phóng liền kề nhau thành một khối lớn hơn. Ngoài ra tổng lượng RAM được đặt bằng cách định nghĩa trong configTOTAL\_HEAP\_SIZE trong FreeRTOSconfig.h. Sự sắp xếp này:

* Có thể dùng khi các ứng dụng gọi lại nhiều lần vTaskCreate() / vTaskDelete() ...
* Không nên sử dụng nếu bộ nhớ phân phối và giải phóng với dung lượng bất kỳ, có thể trong trường hợp đơn giản sau: các task bị xóa có độ sâu stack khác nhau, các hàng đợi bị xóa có độ dài khác nhau.
* Có thể xảy ra vấn đề phân mảnh bộ nhớ khi các ứng dụng tạo các khối, task, hàng đợi không theo trật tự. Có thể sẽ không xảy ra với những ứng dụng gần đây nhưng hãy ghi nhớ để chú ý.
* Không tiền định nhưng nó không phải không có những khả năng đặc biệt.
* Có thể sử dụng trong ARM7 và Flashlite vì nó linh động trong việc tạo và xóa task.

Bảng . Bảng phân phối RAM của heap2

|  |  |
| --- | --- |
| **Mục** | **Lượng RAM sử dụng (bytes)** |
| Bộ lập lịch | 216 (có thể giảm khi sử dụng kiểu dữ liệu khác nhỏ hơn) |
| Mỗi task mới | 64 (TCB trong đó có 4 byte cho tên) + vùng cho ngăn xếp |
| Mỗi hàng đợi | 76 + vùng lưu trữ hàng đợi |

*heap\_2.c thích hợp cho ứng dụng thời gian thực tạo task một cách linh động.*

* **scheme 3 – heap\_3.c:**

Đây là chuẩn cho *malloc()* và *free()*, làm cho chức năng này là thread an toàn:

* Yêu cầu các liên kết để cài đặt heap và các thư viện dịch để giúp *malloc()* và *free()* thực hiện
* Không tiền định
* Sẽ gia tăng dung lượng kernel lên rất nhiều
* Sử dụng cho PC

Khi FreeRTOS lập lịch theo kiểu preemtive, nó sẽ sử dụng kiểu lập lịch ưu tiên kế thừa *(Priority Inheritance)*, báo hiệu qua mutex. Ưu tiên kế thừa tức là trong quá trình chạy đến một thời điểm nào đó task có mức ưu tien thấp hơn nắm giữ tài nguyên mà task có mức ưu tiên cao hơn đang yêu cầu thì task ưu tiên thấp hơn sẽ nhận mức ưu tiên của task cao hơn để chạy. Khi nào task ưu tiên thấp giải phóng tài nguyên mà task ưu tiên cao cần thì mức ưu tiên trở lại như cũ.

Ta lấy ví dụ minh họa với 4 tiến trình như sau:

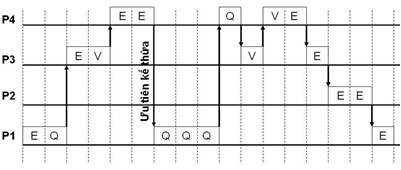
Bảng . Bảng phân chi tiết các tiến trình

|  |  |  |  |
| --- | --- | --- | --- |
| **Tiến trình** | **Mức ưu tiên** | **Chuỗi tài nguyên** | **Thời điểm bắt đầu** |
| P4 | 4 | EEQVE | 4 |
| P3 | 3 | EVVE | 2 |
| P2 | 2 | EE | 2 |
| P1 | 1 | EQQQE | 0 |

Trong đó:

* E: đơn vị không cần tài nguyên
* Q: đơn vị thời gian cần tài nguyên Q
* V: đơn vị thời gian cần tài nguyên V

Ta sẽ có sơ đồ chạy sau:



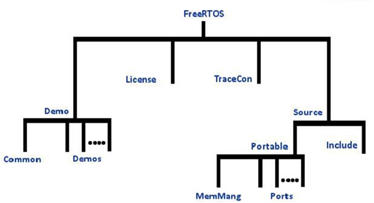
Hình . Sơ đồ lập lịch của ví dụ về ưu tiên kế thừa

Có thể giải thích sơ đồ như sau:

* Ở thời điểm đầu tiên P1 được chạy do chỉ có mình nó yêu cầu
* Khi P3 bắt đầu thì P2 cũng bắt đầu, nhưng do P3 có mức ưu tiên cao hơn P1 và P2 nên nó giành lại quyền chạy từ P1.
* Tương tự tại điểm tiếp theo P4 chạy.
* Khi P4 cần tài nguyên Q thì P1 đang giữ, P4 phải dừng lại, P1 kế thừa mức ưu tiên từ P4 và P1 được chạy.
* Đến khi P1 giải phóng tài nguyên Q thì nó trở về mức ưu tiên 1.
* Tiếp sau đó cứ tiến trình nào mức ưu tiên cao hơn thì được chạy trước cho đến khi hoàn thành.

## Các file trong kernel của FreeRTOS

Trong phần tìm hiểu kỹ về FreeRTOS này ta tiếp cận theo từng file. Mỗi file cũng chính là một modun, tiếp cận từng file cũng chính là tiếp cận từng modun của FreeRTOS, từ đó ta có thể trình bày cụ thể lần lượt từng vấn đề.



Hình . Sơ đồ các file và thư mục trong gói FreeRTOS.zip tải về [3]

### Các file chính trong kernel

Trong kernel của FreeRTOS có năm file chính, tất cả các chương trình port buộc phải có:

* *FreeRTOS.h:* kiểm tra xem *FreeRTOSconfig,h* đã định nghĩa các ứng dụng macro phụ thuộc vào từng chương trình một cách rõ ràng hay chưa.
* *task.h:* tạo ra các hàm và các macro liên quan đến các task, như khởi tạo, xóa, treo,…
* list.h: tạo ra các hàm và các macro liên quan đến việc tạo và xoá danh sách trạng thái các task như các danh sách ready, running, block, suppend, waiting.
* *croutine.h:* tạo ra các hàm và các macro liên quan đến task và queue nhưng chủ yếu dùng cho coorporative.
* *portable.h:* tạo tính linh động cho lớp API. Với mỗi chương trình port cho mỗi vi điều khiển và mỗi trình dịch khác nhau đều cần thay đổi file này để phù hợp các API.

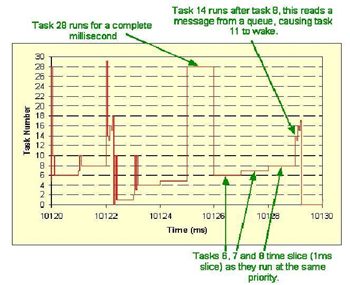
#### FreeRTOS.h

File này nhằm định hướng cho hệ điều hành xem sử dụng các chức năng như thế nào. Kiểm tra xem *FreeRTOSconfig,h* đã định nghĩa các ứng dụng macro phụ thuộc vào từng chương trình một cách rõ ràng hay chưa. Nếu hàm hoặc macro nào muốn sử dụng cần được đặt lên 1, ngược lại đặt ở 0.

#### task.h

Gồm năm phần:

* **Các macro và các định nghĩa:** khai báo một số kiểu sẽ dùng trong file, khai báo các macro như *taskIDLE\_PRIORITY(), taskYIELD(), taskENTER\_CRITICAL(),…* và định nghĩa một số hằng số để sử dụng.
* **Các task tạo API:** có hai nhiệm vụ rất quan trọng là tạo mới và xóa task.
* Tạo ra task mới và thêm nó vào danh sách task sẵn sàng chạy là nhiệm vụ của *xTaskCreate(),* trong hàm này phải khai báo tên task, độ sâu stack sử dụng cho task, mức ưu tiên của task, ngoài ra còn một số nhiệm vụ khác.
* Task bị xoá sẽ được gỡ bỏ từ tất cả các danh sách sẵn sàng, khoá, ngắt và sự kiện. Chú ý là idle task có nhiệm vụ về giải phóng vùng nhớ dành cho kernel khỏi task cừa bị xoá. Vì thế điều quan trọng là idle task phải có thời gian của vi điều khiển nếu trong ứng dụng có gọi đến *vTaskDelete().*
* **Các task điều khiển API:** tạo ra các hàm điều khiển API cụ thể là các nhiệm vụ như sau:
* Tạo trễ: *vTaskDelay()* và *vTaskDelayUntil().* *vTaskDelay()* dùng để tạo trễ trong một khoảng thời gian nhất định, còn *vTaskDelayUntil()* tạo trễ đến một thời điểm nhất định.
* Mức ưu tiên: *xTaskPriorityGet()* và *vTaskPrioritySet().* Hai hàm làm nhiệm vụ giành lại mức ưu tiên và đặt mức ưu tiên cho task.
* Thay đổi trạng thái task như treo, khôi phục. *vTaskSuspend()* nhằm để treo bất kỳ task nào. *vTaskResume()* được gọi sau khi task bị treo muốn quay về trạng thái sẵn sàng. Muốn gọi hàm *vTaskResume()* từ ngắt thì sử dụng xTaskResumeFromISR(). Ngoài ra hai hàm *vTaskSuspendAll()* và vTaskResumeAll() cũng tương tự nhưng nó thực hiện với tất cả các task trừ ngắt.
* Lập lịch: *vTaskStartScheduler()* và *vTaskEndScheduler()* là các hàm thực hiện việc bắt đầu và kết thúc việc lập lịch. Chú ý là khi bắt đầu việc lập lịch thì Idle task tự động được tạo ra. Sau khi gọi *vTaskEndScheduler()* mà gọi lại *vTaskStartScheduler()* thì sẽ phục hồi từ thời điểm đó.
* **Các task tiện ích:**
* *xTaskGetTicksCount:* trả lại giá trị ticks đếm được từ khi *vTaskStartScheduler* bị hủy bỏ.
* *uxTaskGetNumberOfTasks(void):* trả lại số lượng các task mà kernel đang quản lý. Hàm này còn bao gồm cả các task đã sẵn sàng, bị khóa hoặc bị treo. Task đã bị delete mà chưa được giải phóng bởi idle cũng được tính vào.
* *vTaskList():* hàm này sẽ không cho phép ngắt trong khoảng thời gian nó làm việc. Nó không tạo ra để chạy các ứng dụng bình thường nhưng giúp cho việc debug.
* *vTaskStartTrace():* đánh dấu việc bắt đầu hoạt động của kernel. Việc đánh dấu chia ra để nhận ra task nào đang chạy vào lúc nào. Đánh dấu file được lưu trữ ở dạng nhị phân. Sử dụng những tiện ích độc lập của DOS thì gọi convtrce.exe để chuyển chúng sang kiểu text file dạng mà có thể được xem và được vẽ.



Hình . Ví dụ về đánh dấu hoạt động của kernel

* *ulTaskEndTrace():* Dừng đánh dấu kernel hoạt động, trả lại số byte mà đã viết vào bộ đệm đánh dấu.
* **Lập lịch nội bộ cho mục đích port:**
* *vTaskIncrementTick():* không sử dụng để code cho các ứng dụng. Gọi từ kernel tick, tăng bộ đếm tick và kiểm tra xem có phải thời điểm cần chuyển trạng thái của task hay không, ví dụ task đang bị khóa đến thời điểm khôi phục sẽ được loại bỏ khỏi danh sách bị khóa và thay vào đó là danh sách sẵn sàng.
* *vTaskPlaceOnEventList():* không sử dụng để code cho các ứng dụng. Hàm này được gọi khi không cho phép ngắt. Loại bỏ tất cả các task đang gọi từ danh sách sẵn sàng và thay vào đó là thêm vào danh sách task chờ sự kiện và danh sách của task trễ. Task sẽ được giải phóng trở lại khi sự kiện xảy ra hoặc hết thời gian trễ.
* *xTaskRemoveFromEventList():* không sử dụng để code cho các ứng dụng. Hàm này được gọi khi không cho phép ngắt. Loại bỏ task từ cả list sự kiện và list các task bị khóa thay vào là hàng đợi sẵn sàng.
* *vTaskCleanUpResources():* không sử dụng để code cho các ứng dụng. Xóa hàng đợi sẵn sàng và trễ của khối điều khiển task, giải phóng bộ nhớ cấp phát cho khối điều khiển task và các ngăn xếp task.
* *xTaskGetCurrentTaskHandle():* trả lại kênh điều khiển cho task đang gọi.
* *vTaskSetTimeOutState():* giữ lại những trạng thái hiện thời để tham chiếu sau này.
* *xTaskCheckForTimeOut():* kiểm tra xem có time out hay không.
* *vTaskMissedYield():* sử dụng để ngăn cản những lời gọi hàm taskYield() không cần thiết.
* vTaskPriorityInherit: nâng mức ưu tiên của mutex holder lên đến task đang gọi nếu mutex holder có mức ưu tiên thấp hơn task đang gọi.
* vTaskPriorityDisinherit: đặt mức ưu tiên cho task trở lại đúng như mức ưu tiên của nó trong trường hợp mà nó kế thừa mức ưu tiên cao hơn trong khi nó đang giữ semaphore.

#### list.h

Trong file list.h, FreeRTOS định nghĩa các cấu trúc, các macro và các hàm phục vụ cho các tiện ích về danh sách. Chức năng của file là tạo mới, thêm, bớt các tác vụ vào danh sách các task đang chạy (*running*), sẵn sàng (*ready*), khoá (*block*), treo (*suppend*). Các chức năng này cụ thể là:

* Khởi tạo danh sách
* Khởi tạo các phần tử trong danh sách.
* Đặt đối tượng sở hữu các phần tử của danh sách.
* Đặt giá trị của phần tử danh sách. Trong hầu hết trường hợp giá trị đó được dùng đểsắp xếp danh sách theo một thứ tự nhất định nào đó.
* Để lấy giá trị của phần tử danh sách. Giá trị này có thể biểu thị bất cứ cái gì, ví dụ như mức ưu tiên của tác vụ hoặc thời gian mà task có thể bị khoá.
* Xác định xem danh sách còn chứa phần tử nào không, chỉ có giá trị true nếu danh sách rỗng.
* Kiểm tra số phần tử trong danh sách.
* Xác định phần tử tiếp theo của danh sách.
* Tìm chương trình chủ của phần tử đầu tiên trong danh sách.
* Kiểm tra xem phần tử có nằm trong danh sách không.
* Thêm phần tử vào danh sách.
* Loại bỏ phần tử từ danh sách.

#### croutine.h

Tạo ra các hàm và các macro liên quan đến task và queue nhưng chủ yếu dùng cho coorporative. Các chức năng của file như sau:

* Ẩn những thực thi của khối điều khiển co-routine.
* Tạo mới các co-routine và thêm vào danh sách các co-routine đã sẵn sàng.
* Lập lịch cho co-routine, cho phép co-routine có mức ưu tiên cao nhất được chạy. Co-routine này sẽ chạy đến khi nó bị khóa, phải nhường hoặc bị ngắt bởi tác vụ. Co-routine chạy trong cooperatively thì một co-routine không bị ngắt bởi các co-routine khác nhưng có thể bị ngắt bởi task. Nếu ứng dụng bao gồm cả task và co-routine thì *vCoRoutineScheduler* có thể được gọi từ idle task (trong idle task hook).
* Các co-routine phải được bắt đầu với những lời gọi macro *crSTART().*
* Các co-routine phải được kết thúc với những lời gọi macro *crEND().*
* Tạo trễ cho các co-routine trong khoảng thời gian cố định.
* Các macro *crQUEUE\_SEND(), crQUEUE\_RECEIVE()* là các co-routine tương đương với các hàm *xQueueSend()* và *xQueueReceive()* chỉ được sử dụng trong các task.
* Macro *crQUEUE\_SEND\_FROM\_ISR()* và *crQUEUE\_RECEIVE\_ FROM\_ISR()* là co-routine tương đương với *xQueueSendFromISR()* và *xQueueReceiveFromISR()* được sử dụng bởi task.
* *vCoRoutineAddToDelayedList*: chỉ được sử dụng co-routine macro. Các macro nguyên thủy của thực thi co-routie đòi hỏi có những nguyên mẫu ở đây. Hàm này loại bỏ co-routine hiện thời từ list sẵn sàng và đặt chúng vào list trễ thích hợp.

#### portable.h

Đây có thể coi là file header của port.c, các hàm này sẽ được tìm hiểu kỹ hơn trong phần port.c. Bên cạnh đó file làm một số nhiệm vụ quan trọng nhằm tạo project:

* Khai báo đường dẫn vào file portmacro.h cho từng project riêng biệt cho phù hợp với vi điều khiển và chương trình dịch.
* Với một số vi điều khiển file mày con include thêm một số file cần thiết để tạo project. Ví dụ như tạo project cho PC ngoài tạo đường dẫn đến portmacro.h còn phải include thêm file *frconfig.h.*

Ngoài ra còn đặt ra các chương trình con quản lý bộ nhớ yêu cầu cho port.

### Các file còn lại trong kernel của FreeRTOS

Các file còn lại trong kernel là ba file:

* *project.h*: định nghĩa các kiểu ban đầu mà các hàm thực hiện phải phù hợp.
* *queue.h*: tạo các hàm nhằm sử dụng hàng đợi.
* *semphr.h*: tạo các hàm nhằm sử dụng semaphore

#### project.h

Nhiệm vụ của file chỉ là định nghĩa các hằng số mà các hàm nên theo đó mà sử dụng. Nếu không sử dụng thì hoàn toàn có thể bỏ file này đi nhưng chú ý rằng phải sửa lại hết các hằng số trong các file dùng sẵn do người viết mã nguồn FreeRTOS luôn tuân thủ chuẩn này. Ngoài ra trong file định nghĩa các lỗi.

#### queue.h

Như tên gọi của file, tất cả các hàm và macro được khai báo trong file nhằm phục vụ cho việc sử dụng hàng đợi cho thuận tiện. Các chức năng cụ thể:

* Tạo hàng đợi mới.
* *xQueueSendToToFront():* Gửi phần tử vào đầu hàng đợi.
* *xQueueSendToToBack():* Gửi phần tử vào sau hàng đợi.
* *xQueueGernericSend():* Gửi phần tử vào hàng đợi.
* *xQueuePeek():* Lấy phần tử ra khỏi hàng đợi mà không loại bỏ nó khỏi hàng đợi. Phần tử được gửi từ hàng đợi bằng cách copy ra một bộ đệm nên phải cung cấp cho bộ đệm dung lượng đủ. Số lượng byte được copy vào bộ đệm phải được khai báo từ khi tạo hàng đợi.
* *xQueueReceive():* Nhận phần tử từ hàng đợi. Phần tử được gửi từ hàng đợi bằng cách copy ra bộ đệm nên phải cung cấp cho bộ đệm dung lượng đủ. Lượng byte được copy vào bộ đệm phải được khai báo từ khi tạo hàng đợi.
* Tương tự các hàm trên nhưng với hàng đợi trong phạm vi phục vụ ngắt có các hàm: *xQueueSendToFrontFromISR(),xQueueSendToBackFromISR(),* *xQueueGenericSendFromISR(), xQueueReceiveFromISR().*
* Tìm số message lưu trữ trong hàng đợi.
* Xóa hàng đợi, giải phóng bộ nhớ phân phối cho hàng đợi.

#### semphr.h

Tất cả các hàm và macro được khai báo trong file nhằm phục vụ cho việc sử dụng semaphore cho thuận tiện. Các chức năng cụ thể:

* Tạo ra semaphore nhị phân, là kiểu đầu tiên được sử dụng trong đồng bộ giữa các tác vụ hoặc giữa tác vụ và ngắtThis type of semaphore can be used for pure synchronisation between tasks or between an interrupt and a task.
* Kiểu semaphore này chỉ là nhị phân nên nếu một task đang cứ sản xuất trong khi task khác cứ tiêu thụ thì sẽ không thỏa mãn. Do đó kiểu này không được sử dụng cho thuật toán ưu tiên kế thừa mà sử *xSemaphoreCreateMutex().*
* Lấy semaphore qua hàm *xSemaphoreTake(),* sử dụng *xQueueReceive().*
* Trả semaphore qua hàm *xSemaphoreGive(),* sử dụng *xQueueGenericSend().*
* Tương tự có semaphore phục vụ ngắt *xSemaphoreGiveFromISR( ),* sử dụng hàm *xQueueGenericSendFromISR( ).*
* Tạo mutex qua *xSemaphoreCreateMutex(),* sử dụng *xQueueCreateMutex().*

# MÔ PHỎNG TRIỂN KHAI HỆ ĐIỀU HÀNH FREERTOS LÊN KIT STM32F103C8

## Mô phỏng triển khai FreeRTOS lên KIT STM32F103C8

### Phân tích bài toán mô phỏng

#### Ý tưởng và mục tiêu của bài toán mô phỏng

Bài toán cần đặt ra ở đây là mô phỏng cho hệ điều hành thời gian thực nên các yêu cầu đặt ra cho bài toán phải gắn liền với các đặc điểm của hệ điều hành thời gian thực. Từ đó ta phải đặt ra các mục tiêu trong phần mô phỏng này:

* Làm nổi bật ý nghĩa của việc có hệ điều hành thời gian thực, tức là trong một hệ thống tài nguyên hạn chế, tranh chấp giữa các tác vụ thường xuyên xảy ra. Như vậy ý tưởng bài toán được thiết kế là với cùng một số tác vụ như nhau nếu tăng yêu cầu đáp ứng về thời gian của một số tác vụ lên thì hệ thống sẽ lỗi không thỏa mãn được yêu cầu đặt ra.
* Bài toán mô phỏng được hầu hết các dạng tác vụ của hệ điều hành thời gian thực. Các dạng tác vụ cần mô phỏng như: tác vụ sự kiện, tác vụ theo chu kỳ, tác vụ truyền thông, …
* Nổi bật việc thêm bớt tác vụ vào hệ thống một cách dễ dàng.

#### Bài toán mô phỏng

Từ những ý tưởng và mục tiêu mô phỏng trên. Ta đặt ra bài toán với 2 tác vụ:

* Tác vụ 1: Nháy Led tự động.

Led (đỏ) nháy tự động với chu kì 1 giây, chân dương nối vào Pin A1, âm nối GND

* Tác vụ 2: Nháy Led khi nhấn nút.

Led (xanh) sẽ sáng khi bấm nút và tắt khi nhả nút

Hai tác vụ này có thể xảy ra tranh chấp về mặt thời gian, mục đích của mô phỏng là sử dụng hệ điều hành FreeRTOS xử lý tranh chấp này

### Các file cần thiết để triển khai FreeRTOS lên vi điều khiển

#### FreeRTOSconfig.h

File được tạo ra với hai nhiệm vụ chính:

* Định nghĩa các thông số, các chức năng cơ bản mà FreeRTOS hỗ trợ được định yêu cầu trong FreeRTOS.h. Các hàm, macro này nếu muốn khai báo có sử dụng thì định nghĩa là 1, ngược lại là 0.
* Các thông số cần định nghĩa cho từng vi điều khiển và từng project cụ thể:
* *configCPU\_CLOCK\_HZ*: khai báo tần số làm việc của vi điều khiển theo đơn vị Hz.
* *configTICK\_RATE\_HZ*: khai báo tần số tick muốn sử dụng, đơn vị Hz.
* *configMAX\_PRIORITIES*: giới hạn mức ưu tiên cao nhất được hỗ trợ để lập lịch.
* *configMINIMAL\_STACK\_SIZE*: giới hạn độ sâu nhỏ nhất của ngăn xếp được dùng cho mỗi task.
* *configTOTAL\_HEAP\_SIZE*: giới hạn tổng lượng RAM trong heap để cấp phát cho từng nhiệm vụ.
* *configMAX\_TASK\_NAME\_LEN*: giới hạn độ dài của tên các tác vụ, đơn vị tính bằng byte.

#### port.c

Đây là file quan trọng nhất trong việc tạo ra các hàm định nghĩa trong portable.h cho việc port lên vi điều khiển. Các nhiệm vụ chính cụ thể như sau:

* *portTIMER\_FOSC\_SCALE* = 4: khai báo để cài đặt phần cứng cho tick.
* *portINITAL\_INTERRUPT\_STATE* = 0xc0: khởi tạo trạng thái cho phép ngắt cho các task được tạo mới. Giá trị này được copy vào *INTCON* khi chuyển task trong lần đầu tiên.
* *portGLOBAL\_INTERRUPT\_FLAG* = 0x80: định nghĩa này chỉ cho các bit nằm trong *INTCON*, ngắt toàn cục.
* *portINTERRUPTS\_UNCHANGED* = 0x00: hằng số được sử dụng cho việc chuyển ngữ cảnh khi yêu cầu ngắt chuyển từ trạng thái cho phép ngắt sang trạng thái không thay đổi khi task vừa bị ngắt khôi phục lại.
* *portCOMPILER\_MANAGED\_MEMORY\_SIZE* = 0x13: một số vùng nhớ cần được lưu lại như một phần của ngữ cảnh tác vụ. Những vùng nhớ này được sử dụng bởi trình dịch cho việc lưu giữ trung gian, đặc biệt là khi thực hiện các phép tình toán học hoặc khi sử dụng dữ lệu 32 bit. Hằng số này định nghĩa độ lớn vùng nhớ phải lưu.
* *vSerialTxISR()* và *vSerialRxISR()*: chương trình phục vụ ngắt cho cổng truyền tin nối tiếp được định nghĩa trong serial.c nhưng vẫn được gọi từ portable, coi như cũng là vector như tick ISR. Trong phần demo cụ thể mà em làm, em đã bỏ phần này đi để làm gọn lõi hệ của hệ điều hành, còn người sử dụng khi có thể tự cài đặt thêm nếu cần.
* *prvSetupTimerInterrupt()*: cài đặt phần cứng để cho phép tick.
* *prvTickISR():* chương trình phục vụ ngắt để duy trì tick và thức hiện chuyển đổi ngữ cảnh tick nếu sử dụng kiểu preemptive.
* *prvLowInterrupt():* chương trình phục vụ ngắt thay thế cho vector mức ưu tiên thấp. Nó gọi những chương trình phục vụ ngắt thích hợp cho các ngắt thực tế.
* Phần quan trọng cũng là khó nhất trong file là lưu và khôi phục ngữ cảnh trong mỗi lần chuyển đổi ngữ cảnh. Đó là hai macro *portSAVE\_CONTEXT()* và *portRESTORE\_CONTEXT().* Với macro lưu ngữ cảnh, nó cất tất cả các thanh ghi làm nên ngữ cảnh của tác vụ vào ngăn xếp, sau đó cất đỉnh mới của ngăn xếp này vào *TCB*. Nếu lời gọi hàm này đến từ *ISR* thì bit cho phép ngắt này đã được set để ISR được gọi. Vì thế ta muốn lưu thanh ghi *INTCON* với các bit đã được set và *ucForcedInterruptFlags*. Điều này có nghĩa là các ngắt sẽ được cho phép trở lại khi task vừa bị ngắt khôi phục. Nếu lời gọi từ thao tác (bằng tay) chuyển ngữ cảnh (ví dụ từ yield) thì ta sẽ lưu *INTCON* với trạng thái hiện thời của nó, và *ucForcedInterruptFlags* phải ở 0. Nó cho phép yield trong vùng bất ly. Ngoài ra, trình dịch thường sử dụng một số vùng ở phía dưới bộ nhớ dùng làm lưu trữ trung gian cho các tính toán. Điều này thực sự đúng khi kiểu dữ liệu 32bit được sử dụng. Các đoạn tmpdata và MATH\_DATA phải được lưu trữ như một phần của ngữ cảnh. Macro này sẽ lưu trữ từ địa chỉ 0x00 đến *portCOMPLIER\_MANAGED\_MEMORY\_SIZE*.
* Lưu thanh ghi *WREG* đầu tiên, nó sẽ bị thay đổi ngay trong các thao tác dưới đây.
* Lưu thanh ghi *INTCON* với các bit thích hợp.
* Lưu các thanh ghi cần thiết vào ngăn xếp như: *BSR*, *FSR2L*, *FSR2H*, …
* Lưu *.tmpdata* và *MATH\_DATA*.
* Lưu con trỏ ngăn xếp phần cứng trong thanh ghi trung gian trước khi tat hay đổi chúng.
* Lưu đỉnh của con trỏ ngăn xếp mềm vào *TCB*.

với macro *portRESTORE\_CONTEXT* ta làm gần như ngược lại. Nhưng hết sức chú ý rằng các lưu trữ này đúng với hầu hết các ứng dụng nhưng không phải hoàn toàn. Cần phải kiểm tra lại với từng ứng dụng cụ thể.

* *\*pxPortInitialiseStack():* cài đặt ngăn xếp của task mới để nó sẵn sàng hoạt động khi bộ lập lịch điều khiển. Các thanh ghi phải được gửi vào ngăn xếp theo thứ tự để port có thể tìm được chúng.
* *xPortStartScheduler():* cài đặt phần cứng sẵn sàng cho bộ lập lịch điều khiển. Nhìn chung là cài đặt cho ngắt tick và cài đặt timer cho tần số đúng của tick. Hàm này được sử dụng ở preemptive (tức là *configUSE\_PREEMTIVE* được đặt bằng 1)
* *vPortEndScheduler():* hủy toàn bộ càiđặt cho phần cứng/ISR đã được thực hiện bởi *xPortStartScheduler()* vì thế phần cứng được để lại các điều kiện đầu tiên sau khi bộ lập lịch dừng hoạt động. Hàm này không thể xảy ra trong bộ lập lịch cho port PIC do không thể dừng sau 1 lần chạy.
* *vPortYield():* chuyển ngữ cảnh thủ công. Hàm này giống như chuyển đổi ngữ cảnh tick nhưng không tăng biến đếm tick. Nó phải đúng như chuyển đổi ngữ cảnh tick trong việc lưu trữ vào ngăn xếp của task như thế nào.

#### portmacro.h

File này định nghĩa cho riêng phần port. Các định nghĩa này cấu hình cho FreeRTOS đúng với phần cứng và trình dịch. Các cài đặt này không được biến đổi.

Các nhiệm vụ của file như sau:

* Định nghĩa các kiểu số liệu cơ bản sử dụng trong FreeRTOS, như: char (*portCHAR*), float (*portFLOAT*), int (*portSHORT*), ...
* Kiểm tra xem nếu sử dụng *USE\_16\_BIT\_TICKS* thì đặt cho thời gian cực đại delay là 0xFFFF, ngược lại delay sẽ lớn hơn 0xFFFFFFFF.
* Ngoài ra phần rất quan trọng là khai báo vị trí thanh ghi ngắt toàn cục, hàm cho phép và không cho phép ngắt.
* Vấn đề khác trong file là tạo hàm *ENTER\_CRITICAL()* và *EXIT\_CRITICAL()*. Khi bắt đầu đoạn bất ly cần cất thanh ghi ngắt vào ngăn xếp sau đó không cho phép ngắt toàn cục. Ngược lại, khi ra khỏi đoạn bất kỳ cần khôi phục thanh ghi ngắt từ ngăn xếp và cho phép ngắt nếu trước khi ngắt có cho phép. Không được thay đổi bất kỳ bit nào khác trong thanh ghi điều khiển ngắt.

### Triển khai bài toán và kết quả mô phỏng

#### Triển khai bài toán

* Phần cứng:
* Laptop Dell Inspiron 5559
* KIT STM32F103C8
* Phần mềm:
* Hệ điều hành Window 10
* IDE Keil C Version 5.22

#### Kết quả mô phỏng

Với các phần mềm được sử dụng như trên, bài toán đã được giải quyết đúng yêu cầu đề ra. Hai lần mô phỏng để xem đáp ứng của hệ điều hành thời gian thực FreeRTOS đều đúng như tiên liệu đề ra. Hình ảnh thực tế khi chạy:



*Hình 3.1 Kết quả mô phỏng*

# TÀI LIỆU THAM KHẢO

[1] Richard Barry, FreeRTOS.org – Copyright (C) 2003-2007, www.freertos.org

[2] Andrews S. Tanenbaum, Modern Operating Systems, second edition, Prentice Hall PTR.

[3] A. Burn & A.Wellings, Real Time Systems and Programing language, Addison Wesley, 1997.

[4] G.Olsson, Computer System for Automation and Control, G.Piani – Prentice – Hall, 1996.

[5] John A. Stankovic, Strategic Directions in Real-Time and Embedded Systems ACM Computing Surveys, Vol. 28, No. 4, December 1996.

[6] Jason McDonald, senior edition, Selecting an embedded RTOS, eg3.com.

[7] Real-time and Embedded Systems forum: www.opengroup.org/rtforum/