Lời cảm ơn

Mục lục

Danh mục hình vẽ và bảng biểu

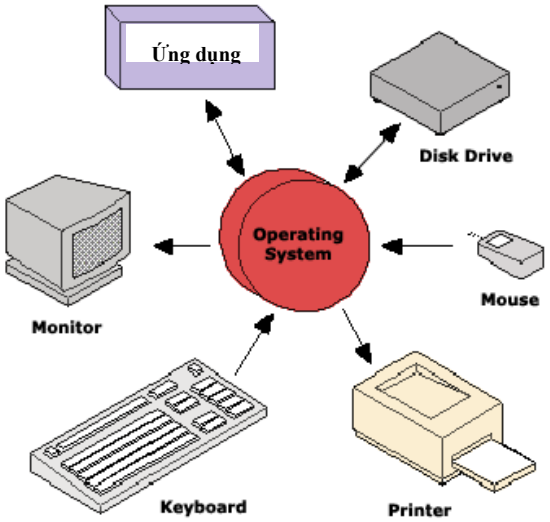
Mở đầu

Chú thích [PNH1]: Nói rõ về ý nghĩa của việc port lên VDK và sử dụng những phần mềm free

Phần I: Tổng quan về hệ điều hành thời gian thực

I. 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 1: 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

1. Hệ điều hành cho Mainframe

Hệ điều hành cho máy Mainframe là hệ điều hành ở mức cao nhất, loại máy tính này được thấy nhiều ở các trung tâm cơ sở dữ liệu lớn. Những máy tính này phân biệt với máy tính cá nhân ở giới hạn xử lý và sức chứa của chúng. Một máy tính Mainframe với hàng nghìn đĩa và hàng nghìn gigabybe dữ liệu là điều bình thường hầu như không có máy tính cá nhân thông thường nào có những tính năng mạnh này. Hầu hết mainframe dùng cho những ứng dụng rất lớn.

Hệ điều hành cho mainframe có tính định hướng cao cho việc xử lý nhiều việc mà mỗi việc cần lượng lớn I/O. Chúng thường được sử dụng ở những kiểu: mẻ, quản

lý xử lý và phân chia thời gian. Hệ thống mẻ là hệ thống mà những công việc xử lý đều đặn không có bất kỳ ảnh hưởng nào đến người sử dụng hiện tại. Nó được vận dụng để giải quyết một số lượng lớn các yêu cầu nhỏ, ví dụ như quá trình kiểm tra ở nhà băng hoặc sân bay. Mỗi phần của công việc thì nhỏ nhưng hệ thống phải xử lý hàng trăm hay hàng nghìn công việc nhỏ trên một giây. Hệ thống chia sẻ thời gian chấp nhận điều khiển từ xa để thực hiện công việc trên máy tính trong 1 lần, như những yêu cầu về dữ liệu. Những chương trình này gần với: hệ điều hành mainframe thường thực hiện tất cả. Ví dụ về hệ điều hành mainframe là OS/390, thế hệ sau của OS/360.

2. Hệ điều hành cho các Server

Dưới hệ điều hành mainframe một cấp là hệ điều hành chạy trên các server, một trong các loại máy tính cá nhân rất lớn, những máy trạm, hoặc kiểu mainframe. Chúng phục vụ nhiều user trong một thời điểm qua mạng và chấp nhận để các user chia sẻ nguồn phần cứng hay phần mềm. Các Server có thể cung cấp dịch vụ in ấn, dịch vụ file hoặc dịch cụ web. Các cung cấp Internet chạy trên nhiều server để hỗ trợ khách hàng và các Website sử dụng trong server để lưu trữ các trang Web và trả lời các yêu cầu đến. Điển hình là các hệ điều hành UNIX và Window2000. Linux cũng được sử dụng cho các server.

3. Hệ điều hành đa vi xử lý

Cách kết hợp nhiều tổ hợp máy tính phổ biến hiện nay là kết nối nhiều CPU trong một hệ thống. Phụ thuộc vào sự chuẩn xác kết nối thế nào và chia sẻ cái gì, những hệ thống này được gọi là máy tính song song, hệ đa máy tính hay hệ đa xử lý. Chúng cần những hệ điều hành đặc biêt nhưng thường những biến đổi này diễn ra trên hệ điều hành server, với những tính năng đặc biệt cho giao tiếp và kết nối.

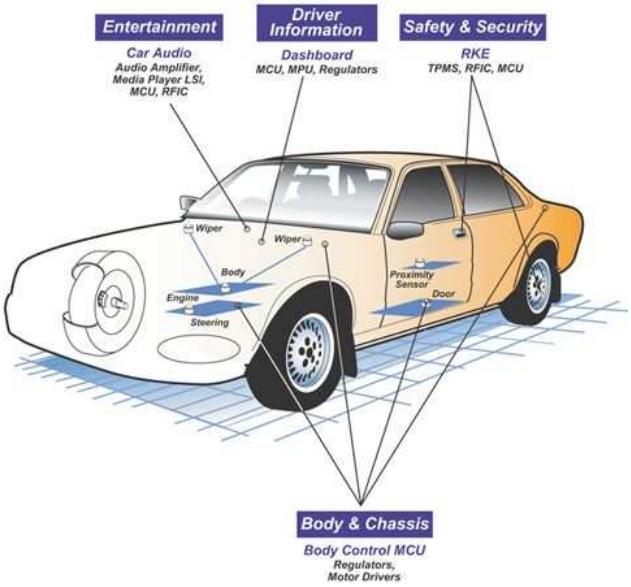
4. Hệ điều hành cho máy tính cá nhân

Loại tiếp theo là hệ điều hành cho máy tính cá nhân. Công việc của nó là cung cấp giao diện tốt cho một người sử dụng. Nó được dùng chủ yếu để soạn thảo văn bản, sửdụng bảng tính, và truy cập Internet. Ví dụ như Windows98, hệ điều hành Macintosh và Linux. Hệ điều hành máy tính cá nhân được biết đến rất rộng rãi do chỉ cần ít những chỉ dẫn. Trên thực tế, nhiều người có trình độ khác nhau đều có thể sử dụng được.

5. Hệ điều hành thời gian thực

Một loại hệ điều hành khác là hệ điều hành thời gian thực. Hệ điều hành này được mô tả là quản lý thời gian như thông số chìa khóa. Ví dụ như trong hệ thống điều khiển công nghiệp, máy tính thời gian thực cần phải thu thập dữ liệu về quá trình sản xuất và sử dụng dữ liệu đó để điều khiển các máy trong xí nghiệp theo đó các deadline

cứng phải được thỏa mãn. Một ví dụ khác với dây chuyền lắp ráp ôtô, một hành động nhất định phải được thực hiện thời điểm nhất định nào đó, nếu robot thực hiện sớm quá hoặc muộn quá thì chiếc xe sẽ bị lỗi có thể dẫn đến phá hủy. Trong hệ thống mà hành động buộc phải thực hiện vào thời điểm nhất định (hoặc trong khoảng cho phép) thì đó là hệ thống thời gian thực cứng. Một hệ thời gian thực khác là hệ thời gian thực mềm, trong đó có những đáp ứng deadline có thể không thỏa mãn toàn bộ, hệ truyền tiếng nói số hay đa phương tiện là một ví dụ cho hệ này. Điển hình cho hệ điều hành thời gian thực là VxWorks và QNX.



Hình 2: Sơ đồ hệ thống thời gian thực trong ô tô

6. Hệ điều hành nhúng

Tiếp tục với hệ nhỏ hơn, chúng ta đến với những máy tính palmtop và hệ nhúng. Palmtop hay PDA là loại máy tính nhỏ hoạt động với số lượng nhỏ các chương trình. Hệ nhúng chạy trên những máy tính điều khiển các thiết bị nói chung không giống với máy tính, như TV sets, lò vi sóng, điện thoại di động. chúng thường có tính thời gian thực nhưng kích cỡ, bộ nhớ và sự hạn chệ năng lượng làm chung đặc biệt. Những hệ thông dụng là PalmOS và Windows CE.

7. Hệ điều hành cho thẻ thông minh

Hệ điều hành nhỏ nhất chạy trên các thẻ thông minh, như credit card, thiết bị bao gồm chip CPU. Chúng quản lý rất nghiêm ngặt năng lượng sử dụng và cách nén

bộ nhớ. Một số hệ chỉ sử dụng cho một chương trình duy nhất như thanh toán điện tử, nhưng những hệ khác phục vụ nhiều chương trình trong cùng một thẻ thông minh. Đó thường là những hệ thống độc quyền.

Một số thẻ thông minh được viết trên Java, trên ROM của thẻ thông minh cần chương trình dịch cho JVM. Java applet được tải về thẻ và được dịch ra để sử dụng. Một số thẻ có thể chạy nhiều Java applet trong cùng một thời điểm, để quản lý nhiều chương trình cần bộ lập lịch. Cần quản lý và đảm bảo cho hệ thống khi có hai hay nhiều applet chạy trong cùng một thời điểm. Những vấn đề này cần được quản lý bởi hệ điều hành hiện tại trên thẻ.

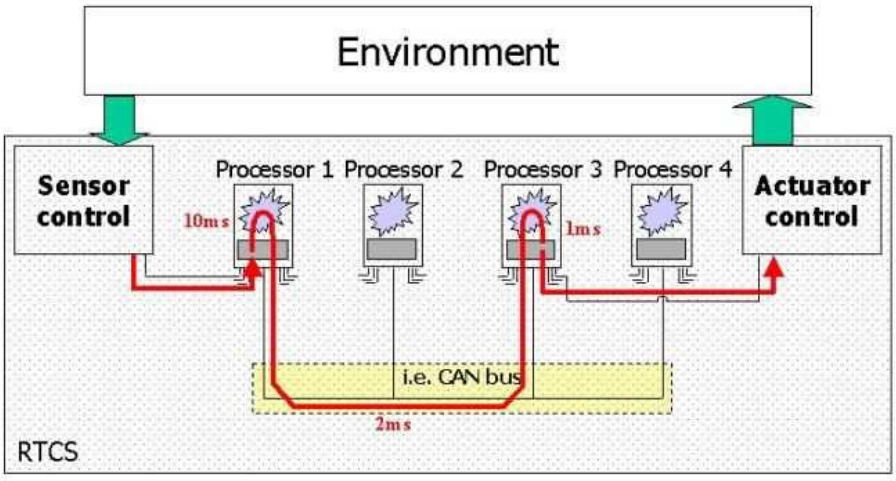
II. Tìm hiểu hệ điều hành thời gian thực

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

1. Hệ điều hành thời gian thực (RTOS)

a) 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.

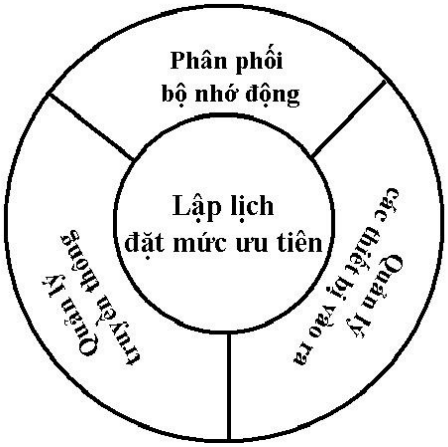


Hình 3: Hệ điều hành thời gian thực trong điều khiển

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 4: Sơ đồ chức năng của hệ điều hành thời gian thực

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.

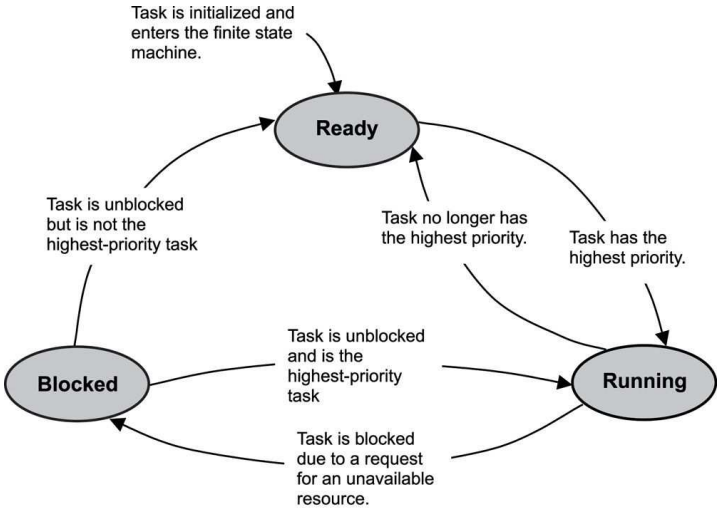
b) ***Các đặc điểm của RTOS***

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 5: Trạng thái các tác vụ

- **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.

2. 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,...

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	phụ thuộc môi trường	phụ thuộc vi diều khiển
Đoạn dữ liệu	đoạn ngắn	đoạn dài
Bắt lỗi	tự động	phụ thuộc người dùng

Bảng 1: Bảng so sánh hai loại hệ điều hành thời gian thực

3. 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.

4. Các hệ điều hành thời gian thực phổ biến

a) LynxOS

LynxOS [12] là loại hệ điều hành thời gian thực cứng, làm việc với Unix và Java. Các thành phần LynxOS được thiết kế tiền định (thời gian thực cứng) có nghĩa là phải đáp ứng trong khoảng thời gian biết trước. Thời gian đáp ứng tiền định được đảm bảo ngay cả khi cổng vào ra phải tải nặng do mô hình đơn luồng của kernel, làm cho chương trình phục vụ ngắt rất ngắn và nhanh.

LynxOS 3.0 phát triển từ kiến trúc nguyên gốc (chỉ gồm một mô đun làm tất cả mọi việc) của các phiên bản ban đầu lên đến thiết kế microkernel. Dung lượng của microkernel là 28Kbyte và cung cấp các dịch vụ cơ bản như lập lịch, phần luồng ngắt và đồng bộ. Các dịch vụ khác được cung cấp bởi các mô đun dịch vụ nhỏ hơn gọi là Kernel Plug-Ins (KPIs). Nhờ thêm KPIs vào microkernel, hệ thống có thể cấu hình để hỗ trợ vào ra và quản lý file, TCP/IP,... nó hoạt động như hệ điều hành UNIX đa nhiệm giống phiên bản trước.

b) Vxworks

VxWorks [11] là hệ điều hành thời gian thực được sử dụng trong hệ nhúng điển hình có thể kể đến hệ điều hành cho robot trên sao Hỏa năm 1997. VxWorks có thể hỗ trợ khách hàng dễ dàng và có thể chạy trên hầu hết các vi xử lý được thiết kế tính toán phân tán. Nó còn được sử dụng điều khiển mạng và các thiết bị truyền thông, các thiết bị đo và kiểm tra, thiết bị ngoại vi, hệ thống tự động hoá, thiết bị hàng không và vũ trụ, và các loại sản phẩm tiêu dùng khác.

Vxworks có vài điểm giống với Unix như shell, các hàm debugging, quản lý bộ nhớ, giám sát thực thi và hỗ trợ đa nhiệm. Hệ điều hành bao gồm lõi cho lập lịch đa nhiệm, trả lời ngắt, truyền thông các tiến trình và hệ thống file. Lập trình trong VxWorks có thể khó cho người mới lập trình vì lập trình viên phải viết code trên yêu cầu cơ bản tối thiểu. Tuy nhiên, thực tế cho thấy chỉ một lượng nhỏ nội dung cần lưu trữ và phục hồi trong VxWorks, nó sử dụng ít tài nguyên hơn Unix, vì vậy nó chạy nhanh hơn.

c) QNX/Neutrino

Mục đích chính của QNX Neutrino [11] là phổ biến rộng rãi hệ thống mở POSIX API theo mô hình bậc thang mạnh cho các hệ thống trên diện rộng từ các hệ nhúng tài nguyên hẹp đến các môi trường máy tính phân tán mạnh nhất. Hệ điều hành QNX Neutrino hỗ trợ nhiều dòng vi xử lý như Intel x86, ARM, XScale, PowerPC, MIPS và SH-4. QNX Neutrino là chuẩn mực cho các ứng dụng nhúng thời gian thực. Nó có thể co giãn tới một kích thước rất nhỏ và cung cấp nhiều tác vụ hoạt động đồng thời, các tiến trình, điều khiển lập lịch thứ tự ưu tiên và chuyển ngữ cảnh nhanh… tất cả các thành phần cơ bản của một ứng dụng nhúng thời gian thực.

QNX Neutrino là rất mềm dẻo. Người phát triển có thể dễ dàng tuỳ biến OS cho phù hợp với ứng dụng của mình. Từ một cấu hình cơ bản nhất của một nhân nhỏ (microkernel) cùng với một vài module nhỏ tới một hệ thống được trang bị phát triển mạng diện rộng để phục vụ hàng trăm người dùng.

QNX Neutrino đạt được mức độ chuyên môn hoá về tính hiệu quả, tính module hoá và tính đơn giản với hai yếu tố cơ bản sau:

- Kiến trúc microkernel của QNX đảm bảo cách ly bộ nhớ các process, không cho chúng sơ ý phá hoại lẫn nhau
- Giao tiếp liên quá trình dựa trên thông điệp

d) uCOS

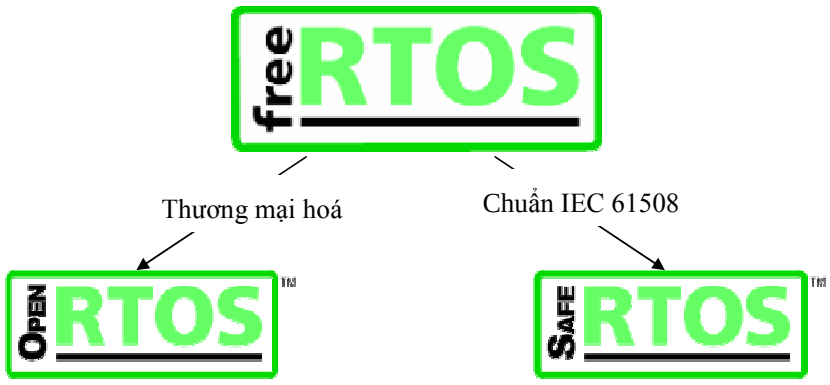
µC/OS [10] là nhân một hệ điều hành thời gian thực do J. Labrosse xây dựng. Nó là nhân hệ điều hành thời gian thực có tính khả chuyển cao, mềm dẻo, có tính năng ưu tiên, thời gian thực và đa tác vụ. µC/OS hỗ trợ các tính năng sau: bộ lập lịch, truyền thông điệp, đồng bộ và chia sẻ dữ liệu giữa các tác vụ, quản lý bộ nhớ, các thiết bị vào ra, hệ thống file, mạng.

µC/OS hỗ trợ nhiều bộ vi xử lý và rất gọn nhẹ để sử dụng cho nhiều hệ thống nhúng. Phiên bản µC/OS-II tăng cường thêm khả năng đặt tên tới mỗi đối tượng của nhân. Đặc biệt với phiên bản V2.6, chúng ta có thể gán tên tới một tác vụ, đèn báo, mailbox, hàng đợi, một nhóm sự kiện hay một vùng bộ nhớ. Do đó, bộ phát hiện lỗi có thể hiển thị tên của những đối tượng đó, và cho phép chúng ta nhanh chóng xem thông tin về những đối tượng. Ngoài ra, V2.6 cho phép bộ sửa lỗi đọc thông tin cấu hình của một ứng dụng và hiển thị những thông tin đó.

Phần II: Tìm hiểu chi tiết về FreeRTOS

I. Tổng quan về FreeRTOS

1. Khái niệm FreeRTOS



Hình 6: Sơ đồ phát triển của FreeRTOS

FreeRTOS [1] 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:

	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

Bảng 2: Các hỗ trợ khác nhau từ FreeRTOS và OpenRTOS

2. Các đặc điểm của FreeRTOS

a) Các ch*ức 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] :

- Lõi 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.

b) T*ại 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.

c) *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.

3. 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)

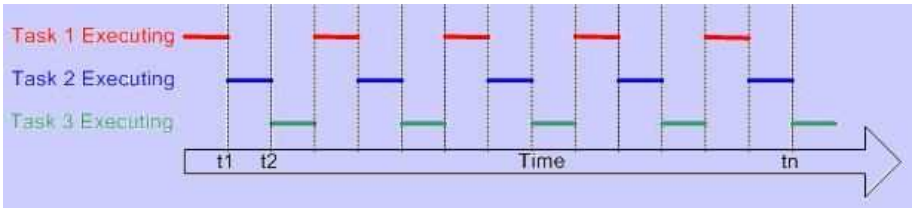
a) *Đa nhiệm*

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.

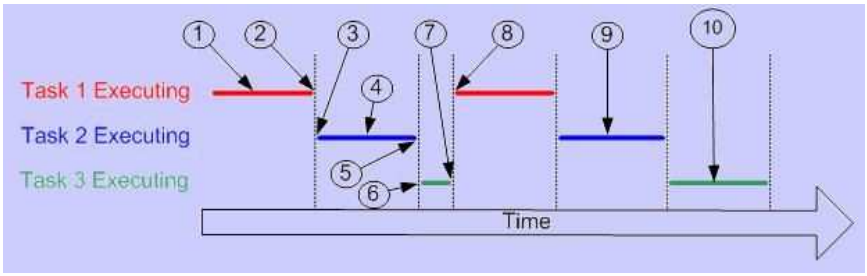


Hình 7: Sơ đồ phân chia thời gian các tác vụ thực hiện

b) L*ập 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 8: Sơ đồ chuyển giao các tác vụ

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

c) Chuy*ển 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.

d) 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.

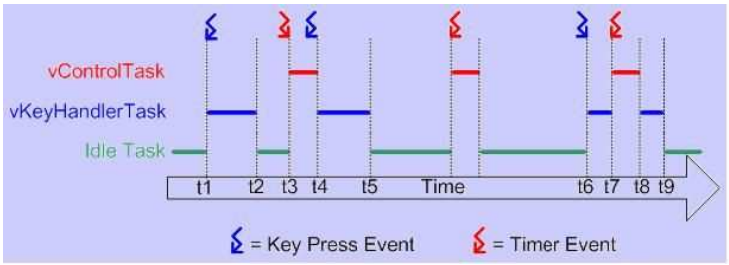
Để thực hiện những mục tiêu trên, kỹ sư lập trình phải gán quyền ưu tiên cho mỗi một tác vụ. Sau đó cơ chế lập lịch của hệ RTOS chỉ đơn giản là xác định tác vụ có quyền ưu tiên cao nhất được phép thi hành ở thời điểm đang xử lý. Điều này dẫn đến cần chia sẻ thời gian xử lý một cách công bằng giữa các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

e) B*ộ 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 9: Sơ đồ phân chia các sự kiện theo thời gian

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.

4. Cách phân phối tài nguyên của FreeRTOS

a) 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.

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

Bảng 3: Bảng phân phối RAM của heap1

heap 1 r*ất 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.

heap_2.c thích h*ợp cho ứng dụng thời gian thực tạo task một cách linh động*.

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

Bảng 4: Bảng phân phối RAM của heap2

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

b) Cách l*ập lịch*

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:

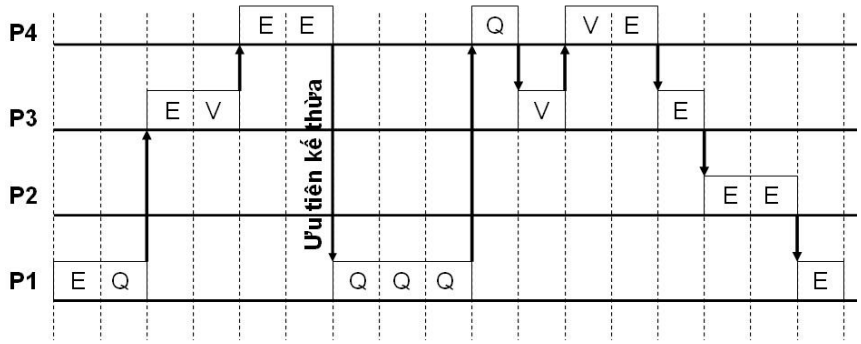
Tiến trình	Mức ưu tiên	Chuỗi tài nguyên	Thời điểm bắt đầu
P4	4	EEQVE	4
P3	3	EVVE	2
P2	2	EE	2
P1	1	EQQQE	0

Bảng 5: Bảng phân chi tiết các tiến trình

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 10: 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.

5. So sánh hệ FreeRTOS với hệ điều hành thời gian thực uCOS

Ta so sánh hai hệ điều hành này trên các cơ sở sau:

- Thời gian đáp ứng sau khi gọi ngắt, chuyển ngữ cảnh gữa các task.
- Dung lượng bộ nhớ chương trình khi dịch ra file hex nạp vào chip.
- Lượng RAM cung cấp cho bộ lập lịch, khi tạo task mới, tạo hàng đợi mới, tạo semaphore mới.

Trong ba yếu tố này điểm coi trọng nhất là yếu tố đáp ứng thời gian, sau đó là lượng RAM cần cung cấp cho mỗi hoạt động và cuối cùng là bộ nhớ chương trình. Do hai yếu tố về tài nguyên ta có thể chọn chip phù hợp, còn yếu tố về thời gian là yếu tố phụ thuộc vào bản chất của hệ điều hành. Khác với bộ nhớ chương trình, RAM được cung cấp hạn chế và quy định cho từng tác vụ con bộ nhớ chương trình hầu như là tĩnh và được cung cấp không ngặt nghèo như RAM.

a) *Dung lượng bộ nhớ chương t*rình

Dung lượng bộ nhớ chương trình cho mỗi lõi hệ điều hành tuỳ thuộc vào từng trình dịch khác nhau. Với so sánh này ta dựa trên vi điều khiển PIC18F452 và trình dịch MPLAB C18.

Lõi của FreeRTOS chiếm cỡ 10 KBytes bộ nhớ chương trình.

Lõi của uCOS chiếm cỡ KBytes bộ nhớ chương trình.

b) *Dung lượng RAM* cung c*ấp*

Với dung lượng RAM cung cấp ta có bảng sau:

Mục	FreeRTOS (byte)	uCOS (byte)
Bộ lập lịch	83	
Mỗi task mới	20 (2 byte cho tên) + ngăn xếp	
Mỗi mức ưu tiên	16	
Mỗi hàng đợi	45 + vùng lưu trữ hàng đợi	
Mỗi semaphore	45	

Bảng 6: So sánh lượng RAM cung cấp giữa FreeRTOS và uCOS

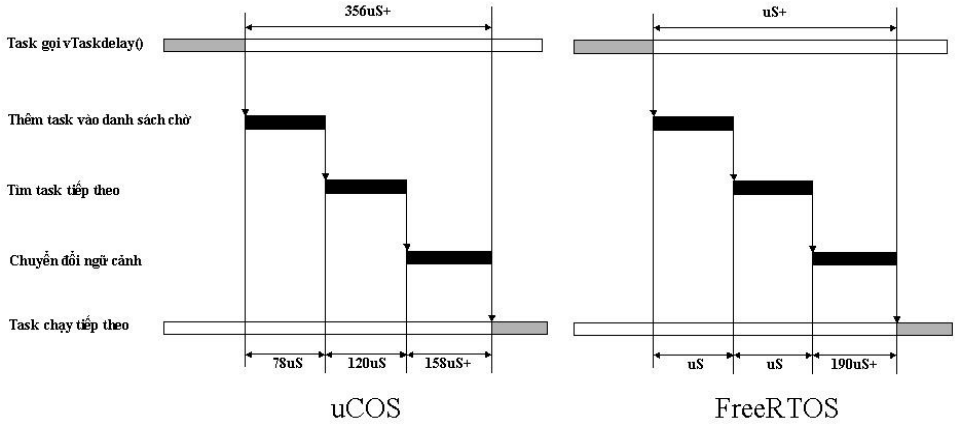
c) T*hời gian đáp ứng*

Ta cần so sánh hai kiểu đáp ứng thời gian chính:

Đáp ứng thời gian khi một task đã thực hiện xong chu kỳ của mình và cho task khác chạy. Các công việc chuyển đổi này gồm 3 bước trung gian

- Thêm task đã thực hiện xong vào danh sách task chờ.

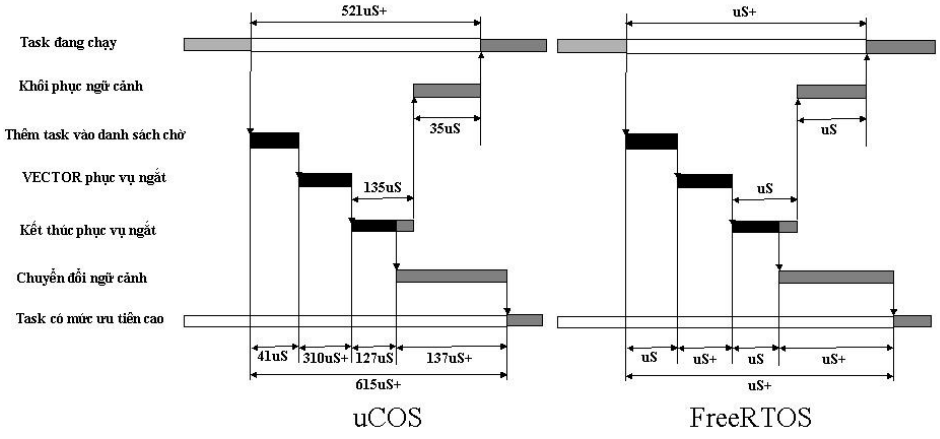
- Bộ lập lịch tìm task tiếp theo để thực hiện
- Chuyển đổi ngữ cảnh



Hình 11: Bảng so sánh thời gian đáp ứng 1

Đáp ứng thời gian khi gọi ngắt trong lúc một task đang thực hiện. Công việc này gồm 4 bước trung gian:

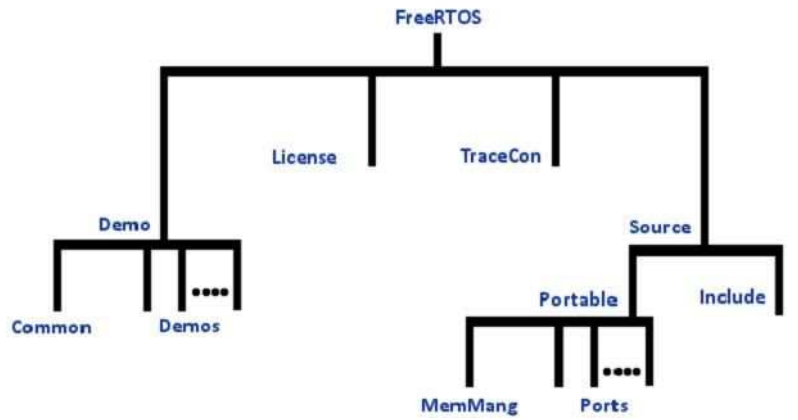
- Thêm task bị ngắt vào danh sách task chờ
- VECTOR phục vụ ngắt, gồm cả việc lưu trữ ngữ cảnh của task đang chạy.
- Kết thúc phục vụ ngắt
- Trong kết thúc phục vụ ngắt cần tìm xem có ngắt nào có mức ưu tiên cao hơn không, nếu có task có mức ưu tiên cao hơn thì chuyển đổi ngữ cảnh, ngược lại cần khôi phục ngữ cảnh.



Hình 12: Bảng so sánh thời gian đáp ứng 2

II. 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 13: Sơ đồ các file và thư mục trong gói FreeRTOS.zip tải về

1. 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.

a) 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.

b) 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 tskIDLE_PRIORITY(), taskYIELD(), taskENTER_CRITICAL(),… và định nghĩa một số hằng số để sử dụng.

Các task t*ạo 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 cửa 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().

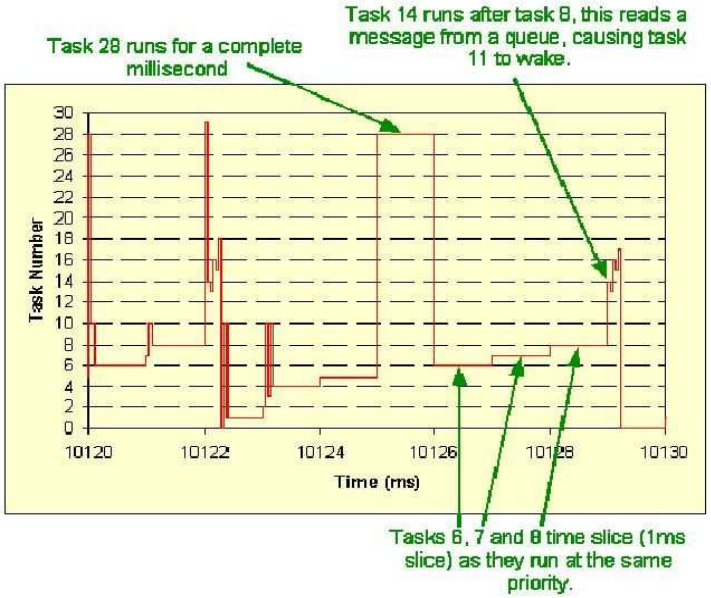
Các task điều khiển API: tạo ra các hàm điều khiển API cụ thể là các nhiệm vụ như sau:

- 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 phuc. 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 ti*ện 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 14: 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.

c) 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 để sắp 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.

d) 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.

e) 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

2. Các file còn lại trongkernel 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

a) projdef.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.

b) 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.

c) 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().

III. Port FreeRTOS lên vi điều khiển PIC18F452

1. Một số chú ý khi port FreeRTOS lên vi điều khiển

Chú thích [PNH2]: Xem lại phần này

a) Qu*ản lý và sử dụng RAM [1]*

Hàng đợi sử dụng nhiều RAM? do quản lý sự kiện được xây dựng thành chức năng hàng đợi. Có nghĩa là cấu trúc dữ liệu hàng đợi bao gồm toàn bộ RAM mà những hệ thống thời gian thực khác thường phân phối tách biệt. Ở đây không có khái niệm về khối điều khiển sự kiện trong FreeRTOS.

FreeRTOS s*ử dụng bao nhiêu ROM?* phụ thuộc vào trình biên dịch và kiến trúc từng chương trình. Riêng kernel sẽ sử dụng khoảng 4KBytes ROM khi sử dụng cùng một cấu hình trạng thái.

Để giảm lượng RAM sử dụng ta làm như sau:

- Đặt configMAX_PRIORITIES và configMINIMAL_STACK_SIZE (nằm trong portmacro.h) đến giá trị nhỏ nhất chấp nhận được trong ứng dụng.
- Nếu được hỗ trợ bởi trình dịch – định nghĩa tác vụ chức năng và main() bằng “naked”. Nó ngăn không cho trình dịch nhớ những thanh ghi vào ngăn xếp khi chương trình chạy. Vì chương trình không bao giờ kết thúc, các thanh ghi sẽ không bao giờ được phục hồi và không bị yêu cầu.
- Lấy lại ngăn xếp được sử dụng bởi main(). Ngăn xếp sử dụng ở trên lúc chương trình bắt đầu không được yêu cầu lần nào khi bộ lịch trình khởi động (trừ khi ứng dụng có gọi vTaskEndScheduler() mà chỉ được hỗ trợ

trực tiếp trong sự sắp xếp cho PC và Flashlite port). Mọi tác vụ đều có ngăn xếp cấp phát riêng nhưng ngăn xếp phân phối cho main() tồn tại để sử dụng một lần khi bộ lịch trình bắt đầu.

- Giảm ngăn xếp sử dụng bởi main() xuống mức nhỏ nhất. Idle task tự động được tạo ra khi task ứng dụng đầu tiên được tạo ra. Ngăn xếp sử dụng cho chương trình khi bắt đầu (trước khi bộ lịch trình bắt đầu) phải đủ lớn cho lệnh gọi lồng đến xTaskCreate(). Tạo idle task thủ công có thể chỉ cần một nửa ngăn xếp yêu cầu. Tạo idle task bằng tay như sau:
    1. Xác định vị trí chức năng prvInitialiseTaskList() trong Source/task.c.
    2. Idle task được tạo ra ở dưới cùng của chức năng gọi bởi xTaskCreate(). Cắt dòng này và paste lại vào main()
- Số task đưa ra đều có ý nghĩa. Idle task không cần thiết nếu:
    1. Ứng dụng có task không bao giờ bị khóa
    2. Ứng dụng không bao giờ gọi vTaskDelete()
- Giảm dung lượng dữ liệu bằng cách định nghĩa portBASE_TYPE (điều này có thể tăng thời gian thực hiện)
- Có những ngắt không quan trọng khác có thể được thực hiện (ví dụ như hàng đợi mức ưu tiên tác vụ không phụ thuộc vào quản lý sự kiện), nhưng nếu giảm cấp xuống thì sẽ cần nhiều RAM hơn!

M*ỗi task được phân phối RAM như thế nào?* Để tạo task thì kernel có 2 lệnh gọi đến pvPortMalloc(). Thứ nhất để chỉ định khối điều khiển task, thứ hai là chỉ định ngăn xếp task.

M*ỗi hàng đợi được phân phối RAM như thế nào?* Để tạo hàng đợi, kernel có hai lệnh gọi đến pvPortMalloc(). Thứ nhất để chỉ định cấu trúc hàng đợi, thứ hai là vùng cất giữ của hàng đợi (dung lượng của nó là thông số đến xQueueCreate()).

Ngăn xếp nên lớn bao nhiêu? Điều này hoàn toàn phụ thuộc vào ứng dụng cụ thể và không dễ để tính được. Nó phụ thuộc vào độ sâu của phép gọi chương trình, số biến cục bộ, số thông số trong chương trình gọi, yêu cầu của ngăn xếp ngắt... Ngăn xếp phải đủ lớn để chứa ngữ cảnh thực hiện (tất cả thanh ghi quá trình). Ngăn xếp của mỗi task được phân bổ 0xa5 bytes trong lúc tạo mà cho phép mức cao nhất có thể thấy được sử dụng phù hợp với công cụ debug.

b) Tick và Idle Hooks

Có th*ể thêm code vào RTOS idle task?* Idle task được thực hiện trong Source/task.c (tìm prvIdleTask). Ta có thể thêm những gì cần vào, code thêm vào sẽ không làm idle task bị khóa.

Có th*ể thêm code để đặt vi điều khiển vào trạng thái power save?* Cách thuận tiện nhất để hoàn tất là sử dụng idle task hook.

Có th*ể sử dụng idle task hook* bằng cách đặt configUSE_IDLE_HOOK lên 1 trong FreeRTOSconfig.h.

Có th*ể sử dụng tick hoặc context switch hook* bằng cách đặt configUSE_TICK_HOOK lên 1 trong FreeRTOSconfig.h.

c) B*ộ lập lịch*

Làm th*ế nào với các task có mức ưu tiên ngang nhau trong bộ lập lịch?* Ưu tiên quay vòng. Mỗi tác vụ sẽ được chia sẻ thời gian bằng nhau trong bộ xử lý.

Các task mà chia s*ẻ idle priority scheduled?* Như các task chia sẻ bất kỳ mức ưu tiên khác. Nên đặt nó chú ý hơn vì khi preemptive scheduler được sử dụng, idle task sẽ chạy trong time slot của nó, không thay đổi gì nếu các task khác chia sẻ mức ưu tiên của chúng.

d) Các thanh ghi ph*ục vụ ngắt (ISR's)*

Chuy*ển đổi ngữ cảnh có thể được thực hiện trong ISR:* mỗi port chứa ngắt đơn giản drive cổng nối tiếp mà được sử dụng như một ví dụ cho kiến trúc vi điều khiển. Tức là các drive đã được viết với mục đích kiểm tra chuyển đổi ngữ cảnh từ ISR nhưng không được tốc độ tối ưu

Các ng*ắt có thể gọi lồng nhau được không?* Những port tải xuống không thực hiện gọi lồng các ngắt. Trong hầu hết trường hợp sử dụng của kernel thời gian thực nhanh gỡ bỏ việc gọi ngắt lồng. Những ngắt lồng cho thấy sự không chắc chắn trong nhu cầu sử dụng ngăn xếp và phức tạp trong việc phân tích hành vi của hệ thống. Thay vào đó, người ta thích các kênh điều khiển ngắt (interrupt handlers) không làm gì cả nhưng thu thập dữ liệu sự kiện, đưa dữ liệu cho các task và xóa nguồn ngắt. Điều này cho phép các ngắt có thể thoát được nhanh chóng các trì hoãn bất ngờ trong quá trình tính toán dữ liệu sự kiện. Task level có thể được thực hiện bằng cách cho phép ngắt, không cho ngắt lồng.

Sự phối hợp này có thêm những thuận lợi về sự mềm dẻo trong việc xử lý ưu tiên hóa các sự kiện. Mức ưu tiên các tác vụ được sử dụng thay cho sự ưu tiên phụ thuộc vào mức ưu tiên ấn định cho mỗi nguồn ngắt bởi mục tiêu xử lý. Quyền ưu tiên của các tác vụ nắm bắt ngắt có thể được chọn cao hơn mức thông thường trong phạm vi cùng một ứng dụng, cho phép việc nắm bắt ngắt quay lại trực tiếp từ tác vụ nắm bắt ngoại vi. Ngắt có thể ngắt các tác vụ bình thường, nhận dữ liệu, sau đó quay về tác vụ nắm bắt ngắt. Khi tác vụ nắm bắt ngắt hoàn thành, các tác vụ trước ngắt tự động thực hiện tiếp từ điểm bị ngắt. Quá trình xử lý ngắt tự nó và tác vụ nắm bắt ngắt liền nhau

theo thời gian như là các xử lý tự được thực hiện trong ngắt nhưng sử dụng nhiều cơ cấu đơn giản hơn. Trong trường hợp trả lời ngắt rất nhanh được yêu cầu cho đích xác thiết bị ngoại vi thì mức ưu tiên ngoại vi có thể được nâng lên. Điều này có nghĩa là xử lý của thiết bị ngoại vi sẽ không bị trễ bởi hoạt động của kernel.

2. Các file cần để port lên vi điều khiển PIC18 sử dụng MPLAB

Khi port cho PIC cần 3 file chính như sau:

- FreeRTOSconfig.h: file này định nghĩa riêng cho từng ứng dụng. Các định nghĩa này phải được phù hợp với phần cứng
- 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.
- portmacro.h: định nghĩa cho riêng phần port. Các định nghĩa này cấu hình cho FreeRTOS đúng với từng phần cứng và từng trình dịch

a) FreeRTOSconfig.h

File được tạo ra với hai nhiệm vụ chính:

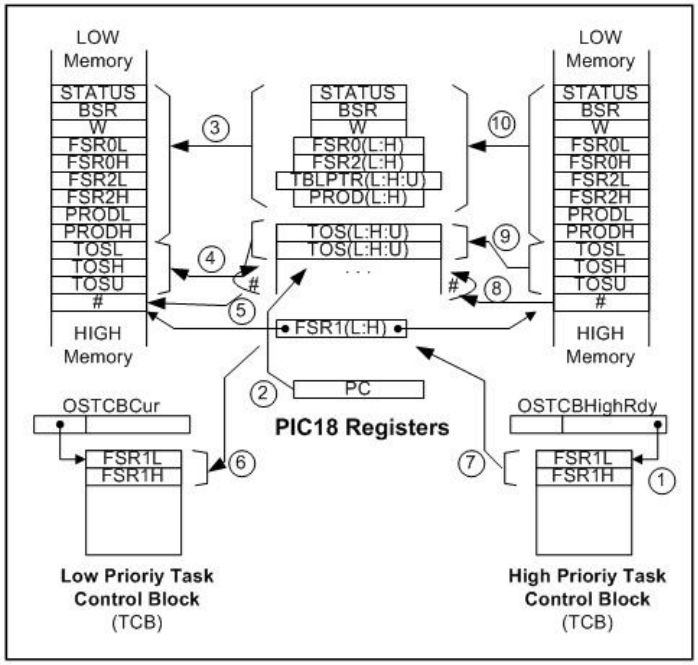
- Định nghĩa các thông số, các chức năng cơ bản mà FreeRTOS hỗ trợ được định yêu cầu trong FreeRTOS.h. Các hàm, macro này nếu muốn khai báo có sử dụng thì định nghĩa là , ngược lại là 0. Để xem thêm các thông số cần khai báo cho phần này xem FreeRTOS.h <PhầnII.I.a>
- Các thông số cần định nghĩa cho từng 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.

b) port.c

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

- Khai báo để cài đặt phần cứng cho tick.

- 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.
- Định nghĩa này chỉ cho các bit nằm trong INTCON, ngắt toàn cục.
- Khai báo 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.
- 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.
- Cài đặt phần cứng để cho phép tick. Và có thêm chương trình phục vụ ngắt để duy trì tick và thức hiện chuyển đổi ngữ cảnh tick nếu sử dụng kiểu preemptive.



Hình 15: Sơ đồ chuyển đổi ngữ cảnh

- 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(). Các thao tác như trong hình dưới đây. Tại (1), bộ lập lịch tải con trỏ trỏ tới TCB của task mới về. Sau đó, tại (2), thanh

ghi PC được cất vào ngăn xếp. Tiếp đến tại (3) và (4), chương trình cất các thanh ghi quan trọng liên quan và ngăn xếp cứng của chip, cùng với đó tại bước (5), ta lưu số lượng ngăn xếp cứng vừa được lưu. Bước cuối cùng là cất đỉnh của ngăn xếp vừa lưu vào TCB của task vừa bị ngắt. 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. Với hàm khôi phục ngữ cảnh ta thao tác gần như ngược lại với lưu trữ. 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.

- Cài đặt ngăn xếp cho 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.
- 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.
- Dừng bộ lập lịch tức là hủy toàn bộ cài đặt cho phần cứng/ISR đã được thực hiện bởi xPortStartScheduler() vì thế phần cứng được trở 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.
- 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. Có thể giải thích hình vẽ như sau : khi task có mức ưu tiên thấp đang chạy, có tín hiệu báo ngắt. Khi bắt đầu bị ngắt thì task sẽ được thêm vào danh sách chờ. Sau đó gọi hàm phục vụ ngắt, riêng với bước này, task bị ngắt sẽ được lưu ngữ cảnh rồi mới phục vụ ngắt. Khi kết thúc ngắt, bộ lập lịch sẽ tìm task tiếp theo chạy. Nếu không có task nào có mức ưu tiên cao hơn thì khôi phục task tiếp tục chạy. Ngược lại, sẽ chuyển đổi cho task có mức ưu tiên cao hơn chạy (xem hình 11).
- 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.

c) portmacro.h

File này định nghĩa cho riêng phần port. Các định nghĩa này cấu hình cho FreeRTOS đúng với từng phần cứng và từng trình dịch. Các cài đặt này không được thay 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. Ví dụ như đối với PIC18F452 cần khai báo vị trí thanh ghi ngắt toàn cục là 0x80, bit cho phép ngắt toàn cục hay không là INTCONbits.GIEH.
- 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 ly 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.

Phần III: Mô phỏng và giao diện hỗ trợ port FreeRTOS lên PIC

I. Mô phỏng port FreeRTOS lên vi điều khiển PIC

1. Phân tích bài toán mô phỏng

a) 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 một cách dễ dàng.

b) Bài toán mô phỏng

Từ những ý tưởng và mục tiêu mô phỏng trên. Ta đặt ra bài toán với 5 tác vụ:

- Tác vụ 1: Reset Watdog Timer. Tác vụ này sẽ không cần thiết trong chương trình ta disable Watdog Timer ngay từ đầu nhưng sử dụng Watdog với 2 mục đích. Thứ nhất đây là mô phỏng cho hệ điều hành thời gian thực nên chức năng bắt lỗi và chạy ổn định là khá quan trọng, với Watdog Timer hệ thống có thể thoát ra khỏi trạng thái dead lock, khôi phục trạng thái ban đầu. Thứ hai, khi ta sử dụng reset Watdog là tác vụ ở mức ưu tiên thấp nhất, nếu hệ điều hành thời gian thực không đảm bảo chạy đúng cho các tác vụ thì tác vụ này sẽ bị ảnh hưởng đầu tiên. Nếu Watdog bị reset ta sẽ thấy ngay được kết quả mô phỏng.
- Tác vụ 2: Nhân chia liên tục bốn số 32 bit và so sánh với kết quả đúng. Tác vụ này được đưa ra do với vi điều khiển PIC18 làm các thao tác tính toán trên số 32 bit mất rất nhiều thời gian của bộ xử lý toán học. Khi mô phỏng ta sẽ thay đổi chu kỳ làm việc của tác vụ. Với chu kỳ dài thì hệ điều hành thời gian thực còn đảm trách được, còn khi giảm chu kỳ thực hiện xuống thấp sẽ

thấy ngay lỗi của hệ điều hành không lập lịch đủ cho các tác vụ hoàn thành công việc.

- Tác vụ 3: Nháy LED theo chu kỳ. Tác vụ này được đặt ra nhằm mô phỏng tác vụ theo chu kỳ và kiểm tra kết quả của tác vụ hai. Cụ thể tác vụ bố trí như sau, có bốn LED được nối với bốn chân vi điều khiển, mức ưu tiên của tác vụ 2 sẽtương đương với đèn mấy sang. Nếu tác vụ 2 chạy với kết quả đúng thì LED sẽ nháy theo 1 chu kỳ nhất định, còn tác vụ 2 chạy sai thì sẽ theo một chu kỳ khác nhanh hơn hẳn chu kỳ cũ.
- Tác vụ 4: Nháy LED theo sự kiện, tức là có một công tắc ở ngoài nối vào một chân của vi điều khiển, nếu được đóng mạch thì LED sẽ nháy, còn không được đóng mạch thì LED sẽ sáng.
- Tác vụ 5: truyền thông qua cổng USART, nhằm mô phỏng tác vụ truyền thông trong FreeRTOS.

2. Triển khai bài toán và kết quả mô phỏng

a) Tri*ển khai bài toán*

Do FreeRTOS là hệ điều hành thời gian thực mã nguồn mở nên ta xác định các công cụ để đi đến chương trình cũng nên sử dụng các bản miễn phí. Để có giao diện lập trình ta sử dụng MPLAB IDE 8.0 [9] là bản IDE miễn phí của hãng Microchip [8], cùng với đó ta sử dụng trình dịch là C18 Student [9] cũng là bản miễn phí.

Với bài toán đặt ra ở trên ta đặt ra các yêu cầu cần giải quyết cho hệ điều hành thời gian thực như sau:

STT	Tên tác vụ	Mức ưu tiên	Chu kỳ hoạt động	Độ sâu ngăn xếp
1	Reset Watdog Timer	Idle	20ms	105 byte
2	Tính toán các số 32 bit	Idle + 3	100ms (bài 1) 1ms (bài 2)	105 byte
3	Nháy LED theo task 2	Idle + 1	1000ms nếu đúng 100ms nếu sai	105 byte
4	Nháy LED khi bấm nút	Idle + 2	200ms	105 byte
5	Truyền thông USART	Idle + 1	200ms	105 byte

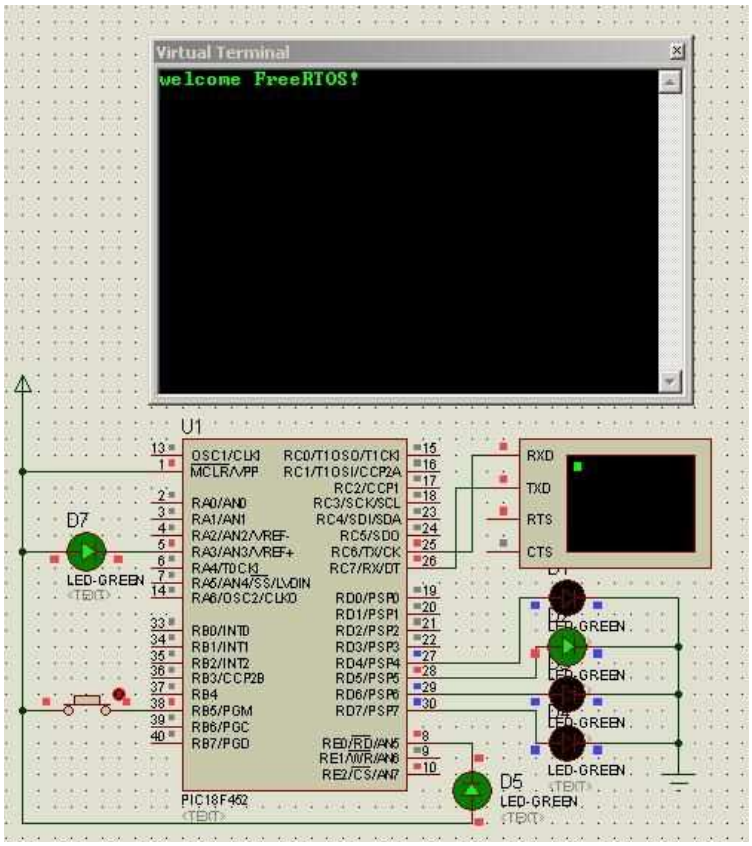
Bảng 7: Mô tả chi tiết về các task mô phỏng

Các thông số được đặt là hoàn toàn để mô phỏng, tùy từng ứng dụng cụ thể ta sẽ tính toán các thông số cho phù hợp. Để mô phỏng được khả năng của hệ điều hành thời gian thực ta sẽ thay đổi chu kỳ hoạt động của task 2.

Sau khi sử dụng các phần mềm miễn phí của Microchip để có được file *.hex, ta sử dụng phần mềm Proteus 7.0 để mô phỏng kết quả. Ta sẽ mô phỏng hai lần bằng cách thay đổi chu kỳ hoạt động của task 2.

b) *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 mô phỏng trên Proteus như sau:



Hình 16: Mô phỏng trên Proteus

II. Giao diện hỗ trợ port FreeRTOS lên PIC

1. Ý tưởng, mục đích và nhiệm vụ của giao diện hỗ trợ

2. Trình bày cụ thể về các bước cài đặt và chạy thử

Kết luận

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/

[8] Microchip, 2001, **PIC18FXX2 Data Sheet**,

www.microchip.com/download/lit/pline/picmicro/families/18fxx2/39564b.pdf

[9] Microchip, 2000, **MPLAB-CXX Compiler User's Guide**,

www.microchip.com/download/tools/picmicro/code/mplab17/51217b.pdf

[10] Jean J. Labrosse, **µC/OS-II for the Philips XA**,

www.ucos-ii.com/contents/support/downloads/an1000.pdf

[11] www.en.wikipedia.org/wiki/

[12] www.lynuxworks.com/rtos/rtos.php

Phụ lục

I. Giải thích rõ các file trong FreeRTOS

1. Các ký hiệu viết tắt trong các hàm và biến

Các ký hiệu biến:

- Biến thuộc kiểu char có phần đầu của tên là: c
- Biến thuộc kiểu short có phần đầu của tên là: s
- Biến thuộc kiểu long có phần đầu của tên là: l
- Biến thuộc kiểu float có phần đầu của tên là: f
- Biến thuộc kiểu double có phần đầu của tên là: d
- Biến đếm có phần đầu của tên là: e
- Các kiểu khác (ví dụ như kiểu cấu trúc) có phần đầu của tên là: x
- Biến con trỏ có phần đầu của tên là p, ví dụ con trỏ trỏ tới kiểu short có phần đầu của tên là ps
- Biến không dấu có phần đầu của tên là u, ví dụ biến không dấu kiểu short có phần đầu của tên là us

Các ký hiệu hàm:

- Hàm private có phần đầu của tên là: prv
- Hàm API có phần đầu của tên là kiểu mà nó trả lại, như quy ước của biến.
- Tên hàm được bắt đầu với tên file mà nó được định nghĩa, ví dụ xTaskcreate() được định nghĩa trong file task.

2. Các file chính cần có trong lõi FreeRTOS

a) FreeRTOS.h

Các thông số cần kiểm tra xem đã khai báo trong FreeRTOSconfig.h hay chưa:

- configUSE_PREEMTION: xác định có sử dụng PREEMTION hay không (trong PIC ta có sử dụng PREEMTION nên được định nghĩa bằng 1)
- configUSE_IDLE_HOOK: xác định có sử dụng IDLE HOOK hay không
- configUSE_TICK_HOOK: xác định có sử dụng TICK HOOK hay không
- configUSE_CO_ROUTINE: xác định có sử dụng CO-ROUTINE hay không
- vTaskPrioritySet: đặt mức ưu tiên cho các task
- uxTaskPriorityGet: giữ mức ưu tiên cho các task.
- vTaskDelete: xóa các task (trong PIC không sử dụng hàm này)
- vTaskCleanUpResources: xóa nguồn các task
- vTaskSuspend: treo task, khi bị treo thì task không được xử lý bất kể mức ưu tiên của nó thế nào

- vTaskDelayUntil: tạo trễ đến một thời điểm nào đó
- vTaskDelay: tạo trễ task trong khoảng tick được đưa ra. Thời gian thực hiện còn lại của task phụ thuộc vào nhịp tick.
- configUSE_16_BIT_TICKS: được dùng để xác định có sử dụng 16 bit tick hay không
- configUSE_MUTEXES: được sử dụng để định nghĩa có sử dụng mutexes hay không, khi sử dụng mutex ta cần sử dụng phương pháp mức ưu tiên kế thừa nên ta phải định nghĩa xTaskGetCurrentTaskHandle bằng 1.

b) task.h

Các macro và các định nghĩa

- xTaskHandle kiểu mà các task được tham chiếu. Chẳng hạn, khi gọi xTaskCreate trở lại (qua một tham số con trỏ) một biến xTaskHandle là có thể sau đó sử dụng như một tham số tới vTaskDelete để xóa task.
- tskIDLE_PRIORITY: Định nghĩa mức ưu tiên của idle task, mức ưu tiên này không thể thay đổi.
- taskYIELD(): macro cho việc bắt buộc chuyển ngữ cảnh
- taskENTER_CRITICAL(): macro đánh dấu đoạn mã không thể phân chia (critical). Không thể thực hiện chuyển đổi ngữ cảnh preemptive trong một đoạn critical. Chú ý rằng điều này có thể làm thay đổi ngăn xếp (phụ thuộc vào việc thi hành portable) vì thế phải rất cẩn thận. Tương tự như thế cũng tồn tại macro đánh dấu kết thúc đoạn critical: taskEXIT_CRITICAL().
- taskDISABLE_INTERRUPTS(): macro vô hiệu các ngắt che được. Tương tự cũng có macro cho phép các ngắt: taskENABLE_INTERRUPTS()
- Ngoài ra còn có các định nghĩa để trở lại cho xTaskGetSchedulerState():
  - taskSCHEDULER_NOT_START: 0
  - taskSCHEDULER_RUNNING: 1
  - taskSCHEDULER_SUPPEND: 2

Các task t*ạo API*

- xTaskCreate: Tạo ra task mới và thêm nó vào danh sách các task đã sẵn sàng chạy.
  - pvTaskCode trỏ đến task vào các chức năng. Các task phải thi hành mà không bao giờ được gọi trở lại.
  - pcName biểu thị tên cho task. Điều này chính là để gỡ rối một cách dễ dàng. Độ dài lớn nhất được định nghĩa trong taskMAX_TASK_NAME_LEN, được mặc định là 16.

- usStackDepth: dung lượng của ngăn xếp task đặc trưng cho số biến mà stack có thể giữ được (không phải là số byte). Chẳng hạn, nếu ngăn xếp có độ rộng 16 bit và usStackDepth được định nghĩa thì 100, 200 byte sẽ được phân phối cho dự trữ của ngăn xếp.
- pvParameter sẽ được sử dụng như thông số cho task được tạo ra.
- uxPriority: mức ưu tiên mà tại đó task nên chạy
- pvCreatedTask sử dụng để chuyển lại các điểm có thể lợi dụng được mà nó tham chiếu vào task đã tạo ra.
- pdPASS: nếu task được tạo thành công cà thêm vào danh sách sẵn sàng, trường hợp có lỗi code sẽ được định nghĩa trong file errors.h

- vTaskDelete phải được định nghĩa là 1 nếu muốn sử dụng. Dời các task từ quản lý của lõi thời gian thực. 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 cửa 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(). Bộ nhớ được phân phối bởi code task không tự động được giải phóng, và nên giải phóng trước khi xoá task.
  - pxTask dùng cho xoá task.

Các task điều khiển API

- vTaskDelay: Tạo trễ cho task trong một số nhịp tick cho trước.thời gian thực mà task bị khóa phụ thuộc vào nhịp tick. Hằng số portTICK_RATE_MS có thể được sử dụng để tính thời gian thực từ nhịp tick với thời giancủa 1 tick. .vTaskDelay phải được định nghĩa là 1 nếu muốn sử dụng. xTicksToDelay : Lượng thời gian, trong từng kỳ tick, mà việc gọi task sẽ phải khóa.
- vTaskDelayUntil: phải được định nghĩa là 1 nếu muốn sử dụng. Tạo trễ task cho đến một thời gian được chỉ rõ. Chức năng này có thể được sử dụng bởi những chu kỳ task để bảo đảm một tần số thực hiện không đổi. Chức năng này không cùng một khía cạnh quan trọng với vTaskDelay: vTaskDelay () sẽ khóa task trong khoảng tick nhất định từ khoảng thời gian vTaskDelay được gọi. Bởi vậy khó sử dụng vTaskDelay () bởi chính nó để sinh ra tần số thực hiện cố định vì thời gian giữa lúc task bắt đầu thực hiện đến khi task gọi vTaskDelay() có thể không cố định (task có thể đi qua những đường dẫn khác nhau qua những lệnh gọi, hoặc có những ngắt hoặc đảo ưu tiên hoạt động). Như vậy sẽ mất những khoảng thời gian khác nhau cho mỗi lần thực hiện

  - pxPreviousWakeTime trỏ tới biến giữ thời gian tại đó task được cho thực thi lần cuối. Biến này phải được khởi tạo trước thời điểm mà nó được sử dụng lần đầu. Theo đó biến này sẽ được tự động cập nhật bên trong vTaskDelayUntil ().
  - xTimeIncrement chu kỳ thời gian. Nhiệm vụ sẽ được cho phép thực thi tại thời điểm pxPreviousWakeTime + xTimeIncrement. Gọi vTaskDelayUntil với cùng giá trị tham số xTimeIncrement sẽ làm cho task thực hiện với interface period cố định
- xTaskPriorityGet phải được định nghĩa là 1 nếu muốn sử dụng. pxTask là vị trí của task được hỏi. Chuyển về rỗng nhằm sử dụng những kết quả trong mức ưu tiên của việc goi trở lại task
- vTaskPrioritySet phải được định nghĩa là 1 nếu muốn sử dụng. Hàm này đặt các mức ưu tiên cho các task. Việc chuyển ngữ cảnh sẽ được thực hiện sau khi hàm trở lại nếu mức ưu tiên được đặt cao hơn task đang hoạt động hiện thời
- pxTask sử dụng cho task ở mức ưu tiên được đặt. Chuyển về rỗng sử dụng các kết quả trong mức ưu tiên của task đang gọi đã được đặt
- vTaskSuspend phải được định nghĩa là 1 nếu muốn sử dụng. Hàm này để treo các task. Khi bị treo các task này sẽ không được sử dụng thời gian của bộ vi điều khiển bất kể mức ưu tiên nào. Những lời gọi đến vTaskSuspend không được gọi chồng, ví dụ gọi vTaskSuspend () hai lần trong cùng một task thì vẫn là 1 lời gọi đến vTaskResume () để treo task. pxTaskToSuspend sử dụng trong task bị treo. Chuyển về rỗng sau khi gọi đến treo task.
- vTaskResume: phải được định nghĩa là 1 nếu muốn sử dụng. Hàm sử dụng để tiếp tục các task bị treo. Các task bị treo bởi 1 hay nhiều lời gọi đến vTaskSuspend () sẽ được trả lại sẵn sang để chạy chỉ bởi 1 lời gọi vTaskResume (). pxTaskToResume sử dụng cho task được sẵn sàng.
- xTaskResumeFromISR được định nghĩa là 1 nếu muốn sử dụng. Việc chạy hàm vTaskResume() có thể được gọi trong ISR. Các task bị treo bởi 1 hay nhiều lời gọi đến vTaskSuspend() sẽ được trả lại sẵn sang để chạy chỉ bởi 1 lời gọi vTaskResumeFromISR(). pxTaskToResume sử dụng cho task được sẵn sàng.
- vTaskStartScheduler(void) bắt đầu tick xử lý của lõi thời gian thực. Trước khi việc gọi kernel được làm chủ qua đó các task được thực hiện. Chức năng này không gọi trở lại đến khi có lời gọi vTaskEndScheduler (). Ít nhất là 1

task được tạo ra qua lời gọi xTaskCreate () sau lời gọi vTaskStartScheduler (). Idle task được tự động tạo ra khi task ứng dụng đầu tiên được tạo ra.

- vTaskEndScheduler(void) dừng tick của kernel thời gian thực. Tất cả các task được tạo ra tự động bị xóa và đa nhiệm (hoặc preemptive hoặc cooperative) sẽ dừng lại. Giả sử vTaskStartScheduler được gọi lại thì sẽ thực hiện phục hồi từ điểm mà vTaskStartScheduler() được gọi. vTaskEndScheduler yêu cầu chức năng thoát được định nghĩa trong lớp porable (xem vPortEndScheduler() trong port.c cho PC port). Điều này được xây dựng cho phần cứng đặc trưng riêng biệt như dừng kernel tick. vTaskEndScheduler() giải phóng toàn bộ tài nguyên được cấp phát bởi kernel nhưng sẽ không giải phóng tài nguyên cấp phát bởi task ứng dụng.
- vTaskSuspendAll() treo tất cả kernel thời gian thực hoạt động trong khi vẫn giữ ngắt (bao gồm cả kernel tick) hoạt động. Sau khi gọi vTaskSuspendAll() các task đang được gọi vẫn tiếp tục thực hiện ngoại trừ nguy cơ hoán đổi ra ngoài cho đến khi lời gọi xTaskResumeAll() được thực hiện.
- xTaskResumeAll(void) phục hồi kernel hoạt động thời gian thực theo lời gọi vTaskSuspendAll(). Sau lời gọi vTaskSuspendAll(), kernel sẽ đặt điều khiển vào các task đang thực hiện vào bất kỳ thời gian nào. Nếu phục hồi lập lịch gây ra chuyển ngữ cảnh thì pdTRUE được trả lại, trường hợp khác pdFALSE được trả lại.

Các task ti*ện ích*

- xTaskGetTicksCount: trả lại giá trị ticks đếm được từ khi vTaskStartScheduler bị hủy bỏ.
- uxTaskGetNumberOfTasks(void): hàm này sẽ 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 bộ đếm
- vTaskList(): configUSE_TRACE_FACILITY, vTaskDelete và vTaskSuspend phải được định nghĩa là 1 nếu muốn sử dụng. Chú ý: 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. Hàm này cũng liệt kê các task hiện thời, cùng với trạng thái hiện thời và ngăn xếp sử dụng của chúng đánh dấu ở mức water cao. Tasks được ghi nhận là blocked ('B'), ready ('R'), deleted ('D') hoặc suspended ('S').

  - pcWriteBuffer: một bộ đệm được đề cập chi tiết ở trên được viết bằng mã ASCII. Bộ đệm này được coi là đủ lớn để chứa các báo các phát sinh. Khoảng 40 bytes cho 1 task là đủ
- 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
  - pcBuffer: đệm mà vết được ghi.
- ulBufferSize dung lượng của pcBuffer tính bằng byte. Việc đánh dấu sẽ được tiếp tục đến khi bộ đệm đầy hoặc ulTaskEndTrace () được gọi.
- ulTaskEndTrace(void): Dừng đánh dấu kernel hoạt động, trả lại số byte mà đã viết vào bộ đệm đánh dấu.

L*ập lịch nội bộ cho mục đích port*

- vTaskIncrementTick(void): Hàm này không sửdụng từ code ứng dụng. Nó được tạo ra khi thực thi việc port của lập lịch và nó là giao diện dùng cho sửdụng loại trừ của bộ lập lịch. Gọi từ kernel tick (hoặc preemptive hoặc cooperative), nó tăng bộ đếm tick và kiểm tra nếu bất kỳ task nào bị khóa đến một điểm hữu hạn sẽ được loại bỏ khỏi danh sách bị khóa và thay vào đó là danh sách sẵn sàng.
- vTaskPlaceOnEventList(): Hàm này không sửdụng từ code ứng dụng. Nó được tạo ra khi thực thi việc port của lập lịch và nó là giao diện dùng cho sửdụng loại trừ của bộ lập lịch. Hàm này được gọi khi không cho phép ngắt. Loại bỏ tất cả các task đang gọi từ danh sách sẵn sàng và thay vao đó cả danh sách task chờ sự kiện liên quan và danh sách của task trễ. Task sẽ bị loại bỏ từ các list và thay thế ở list sẵn sàng hoặc sự kiện xuất hiện (có thể không phải là task có mức ưu tiên cao hơn trong cùng một sự kiện) hoặc kỳ trễ hết hạn.
- pxEventList: gồm các task bị khóa chờ đến khi có sự kiện xảy ra.
- xTicksToWait: Lượng thời gian lớn nhất mà task phải chờ đến khi sự kiện xuất hiện. Điều này rõ ràng trong kernel ticks, hằng số portTICK_RATE_MS có thể được sử dụng ssể chuyển kernel ticks sang thời gian thực tế.
- xTaskRemoveFromEventList(): Hàm này không sửdụng từ code ứng dụng. Nó được tạo ra khi thực thi việc port của lập lịch và nó là giao diện dùng

cho sửdụng loại trừ của bộ lập lịch. 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. xTaskRemoveFromEventList() sẽ được gọi nếu hoặc sự kiệnxuất hiện giải phóng task hoặc thời gian block bị timeout. Hàm này trả lại pdTRUE nếu task bị loại bỏ có mức ưu tiên cao hơn task tạo lời gọi, ngược lại pdFALSE.

- vTaskCleanUpResources(): Hàm này không sửdụng từ code ứng dụng. Nó được tạo ra khi thực thi việc port của lập lịch và nó là giao diện dùng cho sửdụng loại trừ của bộ lập lịch. Hàm này phải được định nghĩa là 1 nếu muốn sử dụng. Xóa hết hàng đợi sẵn sàng và trễ của khối điều khiển task, giải phóng bộ nhớ cấp phát cho khối điều khiển ták và các ngăn xếp task.
- xTaskGetCurrentTaskHandle(): trả lại kênh điều khiển cho các task đang gọi.
- vTaskSetTimeOutState(): giữ lại những trạng thái hiện thời để tham chiếu sau này.
- xTaskCheckForTimeOut(): so sánh trạng thái thời gian hiện tại với thời điểm trước để xem có bị timeout hay không.
- vTaskMissedYield(): shortcut được sử dụng bởi hàng đợi thực thi 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.

c) 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. Các chức năng của file như tạo mới, thêm, bớt các tác vụ vào danh sách các task đang chạy (running), sẵn sàng (ready), khoá (block), treo (suppend).

- listSET_LIST_ITEM_OWNER(): macro để đặt đối tượng sở hữu các phần tử của danh sách. Các đối tượng sở hữu phần tử danh sách là đối tượng (thường là TCB) nằm trong phần tử danh sách
- listSET_LIST_ITEM_VALUE(): macro để đặt giá trị của phần tử danh sách. Trong hầu hết trường hợp giá trị đó được dùng để sắp xếp danh sách theo một thứ tự nào đó.

- listGET_LIST_ITEM_VALUE(): macro để lấy lại 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.
- listLIST_IS_EMPTY: macro để xác định xem danh sách còn chứa phần tử nào không. Macro chỉ có giá trị true nếu danh sách rỗng.
- listCURRENT_LIST_LENGTH(): trả lại số phần tử trong danh sách.
- listGET_OWNER_OF_NEXT_ENTRY(): hàm này trả lại phần tử tiếp theo cảu danh sách. Tham số pxIndex được sử dụng để duyệt dọc theo danh sách. Khi gọi hàm này thì pxIndex tăng lên đến phần tử tiếp theo của danh sách và trả lại ở thông số pxOwner. Vì thế sử dụng nhiều lời gọi hàm này để loại mọi phần tử ra khỏi danh sách. Thông số pxOwner của phần tử danh sách trỏ vào đối tượng sở hữu phần tử danh sách. Trong bộ lập lịch thì đó thường là khối điều khiển tác vụ (TCB).
- listGET_OWNER_OF_HEAD_ENTRY(): sử dụng để tìm chương trình chủ của phần tử đầu tiên trong danh sách. Các danh sách thường được sắp xếp theo gái trị phần tử tăng dần. Giá trị cũng được trả lại vào pxOwner. Thông số pxOwner có thể được tạo ra theo hai cách liên kết giữa phần tử danh sách và chương trình sở hữu nó.
- listIS_CONTAINED_WITHIN(): kiểm tra xem phần tử có nằm trong danh sách không. Phần tử danh sách duy trì con trỏ "container" mà trỏ vào danh sách mà nó nằm trong đó. Công việc mà macro này là kiểm tra xem "container" và danh sách có khớp nhau không.
- vListInitialise(): phải gọi ngay trước khi danh sách được sử dụng. Hàm này khởi tạo tất cả các phần tử trong cấu trúc danh sách và thêm phần tử xListEnd vào danh sách để ghi lại dùng cho việc khôi phục lại danh sách.
- vListInitialiseItem(): phải gọi ngay trước khi phần tử danh sách được sử dụng. Hàm này khởi tạo toàn bộ danh sách trống vì thế phần tử không nằm trong danh sách.
- vListInsert(): thêm phần tử vào danh sách. Phần tử sẽ được thêm vào danh sách ở vị trí được xác định rõ bởi giá trị của phần tử (sắp xếp theo thứ tự tăng dần).
- vListInsertEnd(): thêm phần tử vào danh sách. Phần tử sẽ được thêm vào vị trí chẳng hạn như vị trí phần tử cuối cùng trong phạm vi danh sách trở lại từ nhiều lời gọi hàm listGET_OWNER_OF_NEXT_ENTRY. Phần tử danh sách pvIndex dùng để duyệt danh sách, thông số này sẽ tăng đến phần tử tiếp theo

của danh sách. Thay thế phần tử danh sách sử dụng vListInsertEnd hiệu quả để thay vào vị trí trỏ bởi pvIndex. Có nghĩa là mọi phần tử khác trong danh sách sẽ được trả lại bởi hàm listGET_OWNER_OF_NEXT_ENTRY trước khi thông số pvIndex trỏ trở lại phần tử đang được thêm vào.

- vListRemove(): loại bỏ phần tử từ danh sách. Phần tử danh sách có con trỏ đến danh sách chứa nó, vì thế chỉ phần tử danh sách cần được chuyển thành hàm. Phần tử sẽ loại bỏ chính nó từ con trỏ danh sách bởi thông số pxContainer của nó.

d) croutine.h

Nhiệm cụ của file là:

- xCoRoutineHandle: Sử dụng để ẩn những thực thi của khối điều khiển co-routine. Cấu trúc khối điều khiển dù sao cũng phải nằm trong header quy định trong các macro thực thi của các chức năng của co-routine.
- xCoRoutineCreate: tạo mới các co-routine và thêm chúng vào danh sách các co-routine đã sẵn sàng. Hàm này trả lại pdPASS nếu co-routine tạo ra vè thêm vào thành công list co-routine sẵn sàng chạy. Các biến phải được cho là static nếu nuốn duy trì qua những lời gọi block. Các thông số là:
  - pxCoRoutineCode: Trỏ vào chức năng của co-routine. Các chức năng co-routine yêu cầu những cú pháp đặc biệt
  - uxPriority: Mức ưu tiên đối với co-routine khác tại đó co-routine sẽ chạy.
  - uxIndex: được sử dụng để phân biệt các co-routine chạy trong cùng một chức năng.
- vCoRoutineSchedule(): 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(). Các macro được dự phòng cho sử dụng bên trong chỉ bởi các thực thi co-routine như crEND, crSET_STATE. Các macro này có không được sử dụng bởi người viết ứng dụng

- crDELAY: Tạo trễ cho các co-routine trong khoảng thời gian cố định. crDELAY chỉ có thể gọi từ các hàm co-routine chứ không thể gọi nó từ một hàm của co-routine vì crDELAY không thể duy trì stack riêng của nó.
  - xHandle: thông số để điều kiển trễ co-routine.
  - xTickToDelay: số lượng tick mà co-routine tạo trễ. Lượng thời gian thực tế bởi configTICK_RATE_HZ (đặt trong freeRTOSconfig.h). Hằng số portTICK_RATE_MS có thể được sử dụng để đổi từ tick ra thời gian thực.
- Các macro crQUEUE_SEND(), crQUEUE_RECEIVE() là các co-routine tương đương với các hàm xQueueSend() và xQueueReceive(). crQUEUE_SEND(), crQUEUE_RECEIVE() chỉ có thể sử dụng trong các co-routine ngược lại các hàm xQueueSend() và xQueueReceive() chỉ được sử dụng trong các task. CrQUEUE_SEND() chỉ có thể gọi từ các hàm co-routine chứ không thể gọi nó từ một hàm của co-routine vì CrQUEUE_SEND() không thể duy trì stack riêng của nó.
  - pxQueue: là một kênh điều khiển của hàng đợi, tại đó dữ liệu sẽ được đưa lên. Kênh điều khiển này sử dụng như việc trả lại giá trị khi mà hàng đợi được tạo ra bằng cach sử dụng hàm xQueueCreate() của API.
  - pvItemToQueue: trỏ vào dữ liệu sẽ được đưa vào queue. Số lượng byte của mỗi hàng đợi là riêng biệt. Số lương byte này được sao chép từ pvItemToQueue vào chính queue đó.
  - xTickToDelay: số lương tick mà co-routine bị khoá để chờ tồn tại trong queue, thường là khoảng này không tức thời.
  - pxResult: biến được trỏ bởi pxResult sẽ được đặt đến pdPASS nếu dữ liệu được đưa hành công vào hàng đợi, ngược lại nó sẽ báo lỗi được định nghĩa trong projdef.h.
- Macro crQUEUE_SEND_FROM_ISR() và crQUEUE_RECEIVE_ FROM_ISR() là co-routine tương đương với xQueueSendFromISR() và xQueueReceiveFromISR() được sử dụng bởi task. crQUEUE_SEND_FROM_ISR() và crQUEUE_RECEIVE_FROM_ISR chỉ có thể sử dụng để chuyển dữ liệu giữa co-routine và ISR, ngược lại xQueueSendFromISR() và xQueueReceiveFromISR() chỉ được sử dụng để chuyển dữ liệu giữa task và ISR. crQUEUE_SEND_FROM_ISR chỉ có thể gọi từ ISR để gửi dữ liệu đến queue trong phạm vi co-routine.
  - pxQueue : kênh điều khiển hàng đợi tại đó các mục được đưa lên.

- pvItemToQueue trỏ đến mục sẽ được đặt vào hàng đợi. Dung lượng của mục mà hàng đợi sẽ giữ được định nghĩa khi hàng đợi được tạo ra, do đó nhiều byte sẽ được sao chép từ pvItemToQueue vào vùng dự trữ của hàng đợi.
- xCoRoutinePreviouslyWoken: cho ISR có thể đưa dữ liệu vào cùng một hàng đợi trong nhiều thời điểm từ các ngắt đơn. Lời gọi đầu luôn chuyển vào pdFALSE. Các lời gọi sau chuyển vào giá trị từ lời gọi trước
- Trả lại giá trị pdTRUE nếu co-routine được đánh thức bởi đưa dữ liệu vào hàng đợi. Điều này được sử dụng bởi ISR để tiên liệu nếu chuyển ngữ cảnh được yêu cầu theo các ngắt.

- vCoRoutineAddToDelayedList: Hàm này được tạo ra cho những sử dụng nội tại chỉ bởi các co-routine macro. Các macro nguyên thủy của thực thi co-routie đòi hỏi có những nguyên mẫu ở đây. Hàm này không được sử dụng bởi người viết ứng dụng. Hàm này loại bỏ co-routine hiện thời từ list sẵn sàng và đặt chúng vào list trễ thích hợp.

e) 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 (Ph*ần II.III.2*):

- *pxPortInitialiseStack()
- xPortStartScheduler()
- vPortEndScheduler()

Bên cạnh đó file làm một số nhiệm vụ quan trọng cho việc 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

- *prvPortMalloc()
- vPortFree()
- vPortInitialiseBlocks()

3. 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:

- semphr.h:

a) projdef.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 file dùng sẵn do người viết luôn tuân thủ chuẩn này:

- pdTRUE = 1
- pdFALSE = 0
- pdPASS = 1
- pdFAIL = 0
- errQUEUE_EMPTY = 0
- errQUEUE_FULL = 0

Ngoài ra định nghĩa các lỗi:

- errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY = -1
- errNO_TASK_TO_RUN = -2
- errQUEUE_BLOCKED = -4
- errQUEUE_YIELD = -5

b) 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.

c) 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().

4. Các file cần để port FreeRTOS lên vi điều khiển

a) 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 , ngược lại là 0. Để xem thêm các thông số cần khai báo cho phần này xem FreeRTOS.h <PhầnII.I.a>
- 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.

b) port.c

Đây là file quan trọng nhất trong việc tạo ra các hàm định nghĩa trong portable.h cho việc port lên PIC. 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. Macronayf 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.

c) 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. Ví dụ như đối với PIC18F452 cần khai báo vị trí thanh ghi ngắt toàn cục là 0x80, bit cho phép ngắt toàn cục hay không là INTCONbits.GIEH. Tiếp đó là lệnh delay của vi điều khiển portNOP().
- 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 ly 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.

II. Giải thích rõ về giao diện

Các phần bổ sung

Rtos ppts.ppt: File này trình bày về uCOS

Rtproj.pdf: so sánh Free với RTLinux

36-danlu911…pdf: so sánh Free và eCOS

Freertos.pdf: các câu hỏi về freeRTOS

RTOS.doc: xem lại kỹ file này có phần so sánh

Việc còn lại

Hoàn thành phần so sánh

Viết lời cảm ơn, lời mở đầu, kết luận, hướng phát triển

Chỉnh lại phần phụ lục.

Xem lại các câu chữ