[**Chương I. Tổng quan về hệ điều hành thời gian thực** 2](#_Toc28735833)

[**Câu 1: Thế nào là thời gian thực? hãy mô tả ngắn gọn các dạng thời gian thực?** 2](#_Toc28735834)

[**Câu 2: Trình bày các chức năng chính của một hệ điều hành thời gian thực.** 2](#_Toc28735835)

[**Câu 3: So sánh hệ điều hành thời gian thực với hệ điều hành trên PC.** 2](#_Toc28735836)

[**Câu 4: Phân loại hệ điều hành thời gian thực. Mô tả ngắn gọn đặc điểm của từng loại** 3](#_Toc28735837)

[**Câu 5: Trình bày lịch sử phát triển và mô tả đặc điểm cơ bản của hệ điều hành thời gian thực** 3](#_Toc28735838)

[**Câu 6: hệ điều hành FreeRTOS có đặc điểm gì?** 3](#_Toc28735839)

[**Câu 7: Trình bày các thành phần cơ bản của hệ điều hành thời gian thực** 4](#_Toc28735840)

[**Chương II. FreeRTOS quản lý tác vụ và hàng đợi** 5](#_Toc28735841)

[**Câu 1: Khái niệm TCB (Task Control Block)?** 5](#_Toc28735842)

[**Câu 2: Quản lý tác vụ** 5](#_Toc28735843)

[**Câu 3: Các trạng thái của tác vụ** 5](#_Toc28735844)

[**Câu 4: Các hàm API** 6](#_Toc28735845)

[**Chương III. FreeRTOS: quản lý ngắt** 8](#_Toc28735846)

[**Câu 1: Quản lý ngắt** 8](#_Toc28735847)

[**Câu 2: Ngắt lồng nhau** 8](#_Toc28735848)

[**Câu 3: Các hàm API** 9](#_Toc28735849)

[**Chương IV. Quản lý tài nguyên và bộ nhớ** 12](#_Toc28735850)

[**Câu 1: Các hàm API cấp phát bộ nhớ,giải phóng bộ nhớ** 12](#_Toc28735851)

[**Câu 2: Mô tả, so sánh các thư viện heap\_1.c,heap\_2.c, heap\_3.c, heap\_4.c, heap\_5.c** 13](#_Toc28735852)

[**Câu 3: Khái niệm Semaphone, Mutex và mục đích sử dụng** 15](#_Toc28735853)

[**Câu 4: Phân biệt bộ nhớ heap và stack** 16](#_Toc28735854)

[**Chương V. xử lý sự cố** 17](#_Toc28735855)

[**Câu 1: Trình bày một số lỗi thông dụng của FreeRTOS** 17](#_Toc28735856)

[**Chương VI. Cách sử dụng one-shot timer và auto- reaload timer** 20](#_Toc28735857)

[**Câu 1: Khởi tạo one-shot and auto-reload timers bằng hàm xCreateTimer();** 20](#_Toc28735858)

# **Chương I. Tổng quan về hệ điều hành thời gian thực**

## **Câu 1: Thế nào là thời gian thực? hãy mô tả ngắn gọn các dạng thời gian thực?**

***Khái niệm:*** Thời gian thực thể hiện ở chỗ: một hệ thống phải có những phản ứng thích hợp, đúng thời điểm với môi trường của nó. Thời gian thực có nghĩa là “ đủ nhanh” trong một ngữ cảnh, một môi trường mà hệ thống đang hoạt động.

***Khái niệm hệ điều hành thời gian thực:*** Real Time Operating Systems là phần mềm điều khiển chuyên dụng thường được dùng trong những ứng dụng điện toán nhúng có tài nguyên bộ nhớ hạn chế và yêu cầu ngặt nghèo về thời gian đáp ứng tức thời, tính sẵn sàng cao và khả năng tự kiểm soát một cách chính xác.

***Các dạng thời gian thực:***

* Thời hạn lập lịch (scheduling deadline): thời gian thực đòi hỏi một giới hạn cao hơn về thời gian trễ.
* Thời gian thực cứng (hard real-time): hệ thống phải tiếp nhận và nắm bắt được thời gian lập lịch của nó tại mỗi và mọi thời điểm. Sự sai sót trong việc tiếp nhận thời hạn có thể dẫn đến hậu quả nghiêm trọng thậm chí chết người.
* Thời gian thực mềm” (soft real- time): thời gian lập lịch có thể dễ thở hơn một chút. Hệ thống phản ứng lại các sự kiện trong thời gian cho phép nhưng không có gì thực sự nghiêm trọng xảy ra nếu hệ thống thỉnh thoảng bị trễ. Lỗi về mặt thời gian có thể chỉ đơn giản là dẫn đến hậu quả giảm độ tin cậy của đối tượng đồi với hệ thống mà không có hậu quả thê thảm nào khác xảy ra.

**Đặc điểm:**

* Đơn giản
* Phản hồi được đảm bảo
* Ràng buộc thời gian nghiêm ngặt
* Giới hạn băng thông lớn nhất và nhỏ nhất
* Các thành phần xác định
* Hành vi có thể đoán trước
* Có thể nâng cấp

## **Câu 2: Trình bày các thành phần cơ bản của hệ điều hành thời gian thực**

* Bộ lập lịch (Scheduler): Đây là thành phần RTOS chính, xác định thứ tự thực hiện các tác vụ hoặc luồng thường dựa trên sơ đồ ưu tiên và theo kiểu Round-Robin hoặc completion
* Đa xử lý đối xứng (SMP – Symmetric Multiprocessing): RTOS có khả năng xử lý và phân tách nhiều tác vụ hoặc luồng để chúng có thể chạy trên nhiều lõi để cho phép xử lý mã song song (nghĩa là đa nhiệm).
* Thư viện chức năng (Function library): Là một giao diện tiêu chuẩn có thể chứa giao diện chương trình ứng dụng (API) để gọi các tường trình bên trong nó, đây là giao diện kết nối mã ứng dụng đó và kernel. Mã ứng dụng thực thể yêu cầu trực tiếp đến kernel thông qua thư viện hàm để nhắc ứng dụng đưa ra hành vi lập trình mong muốn.
* Thời gian chuyển đổi ngữ cảnh/độ trễ điều chỉnh nhanh (Fast dispatch latency/context switch time: Chuyển đổi ngữ cảnh là thời gian cần thiết để chuyển đổi từ một luồng đang chạy sang một luồng khác, điều này liên quan đến việc lưu bối cảnh của tác vụ hiện tại và thay thế nó bằng bối cảnh của luồng mới để chạy. Trong RTOS thời gian chuyển đổi nên duy trì tính quyết định và tối thiểu.
* Các đối tượng và lớp dữ liệu do người dùng định nghĩa (User-defined data objects and classes): RTOS dựa trên các ngôn ngữ lập trình với các cấu trúc dữ liệu được tổ chức dựa trên loại hoạt động của chúng. Người dùng định nghĩa các bộ đối tượng thông qua một ngôn ngữ lập trình được chỉ định như C ++ mà RTOS sẽ sử dụng để kiểm soát ứng dụng đã chỉ định.
* Quản lý bộ nhớ (Memory Management): Cần quản lý bộ nhớ để phân bổ bộ nhớ cho mọi chương trình được chạy hoặc đối tượng được tham chiếu trong bộ nhớ.

## **Câu 3: Trình bày các chức năng chính của một hệ điều hành thời gian thực.**

* RTOS phải quản lý các tài nguyên như bộ nhớ, bộ xử lý và I/O
* Lập lịch và xử lý ưu tiên
* Đảm bảo các ràng buộc thời gian nghiêm ngặt và cung cấp hoạt động đáng tin cậy
* Tính mềm dẻo để phù hợp với các yêu cầu ứng dụng khác nhau
* Hỗ trợ đồng bộ hóa và giao tiếp giữa các tác vụ
* Khi các hệ thống phức tạp được xây dựng, RTOS quản lý việc tích hợp các thành phần

## **Câu 4: So sánh hệ điều hành thời gian thực với hệ điều hành trên PC**

Hệ điều hành thời gian thực (RTOS) cho các hệ thống nhúng rất khác so với hệ điều hành khác trên máy tính desktop như Windows hay UNIX:

* Thứ nhất: trên máy tính desktop hệ điều hành sẽ chiếm quyền điều khiển ngay sau khi máy được bật và sau đó mới cho ta bắt đầu ứng dụng của mình.Còn trong hệ thống nhúng, ứng dụng thường xuyên liên kết với RTOS, tại lúc khởi động, ứng dụng nhận lấy quyền điều khiển đầu tiên sau đó mới bắt đâu RTOS.
* Thứ hai, rất nhiều RTOS không bảo vệ chính bản thân chúng một cách cẩn thận như hệ điều hành trên desktop.
* Thứ ba, để tiết kiệm bộ nhớ các RTOS mặc định chỉ bao gồm các dịch vụ cần thiết cho ứng dụng nhúng của ta
* Thứ tư, hầu hết hệ điều hành trên máy desktop quan tâm đến hiệu năng trung bình trong khi các hệ điều hành thời gian thực quan tâm đến các giới hạn, thời gian đáp ứng

## **Câu 4: Phân loại hệ điều hành thời gian thực. Mô tả ngắn gọn đặc điểm của từng loại**

* Hệ điều hành thời gian thực thương mại: HĐH nhỏ với mục đích thương mại là các hệ điều hành nhỏ và nhanh như: QNX, PDOS, Psos, VxWorks, ERCOS, WinCE…
* Hệ điều hành thời gian thực mã nguồn mở: RT-UNIX, RT-LINUX, RT-MACH, RT-POSIX chúng chậm hơn và có khả năng dự đoán ít hơn so với các hệ điều hành thời gian thực thương mại nhưng chúng có nhiều chức năng và môi trường phát triển tốt hơn dựa trên tập các giao tiếp chuẩn và thân thiện
* Các nhân cho mục đích nghiên cứu: hệ điều hành có các đặc điểm:

+ Hỗ trợ các thuật toán lập lịch thời gian thực và việc phân tích thời gian

+ Hỗ trợ các dịch vụ cơ bản để đồng bộ thời gian thực

+ Nhấn mạnh khả năng dự đoán hơn là hiệu năng trung bình

+ Hỗ trợ cho khả năng chịu lỗi

Ví dụ: Sping, MARS,CHAOS,DARK…

## **Câu 5: Trình bày lịch sử phát triển và mô tả đặc điểm cơ bản của hệ điều hành thời gian thực**

* Tài nguyên bộ nhớ hạn chế
* Yêu cầu ngặt nghèo về thời gian đáp ứng tức thời
* Tính sẵn sàng cao và khả năng tự kiểm soát một cách chính xác
* RTOS đóng vai trò rất quan trọng trong phát triển hệ thống nhúng thời gian thực

## **Câu 6: Hệ điều hành FreeRTOS có đặc điểm gì?**

FreeRTOS là một hệ điều hành thời gian thực (RTOS) được thiết kế đủ nhỏ để chạy trên một vi xử lý

* FreeRTOS chỉ cung cấp chức năng lập lịch thời gian thực lõi, truyền thông liên tác vụ, định thời và đồng bộ
* FreeRTOS được thiết kế phù hợp cho nhiều hệ thống nhúng nhỏ gọn vì nó chỉ triển khai rất ít các chức năng
* FreeRTOS là một hệ điều hành nhúng thời gian thực mã nguồn mở
* Là một hệ điều hành nhúng rất phù hợp cho nghiên cứu, học tập về các kỹ thuật, công nghệ cũng như việc phát triển mở rộng tiếp các thành phần cho hệ điều hành

# **Chương II. FreeRTOS quản lý tác vụ và hàng đợi**

## **Câu 1: Khái niệm TCB (Task Control Block)?**

FreeRTOS kernel quản lý các task qua TCB. Một TCB tồn tại cho từng task trong FreeRTOS và bao gồm toàn bộ các thông tin về trạng thái của một task

TCB của RTOS quản lí các thông tin về trạng thái làm việc của Task tương ứng, giá trị tạm thời các thanh ghi mà nó sử dụng khi bị task khác block. Ngoài ra nó còn chứa một số thông tin khác như các event gắn với task ( với OS hỗ trợ semaphore chẳng hạn) hoặc time wait của task( với OS hỗ trợ các hàm delay,supend).

## **Câu 2: Quản lý tác vụ**

## **Câu 3: Các trạng thái của tác vụ**

* Một ứng dụng có thể bao gồm nhiều tác vụ, một tác vụ có thể tồn tại trong một hoặc hai trạng thái là “Running” và “Not Running”.
* Khi một tác vụ đang ở trạng thái “Running”, bộ xử lý đang thực sự thực thi mã lệnh của nó
* Khi một tác vụ đang ở trạng thái “Not Running”, tác vụ đang không hoạt động, trạng thái của nó đã được lưu sẵn sàng để tiếp tục thực hiện lần tiếp theo khi trình lập lịch quyết định nó nên chuyển sang trạng thái “Running”.
* Một tác vụ được chuyển từ tt “Not Running” sang “Running” được gọi là “chuyển vào” hoặc “đổi vào”, ngược lại là chuyển ra hoặc đổi ra.

1. **Trạng thái “Block”:**

* Một tác vụ đang chờ đợi một sự kiện được gọi là đang ở trạng thái “Block”, đây là một trạng thái con của trạng thái “Not Running”
* Các tác vụ có thể nhập vào trạng thái “Block” để đợi hai loại sự kiện khác nhau đó là sự kiện về thời gian và các sự kiện đồng bộ hóa

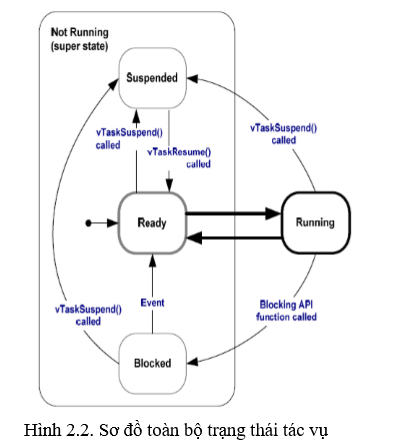
1. **Trạng thái “Suspend” ( đình chỉ)**

* Là trạng thái con của “Not Running”. Các tác vụ trong trạng thái này không sẵn sàng cho trình lập lịch
* Cách chuyển vào trạng thái “suspend” là thông qua lời gọi đến hàm API vTaskSuspend(), các duy nhất để chuyển ra thông qua lời gọi hàm API vTaskResume() hoặc hàm xTaskResumeFromISR()

1. **Trạng thái “Ready” (sẵn sàng)**

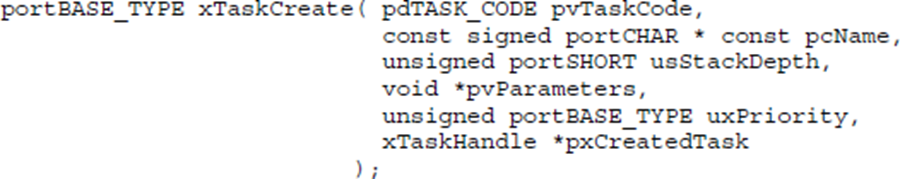
* Là tt khi Các tác vụ đang ở tt “Not Running” nhưng không phải trạng thái “Block” hoặc “Suspend”
* Chúng có thể chạy hoặc sẵn sàng để chạy nhưng không phải ở tt “Running”

Sơ đồ chuyển trạng thái như sau:



## **Câu 4: Các hàm API**

1. **Khởi tạo tác vụ:**

****

**+**pvTaskCode: con trỏ trỏ đến hàm thực thi tác vụ

**+** pcName: Tên mô ta tác vụ, tên này không được sử dụng bởi FreeRTOS dưới bất kỳ hình thức nào, là sự hỗ trỡ cho việc debug

**+** usStrackDepth: kích thước của ngăn xếp (Stack), xác định số lượng từ (word) ngăn xếp có thể giữ, không phải là số byte

**+** pvParameters: Các hàm tác vụ chấp nhận một tham số của con trỏ kiểu void (void \*). Giá trị gán cho pvParameters sẽ là giá trị được đưa vào tác vụ

**+** uxPriority: Mức ưu tiên của tác vụ sẽ thực thi

**+** pxCreatedTask: Có thể được sử dụng để vượt qua một xử lý (handle) của tác vụ đang được khởi tạo. Xử lý này từ đó có thể được sử dụng để tham chiếu đến tác vụ trong lời gọi API, ví dụ, thay đổi mức ưu tiên của tác vụ hoặc xóa tác vụ

**+** Giá trị trả về:

pdTRUE: Tác vụ đã được tạo thành công

errCOULD\_NOT\_ALLOCATE\_REQUIRED\_MEMORY: Tác vụ không thể được tạo vì không có đủ bộ nhớ heap khả dụng cho FreeRTOS để cấp phát đủ RAM nhằm giữ cấu trúc dữ liệu tác vụ và ngăn xếp

1. **Xóa tác vụ**

**vTaskDelete(xTaskHandle pxTaskToDelete)**

pxTaskToDelete: Trình điều khiển của tác vụ bị xóa (tác vụ chủ thể) - xem tham số pxCreatedTask của hàm API xTaskCreate () để biết thông tin về cách xử lý các tác vụ. Một tác vụ có thể xóa chính nó bằng cách cách chuyển NULL vào vị trí của một tác vụ xử lý hợp lệ.

1. **Thay đổi mức độ ưu tiên tác vụ**

**void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE\_TYPE uxNewPriority );**

+ pxTask: Xử lý của tác vụ có quyền ưu tiên đang được sửa đổi

+ uxNewPriority: Mức độ ưu tiên mới

1. **Truy vấn mức độ ưu tiên của một tác vụ:**

uxTaskPriorityGet(pxTask): Tên của tác vụ muốn truy vấn

**Câu 5: QCB (Queue Control Block) là gì? Đặc điểm của hàng đợi, các hàm API.**

5.1. QCB (Queue Control Block)

5.2. Đặc điểm của hàng đợi

**Câu 6: Cách thức đồng bộ tác vụ và truyền thông liên tác vụ**

Signal

EventFlagGroup

Semaphores

# **Chương III. FreeRTOS: Quản lý ngắt**

## **Câu 1: Quản lý ngắt**

**Semaphore:** Một semaphore nhị phân có thể được sử dụng để mở khóa một tác vụ mỗi lần khi có ngắt cụ thể xảy ra, đồng bộ hóa hiệu quả tác vụ với ngắt. Điều này cho phép phần lớn xử lý sự kiện ngắt được thực hiện trong tác vụ được đồng bộ hóa, chỉ với một phần rất nhanh và ngắn còn lại trực tiếp trong ISR. Xử lý ngắt tàm ngưng tác vụ đang xử lý.

**Cờ hiệu:** Trình tự thực thi như sau:

* Một ngắt xảy ra
* Chương trình phục vụ ngắt thực hiện, đưa ra cờ hiệu mở khóa trình xử lý tác vụ.
* Trình xử lý tác vụ thực thi khi ngắt hoàn thành. Điều đầu tiên trình xử lý tác vụ thực hiện là lấy cờ hiệu.
* Trình xử lý tác vụ thực hiện xử lý sự kiện trước khi cố gắng lấy lại cờ hiệu – nhập trạng thái Block nếu cờ hiệu không khả dụng ngay tức thời.

**Hàng đợi:** Các hàng đợi được dùng để giao tiếp sự kiện và truyền dữ liệu.

## **Câu 2: Ngắt lồng nhau**

Ngắt lồng nhau là khi một chương trình ngắt gọi một chương trình ngắt khác

Những cổng này yêu cầu một hoặc cả hai hằng số sau:

|  |  |
| --- | --- |
| **Hằng số** | **Mô tả** |
| configKERNEL\_INTERRUPT\_PRIORITY | Thiết lập mức ưu tiên ngắt sử dụng bởi ngắt đánh dấu.  Nếu cổng không dùng hằng configMAX\_SYSCALL\_INTERRUPT\_PRIORITY thì bất kì ngắt nào sử dụng hàm API FreeRTOS an toàn ngắt cũng phải thực thi ở mức ưu tiên này. |
| configMAX\_SYSCALL\_INTERRUPT\_PRIORITY | Thiết lập mức ưu tiên ngắt cao nhất từ các hàm API FreeRTOS an toàn ngắt được gọi. |

Một mô hình lồng ngắt hoàn chỉnh được tạo ra bởi việc thiết lập configMAX\_SYSCALL\_INTERRUPT\_PRIORITY ở mức ưu tiên cao hơn configKERNEL\_INTERRUPT\_PRIORITY

## **Câu 3: Các hàm API**

1. Hàm **API vSemaphoreCreateBinary()** :Được sử dụng để tạo một semaphore nhị phân

* Hàm:

*Void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );*

* Tham số xSemaphore :Cờ hiệu đang được khởi tạo.

1. **Hàm API xSemaphoreTake()**

* Semaphore chỉ có thể được nhận nếu nó khả dụng
* xSemaphoreTake() không được sử dụng từ một chương trình phục vụ ngắt
* hàm:

portBASE\_TYPE xSemaphoreTake( xSemaphoreHandle xSemaphore, portTickType xTicksToWait );

* + xSemaphore: Semaphore được lấy
  + xTicksToWait :lượng thời gian lớn nhất tác vụ nên duy trì ở trạng thái block để đợi cờ hiệu nếu nó chưa sẵn sàng
  + giá trị trả về
    - pdPASS Chỉ được trả về khi lời gọi xSemaphoreTake() thành công trong việc có được Semaphore
    - pdFALSE Semaphore không khả dụng.

1. **Hàm API xSemaphoreGiveFromISR()**

* Tất cả các loại Semaphore FreeRTOS khác nhau ngoại trừ cờ hiệu đệ quy có thể sử dụng hàm xSemaphoreGiveFromISR()
* xSemaphoreGiveFromISR() là một dạng đặc biệt của xSemaphoreGive() đặc biệt sử dụng trong chương trình phục vụ ngắt.
* hàm

*portBASE\_TYPE xSemaphoreGiveFromISR( xSemaphoreHandle xSemaphore,*

*portBASE\_TYPE \*pxHigherPriorityTaskWoken );*

* Tham số
  + *xSemaphore:*Semaphore được đưa ra *pxHigherPriorityTaskWoken*Có thể là một Semaphore duy nhất sẽ có một hoặc nhiều tác vụ bị chặn trên nó chờ đợi Semaphore để trở nên sẵn sàng. Việc gọi xSemaphoreGiveFromISR() có thể làm cho Semaphore khả dụng, và do đó khiến tác vụ thoát khỏi trạng thái Block. Nếu việc gọi xSemaphoreGiveFromISR() khiến tác vụ thoat khỏi trạng thái Block và tác vụ được mở khóa có mức độ ưu tiên cao hơn tác vụ hiện đang thực thi (tác vụ bị ngắt), thì xSemaphoreGiveFromISR() sẽ thiết lập bên trong \*pxHigherPriorityTaskWoken thành pdTRUE.Nếu xSemaphoreGiveFromISR() thiết lập giá trị này thành pdTRUE thì một sự chuyển đổi ngữ cảnh nên được thực hiện trước khi ngắt được thoát. Điều này đảm bảo ngắt trực tiếp trả về tác vụ trạng thái Ready có mức ưu tiên cao nhất.
* giá trị trả về:
  + pdPASS :Chỉ được trả về khi lời gọi xSemaphoreGiveFromISR() thàn công.
  + pdFAIL :Nếu một cờ hiệu đã có sẵn, nó không thể được đưa ra, và xSemaphoreGiveFromISR() sẽ trả về pdFAIL.
* Các hàm API trong cờ đếm:

1. **Hàm API xSemaphoreCreateCounting()**

* Trước khi một cờ hiệu thực sự được sử dụng, đầu tiên nó phải được tạo ra. Sử dụng hàm API xSemaphoreCreateCounting() để tạo một cờ đếm.
* Hàm:

*xSemaphoreCreateCounting( unsigned portBASE\_TYPE uxMaxCount, unsigned portBASE\_TYPE uxInitialCount );*

* tham số trả về:
  + *uxMaxCount* :Giá trị lớn nhất mà cờ hiệu có thể đếm tới. Giá trị uxMaxCount có ảnh hưởng đến độ dài của hàng đợi
  + *uxInitialCount* :Giá trị đếm khởi tạo của cờ hiệu sau khi nó được tạo.
* Giá trị trả về :
  + Nếu trả về giá trị NULL thì cờ hiệu không được tạo bởi vì không có đủ bộ nhớ heap khả dụng cho FreeRTOS cấp phát cho các cấu trúc dữ liệu cờ hiệu.
  + Nếu trả về giá trị khác NULL cho biết rằng cờ hiệu đã được tạo thành công. Giá trị trả về nên được lưu trữ là công cụ xử lý để tạo cờ hiệu.
* Hàm API trong chương trình phục vụ ngắt
* Có 3 hàm: **xQueueSendToFrontFromISR(), xQueueSendToBackFromISR() và xQueueReceiveFromISR()** là các phiên bản tương ứng của xQueueSendToFront(), xQueueSendToBack() và xQueueReceive() sử dụng an toàn trong chương trình phục vụ ngắt.
* Các hàng đợi được dùng để giao tiếp sự kiện và truyền dữ liệu.

Nguyên mẫu hàm API xQueueSendToFrontFromISR()

*portBASE\_TYPE xQueueSendToFrontFromISR ( xQueueHandle xQueue,*

*void \*pvItemToQueue portBASE\_TYPE \*pxHigherPriorityTaskWoken );*

Nguyên mẫu hàm API xQueueSendToBackFromISR()

*portBASE\_TYPE xQueueSendToBackFromISR( xQueueHandle xQueue,*

*void \*pvItemToQueue portBASE\_TYPE \*pxHigherPriorityTaskWoken );*

* Tham số:
  + xQueue: Xử lý của hàng đợi mà dữ liệu đang được gửi đến (ghi). Xử lý hàng đợi sẽ được trả về từ lời gọi hàm xQueueCreate() được dùng để tạo hàng đợi.
  + pvItemToQueue: Con trỏ trỏ đến dữ liệu được sao chép vào trong hàng đợi.
  + pxHigherPriorityTaskWoken: Một hàng đợi đơn có thể có một hoặc nhiều tác vụ bị chặn trong nó đợi dữ liệu trở nên khả dụng. Gọi xQueueSendToFrontFromISR() hoặc xQueueSendToBackFromISR() có thể làm dữ liệu khả dụng và khiến tác vụ rời trạng thái Block. Nếu việc gọi hàm API khiến một tác vụ rời trạng thái Block, và tác vụ không bị chặn có mức ưu tiên cao hơn tác vụ đang thực thi hiện tại (tác vụ bị ngắt), sau đó hàm API sẽ tự thiết lập \*pxHigherPriorityTaskWoken thành pdTRUE. Nếu xQueueSendToFrontFromISR() hoặc xQueueSendToBackFromISR() thiết lập giá trị này thành pdTRUE, sau đó một sự chuyển đổi ngữ cảnh nên được thực hiện trước khi ngắt bị thoát ra. Điều này đảm bảo ngắt trực tiếp trả về tác vụ trạng thái Ready có mức ưu tiên cao nhất.
* Giá trị trả về :

1. pdPASS: Chỉ được trả về khi dữ liệu được gửi thành công đến hàng đợi

2. errQUEUE\_FULL: Được trả về khi dữ liệu không thể gửi đến hàng đợi vì hàng đợi đã đầy

# **Chương IV. Quản lý tài nguyên và bộ nhớ**

## **Câu 1: Các hàm API cấp phát bộ nhớ,giải phóng bộ nhớ**

**Hàm xPortGetFreeHeapSize():**

Hàm API xPortGetFreeHeapSize () trả về số lượng byte miễn phí trong heap tại thời gian hàm được gọi. Nó có thể được sử dụng để tối ưu hóa kích thước heap.

Ví dụ, nếu xPortGetFreeHeapSize () trả về 2000 sau khi tất cả các đối tượng kernel đã được tạo, sau đó giá trị của configTOTAL\_HEAP\_SIZE có thể giảm đi 2000.

xPortGetFreeHeapSize () không khả dụng khi heap\_3 được sử dụng.

**Hàm xPortGetMinimumEverFreeHeapSize()**

Hàm API xPortGetMinimumEverFreeHeapSize () trả về số lượng tối thiểu byte chưa được phân bổ đã tồn tại trong heap kể từ khi ứng dụng FreeRTOS bắt đầu thực thi.

Giá trị được trả về bởi xPortGetMinimumEverFreeHeapSize () là một dấu hiệu cho thấy mức độ của ứng dụng đã hết khi hết dung lượng.

Ví dụ: nếu xPortGetMinimumEverFreeHeapSize () trả về 200, thì tại một thời điểm kể từ khi ứng dụng bắt đầu thực thi, nó đã xuất hiện trong vòng 200byte hết dung lượng heap. xPortGetMinimumEverFreeHeapSize () chỉ khả dụng khi sử dụng heap\_4 hoặc heap\_5

**Hàm pvPortMalloc()**

pvPortMalloc () có thể được gọi trực tiếp từ mã ứng dụng. Nó cũng được gọi trong FreeRTOS tập tin nguồn mỗi khi một đối tượng kernel được tạo. Ví dụ về các đối tượng kernel bao gồm các tác vụ, hàng đợi, semaphores và các nhóm sự kiện.

Giống như hàm malloc () của thư viện chuẩn, nếu pvPortMalloc () không thể trả về một khối RAM bởi vì một khối kích thước được yêu cầu không tồn tại, sau đó nó sẽ trả về NULL. Nếu pvPortMalloc () được thực thi do trình ghi ứng dụng đang tạo một đối tượng kernel và lệnh gọi pvPortMalloc () trả về NULL, thì đối tượng kernel sẽ không được tạo. Tất cả các lược đồ phân bổ heap mẫu có thể được cấu hình để gọi hàm hook (hoặc gọi lại) nếu một lệnh gọi pvPortMalloc () trả về NULL. Nếu configUSE\_MALLOC\_FAILED\_HOOK được đặt thành 1 trong FreeRTOSConfig.h, thì ứng dụng phải cung cấp hàm hook malloc Failed.

**Hàm pvPortFree():**

Dùng để giải phóng bộ nhớ

## **Câu 2: Mô tả, so sánh các thư viện heap\_1.c,heap\_2.c, heap\_3.c, heap\_4.c, heap\_5.c**

***a. Heap\_1.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

1. ***Heap\_2.c***

Sự sắp xếp này được coi là thuật toán tốt nhất, không giống heap\_1, nó cho phép các khối nhớ trước được giải phóng. Nó không kết hợp các khối free liền kề thành một khối lớn hơn nên dễ bị phân mảnh. 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.
* 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 hợp cho ứng dụng thời gian thực tạo task một cách linh động.

1. ***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

1. ***Heap\_4.c***

Giống như heap\_1 và heap\_2, heap\_4 hoạt động bằng cách phân chia một mảng thành các khối nhỏ hơn. Như trước đó, mảng được khai báo tĩnh, và được định kích thước của bằng TOTAL\_HEAP\_SIZE, do đó, làm cho các ứng dụng xuất hiện để tiêu thụ rất nhiều RAM, thậm chí trước khi bất kỳ bộ nhớ đã thực sự được phân bổ từ mảng.

Heap\_4 sử dụng thuật toán phù hợp để phân bổ bộ nhớ. Không giống như heap\_2, heap\_4 kết hợp các khối bộ nhớ trống liền kề thành một khối lớn hơn, giúp giảm thiểunguy cơ bị phân mảnh bộ nhớ.

Thuật toán phù hợp đầu tiên đảm bảo pvPortMalloc () sử dụng khối bộ nhớ miễn phí đầu tiên lớn đủ để giữ số byte được yêu cầu. Ví dụ, hãy xem xét kịch bản trong đó:

* Heap chứa ba khối bộ nhớ trống, theo thứ tự xuất hiện trong mảng, lần lượt là 5byte, 200byte và 100byte.
* pvPortMalloc () được gọi để yêu cầu 20byte RAM.

Khối RAM miễn phí đầu tiên mà số byte được yêu cầu sẽ phù hợp là 200byte chặn, vì vậy pvPortMalloc () chia khối 200byte thành một khối 20byte và một khối 180byte, trước khi trả về một con trỏ tới khối 20byte. Khối 18byte mới vẫn còn có sẵn cho các cuộc gọi trong tương lai tới pvPortMalloc ().

Heap\_4 kết hợp (kết hợp) các khối miễn phí liền kề thành một khối lớn hơn duy nhất, giảm thiểu nguy cơ phân mảnh và làm cho nó phù hợp với các ứng dụng phân bổ nhiều lần và miễn phí các khối RAM có kích thước khác nhau

1. ***Heap\_5.c***

Thuật toán được sử dụng bởi heap\_5 để phân bổ và bộ nhớ trống giống hệt với thuật toán được sử dụng bởi heap\_4.

Không giống như heap\_4, heap\_5 không giới hạn trong việc phân bổ bộ nhớ từ một khai báo mảng tĩnh; heap\_5 có thể phân bổ bộ nhớ từ nhiều không gian bộ nhớ và tách biệt. Thuật toán này sử dụng cùng một thuật toán kết hợp bộ nhớ và bộ nhớ đầu tiên giống như heap\_4 và cho phép vùng heap trải rộng trên nhiều vùng nhớ không liền kề (không liền kề).

Tại thời điểm ghi, heap\_5 chỉ cung cấp lược đồ cấp phát bộ nhớ được cung cấp duy nhất phải là được khởi tạo rõ ràng trước khi pvPortMalloc () có thể được gọi. Heap\_5 được khởi tạo bằng cách sử dụng hàm vPortDefineHeapRegions (). Hàm này phải được gọi trước bất kỳ lệnh gọi nào tới pvPortMalloc () - không tạo tác vụ, hàng đợi, semaphore, mutex, bộ đếm thời gian phần mềm, nhóm sự kiện, v.v. sẽ dẫn đến pvPortMalloc được gọi.



pxHeapRegions truyền vào một mảng các cấu trúc HeapRegion\_t - mỗi cấu trúc xác định một vùng bộ nhớ có thể được sử dụng làm heap. Mảng được kết thúc bởi cấu trúc HeapRegions\_t có kích thước 0. Vùng có địa chỉ bắt đầu thấp nhất phải xuất hiện đầu tiên trong mảng và các vùng nhớ được xác định trong mảng phải xuất hiện theo thứ tự địa chỉ, từ địa chỉ thấp đến địa chỉ cao.

## **Câu 3: Khái niệm Semaphone, Mutex và mục đích sử dụng**

**Semaphore** là một mục dữ liệu chương trình được sử dụng để quyết định xem các tác vụ có thể tiến hành hay không hay nên tạm dừng. Có hai loại semaphore: các semaphore “nhị phân” (binary) và “chung” (general) hay “đếm” (counting).

Semaphore được sử dụng đồng thời cho hai mục đích rất khác biệt trong phần mềm. Nó hoạt động nhƣ một thiết bị loại trừ lẫn nhaumutex (loại bỏ tranh chấp), một semaphore đƣợc phân bổ cho mỗi tài nguyên đƣợc chia sẻ. Nó cũng được sử dụng để đồng bộ hóa các tương tác tác vụ

**Mutex** (MUTual Exclusion - Tức là loại trừ lẫn nhau) rất giống với một semaphore nhưng được dành riêng cho kiểm soát quyền truy cập vào các nguồn tài nguyên được chia sẻ.

Mục đích chính của cơ chế loại trừ lẫn nhau là kiểm soát quyền truy cập vào tài nguyên được chia sẻ. Tài nguyên đó bao gồm phần cứng, dữ liệu được chia sẻ và phần mềm hệ thống

## **Câu 4: Phân biệt bộ nhớ heap và stack**

Vùng heap của FreeRTOS không chỉ dùng để chứa các Task mà còn chứa các biến được cấp phát động bằng hàm malloc() hoặc pvPortMalloc(), semaphore, Queue.

Khi mỗi Task được tạo ra sẽ chiếm lấy 1 vùng của vùng heap, vùng nhớ mà Task đó chiếm sẽ được Task đó tạo ra 1 vùng nhớ Stack

# **Chương V. xử lý sự cố**

## **Câu 1: Trình bày một số lỗi thông dụng của FreeRTOS**

1. **Lỗi khi thêm tác vụ**

Triệu chứng: Thêm một tác vụ đơn giản vào bản demo khiến cho bản demo bị lỗi.

Việc tạo một tác vụ đòi hỏi bộ nhớ phải được lấy từ heap. Nhiều dự án ứng dụng demo kích thước vùng heap đủ lớn một cách chính xác để tạo các tác vụ demo - vì vậy sau khi các tác vụ được tạo ra sẽ không có đủ dung lượng heap còn lại cho bất kỳ tác vụ, hàng đợi hoặc semaphore nào nữa được thêm vào

Tác vụ nhàn rỗi được tự động tạo khi vTaskStartScheduler() được gọi. vTaskStartScheduler() sẽ chỉ trả lại nếu không có đủ bộ nhớ heap còn lại cho tác vụ nhàn rỗi vừa tạo ra. Bao gồm một vòng lặp rỗng [for (;;); ] sau khi gọi đến vTaskStartScheduler() có thể làm cho lỗi này dễ dàng gỡ lỗi hơn

Để có thể thêm nhiều tác vụ hơn, hãy tăng kích thước heap hoặc xóa một số tác vụ demo hiện có.

1. **Lỗi khi sử dụng hàm API khi ứng dụng đang bị ngắt**

Triệu chứng: Sử dụng chức năng API trong một ngắt khiến ứng dụng bị lỗi

Không sử dụng các hàm API trong các chương trình phục vụ ngắt trừ khi tên của hàm API kết thúc bằng "... FromISR ()"

1. **Lỗi khi lập lịch**

**Triệu chứng 1**: Đôi khi ứng dụng gặp sự cố trong một chương trình phục vụ ngắt.

Điều đầu tiên cần kiểm tra là ngắt không gây ra tràn ngăn xếp overflow.

Cách thức ngắt được định nghĩa và sử dụng khác nhau giữa các cổng và giữa các trình biên dịch - vì vậy điều thứ hai cần kiểm tra là cú pháp, macro và quy ước gọi được sử dụng trong chương trình phục vụ ngắt thường được mô tả chính xác trên trang tài liệu cho bản demo và chính xác như thể hiện bởi chương trình phục vụ ngắt khác trong bản demo.

Nếu ứng dụng đang chạy trên một Cortex M3 thì đảm bảo mức độ ưu tiên được gán cho mỗi ngắt sẽ tính đến các số ưu tiên mức thấp được sử dụng để biểu diễn các ngắt mức ưu tiên logic cao. Điều này có vẻ phản trực giác. Đó là một lỗi phổ biến để vô tình gán một ngắt có sử dụng hàm FreeRTOS API một mức ưu tiên cao hơn mức được xác định bởi configMAX\_SYSCALL\_INTERRUPT\_PRIORITY.

**Triệu chứng 2**: Các lỗi trình lập lịch khi cố gắng để khởi động tác vụ đầu tiên

Nếu một vi điều khiển ARM7 đang được sử dụng thì hãy đảm bảo bộ xử lý đang ở chế độ Giám sát (Supervisor) trước khi vTaskStartScheduler() được gọi. Cách dễ nhất để đạt được điều này là đặt bộ xử lý vào chế độ Supervisor bên trong mã khởi động C trước khi hàm main() được gọi. Đây là cách các ứng dụng demo ARM7 được cấu hình. Bộ lập lịch sẽ không khởi động trừ khi bộ xử lý ở chế độ Supervisor.

**Triệu chứng 3**: Các phần quan trọng (critical section) không lồng nhau một cách chính xác

Không thay đổi các bit cho phép vi điều khiển ngắt hoặc cờ ưu tiên sử dụng bất kỳ phương thức nào khác ngoài các lệnh gọi đến taskENTER\_CRITICAL() và taskEXIT\_CRITICAL(). Các macro này giữ một số lượng chiều sâu lồng lời gọi để đảm bảo các ngắt chỉ được bật lại khi cuộc gọi lồng nhau đã hoàn toàn không bị xóa.

1. **Ứng dụng lỗi trước khi lập lịch**

Triệu chứng: Ứng dụng bị treo ngay cả trước khi trình lập lịch bắt đầu

Một chương trình phục vụ ngắt có khả năng gây ra một chuyển đổi trạng thái không được phép thực hiện trước khi bộ lập lịch đã được bắt đầu. Điều này cũng đúng với bất kỳ chương trình phục vụ ngắt nào cố gửi hoặc nhận từ hàng đợi hoặc semaphore. Một chuyển đổi trạng thái không thể xảy ra cho đến sau khi bộ lập lịch được bắt đầu

Nhiều hàm API không thể được gọi trước khi bộ lập lịch bắt đầu. Tốt nhất là hạn chế sử dụng API để tạo các tác vụ, hàng đợi và các semaphore cho đến sau khi vTaskStartScheduler() được gọi.

1. **Gọi hàm API khi lập lịch bị treo**

Triệu chứng: Gọi các hàm API trong khi trình lập lịch bị tạm ngừng khiến ứng dụng bị lỗi

Trình lập lịch bị tạm ngừng (suspend) bằng cách gọi vTaskSuspendAll() và tiếp tục (không được trả lời) bằng cách gọi xTaskResumeAll().

Không gọi các hàm API trong khi trình lên lịch bị tạm ngừng.

Triệu chứng: Nguyên mẫu của pxPortInitialiseStack() khiến cho việc biên dịch bị lỗi.

Mỗi cổng yêu cầu một macro được định nghĩa để đảm bảo các tệp tiêu đề kernel chính xác được bao gồm trong bản dựng (build). Một lỗi khi biên dịch nguyên mẫu pxPortInitialiseStack() gần như chắc chắn là một triệu chứng của macro này được đặt không đúng cho cổng đang được sử dụng.

Các ứng dụng mới cơ bản trên dự án demo được cung cấp liên quan đến cổng đang được sử dụng. Bằng cách này các tập tin chính xác sẽ được bao gồm và các tùy chọn trình biên dịch chính xác sẽ được thiết lập.

# **Chương VI. Cách sử dụng one-shot timer và auto- reaload timer**

## **Câu 1: Khởi tạo one-shot and auto-reload timers bằng hàm xCreateTimer();**

* One-shot timers:

Sau khi được bắt đầu, chúng chỉ thực hiện chức năng gọi lại đúng một lần. Một one-shot timers có thể được khởi động lại bằng cách tác động trực tiếp chứ không thể tự động tải lại được.

* Auto-reload timers:

Sau khi bắt đầu, auto-reload timers tự khởi động lại mỗi khi hết hạn. Điều này dẫn đến việc thực hiện định kỳ chức năng gọi lại của nó

#include<Arduino\_FreeRTOS.h>

#include <timers.h>

#define mainONE\_SHOT\_TIMER\_PERIOD pdMS\_TO\_TICKS( 3333 )

#define mainAUTO\_RELOAD\_TIMER\_PERIOD pdMS\_TO\_TICKS(500)

TimerHandle\_t xAutoReloadTimer, xOneShotTimer;

void prvAutoReloadTimerCallback( TimerHandle\_t xTimer )

{

Serial.println( "Auto-reload timer callback executing");

}

void prvOneShotTimerCallback( TimerHandle\_t xTimer )

{

Serial.println( "One-shot timer callback executing");

}

void setup() {

Serial.begin(9600);

// Create timer

xAutoReloadTimer = xTimerCreate("AutoReload", mainAUTO\_RELOAD\_TIMER\_PERIOD, pdTRUE, 0, prvAutoReloadTimerCallback);

xOneShotTimer = xTimerCreate("OneShot", mainONE\_SHOT\_TIMER\_PERIOD, pdFALSE, 0, prvOneShotTimerCallback);

BaseType\_t xTimer1Started,xTimer2Started;

if (xAutoReloadTimer == NULL && xOneShotTimer == NULL) {

Serial.println("Timer can not be created");

} else {

// Start timer

xTimer1Started = xTimerStart( xAutoReloadTimer, 0);

xTimer2Started = xTimerStart(xOneShotTimer, 0);

if ( xTimer1Started == pdPASS && xTimer2Started == pdPASS ) { // Start the scheduler

vTaskStartScheduler();

}

}

}

void loop() {

// put your main code here, to run repeatedly:

}