BÀI TỔNG HỢP VỀ RTOS : Real Time Operating System – RTOS

Hệ điều hành thời gian thực ( RTOS ) , cụm từ **thời gian thực** ở đây chỉ ra rằng thời gian phản hồi của hệ thống là rất nhanh.

* **Hệ điều hành thông thường** hiện diện trong máy tính của bạn, khi bạn mở ứng dụng lên thì sẽ có nhiều lúc bạn phải chờ khá lâu. Việc chờ như vậy hầu như không ảnh hưởng gì nhiều lắm, và bạn có thể pha cho mình 1 tách cafe trong khi chờ ứng dụng khởi chạy. Đôi khi ứng dụng lỗi thì chúng ta chỉ cần đóng process rồi chạy lại, gần như chẳng ảnh hưởng gì đến ai, có chăng thì khó chịu một chút thôi.
* **Hệ điều hành thời gian thực** được thiết kế ra cho các nhiệm vụ đặc biệt. Các ứng dụng cần được thực thi với thời gian thật chính xác, các lỗi phát sinh cần được cô lập và xử lý nhanh chóng. Mọi sự chậm trễ, lỗi phát sinh không lường trước có thể khiến hệ thống bị đổ vỡ.

Vì vậy, RTOS sử dụng trong những ứng dụng yêu cầu thời gian đáp ứng nhanh, chính xác về thời gian. Vi điều khiển có tài nguyên rất giới hạn. Do đó, hệ điều hành này chỉ tập trung vào một số ít các tính năng. Chúng cố gắng tối ưu tối đa số luồng, bộ lập lịch và các tác vụ (task) trên một hệ thống cỡ nhỏ.

Thông thường, RTOS là một phân đoạn hoặc một phần của chương trình, trong đó nó giải quyết việc điều phối các task, lập lịch và phân mức ưu tiên cho task, nắm bắt các thông điệp gửi đi từ task. RTOS khá phức tạp, nói một cách dễ hiểu hơn là nó thực hiện việc xử lý các trạng thái máy (State Machine).

Nhân (kernel) của hệ điều hành giao cho CPU thực hiện việc xử lý các task theo những khoảng thời gian. Nó cũng liên tục kiểm tra mức ưu tiên của các task, xắp xếp các thông điệp từ task và tiến hành lập lịch. Trong một hệ thống chạy RTOS, cũng có các tài nguyên dùng chung cùng với phần được cấp phát riêng cho mỗi task.

Các chức năng cơ bản của một RTOS:

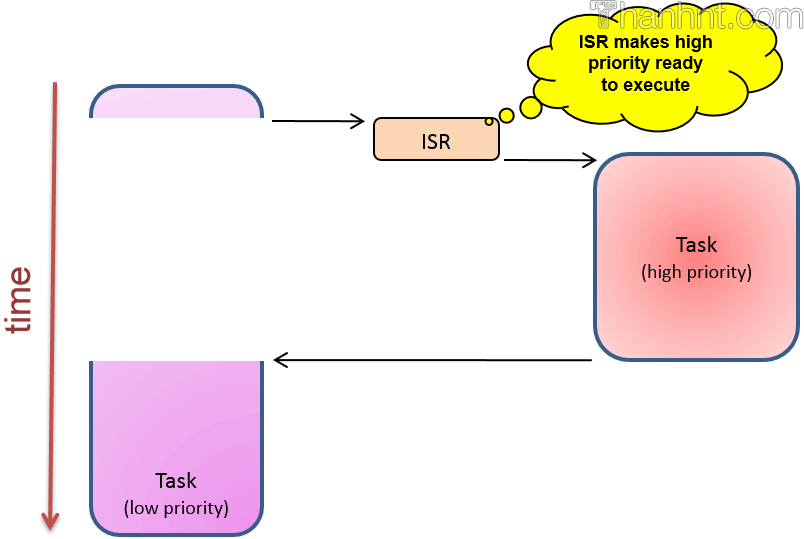
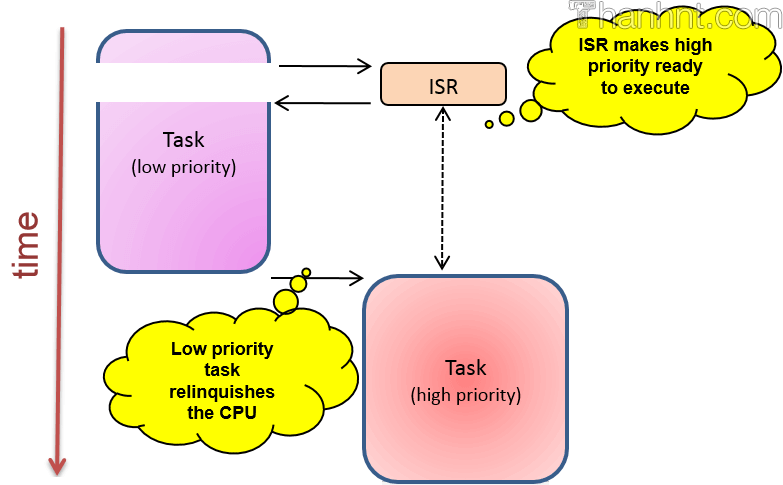
* Bộ lập lịch (Scheduler).
* Các dịch vụ thời gian thực (Realtime Services).
* Đồng bộ và xử lý thông điệp (Synchronization and Messaging).

**Scheduler**

Mỗi task có thể có 3 trạng thái.

* **Ready to run:** Là trạng thái mà task đã có đủ các tài  nguyên để khởi chạy nhưng chưa chạy. Đây là trạng thái chuẩn bị của task.
* **Running:** Là trạng thái mà task đang được thực thi.
* **Blocked:** Khi task không có đủ các tài nguyên cần thiết để chạy thì nó sẽ được đưa về trạng thái blocked.

Để lập lịch cho Task, có 3 kỹ thuật được áp dụng:

* **Co-operative scheduling:** Mỗi task được thực thi đến khi kết thúc quá trình thì task tiếp theo mới được thực thi.
* **Round Robin Scheduling:** Mỗi task được chia cho một khe thời gian cố định, nếu trong khoảng thời gian được chia đó mà task chưa thực hiện xong thì sẽ bị tạm dừng, chờ đến lượt tiếp theo để thực hiện tiếp công việc sau khi hệ thống xử lý hết một lượt các task.
* **Preemptive Scheduling:** Phương pháp này ưu tiên phân bổ thời gian cho các task có mức ưu tiên cao hơn. Mỗi task được gán 1 mức ưu tiên duy nhất. Có thể có 256 mưc ưu tiên trong hệ thống, và có thể có nhiều task có cùng mức ưu tiên.
  + **Preemptive:** Các task có mức ưu tiên cao nhất luôn được kiểm soát bởi CPU, khi phát sinh ISR thì hệ thống sẽ tạm dừng task đang thực thi, hoàn thành ISR sau đó hệ thống thực thi task có mức ưu tiên cao nhất tại thời điểm đó. Sau đó hệ thống mới tiến hành nối lại các task đang bị gián đoạn. Ở chế độ preemptive, hệ thống có thể đáp ứng các công việc khẩn cấp một cách nhanh chóng. Đa số các hệ thống thực tế đang chạy ở chế độ này.
  + **Non-preemptive:** Ở chế độ non-preemptive thì các task được chạy cho đến khi nó hoàn tất. Khi phát sinh ISR thì hệ thống sẽ tạm dừng task đang thực thi và hoàn thành ISR , sau khi hoàn thành ISR thì hệ thống sẽ quay lại thực thi nốt phần việc còn lại của task bị gián đoạn. Task có mức ưu tiên cao hơn sẽ giành quyền kiểm soát CPU sau khi task bị gián đoạn thực thi xong.
  + Ưu điểm của non-preemptive: độ trễ gián đoạn thấp.
  + Nhược điểm của non-preemptive: do phải chờ task thực thi xong thì task có mức ưu tiên cao mới được thực thi, do đó mức đáp ứng của hệ thống thấp. Vì vậy rất ít hệ thống có sử dụng non-preemptive.

Kernel tiến hành quản lý task ở nhiều giai đoạn. Chúng bao gồm:

* Tạo task.
* Huỷ task.
* Thay đổi mức ưu tiên của task.
* Thay đổi trạng thái của task.

**RTOS Services (Dịch vụ thời gian thực)**

Kernel là trái tim của mỗi hệ điều hành. Nó thực hiện chức năng quản lý và lập lịch các quá trình sử dụng CPU và điều phối tài nguyên. Mỗi task chỉ được thưc thi bởi CPU trong một khoảng thời gian, trong các khoảng thời gian còn lại thì task được quản lý bởi các dịch vụ quản lý của hệ điều hành.

Các dịch vụ của RTOS bao gồm:

* Xử lý ngắt (Interrupt handling services).
* Dịch vụ quản lý thời gian (Time services).
* Dịch vụ quản lý thiết bị (Device management services).
* Dịch vụ quản lý bộ nhớ (Memory management services).
* Dịch vụ quản lý các kết nối Vào - Ra (IO services).

**Messaging (Các thông điệp)**

Các thông điệp sử dụng để trao đổi thông tin giữa các hệ thống khác nhau hoặc giữa các task. Các dịch vụ quản lý thông điệp bao gồm:

* Semaphores
* Event flags
* Mailboxes
* Pipes
* Message queues

Semaphores: Dùng để đồng bộ hóa quyền truy cập vào các tài nguyên dùng chung.

Event Flags: Dùng để đồng bộ hóa các hoạt động cần có sự phối hợp của nhiều task.

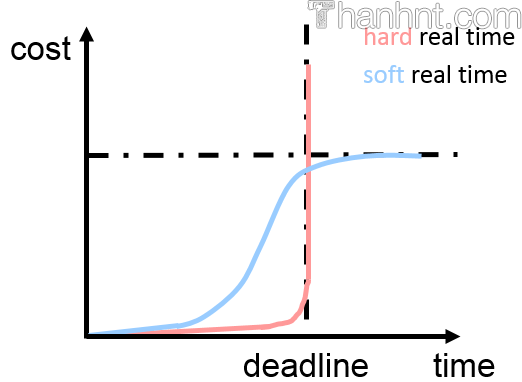
Mailboxes, Pipes, Message queues: Dùng để quản lý các thông điệp gửi đi - đến giữa các task.

**Các dạng thời gian thực**

Một hệ thống có thể chia làm 2 loại thời gian thực dựa vào mức độ trễ - được gọi là thời hạn lập danh mục (scheduling deadline) :

* **Hard - Realtime:** hệ thống phải tiếp nhận và nắm bắt được scheduling deadline của nó tại mỗi và mọi thời điểm. Sự sai sót trong việc tiếp nhận deadline có thể dẫn đến hậu quả nghiêm trọng.
* **Soft - Realtime:** scheduling deadline có dễ thở hơn chút ít, với soft-realtime thì không có gì quá 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 đối với hệ thống mà không dẫn đến hậu quả nghiêm trọng nào khác xảy ra.

Có hệ thống bao gồm cả 2 loại này.



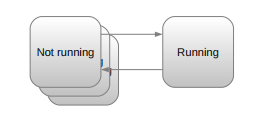
**Tác vụ trong FreeRTOS**

**1. Khái niệm tác vụ (task) trong FreeRTOS**

FreeRTOS cho phép một số lượng không giới hạn các tác vụ miễn là tài nguyên phần cứng và bộ nhớ vẫn còn đáp ứng được. Một tác vụ trong FreeRTOS được định nghĩa bởi một hàm C đơn giản, với đầu vào là một tham số void\* và trả về kiểu void.

FreeRTOS cung cấp các API có sẵn để quản lý tác vụ như là tạo tác vụ (vTaskCreate()), hủy tác vụ (vTaskDelete()), quản lý mức ưu tiên của tác vụ (uxTaskResumeFreeRTOSomISR()).

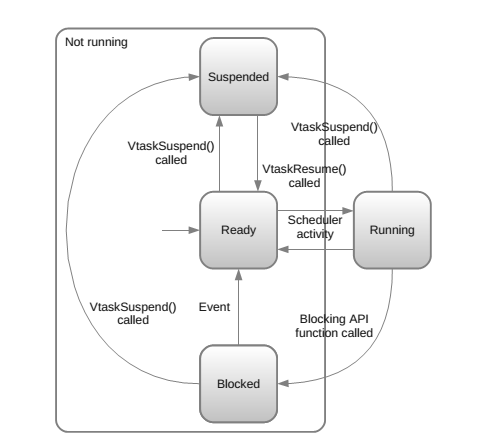
Chu trình sống của một tác vụ là quá trình kể từ khi nó được tạo ra cho đến khi nó bị hủy bỏ. Các tác vụ có thể rơi vào một trong hai trạng thái: “running” hoặc “not running”. Giả sử bộ vi xử lý có một nhân thì duy nhất một tác vụ có thể chạy ở một thời điểm. Các tác vụ khác ở trạng thái “not running”. Hình 1 thể hiện một cách đơn giản một chu kỳ sống của tác vụ. Khi một tác vụ chuyển trạng thái của nó từ “Not running” sang trạng thái “running” được gọi là “swapped in” hoặc “switched in” ngược lại gọi là “swapped out” hay “switch out” khi nó thay đổi trạng thái từ “running” sang “not running”.

[](https://sites.google.com/site/embedded247/embedded_system/tim-hieu-ve-freertos-2/TaskLifeCycle.png?attredirects=0)

Hình 1. Chu kỳ sống đơn giản của một task

Có một vài lý do để các tác vụ còn lại không ở chế độ running. Trạng thái “not running” được diễn tả ở hình 2. Một tác vụ bị ngăn không cho chạy có thể vì độ ưu tiên của nó thấp hơn độ ưu tiên của các tác vụ khác hoặc vì bị delay hoặc đang đợi một sự kiện xảy ra. Khi một tác vụ có thể “running” nhưng nó đang đợi hệ điều hành cấp phát tài nguyên bộ vi xử lý, trạng thái đó được gọi là “ready”.  Điều này cũng có thể xảy ra khi một tác vụ đủ điều kiện để chuyển sang “running” nhưng lại có tác vụ khác có độ ưu tiên cao hơn đang chạy. Khi một tác vụ bị delay hoặc đang đợi tác vụ khác (đồng bộ hóa qua semaphores hoặc mutex) thì tác vụ đó được gọi là “blocked”. Và cuối cùng, khi một lời gọi tới vTaskSuspend(), vTaskResume() hoặc xTaskResumeFreeRTOSomISR() sẽ làm tác vụ chạy và thoát khỏi trạng thái “Suspend”.

Điều quan trọng cần nhấn mạnh ở đây là khi một tác ra khỏi trạng thái “Running” bởi chính nó (delay, suspend hoặc chờ đợi một sự kiện), chỉ có bộ lập lịch (scheduler) mới có thể chuyển trạng thái của nó trở lại về “running”. Khi một tác vụ muốn trở về trạng thái “running” thì nó phải trở về trạng thái “ready” trước. Sau đó bộ lập lịch có thể chọn một trong các tác vụ ở trạng thái “ready” để chạy.

[](https://sites.google.com/site/embedded247/embedded_system/tim-hieu-ve-freertos-2/TaskLifeCycle2.png?attredirects=0)

Hình 2. Chu trình sống của một task

**2. Tạo và xóa một tác vụ**

Một tác vụ được định nghĩa bởi một hàm C đơn giản với tham số là một con trỏ void\* và không cần giá trị trả về:

      // A typical task signature

void ATaskFunction( void \*pvParameters );

Bất cứ đoạn chương trình nào tạo một tác vụ mới thì đoạn chương trình đó không nên kết thúc trước khi tác vụ đó bị hủy bỏ. Thông thường đoạn chương trình tạo task đó có một vòng lặp vô hạn hoặc là có một lời gọi vTastDetroy(NULL) trước khi kết thúc. Việc cài đặt một task điển hình được chỉ ra trong text 3.

Có thể tạo một task bằng hàm vTaskCreate()   (text 2). Các tham số như sau:

·        pvTaskCode: con trỏ tới hàm task

·        pcName: Tên, không có ý nghĩa đối với hdh FreeRTOS và chỉ có mục đích là phục vụ debug

·        usStackDepth: kích thước của stack tính bằng WORD. Kích thước thực sự của stack phục thuộc vào trình điều kiển thiết bị. Nếu stack là 32 bit (4byte) và usStackDepth là 100 thì kích thước stack sẽ là 4\*100 = 400 byte.

·        pvParameters: con trỏ tới các tham số đã được chứa trong stack,  Cách tốt nhất là tạo một cấu trúc dữ liệu riêng, điền các tham số vào cấu trúc dữ liệu đó và truyền tham số là con trỏ tới cấu trúc đó.

·        uxPriority: Độ ưu tiên cho task, một số giữa 0 tới MAX\_PRIORITIES   –   1. Được trình bày trong mục 2.

·        pxCreatedTask: con trỏ tới một định danh cho phép xử lý task, nếu task không có xử lý sau này thì có thể để tham số này là NULL

portBASE\_TYPE xTaskCreate( pdTASK\_CODE pvTaskCode,

                           const signed portCHAR \* const pcName,

                           unsigned portSHORT usStackDepth,

                           void \*pvParameters,

                           unsigned portBASE\_TYPE uxPriority,

                           xTaskHandle \*pxCreatedTask

                         );

**Text 2: Task creation routine**

void ATaskFunction( void \*pvParameters )

{

/\* Variables can be declared just as per a normal function.  Each instance

of a task created using this function will have its own copy of the

iVariableExample variable.  This would not be true if the variable was

declared static – in which case only one copy of the variable would exist

and this copy would be shared by each created instance of the task. \*/

int iVariableExample = 0;

    /\* A task will normally be implemented as in infinite loop. \*/

    for( ;; )

    {

        /\* The code to implement the task functionality will go here. \*/

    }

    /\* Should the task implementation ever break out of the above loop

    then the task must be deleted before reaching the end of this function.

    The NULL parameter passed to the vTaskDelete() function indicates that

    the task to be deleted is the calling (this) task. \*/

    vTaskDelete( NULL );

}

**Text 3: A typical task**

Sử dụng hàm xTaskDetroy() để hủy một task. Hàm sử dụng tham số như tham số trong hàm pxCreatedTask():

void vTaskDelete( xTaskHandle pxTask );

**Text 4: Deleting a task**

Khi một task bị xóa, các bộ nhớ cấp phát sẽ được giải phóng bởi kernel. Lưu ý các bộ nhớ vùng động được cấp phát trước đó phải giải phòng bằng người lập trình.

# Cấu trúc chương trình user\_main.c

Chương trình có nhiệm vụ tạo ra 2 task riêng biệt, một dùng để nháy LED có chu kỳ 200 ticks, task còn lại để in thông tin ra UART0 với chu kỳ 1000 ticks.

Muốn vậy, trước hết phải tạo ra 2 hàm con, tuân theo tiền khai báo kiểu TaskFunction\_t có dạng void vTaskCode( void \* pvParameters ) cho task LED và UART:

**void** **task\_led**(**void** \*pvParameters)

{

**for**(;;){

vTaskDelay(100);

GPIO\_OUTPUT\_SET(LED\_GPIO, led\_state);

led\_state ^=1;

}

}

**void** **task\_printf**(**void** \*pvParameters)

{

**for**(;;){

printf("task\_printf\n");

vTaskDelay(500);

}

}

**Lưu ý**

* Bên trong hàm con phải thực hiện vòng lặp vô tận (infinite loop), không được return.
* Ngoài ra, task có thể tự hủy (delete) chính nó khi cần (bằng hàm vTaskDelete( TaskHandle\_t xTask ) - xem ví dụ bên dưới)

**Tick**

* Tick là hành động khi timer ngắt định kỳ dùng trong nhân FreeRTOS để MCU thực hiện chuyển ngữ cảnh (context) khi chuyển qua lại giữa các task với nhau, khái niệm thực hiện tác vụ song song, đồng thời, cùng lúc chỉ mang tính tương đối, vì RTOS thực hiện điều này 1 cách tuần tự.
* Giá trị của tick không phải lúc nào cũng là 1ms, tùy thuộc vào cấu hình khi port FreeRTOS lên từng MCU khác nhau (trong trường hợp, configTICK\_RATE\_HZ của RTOS-SDK có giá trị là 100). Có thể kiểm tra chính xác chu kỳ ms của tick bằng macro portTICK\_RATE\_MS –> Để delay chính xác t(ms) thì tham số truyền cho vTaskDelay là t/portTICK\_RATE\_MS

Trong hàm user\_init() của ESP8266, sau khi khởi tạo các giá trị cần thiết cho UART và chân GPIO Ouput để nháy LED, sử dụng hàm xTaskCreate trong FreeRTOS để tạo task thực thi 2 hàm con này, cú pháp xTaskCreate:

BaseType\_t **xTaskCreate**( TaskFunction\_t pvTaskCode,

**const** **char** \* **const** pcName,

**unsigned** **short** usStackDepth,

**void** \*pvParameters,

UBaseType\_t uxPriority,

TaskHandle\_t \*pxCreatedTask

);

Trong đó:

* pvTaskCode: trỏ đến hàm con cần thực hiện khi tạo task
* pcName: chuỗi mô tả tên của task này
* usStackDepth: độ lớn của con trỏ ngăn xếp, chọn sao cho lớn hơn độ lớn của con trỏ ngăn xếp khi thực hiện hàm con, ví dụ như khi hàm con gọi càng nhiều hàm khác bên trong lồng nhau, khi đó độ lớn này càng tăng.
* pvParameters: trỏ đến tham số cần truyền vào hàm con khi task khởi tạo.
* uxPriority: mức độ ưu tiên của task.
* pxCreatedTask: trỏ đến biến kiểu TaskHandle\_t, biến sẽ được gán sau khi gọi xTaskCreate thành công, xem như ID để phân biệt các task với nhau, và được sử dụng cho nhiều mục đích, ví dụ như xóa task (dùng hàm vTaskDelete( TaskHandle\_t xTask )) Sử dụng xTaskCreate để tạo task LED và UART như sau:

xTaskCreate(task\_led, "task\_led", 256, NULL, 2, NULL);

xTaskCreate(task\_printf, "task\_printf", 256, NULL, 2, NULL);

# Biên dịch và chạy chương trình

make clean

make

make flash

# Nâng cao một chút

* In vài thông tin cơ bản về portTICK\_RATE\_MS và configMAX\_PRIORITIES
* Ví dụ về vTaskDelete() cho task\_printf

git checkout task\_delete

make clean

make

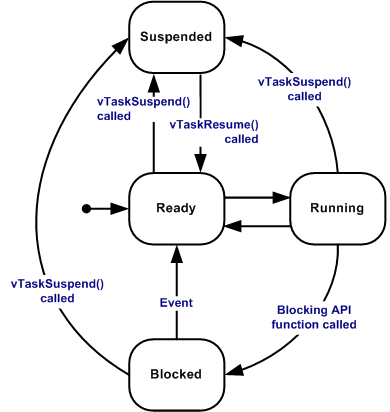
make flash

# Giới thiệu về tác vụ (task) trong FreeRTOS

## Các trạng thái (states) của task

Có 4 trạng thái: Running, Ready, Blocked và Suspended

* Running: là trạng thái task đang được MCU thực thi thực sự, vì thế trong một thời điểm chỉ có duy nhất một task ở trạng thái running.
* Ready: là trạng thái task đã sẵn sàng để thực thi (không bị blocked hoặc suspended) nhưng đang không được MCU thực thi bởi vì MCU đang thực thi một task khác.
* Blocked: task đang bị blocked nếu nó đang đợi sự kiện (event) bên ngoài hoặc sự kiện thời gian. Ví dụ: khi task gọi hàm vTaskDelay() thì nó sẽ bị blocked cho đến khi hết thời gian delay (sự kiện thời gian). Hoặc task có thể bị blocked để đợi một hàng đợi (queue), semaphore, notification… nào đó. Thông thường, task bị blockedtrong khoảng thòi gian quá hạn timeout cho trước, vì thế task sẽ luôn được unblocked (nếu có sự kiện bên ngoài) hoặc timeout nếu hết thời gian chờ.
* Suspended: Cũng giống như trạng thái blocked nhưng không có thời gian timeout, vì vậy chỉ có thể enterhoặc exit khỏi trạng thái suspended bởi hàm gọi từ bên ngoài tương ứng là vTaskSuspend() và xTaskResume()



## Các mức ưu tiên (priorities) của task

Mỗi task được tạo ra với mức ưu tiên được gán từ 0 đến giá trị (configMAX\_PRIORITIES - 1), với configMAX\_PRIORITIES là giá trị được định nghĩa trong FreeRTOSConfig.h (với bản RTOS-SDK v1.4 đang sử dụng, configMAX\_PRIORITIES là 15)

Task đang ở trạng thái ready có ưu tiên cao hơn sẽ được chọn để thực thi (chuyển sang running) trong mỗi lần tick

Nếu các task có cùng mức ưu tiên? Trong trường hợp RTOS sẽ chia đều ra xử lý (do giá trị configUSE\_TIME\_SLICING đã được định nghĩa là 1 trong bản RTOS-SDK này)

# Vấn đề đồng bộ / trao đổi dữ liệu giữa các task

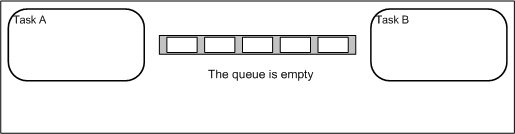
Mỗi task thực hiện tác vụ riêng của mình, chúng có không gian bộ nhớ riêng và hoàn toàn độc lập với nhau.

* Vậy làm thế nào để phối hợp chúng trong một chương trình lớn hơn, nếu xét mỗi task như là một module?
* Nếu task này muốn giao tiếp/trao đổi thông tin với task kia thì làm thế nào?
* Nếu 2 task cùng mức ưu tiên, cùng chạy song song và cùng muốn truy cập đến một tài nguyên chung thì làm sao để tránh xung đột?

Tất cả chỉ gói gọn trong hàng đợi (queue), semaphore và mutex

# Hàng đợi (queue)

Hoạt động theo cơ chế FIFO (First In - First Out) dùng trong FreeRTOS để trao đổi thông điệp (message) giữa các task với nhau.



## Ví dụ

Sử dụng lại ví dụ esp-rtos-basic-task, nhưng 2 task sửa lại theo mục đích sau:

* task\_led: nháy led, đếm biến counter, đếm xong thì gửi giá trị biến đếm này qua queue cho task\_printfin ra giá trị.
* task\_printf: chỉ chờ có thông điệp mới trên queue thì lấy thông điệp này ra khỏi queue và in giá trị này!

Mã nguồn chương trình:

git clone https://github.com/esp8266vn/esp-rtos-basic-queue-sem-mutex.git

cd esp-rtos-basic-queue-sem-mutex

Để sử dụng được queue phải #include "freertos/queue.h"

## Tạo queue

Cú pháp:

xQueueHandle xQueueCreate( UBaseType\_t uxQueueLength,

UBaseType\_t uxItemSize )*;*

* uxQueueLength: kích thước/số phần tử tối đa trên queue
* uxItemSize: kích thước của mỗi phần tử, tùy thuộc vào kiểu dữ liệu của phần tử trên queue.

Trường hợp này, biến counter là uint32\_t. Giả sử cần tối đa 10 phần tử trên queue. Khai báo và tạo như sau:

xQueueHandle xCounterQueue;

*// ...*

xCounterQueue = xQueueCreate( 10, **sizeof**(**uint32\_t**));

## Gửi thông điệp lên queue

Theo cú pháp:

BaseType\_t **xQueueSend**(

QueueHandle\_t xQueue,

**const** **void** \* pvItemToQueue,

TickType\_t xTicksToWait

);

* xQueue: queue handle (xCounterQueue)
* pvItemToQueue: trỏ đến phần tử/thông điệp cần gửi lên queue (&counter)
* xTicksToWait: thời gian chờ tối đa (timeout - tính bằng tick) trong trường hợp queue bị đầy không gửi thêm vào được.

task\_led sẽ làm nhiệm vụ này:

**void** **task\_led**(**void** \*pvParameters)

{

**uint32\_t** counter = 0;

**for**(;;){

vTaskDelay(100);

GPIO\_OUTPUT\_SET(LED\_GPIO, led\_state);

led\_state ^=1;

counter++;

**if**( xQueueSend( xCounterQueue,( **void** \* ) &counter, 10 ) != pdPASS )

{

*// Failed to send msg to queue, TODO handle*

}

}

}

## Nhận thông điệp từ queue

Theo cú pháp:

BaseType\_t **xQueueReceive**(

QueueHandle\_t xQueue,

**void** \*pvBuffer,

TickType\_t xTicksToWait

);

* xQueue: queue handle (xCounterQueue)
* pvBuffer: trỏ đến nơi sẽ chứa thông điệp lấy từ queue xuống.
* xTicksToWait: timeout trong trường hợp không có phần từ nào trên queue (đang trống)

task\_printf sẽ thực hiện nhiệm vụ này:

**void** **task\_printf**(**void** \*pvParameters)

{

**uint32\_t** receiveCounter;

printf("task\_printf started\n");

**for**(;;){

**if**( xQueueReceive( xCounterQueue, &( receiveCounter ), 100 ) )

{

printf("task\_printf, received counter = %u\n", receiveCounter);

}

}

}