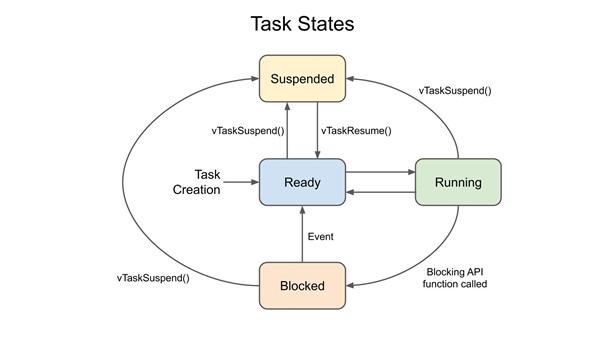
# 1\_GIỚI THIỆU

Ở trong FreeRtos thì người ta coi “task” là “thread” nhưng bạn cần phải phân biệt chúng 1 cách rạch ròi:

·       **Task**: tập hợp các lệnh được tải vào bộ nhớ. Có thể hiểu đơn giản đây là 1 số đơn vị công việc hoặc mục tiêu cần phải hoàn thành.

·       **Thread**: là 1 đơn vị của CPU với bộ đếm chương trình và bộ nhớ ngăn xếp của riêng nó.

## TASK STATE:



Trong Rtos thì task có 4 trạng thái:

·       **Ready**: Task đã sẵn sàng để có thể thực thi nhưng chưa được thực thi do có các task khác với độ ưu tiên ngang bằng hoặc hơn đang chạy (tương tự như đối với ngắt).

·       **Running**: Task đang thực thi.

·       **Blocked**: Task đang chờ 1 sự kiện nào đó xảy ra, sự kiện này có thể là khoảng thời gian hoặc 1 sự kiện nào đó từ task khác.

·       **Suspended**: Task ở trạng thái treo, về cơ bản thì trạng thái này cũng tương tự như Blocked. Nhưng điểm khác nhau là “cách” chuyển từ trạng thái hiện tại sang Ready State. Chỉ khi gọi hàm vTaskResume() thì task bị treo mới được chuyển sang trạng thái Ready để có thể thực thi.

## TASK STRUCTURE:

Void MyTask1(**void \*param**)

{

           //các lệnh khởi tạo.

           While(**1**)

           {

                       //các lệnh thực thi.

           }

}

Khi xây dựng 1 task ta cần chú ý, mỗi task nên là 1 “siêu vòng lặp” vì nếu không chip của bạn sẽ bị reset khi task chạy xong các lệnh của nó. Để tránh điều này ta cần Delete task đó trước khi nó kết thúc bằng hàm: vTaskDelTrong FreeRtos mỗi Task sẽ được gán:

·       1 mức ưu tiên (priority): tùy vào thư viện mà rank của mức độ ưu tiên có thể được định nghĩa khác nhau (càng thấp càng được ưu tiên hoặc càng cao càng được ưu tiên).

·       1 “cái tên”: chủ yếu được sử dụng để debug.

·       Size: Kích thước ngăn xếp của task.

·       Tham số truyền vào.

·       1 con trỏ đại diện cho task (xTaskHandle): thường được sử dụng để điều khiển task từ 1 task khác.

## TASK PROPERTIES:

Trong FreeRtos mỗi Task sẽ được gán:

·       1 mức ưu tiên (priority): tùy vào thư viện mà rank của mức độ ưu tiên có thể được định nghĩa khác nhau (càng thấp càng được ưu tiên hoặc càng cao càng được ưu tiên).

·       1 “cái tên”: chủ yếu được sử dụng để debug.

·       Size: Kích thước ngăn xếp của task.

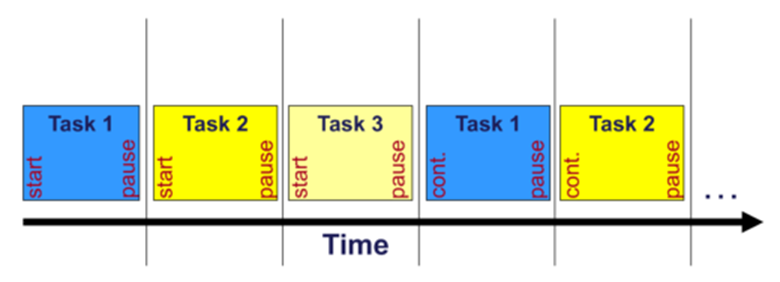
·       Tham số truyền vào.

·       1 con trỏ đại diện cho task (xTaskHandle): thường được sử dụng để điều khiển task từ 1 task khác.

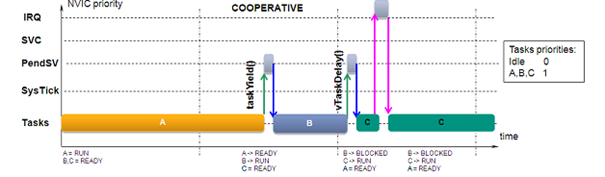
## TASK SCHEDULING:

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

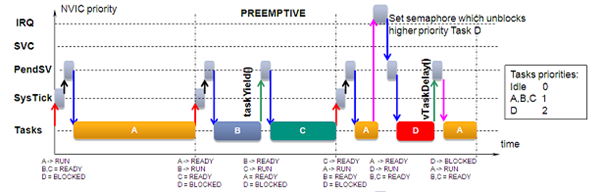
·       **Round Robin Scheduling:** Mỗi task được chia cho một khe thời gian cố định, nếu trong khoảng thời gian được chia đó mà task chưa thực hiện xong thì sẽ bị tạm dừng, chờ đến lượt tiếp theo để thực hiện tiếp công việc sau khi hệ thống xử lý hết một lượt các task.



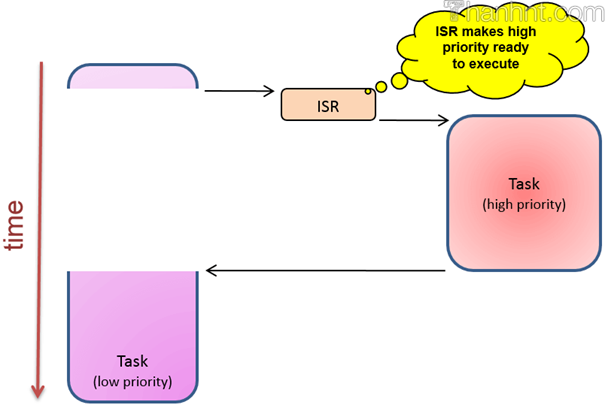
·       **Co-operative scheduling:** Mỗi task được thực thi đến khi kết thúc quá trình thì task tiếp theo mới được thực thi.



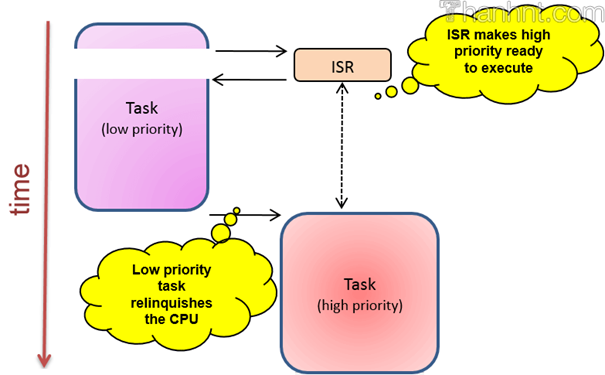
·       **Preemptive Scheduling:** Phương pháp này ưu tiên phân bổ thời gian cho các task có mức ưu tiên cao hơn. Mỗi task được gán 1 mức ưu tiên duy nhất. Có thể có 256 mưc ưu tiên trong hệ thống, và có thể có nhiều task có cùng mức ưu tiên.



o  **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.



o  **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.



# 3\_Interrupt

Khi ngắt thông thường xảy ra nó sẽ nhảy vào hàm ngắt và các công việc cần phải làm đều đặt trong hàm ngắt.

Còn ở RTOS thì khi ngắt xảy ra nó sẽ nhảy vào hàm thủ tục ngắt, hàm thủ tục ngắt lại phát đi 1 "tín hiệu" để hàm thực hiện chức năng của ngắt Unblocking và hàm này có Priority cao hơn tất cả các task khác nên nó sẽ thực hiện ngay lập tức.

  Không phải API nào của RTOS cũng được phép gọi trong interrupt, chỉ những API có đuôi FromISR mới được phép gọi trong interrupt. Bởi vì đây là các hàm an toàn. Như là:

o   xTaskResumeFromISR()

o   xQueueSendFromISR()

# 4\_Queue  các task trao đổi dữ liệu với nhau

Trong quá trình chạy multi-task sẽ có những lúc các task cần sử dụng chung tài nguyên cũng như trao đổi dữ liệu với nhau, thường thì những bạn mới tiếp cận có thể sẽ nghĩ ngay đến biến toàn cục,… Việc này tuy khả thi nhưng có rất nhiều bất cập như không kiểm soát đúng được luồng dữ liệu. ---🡪 Queue sẽ giúp giải quyết vấn đề này 1 cách ổn thỏa nhất.

## **Khái niệm**

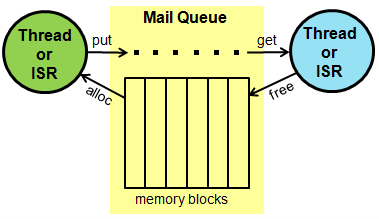
Queue là một loại buffer theo kiểu FIFO (first in first out) nơi dữ liệu sẽ được ghi vào cuối của queue. Tuy nhiên ta vẫn có thể ghi dữ liệu vào đầu queue bằng cách sử dụng hàm của thư viện.

Các thao tác đơn giản với queue



## Phụ lục: Mail Queue

Thực ra Mail Queue là 1 dạng của Queue. Ở trong thư viện FreeRTOS thì không hỗ trợ API của phần này nhưng trong thư viện CMSIS (thư viện RTOS trong STM32-cubeMx) thì có. Với Mail Queue thì data được gửi đi dưới dạng block, các memory block này cần được cấp phát (allocate) trước khi đưa data vào và giải phóng(free) sau khi cho data ra.



## **Các thao tác đơn giản với Mail Queue**





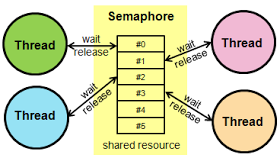
# 5\_Semaphore

cách đồng bộ giữa các task với nhau và với ngắt thông qua Semaphore

Khi thực hiện multi-task trong 1 project. Nếu ta cứ tạo ra các task và để cho chúng chạy độc lập với nhau thì sẽ rất khó khăn trong việc phân chia tài nguyên cho các task cũng như kiểm soát thuật toán của bản thân. Vì thế nên Semarphore được xây dựng để giúp chúng ta giải quyết vấn đề này 1 cách ổn áp nhất.

## Khái niệm:

Semaphore giống như 1 cái hàng đợi (queue), dùng để quản lý và bảo vệ tài nguyên dùng chung(share resource). Các thread/task khác nhau khi có yêu cầu sử dụng tài nguyên dùng chung sẽ bị tống vào hàng đợi này. Khi nhận được semaphore token thì thread/task nào được tống vào queue trước thì sử dụng tài nguyên trước. Sau đó nó lại release ra cho thread/task khác dùng.

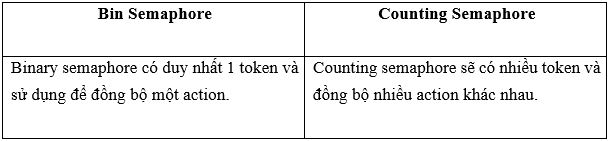


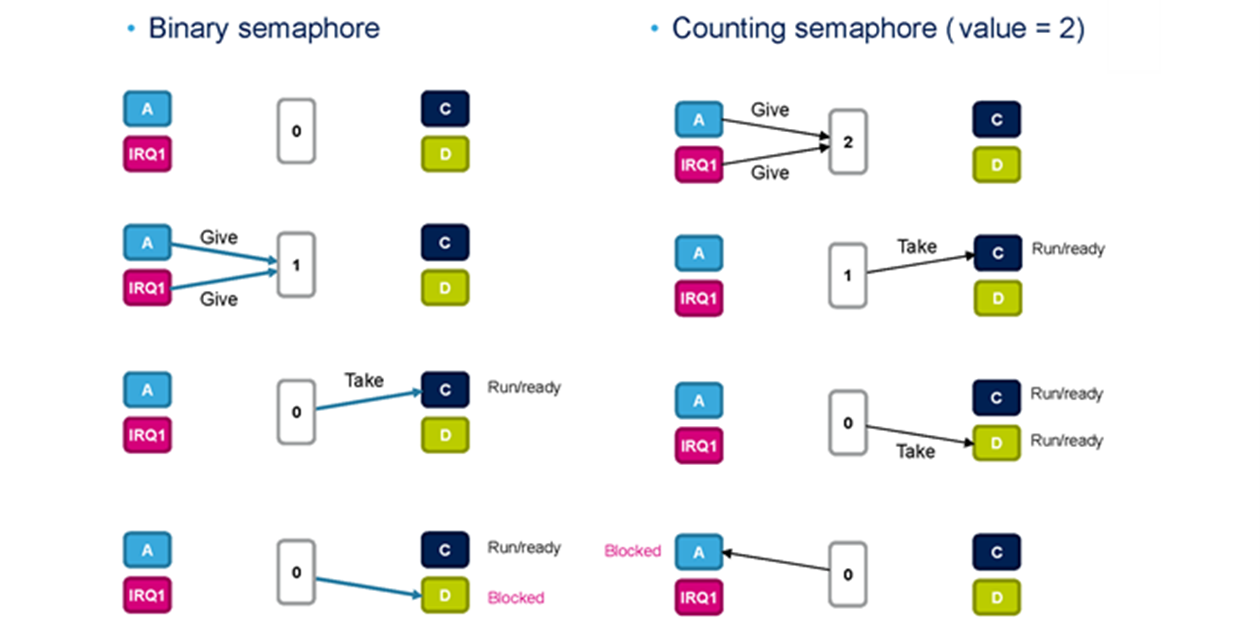
## **Nguyên lý hoạt động:**

Các bạn hãy tưởng tượng, có 1 căn phòng (tài nguyên của CPU), và để có thể vào được căn phòng đó để thực hiện nhiệm vụ (các câu lệnh) thì các task cần phải có chìa khóa (Semaphore). Và số lượng chìa khóa có thể đánh ra được nhiều cái tùy người dùng config. Trước khi vào phòng, các task sẽ “take” semaphore có sẵn, nếu không thì task sẽ “wait” cho tới khi có semaphore, sau khi đã có được semarphore task sẽ thực thi nhiệm vụ của mình và “Give” semaphore về lại “chỗ cũ”. Ngoài các task ra thì các hàm xử lý ngắt cũng có thể “Give” semaphore.

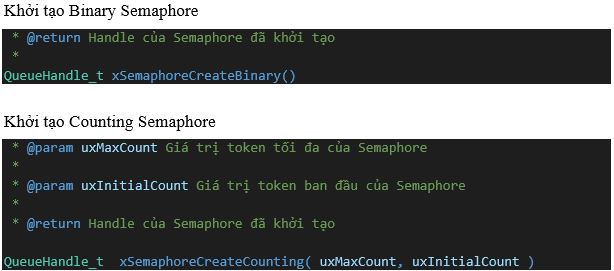
## Phân loại

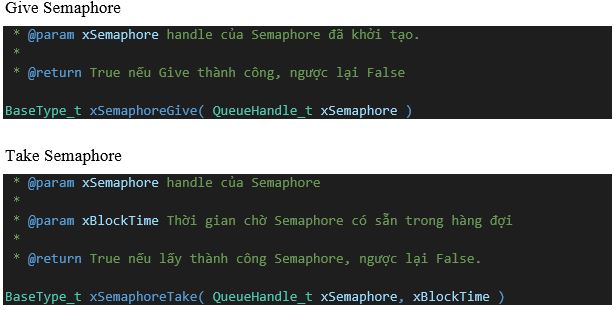
Semaphore được phân làm 2 dạng chính:





## Các thao tác cơ bản với Sermaphore



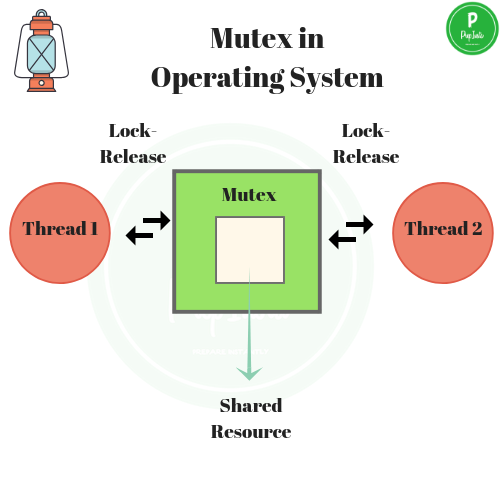


**6\_MUTEX**

Một người anh em cùng cha khác mẹ với Semaphore

## Tổng quan

Về cơ bản thì MUTEX tương tự như bin Semaphore nhưng có tích thêm cơ chế “kế thừa mức ưu tiên” và được dùng cho mục đích loại trừ (hạn chế quyền truy cập vào resource của các task khác) chứ không phải là để đồng bộ như Semaphore.



**Kế thừa mức ưu tiên**: Khi task có mức ưu tiên cao hơn muốn take MUTEX trong khi task có mức ưu tiên thấp hơn đang giữ nó thì, mức ưu tiên của 2 task sẽ được tráo cho nhau, cho tới khi task kia give MUTEX thì mức ưu tiên sẽ hoàn trả lại như cũ. Cơ chế này được thiết kế nhằm chắc chắn rằng các task có mức ưu tiên cao chỉ ở trạng thái block trong thời gian ngắn nhất có thể nhằm tránh tình trạng deadlock của hệ thống.

## Nguyên lý hoạt động.

Khi 1 task muốn truy cập vào tài nguyên để thực thi nhiệm vụ thì sẽ take Mutex. Trong lúc đó, bất kì task nào muốn take Mutex đều bị block cho tới khi task đang giữ Mutex “give” về chỗ cũ

Có thể hiểu đơn giản Mutex giống như khóa tủ gửi đồ trong siêu thị, muốn mở được tủ khóa để cất đồ thì cần có chìa khóa, và sau khi dùng xong phải để chìa khóa về chỗ cũ cho người khác dùng.

## Một số điểm khác so với Semaphore.

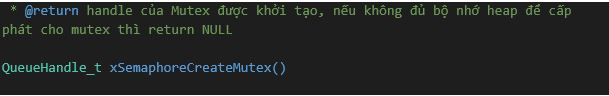
Task đang sở hữu mutex sẽ không bao h phải chờ task khác thực thi, còn task đang sở hữu semaphore thì có thể vẫn phải chờ task khác thực thi xong.

Chỉ có thằng cầm mutex mới được give mutex, bất kì th nào cũng có thể give semaphore. Chính vì vậy trình phục vụ ngắt có thể give Semaphore nhưng đ.v Mutex thì không.

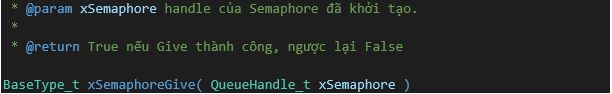
Có thể có nhiều Semaphore, nhưng chỉ có duy nhất 1 mutex.

## Các thao tác cơ bản với Mutex.

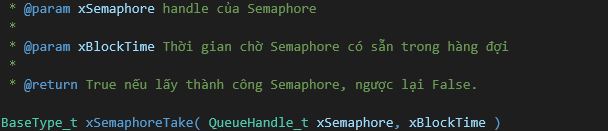
Khởi tạo Mutex



Give Mutex



Take Mutex



# 7\_Software Timer

Timer "mềm", liệu có được xịn sò như Hareware Timer

## Tổng quan

**Khái niệm**

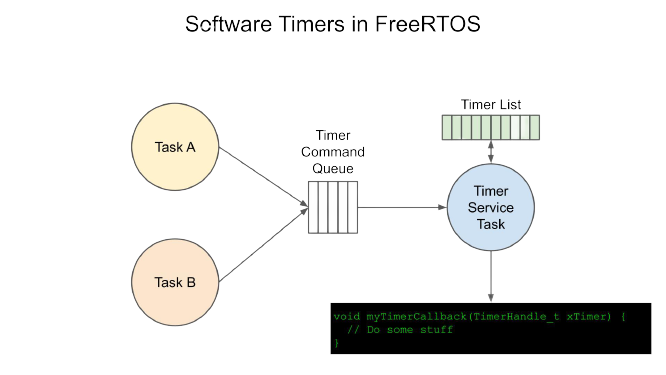
Có thể hiểu đơn giản, trong RTOS, software timer giúp chúng ta thực hiện 1 nhiệm vụ nào đó với 1 chu kỳ nhất định.

## **Phân loại**

One-shot timer: khi tới thời gian định kỳ được cài đặt từ trước thì hàm callback (vai trò như trình phục vụ ngắt) chỉ được thực thi 1 lần duy nhất. Tuy nhiên vẫn có thể khởi động lại theo cách thủ công chứ không tự khởi động lại.

Auto-reload timer: Khác với One-shot timer là nó sẽ tự khởi động lại sau khi thực thi hàm callback.

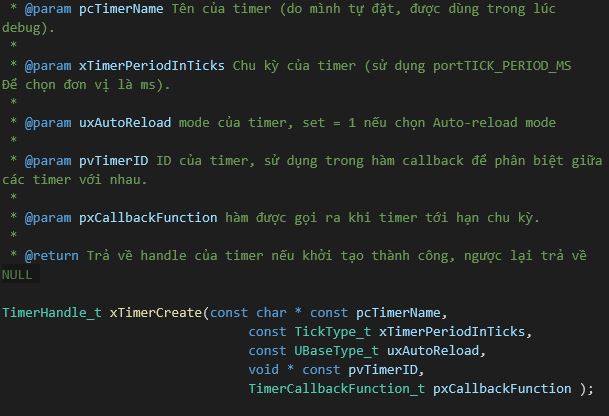
Nguyên lý hoạt động



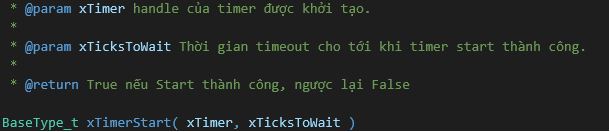
Có 1 task được gọi là Tmr Svc (Timer Serivce) sẽ duy trì “list software timer” được đặt ra. Task này sẽ không chạy liên tục vì nó biết thời gian mà mỗi timer trong timer list “wake up”. Khi timer đến kỳ hạn của nó, thì Tmr Svc sẽ gọi hàm callback của timer đó.Ngoài ra, Tmr Svc còn sử dụng 1 hàng đời Queue, để các task vụ khác có thể giao tiếp và điều khiển timer trong này (start hoặc stop timer).

## Các thao tác cơ bản với Software Timer

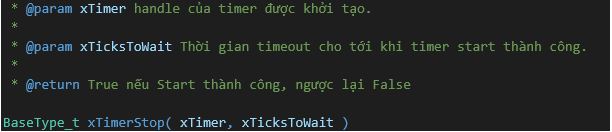
**Tạo Software Timer**



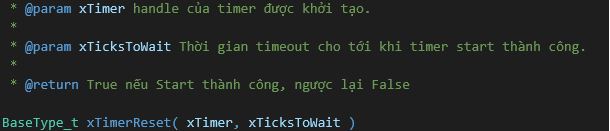
**Start Software Timer**



**Stop Software Timer**



**Reset Timer:**

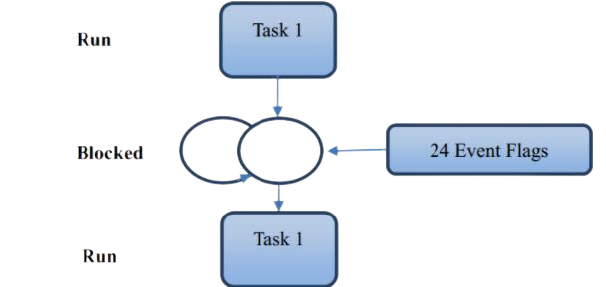


# 8\_Event Groups

Một cách khác để đồng bộ các task mà không phải dùng Semaphore nhưng tiết kiệm bộ nhớ hơn

## Tổng quan:

Cũng như Semaphore hay mutex, Event group có thể kiến 1 task rơi vào trạng thái block cũng như unblock task đó. Tuy nhiên, khác ở chỗ, task bị block có thể phải chờ cho nhiều sự kiện. và cùng một lúc có thể unblock nhiều task bằng 1 hoặc nhiều sự kiện.

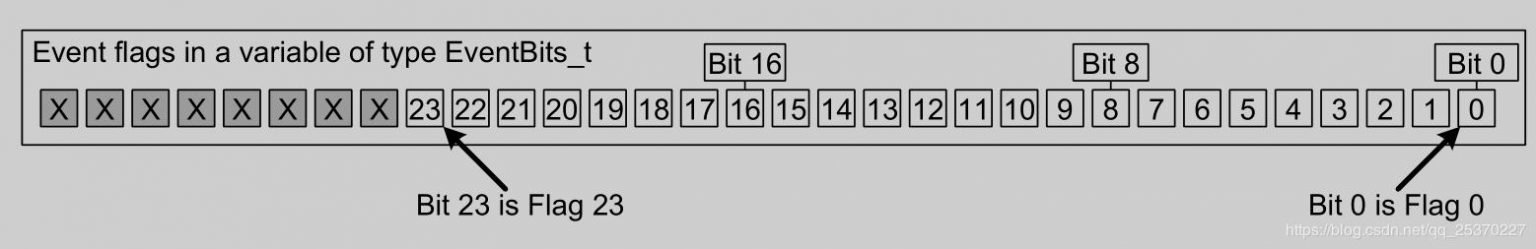


## Tại sao nên dùng event group:

Như các bạn đã biết thì các ứng dụng về lập trình nhúng đòi hỏi người lập trình phải biết tối ưu về bộ nhớ. Bằng cách sử dụng event group,       chúng ta có thể giảm thiểu việc sử dụng RAM. Bởi thay vì sử dung nhiều Bin-Sema, chúng ta chỉ cần thay thế chúng bằng một event group duy nhất trong nhiều trường hợp.

## Nguyên lý hoạt động:

Event group về có bản là một tập hợp các event flag mang giá trị 0 hoặc 1. Những giá trị này thể hiện sự có hay không của 1 sự kiện nào đó.



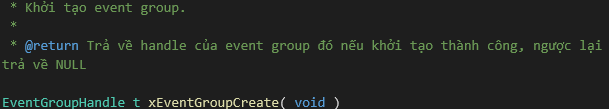
Nhờ đó thì ta có thể lưu trạng thái của 1 event flag chỉ bằng 1 bit duy nhất, và các bit đó được lưu giữ trong một biến gọi là event group. Có tối đa 24 event flag tương ứng với 24 event trong 1 event group. Được cấu hình qua configUSE\_16\_BIT\_TICKS

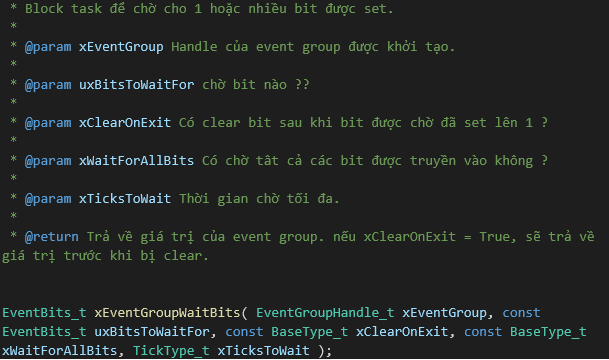
·       If configUSE\_16\_BIT\_TICKS = 1: 8 flags.

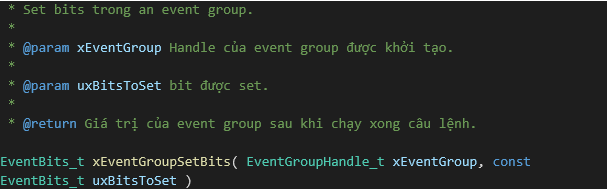
·       If configUSE\_16\_BIT\_TICKS = 0: 24 flags.

Kiểu dữ liệu **EventBits\_t** được dùng để chứa các event bits trong 1 event group.

## Các thao tác với event group.







# RTOS Ngoại Truyện: Delay vs vTaskDelay vs vTaskDelayUntil

## **Tổng quan:**

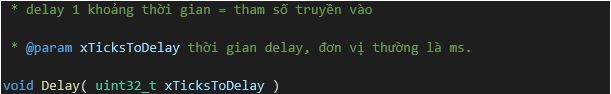
**Delay**: là loại Delay được sử dụng phổ biến, không thuộc hệ điều hành nào cả. Tùy vào thư viện thì nó sẽ có API khác nhau ( HAL\_Delay() trong STM32 thư viện HAL, hay delay() trong Arduino). Khi dùng hàm này trong RTOS sẽ không đưa task vào trạng thái block

**vTaskDelay**: Hàm delay thuộc hệ điều hành RTOS, khi thực hiện hàm này sẽ đưa task vào trạng thái block để có thể thực hiện task các.

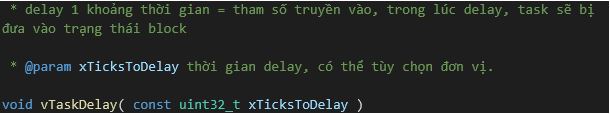
**vTaskDelayUntil**: Giống như vTaskDelay. Nhưng hoạt động dựa trên timeline Tick của hệ điều hành. (mỗi Tick = 1ms).

## **Cấu trúc câu lệnh:**

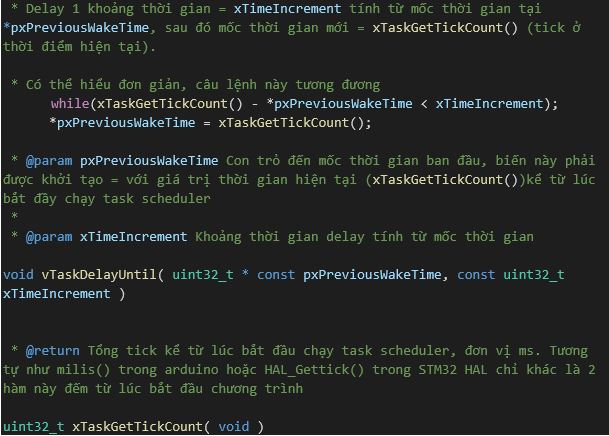
**Delay**



**vTaskDelay**



**vTaskDelayUntil**



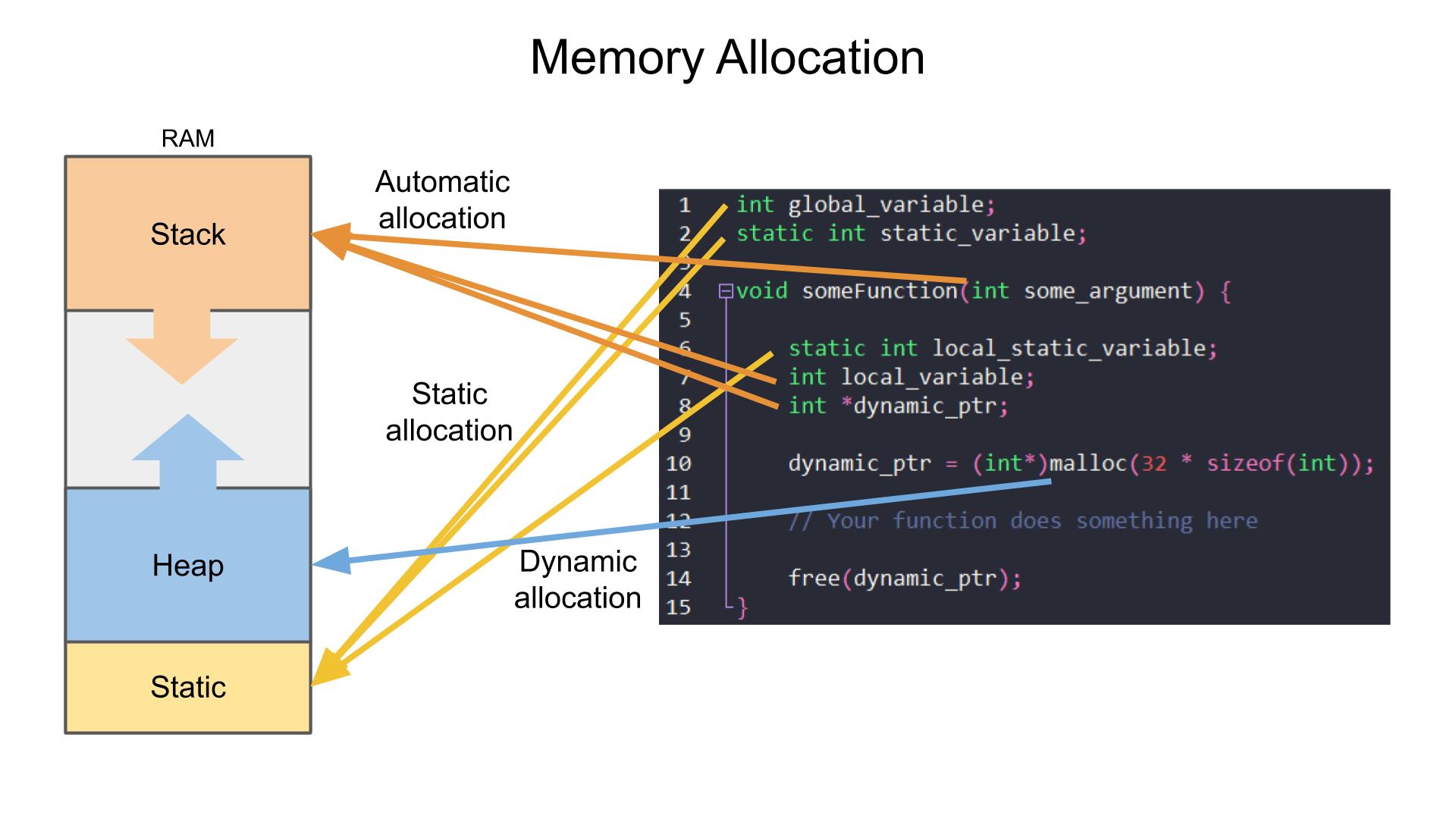
# Ngoại Truyện 2: Memory Management

Phần kiến thức quan trọng trước khi bắt đầu làm quen với hệ điều hành RTOS

Khi làm việc với hệ điều hành RTOS, ta cần cung cấp 1 vùng nhớ cố định khi khởi tạo 1 task, chính vì thế việc hiểu cách phân vùng bộ nhớ trong RTOS khá là quan trọng, không thể cứ cấp phát bừa bãi được, như thế khá là tốn tài nguyên khi làm dự án lớn.

## NON-RTOS

Volatile memory ( như RAM). Trong phần lớn hệ thống vi điều khiển được chia làm 3 vùng: static, stack, và heap.



Static memory được dùng để lưu trữ các biến toàn cục và các biến được khai báo “static”.

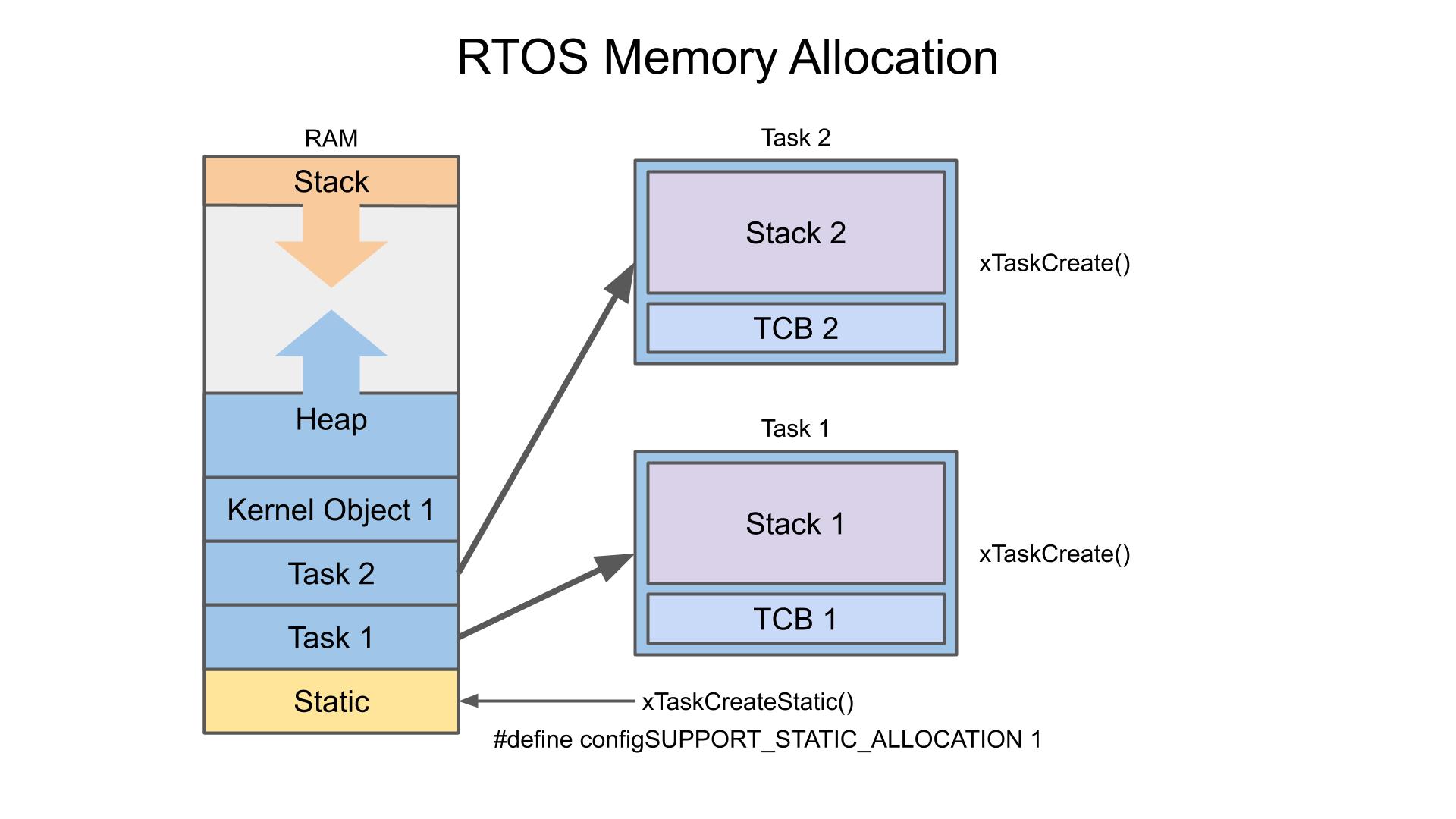
Stack memory là nơi lưu trữ các biến cục bộ, được tổ chức dưới dạng (LIFO) last-in-first-out để các biến của 1 hàm có thể được đẩy vào khi hàm đó được gọi.

Heap memory thì phải được phân bổ rõ ràng bởi lâp trình viên. Thông thường thì bạn hay sử dụng hàm MALLOC() để phẩn bổ heap cho các biến, vv.v. Hay còn được hiểu là cấp phát động. Và lưu ý, khi biến đó không còn được sử dụng ta nên giải phóng vùng nhớ đã cấp phát cho nó.

Một lưu ý nữa, vì Stack và Heap không có “ranh giới rạch ròi” nên dữ liệu ở 2 vùng nhớ này có thể ghi đè lên nhau, như các bạn thấy ở trên hình, thằng heap thì “đi từ dưới lên”, thằng stack thì lại “đi từ trên xuống” nên nếu không có sự kiểm soát thì sớm hay muộn cả 2 cũng sẽ “gặp nhau” và “đè lên nhau”.

## RTOS

Khi bạn khởi tạo 1 task vụ trong RTOS (ví dụ xTaskCreate() hoặc osThreadCreate() ), thì hệ điều hành sẽ phân bổ 1 phần bộ nhớ heap cho task đó.



Một phần trong đó được gọi là Task control block (TCB). Là nơi lưu trữ các thông tin của task như mức ưu tiên, con trỏ đại diện,…

Phần còn lại là local stack, hoạt động như global stack (nhưng quy mô nhỏ hơn). Các biến cục bộ được khởi tạo trung suốt quá trình gọi hàm trong task sẽ được lưu trữ ở đây. Do đó ta cần tính toán được kích thước stack và truyền vào tham số stack size khi khởi tạo task.

Trong RTOS, không nên sử dụng malloc() và free() mà nên thay thế bằng pvPortMalloc() và vPortFree() để thay thế, vì nó sẽ an toàn hơn trong hệ điều hành. Khi sử dụng chúng, thì bộ nhớ cấp phát sẽ nằm trong vùng heap của toàn hệ thống thay vì vùng heap được cấp phát riếng cho task.

Đặc biệt, ở trong các phiên bản gần đây của thư viện RTOS, thì nó cho phép khởi tạo các task static (xTaskCreateStatic() ). Điều này giúp cho chúng ta rất nhiều khi mà ta không thể hoặc khổng muốn sử dụng bộ nhớ heap nữa để tránh tràn heap.

Contents

[1\_GIỚI THIỆU 1](#_Toc95729457)

[TASK STATE: 1](#_Toc95729458)

[TASK STRUCTURE: 2](#_Toc95729459)

[TASK PROPERTIES: 2](#_Toc95729460)

[TASK SCHEDULING: 2](#_Toc95729461)

[3\_Interrupt 5](#_Toc95729462)

[4\_Queue  các task trao đổi dữ liệu với nhau 5](#_Toc95729463)

[**Khái niệm** 6](#_Toc95729464)

[Phụ lục: Mail Queue 7](#_Toc95729465)

[**Các thao tác đơn giản với Mail Queue** 7](#_Toc95729466)

[5\_Semaphore 8](#_Toc95729467)

[Khái niệm: 9](#_Toc95729468)

[**Nguyên lý hoạt động:** 9](#_Toc95729469)

[Phân loại 9](#_Toc95729470)

[Các thao tác cơ bản với Sermaphore 10](#_Toc95729471)

[Tổng quan 11](#_Toc95729472)

[Nguyên lý hoạt động. 12](#_Toc95729473)

[Một số điểm khác so với Semaphore. 12](#_Toc95729474)

[Các thao tác cơ bản với Mutex. 13](#_Toc95729475)

[7\_Software Timer 13](#_Toc95729476)

[Tổng quan 13](#_Toc95729477)

[**Phân loại** 14](#_Toc95729478)

[Các thao tác cơ bản với Software Timer 14](#_Toc95729479)

[8\_Event Groups 16](#_Toc95729480)

[Tổng quan: 16](#_Toc95729481)

[Tại sao nên dùng event group: 16](#_Toc95729482)

[Nguyên lý hoạt động: 17](#_Toc95729483)

[Các thao tác với event group. 17](#_Toc95729484)

[RTOS Ngoại Truyện: Delay vs vTaskDelay vs vTaskDelayUntil 18](#_Toc95729485)

[**Tổng quan:** 18](#_Toc95729486)

[**Cấu trúc câu lệnh:** 18](#_Toc95729487)

[Ngoại Truyện 2: Memory Management 19](#_Toc95729488)

[NON-RTOS 20](#_Toc95729489)

[RTOS 20](#_Toc95729490)