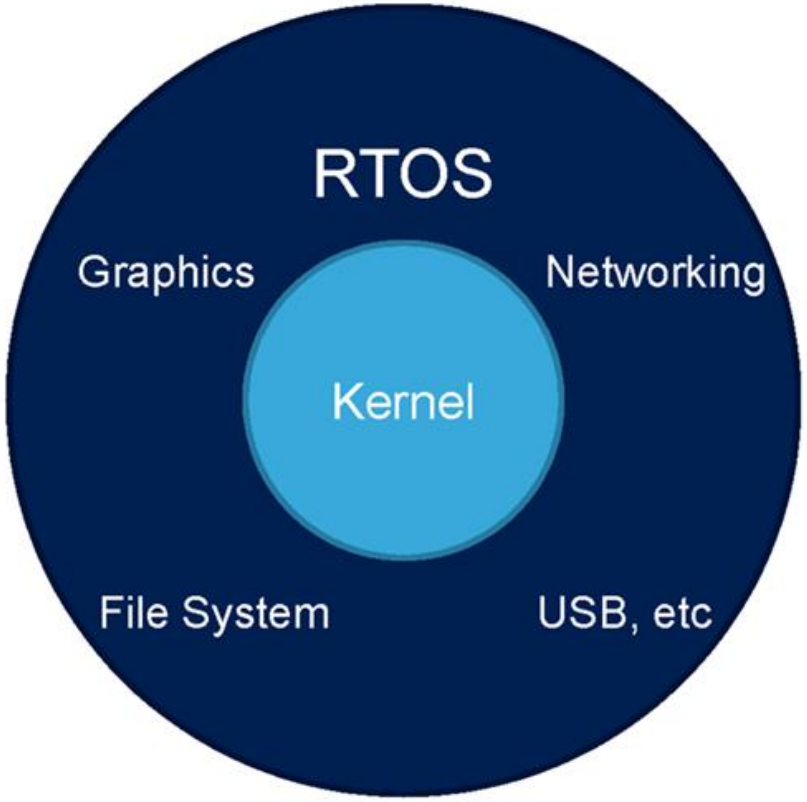
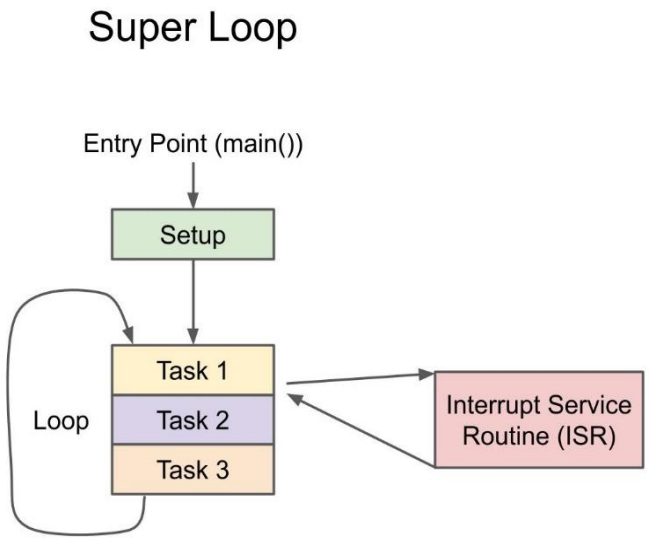
FREERTOS với ESP32

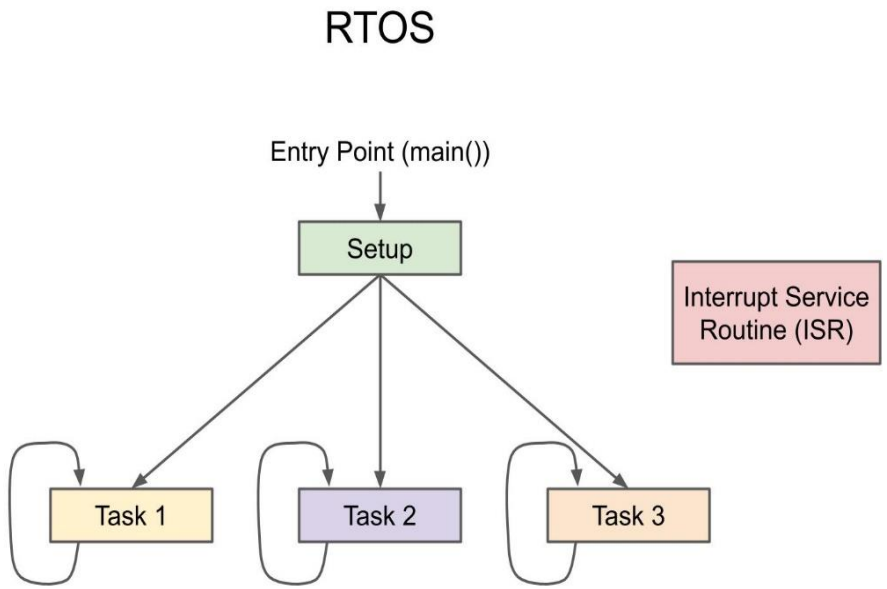
Khái quát về RTOS

- RTOS (Real-Time Operating System) hay được gọi là hệ điều hành thời gian thực cho phép ứng dụng chạy đa tác vụ.
- Có nhiều hệ điều hành cho hệ thống nhúng như:
  - RT Thread
  - FreeRTOS
  - µC/OS
  - CMSIS



Tại sao lại sử dụng RTOS





Trong vòng lặp, thực hiện nhiều tác vụ định kỳ bằng cách bắt chúng "xếp hàng" – thực hiện tuần tự.

Trong RTOS, mọi tác vụ được thực hiện gần như đồng thời.

Task API

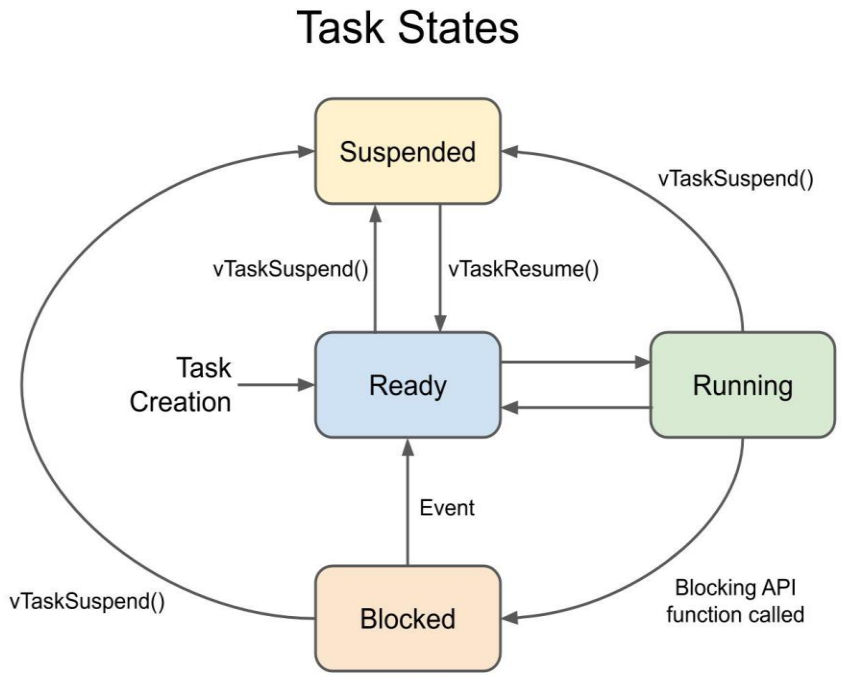
Hiểu đơn giản task là công việc cần phải xử lý. Trong FreeRTOS, ta có thể coi task giống như thread.
Thông qua hàm **`xTaskCreate()`** ta có thể tạo ra 1 task:

```
BaseType_t xTaskCreate (TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        const uint32_t usStackDepth,
                        void * const pvParameters,
                            UBaseType_t uxPriority,
                            TaskHandle_t * const pvCreatedTask)
```

- **`pvTaskCode:`** tên Task (tên hàm) muốn chạy
- **`*pcName:`** tên do người dùng đặt cho cái task đó.
- **`usStackDepth:`** size hay gọi là Depth của Task, đơn vị của nó là 4 bytes.
- **`*pvParameters:`** 1 con trỏ dùng để chứa các tham số của Task khi task còn hoạt động, thường để NULL.
- **`uxPriority:`** độ ưu tiên của task này, số càng lớn thì độ ưu tiên càng lớn.
- **`*pxCreatedTask:`** 1 con trỏ đại diện Task, dùng để điều khiển task từ 1 task khác, ví dụ xóa task này từ 1 task đang chạy khác.

Sau khi tạo task, ta gọi hàm `vTaskStartScheduler()` dùng để yêu cầu nhân (kernel) của RTOS bắt đầu chạy. Ta sẽ xóa task thông qua hàm `xTaskDelete()`

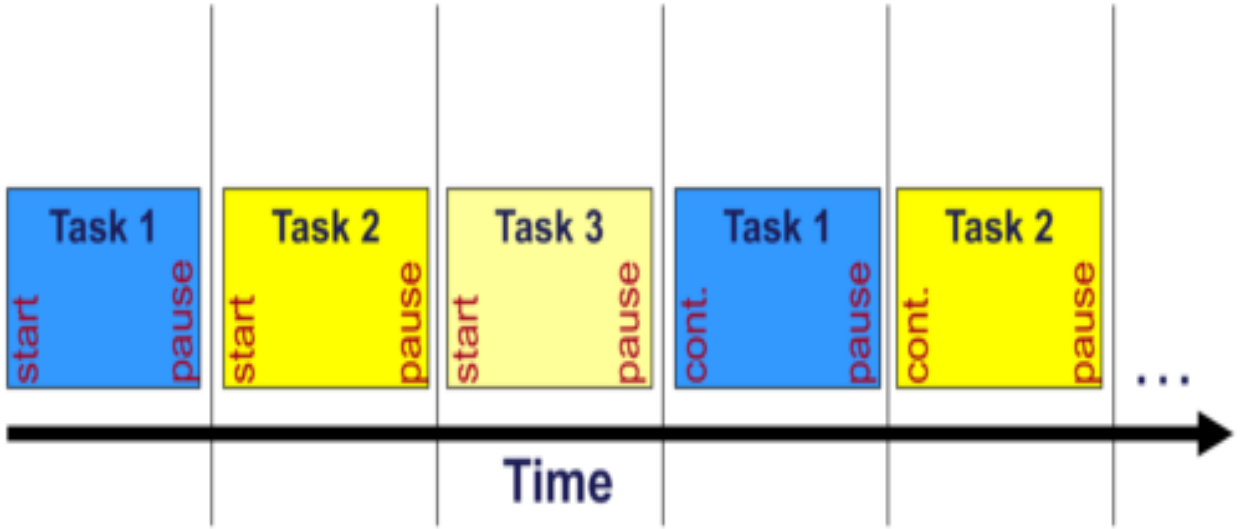
Task state



- **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.

Round Robin Scheduling

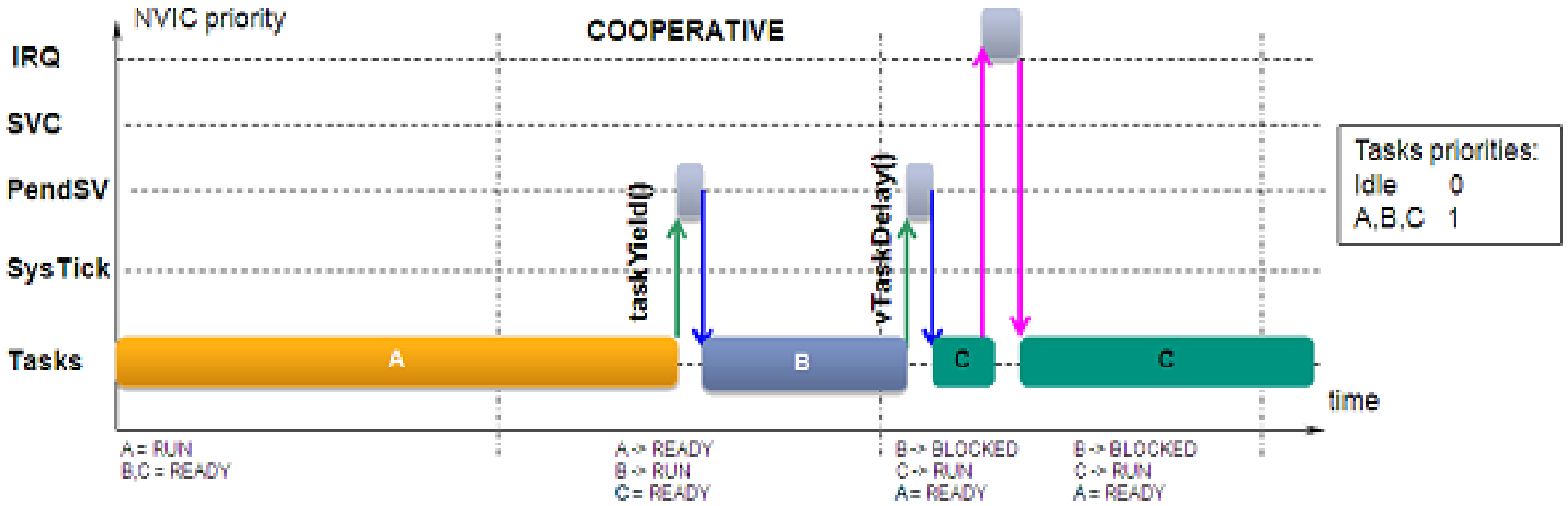
Round Robin Scheduling: Mỗi task được chia cho một khe thời gian cố định, nếu trong khoảng thời gian được chia đó mà task chưa thực hiện xong thì sẽ bị tạm dừng, chờ đến lượt tiếp theo để thực hiện tiếp công việc sau khi hệ thống xử lý hết một lượt các task.



Round Robin Scheduling

Co-operative Scheduling

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

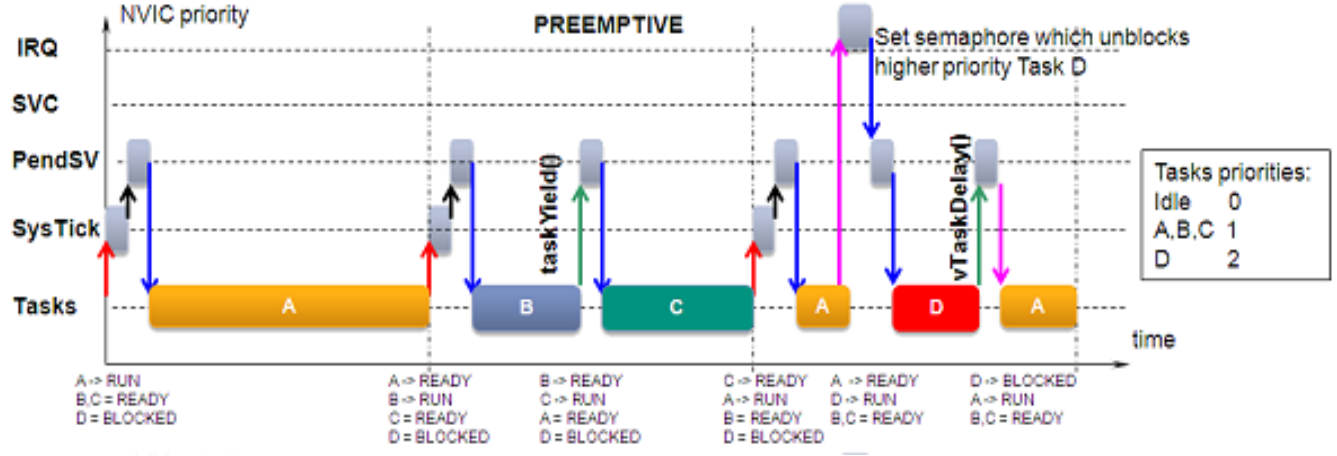


Co-operative Scheduling

Preemptive Scheduling

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 mức 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
- **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.



Preemtive Scheduling

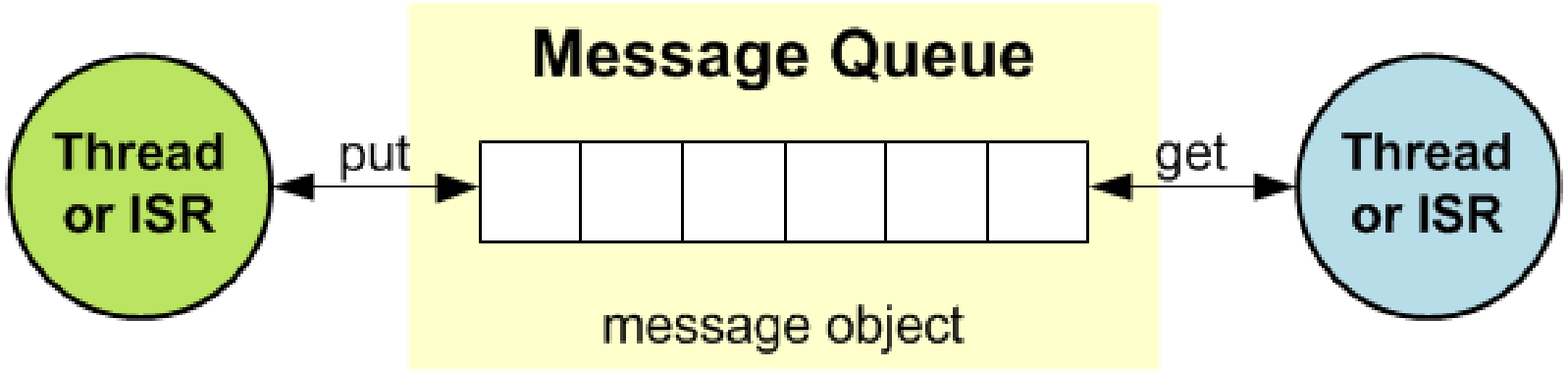
Task interrupt (ISR)

Đây là ứng dụng việc xử lý các trạng thái của task để thực hiện task ngắt (ISR). Ví dụ, ta sẽ tạo 1 hàm thay đổi trạng thái led trên board bằng nút boot trên board. Trong phần này, ta chỉ quan tâm đến khởi tạo task ISR.

- B1: Tạo 1 `TaskHandle_t ISR1 = NULL;`
- B2: viết 1 hàm xử lý nút bấm, ví dụ `void button_task(void *arg).` Trong hàm này ta sẽ gọi hàm `vTaskSuspend(ISR1);`
- B3: Viết 1 hàm khác dùng để gọi sự kiện ngắt thông qua 1 con trỏ đại diện task button `xTaskResumeFromISR(ISR1);`
- B4: Setup các chân, cạnh ngắt và khởi tạo 1 task `button_task` xử lý nút bấm trong hàm main

Queue

Queue là một cấu trúc dữ liệu dùng để chứa các đối tượng làm việc theo cơ chế **FIFO** (*First In First Out*) - "vào trước ra trước“. Các đối tượng có thể được thêm vào hàng đợi bất kỳ lúc nào, nhưng chỉ có đối tượng thêm vào đầu tiên mới được phép lấy ra khỏi hàng đợi. Thao tác thêm vào và lấy một đối tượng ra khỏi hàng đợi được gọi lần lượt là "enqueue" và "dequeue". Việc thêm một đối tượng luôn diễn ra ở cuối hàng đợi và một phần tử luôn được lấy ra từ đầu hàng đợi.



Queue API

Tương tự như task, FreeRTOS cũng có API để lập trình với queue:

- Hàm khởi tạo 1 queue với 2 tham số là độ dài queue và kích thước của kiểu dữ liệu nhận vào

```
QueueHandle_t xQueueCreate( uint32_t uxQueueLength,
                            uint32_t uxItemSize );
```

- Hàm gửi dữ liệu vào queue, truyền 3 tham số lần lượt là queue, con trỏ tới dữ liệu muốn truyền vào queue và thời gian đợi

```
uint32_t xQueueSend(QueueHandle_t xQueue,
                    const void * pvItemToQueue,
                    TickType_t xTicksToWait);
```

- Hàm đọc dữ liệu từ queue để ghi vào 1 bộ đệm, 3 tham số lần lượt là queue, con trỏ tới bộ đệm và thời gian đợi

```
uint32_t xQueueReceive(QueueHandle_t xQueue,
                       void *pvBuffer,
                       TickType_t xTicksToWait);
```

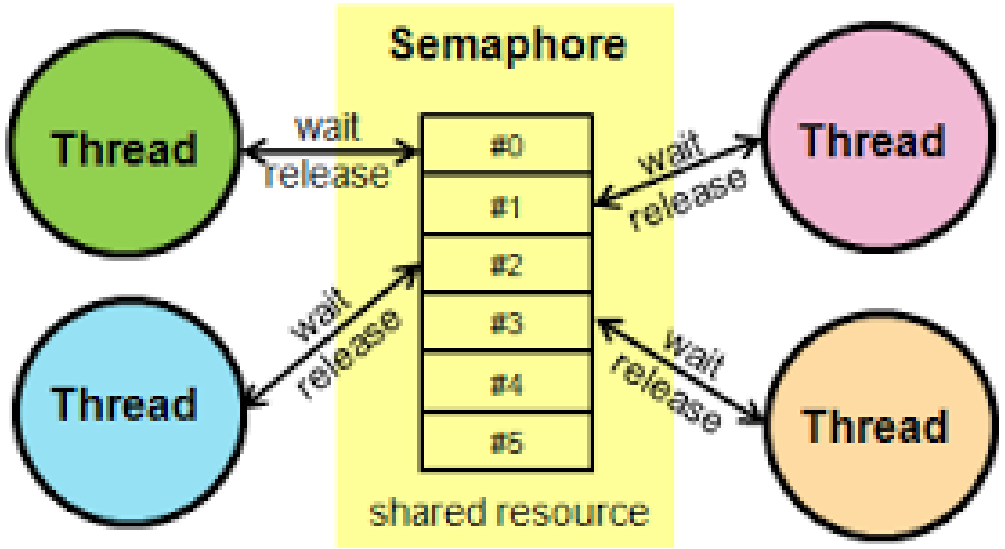
xQueueSendFromISR

Đây là ứng dụng việc kết hợp gửi vào queue khi có ngắt (ISR), ta sẽ ví dụ là khi bấm nút trên board thì sẽ gửi 1 chuỗi vào queue ngay lập tức

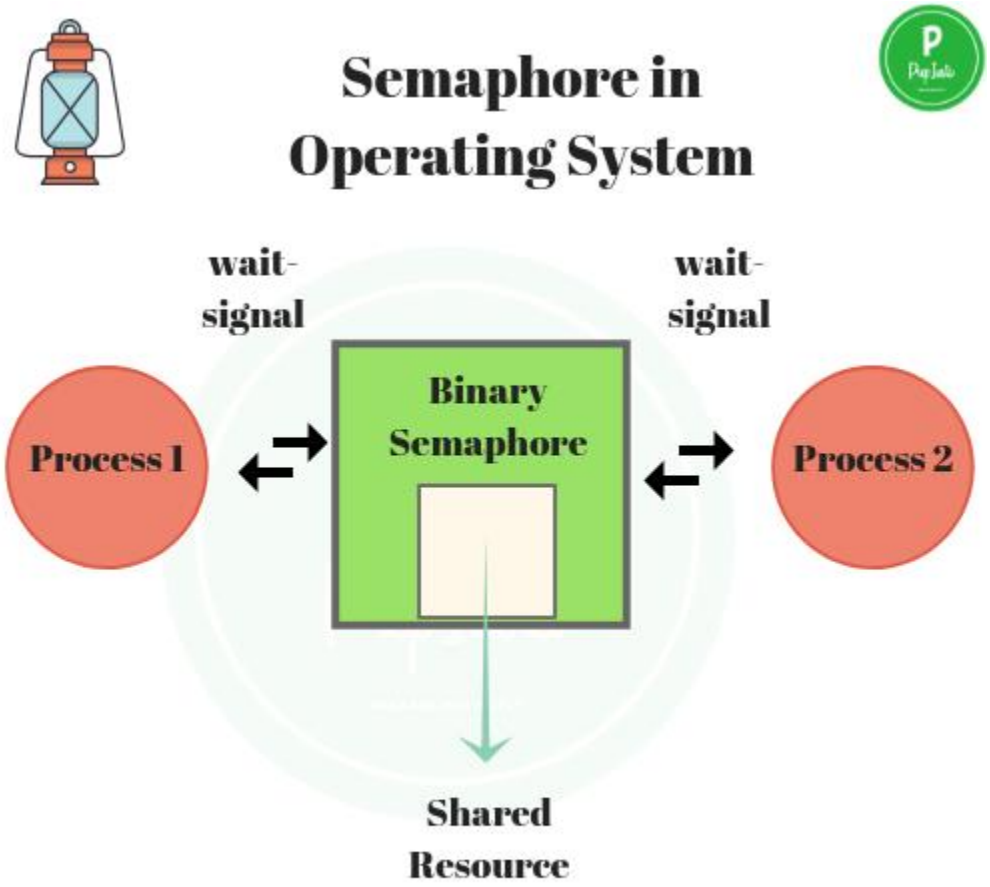
- B1: Tạo các tác vụ ngắt như trong ví dụ task ISR
- B2: Viết 1 hàm dùng để gọi sự kiện ngắt `void IRAM_ATTR button_isr_handler(void *arg).` Trong hàm này ta sẽ gọi `xQueueSendFromISR()` để gửi dữ liệu vào queue mỗi khi có ngắt.
- B3: Khởi tạo các task trong main: ngắt và truyền nhận dữ liệu qua queue

Semaphore

Semaphore cũng là 1 cơ chế của RTOS để quản lý các task. Semaphore giống như 1 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 cho vào queue trước thì sử dụng tài nguyên trước, sau đó nó lại release ra cho thread/task khác dùng. Có thể coi Semaphore như 1 chiếc chìa khóa cho task.

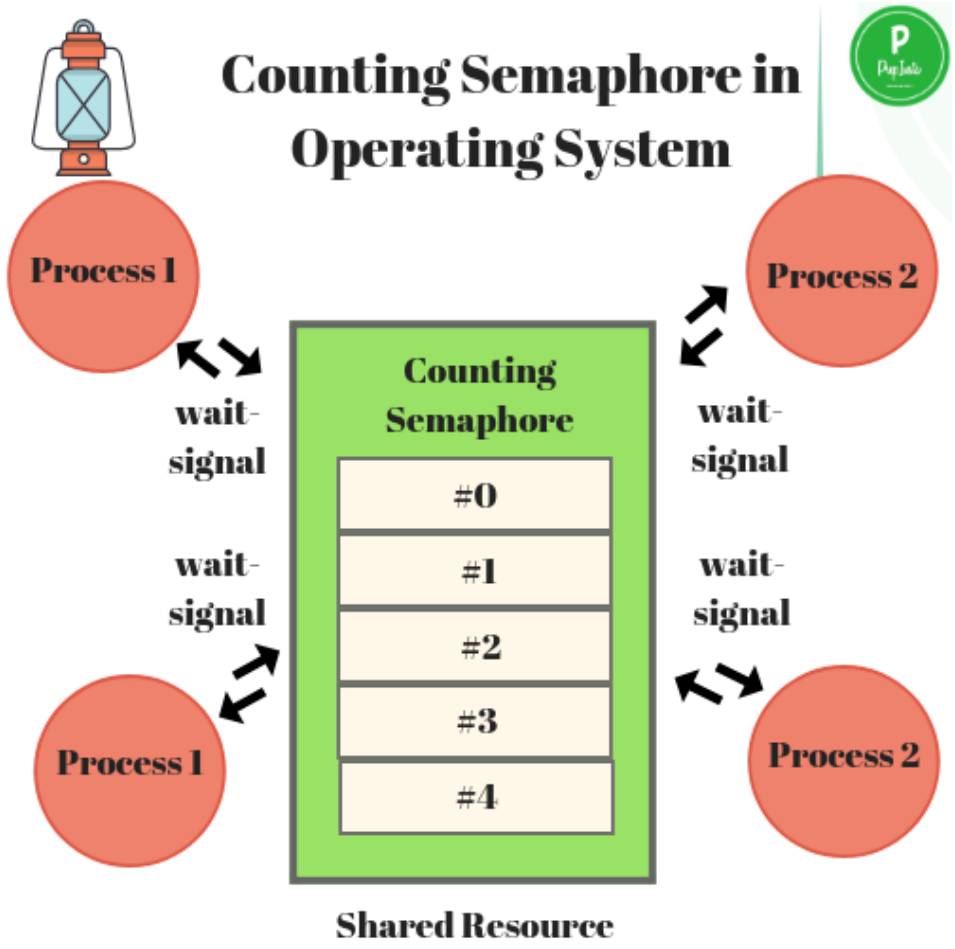


Binary Semaphore



- Binary semaphore có duy nhất 1 token và nó chính là chìa khóa để sử dụng nguồn tài nguyên chung
- 1 Task sẽ nắm giữ chìa khóa, khi nào nó dùng xong tài nguyên thì sẽ đẩy ra 1 thông báo cho task nào muốn sử dụng thông qua `xSemaphoreGive()`
- Task nào muốn sử dụng tài nguyên chung thì phải bắt được chìa khóa, tức là nhận được thông báo từ task vừa give thông qua hàm `xSemaphoreTake()`
- Như vậy, Binary semaphore được sử dụng như là 1 thông báo từ task này sang task kia. Đây chính là sự khác biệt cơ bản của Semaphore so với mutex.

Counting Semaphore



- Cơ chế hoạt động của nó cũng giống như binary nhưng khác ở chỗ thay vì chìa khóa chỉ có 2 giá trị là true hoặc false thì ở Counting Semaphore
- Mỗi khi 1 task thêm 1 cái gì đó vào vùng đệm thì nó sẽ tăng giá trị của semaphore lên 1. Khi tăng đến giá trị max (có thể cấu hình được) thì dừng lại.
- Mỗi khi 1 task lấy được chìa khóa thì nó sẽ trừ giá trị của semaphore đi, khi đến giá trị 0 thì các task sẽ không thể truy cập vào tài nguyên chung được.

Như vậy có thể thấy Counting Semaphore sẽ quy định số lượng task được truy cập vào tài nguyên chung

Semaphore API

Các hàm cơ bản thường dùng với Semaphore:

- Khởi tạo 1 binary semaphore

```
SemaphoreHandle_t xSemaphore = xSemaphoreCreateBinary();
```

- Khởi tạo 1 couting semaphore với 2 tham số truyền vào là số lượng max và số khởi đầu

```
xSemaphoreCreateCounting( uxMaxCount, uxInitialCount )
```

- Trả “chìa khóa “ với tham số truyền vào là Semaphore đã khởi tạo

```
xSemaphoreGive(SemaphoreHandle_t xSemaphore)
```

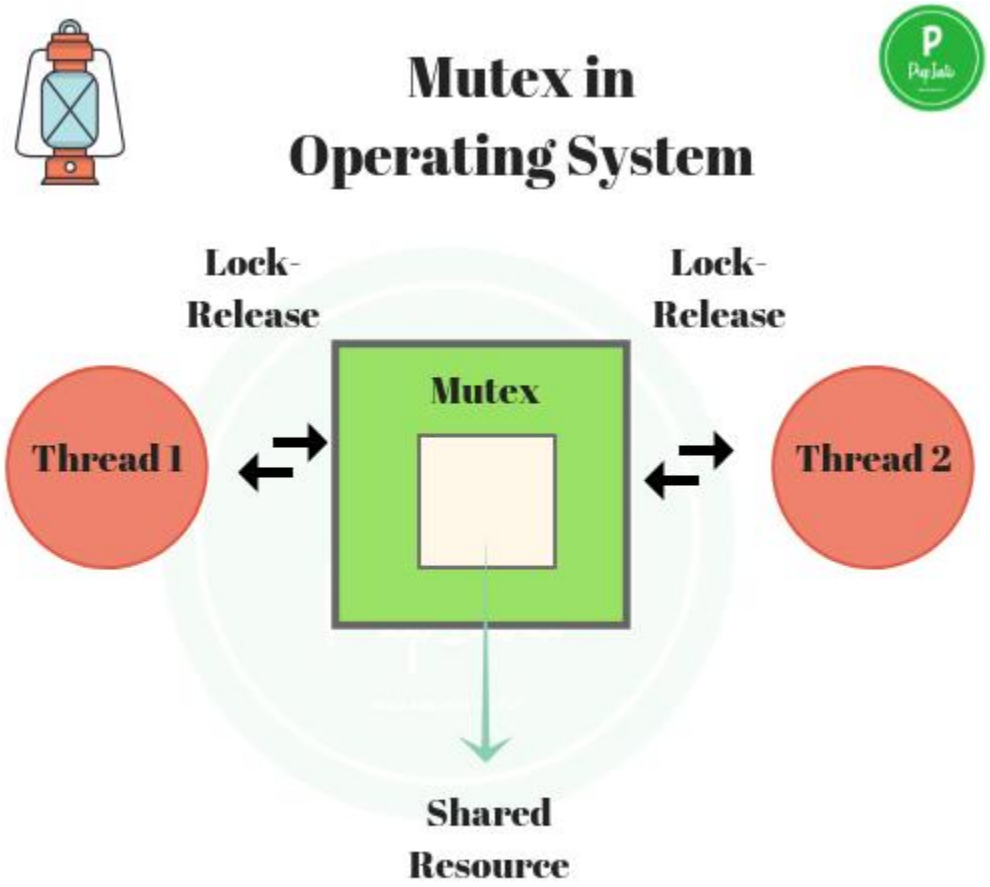
- Lấy “chìa khóa “ với tham số truyền vào là Semaphore đã khởi tạo

```
xSemaphoreTake(SemaphoreHandle_t xSemaphore)
```

- Trả “chìa khóa “ qua ISR, tham số truyền vào là Semaphore đã khởi tạo và

```
xSemaphoreGiveFromISR( xSemaphore, pxHigherPriorityTaskWoken )
```

Mutex

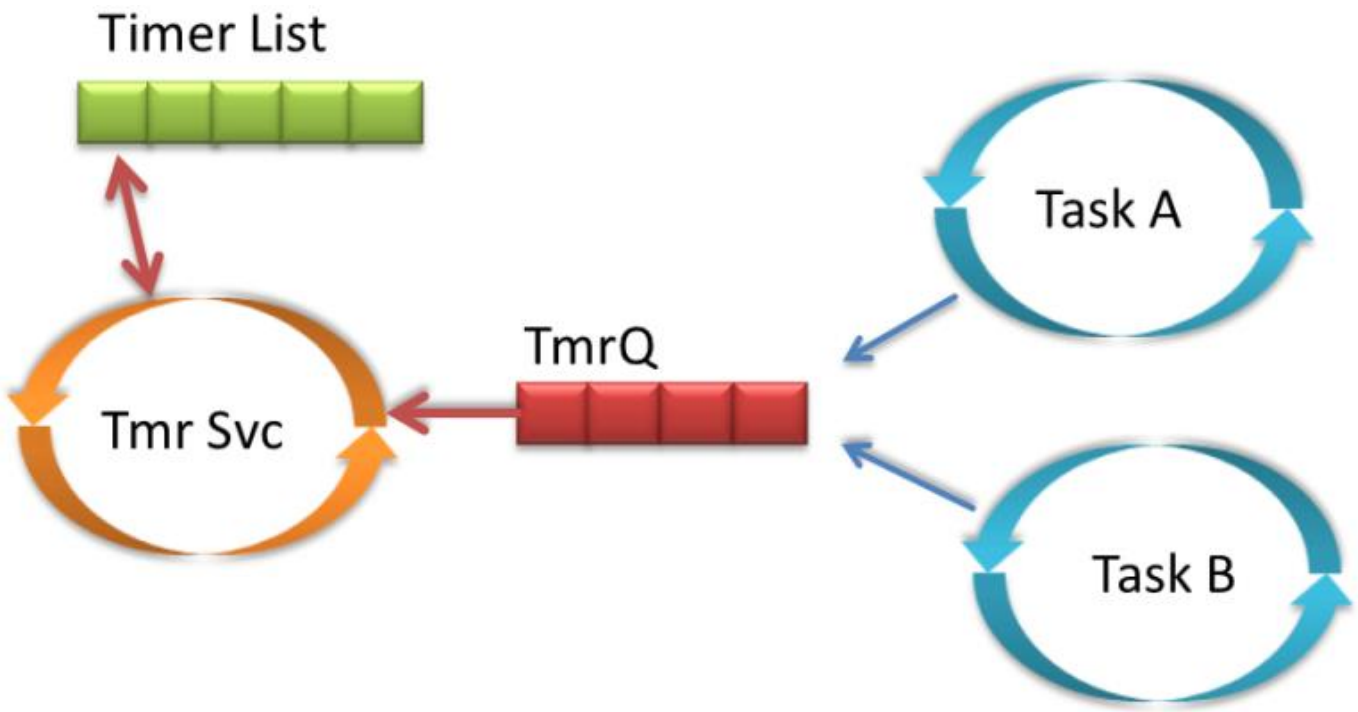


- Về cơ bản thì MUTEX tương tự như Binary Semaphore nhưng có tích thêm cơ chế "kế thừa mức ưu tiên" và được dùng cho mục đích hạn chế quyền truy cập vào tài nguyên của các task khác chứ không phải là để đồng bộ như Semaphore.
- Khi 1 task muốn truy cập vào tài nguyên chung để thực thi nhiệm vụ thì sẽ 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.
- Điểm khác biệt so với Semaphore là ở chỗ ở Mutex thì task nào giữ Mutex mới được give, còn Semaphore thì mọi task đều có thể give. Chính vì vậy trình phục vụ ngắt có thể give Semaphore nhưng đối với Mutex thì không.

Software Timer

- Software timer cho phép chúng ta có thể thực thi 1 tác vụ nào đó trong khoảng thời gian được định trước. Hàm được thực thi bởi timer sẽ được gọi bởi hàm timer callback (tương đương với cơ chế ngắt của timer thông thường)

- Thời gian giữa lúc timer start và callback function của nó được thực thi được gọi là thời gian của timer (timer's period).



Software Timer API

- Khởi tạo 1 Software timer với các tham số lần lượt là tên timer, period, trạng thái có autoreload không, ID của timer và hàm callback

```
TimerHandle_t xTimerCreate(const char *const pcTimerName, const TickType_t xTimerPeriodInTicks, const UBaseType_t uxAutoReload, void *constpvTimerID, TimerCallbackFunction_t pxCallbackFunction)
```

- Lấy ID của soft timer đang chạy :

```
void *pvTimerGetTimerID (const TimerHandle_t xTimer)
```

- Bắt đầu cho timer chạy:

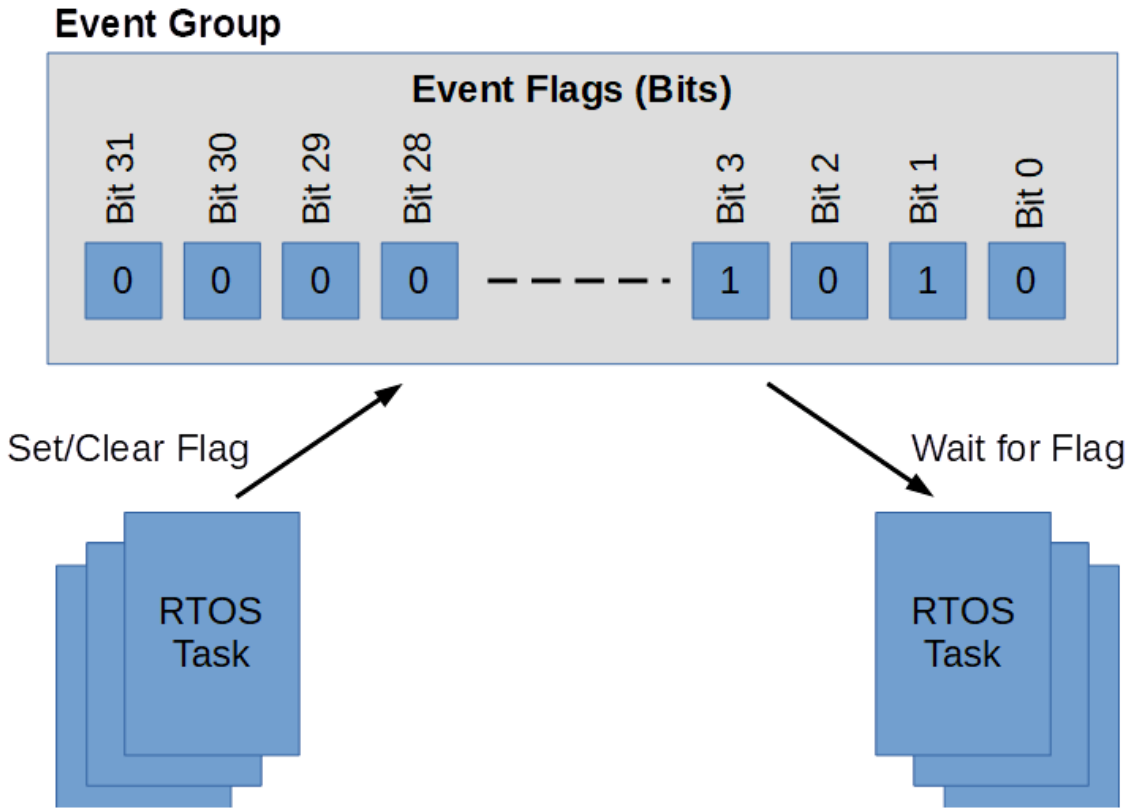
```
xTimerStart( xTimer, xTicksToWait )
```

- Dừng timer:

```
xTimerStop( xTimer, xTicksToWait )
```

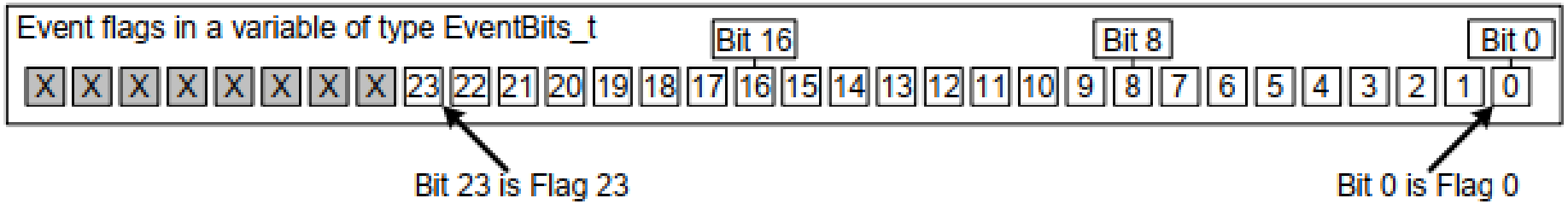
Event Group

- Event Group cũng như Semaphore hoặc Mutex với việc đưa các task vào hàng đợi trong trạng thái block, và unblock chúng khi có sự kiện xảy ra (giveSemaphore hoặc give). Nó có thể được sử dụng để đồng bộ hóa nhiều tác vụ bằng cách sử dụng một sự kiện duy nhất hoặc nhiều sự kiện.
- Để sử dụng Event groups, ta cần quan tâm đến **EVENT FLAG** và **EVENT BITS.**



Event Flag và Event Bits

- Về cơ bản EVENT GROUP là một tập hợp các EVENT FLAG. EVENT FLAG là một giá trị bool với chỉ ‘0’ hoặc ‘1’. Giá trị này mô tả sự hiện hữu hay không của một sự kiện và được dùng tối đa 24 bit trong 1 EVENTGROUP. Kiểu dữ liệu trả về được định nghĩa là EventBits_t.
- Chúng ta có thể định nghĩa số lượng flag bit trong 1 EVENT GROUP bằng cách sử dụng hằng số cấu hình **configUSE_16_BIT_TICKS.**
  - Nếu **configUSE_16_BIT_TICKS = 1**, nhóm sự kiện chứa 8 flag.
  - Nếu **configUSE_16_BIT_TICKS = 0**, nhóm sự kiện bao gồm 24 flag.



Event Group API

- Khởi tạo 1 Event Group

```
EventGroupHandle_t xEventGroupCreate( void )
```

- Đợi một hoặc nhiều bit được đặt trong nhóm sự kiện đã tạo trước đó

```
EventBits_t xEventGroupWaitBits( EventGroupHandle_t xEventGroup,
const EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit,
const BaseType_t xWaitForAllBits, TickType_t xTicksToWait )
```

- Xóa các bit trong một nhóm sự kiện. Hàm này không thể được gọi từ một ngắt. Có 1 phiên bản khác có thể gọi từ ngắt là `xEventGroupClearBitsFromISR()`

```
EventBits_t xEventGroupClearBits( EventGroupHandle_t xEventGroup,
                                  const EventBits_t uxBitsToClear )
```

- Đặt các bit trong một nhóm sự kiện. Hàm này không thể được gọi từ một ngắt. Đặt các bit trong một nhóm sự kiện sẽ tự động bỏ chặn các tác vụ bị chặn đang chờ các bit. Có 1 phiên bản khác có thể gọi từ ngắt là `xEventGroupSetBitsFromISR()`

```
void xEventGroupSetBits( pvEventGroup, ( EventBits_t ) ulBitsToSet )
```

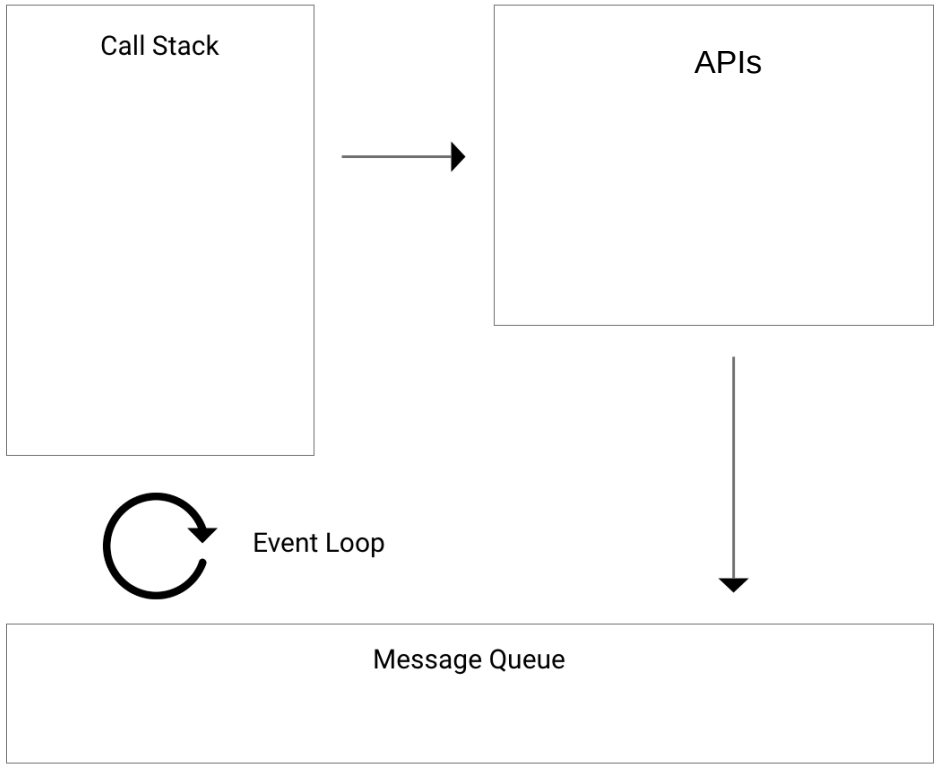
- Xóa Event Group:

```
void vEventGroupDelete( EventGroupHandle_t xEventGroup )
```

Event Loop

Đây là 1 cơ chế rất hay của ESP32. Event loop là một vòng lặp vô hạn dùng để “lắng nghe” các sự kiện. Nhiệm vụ của Event loop rất đơn giản đó là đọc Stack và Event Queue.

Đầu tiên nó sẽ nạp các sự kiện vào 1 Event Queue, sau đó vòng loop sẽ chạy và nạp các sự kiện trong queue vào Stack để gọi. Nếu nhận thấy Stack rỗng nó sẽ nhặt Event đầu tiên trong Event Queue và Handler (callback) gắn với event đó và đẩy vào Stack



Event Loop API

User Event Loops	Default Event Loops	Tác dụng
`esp_event_loop_create()`	`esp_event_loop_create_default()`	Tạo event loop
`esp_event_loop_delete()`	`esp_event_loop_delete_default()`	Xóa event loop
`esp_event_handler_register_with()`	`esp_event_handler_register()`	Đăng ký sự kiện trong event loop
`esp_event_handler_unregister_with()`	`esp_event_handler_unregister()`	Hủy đăng ký sự kiện trong event loop
`esp_event_post_to()`	`esp_event_post()`	Post sự kiện vào event loop