# 2. RTOS introduction

**Hệ điều hành (OS)** là phần mềm cơ bản và quan trọng nhất trong một hệ thống máy tính. Nó giúp điều phối và quản lý tài nguyên của máy tính, bao gồm bộ vi xử lý (CPU), bộ nhớ, thiết bị ngoại vi (như ổ cứng, màn hình, bàn phím), và các ứng dụng phần mềm khác.

Hệ điều hành là **cầu nối giữa phần cứng và phần mềm**, giúp người dùng và các ứng dụng giao tiếp với hệ thống máy tính một cách hiệu quả.

## 2.1.Real time application

**Real-Time System (Hệ thống thời gian thực)** là một hệ thống máy tính hoặc hệ thống điều khiển mà yêu cầu các tác vụ phải được thực hiện trong một thời gian cụ thể, hoặc đúng vào thời điểm đã định.

Đặc điểm quan trọng nhất của hệ thống thời gian thực là **thời gian phản hồi** (response time) phải đảm bảo và **dự đoán được**.

**Ứng dụng thời gian thực (Real-Time Application)** là những ứng dụng hoặc hệ thống yêu cầu xử lý và phản hồi với độ trễ rất thấp trong một khoảng thời gian cố định

A screenshot of a computer

AI-generated content may be incorrect.

## 2.2.Real Time Operating System

**RTOS** là một loại **hệ điều hành** được thiết kế đặc biệt để quản lý và xử lý các tác vụ trong một khoảng **thời gian cực kỳ chính xác**, với mục đích đảm bảo rằng các tác vụ quan trọng được thực hiện đúng thời gian quy định (deadline).

Để được xem là real time, 1 os phải:

* Đảm bảo thời gian phản hồi chính xác
* Quản lý interrupt và exception
* Quản lý đa nhiệm (critical session)
* Quản lý tài nguyên, bộ nhớ
* Cơ chế lập lịch(scheduler)

## 2.3 RTOS vs GPOS

- GPOS(genreral purpose OS) – hệ điều hành tổng quát

| **Tiêu chí** | **GPOS (General Purpose OS)** | **RTOS (Real-Time OS)** |
| --- | --- | --- |
| **Mục đích** | Hệ điều hành tổng quát | Hệ điều hành thời gian thực |
| **Quản lý thời gian** | Không đảm bảo thời gian thực | Đảm bảo phản hồi thời gian thực |
| **Quản lý đa nhiệm** | Hỗ trợ đa nhiệm, không có độ trễ thấp | Hỗ trợ đa nhiệm với độ trễ thấp |
| **Độ trễ** | Độ trễ không thể dự đoán | Độ trễ thấp và có thể dự đoán được |
| **Quản lý tài nguyên** | Chia sẻ tài nguyên giữa các tiến trình | Quản lý tài nguyên nghiêm ngặt |
| **Ứng dụng** | Máy tính cá nhân, máy chủ, di động | Điều khiển tự động, hệ thống nhúng, y tế, hàng không |
| **Ví dụ** | Windows, Linux, MacOS | VxWorks, FreeRTOS, QNX |

A diagram of a computer

AI-generated content may be incorrect.

Skip tầng CMSIS, gọi trực tiếp từ application xuống rtos kernel

FreeRTOS sử dụng Arm Cortex Mx processor’s internal systick timer làm time base(RTOS ticking)

=> nên sử dụng nguồn khác làm timebase cho stm32 cube HAL ( timer ngoại vi)

A screenshot of a computer

AI-generated content may be incorrect.

# 6. freeRTOS task creation

## 1. Task?

- Task là 1 công việc cần phải làm

- 1 ứng dụng có nhiều task khác nhau

-

A computer screen shot of a computer program

AI-generated content may be incorrect.

## 2.Task implementation

* Task handler thường được implemented trong 1 infinite loop
* Nếu task ra khỏi infinite loop, phải delete task trước khi kết thúc hàm

A screenshot of a computer program

AI-generated content may be incorrect.

## 3.Task creation

### 3.1.Các phương pháp tạo tác vụ trong RTOS:

**Dynamic** Allocation: bộ nhớ động, RTOS tự động cấp phát bộ nhớ cho **stack** và **TCB** (Task Control Block).

**Static** Allocation: user tự cấp phát bộ nhớ trước cho **stack** và **TCB** của tác vụ. Điều này giúp tránh các vấn đề phân mảnh bộ nhớ và không sử dụng bộ nhớ động.

| **Tiêu chí** | **Dynamic Task Creation** | **Static Task Creation** |
| --- | --- | --- |
| **Cấp phát bộ nhớ** | Bộ nhớ được cấp phát trong thời gian chạy (runtime). | Bộ nhớ được cấp phát trong compile time |
| **Linh hoạt** | Rất linh hoạt, có thể tạo và hủy tác vụ trong quá trình chạy. | Ít linh hoạt, số lượng tác vụ phải được xác định trước. |
| **Quản lý bộ nhớ** | Cần phải quản lý bộ nhớ động, dễ gặp vấn đề phân mảnh bộ nhớ. | Không có vấn đề phân mảnh bộ nhớ. |
| **Hiệu suất** | Có thể gây overhead do việc cấp phát bộ nhớ động. | Thực thi nhanh, không có overhead cấp phát bộ nhớ. |
| **Sử dụng bộ nhớ** | Tiết kiệm bộ nhớ vì chỉ cấp phát bộ nhớ khi cần thiết. | Tiêu tốn bộ nhớ vì bộ nhớ đã được cấp phát cho tất cả tác vụ. |
| **Ứng dụng** | Phù hợp với hệ thống có yêu cầu linh hoạt và thay đổi tác vụ trong thời gian thực. | Phù hợp với hệ thống có **số lượng tác vụ cố định** và không thay đổi. |
| **VD** |  | **OSEK** |

#### 3.1.1. tạo task với dynamic allocation

https://www.freertos.org/Documentation/02-Kernel/04-API-references/01-Task-creation/01-xTaskCreate

Để tạo một tác vụ trong FreeRTOS, bạn sử dụng hàm **xTaskCreate().** Hàm này cho phép bạn tạo một tác vụ, chỉ định một hàm xử lý cho tác vụ đó, và xác định các tham số quan trọng như kích thước bộ nhớ, ưu tiên và các đối số.

BaseType\_t xTaskCreate(

    TaskFunction\_t pxTaskCode,     // Hàm xử lý tác vụ

    const char \* const pcName,     // Tên của tác vụ

    uint16\_t usStackDepth,         // Kích thước ngăn xếp của tác vụ(đơn vị byte)

    void \*pvParameters,            // Tham số đầu vào cho tác vụ (có thể là NULL)

    UBaseType\_t uxPriority,        // Ưu tiên của tác vụ

    TaskHandle\_t \*pxCreatedTask    // Con trỏ đến task handle (có thể là NULL)

);

 **pxTaskCode**: Con trỏ đến hàm xử lý tac vụ. Hàm này phải có kiểu void và tham số đầu vào là void \*.

Ví dụ:

void vTaskFunction(void \*pvParameters)

{

    // Code cho tác vụ

    while(1) {

        // Công việc của tác vụ

    }

}

 **pcName**: Tên của tác vụ ->không ảnh hưởng đến hoạt động của hệ thống.

 **usStackDepth**: Kích thước của stack mà tác vụ sử dụng, tính bằng byte. Kích thước này phải đủ lớn để chứa tất cả các biến cục bộ và dữ liệu cần thiết cho tác vụ.

 **pvParameters**: Tham số đầu vào cho tác vụ. Đây là một con trỏ có thể được sử dụng để truyền các tham số vào hàm xử lý tác vụ khi nó bắt đầu. Nếu không cần tham số, bạn có thể truyền NULL.

 **uxPriority**: Ưu tiên của tác vụ. Tác vụ có mức ưu tiên cao hơn sẽ được thực thi trước các tác vụ có mức ưu tiên thấp hơn. FreeRTOS cho phép bạn thiết lập mức ưu tiên từ 0 (thấp nhất) đến configMAX\_PRIORITIES-1 (cao nhất).

 **pxCreatedTask**: Con trỏ đến một biến TaskHandle\_t, mà trong đó FreeRTOS sẽ lưu lại "task handle" của tác vụ mới tạo. Biến này có thể sử dụng để tham chiếu tác vụ sau này. Nếu không cần, bạn có thể truyền NULL.

Vd:

#include "FreeRTOS.h"

#include "task.h"

// Prototype các hàm tác vụ

void vTaskFunction1(void \*pvParameters);

void vTaskFunction2(void \*pvParameters);

int main(void)

{

    // Khởi tạo phần cứng ở đây

    // Tạo hai tác vụ

    xTaskCreate(vTaskFunction1, "Task 1", 100, NULL, 1, NULL);

    xTaskCreate(vTaskFunction2, "Task 2", 100, NULL, 1, NULL);

    // Bắt đầu bộ lập lịch của FreeRTOS

    vTaskStartScheduler();

    // Nếu mọi thứ hoạt động tốt, bộ lập lịch sẽ không trả về ở đây

    for( ;; );

}

void vTaskFunction1(void \*pvParameters)

{

    while(1)

    {

        // Mã của tác vụ 1

        // Ví dụ: bật/tắt LED

        vTaskDelay(500 / portTICK\_PERIOD\_MS);  // Chờ 500ms

    }

}

void vTaskFunction2(void \*pvParameters)

{

    while(1)

    {

        // Mã của tác vụ 2

        // Ví dụ: bật/tắt một LED khác

        vTaskDelay(1000 / portTICK\_PERIOD\_MS); // Chờ 1 giây

    }

}

#### 3.1.2. Tạo task với static allocation

xTaskCreateStatic()

Các step:

* Dinh nghia vùng nhớ cho stack và task control block
* Tạo hàm thực thi
* Tạo task bằng xTaskCreateStatic()
* Khởi động scheduler vTaskStartScheduler();
* Vòng lặp chính for (;;);

Vd:

#include "FreeRTOS.h"

#include "task.h"

// 1. Định nghĩa Stack và Control Block cho task

static StackType\_t xTask1Stack[configMINIMAL\_STACK\_SIZE];

static StaticTask\_t xTask1Buffer;

// 2. Hàm task

void vTask1(void \*pvParameters)

{

    for (;;)

    {

        // Code thực hiện trong task 1

    }

}

int main(void)

{

    // 3. Tạo task sử dụng static allocation

    xTaskCreateStatic(

        vTask1,               // Hàm task

        "Task 1",             // Tên task

        configMINIMAL\_STACK\_SIZE, // Kích thước stack

        NULL,                 // Tham số cho task

        tskIDLE\_PRIORITY,     // Ưu tiên task

        xTask1Stack,          // Con trỏ stack cho task

        &xTask1Buffer         // Control Block của task

    );

    // 4. Khởi động RTOS scheduler

    vTaskStartScheduler();

    // 5. Nếu RTOS không bắt đầu scheduler, chương trình sẽ vào vòng lặp vô tận

    for (;;);

}

### 3.2.Quá trình tạo task.

Giả sử project sử dụng RAM = SRAM1 + SRAM2 như hình

freeRTOS sẽ config 1 phần trong đó làm vùng nhớ HEAP

hàm xTaskCreate sẽ dùng malloc để khởi tao **TCB và stack** cho task đó trong vùng này

=> vùng nhớ task đó chiếm trong heap = sizeof(stack) + sizeof(TCB)

1 số hàm khác cũng sẽ sử dụng vùng nhớ này:xSemaphoreCreateBinary(), xQueueCreate()

A diagram of a diagram

AI-generated content may be incorrect.

Khi sử dụng xTaskCreate, 3 việc quan trong sau sẽ được thực khi

1. TCB(task control block) sẽ được khởi tạo trong RAM (HEAP session).
2. Khởi tạo stack cho task, stack này sẽ được tracked bởi PSP reg of ARM cortex
3. Đưa task vào “Ready” list để scheduler chọn

A screenshot of a computer

AI-generated content may be incorrect.

### 3.2.TCB-Task control block:

Trong **FreeRTOS**, **TCB** (Task Control Block) là một data structure( C struct) lưu trữ thông tin liên quan đến mỗi task.

Khi tạo một task mới thông qua **xTaskCreate**, một **TCB** sẽ được tạo ra để quản lý và theo dõi trạng thái của tác vụ đó.

**TCB trong FreeRTOS**

**TCB (Task Control Block)** lưu trữ thông tin như:

1. **TopOfStack**: con trỏ đến vùng nhớ stack của task
2. **Context của tác vụ**: Bao gồm các thông tin về các thanh ghi (registers) của tác vụ tại thời điểm context switch. Điều này giúp FreeRTOS có thể lưu lại trạng thái của tác vụ khi nó không còn chạy và phục hồi lại khi tác vụ được quay lại.
3. **taskState**: TCB chứa thông tin về trạng thái của tác vụ (chạy, chờ, tạm dừng, v.v.).
4. **Độ ưu tiên của tác vụ**:
5. **Thông tin về thời gian chờ**: Nếu tác vụ phải chờ đợi một sự kiện, TCB sẽ lưu trữ thời gian chờ của tác vụ, chẳng hạn như thời gian hết hạn của một semaphore, queue hoặc timeout.
6. **Thông tin khác**: TCB có thể chứa các dữ liệu khác như các cờ trạng thái (flags), con trỏ đến các resource như semaphore, queue mà tác vụ có thể đang chờ hoặc sử dụng.

Vd:

typedef struct tskTaskControlBlock

{

    volatile StackType\_t \* pxTopOfStack; /\*< Points to the location of the last item placed on the tasks stack, must first element of TCB

volatile UBaseType\_t    uxTaskNumber;       /\* Task number for tracking \*/

    ListItem\_t              xStateListItem;     /\* List item used to store task in state lists (e.g., ready, blocked) \*/

    char \*                  pcTaskName;         /\* Name of the task \*/

    UBaseType\_t             uxPriority;         /\* Task's priority \*/

    TaskHandle\_t            xTaskHandle;        /\* Handle to the task \*/

    ...

} TCB\_t;

TCB có con trỏ pxTopOfStack, sẽ được sử dụng để tạo stack cho task.

## 4.Scheduling – lập lịch

là quá trình quản lý và quyết định thứ tự mà các tác vụ (tasks) sẽ được thực thi. Mục tiêu của việc lập lịch là phân phối tài nguyên hệ thống (chẳng hạn như CPU) một cách hiệu quả để đảm bảo các tác vụ quan trọng được thực thi đúng hạn và hệ thống hoạt động trơn tru.

* Scheduler(**bộ lập lịch**) là 1 phần code của freeRTOS kernel, chạy rong privileged mode của processor. Các task sẽ được điều phối bởi scheduler
* Invoke scheduler by call : vTaskStartScheduler();

Trong freeRTOS, scheduling được chia làm 2 loại, tùy vào config

* Pre-emtive scheduling
* Co-operative scheduling

=> hệ thống chỉ được config để sử dụng 1 trong 2

A diagram of a scheduler

AI-generated content may be incorrect.

|  |  |
| --- | --- |
| Pre-emptive | Co-operative |
| Pre-emption : thay đổi 1 running task bằng 1 task khác  OS có thể cướp CPU từ task đang chạy để chuyển sang thực thi 1 task khác có độ ưu tiên cao hơn  Task bị ngừng sẽ chuyển sang **ready state** | Task không bị ngừng đột ngột bởi OS. Task đang chạy sẽ tự ngừng hoặc ngường quyền cho task khác.  Task sẽ chạy liêntucj cho đến khi gọi hàm **vTaskDelay**() hoặc đã hoàn thành công việc |

### 4.1. Scheduler

Scheduler là 1 đoạn code dùng để quyết định task tiếp theo được chạy

Scheduler trong pre-emptive và co-operative họng đọng khác nhau

* Pre-emptive scheduling: scheduler có thể **cướp quyền** của tác vụ hiện tại để chuyển sang task vụ khác
* Co-operative scheduling: chỉ chuyển đổi task khi task **tự nguyện** (task done or call taskYIELD())

Scheduler được excuted khi:

* + Sau 1 RTOS tick (timer interrupt)
  + Khi task done 1 timeslice
  + Task blocking hoặc unblocking
  + Change task priority
  + Explicit bằng cách call taskYIELD(), gọi scheduler ngay lập trức

### 4.2. Pre-emptive scheduling

Được chia làm 2 loại

* Round-robin pre-emptive scheduling
* Priority based pre-emptive scheduling

#### Round-robin pre-emptive scheduling:

**các task sẽ có độ ưu tiên như nhau**.

1. **Time slice/quantum**: 1 task có 1 t/g thực thi cụ thể, nếu task hoàn thành trong t/g này => hệ thống sẽ chuyển sang task tiếp theo trong hàng đợi sử dụng scheduler và context switching.
2. **Scheduler** là 1 phần code được excute với chu ky nhất định theo **RTOS tick(**timer interrupt**)**, scheduler(điều phối/lập lịch) dùng để quyết định task được thực thi tiếp theo.

* **RTOS tick khác với time slice**

1. Nếu task không hoàn thành trong time slice. Hệ thống sẽ cướp qyền điều khiển (pre-empt) và chuyển sang task khác trong hàng đợi -> task vụ bị cướp sẽ phải đợi đến lượt tiếp theo
2. Chu trình round-robin: sau khi all task done, hệ thống sẽ quay lại task đầu tiên

A diagram of a computer program

AI-generated content may be incorrect.

Trong free RTOS có 1 macro để cấu hình cho RTOS tick. Mặc định là 1000 => schedule(timer interrupt) mỗi 1ms

#define configTICK\_RATE\_HZ              ( ( TickType\_t ) 1000 )

#### Priority based pre-emptive scheduling

1. **mỗi task có 1 độ ưu tiên**, task có độ ưu tiên cao sẽ được quyền sử dụng CPU trước task có độ ưu tiên thấp.
2. nếu 1 task có độ ưu tiên cao hơn + ready + có trong hàng đợi => sẽ được thực thi ngay lập tức
3. Task vụ hiện tại (độ ưu tiên thấp) sẽ bị cướp quyền(preempted) và đưa vào trạng thái đợi cho đến lượt thực thi tiếp theo
4. Nếu hết time slice(RTOS tick) mà task đang chạy + độ ưu tiên cao nhất + ready => thì scheduler vẫn để task đó chạy tiếp( t2, t3 trong hình).

A screenshot of a schedule

AI-generated content may be incorrect.

T1: start vơits task có độ ưu tiên cao nhất (task2)

T2, t3: tiếp tục chạy task2

* Ko phải context switching

T4: task2 blocked, chuyển sang task có độ ưu tiên cao nhất trong queue.

* Đây xảy ra context switching nhưng không phải pre-emption( vì task 2 tự ngừng).

T5: scheduler chuyển sang task1

T6: task2 được unblock(read in queue) => chuyển sang task2(độ ưu tiên cao hơn)

* Pre-emption + context switching

Question: sao scheduler có thể chạy không theo RTOS tick( t4, t6)

### 4.3. Co-operative scheduling

Không có sự cướp quyền**(pre-emption),** chuyển task dựa trên tính tự nguyện( khi task blocked/suspend or call **yield()** )

RTOS tick/scheduler không không gây ra cướp quyền(pre-emption), nhưng vẫn cần tick iintterupt để đảm bảo tính real tiem của kernel

A screenshot of a computer program

AI-generated content may be incorrect.

### Summary

| **Yếu tố** | **Round-Robin Pre-emptive** | **Priority-Based Pre-emptive** | **Co-operative Scheduling** |
| --- | --- | --- | --- |
| **Cách thức lập lịch** | Các tác vụ **cùng độ ưu tiên**, chạy theo vòng, mỗi tác vụ có **time slice** | Các tác vụ **có độ ưu tiên riêng**, tác vụ có độ ưu tiên cao được chạy trước | **Không có độ ưu tiên**  Các tác vụ phải tự nguyện nhường quyền cho hệ điều hành |
| **Preemption** | **Pre-emptive** (tác vụ có thể bị cướp quyền sau khi hết time slice) | **Pre-emptive** (tác vụ có độ ưu tiên thấp có thể bị cướp quyền) | **Non-preemptive** (tác vụ tự nguyện nhường quyền) |
| **Quản lý độ ưu tiên** | Không phân biệt độ ưu tiên giữa các tác vụ | Có phân biệt độ ưu tiên, tác vụ có độ ưu tiên cao hơn sẽ được chạy trước | Không có độ ưu tiên, các tác vụ được xử lý theo thứ tự gọi yield() |
| **Tính công bằng** | Công bằng (mỗi tác vụ có cơ hội như nhau) | Không công bằng (tác vụ với độ ưu tiên thấp có thể bị bỏ qua) | Tác vụ có thể chiếm CPU vô thời hạn nếu không nhường quyền |
| **Hiệu quả với hệ thống thời gian thực** | Không phù hợp với các hệ thống có yêu cầu thời gian thực cao | Phù hợp cho các hệ thống có yêu cầu thời gian thực và ưu tiên cao | Không phù hợp với hệ thống thời gian thực |
| **Độ phức tạp** | Trung bình (quản lý time slice và context switch) | Cao (quản lý độ ưu tiên và xử lý starvation) | Thấp (đơn giản nhưng không linh hoạt) |

**Starvation** (tạm dịch là "bỏ rơi") là một vấn đề trong các hệ thống lập lịch đa nhiệm (multitasking) khi một tác vụ hoặc tiến trình không được thực thi trong một khoảng thời gian dài do các tác vụ khác chiếm giữ CPU hoặc tài nguyên cần thiết một cách liên tục.

A black screen with white text

AI-generated content may be incorrect.

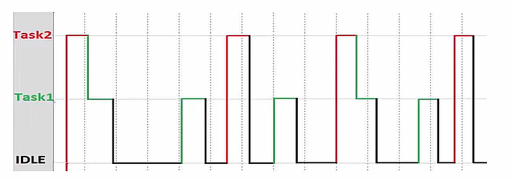
# Trace tool integration

A screenshot of a computer

AI-generated content may be incorrect.

# 8. Idle Task and Timer SvC Task

**Idle Task** là một tác vụ mặc định trong **RTOS**, được chạy khi **không có task nào khác sẵn sàng thực thi**. Đây là task có **độ ưu tiên thấp nhất (thường là 0)**, đảm bảo CPU không bị lãng phí khi hệ thống rảnh.



Vd trong freeRTOS: void vApplicationIdleHook(void)

## Idle Task

* **Idle Task được tạo tự động bởi RTOS** khi hệ thống khởi động.
* **Nó không bao giờ bị chặn (blocked)** và luôn ở trạng thái **Ready** khi không có tác vụ khác chạy.
* **Khi một task mới cần chạy, Idle Task bị tạm dừng ngay lập tức**

**Vai trò của Idle Task**

1. **Sử dụng CPU khi hệ thống rảnh**
   * RTOS không thể để CPU **ở trạng thái "chờ"** khi không có tác vụ nào để chạy.
   * Idle Task giữ cho CPU **bận rộn** thay vì để nó **chạy vào trạng thái không xác định**.
2. **Thực hiện các tác vụ bảo trì hệ thống**
   * **Giải phóng bộ nhớ động** nếu hệ thống hỗ trợ **Dynamic Memory Allocation (malloc/free)**.
   * **Dọn dẹp tài nguyên** của các task đã bị xóa.
3. **Chuyển CPU về chế độ tiết kiệm năng lượng (Low Power Mode)**
   * Trên một số hệ thống, Idle Task có thể **cho phép CPU ngủ (sleep mode)** để giảm mức tiêu thụ năng lượng.

## Timer Service Task

**📌 Timer Service Task là gì?**

**Timer Service Task** (còn gọi là **Daemon Task**) là một **task đặc biệt** trong FreeRTOS dùng để **xử lý các Software Timer và Callback Function**.

Khi một **Software Timer hết hạn**, thay vì xử lý ngay trong **ISR (Interrupt Service Routine)**, FreeRTOS **đưa yêu cầu vào hàng đợi** và để **Timer Service Task xử lý**. Điều này giúp giảm tải cho ngắt và cải thiện hiệu suất.

**📌 Cách Hoạt Động Của Timer Service Task**

1. Software Timer được tạo bằng **xTimerCreate()**.
2. **Timer Service Task được tạo khi khởi động scheduler** (nếu có ít nhất một timer).
3. Khi Timer hết hạn, FreeRTOS đưa sự kiện vào hàng đợi của Timer Service Task.
4. Timer Service Task chạy và thực thi Callback Function của Timer.

✅ **Ưu điểm:**  
✔ Không chặn ngắt (ISR chỉ cần gửi thông báo, không xử lý ngay).  
✔ Dễ quản lý tác vụ theo thời gian.  
✔ Giảm tải CPU và cải thiện hiệu suất.

**📌 Cách Kích Hoạt Timer Service Task**

**🔹 Bật Timer trong FreeRTOSConfig.h**

Để sử dụng Timer Service Task, cần bật các cấu hình sau trong **FreeRTOSConfig.h**:

c

Sao chépChỉnh sửa

#define configUSE\_TIMERS 1 // Bật Timer

#define configTIMER\_TASK\_PRIORITY 2 // Độ ưu tiên của Timer Task

#define configTIMER\_QUEUE\_LENGTH 10 // Độ dài hàng đợi Timer

#define configTIMER\_TASK\_STACK\_DEPTH 256 // Stack size của Timer Task

📌 **Lưu ý:**

* **configUSE\_TIMERS = 1** để bật Timer.
* **configTIMER\_TASK\_PRIORITY** đặt ưu tiên của Timer Task (nên cao hơn các task khác nếu Timer quan trọng).
* **configTIMER\_QUEUE\_LENGTH** quyết định số lượng Timer tối đa có thể xử lý cùng lúc.
* **configTIMER\_TASK\_STACK\_DEPTH** đặt kích thước Stack cho Timer Task.

**📌 Cách Tạo Software Timer trong FreeRTOS**

**🔹 Tạo Timer**

c

Sao chépChỉnh sửa

// Định nghĩa Callback Function khi Timer hết hạn

void TimerCallback(TimerHandle\_t xTimer) {

printf("Timer hết hạn! Thực hiện hành động...\n");

}

// Khai báo Timer Handle

TimerHandle\_t xTimer;

void setup() {

// Tạo Software Timer (tên, chu kỳ, auto-reload, ID, callback function)

xTimer = xTimerCreate("MyTimer", pdMS\_TO\_TICKS(1000), pdTRUE, (void\*)0, TimerCallback);

if (xTimer != NULL) {

// Bắt đầu Timer

xTimerStart(xTimer, 0);

}

}

📌 **Giải thích:**

* xTimerCreate() tạo một **Software Timer** có chu kỳ **1000ms (1 giây)**.
* pdMS\_TO\_TICKS(1000) chuyển đổi **milliseconds** sang **ticks**.
* pdTRUE cho phép Timer **tự động lặp lại** sau mỗi lần hết hạn.
* xTimerStart() bắt đầu chạy Timer.

**🔹 Tạm Dừng và Khởi Động Lại Timer**

c

Sao chépChỉnh sửa

xTimerStop(xTimer, 0); // Dừng Timer

xTimerStart(xTimer, 0); // Bắt đầu lại Timer

xTimerReset(xTimer, 0); // Reset Timer (đếm lại từ đầu)

**📌 Cách Xóa Timer**

Nếu không cần Timer nữa, hãy xóa nó để **giải phóng bộ nhớ**:

c

Sao chépChỉnh sửa

xTimerDelete(xTimer, 0);

**📌 Câu Hỏi Thường Gặp**

**1️⃣ Timer Service Task có thể bị xóa không?**

❌ **KHÔNG!** Nó là một phần của kernel FreeRTOS và **chạy ngầm** nếu có ít nhất một Timer đang hoạt động.

**2️⃣ Nếu có nhiều Timer hết hạn cùng lúc thì sao?**

✅ FreeRTOS **xử lý theo hàng đợi** (FIFO). Timer nào hết hạn trước sẽ được xử lý trước.

**3️⃣ Timer có thể chạy trong Interrupt không?**

❌ **KHÔNG!** Callback Function của Timer **không được gọi trong ISR**. Nếu cần sử dụng trong ngắt, hãy dùng **Task Notification hoặc Semaphore**.

**📌 Kết Luận**

✔ **Timer Service Task giúp xử lý Software Timer một cách hiệu quả**.  
✔ **Giúp giảm tải CPU và tránh chặn ISR**.  
✔ **Có thể dùng để thực hiện các tác vụ theo chu kỳ (ví dụ: đọc cảm biến, gửi dữ liệu UART, bật/tắt đèn LED, v.v.).**

Bạn muốn triển khai Timer trong ứng dụng cụ thể nào?

# 9. FreeRTOS scheduler

## 9.1FreeRTOS scheduler

- trong freeRTOS, scheduler code = freeRTOS generic code + architecture spefic code

A blue and red scheduler

AI-generated content may be incorrect.

### 9.1.1.FreeRTOS Generic Code (Mã chung của FreeRTOS)

* Đây là phần mã nguồn **độc lập với kiến trúc** (architecture-independent) và có thể chạy trên nhiều vi xử lý khác nhau.
* Được viết chủ yếu trong **C**.
* Chứa các thành phần chính như:
  + Quản lý **task** (tạo, xóa, chuyển đổi trạng thái).
  + Quản lý **queue**, **semaphore**, **mutex**.
  + Lập lịch (scheduling) dựa trên **preemptive** hoặc **cooperative**.
  + Quản lý **tick timer** và **idle task**.
  + Cấu hình **memory management**.

### 9.1.2.Architecture-Specific Code (Mã đặc thù theo kiến trúc)

* Phụ thuộc vào **vi xử lý** và **bộ vi điều khiển** mà FreeRTOS đang chạy.
* Được viết chủ yếu bằng **assembly** và **C**.
* Chứa các thành phần như:
  + **Context switching** (lưu và khôi phục trạng thái của task khi chuyển đổi).
  + **Interrupt handling** (ISR) cho các kiến trúc khác nhau.
  + **Timer tick implementation** (thường dùng bộ định thời của vi điều khiển).
  + **Stack management** (cách stack được sử dụng tùy vào vi xử lý).

📌 **Ví dụ về FreeRTOS trên kiến trúc ARM Cortex-M**:

* **Generic code**:
  + tasks.c, queue.c, list.c, timers.c
* **Architecture-specific code**:
  + port.c, portmacro.h (chứa phần cứng đặc thù)
  + portASM.S (chứa assembly code cho context switch)

Đối với ARM cortex Mx, 3 ngắt dưới nên được đặt trong port.c của freeRTOS

| **Handler** | **Chức năng chính** |
| --- | --- |
| vPortSVCHandler | Chuyển vào chế độ **privileged** khi khởi tạo task.  Được sử dụng để launch very first task, sau đó không được dùng nữa |
| xPortPendSVHandler | **Chuyển đổi context (context switch)** giữa các task |
| xPortSysTickHandler | **Tick interrupt/RTOS Tick management** để cập nhật scheduler |

## 9.2 Start scheduler

### 9.2.1.vTaskStartScheduler – generic code

- generic code của free RTOS

- được implemented trong task.c, sử dụng để start RTOS scheduler

### 9.2.2. xPortStartScheduler

A diagram of a program

AI-generated content may be incorrect.

# 10. FreeRTOS vs ARM architecture specific

## 10.1.FreeRTOS kernal interrupt

- FreeRTOS chạy trên ARM cortex Mx, below interrupt được implement để scheduling task

1. **SVC interrupt** : launch very first task, chỉ chạy 1 lần
2. **PendSV interrupt**: chuyển đổi context switching
3. **SysTick interrupt**: implement RTOS Tick management => nếu SysTick interrupt được sử dụng cho mục đích khác của ứng dụng, có thể sử dụng TIMER Interrupt để thay thế

* Tất cả interrupt phải được config **độ ưu tiên thấp nhất** có thể

## 10.2. RTOS TICK và SysTick Timer

-

- Trong FreeRTOS, **RTOS tick** là một tín hiệu định kỳ do bộ đếm thời gian (timer) của vi điều khiển tạo ra để cập nhật kernel và quản lý việc lập lịch (scheduling) tác vụ. Tick này giúp hệ thống biết thời gian đã trôi qua, từ đó có thể chuyển đổi giữa các tác vụ đúng thời điểm.

### 10.2.1.Cách Hoạt Động của RTOS Tick

* RTOS TICK thường được implemented sử dụng Systick Timer của Arm Cortex Mx (nhiều MCU dùng Arm sẽ như nhau)
* Mỗi lần có ngắt **tick interrupt** -> **trigger run scheduler**:
  + Cập nhật thời gian hệ thống (xTickCount+=1).
  + Kiểm tra xem có tác vụ nào cần chuyển trạng thái (ví dụ, tác vụ chờ bị hết thời gian delay).
  + Chuyển đổi ngữ cảnh (context switch) nếu cần thiết.

Mặc định, FreeRTOS dùng **SysTick** (trên Cortex-M), nhưng bạn có thể thay đổi sang một timer khác nếu cần tối ưu hóa

### 10.2.2. xPortSysTickHandler – SysStick interrupt function

**#define** xPortSysTickHandler SysTick\_Handler

Vd: freeRTOS dùng SysTick Timer để làm update RTOS tick

* xPortSysTickHandler là hàm xử lý ngắt SysTick trong FreeRTOS
* được call khi có ngắt sysTick (**24-bit** được tích hợp sẵn trong vi điều khiển **ARM Cortex-M)**
* #define configTICK\_RATE\_HZ 1000 // 100Hz = ngắt 1ms

SysTick là một **timer 24-bit** được tích hợp sẵn trong vi điều khiển **ARM Cortex-M**. Nó hoạt động theo nguyên tắc:  
**Nạp giá trị SysTick->LOAD**

* Đây là giá trị **đếm ngược** (reload value).
* Khi bộ đếm đếm hết (SysTick->VAL về 0), nó sẽ **tạo ra ngắt**.

**Bắt đầu đếm ngược từ SysTick->LOAD xuống 0**

* Giá trị này được giảm mỗi chu kỳ xung clock.

**Khi bộ đếm về 0**

* Ngắt **SysTick** được tạo ra.
* Nếu chế độ Auto-reload được bật, bộ đếm sẽ tự động nạp lại từ SysTick->LOAD và tiếp tục đếm.

void xPortSysTickHandler( void )

{

    /\* The SysTick runs at the lowest interrupt priority, so when this interrupt

     \* executes all interrupts must be unmasked.  There is therefore no need to

     \* save and then restore the interrupt mask value as its value is already

     \* known. \*/

    portDISABLE\_INTERRUPTS();

    traceISR\_ENTER();

    {

        /\* Increment the RTOS tick. \*/

        if( xTaskIncrementTick() != pdFALSE )

        {

            traceISR\_EXIT\_TO\_SCHEDULER();

            /\* A context switch is required.  Context switching is performed in

             \* the PendSV interrupt.  Pend the PendSV interrupt. \*/

            portNVIC\_INT\_CTRL\_REG = portNVIC\_PENDSVSET\_BIT;

        }

        else

        {

            traceISR\_EXIT();

        }

    }

    portENABLE\_INTERRUPTS();

}

* Flow char của sysTick interrupt

1. Interrupt xảy ra -> trigger hàm ngắt
2. Disable other interrupts: vì sysTick interrupt được cấu hình ở độ ưu tiên nhấp nhất, cần tắt các interrupt khác trong lúc nó chạy
3. Call hàm xTaskIncrementTick()
   1. xTickCount++
   2. Kiểm tra các task trong DelayedList. Nếu task hết delay thì xóa khỏi DelayedList, đưa sang ReadyList.
   3. Nếu task có độ ưu tiên cao hơn task hiện tại => đánh dấu cần context switching
4. Nếu xTaskIncrementTick() return true nghĩa là cần context switching => pend(treo) pend SV interrupt bằng cách set bit pending.
5. Enable interrupt.

A diagram of a flowchart

AI-generated content may be incorrect.

**Note:**

* xTickCount: là biến đếm số tick của hệ thống, khi một tick interrupt xảy ra, xTickCount+=1.
* pxDelayedTaskList : danh sách delay task, được sắp xếp **tăng dần theo** **xItemValue**
* xItemValue : lưu trữ thời điểm một task cần được unblock.
* pxReadyTasksLists: danh sách chứa các task đang sẵn sàng chạy (Ready State), được tổ chức theo độ ưu tiên.
* PendSV không được xử lý ngay lập tức. CPU sẽ xử lý nó khi:
* Không có ngắt nào có độ ưu tiên cao hơn đang chạy.
* CPU trở về chế độ thread mode hoặc kết thúc các ISR khác.

### 10.2.3. ReadyTaskList vs DelayedTaskList

|  |  |  |
| --- | --- | --- |
| Tiêu chí | **pxReadyTasksLists** | **pxDelayedTaskList** |
| Chứa task ở trạng thái | Ready (Sẵn sàng chạy) | Blocked (Đang chờ timeout) |
| Dữ liệu lưu trữ | Danh sách các task theo **độ ưu tiên** | Danh sách các task theo **thời gian chờ** (xItemValue) |
| Số lượng danh sách | **configMAX\_PRIORITIES** danh sách (mỗi mức ưu tiên 1 danh sách) | 2 danh sách (pxDelayedTaskList và pxOverflowDelayedTaskList) |
| Sắp xếp task theo | Độ ưu tiên (priority) | xItemValue (tick khi task sẽ hết chờ) |
| Task được thêm vào khi nào? | Khi task được tạo, hết chờ hoặc được kích hoạt lại | Khi task gọi vTaskDelay(), vTaskDelayUntil(), hoặc chờ event |
| Task bị xóa khỏi khi nào? | Khi task được chọn để chạy hoặc bị block | Khi timeout (đến xItemValue), task được chuyển sang Ready |

# 11. Context Swiching

**Context Switching** là quá trình **chuyển đổi từ task này sang task khác** trong RTOS

Trong **FreeRTOS**, context switch có thể xảy ra trong các tình huống sau:

1. **Preemptive Scheduling**: Khi một task có ưu tiên cao hơn được sẵn sàng chạy.
2. **Time Slicing (Round Robin)**: Khi nhiều task có cùng ưu tiên, chúng chia sẻ CPU theo thời gian.
3. **Task bị Blocked hoặc Delay**
4. **Gọi taskYIELD()**: Một task có thể tự nguyện nhường CPU.
5. **Ngắt hệ thống (ISR) yêu cầu context switch**: Khi có sự kiện ngắt làm thay đổi trạng thái task.

## 11.2.Task State

- Task State = content of CPU reg + Task Stack content

- Task Stack được track bởi PSP

- Kernel stack sẽ được track bởi MSP ( kernel code, interrupt,…)

A screenshot of a computer screen

AI-generated content may be incorrect.

A diagram of a ram and ram

AI-generated content may be incorrect.

## 11.3. Task switching out procedure

* Trước khi switch out task, sẽ làm những thứ như này:

1. Processor core reg R0,R1,R2,R3,R12,LR,PC,xPSR(stack frame) được lưu vào stack của task tự động bởi processor sysTick interrupt entry sequence => quá trình này là STACKING của sysTick interrupt.
2. Nếu context switching is required, sysTick interrupt sẽ pend PendSV exception và PendSV handle fucntion được chạy
3. Processor core reg (R4-R11, R14) phải được lưu lại 1 cách manually trong stack memory của task ( save the context) => quá trình này được thực thi ở PendSV Handle
4. Lưu lại giá trị của lastest top of stack value(PSP) trong first member của TCB.

A screen shot of a computer

AI-generated content may be incorrect.

1. Chọn task có thể chạy tiếp theo và update vào **pxCurrentTCB** by call vTaskSwitchContext().

## 11.3.Task switching In procedure

- Tại thời điểm này, đã biết được task nào sẽ được chạy tiếp theo nhờ vào **pxCurrentTCB.**

1. Get địa chỉ của top of stack lưu vào PSP: PSP = pxCurrentTCB->pxTopOfStack
2. Pop all reg (R4-R11, R14) -> restore the context (đã được lưu ở lần switch out trước đó)
3. Exeption exit (UNSTACKING) : các reg R0,R1,R2,R3,R12,LR,PC,xPSR sẽ được update giá trị từ stack của task.
4. PC lúc này sẽ lưu vị trí của task đang được thực thi lần trước đó => task resume tại vị trí đó

A diagram of a computer system

AI-generated content may be incorrect.

# FreeRTOS memory management

* RTOS kernel sử dụng 1 phần của RAM để làm HEAP nhằm cấp phát bộ nhớ động cho task, queue, semaphore,…