**Memory layout**

Text Segment là nơi chứa các đoạn mã lệnh của chương trình.

Initialized Data Segment ( Data Segment ) là nơi lưu trữ **biến toàn cục, biến tĩnh**với giá trị được khởi tạo với **giá trị khác 0.**

Uninitialized Data Segment (BSS) là nơi lưu trữ **biến toàn cục, biến tĩnh không được khởi tạo**hoặc khởi tạo với **giá trị bằng 0.**

Diagram, table

Description automatically generated with medium confidence

Stack là nơi các biến local lưu trữ và thông tin được lưu mỗi khi gọi hàm, Mỗi khi hàm được gọi sẽ có một Function Frame của chính nó.

Heap là nơi diễn ra quá trình cấp phát bộ nhớ động và được quản lý bởi các hàm malloc, relloc và free.

Sự khác biệt:

* Trong stack, việc cấp phát và giải phóng bộ nhớ được thực hiện tự động, trong khi đó, heap cần phải được thực hiện thủ công.
* Stack không linh hoạt, kích thước bộ nhớ được cấp phát không thể thay đổi trong khi heap là linh hoạt và bộ nhớ được cấp phát có thể thay đổi được.
* Thời gian truy cập của heap mất nhiều hơn stack.

**Preprocessor**

Mục đích của file header là cung cấp khai báo hàm, biến và các macro để sử dụng và share giữa các file mã nguồn.

Preprocessor là những thông tin được xử lý trước khi bắt đầu quá trình biên dịch.

Macro là các đoạn mã được đặt tên bởi chỉ thị #define được sử dụng để thay thế văn bản. Trong quá trình preprocessor, khi gặp tên này, nó sẽ được thay thế bằng nội dung của macro, hiểu đơn giản là sự thay thế văn bản.

*Ưu điểm:*

* Giảm mã trùng lặp và tiết kiệm thời gian
* Tốc độ thực thi nhanh do thay thế ngay tại chỗ trong mã nguồn, nên chúng có thể loại bỏ chi phí bộ nhớ và thời gian do phân bổ ngăn xếp cho lời gọi hàm

*Nhược điểm:*

* Khó debug do macro được xử lý trong Preprocessor.
* Dễ bị lỗi vì không thực hiện kiểm tra kiểu.
* Tăng độ dài của code

**Struct, Enum, Union**

**Enum**

Enum là một kiểu dữ liệu do người dùng định nghĩa trong C. Nó chủ yếu được sử dụng để gán tên cho các hằng số nguyên, giúp chương trình dễ đọc và bảo trì.

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, màu xanh lá cây

Mô tả được tạo tự động

Nếu không gán giá trị cho tên enum, nó sẽ mặc định gán giá trị bắt đầu từ 0. Ngoài ra, tên enum chưa được gán giá trị có thể nhận giá trị bằng giá trị của tên trước đó cộng 1.

Giá trị được gán cho tên enum phải là hằng số nguyên và tất các các hằng số enum phải là duy nhất trong phạm vi của chúng.

Điều làm cho “enum” khác với “macro” là nó tự động gán giá trị cho các member trong nó.

Ảnh có chứa văn bản, Phông chữ, hàng, ảnh chụp màn hình

Mô tả được tạo tự động

**Struct**

Được sử dụng để định nghĩa một kiểu dữ liệu tuỳ chỉnh gồm nhiều kiểu dữ liệu khác nhau giúp tổ chức dữ liệu và truy cập các member trong nó một cách dễ dàng.

**Struct padding**

Struct padding là quá trình phân bổ bộ nhớ, trình biên dịch sẽ thêm một số byte vào giữa các member của struct để đảm bảo các member được canh chỉnh(align).

Do CPU hiện đại truy xuất dữ liệu theo từng khối có dung lương bằng 2^N byte như 2,4,8…

* Những (member) nằm ở địa chỉ mà chia hết cho chính kiểu dữ liệu của nó.
* Kích thước của kiểu struct là bội số của kiểu dữ liệu lớn nhất trong nó.

Việc này giúp các dữ liệu được truy cập và sử dụng bộ nhớ một cách nhanh và hiệu quả nhất. Tuy nhiên, nó dẫn đến lãng phí bộ nhớ và giảm hiệu suất chương trình.

**Struct packing**

Struct packing là quá trình xác định cách mà các member trong struct được sắp xếp trong bộ nhớ. Khi sử dụng struct packing, các member trong struct được lưu trữ liên tục nhau, không có byte nào trống ở giữa. Giảm thiểu sự lãng phí bộ nhớ và tăng hiệu suất của chương trình.

Tuy nhiên, nó có thể dẫn đến sự mất mát dữ liệu hoặc lỗi trong các trường hợp khi các trường bị sắp xếp không đúng, vì chúng không được đặt đúng trên các địa chỉ bộ nhớ bắt đầu bằng bội số của kích thước của chúng.

Sử dụng: struct *name* {

...

}\_\_attribute\_\_((packed));

hoặc #pragma pack(1).

**Union**

Được sử dụng để định nghĩa một kiểu dữ liệu tuỳ chỉnh gồm nhiều kiểu dữ liệu khác nhau giống struct.

Kích thước của union sẽ bằng kích thước của member lớn nhất của nó. Tất cả member có kích thước nhỏ hơn có thể lưu trữ dữ liệu trong cùng một không gian bộ nhớ mà không bị tràn. Điều đó dẫn đến chỉ một member trong union có thể lưu trữ dữ liệu tại một thời điểm.

Hữu ích trong các trường hợp muốn tiết kiệm không gian bộ nhớ.

**Typedef**

typedef là một từ khóa được sử dụng để định nghĩa một cái tên mới cho một kiểu dữ liệu có sẵn hoặc tự định nghĩa.

**Struct và Enum**

Khi struct được khai báo, giá trị của các phần tử bên trong nó có thể được sửa đổi => tổ chức và lưu trữ dữ liệu hơn enum.

Sau khi khai báo enum, giá trị của nó không thể thay đổi được, nếu không, trình biên dịch sẽ báo lỗi.

**Function**

Hàm là một khối lệnh mà khi được gọi sẽ thực hiện một số tác vụ cụ thể. Nó là khối xây dựng cơ bản của chương trình C cung cấp khả năng sử dụng lại mã.

* Chia bài toán lớn thành nhỏ hơn
* Khả năng tái sử dụng
* Thực hiện task cụ thể
* Dễ dàng quản lý và bảo trì
* Chương trình dễ hiểu và có cấu trúc hơn

Prototype được sử dụng để khai báo cho trình biên dịch biết thông tin của một hàm.

**When call function?**

Trong quá trình function call, các resister trong cpu sẽ tham gia vào quá trình lưu trữ các argument của function, lưu trữ các biến local trong function => resister hoàn toàn có thể bị thay đổi

Tiếp theo là thanh ghi PC - Program Counter, thanh ghi này sẽ lưu trữ địa chỉ câu lệnh hiện tại của chương trình.

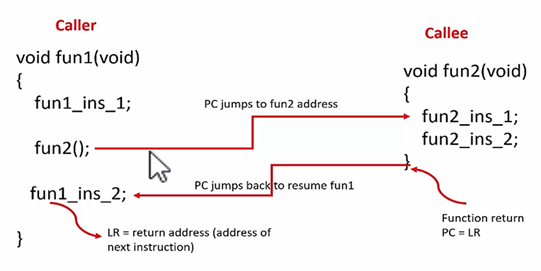
Khi thoát ra khỏi hàm, chương trình cần biết chính xác địa chỉ của câu lệnh tiếp theo. Vì vậy, giá trị địa chỉ của câu lệnh tiếp theo này cần được lưu trữ khi call function.

**Quá trình stacking - unstacking**

Stack hoạt động theo Last In First Out, do đó, khi call function, các thành phần của function sẽ được Push vào Stack - gọi là quá trình Stacking, và được pop ra khỏi stack khi thoát khỏi function - gọi là unstacking.

Các thành phần sẽ được push – pop trên stack:

* Các parameters, return value của function
* Địa chỉ câu lệnh tiếp theo(Thanh ghi LR – Link Resister: chứa địa chỉ trả về của các lệnh rẽ nhánh)
* Biến cục bộ
* Resister của CPU ( thanh ghi R0 – R12: các thanh ghi xử lý data, tính toán)



- Khi hàm fun1() gọi tới hàm fun2()

- PC sẽ nhảy đến địa chỉ của hàm fun2(),

- Lúc này LR sẽ chứa địa chỉ của lệnh tiếp theo, tức là lệnh fun1\_ins\_2;

- Và sau khi thực hiện xong hàm fun2() thì PC sẽ lấy giá trị của thanh ghi LR để trỏ đến lệnh tiếp theo (fun1\_ins\_2) để tiếp tục thực hiện.

**Lưu ý**

Về mặt thời gian, thời gian cho chi phí gọi hàm là không lớn, tuy nhiên đối với các function nhỏ và gọi nhiều lần thì chi phí thời gian là không hề nhỏ.

Về mặt bộ nhớ, việc call function chiếm một số bộ nhớ stack khá lớn để lưu Frame Stack.

**Pass by value - Pass by reference**

Kích thước tham số truyền vào quá lớn.

Khi sử dụng Pass by value, khi call function cần tạo một biến tạm với kích thước bằng với kích thước của biến truyền vào. Vad nếu biến có kích thước lớn thì sẽ tốn rất nhiều bộ nhớ stack => dễ dẫn đến stack overflow => sử dụng Pass bu reference => giảm gánh nặng cho bộ nhớ stack khi call function.

**Hàm Inline**

Hàm Inline được sử dụng để thay thế trực tiếp code tại nơi xảy ra lệnh gọi hàm của nó. Điều này làm cho việc thực thi nhanh hơn bằng cách loại bỏ chi phí gọi hàm. Tuy nhiên, điều này làm tăng độ dài của code, gây tốn bộ nhớ.

Với trình biên dịch GCC thì nó sẽ coi Inline Function như một phần của quá trình tối ưu hóa - Optimization. Vì vậy, việc gọi hàm Inline Function sẽ không thành công, do quá trình linker không thể tham chiếu tới hàm này.

Cách giải quyết ở đây là có thể tắt Optimization đi nếu có thể. Còn nếu vẫn muốn tiếp tục bật Optimize thì thêm từ khóa static vào trước hàm inline. Điều này sẽ làm linker phải kiểm tra và tham chiếu đến hàm inline.

Inline chỉ là một yêu cầu cho trình biên dịch, không phải là một lệnh bắt buộc.

Trình biên dịch có thể không thực hiện Inline trong các trường hợp như:

* Hàm chứa vòng lặp (for, while, do-while).
* Hàm chứa các biến tĩnh.
* Hàm đệ quy.
* Hàm chứa câu lệnh switch hoặc goto.

**Build Process For MCU**

Build Process: Về cơ bản tương tự một chương trình C trên window tuy nhiên có một vài điểm khác biệt:

+ Compiler không compile ra file ASM của máy tính nữa, mà ở đây là Cross Compiler.

* Cross Compiler nó cho phép mình build chương trình trên một máy tính (chẳng hạn như chip x86 intel), nạp và chạy trên một chip khác (ARM)
* Máy tính mà mình build chương trình gọi là “host”, trong khi môi trường máy tính (chip) gọi là “target”
* arm-none-eabi-gcc

A diagram of a machine

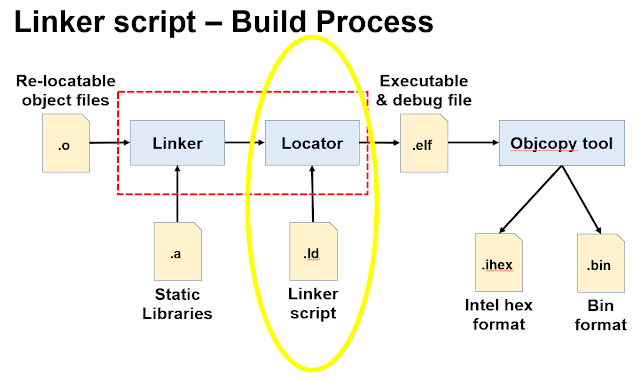
AI-generated content may be incorrect.

+ Các file Object vẫn được link với thư viện tĩnh, nhưng cần trải qua một bước nữa đó là Locator, bước này dùng 1 file Linker Script để phân bố vùng nhớ cho vi điều khiển.

* Locator hoạt động dựa trên cái file Linker Script (.ld)

+ File đích tạo ra không còn là .exe mà là các file như .elf (dùng để execute & debug), các file định dạng khác như .hex/ .bin để nạp code.

Một [Linker Script File](https://www.laptrinhdientu.com/2021/08/LinkerScriptFile.html) (.ld) là một file text trong quá trình locator nhằm phân vùng địa chỉ trên vi điều khiển, do các vi điều khiển có vùng địa chỉ tuyệt đối trên flash, ram (không giống như máy tính).



Dữ liệu trong Data Section được lưu trên cả Flash và Ram để có đặc tính Non - Volatile của Flash (dữ liệu không bị mất khi reset) và được copy sang RAM sau khi reset, đặc tính Ramdom Access Memory của RAM để thay đổi dữ liệu.

Khi microcontroller được boot thì nó sẽ chạy startup code bên trong flash memory trước khi vào hàm main.

Và startup code này sẽ khởi tạo các biến global => được lưu trữ trong cả flash memory và data section trong RAM.

Có hai loại bộ nhớ Flash và RAM:

* Flash: Non volatile, not ramdom và được sử dung để lưu trữ code.
* RAM: volatile, ramdom access memory và được sử dung để lưu trữ data.

**Reset Sequence**

[Startup file](https://www.laptrinhdientu.com/2021/08/StartupFile.html) được thực hiện trước hàm main, là file khởi tạo ban đầu trước khi call hàm main chạy và chứa vector table.

Startup file:

+ Tìm đến Reset\_Handler() ở trong vector table.

+ Khởi tạo section .data và .bss => copy section data và bss từ Flash lên RAM.

+ Khởi tạo standard library

+ Call đến hàm Main

Khi vi điều khiển reset nó sẽ tìm đến hàm Reset\_Handler() ở trong vector table (trong startup file) sau đó thực hiện một số công việc trong startup file.

[Main Stack Pointer - MSP:](https://www.laptrinhdientu.com/2021/09/Core5.html) chứa địa chỉ của bộ nhớ segment ở thời điểm hiện tại, tức là giá trị thanh ghi này sẽ thay đổi theo địa chỉ mà mình đang thao tác với bộ nhớ Stack.

Khi reset, CPU sẽ tìm đến địa chỉ của Vector Table (0x0000.0000), gán giá trị của địa chỉ này vào MSP để quy định vùng nhớ Stack, tiếp theo bộ xử lý lấy giá trị được đặt trong ô nhớ thứ 2 của Vector Table và gán vào PC để thực hiện hàm ResetISR để thực hiện Startup code. Cuối cùng, nó sẽ gọi đến hàm main.

Vector table: là một vùng nhớ lưu trữ giá trị bắt đầu của Stack, địa chỉ bắt đầu của các hàm Handler.

**CMSIS**

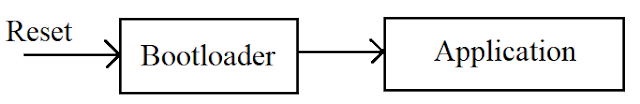
CMSIS: Là một tiêu chuẩn chung giữa các dòng vi điều khiển sử dụng lõi ARM, giúp người dùng dễ dàng giảm thời gian tiếp cận với các thiết bị phần cứng mới và có khả năng tái sử dụng.

Software Interface: Software - Software Interface và Software – Hardware Interface.

**Bootloader**

Bootloader là một phần mềm đặc biệt được nạp vào bộ nhớ của máy tính trước khi khởi động.

Nếu có một chương trình Bootloader được nạp vào trong vi điều khiển thì trước tiên nó sẽ nhảy vào chương trình Bootloader này, thực hiện một số công việc sau đó chuyển sang chương trình Application để thực hiện.

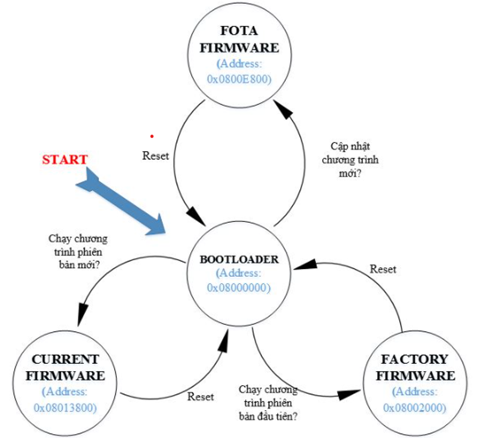


*Tại sao phải nạp bootloader trước khi khởi động?*

* Phát triển phần mềm, update firmware => nạp qua mạng không dây => Bootloader.
* Tính bảo mật => thêm một số đoạn code để kiểm tra firmware được nạp có đúng hay không => không hợp lệ thì không cho update => báo IP xâm nhập

*Hiểu được chương trình Boot của chúng ta nằm ở địa chỉ nào.*

Thực hiện Bootloader bằng cách đặt chương trình Bootloader ở một sector nào đó trên bộ nhớ Flash, chẳng hạn Sector 0. Và cùng lúc đặt Firmware Application 1 vào Sector 1, Firmware Application 2 vào Sector 2.



* Factory Firmware: Là phiên bản đầu tiên của Firmware mà nhà sản xuất cung cấp cho người dùng.
* Current Firmware: Là phiên bản hiện tại của Firmware đang chạy trên Vi điều khiển, được chúng ta lưu trên 1 sector nào đó (chẳng hạn sector 1).
* FOTA Firmware: Là phiên bản cập nhật mới của Firmware. Ở bài toán này, khi chương trình đang chạy với Current Firmware, chúng ta sẽ thực hiện quá trình boot để cập nhật chương trình FOTA mới này.

=> Sau khi reset => CPU gọi chương trình Bootloader => làm một số công việc -> lựa chọn xem chạy Firmware nào.

Ví dụ: Khi đang chạy Current Firmware => thực hiện Boot bằng cách nào đó => Vi điều khiển reset => Thực hiện chương trình Bootloader Firmware => thực hiện FOTA Firmware.

**Coxtex M0+**

**Introduce**

Bộ xử lý Cortex-M0+ là bộ xử lý ARM Cortex 32-bit cơ bản được thiết kế theo kiến trúc ARMv6-M cho các ứng dụng nhúng. Xây dựng theo kiến trúc von Neumann.

Ảnh có chứa văn bản, biểu đồ, Kế hoạch, sơ đồ

Mô tả được tạo tự động

**Stack**

Bộ xử lý sử dụng Descending Stack, nghĩa là Stack Pointer chỉ item cuối cùng trong bộ nhớ ngăn xếp. Tức là khi bộ xử lý push một item mới vào stack, Stack Pointer giảm và sau đó ghi item vào vị trí mới.

Bộ xử lý thực hiện hai Stack là: Main Stack và Process Stack.

Chế độ Thread: Main Stack và Process Stack.

Chế độ Handler: Main Stack.

**Core Resister**

Ảnh có chứa văn bản, ảnh chụp màn hình, biểu đồ, hàng

Mô tả được tạo tự động

**Memory**

Thanh ghi Non – Memory Mapped, là những thanh ghi nằm trong core bộ xử lý và không có bất kỳ địa chỉ nào để truy cập chúng từ chương trình C => Muốn truy cập phải dùng tập lệnh Assembly.

Thanh ghi Memory Mapped – Thanh ghi ánh xạ bộ nhớ: là một phần của memory và là những thanh ghi có địa chỉ. Bằng cách sử dụng địa chỉ này ta có thể đọc và ghi dữ liệu vào thanh ghi này bằng chương trình C.

* Thanh ghi liên quan đến ngoại vi bộ xử lý: NVIC, MPU, SCB, DEBUG
* Thanh ghi liên quan đến ngoại vi của vi điều khiển: GPIO, RTC, I2C, TIMER, CAN, …

Trong bộ xử lý, vùng nhớ có thể định địa chỉ phụ thuộc vào kích thước của bus địa chỉ.

Địa chỉ bắt đầu của memory thường là bộ nhớ Flash.

Bộ nhớ Flash được chia thành các Sector để thao tác với bộ nhớ hiệu quả. Bộ nhớ Flash, ngoài lưu trữ MSP, Vector Table, bộ nhớ Flash sẽ lưu trữ vùng nhớ chương trình ứng dụng, cùng với đó là vùng data (bao gồm read-only data, và vùng nhớ .data).

Bộ nhớ Flash có thể được thao tác ghi trên từng Word (4 bytes), nhưng lại chỉ có thể xóa theo từng Sector. Vì vậy để tiết kiệm số lần ghi/xóa (ảnh hưởng tới tuổi thọ của Flash) cũng như để dễ quản lý, bộ nhớ Flash được chia nhỏ thành các sector, với những sector đầu có kích thước nhỏ, và những sector sau có kích thước lớn dần.

**Interrupt & Exception**

Một bộ vi điều khiển cortex M hỗ trợ tới 256 ngắt, trong đó có 15 ngắt System Exceptions là các ngắt đến từ core.

CMSIS sử dụng giá trị âm cho các System Exceptions. Các ngắt còn lại là các ngắt ngoại vi (Interrupt Request - ISR) được bắt đầu bằng 0.

=> Dễ dàng phân loại System Exceptions và Peripheral Interrupt.

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, phần mềm

Mô tả được tạo tự động

Khi một ngắt được phục vụ, Interrupt Number hoặc Current Exceptions được ghi vào thanh ghi Program Status Register (PSR).

Interrupt Number trong PSR khác với Interrput Number trong hàm CMSIS:

Interrupt Number in PSR = 16 + Interrupt Number for CMSIS.

Kích hoạt System Exception khác với kích hoạt Peripheral Interrupt.

* Một số System Exception như reset và hard fault luôn được kích hoạt. Các System Exception có thể được bật hoặc tắt bởi module tương ứng của chúng chẳng hạn như Systick.
* Bật/tắt tất cả các Peripheral Interrupt được thực hiện bằng cách sửa đổi hại bộ thanh ghi: thanh ghi ISER và thanh ghi ICER.

Thanh ghi ISER được sử dụng để kích hoạt các Peripheral Interrupt. Ghi 1 vào một bit trong ISER để kích hoạt ngắt tương ứng. Tuy nhiên, ghi 0 vào một bit trong thanh ghi không vô hiệu hoá ngắt tương ứng.

Thanh ghi ICER được sử dụng để vô hiệu hoá các Peripheral Interrupt. Ghi 1 vào một bit trong ICER sẽ vô hiệu hoá ngắt tương ứng. Ghi 0 vào một bit trong ICER không có tác động.

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, màn hình

Mô tả được tạo tự động

**Exception States**

Pending: Exception đang chờ bộ xử lý phục vụ, yêu cầu ngắt từ thiết bị ngoại vi hoặc từ phần mềm có thể thay đổi trạng thái của ngắt tương ứng để chờ xử lý.

Active: Exception đang được bộ xử lý phục vụ nhưng chưa hoàn thành.

Inactive: Exception không active và không pending.

**NVIC**

NVIC - Nested Vector Interrupt Controller, là bộ điều khiển vector ngắt lồng nhau, nó là một ngoại vi của Core Cortex Mx.

Khi NVIC nhận được yêu cầu ngắt, nó buộc vi xử lý nhảy tới và thực thi một đoạn mã đặc biệt, được gọi là trình xử lý ngắt - ISR.

Các điểm vào của tất cả các chương trình ngắt, được lưu trữ trong một bảng đặc biệt gọi là Vector Table.

Vector Table được lưu trước tại một khu vực xác định trong bộ nhớ và được ánh xạ tới địa chỉ thấp nhất của bộ nhớ flash. Tuy nhiên, có thể ánh xạ lại nó tới bộ nhớ SRAM.

Vector Table chứa một mảng các địa chỉ bộ nhớ hay một mảng con trỏ hàm. Mỗi loại ngắt được gán một số gọi là Number Interrupt và được sử dụng để lập chỉ mục vào bảng Vector Table. Mỗi chỉ mục này dài 4 byte và chứa địa chỉ bắt đầu của một trình xử lý ngắt - ISR.

Ví dụ: Khi ngắt x được kích hoạt, NVIC sử dụng Number Interrupt x, làm giá trị chỉ mục, để tra cứu bảng vector ngắt, tìm địa chỉ bộ nhớ của trình xử lý ngắt - ISR tương ứng cho ngắt x và buộc vi xử lý nhảy tới và thực hiện ngắt này.

**External Interrupt Controller**

External Interrupt Controller giám sát sự thay đổi của tín hiệu điện áp => Yêu cầu ngắt => Gửi đến NVIC

Hỗ trợ 16 External Interrupt kết hợp với GPIO. Mỗi ngắt này được liên kết với một chân GPIO cụ thể.

Bộ điều khiển có một bộ ghép kênh - MUX cho mỗi External Interrupt này.

*Enable ngoại vi, Enable NVIC, Enable Core => interrupt mới được thực hiện.*

Một bộ xử lý có nhiều hơn 16 chân GPIO => Làm thế nào để bộ điều khiển mask các chân GPIO tới External Interrupt

=> Với STM32, Chỉ các chân có số pin x mới có thể là nguồn của External Interrupt x.

Ví dụ: Nguồn của External Interrupt 3 có thể là PA3, PB3, PC3, PD3, ...

Thanh ghi External Interrupt Configuration Resister(SYSCFG\_EXTICR) chỉ định chân nào được chọn làm nguồn External Interrupt.

Không có nhiều chân có thể được sử dụng đồng thời làm nguồn External Interrupt.

* Ví dụ: Nếu chân PA3 có External Interrupt thì không thể sử dụng chân PB3 làm nguồn cho External Interrupt 3.

**Priority**

Mức độ ưu tiên - Priority được coi là mức độ khẩn cấp của ngắt, tức là quy định ngắt nào cần được thực hiện trước.

Giá trị mức độ ưu tiên – priority value là thước đo mức độ khẩn cấp của ngắt, còn gọi là mức độ ưu tiên – priority levels.

Với Cortex M, giá trị ưu tiên này càng nhỏ thì mức độ ưu tiên càng lớn. Chẳng hạn, các Exception System đa số có mức ưu tiên nhỏ hơn 0 và được cố định (trong đó Reset Handler có priority value nhỏ nhất, tương đương với mức độ ưu tiên lớn nhất).

Nếu hai ngắt xảy ra cùng lúc thì NVIC sẽ tiếp nhận thực thi ngắt có mức độ ưu tiên cao hơn (priority value thấp hơn), và đưa ngắt còn lại vào trạng thái chờ - Pending.

Pre-Empt Priority: Khi bộ xử lý đang chạy một trình xử lý ngắt và một ngắt khác xảy ra, thì các giá trị ưu tiên trước – pre-empt priority sẽ được so sánh và ngắt với mức độ pre-empt priority cao hơn (giá trị nhỏ hơn) sẽ được cho phép chạy.

Sub Priority: Giá trị này chỉ được sử dụng khi hai ngắt có cùng giá trị pre-empty priority xảy ra cùng thời điểm. Trong trường hợp này, ngắt có mức ưu tiên phụ cao hơn sẽ được xử lý trước.

**Systick Timer**

Systick Timer là một timer, là một timer độc lập nằm trong lõi Cortex.

SysTick Timer là một Timer đếm ngược 24 bit. Xảy ra ngắt SysTick Interrupt khi thanh ghi conter value về 0, và tự nạp lại giá trị được lưu trong reload value resister.

SysTick Timer bao gồm:

* Một bộ đếm Counter để đếm xuống.
* Một thanh ghi chứa giá trị nạp lại.
* Một cờ báo tràn Timer.
* Một bộ phận để cấu hình xung clock đầu vào.

Ảnh có chứa văn bản, biểu đồ, hàng, ảnh chụp màn hình

Mô tả được tạo tự động

System Timer hay SysTick được điều khiển bởi 4 thanh ghi bao gồm:

* Thanh ghi SysTick Control and Status Resister(SYST\_CSR)
* Thanh ghi SysTick Reload Value (SYST\_RVR)
* Thanh ghi SysTick Current Value (SYST\_CVR)
* Thanh ghi SysTick Calibration Value(SYST\_CALIB)

Trong thanh ghi SysTick Control and Status Resister(SYST\_CSR): Chỉ có 4 bit được sử dụng bao gồm 1 bit status và 3 bit control. Bit status là COUNTFLAG, Bit Control bao gồm: bit select the systick timer clock CLKSOURCE, bit cho phép ngắt TICKINT, bit cho phép counter.

CLKSOURCE = 1 => Chọn processor clock

CLKSOURCE = 0 => Chọn external clock

Khi thanh ghi Counter xuống 0 => COUNTFLAG đặt thành 1.

Khoảng thời gian giữa hai lần ngắt SysTick là:

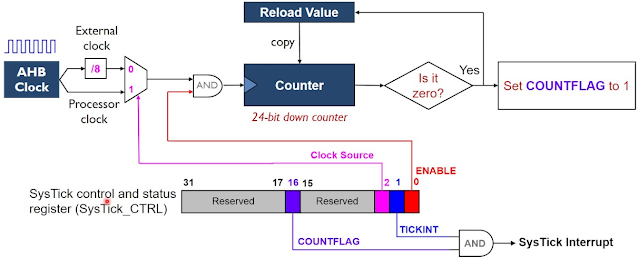
Interval = (RELOAD + 1) \* Source\_Clock\_Period.

Ví dụ: Nếu giữa hai lần ngắt SysTick là 100 chu kỳ Clock thì reload value sẽ là 99.

Ghi giá trị vào thanh ghi Current Value, xoá thanh ghi này và COUNTFLAG xuống 0 => System Timer reload từ thanh ghi reload value trên clock tiếp theo.

Thanh ghi Counter có một giá trị ngẫu nhiêu khi reset => Phải luôn xoá về 0 trước khi enable.

Thanh ghi SysTick Calibration Value(SYST\_CALIB) là thanh ghi only – read.

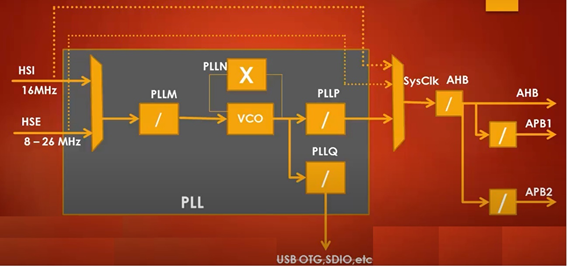


**Clock**

Dao động càng nhanh thì khả năng tính toán và xử lý càng tốt, tuy nhiên, nó sẽ tốn điện hơn => với các dự án đòi hỏi việc tiết kiệm năng lượng, đòi hỏi phải cố gắng giảm tần số hoạt động của con chip xuống tối thiểu.

Các vi điều khiển 32 bit cần phải config clock trước khi sử dụng các ngoại vi. Mặc định, tất cả clock của các thiết bị ngoại vi sẽ bị vô hiệu hóa để tiết kiệm năng lượng.

Đối với VĐK STM32, clock được cấu hình qua thanh ghi RCC.



The system clock (SYSCLK):

* HSI – High Speed Internal: Lấy từ bộ giao động RC nội, mặc định là 16 MHz.
* HSE – High Speed External: Lấy từ dao động thạch anh
* PLL – Phase Lock Loop: bộ nhân tần

2 nguồn clock:

* 40kHz low speed internal RC ( LSI RC) : thực hiện RTC hoặc watchdog để wakeup từ chế độ stop/ standby.
* 32.768kHz low speed external crystal (LSE crystal): real-time clock (RTCCLK).

Các ngoại vi được cấp xung clock thông qua ba đường bus sau:

* AHB (Advanced High Speed Buses ): Đây là Bus kết nối hệ thống.
* APB1, APB2 (Advanced Peripheral Buses 1,2): Đây là các Bus kết nối với thiết bị ngoại vi và kết nối với hệ thống thông qua AHB.

**RTOS**

RTOS - Real-Time Operating system hay hệ điều hành thời gian thực, sử dụng trong những ứng dụng yêu cầu về thời gian đáp ứng nhanh với độ chính xác cao về thời gian.

Với tài nguyên khá nhỏ thì các vi điều khiển sẽ cố gắng thực hiện một số chức năng chính: **tối ưu tối đa số luồng, bộ lập lịch và các tác vụ** (task). Với việc chỉ có một core, nên việc chạy song song các task là không thể. RTOS sẽ có tính năng lập lịch, để việc thực thi các task gần như là song song, và các task sẽ không cần phải đợi quá lâu để được thực thi.

Ảnh có chứa văn bản, ảnh chụp màn hình, hàng, Sơ đồ

Mô tả được tạo tự động

Phân loại RTOS làm 2 loại dựa vào độ trễ:

* Hard Real-time: Hệ thống phải thực hiện task trong khoảng thời gian quy định một cách chính xác, việc không tuân thủ thời gian quy định có thể dẫn đến hậu quả nghiêm trọng.

Ví dụ: Túi khí trên ô tô cần bật lên trong khoảng thời gian nhỏ hơn 0,1s từ khi cảm biến phát hiện xe xảy ra va chạm nguy hiểm. Thì việc hệ thống không đáp ứng thời gian này có thể gây tai nạn nghiêm trọng cho người trên xe. Do đó, task đọc cảm biến và điều khiển túi khí phải yêu cầu là Hard Real-time.

* Soft Real-Time: Có thể không cần đáp ứng gắt gao về mặt thời gian như Hard Real-time, việc đôi khi vị trễ thời gian với các task này có thể không gây ra hậu quả nghiêm trọng.

Chẳng hạn, việc bấm nút thì đèn sáng sau không quá 0,5s. Thì việc bị trễ 1 chút cũng sẽ không ảnh hưởng quá nhiều.

* Có thể nhiều cách phân loại sẽ thêm Firm Real-time nằm giữa 2 loại trên.

Các đặc điểm quan trọng nhất của RTOS, đó là các task, và cơ chế lập lịch - Scheduling.

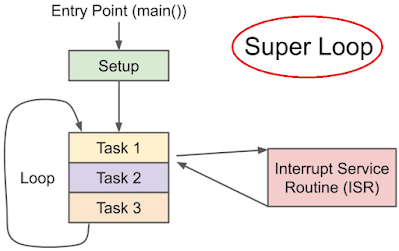
Task là gì? Multi-Task trong RTOS

Đối với vi điều khiển, Task là một tập hợp các câu lệnh, được lưu trong bộ nhớ vi điều khiển, để thực hiện một nhiệm vụ nào đó.

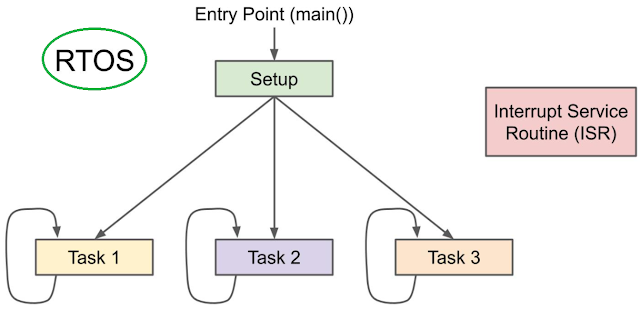
Ví dụ với 3 task:

* Task 1: Nháy LED với chu kỳ 1s.
* Task 2: Đọc giá trị cảm biến và cảnh báo khi cần.
* Task 3: Đọc nút bấm để thay đổi chu kỳ nháy led.

Với chương trình bình thường, ta chạy lần lượt các task trong một vòng while(1) được gọi là Super Loop, và có thể dùng ngắt khi cần.



Còn đối với RTOS, các task này sẽ cần được thực hiện gần như "song song". Vì vậy, mỗi task cần có một "chương trình riêng", ở đây là một function để thực hiện chức năng của chúng. Việc thực hiện đa tác vụ trên cùng một chương trình vi điều khiển (ở đây là 1 core) gọi là Multi-Task.

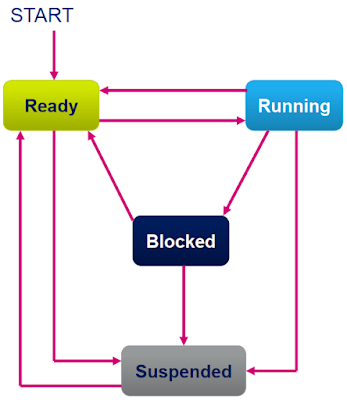


Tóm lại: Task là một đoạn mã hoặc một hàm thực hiện một công việc cụ thể khi nó được phép chạy. Một Task có Stack riêng để tạo các biến cục bộ.

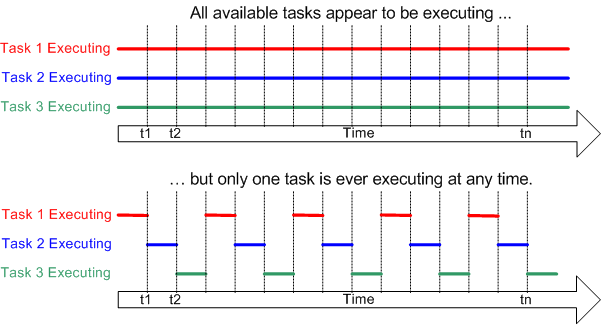
Ở đây do vi điều khiển chỉ có một Core nên tại một thời điểm chỉ có 1 câu lệnh được thực hiện, đồng nghĩa với việc các task không thể thực hiện song song được => cần có một cơ chế để giúp việc thực thi các task gần như là song song, đó là cơ chế lập lịch - Scheduling.

Cơ chế lập lịch - Scheduling

Lập lịch - Scheduling là một thuật toán để xác định Task nào được thực thi. Về cơ bản, một task sẽ có 4 trạng thái chính: READY / RUNNING / BLOCKED / SUSPENDED. Scheduling sẽ quyết định task nào ở trạng thái RUNNING, task nào ở trạng thái SUSPENDED / BLOCKED.



Nhờ có bộ lập lịch mà chúng ta có thể điều khiển các task hoạt động theo yêu cầu, không có task nào bị miss, cũng như việc thực hiện các task sẽ gần như là song song. Hình bên dưới mô tả rất rõ về Multi-Task.



Mặc dù các Task thực hiện tuần tự (như hình dưới) nhưng mắt người nhìn nó như là song song (hình ở trên). Việc các task thực hiện trong một khoảng thời gian rồi ngay lập tức chuyển qua task khác cũng giúp tiết kiệm tài nguyên của hệ thống.

Kernel

Kernel - hay gọi là Nhân của hệ điều hành. Việc lập trình ra các Kernel hay toàn bộ hoạt động của một OS trong vi điều khiển thường được thực hiện bằng ngôn ngữ Assembly, để đảm bảo tốc độ xử lý nhanh nhất của Kernel.

* Kernel sẽ điều phối hoạt động của các Task dựa vào bộ lập lịch - Scheduler và các thuật toán lập lịch.
* Kernel sẽ quản lý tài nguyên phần cứng - bộ nhớ, để lưu trữ hoạt động của các task.
* Kernel quản lý các công việc giao tiếp giữa các task, xử lý ngắt, ...

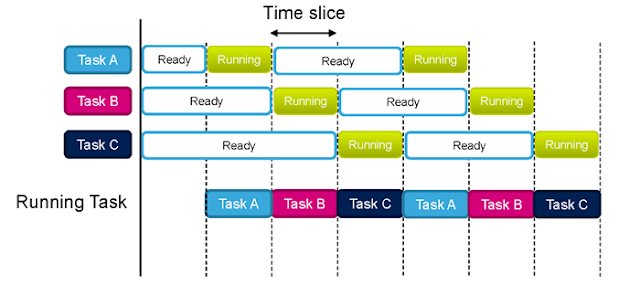
Kernel sẽ gọi bộ lập lịch khi cần biết tiếp theo đến lượt Task nào được thực thi, việc này dựa vào các thuật toán lập lịch. Thuật toán lập lịch sẽ có một list các task đang ở trạng thái READY, cái này giống như một hàng đợi, các task sẽ được phân bổ vào hàng đợi này dựa vào nhiều yếu tố khác nhau (Độ ưu tiên, đến trước hay đến sau, ...).

Chẳng hạn, hệ thống có 3 task 1, 2, 3. Khi task 1 vừa kết thúc, hoặc hết thời gian thực hiện, thì Kernel cần gọi bộ lập lịch để biết tiếp theo task nào sẽ thực hiện.

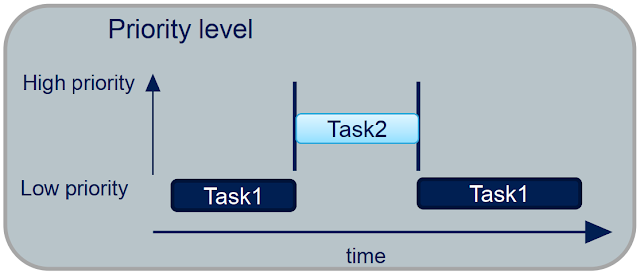
**Round-Robin:** Mỗi task sẽ có một thời gian thực thi nhất định, gọi là Time Slice, hết khoảng thời gian này thì sẽ phải nhường CPU cho task khác thực hiện.

=> dùng một timer, trong stm32 là systick timer để đếm thời gian Time Slide, sau đó sinh ra một ngắt để gọi từng Task thực thi.

=> Nếu 1 task quan trọng cần thực hiện ngay, mà lúc đó một task khác mới bắt đầu chạy, thì task này phải đợi một hoặc nhiều Time Slice mới đến lượt.

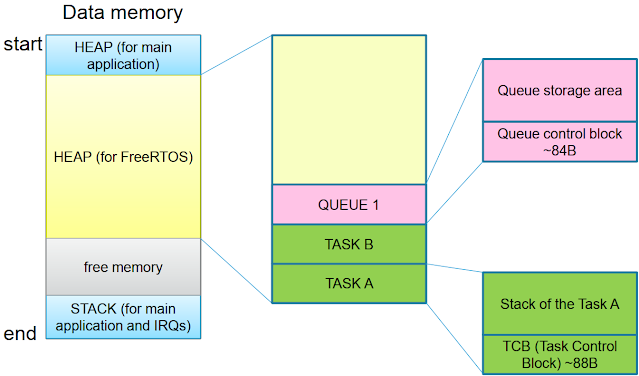


**Preemptive:** Những Task quan trọng sẽ được thực hiện trước, bằng cách gán cho chúng những quyền ưu tiên. Và task nào có quyền ưu tiên cao hơn, sẽ có thể chiếm quyền sử dụng CPU của task đang thực hiện khi cần. Đó chính là cơ chế của Preemptive.



**Memory Management**

Mỗi task là một chương trình riêng biệt, vì vậy chúng cũng cần được cung cấp những vùng nhớ riêng để hoạt động, tránh việc các task bị xung đột bộ nhớ với nhau. Người ta thường sử dụng vùng nhớ Heap để phân bổ bộ nhớ cho các Task.



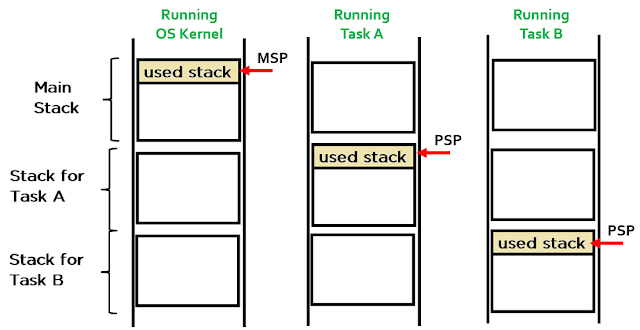
Mỗi task trong RTOS sẽ được cấp một block memory trên Heap. Mỗi task sẽ được chia làm 2 phần:

* TCB (Task Control Block): dùng để điều khiển 1 task, nhiệm vụ chính là lưu trữ lại các context đang thực hiện của một task, trước khi chuyển qua task khác.

Ví dụ, một task đang thực hiện một công việc, chẳng hạn đếm từ 1 đến 100. Nó đang đếm đến 20, thì bỗng dưng bị task có quyền ưu tiên cao hơn chiếm quyền, hoặc bị hết Time-Slice. Điều bắt buộc ở đây là phải có một nơi để nó lưu lại con số 20, sau đó chuyển qua task khác, sau đó khi tiếp tục thực hiện ở lần tiếp theo, nó sẽ load số 20 ra và tiếp tục đếm. Số 20 này sẽ được lưu trong phần TCB.

* Stack: Mỗi task chạy thì đều cần có vùng nhớ dữ liệu để thực thi, ở đây là vùng stack riêng của từng task, khác với vùng nhớ stack của chương trình.

Như vậy, mỗi task hoạt động như một chương trình bình thường, và có cả vùng stack riêng, với stack thì chúng ta cần bố trí một con trỏ ngăn xếp - **Stack Pointer (SP).** Đó chính là một chức năng của SP, cũng là câu trả lời cho câu hỏi tại sao Cortex Mx lại có MSP và PSP. Ở đây chúng ta sẽ dùng **PSP - Process Stack Pointer** cho hoạt động của các Task, và **MSP - Main Stack Pointer** vẫn được sử dụng trong chương trình chính (main).



Có 5 cách cơ bản để cấp phát bộ nhớ cho RTOS:

**Heap\_1:** Đây là cách cấp phát cơ bản nhất và không cho phép giải phóng bộ nhớ. Cách này rất đơn giản là cấp một mảng cố định trên Heap để đặt các Task vào. Mỗi task bao gồm TCB + Stack.

Ưu điểm của phương pháp triển khai này là đơn giản, dễ sử dụng, chỉ cần cấp phát mảng tĩnh cố định trên Heap.

Nhược điểm của nó là không giải phóng bộ nhớ được, nên chỉ dùng với những ứng dụng mà không cần xóa các Task, Semaphore, Mutex.

**Heap\_2:** cho phép giải phóng bộ nhớ, nhưng không đặt các vùng Task cạnh nhau. Cách này có thể gây ra phân mảng bộ nhớ, nếu như không biết trước được kích thước của các Task sử dụng.

**Heap\_3:** Cách này sử dùng các hàm theo thư viện chuẩn C, đó là malloc() và free() để cấp phát và giải phóng bộ nhớ.

**Heap\_4:** Giống như Heap\_2 tuy nhiên cho phép sử dụng lại các vùng Free Space.

**Heap\_5:** Không giống như Heap\_4, Heap\_5 không bị giới hạn trong việc cấp phát bộ nhớ từ một mảng được khai báo tĩnh duy nhất; Heap\_5 có thể cấp phát bộ nhớ từ nhiều không gian bộ nhớ riêng biệt.

Context Switch

Việc một Task đang chạy phải dừng lại để nhường chỗ cho Task khác, sau đó khi trở lại hoạt động thì Task đó vẫn tiếp tục công việc cũ không hề đơn giản => cơ chế Context Switch.

Khi xây dựng việc lập lịch cho RTOS, chúng ta cần có một list các Task cần được thực hiện, cụ thể, để quy định việc Task nào sẽ được thực hiện tiếp theo, chúng ta sẽ xây dựng một Queue - hàng đợi. Task nào đứng đầu Queue sẽ là task tiếp theo cần thực hiện.

Các Task trong Queue này sẽ được gọi thực thi bởi hàm lập lịch - Scheduler(). Hàm lập lịch sẽ gọi trong các trường hợp sau:

* Task có mức độ ưu tiên cao hơn task đang thực thi muốn thực hiện. Khi đó, task này sẽ nhảy lên đầu Queue, task đang thực thi sẽ phải nhường lại quyền điều khiển cho task đó. Lúc này, Kernel sẽ gọi hàm lập lịch Scheduler() để chuyển sang Task có mức ưu tiên cao hơn.
* Khi Task hiện tại gọi hàm vTaskDelay(). Khi gọi hàm này, Task hiện tại sẽ ngưng làm việc trong thời gian delay, vì vậy, để tránh lãng phí tài nguyên hệ thống, Kernel sẽ gọi hàm lập lịch Scheduler() để chuyển qua Task khác.
* Khi hết thời gian Time-Slice, Kernel sẽ chuyển sang task tiếp theo để thực thi bằng cách gọi hàm lập lịch Scheduler().

Việc context switch sinh ra với 2 mục đích:

* Lưu lại context (dữ liệu) của Task đang thực thi trước khi chuyển qua task khác, ngữ cảnh này sẽ được lưu vào vùng nhớ TCB của Task.
* Lấy lại context cũ của Task đang chuẩn bị được thực thi để tiếp tục task đó. Việc này ngược lại với việc trên, đó là lấy dữ liệu từ vùng nhớ TCB của Task tương ứng.

Việc thực thi Context Switch sẽ dựa trên 2 exceptions của hệ thống, đó là: SVC - supervisor call và PendSV Exception.

Queue

Nếu các task muốn giao tiếp với nhau, chúng sẽ làm như thế nào? Dùng biến toàn cục - Global Variable?

Task A và Task B giao tiếp với nhau bằng global\_var1. Task B và Task C giao tiếp với nhau bằng global\_var2. Task C giao tiếp với Task A bằng global\_var3.... Nếu có rất nhiều task, thì chúng ta sẽ phải khai báo rất nhiều biến global (Ở đây 3 Task đã phải sử dụng 3 biến global).

=> khá tốn bộ nhớ nếu có nhiều Task.

Vậy nếu các Task này dùng chung 1 biến global thì sao? Với ví dụ trên nếu Task A đang muốn truyền một giá trị là 5 sang Task B bằng biến Global X. Thì Task B chưa kịp nhận được thì Task C chen ngang và cướp mất biến này, thay đổi nó X = 4, sau đó Task B nhận được giá trị X = 4.

=> sai lệch dữ liệu

=> sử dụng Queue để truyền nhận dữ liệu giữa các Task.

Cơ chế hoạt động của Queue trong RTOS: Khi một Task muốn truyền dữ liệu sang Task khác, nó sẽ đặt dữ liệu này vào Queue, cùng với đó là ID của Task mà nó muốn truyền tới.

RTOS sử dụng 2 loại Queue đó là [Message Queue và Mail Queue](https://www.laptrinhdientu.com/2021/09/Core16.html). Cụ thể, Messege Queue sẽ sử dụng Queue theo kiểu Serial, dữ liệu được sắp xếp theo mảng một chiều. Trong khi Mail Queue sẽ sử dụng Queue theo kiểu Block, mảng đa chiều.

Semaphone

Việc đồng bộ giữa các task là vô cùng quan trọng, đặc biệt là khi có những tài nguyên sử dụng chung. Đồng bộ tức là cơ chế giúp cho các task vẫn hoạt động một cách độc lập, nhưng sử dụng một số tài nguyên chung một cách hiệu quả, không bị conflict. Một vài cơ chế đồng bộ giữa các task thường được sử dụng:

* Semaphore: Sử dụng cho việc đồng bộ hóa tín hiệu và **khả năng tận dụng và chia sẻ tài nguyên.**
* Event Flag: Chỉ ra một hoặc vài sự kiện đã xảy ra, Event Flag giống như mở rộng của Semaphore, trong đó cho phép đồng bộ hóa trên các sự kiện hỗn hợp.
* Mailbox, Queue, Pipe: **Cơ chế truyền dữ liệu giữa các task**.

Semaphore hoạt động giống như **một chiếc chìa khóa** cho việc truy cập tới tài nguyên. Chỉ có task có chìa khóa này mới có quyền sử dụng tài nguyên.

Để có thể sử dụng tài nguyên, tác vụ cần yêu cầu chìa khóa để sử dụng - acquire semarphore. Nếu chìa khóa ở trạng thái sẵn sàng (đang không có task nào sử dụng) thì task yêu cầu có thể sử dụng tài nguyên.

Sau khi dùng xong, task này sẽ phải trả lại chìa khóa - release semaphore để task khác có thể sử dụng => Đây là **cơ chế Binary Seraphore**.

Nhận thấy, binary semaphore dùng để **quản lý đồng bộ 1 hoạt động** duy nhất. Vậy, với việc **đồng bộ nhiều hoạt động**, chúng ta cần một cơ chế khác tổng quát hơn => **Counting Semaphore**.

[Counting Semaphore](https://www.laptrinhdientu.com/2021/10/Core18.html) thường được sử dụng với 2 mục đích:

**Counting Event - Đếm sự kiện**

Semaphore trong trường hợp này giống như một biến đếm, hoạt động theo cơ chế **give - take**.

Mỗi lần có sự kiện xảy ra, một trình xử lý sẽ **give** một semaphore (tương đương với tăng biến đếm lên 1 đơn vị).

Ngược lại, mỗi lần một sự kiện được xử lý xong sẽ **take** một semaphore (tương đương với việc giảm biến đếm đi 1 đơn vị).

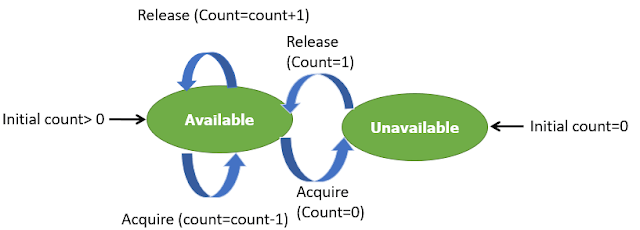
Do đó, giá trị biếm đếm = số sự kiện đã xảy ra - số sự kiện đã xử lý. Mặc định khi khởi tạo, giá trị đếm này là 0 (không có sự kiện nào).

**Resource Management - Quản lý tài nguyên**

Cũng có một giá trị đếm để đếm số lượng các tài nguyên có sẵn. Khi giá trj này bằng 0 thì sẽ không có tài nguyên nào free cả. Mặc định khi khởi tạo, giá trị này bằng MAX (có thể là 255).

Khi một task dùng một tài nguyên, nó sẽ **Acquire** semaphore để giảm biến đếm đi một đơn vị.

Ngược lại, sau khi sử dụng xong nó sẽ **Release** lại để tăng biến đếm lên một đơn vị.



**Mutex**

Race Condition xảy ra khi các task đồng thời truy cập vào một tài nguyên, và các câu lệnh đan xen lẫn nhau, khiến cho chương trình không diễn ra đúng như quy trình đã được thiết kế.

Một số cách để tránh được Race Condition:

* Sử dụng Mutex.
* Sử dụng Critical Section ở mỗi task.

Mutex - Mutial Exclution hay loại trữ lẫn nhau. Thực chất nó là một Flag - cờ, hay một Key - khóa, được sử dụng cho phép hoặc hạn chế quyền truy cập vào một tài nguyên - resource. Tại một thời diểm nó chỉ cho phép một task truy cập vào resource, và nó sẽ chặn tất cả các task khác truy cập vào resource này.

Mutex hoạt động theo cơ chế **acquire** (yêu cầu) - **release** (nhả). Tức là một task sẽ yêu cầu Mutex khi cần sử dụng tài nguyên chung, và sau khi sử dụng xong thì nó sẽ nhả Mutex để cho task khác có thể thực hiện. Nếu Mutex đang được một task sử dụng thì những task khác sẽ phải chờ để nhận được Mutex.

**Có thể thấy Mutex và Binary Semaphore có cách hoạt động tương đối giống nhau, vậy tại sao lại phải tạo ra 2 cơ chế khác nhau?**

Mutex hoạt động theo cơ chế Locking trong khi Binary Semaphore hoạt động theo cơ chế Signaling. Vì vậy, task có ưu tiên cao hơn hoàn toàn có thể giải phóng Binary Semaphore và chiếm quyền sử dụng tài nguyên. Trong khi đó, dù có ưu tiên cao hơn, nhưng task vẫn phải chờ Mutex thực hiện xong.

Binary Semaphore có thể đồng thời được sở hữu bởi nhiều thread trong khi Mutex thì chỉ được sở hữu bởi một thread duy nhất.

Trên thực tế, Binary Semaphore sẽ thực hiện nhanh hơn Mutex vì các thread/process khác có thể giải phóng Semaphore, trong khi chỉ có thread/process mới có thể unlock Mutex.

Mutex thường được sử dụng trong trường hợp có một tài nguyên duy nhất, ví dụ như Race Condition. Trong khi Semaphore có thể sử dụng trong trường hợp có nhiều tài nguyên hơn.

<https://electrovolt.ir/wp-content/uploads/2022/09/MISRA-C_2012_-Guidelines-for-the-Use-of-the-C-Language-in-Critical-Systems-Motor-Industry-Research-Association-2013-2013.pdf>