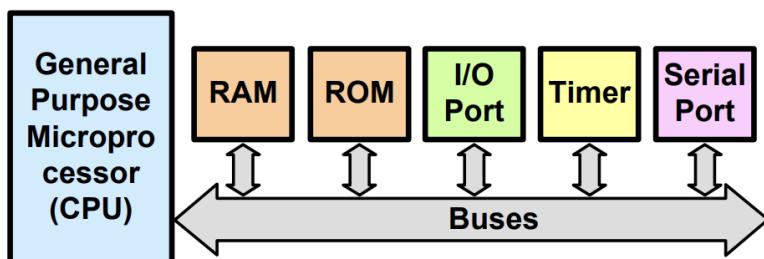


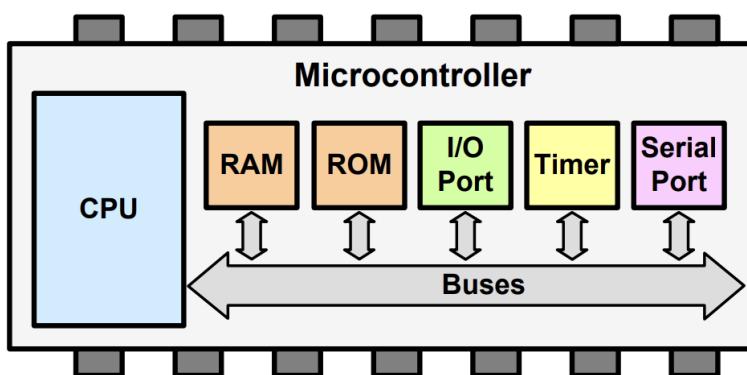
## AFTERC - EMBEDDED C

### 1. Nền tảng Vi điều khiển

#### 1.1. Microprocessor vs Microcontroller – Vi xử lý và Vi điều khiển



Vi xử lý là một bộ điều khiển máy tính có hiệu năng mạnh mẽ được tích hợp vào một IC duy nhất. Vi xử lý chỉ thực hiện các hoạt động tính toán logic như bộ ALU (Arithmetic Logical Unit) và giao tiếp với các ngoại vi khác được đặt bên ngoài vi xử lý. Đặc trưng của vi xử lý là mạnh về tính toán, xử lý dung lượng dữ liệu lớn, giá thành cao.



Vi điều khiển là một tập hợp các thành phần của một máy tính nhỏ, gồm bộ xử lý trung tâm và các thành phần ngoại vi được tích hợp vào trên một IC duy nhất. Nhằm tạo nên một phần cứng có khả năng tích hợp cao, dễ giao tiếp và dễ lập trình theo hướng điều khiển. Đặc trưng của vi điều khiển là hiệu năng tương đối, khả năng giao tiếp điều khiển cao, giá thành rẻ.

## 1.2. Tổng quan về Vi điều khiển hiện nay

### 1.2.1. Các loại Vi điều khiển hiện nay

Họ vi điều khiển AMCC (do tập đoàn "Applied Micro Circuits Corporation" sản xuất). Từ tháng 5 năm 2004, họ vi điều khiển này được phát triển và tung ra thị trường bởi IBM. Các vi điều khiển của họ AMCC sau đây đều thuộc PowerPC 400 family (32-bit RISC):

- 403 PowerPC CPU
- PPC 403GCX
- 405 PowerPC CPU
- PPC 405EP
- PPC 405GP/CR
- PPC 405GPr
- PPC NPe405H/L
- 440 PowerPC Book-E CPU
- PPC 440GP
- PPC 440GX
- PPC 440EP/EPx/GRx
- PPC 440SP/SPe

Họ vi điều khiển Atmel:

- Dòng 8051 (8031, 8051, 8751, 8951, 8032, 8052, 8752, 8952) (8-bit 8051/MCS51)
- Dòng Atmel AT91 (16/32-bit ARM THUMB)
- Dòng AT90, Tiny & Mega – AVR (Atmel Norway design) (8-bit AVR RISC-Based)
- Dòng Atmel AT89 (8-bit 8051/MCS51)
- Dòng MARC4 (4-bit Harvard)

Họ vi điều khiển Freescale Semiconductor. Từ năm 2004, những vi điều khiển này được phát triển và tung ra thị trường bởi Motorola:

- 68HC05 (CPU05) (8-bit Von Neumann)
- 68HC08 (CPU08) (8-bit Von Neumann)
- 68HC11 (CPU11) (8-bit Freescale 68HC11: 8-bit data and 16-bit address)
- 68HC12 (CPU12) (16-bit Enhancement of Freescale 68HC11)
- 68HC16 (CPU16) (16-bit Freescale 68HC11)
- Freescale DSP56800 (DSPcontroller) (16-bit DSP56800E)
- Freescale 683XX (CPU32) (32-bit 68000)
- MPC500 (32-bit RISC)
- MPC 860 (32-bit PowerQUICC)

- 
- MPC 8240/8250 (32-bit PowerQUICC II)
  - MPC 8540/8555/8560 (32-bit PowerQUICC III)

Họ vi điều khiển Fujitsu:

- F<sup>2</sup>MC Family (8/16-bit Core Sub-Architecture: F<sup>2</sup>MC)
- FR Family (32-bit RISC)
- FR-V Family (32 bit RISC)

Họ vi điều khiển Intel:

- 8XC42 (8-bit 8051/MCS51)
- MCS48 (8-bit modified Harvard)
- MCS51 (8-bit 8051/MCS51)
- 8061 (8-bit 8051/MCS51)
- 8xC251 (8-bit 8051/MCS51)
- MCS96 (16-bit MCS-96)
- MXS296 (16-bit MCS-96)
- i960 (32-bit RISC)

Họ vi điều khiển Microchip:

- Instruction 12-bit (Base-line): PIC10F, PIC12F và một vài PIC16F (8-bit RISC)
- Instruction 14-bit (Mid-Range và Enhance Mid-Range): PIC16Fxxx, PIC16F1xxx (8-bit RISC)
- Instruction 16-bit (High Performance): PIC18F (8-bit RISC)
- PIC điều khiển động cơ: dsPIC30F (16-bit Modified Harvard)
- PIC có DSC: dsPIC33F (16-bit Modified Harvard)
- Phổ thông: PIC24F, PIC24E, PIC24H (16-bit Modified Harvard)
- PIC32MX (32-bit M4K – Harvard)

Họ vi điều khiển National Semiconductor:

- COP8 (8-bit CISC)
- CR16 (16-bit RISC)

Họ vi điều khiển STMicroelectronics:

- ST 62 (8-bit Harvard)
- ST7 (16-bit von Neumann)
- STM8 (8-bit Harvard)
- STM32 (32-bit ARM)

Họ vi điều khiển Philips Semiconductors

- 
- LPC2000 (32-bit ARM)
  - LPC900 (8-bit 80C51)
  - LPC700 (32-bit ARM)

Họ vi điều khiển Texas Instruments:

- MSPM0G (32-bit ARM)
- MSPM0L (32-bit ARM)
- MSP430 (16-bit RISC)
- AM24x (32-bit ARM)
- AM26x (32-bit ARM)
- F283x (32-bit Harvard)
- F2800x (32-bit Harvard)
- AM27x (32-bit Harvard)

### 1.2.2. Phân loại Vi điều khiển

Phân loại theo số bit cấu tạo: số bit của Vi điều khiển được xác định là số bit của thanh ghi, độ dài tập lệnh, số bit bộ ALU trong CPU xử lý được, số bit của databus hoặc addressbus.

Số bit cấu tạo Vi điều khiển	Dòng Vi điều khiển
<b>4-bit</b>	Am2900, MARC4, S1C6x, TLCS-47, TMS1000, μCOM-4
<b>8-bit</b>	6800 (68HC05, 68HC08, 68HC11, S08, RS08) 6502 (65C134, 65C265, MELPS 740) 78K, 8048, 8051(XC800) AVR, COP8, H8, PIC10/12/16/17/18, ST6/ST7, STM8, Z8, Z80(eZ80, Rabbit 2000, TLCS-870)
<b>16-bit</b>	68HC12/16, C166, CR16/C, H8S, MSP430, PIC24/dsPIC, R8C, RL78, TLCS-900, Z8000
<b>32-bit</b>	Am29000, ARM/Cortex-M (EFM32, LPC, SAM, STM32, XMC) AVR32, CRX, FR(FR-V), H8SX, M32R, 68000(ColdFire), PIC32, PowerPC(MPC5xx), Propeller, SuperH, TLCS-900, TriCore, V850RX, Z80000
<b>64-bit</b>	PowerPC64

Phân loại theo kiến trúc phần cứng:

- Kiến trúc Harvard và kiến trúc Von-Neumann: (*đây là hai kiến trúc VDK cơ bản*)
  - Harvard: kiến trúc có instruction bus và data bus riêng biệt. Kiến trúc Harvard sử dụng hai địa chỉ vật lý của data và instruction riêng biệt để truy cập đến và lưu trữ. Hầu hết các vi điều khiển ngày nay được xây dựng dựa trên kiến trúc Harvard, kiến trúc này định nghĩa bốn thành phần cần thiết của một hệ thống nhúng bao gồm: CPU core, bộ nhớ chương trình (thông thường là ROM hoặc bộ nhớ flash), bộ nhớ dữ liệu (RAM), một hoặc vài bộ định thời và các cổng I/O để giao tiếp với các thiết bị ngoại vi và các môi trường bên ngoài - tất cả các khối này được thiết kế trong một IC.

- 
- Von-Neumann: chỉ có một bus duy nhất dùng chung cho instruction và data.  
Chỉ sử dụng một địa chỉ vật lý duy nhất để truy cập và lưu trữ cho cả instruction và data.
  - Kiến trúc Vi điều khiển hiện nay: 68000, 8051, ARC, ARM, AVR, CISC, MIPS, PIC, RISC, RISC-V. Ngoài ra còn có các phiên bản khác của các kiến trúc này.

### 1.3. Vi điều khiển kiến trúc ARM

#### 1.3.1. Tổng quan kiến trúc ARM

ARM – Advance RISC Machine là kiến trúc vi điều khiển được phát triển lên từ RISC – Reduced Instruction Set Computer, đây là kiến trúc máy tính được thiết kế để đơn giản hóa các instruction (tập lệnh), trong đó thời gian thực thi tất cả các lệnh đều như nhau. Mục tiêu của RISC chính là đơn giản hóa các tập lệnh, để mỗi lệnh có thể được thực thi chỉ trong 1 chu kỳ máy.

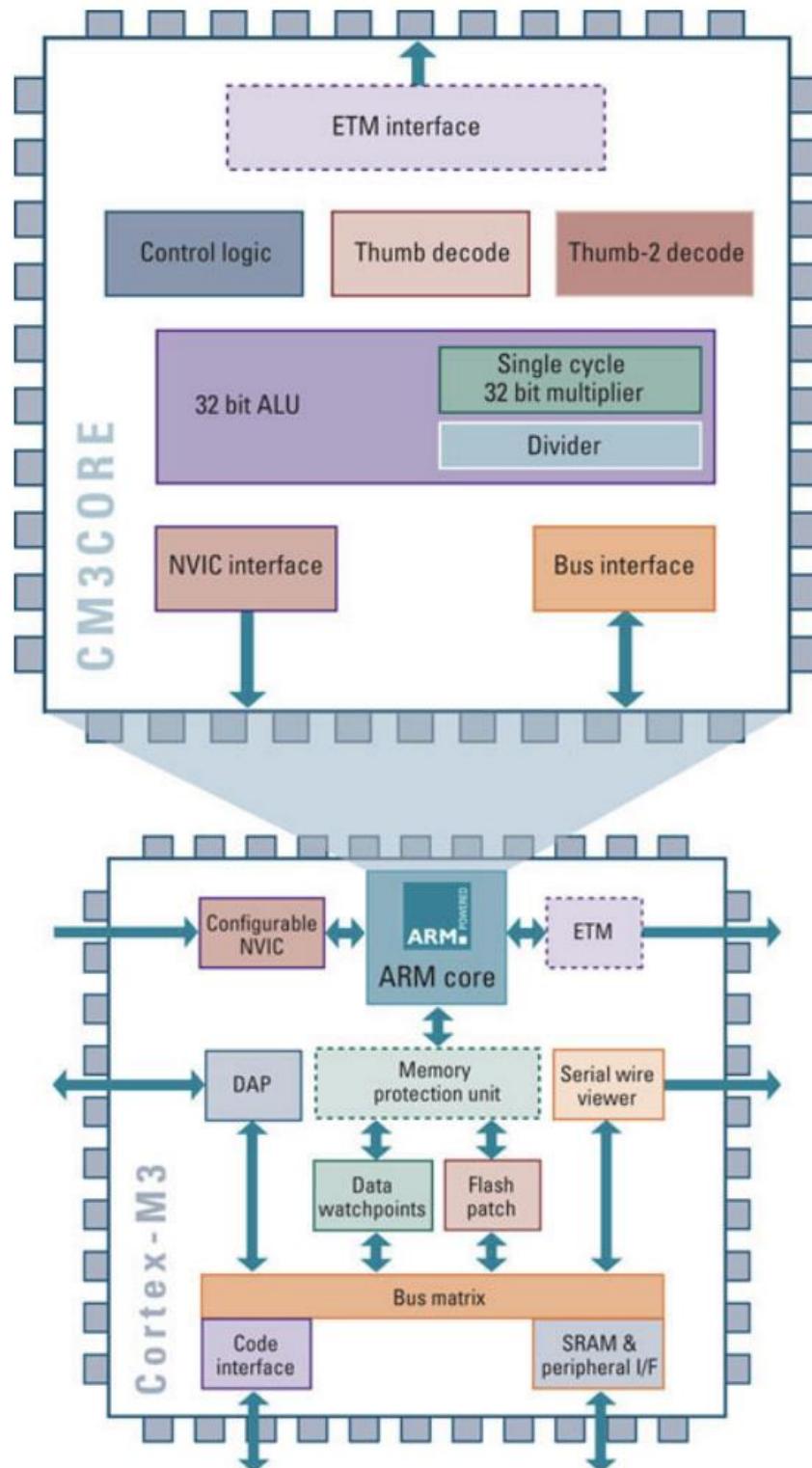
Công ty Arm không trực tiếp sản xuất ra vi điều khiển mà chỉ thiết kế và bán license (bản quyền bản thiết kế) của kiến trúc Vi điều khiển ARM cho các công ty sản xuất Vi điều khiển khác. Hiện nay có hơn 200 công ty đã mua kiến trúc ARM và cung cấp ra thị trường các Vi điều khiển ARM của chính họ.



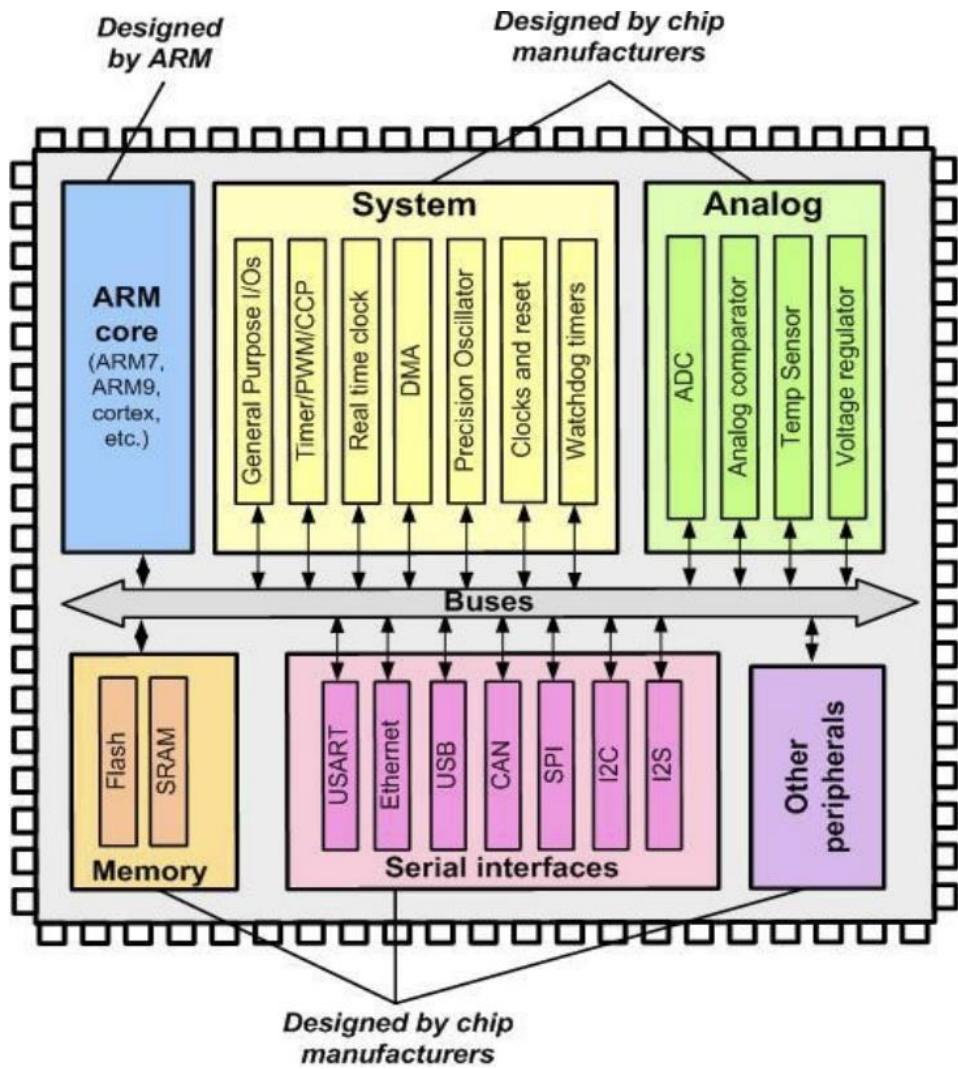
Hiện nay các Vi điều khiển ARM sẽ được build dựa trên một phiên bản được phát triển từ kiến trúc ARM gốc (ARM based). Tập hợp các phiên bản của kiến trúc ARM được phân ra thành các family.

	Application Cortex Processors		Cortex-A15
	Embedded Cortex Processors		Cortex-A9
	Classic ARM Processors		Cortex-A8
CORE		ARM11MP	Cortex-A5
		ARM176JZ	Cortex-M7
	ARM926	Cortex-R7	SC300
	ASC100	Cortex-R5	Cortex-M4
	ARM968	ARM1136J	Cortex-M1
	ARM7TDMI	ARM1156T2	Cortex-R4
	ARM946		Cortex-M3
			Cortex-M0
Family	ARM7TDMI	ARM9E	ARM11
Architecture Version	ARMv4T	ARMv5TJ	ARMv6
			ARMv7A/R
			ARMv7M/ME
			ARMv8M

Vi điều khiển ARM sử dụng kiến trúc Cortex-M:



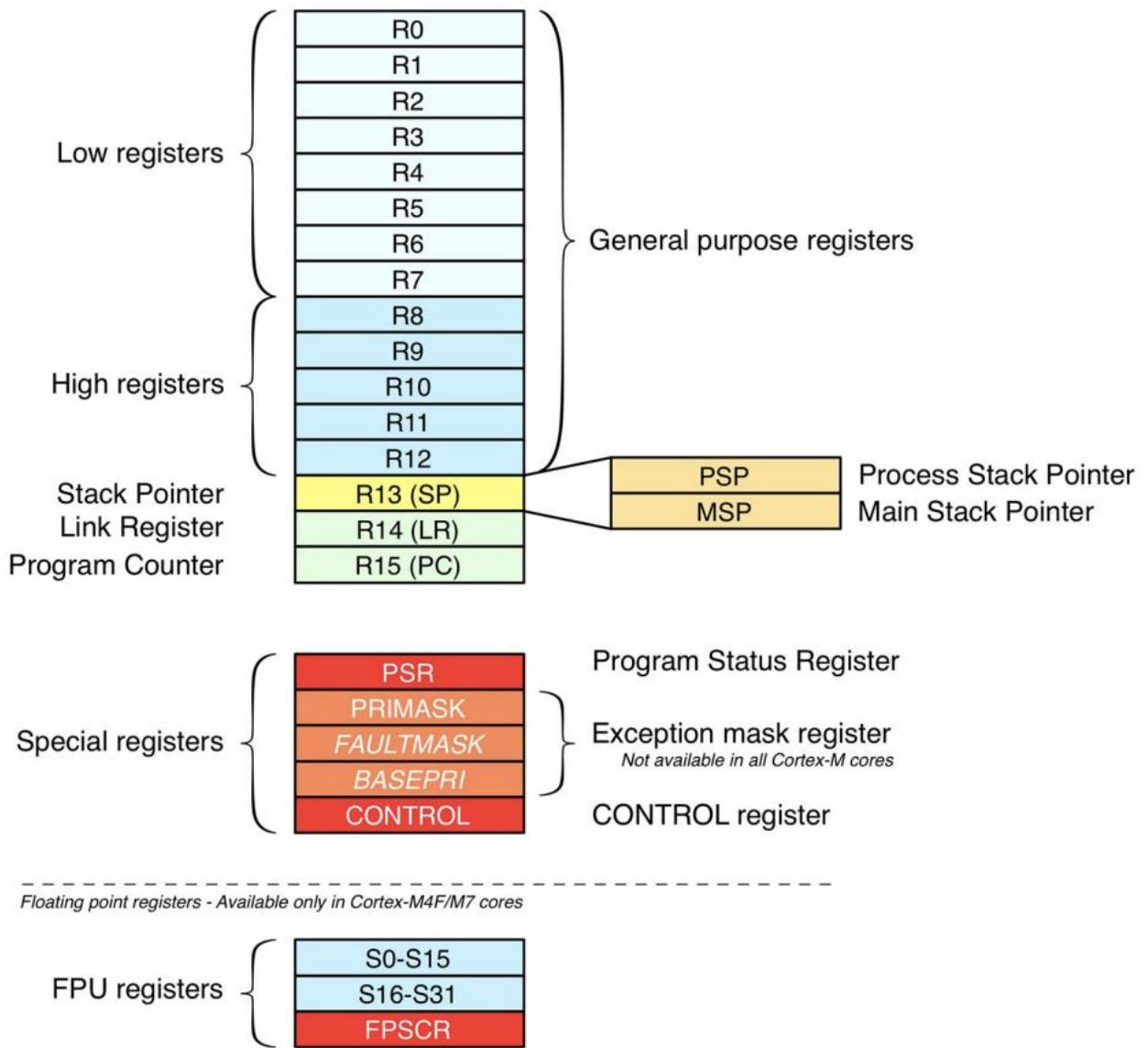
### 1.3.2. Các thành phần của ARM Cortex-M Based microcontroller



Vi điều khiển ARM Cortex-M Based gồm có những thành phần chính sau: ARM Core (Gồm Các Core Register, bộ ALU), Memory, Peripherals, Bus dữ liệu và Bus địa chỉ cho ARM Core giao tiếp với các thành phần bên ngoài.

## Core Register

Bộ xử lý Cortex-M là load/store machine và chỉ thực hiện các hoạt động này trên các thanh ghi của CPU.



**R0-R12** là các general-purpose register dùng để làm toán hạng cho các lệnh hợp ngữ ARM.

**R13** là thanh ghi Stack Pointer (**SP**), trỏ đến vùng nhớ stack (stack memory), dùng cho việc gọi hàm trong chương trình hoặc trong suốt quá trình ngắt hoặc xử lý các ngoại lệ, ngoài ra còn dùng cho việc lưu trữ trạng thái trước đó được thực thi tùy theo CPU mode hiện tại (context switching mode in RTOS) hoặc trỏ vào vùng thực thi code để chuyển lại context khi hoàn thành xong một phần code.

Ví dụ:

```
uint8_t a,b,c;

a = 3;
b = 2;
c = a * b;
```

Trình biên dịch arm-none-eabi-gcc cho đối tượng -mcpu=cortex-m4 sẽ sử dụng các thanh ghi R0-R12 để thành các toán hạng như sau:

```
1  movs    r3, #3      ;move "3" in register r3
2  strb    r3, [r7, #7] ;store the content of r3 in "a"
3  movs    r3, #2      ;move "2" in register r3
4  strb    r3, [r7, #6] ;store the content of r3 in "b"
5  ldrb    r2, [r7, #7] ;load the content of "a" in r2
6  ldrb    r3, [r7, #6] ;load the content of "b" in r3
7  smulbb r3, r2, r3   ;multiply "a" with "b" and store result in r3
8  strb    r3, [r7, #5] ;store the result in "c"
```

Ta có thể thấy, tất cả các hoạt động đều liên quan đến một thanh ghi core. Dòng 1 sẽ lưu giá trị 3 và thanh ghi R3, sau số lưu giá trị trong R3 vào địa chỉ bộ nhớ được lưu trong R7 cộng thêm offset là 7 địa chỉ ô nhớ (R7 lúc này được gọi là frame pointer lưu một địa chỉ của ô nhớ mà CPU quản lý).

**R14 (LR)** – Link Register: là thanh ghi chứa địa chỉ trả về của các lệnh rẽ nhánh (ví dụ như khi gọi 1 hàm thì LR sẽ chứa địa chỉ của lệnh tiếp theo sau khi hàm đó trả về).

**R15 (PC)** – Program Counter: Thanh ghi chứa địa chỉ của lệnh hợp ngữ tiếp theo trong vùng chứa code để fetch → decode → execute.

### Caller

```
void func1(void)
```

```
{
```

```
  uint8_t a, b, c;
```

```
  func2(&b, &c);
```

```
  a = b + c;
```

LR = address of next instruction

### Callee

```
void func2(uint8_t *b, uint8_t *c)
```

```
{
```

```
  *b = 3;
```

```
  *c = 2;
```

PC = LR

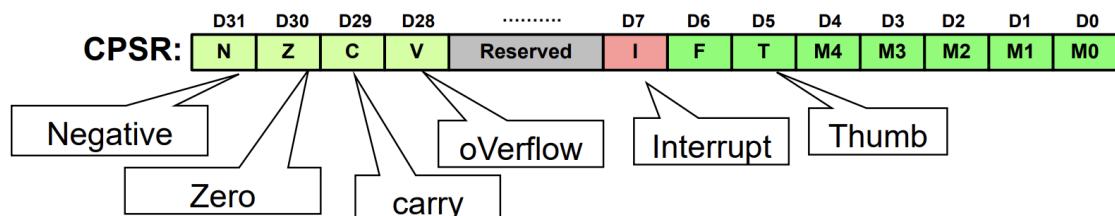
```
}
```

PC jumps back to resume func1

Các thanh ghi đặc biệt:

- **PSR** – Program Status Register: Chứa các cờ trạng thái của chương trình.

Ví dụ: *Thanh ghi PSR của STM32 là CPSR – Control Program Status Register:*



- **PRIMASK, FAULTMASK, BASEPRI** – Eception mask register: chưa gấp.
- **CONTROL** – Control Register: chưa gấp.
- **S0 – S15, S16 – S31, FPSCR** – Floating Point Register: có trên Cortex-M4/M7 Core.

---

## Memory

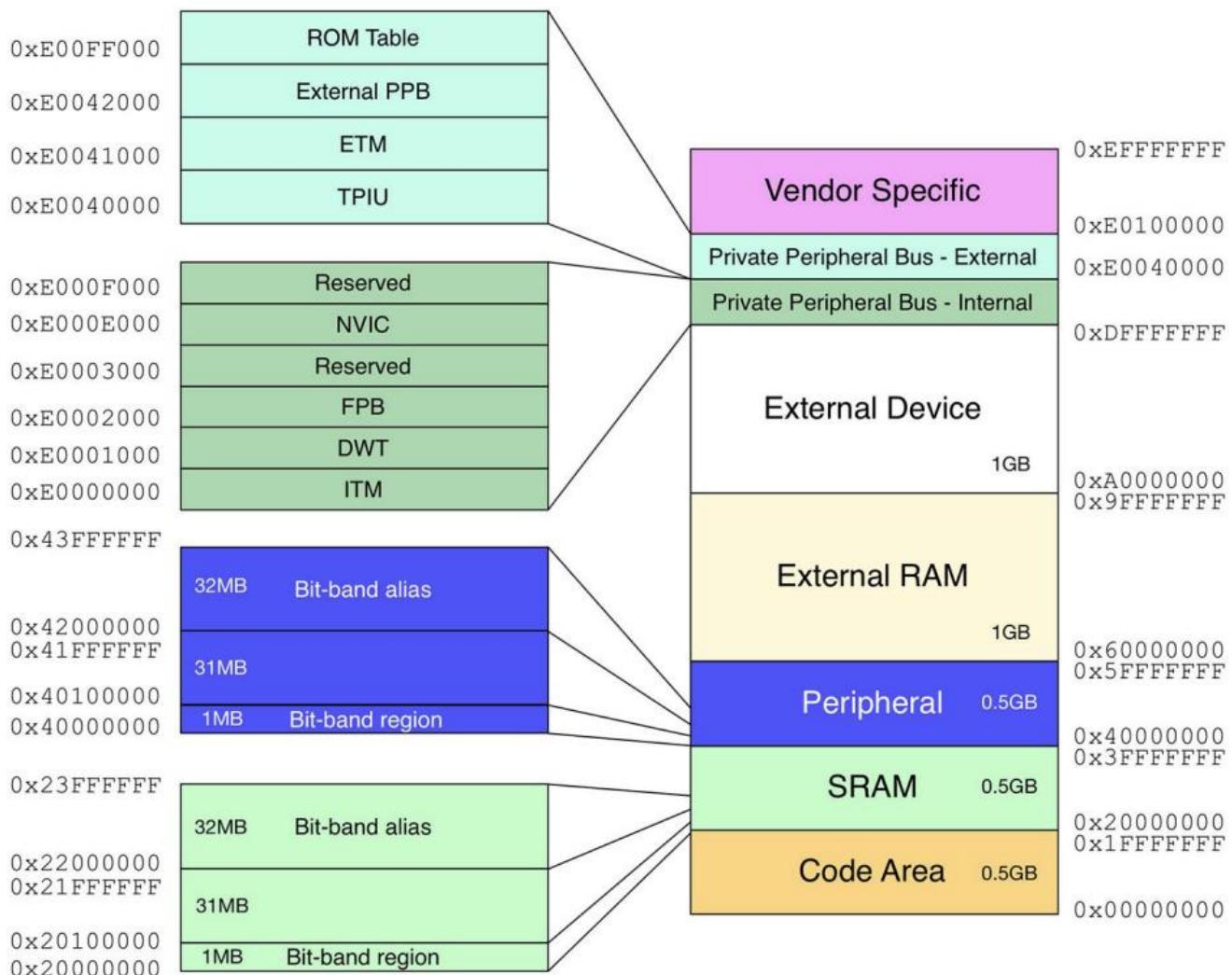
Phân loại và đặc điểm của từng loại Memory: <https://embedded.c/topic/memory.md> at [https://github.com/Hnit3003/embedded\\_c](https://github.com/Hnit3003/embedded_c)

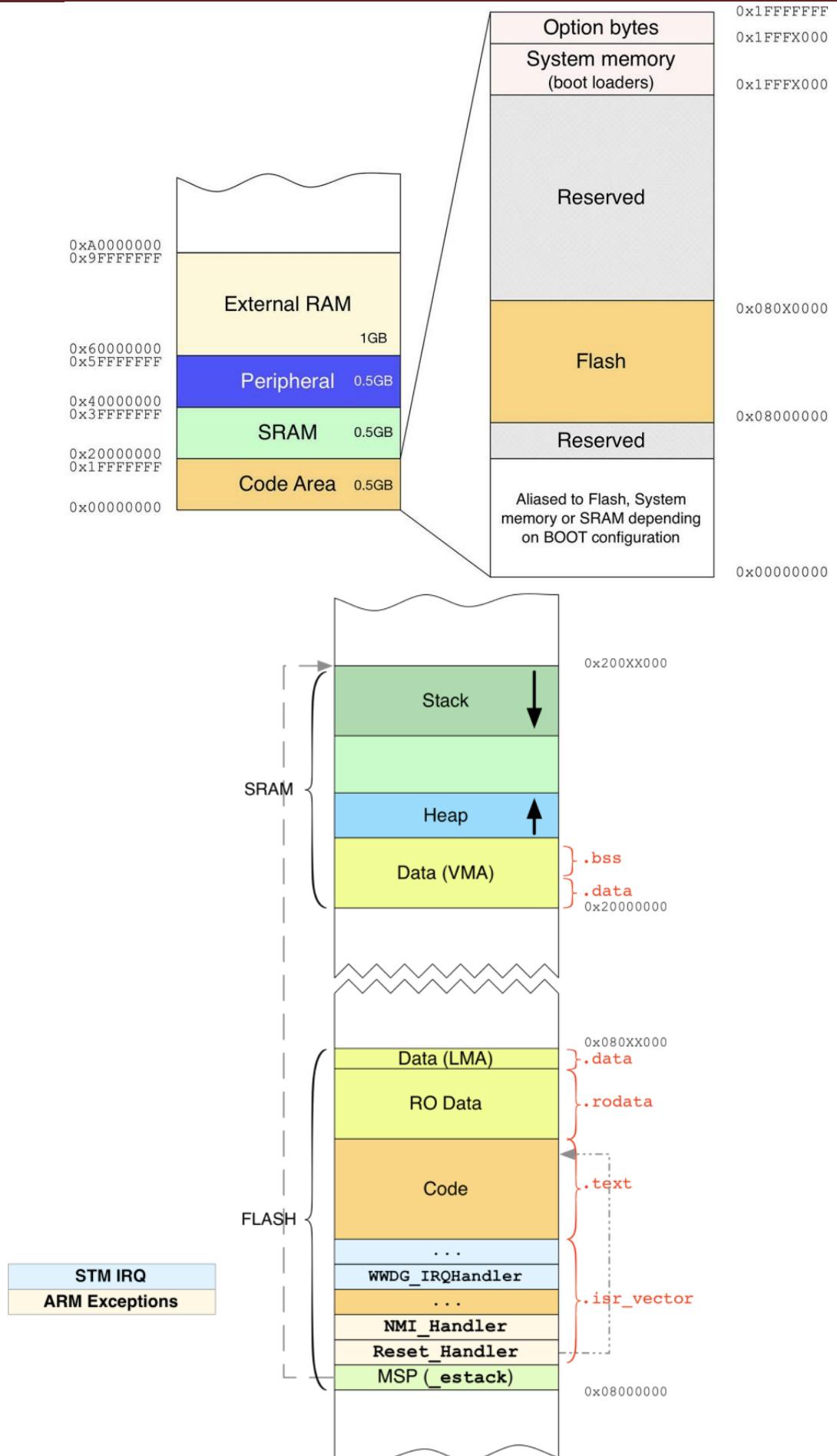
Vi điều khiển ARM Cortex-M sử dụng hai loại memory chính:

- FLASH: chứa code thực thi và các dữ liệu không thay đổi.
- SRAM: tạo các segment để thực thi code.

## Memory map

ARM xác định một không gian địa chỉ bộ nhớ được tiêu chuẩn hóa chung cho tất cả các lõi Cortex-M. Không gian địa chỉ rộng 4GB được chia thành nhiều phân vùng nhỏ tương ứng với một chức năng đặc biệt của từng vùng này. *Lưu ý: do bus địa chỉ có 32-bit  $\rightarrow 2^{2.2^{30}}$  địa chỉ được quản lý. Mà mỗi địa chỉ trỏ tới 1 byte vùng nhớ nên dung lượng là không gian địa chỉ là 4GB. (GiB - 1024 và GB - 1000)*





## Peripheral

Peripherals – Các ngoại vi mà Vi xử lý ARM Cortex-M có thể truy xuất đến thông qua Memory map (512MB Peripheral) và điều khiển theo lệnh của vùng chứa code.

Các ngoại vi có thể được tích hợp trong Vi điều khiển:

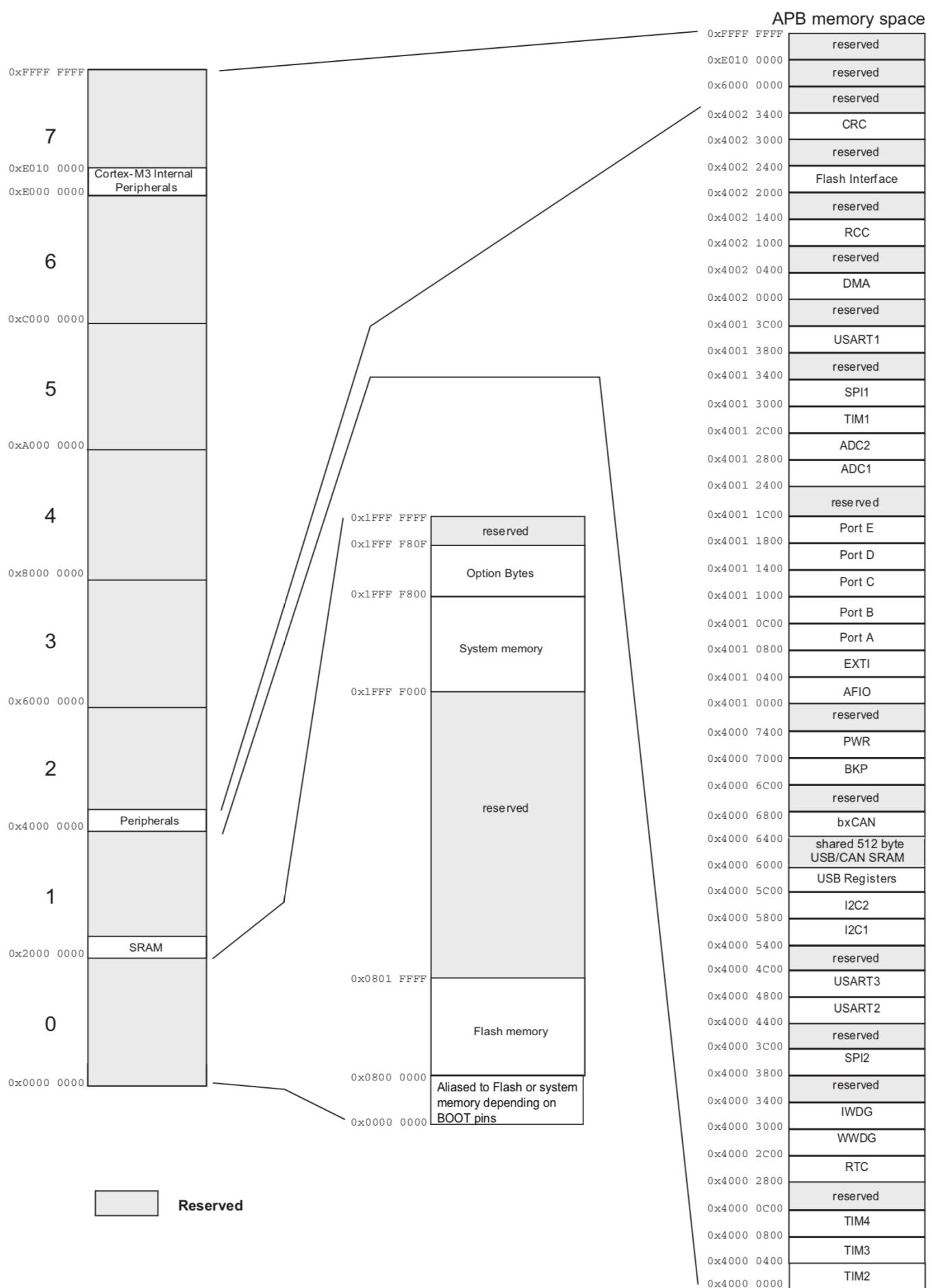
- GPIO.
- Timer.
- RTC.
- ADC.
- UART.
- Ethernet.
- USB.
- CAN.
- SPI.
- I<sup>2</sup>C.
- I<sup>2</sup>S.
- Interrupt.
- DMA.
- ...

#### 1.4. Memory map của STM32F103C8T6

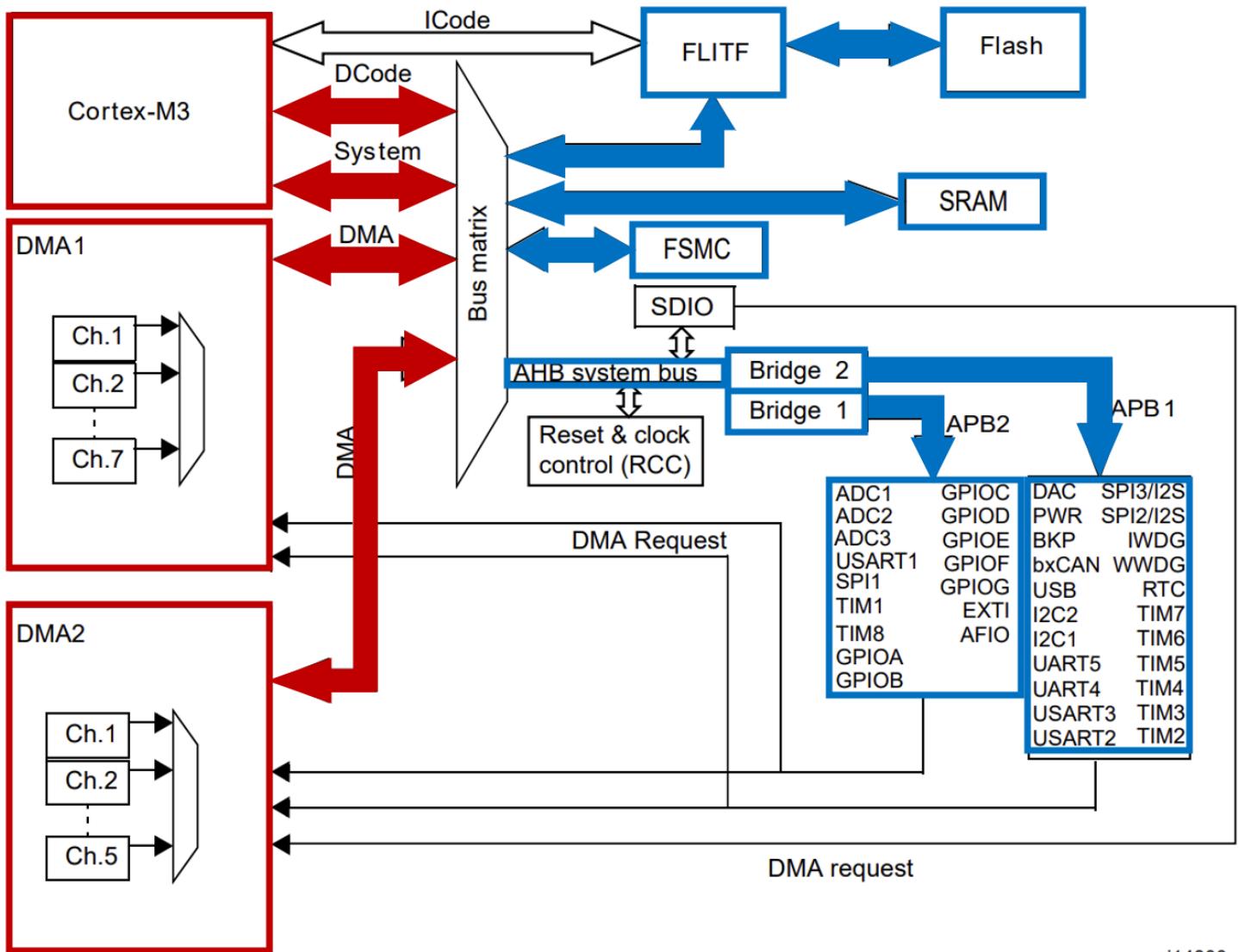
Memory map được định nghĩa theo chuẩn của ARM Cortex-M Based Microcontroller có không gian địa chỉ rộng 4GB, có nghĩa là Cortex-M CPU có thể quản lý tối đa được  $2^{2 \cdot 230}$  địa chỉ, nhưng thực tế ta chỉ cần một vùng rất nhỏ trong không gian này để lưu trữ.

STM32F1 có memory map như sau:

- 0x0000 0000 – 0x0800 0000 (128MB): lưu địa chỉ đầu tiên được nạp vào PC và SP khi khởi động.
- 0x0800 0000 – 0x0801 0000 (64KB): FLASH memory – lưu code thực thi chương trình và các dữ liệu ít bị thay đổi.
- 0x1FFF F000 – 0x1FFF F800 (2KB): system memory – chứa dữ liệu của hệ thống bao gồm chương trình bootloader của nhà sản xuất.
- 0x1FFF F800 - 0x1FFF F80F (16-Bytes): option bytes – chứa các cờ liên quan đến quá trình nạp code nhưng chưa xem rõ là gì.
- 0x2000 0000 – 0x2000 5000 (20KB): SRAM – chứa dữ liệu trong lúc thực thi code.
- 0x4000 0000 – 0x4002 3400 (141KB): peripheral memory.
- 0xE000 0000 – 0xE010 0000 (1MB): Cortex-M3 Internal Peripheral.



### 1.5. Kiến trúc hệ thống của Vi điều khiển STM32F103 (xem lại)



4 Masters:

- Cortex®-M3 core: DCode bus (D-bus) và System bus (S-bus).
- GP-DMA1 & 2 (general-purpose DMA).

4 Slaves:

- Internal SRAM.
- Internal FLASH (qua FLITF – FLASH Interface).
- FSMC – Flexible Static Memory Controller.
- AHB – Advanced High-performance Bus
  - AHB1 – Advanced Peripheral Bus 1.
  - AHB2 – Advanced Peripheral Bus 2.

**Data Code Bus:** bus giao tiếp dữ liệu giữa Cortex-M3 và các memory bên ngoài core.

**System Bus:** bus dành riêng cho Cortex-M3 truy cập đến SRAM và peripheral.

---

**DMA bus:** bus trao đổi dữ liệu giữa ngoại vi và memory hoặc giữa các memory với nhau.

**AHB/APB bridges (APB):** tạo liên kết đồng bộ giữa AHB và 2 bus APB1/2. APB1 có tốc độ tối đa 36 MHz, APB2 lên đến 72 MHz. Sau khi reset, tất cả các clock ngoại vi đều bị tắt (ngoại trừ SRAM và FLITF). Trước khi sử dụng thiết bị ngoại vi, phải bật clock của nó trong thanh ghi RCC\_AHBENR, RCC\_APB2ENR hoặc RCC\_APB1ENR.

**Bus matrix:** phân xử giữa các masters bus.

## 1.6. STM32 Booting Process

**Bước 1:** Vi điều khiển đọc giá trị ở hai chân BOOT0 và BOOT1 để xác định boot mode

STM32 có hai chân BOOT0 và BOOT1 dùng để xác định chế độ boot khi khởi động  
Vi điều khiển, cụ thể xét mức logic ở bảng sau:

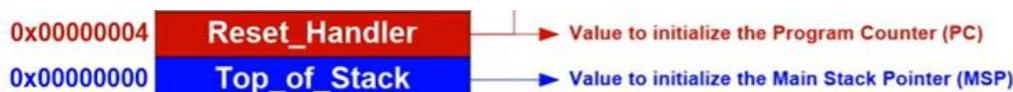
Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
x	0	Main Flash memory	Main Flash memory is selected as boot space
0	1	System memory	System memory is selected as boot space
1	1	Embedded SRAM	Embedded SRAM is selected as boot space

**Bước 2:** Fetch SP – Stack Pointer từ địa chỉ 0x0000 0000

Con trỏ Stack – Stack Pointer là Core Register R13 trong ARM Core sẽ được nạp địa chỉ của đỉnh của Stack Segment trong SRAM.

**Bước 3:** Fetch PC – Program Counter từ địa chỉ 0x0000 0004

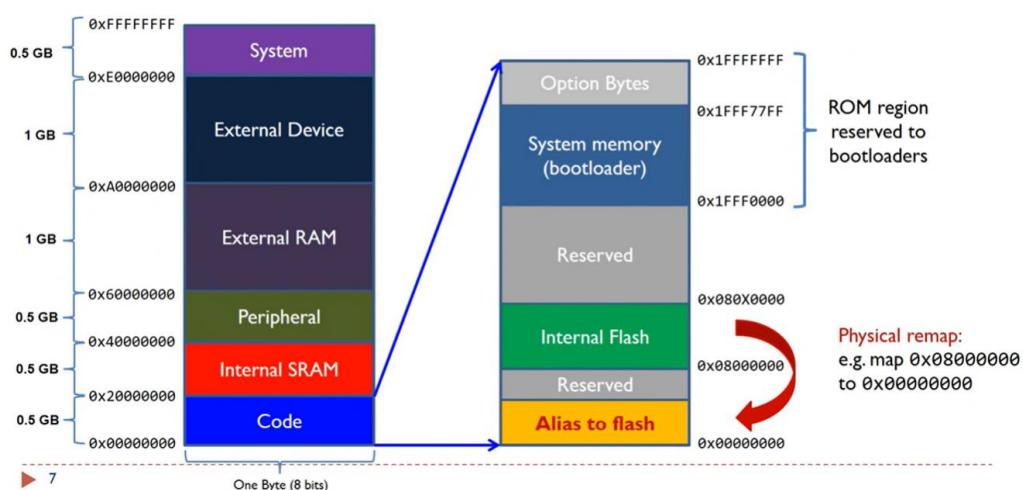
Thanh ghi PC là Core Register R15 trong ARM Core sẽ được nạp địa chỉ ô nhớ chứa mã hợp ngữ được bắt đầu boot.



Tại sao lại chia ra boot mode

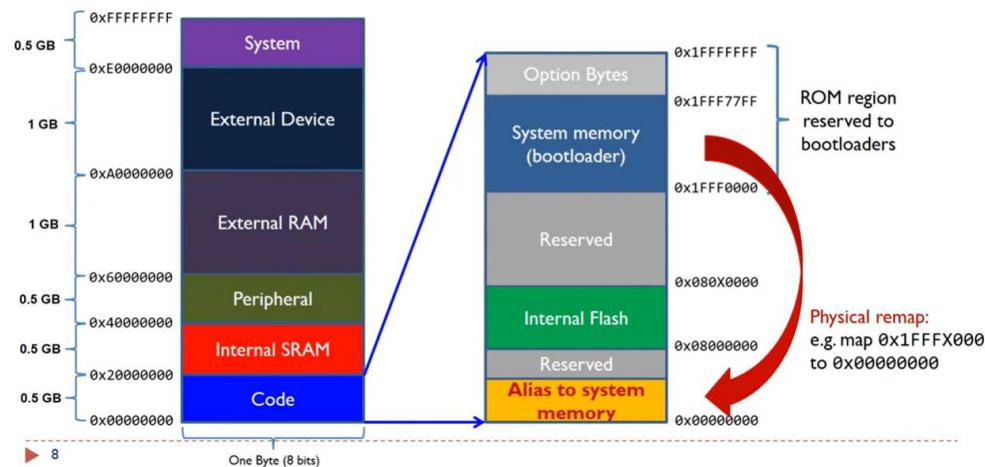
Địa chỉ được nạp vào PC sẽ tùy thuộc vào Boot mode ở Bước 1 đã xác định được:  
Boot từ FLASH memory: application code được lưu trong FLASH sẽ được chạy sau khi reset.

### Boot from main FLASH memory



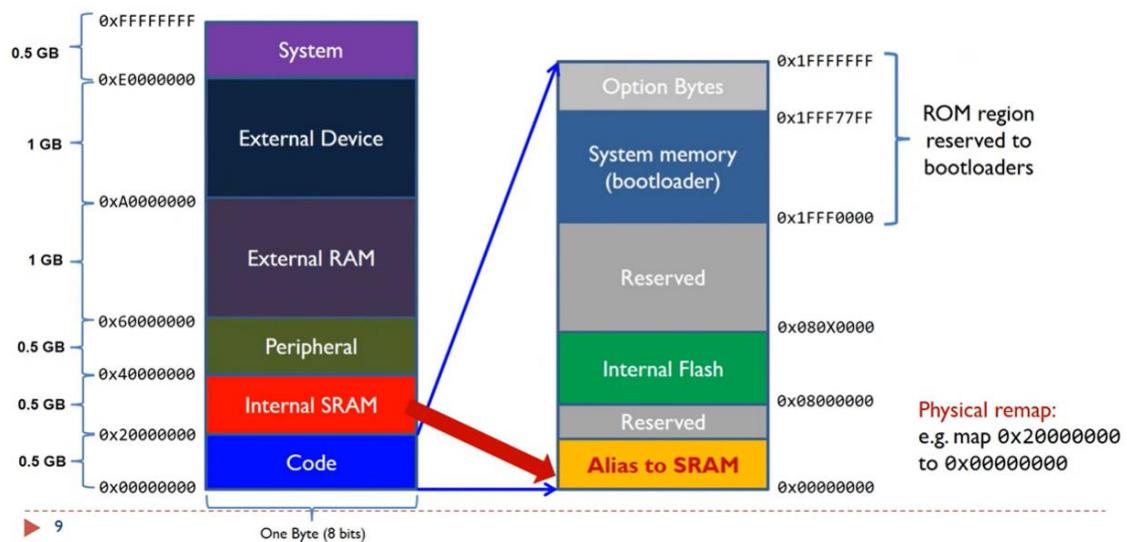
Boot từ system memory: sau khi reset, chương trình bootloader trong system memory sẽ được chạy, giúp code được nạp vào FLASH bằng các chuẩn giao tiếp ngoại vi như UART, I<sup>2</sup>C, SPI.

### Boot from system memory



Boot từ SRAM memory: code được nạp vào SRAM và thực thi sau khi reset thay vì được nạp vào FLASH. Cho phép kiểm tra code nhanh chóng mà không cần nạp code lại vào Vi điều khiển. Ở chế độ này code phải được load vào SRAM, điều này thường được thực hiện thông qua JTAG hoặc bootloader từ FLASH.

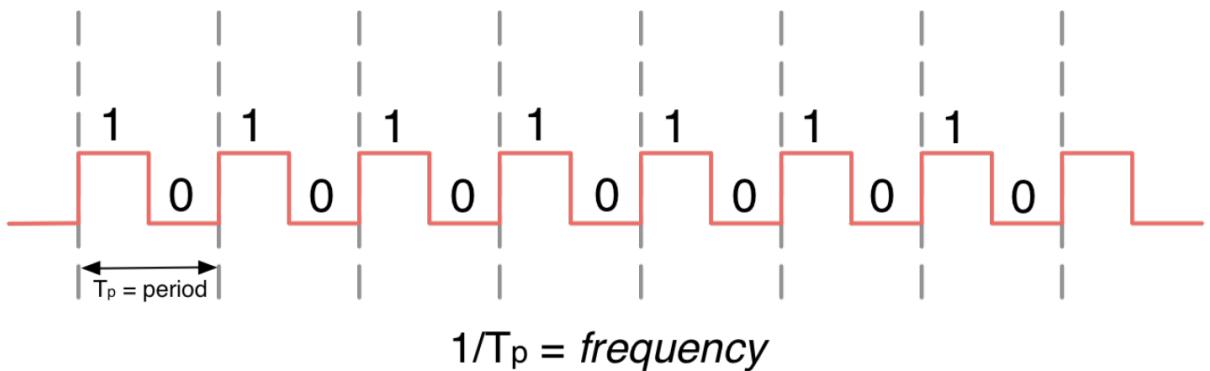
### Boot from main SRAM memory



## 1.7. Clock

### 1.7.1. Tổng quan Clock

Hầu hết các mạch điện kỹ thuật số cần một cách để đồng bộ chu kỳ hoạt động của các thành phần bên trong của nó, hoặc đồng bộ hoạt động nó với các mạch khác. Clock là một tín hiệu có chu kỳ, nó được xem là nhịp tim của thiết bị điện kỹ thuật số.



Clock có tín hiệu là xung vuông với 50% duty cycle. Thông số quan trọng của clock là tần số (frequency).

### 1.7.2. Clock tree của STM32F1

Vi điều khiển STM32F1 có hai nguồn xung clock:

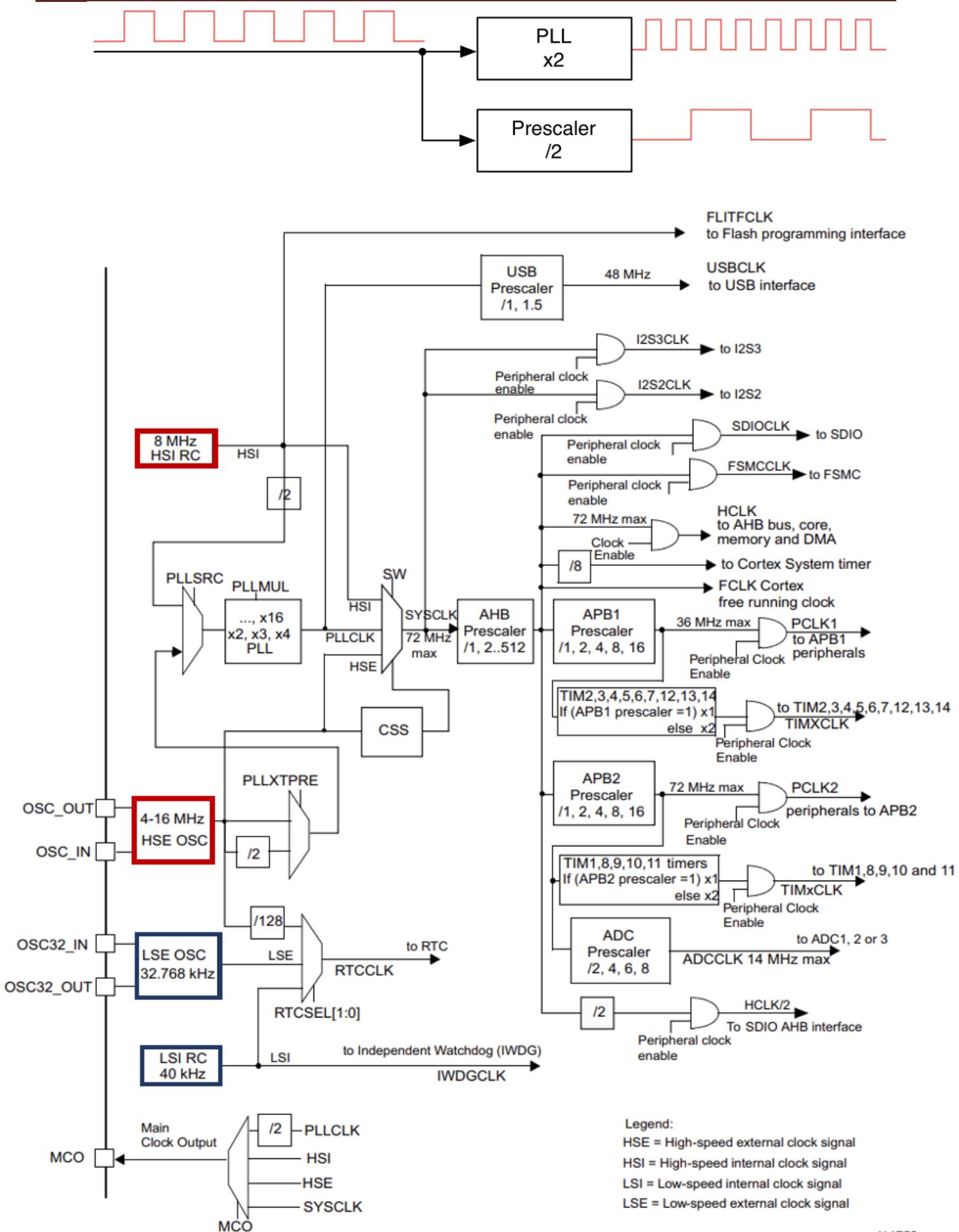
- Bộ dao động RC bên trong Vi điều khiển – có tên là HSI – High Speed Internal.
- Bộ dao động thạch anh bên ngoài – có tên là HSE – High Speed External.

Thạch anh mang lại nguồn clock có độ chính xác cao hơn so với bộ dao động RC bên trong, được đánh giá sai số 1%, đặc biệt khi nhiệt độ môi trường bên ngoài có sự khác biệt so với bên trong vi điều khiển.

Có hai nguồn xung clock tốc độ thấp: LSE – Low Speed External từ thạch anh ngoài hoặc LSI – Low Speed Internal từ bộ dao động RC nội Vi điều khiển. Nguồn clock tốc độ thấp này cấp cho RTC – Real Time Clock và IWDT – Independent Watchdog.

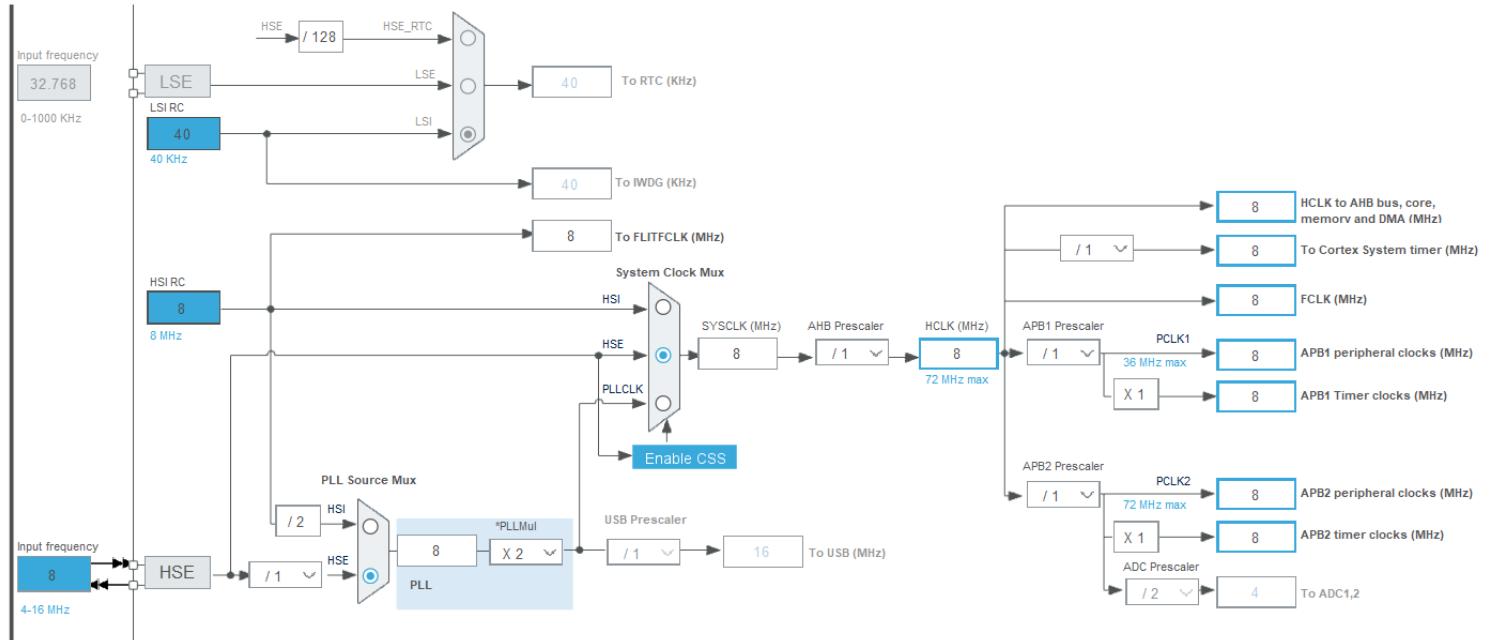
Tần số xung clock tốc độ cao không được đưa trực tiếp vào Cortex-M core hay bất kỳ ngoại vi nào khác mà nguồn clock này sẽ được đưa qua một mạng lưới phân bổ clock phức tạp, được gọi là clock tree. Bằng cách sử dụng Phase-Locked Loops – Vòng khóa pha (PLL) có thể lập trình và bộ Prescaller để tăng giảm tần số theo mục đích của từng ngoại vi và bus.

[why32768hz?](#)

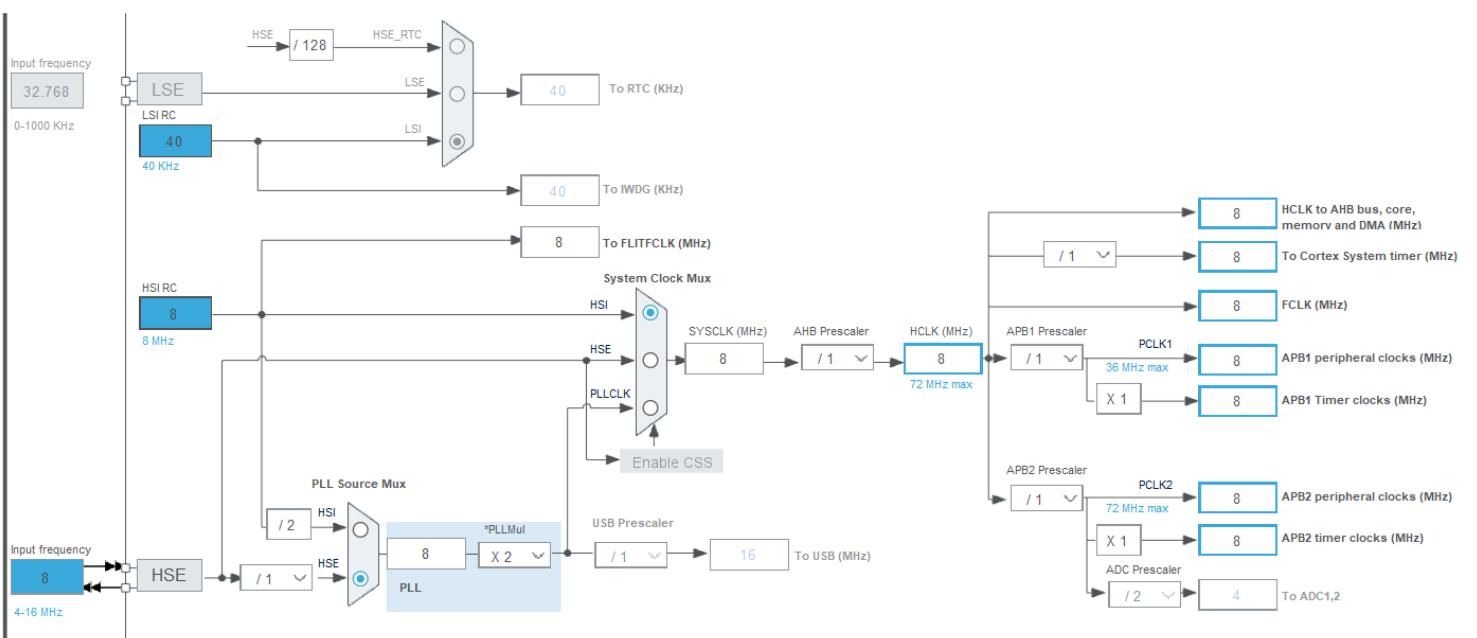


Ví dụ:

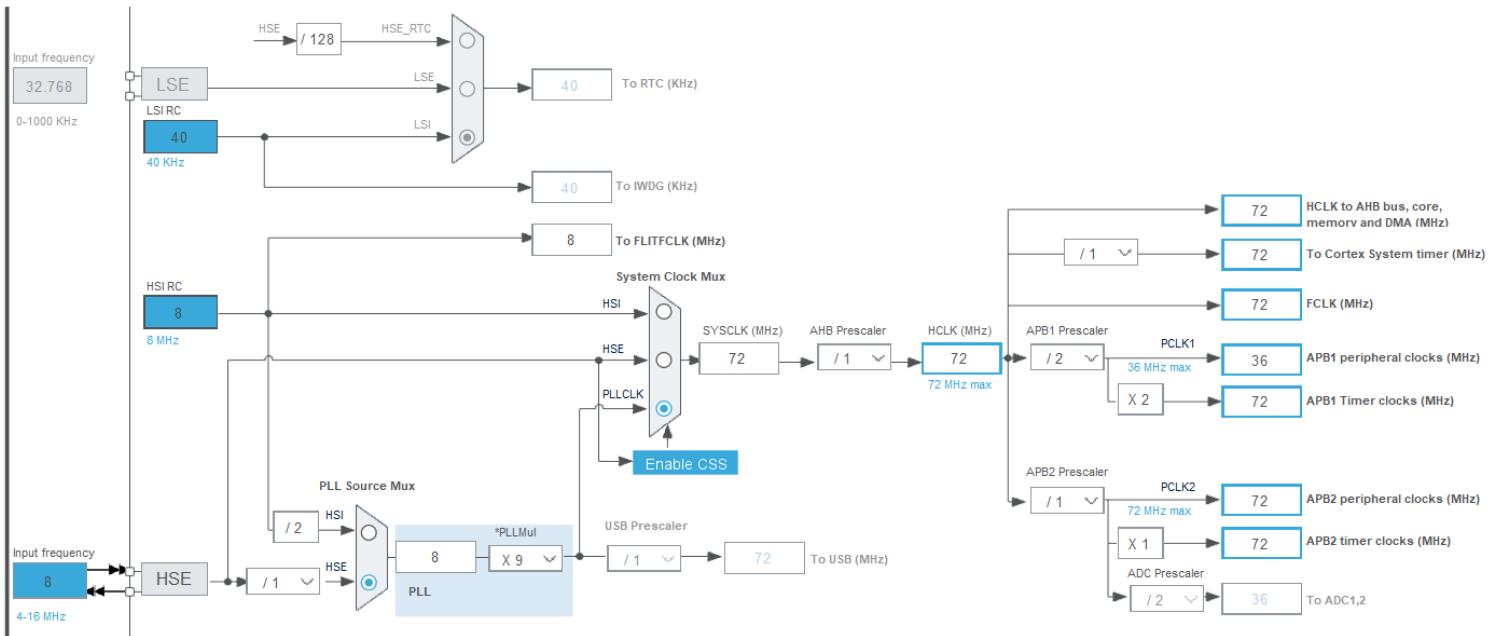
- Trong CubeMX, cấu hình xung clock cho Vi điều khiển STM32F103, chọn nguồn xung clock từ thạch anh 8MHz bên ngoài.



- Cấu hình sử dụng nguồn xung clock nội (HSI) để cấp cho Vi điều khiển.



– Cấu hình sử dụng nguồn xung clock 72MHz bằng cách sử dụng thạch anh 8MHz bên ngoài đưa qua bộ PLL để nhân tần số.



## 1.9. Interrupt

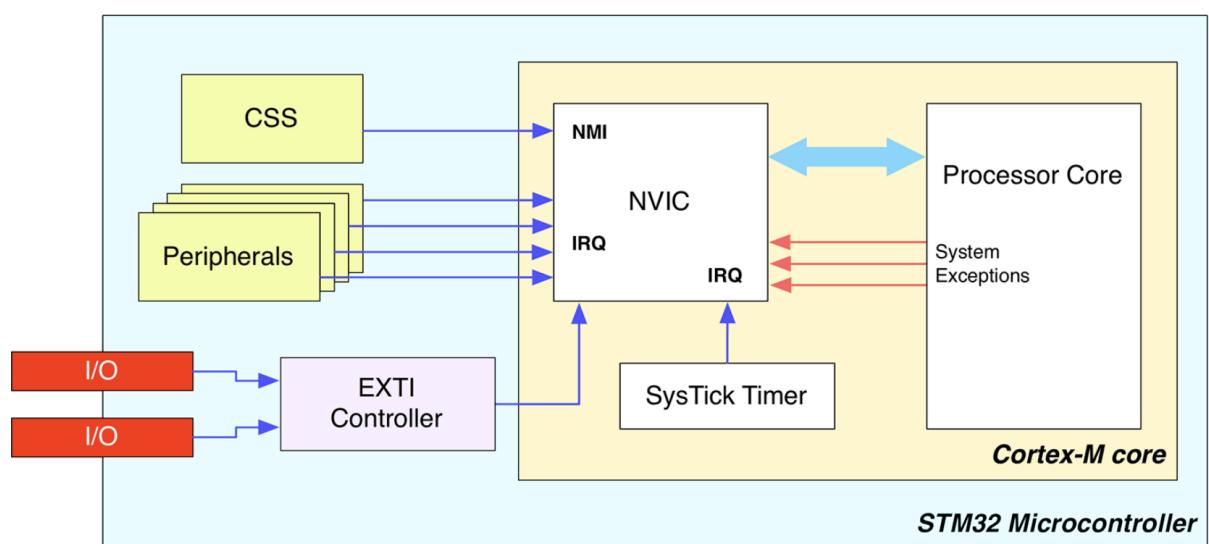
### 1.9.1. Giới thiệu

Tất cả Vi điều khiển cung cấp một tính năng gọi là ngắn – interrupt. Interrupt là một sự kiện bất đồng bộ không đoán định trước khi nào sẽ xảy ra với Vi điều khiển, nếu sự kiện này xuất hiện thì Vi điều khiển sẽ dừng chương trình chính lại (lưu bối cảnh đang thực thi (stack frame, PC,...)) và thực hiện chương trình phục vụ ngắn – Interrupt Service Routine (ISR).

Ngắn có thể được kích khởi bởi phần cứng và phần mềm của chính Vi điều khiển. Kiến trúc ARM phân biệt hai loại của sự kiện ngắn: interrupts bắt nguồn từ phần cứng, exceptions bắt nguồn từ phần mềm. Trong thuật ngữ ARM, interrupt là một loại của exception.

Cortex-M processor cung cấp một phần tử quản lý các exceptions được gọi với tên Nested Vectored Interrupt Controller (NVIC).

### 1.9.2. NVIC Controller



Có hai loại peripherals: loại bên ngoài Cortex-M core nhưng nằm bên trong MCU (như Timer, SPI, UART,...), loại còn lại nằm ngoài MCU thông qua các I/O để gửi tín hiệu ngắn đến External Interrupt/Event Controller (EXTI Controller) xử lý để đưa vào NVIC.

ARM phân biệt giữa system exceptions (được kích khởi bởi CPU core) và hardware exception từ external peripherals (được gọi là Interrupt Requests (IRQ)). Interrupt là một loại của exception.

Interrupt Service Routine (ISR) là đoạn code sẽ được thực hiện khi một exception xảy ra. Processor xác định vị trí của các ISRs này nhờ vào một bảng chứa địa chỉ

trong bộ nhớ của ISRs, được gọi là Vector table. Mỗi bộ Vi điều khiển STM32 đều xác định vector table của riêng nó.

### 1.9.3. Vector Table của STM32

Tất cả Cortex-M processor đều xác định các exceptions cố định (15 cho Cortex-M3/4/7 cores và 13 cho Cortex-M0/0+ cores).

Number	Exception type	Priority <sup>a</sup>	Function
1	Reset	-3	Reset
2	NMI	-2	Non-Maskable Interrupt
3	Hard Fault	-1	All classes of Fault, when the fault cannot activate because of priority or the Configurable Fault handler has been disabled.
4	Memory Management <sup>c</sup>	Configurable <sup>b</sup>	MPU mismatch, including access violation and no match. This is used even if the MPU is disabled or not present.
5	Bus Fault <sup>c</sup>	Configurable	Pre-fetch fault, memory access fault, and other address/memory related.
6	Usage Fault <sup>c</sup>	Configurable	Usage fault, such as Undefined instruction executed or illegal state transition attempt.
7-10	-	-	RESERVED
11	SVCALL	Configurable	System service call with SVC instruction.
12	Debug Monitor <sup>c</sup>	Configurable	Debug monitor – for software based debug.
13	-	-	RESERVED
14	PendSV	Configurable	Pending request for system service.
15	SysTick	Configurable	System tick timer has fired.
16-[47/240] <sup>d</sup>	IRQ	Configurable	IRQ Input

**Reset:** exception này được kích khởi ngay khi CPU được reset. Reset Handler là phần code được thực thi đầu tiên khi của MCU. Handler này chứa đoạn code hợp ngữ khởi tạo môi trường thực thi code cho Vi điều khiển (Stack Pointer, System Init, các segment trong FLASH và SRAM).

**NMI:** Non-Maskable Interrupt là một exception đặc biệt có mức ưu tiên cao nhất sau Reset. Exception này không thể bị vô hiệu hóa (non-maskable) và liên qua đến các hoạt động nghiêm trọng và không thể bị trì hoãn. Trong STM32, exception này lấy tín hiệu kích khởi từ bộ Clock Security (CSS), CSS là một ngoại vi chẩn đoán lỗi sai của HSE. Nếu phát hiện lỗi, HSE sẽ tự động tắt và kích hoạt HSI, sau đó thực hiện NMI Handler.

**Hard Fault:** là một exception lỗi chung liên quan đến gián đoạn trong phần mềm. Khi các fault exceptions khác bị vô hiệu hóa, Hard Fault sẽ được kích khởi bởi bất kỳ nguồn nào trong các fault exceptions đã bị vô hiệu hóa đó. Thông thường exception này được kích khởi khi sử dụng một ngoại vi nhưng chưa khởi tạo nó.

**Memory management Fault:** xảy ra khi thực thi một code đang truy xuất đến một địa chỉ không hợp lệ hoặc vi phạm quy tắc của phần tử Memory Protection Unit (MPU).

**Bus Fault:** xảy ra khi AHB nhận được phản hồi lỗi từ các bus slave. Có thể là hủy bỏ tìm nạp (prefetch abort) nếu trước đó là instruction fetch hoặc data abort nếu trước đó là data access, hoặc có thể là do quyền truy cập vào địa chỉ không hợp lệ.

**Usage Fault:** xảy ra khi có lỗi trong chương trình, như một instruction không hợp lệ, vấn đề phân vùng, truy cập một bộ xử lý không tồn tại.

**SVCCall:** đây không phải là điều kiện lỗi, nó được kích khởi khi Supervisor Call (SVC) instruction được gọi. Đây là exception được sử dụng bởi RTOS.

**Debug Monitor:** exception này được kích khởi khi debug event xuất hiện trong lúc processor trong trạng thái Debug. Debug events có thể là breakpoints và watchpoints được đặt khi debug.

**PendSV:** đây là một exception liên quan đến RTOS, khác với SVCCall ở chỗ khi SVC instruction được thực thi thì PendSV sẽ delay một khoảng thời gian (chờ Task thực thi xong) rồi mới được gọi.

**SysTick:** tất cả Vi điều khiển STM32 đều có SysTick timer bên trong Cortex-M core. Sau khi cấu hình cho timer này, SysTick Handler sẽ được thực thi khi counter của nó đếm xuống tới 0.

**IRQ:** Interurupt Request là các ngắt được kích khởi từ các ngoại vi.

Trong STM32F1, vector table được định nghĩa như sau:

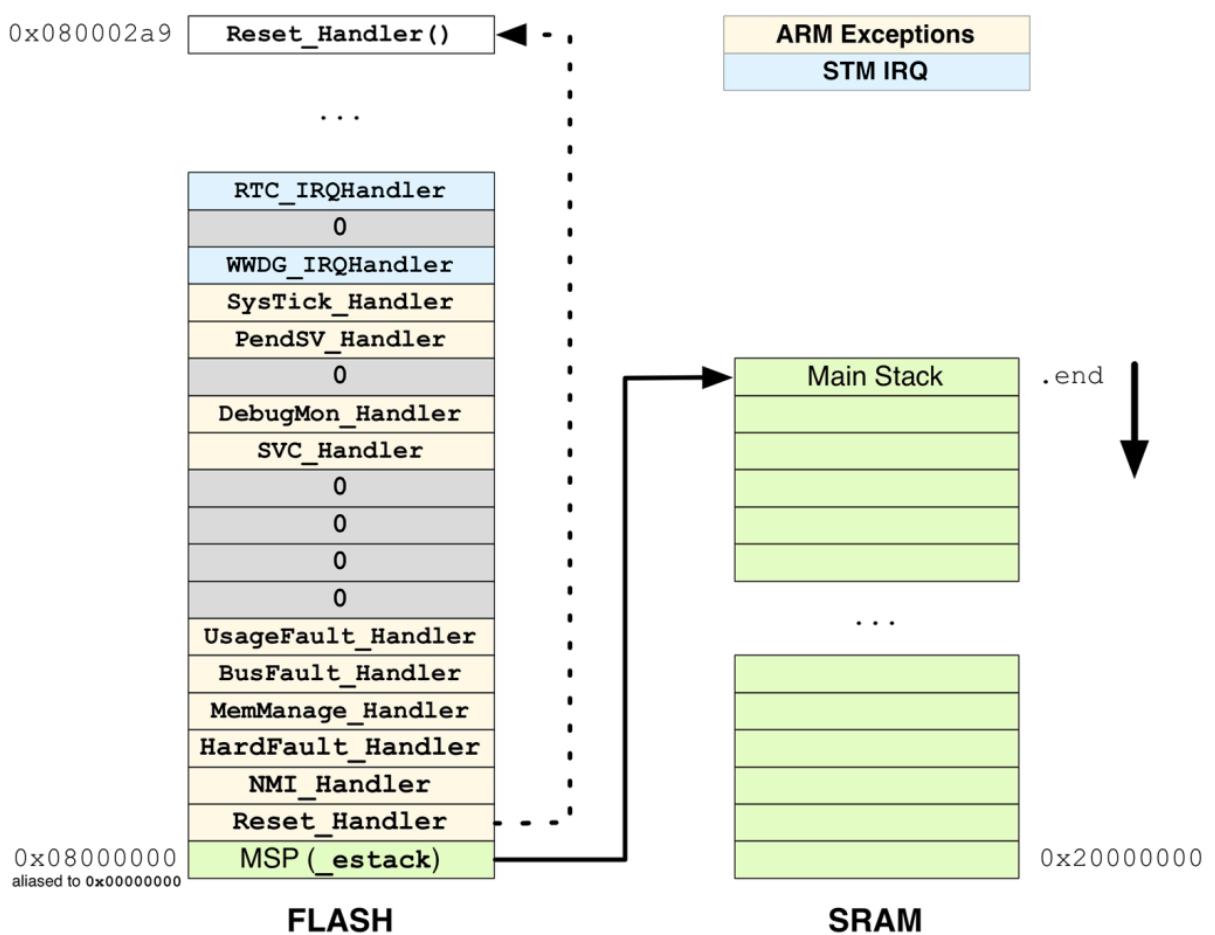
```
.word _estack
.word Reset_Handler
.word NMI_Handler
.word HardFault_Handler
.word MemManage_Handler
.word BusFault_Handler
.word UsageFault_Handler
.word 0
.word 0
.word 0
.word 0
.word SVC_Handler
.word DebugMon_Handler
.word 0
.word PendSV_Handler
.word SysTick_Handler
.word WWDG_IRQHandler
.word PVD_IRQHandler
.word TAMPER_IRQHandler
.word RTC_IRQHandler
.word FLASH_IRQHandler
.word RCC_IRQHandler
```

```

.word EXTI0_IRQHandler
.word EXTI1_IRQHandler
.word EXTI2_IRQHandler
.word EXTI3_IRQHandler
.word EXTI4_IRQHandler
.word DMA1_Channel1_IRQHandler
.word DMA1_Channel2_IRQHandler
.word DMA1_Channel3_IRQHandler
.word DMA1_Channel4_IRQHandler
.word DMA1_Channel5_IRQHandler
.word DMA1_Channel6_IRQHandler
.word DMA1_Channel7_IRQHandler
.word ADC1_2_IRQHandler
.word USB_HP_CAN1_TX_IRQHandler
.word USB_LP_CAN1_RX0_IRQHandler
.word CAN1_RX1_IRQHandler
.word CAN1_SCE_IRQHandler
.word EXTI9_5_IRQHandler
.word TIM1_BRK_IRQHandler
.word TIM1_UP_IRQHandler
.word TIM1_TRG_COM_IRQHandler
.word TIM1_CC_IRQHandler
.word TIM2_IRQHandler
.word TIM3_IRQHandler
.word TIM4_IRQHandler
.word I2C1_EV_IRQHandler
.word I2C1_ER_IRQHandler
.word I2C2_EV_IRQHandler
.word I2C2_ER_IRQHandler
.word SPI1_IRQHandler
.word SPI2_IRQHandler
.word USART1_IRQHandler
.word USART2_IRQHandler
.word USART3_IRQHandler
.word EXTI15_10_IRQHandler
.word RTC_Alarm_IRQHandler
.word USBWakeUp_IRQHandler
.word 0
.word BootRAM

```

Theo quy ước, vector table bắt đầu tại địa chỉ phần cứng 0x0000 0000 trong tất cả Cortex-M based processors. Thông thường vector table được đặt trong FLASH có địa chỉ 0x0800 0000 trong memory map, nên khi CPU khởi động thì địa chỉ FLASH được dời thành 0x0000 0000.



Vector Table phải được đặt đầu tiên trong FLASH tương ứng với file ldscripts/section.ld được nhà sản xuất (hoặc IDE) tạo ra nhằm mục đích chỉ thị nơi lưu và nơi truy xuất của processor khi cần exceptions xảy ra.

Tại địa chỉ đầu tiên của vector table là Main Stack Pointer trỏ đến đỉnh của Stack sẽ được khởi tạo trong SRAM. Tiếp theo là Reset\_Handler trỏ đến địa chỉ của chương trình được thực thi đầu tiên khi reset.

### 1.10. DMA -Direct Memory Access

#### Sự cần thiết của DMA và tầm quan trọng của Internal Buses

Mỗi ngoại vi trong STM32 đều cần trao đổi dữ liệu với Cortex-M core, một vài ngoại vi trong số đó chuyển dữ liệu thành tín hiệu I/O để trao đổi với thế giới bên ngoài Vi điều khiển thông qua một số protocol (UART, SPI, I<sup>2</sup>C, CAN,...). Một số ngoại vi khác chỉ được thiết kế để truy cập đến thanh ghi bên trong vùng nhớ peripheral để thay đổi trạng thái của các bit thanh ghi. Cả quá trình truyền dữ liệu và thay đổi trạng thái thanh ghi đều phải được CPU quản lý.

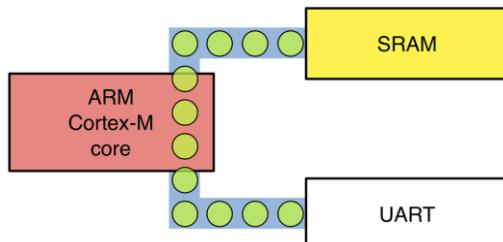
Các ngoại vi có thể được thiết kế có vùng lưu trữ riêng để giảm chi phí liên quan đến bộ nhớ ngoài. Tuy nhiên điều này làm cho kiến trúc MCU rất phức tạp, đòi hỏi chi phí cao và các thành phần sẽ tiêu thụ năng lượng lớn hơn rất nhiều.

→ Vì vậy phương pháp được áp dụng trong tất cả Vi điều khiển nhúng là sử dụng một phần của bộ nhớ trong – internal memory (SRAM hoặc FLASH) làm vùng lưu trữ tạm thời cho các ngoại vi.

*Ví dụ quá trình nhận một chuỗi 20 bytes từ bên ngoài thông qua chuẩn UART:*

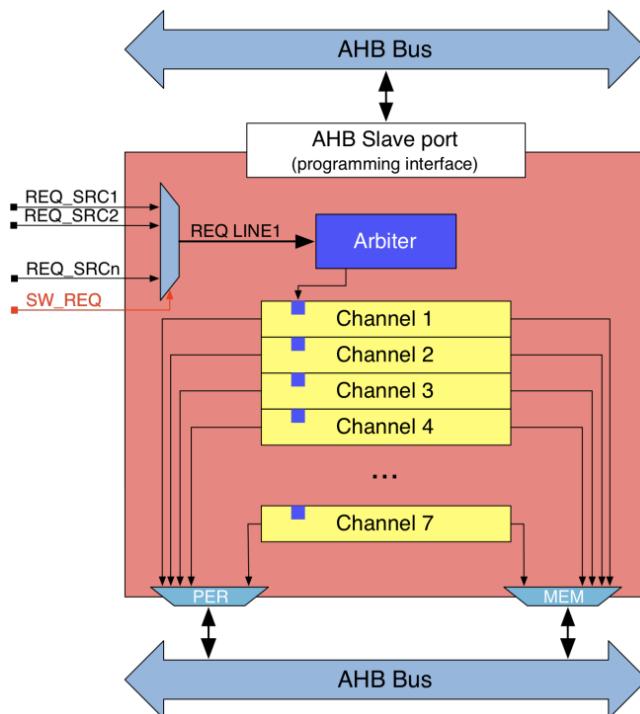
```
uint8_t buf[20];
...
HAL_UART_Receive(&huart2, buf, 20, HAL_MAX_DELAY);
```

Chuỗi buf[20] sẽ được lưu trong Internal Memory, hàm HAL\_UART\_Receive() sẽ truy cập vào thanh ghi dữ liệu huart2.Instance->DR 20 lần để chuyển từng byte dữ liệu trong thanh ghi DR sang vùng lưu trữ của buf[20]. Tất cả quá trình này đều có sự tham gia của CPU, ngay cả khi vai trò của CPU là hạn chế chuyển dữ liệu từ ngoại vi sang Internal Memory.



Có thể thấy, mặc dù phương pháp trên giúp ích trong việc đơn giản hóa thiết kế phần cứng nhưng lại làm cho hiệu suất bị hạn chế rất nhiều. Cortex-M chịu trách nhiệm tải dữ liệu từ bộ nhớ ngoại vi sang SRAM và đây là một hoạt động chặn, nó không chỉ ngăn CPU thực hiện các hoạt động khác mà còn yêu cầu CPU phải đợi các đơn vị chậm hơn phải hoàn thành công việc của chúng (do một số ngoại vi được kết nối với core bằng các bus có tốc độ chậm hơn). Giải quyết vấn đề này bằng cách sử dụng DMA – Direct Memory Access.

DMA là một đơn vị phần cứng lập trình cho phép ngoại vi của MCU truy cập trực tiếp đến Internal Memory trong MCU đó mà không cần CPU can thiệp.



Ở tất cả STM32 MCU, DMA controller là một đơn vị phần cứng có những đặc điểm sau:

- Có hai master ports, có tên là *peripheral* và *memory* được kết nối với AHB bus. Một port giao tiếp với slave peripheral và port còn lại giao tiếp với memory controller (SRAM, FLASH, FSMC,...). Một số DMA controller thì peripheral port cũng có thể giao tiếp với memory controller cho phép truyền memory to memory.
- Có một slave port, kết nối với AHB bus, sử dụng cho việc lập trình DMA controller từ master là CPU.
- Một số independent và programmable *channels* (request source), mỗi channel kết nối với một peripheral request line.
- Cho phép gán các mức độ ưu tiên khác nhau cho các channel, dùng để ưu tiên các thiết bị ngoại vi quan trọng và cần tốc độ cao hơn.
- Cho phép truyền dữ liệu theo cả hai hướng, đó là memory to peripheral và peripheral to memory.

Mỗi lần truyền DMA trải qua 4 giai đoạn:

Giai đoạn 1: Lấy mẫu và phân xử (sample and arbitration phase).

Giai đoạn 2: Chọn lọc địa chỉ (address computation phase).

Giai đoạn 3: Truy cập bus (bus access phase).

Giai đoạn 4: Xác nhận cuối cùng (báo hiệu quá trình truyền đã hoàn tất hay chưa) (final acknowledgement phase).

Mỗi giai đoạn chiếm 1 chu kỳ máy duy nhất, ngoại trừ giai đoạn truy cập bus có thể kéo dài với nhiều chu kỳ hơn tương ứng với lượng dữ liệu gửi.

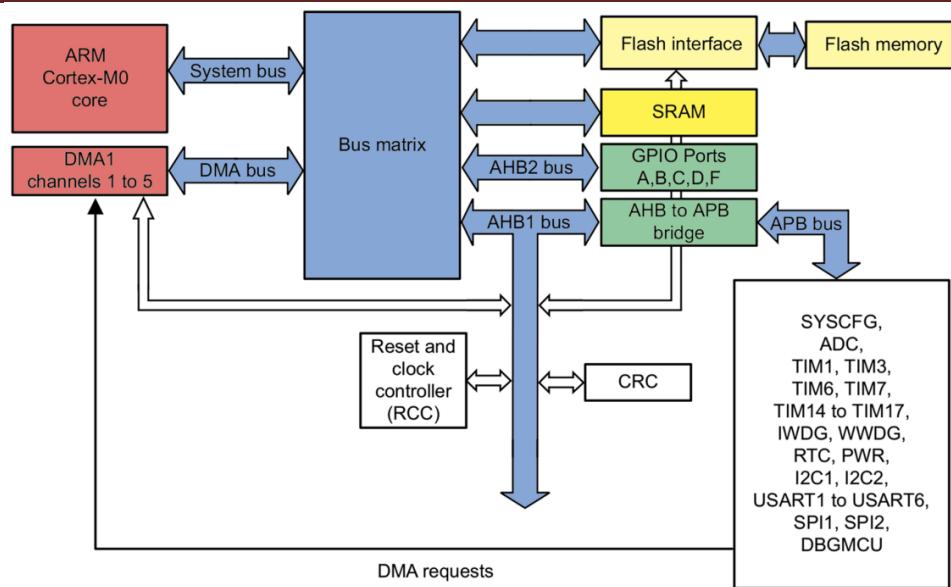


Figure 2: bus architecture of an STM32F030 microcontroller

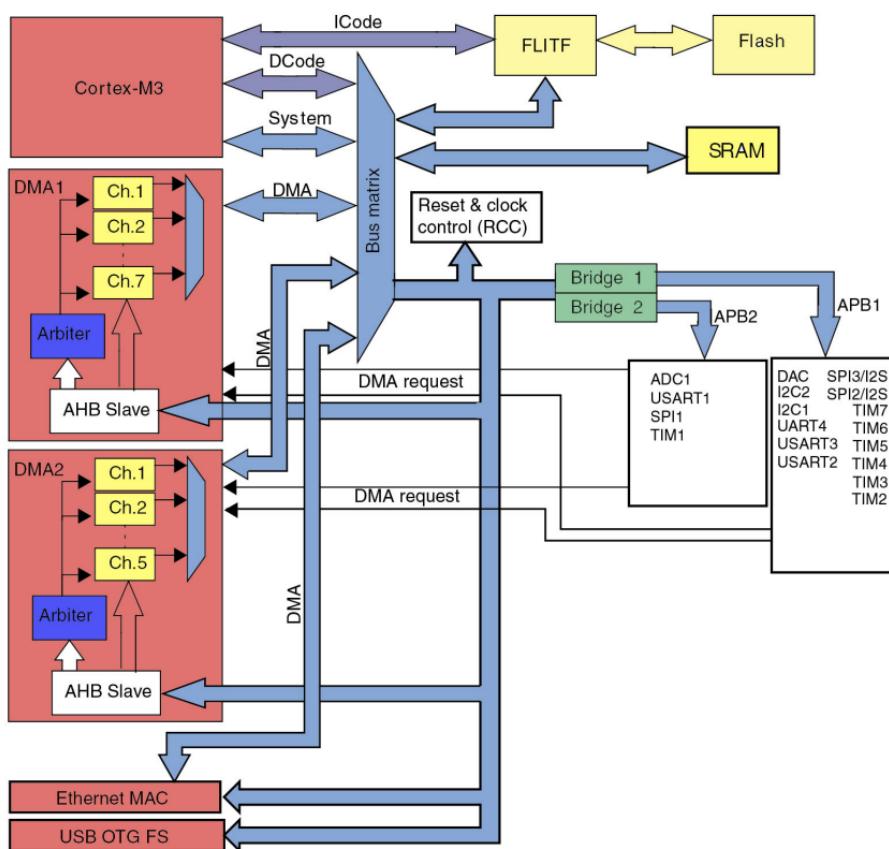


Figure 4: The bus architecture in an STM32F1 MCU from the *Connectivity Line*

## 1.11. Watchdog

### 1.11.1. Tổng quan Watchdog

Watchdog là bộ ngoại vi tích hợp trên vi điều khiển STM32, một loại Timer có khả năng giúp cho người dùng phát hiện ra hệ thống bị treo, chạy sai và tạo ra một ngắt hoặc một tín hiệu reset chip.

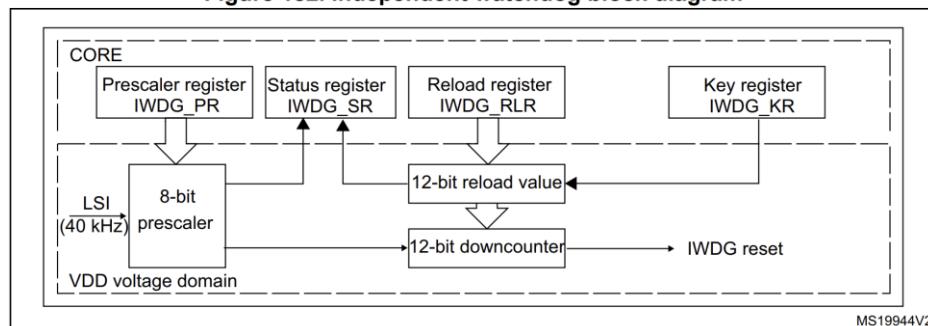
Đối với các sản phẩm nhúng thì do nhiều yếu tố bên trong (code chưa chặt chẽ, code có bug...) và yếu tố bên ngoài (nhiều, điều kiện nhiệt độ, độ ẩm) dẫn đến việc hoạt động sai của VDK dẫn đến các hiện tượng chạy chức năng bị sai, hoặc treo chip. Trong trường hợp như vậy chúng ta cần có các biện pháp khắc phục tạm thời ví dụ như reset chip để chạy lại chương trình. Bộ Watchdog Timer được sinh ra với mục đích tạo ra một tín hiệu reset bằng Software (điều khiển reset hệ thống bằng phần mềm).

Trong STM32 có 2 loại Watchdog Timer:

- Independent Watchdog Timer - IWDG: Timer này sử dụng nguồn xung LSI (Low-speed clock) và có thể hoạt động ngay cả khi nguồn clock của chương trình chính không hoạt động. Điều này phù hợp với chức năng kiểm soát lỗi treo chip kể cả do phần cứng hoặc phần mềm. Khi ngắt IWDG xảy ra, MCU sẽ lập tức bị reset.
- Window Watchdog Timer - WWDG: Timer này có nguồn xung từ bộ APB1 Clock nghĩa là bộ Timer này sẽ hoạt động khi chương trình hoạt động. Một khi xảy ra treo hệ thống liên quan tới Clock chính. Bộ Timer này sẽ không hoạt động. Vậy nên chức năng chính của WWDG là kiểm soát lỗi phần mềm, nếu các Task hoạt động bất thường như kết thúc sớm hơn hoặc muộn hơn dự kiến, ngắt WWDG sẽ xảy ra, ngắt WWDG cho phép chương trình thực thi một số lệnh trong ngắt trước khi Reset hoàn toàn MCU.

### 1.11.2. Independent Watchdog Timer – IWDG

Figure 182. Independent watchdog block diagram



- **IWDG\_PR – Prescaller Register:** Thanh ghi hệ số chia xác định thời gian mỗi 1 lần giảm của counter.

- **IWDG\_SR – Status Register:** Thanh ghi trạng thái lưu trữ trạng thái prescaler update và reload update khi có sự kiện ghi vào 2 thanh ghi này.
- **IWDG\_RLR – Reload Register:** Thanh ghi lưu giá trị nạp lại vào bộ 12-bit reload counter.
- **IWDG\_KR – Key Register:** Thanh ghi điều khiển việc chạy và reload cho bộ IWDG.

Giới thiệu cách dùng nó như thế nào (lúc trình bày)

Cơ chế hoạt động:

IWDG\_PR và IWDG\_RLR được nạp vào giá trị là số chu kỳ tràn của bộ downcounter. Khi nạp vào thanh ghi IWDG\_KR giá trị 0xCCCC thì timer sẽ bắt đầu hoạt động. Trước khi bộ downcounter đếm từ giá trị reload xuống 0, nếu ghi vào IWDG\_KR giá trị 0xAAAA thì bộ counter sẽ đếm lại từ đầu từ giá trị reload. Nếu downcounter đếm đến 0, một sự kiện reset sẽ được sinh ra và reset chip.

Vậy nên để chip không bị reset chúng ta cần nạp 0xAAAA vào IWDG\_KR trước khi counter đếm đến 0.

*Lưu ý: Trong chế độ Debug để chip không reset liên tục ta cần enable bit DBG\_IWDG\_STOP để tắt timer này đi.*

## 2. Ngôn ngữ C và ứng dụng trong lập trình nhúng

### 2.1. Declaration, definition, initialization

#### 2.1.1. Đối với variable (biến)

- Declaration (khai báo) là thông báo cho compiler biết kiểu dữ liệu (data type) và tên định danh (identifier) của biến.
- Definition (định nghĩa) là quá trình gán giá trị cho một biến đã được khai báo.
- Initialization (khởi tạo) là thông báo cho compiler biết kiểu dữ liệu (data type) và tên định danh (identifier) của biến, đồng thời gán giá trị cho biến.



```
1 int a;           /* Declaration */
2
3 void main(void)
4 {
5     a = 20;        /* Definition */
6
7     int b = 24;    /* Initialization */
8 }
```

#### 2.1.2. Đối với function (hàm)

- Declaration function (khai báo hàm) là quá trình thông báo cho compiler biết kiểu dữ liệu trả về và tên định danh của hàm, kiểu dữ liệu và số lượng của các tham số truyền vào hàm (nếu có)
- Definition function (định nghĩa hàm) là quá trình thông báo cho compiler biết nội dung sẽ được thực thi khi gọi hàm.



```
1 /* Declaration function */
2 void age_output(int age);
3
4 /* Definition function */
5 void age_output(int age)
6 {
7     printf("Age: %d", age);
8 }
9
```

## 2.2. Program structure và Scope

Lưu ý: Các phần sau chỉ nói về C được biên dịch bằng gcc cho hệ điều hành Unix và Windows.

### 2.2.1. Program structure

Một chương trình C bao gồm một hoặc nhiều file source .c. Trong đó, file có chứa hàm main() sẽ được trình biên dịch xem là file source chính, chương trình sẽ bắt đầu thực thi từ hàm main().

Theo chuẩn của trình biên dịch gcc cho hệ điều hành Unix và Windows, hàm main() phải được định nghĩa và có giá trị trả về là kiểu int. Ở phiên bản C99, cuối hàm main() nếu không có câu lệnh return (hoặc return nhưng không có giá trị int theo sau) thì tương đương với việc hàm main() đang trả về giá trị 0 (hoặc return 0). Giá trị trả về của hàm main() là 0 cho biết trạng thái exit của chương trình.

```
main.c
int lib1_func1(void);
void lib2_func2(int a);

int main()
{
    lib1_func1();

    int var;
    lib2_func2(var);
}
```

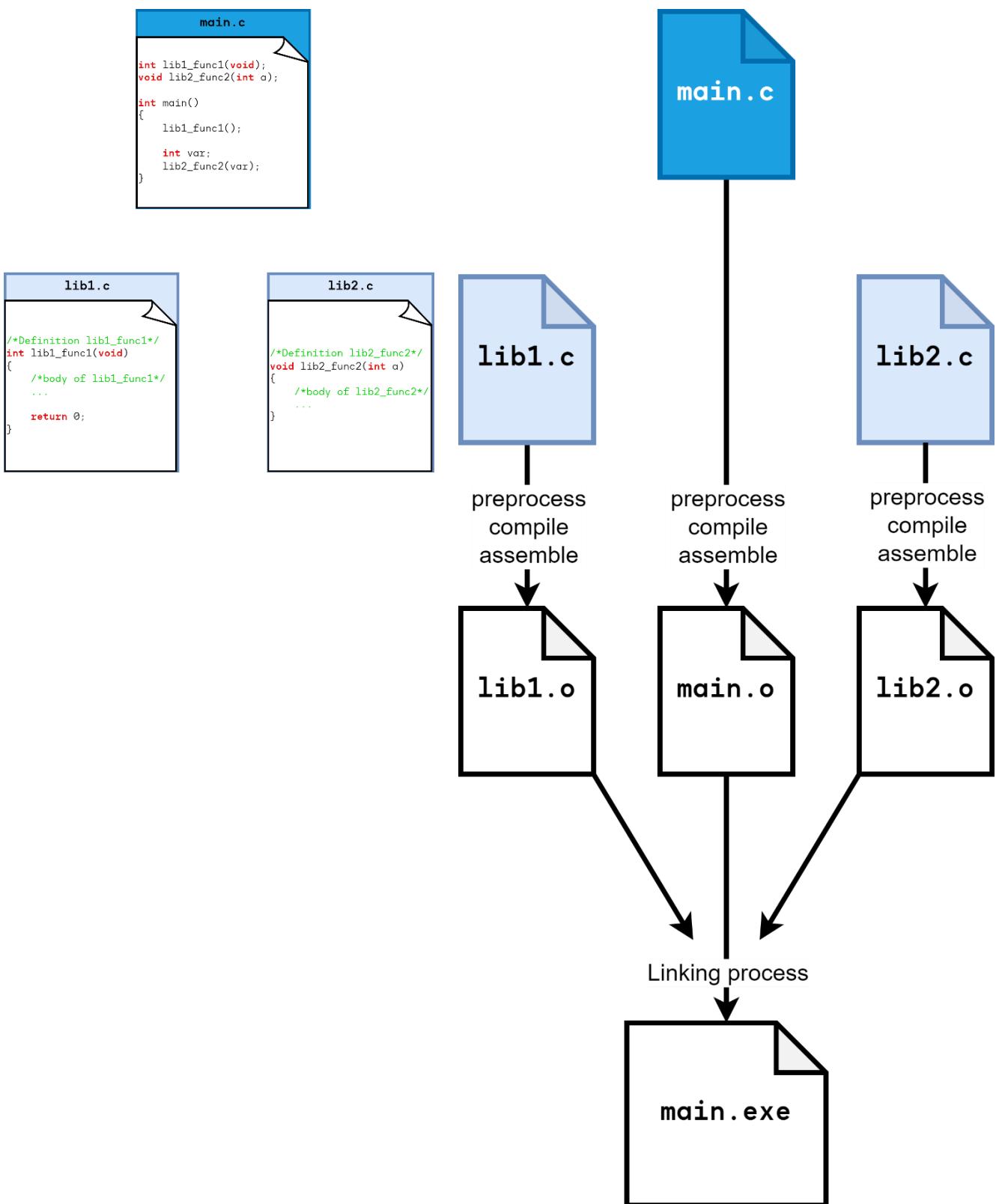
```
lib1.c
/*Definition lib1_func1*/
int lib1_func1(void)
{
    /*body of lib1_func1*/
    ...

    return 0;
}
```

```
lib2.c
/*Definition lib2_func2*/
void lib2_func2(int a)
{
    /*body of lib2_func2*/
    ...
}
```

Một trong một source file .c, ta chỉ được gọi và thực thi hàm khi đã có khai báo của hàm trước đó trong file. Phần định nghĩa hàm có thể ở tất cả các file khác được biên dịch chung với file đã gọi hàm.

> `gcc main.c lib1.c lib2.c -o main.exe`



Nếu nhiều source file .c đều cần dùng các định nghĩa ở một file .c khác thì ở mỗi source file .c này cần phải có khai báo của các định nghĩa đó. Điều này tạo ra sự bất tiện trong code khi phải khai báo lại một nội dung ở nhiều file. Các giải quyết là dùng header file .h.

Header file .h: chứa các khai báo sẽ được copy vào file source .c nếu có cú pháp sau ở file .c:

```
#include "name_header_file.h"
```

Ví dụ:

**main.c**

```
int lib1_func1(void);
void lib2_func2(int a);
void lcd_action(void);

int main()
{
    i2c_transmit(0x47);
    int buf = i2c_receive();

    lcd_action();
}
```

**i2c.c**

```
/*Definition i2c_transmit*/
void i2c_transmit(int data)
{
    /*body of i2c_transmit*/
    ...
}

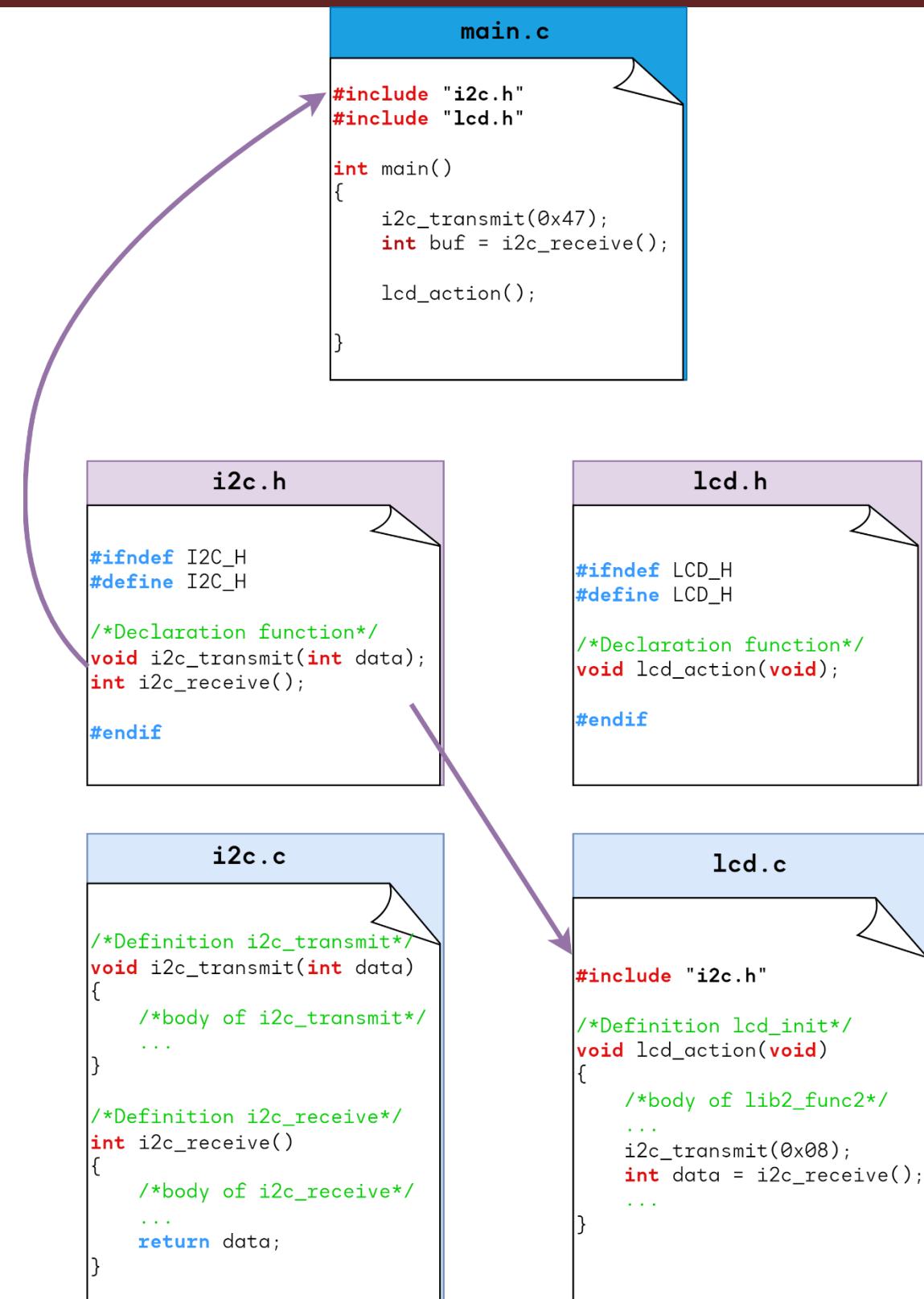
/*Definition i2c_receive*/
int i2c_receive()
{
    /*body of i2c_receive*/
    ...
    return data;
}
```

**lcd.c**

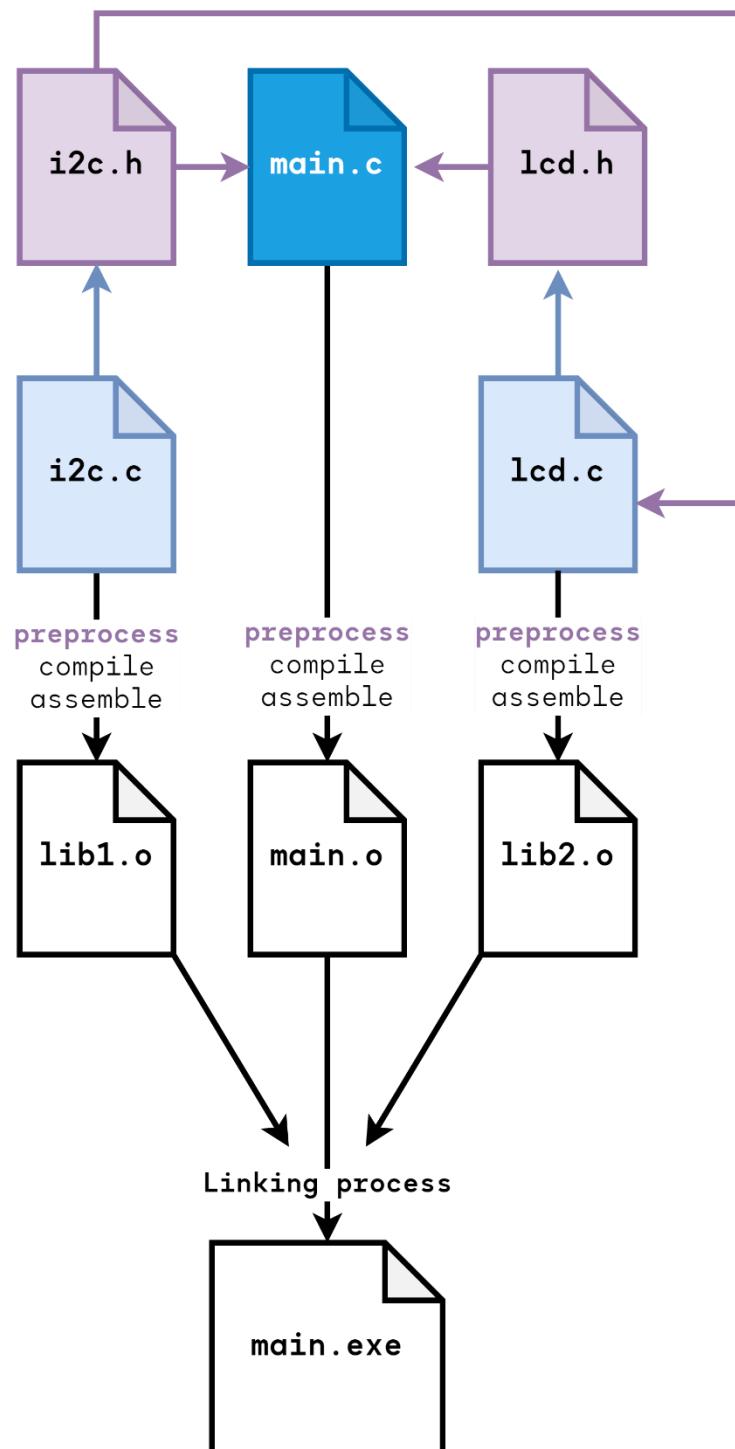
```
void i2c_transmit(int data);
int i2c_receive();

/*Definition lcd_init*/
void lcd_action(void)
{
    /*body of lib2_func2*/
    ...
    i2c_transmit(0x08);
    int data = i2c_receive();
    ...
}
```

Thay vì định nghĩa lại các hàm `i2c_transmit()`, `i2c_receive()`, `lcd_action()` thì ta có thể cho các định nghĩa này vào file header và include các file này ở mỗi file source .c cần sử dụng các hàm.



> **gcc main.c i2c.c lcd.c -o main.exe**



**File header không được chứa các thành phần được cấp phát vùng nhớ** để tránh gây lỗi biên dịch. Theo coding convention, các thành phần của một file header .h bao gồm:

- **Header guard:** nội dung của header file .h phải nằm bên trong phần header guard này. Mục đích để tránh cho việc khai báo được lặp lại nhiều lần trong một file source .c, trình biên dịch sẽ báo lỗi và không build được file .o của source file đó.

Cấu trúc của header guard như sau:



```

1  #ifndef HEADER_NAME_H
2  #define HEADER_NAME_H
3
4  /* Inside header file content */
5
6  #endif

```

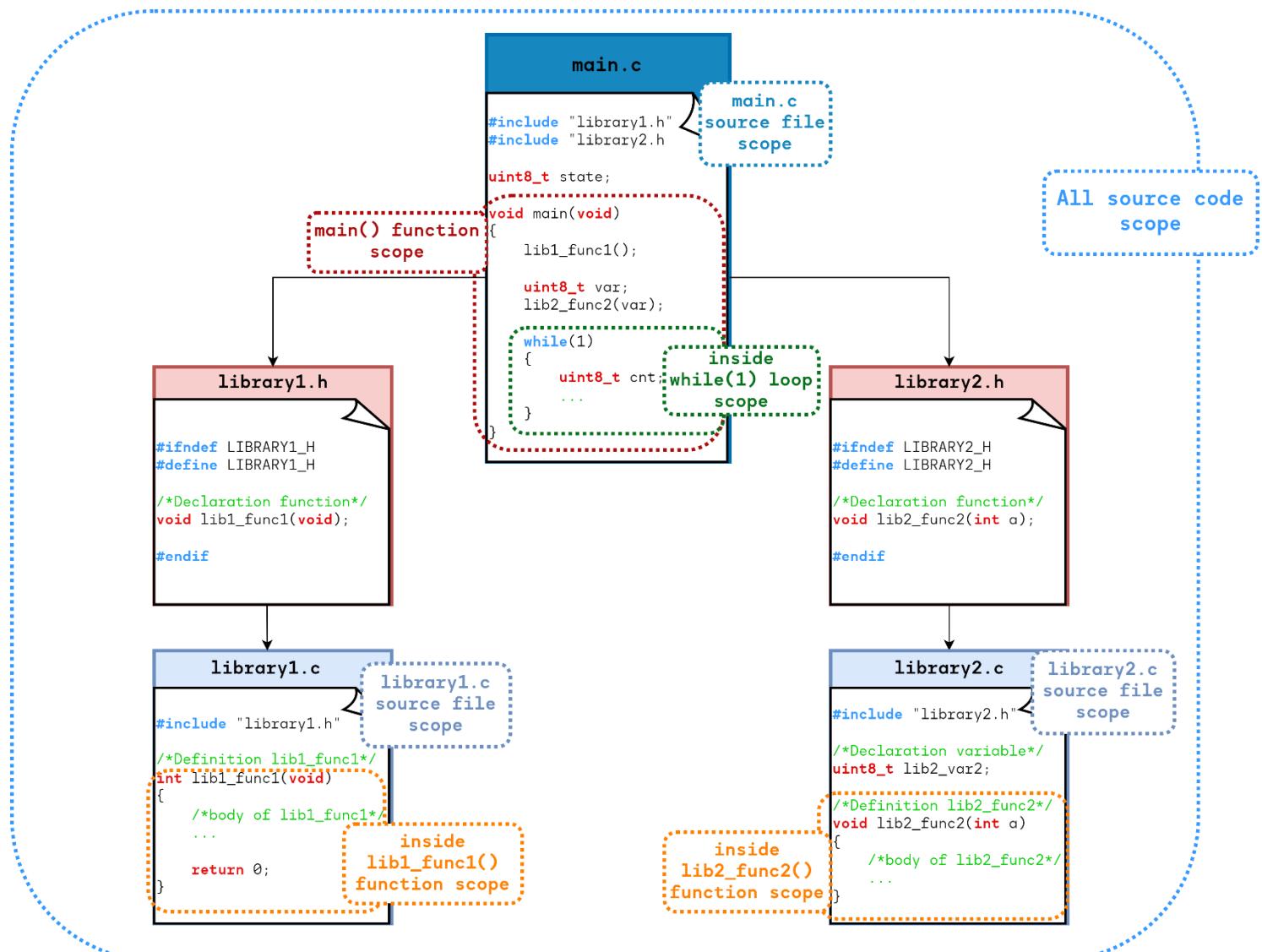
- **Function Declaration/Function Prototype:** các khai báo hàm sẽ được sử dụng khi include header file này.
- **Include header file .h:** một header file có thể include các header file khác.
- **Extern Global variable:** khai báo các biến toàn cục của một source file .c sẽ được sử dụng bởi các source file có include header file .h này.
- **Định nghĩa macro:** #define
- **Định nghĩa kiểu dữ liệu (có hoặc không có typedef):** enum, struct, union.

Một thư viện sẽ bao gồm một source file .c chứa các định nghĩa hàm và khai báo của các biến toàn cục, source file này sẽ include một header file .h tương ứng với thư viện đó chứa các thành phần đã nêu trên.

<p><b>lib.h</b></p> <pre> #ifndef LIB_H #define LIB_H  /*Include other header file*/ #include &lt;stdio.h&gt; #include "other_lib.h"  /*Declaration function*/ void lib_func1(); void lib_func2();  /*Extern global variable*/ extern int lib_var;  #endif </pre>	<p><b>lib.c</b></p> <pre> #include "lib.h"  /*Declaration global variable*/ int lib_var;  /*Definition lib_func1*/ void lib_func1() {     /*body of i2c_transmit*/     ... }  /*Definition lib_func2*/ void lib_func2() {     /*body of i2c_receive*/     ... } </pre>
---	--

## 2.2.2. Scope

Scope đề cập đến những phần nào của chương trình sẽ có thể nhìn thấy một đối tượng đã được khai báo. Phần chương trình có thể truy xuất đến các đối tượng có scope cùng mức hoặc lớn hơn, các phần chương trình nằm ngoài scope.



Các đối tượng có scope là toàn source code: global variables trong được khai báo trong các source file .c, các hàm được khai báo trong các header file .h.

Các đối tượng có scope trong nội bộ một file source .c: các biến và hàm khai được khai báo với từ khóa static.

Các đối tượng có scope trong một hàm: các biến được khai báo trong một hàm.

---

## 2.3. Variable

Storage Class Specifiers: có bốn bộ từ khóa có thể thêm vào trước phần khai báo biến của để thay đổi cách lưu trữ các biến trong bộ nhớ: `extern`, `static`, `auto`, `register`.

### 2.3.1. External variable

Khi một source file .c mong muốn truy xuất đến một global variable đã được khai báo ở một source file .c khác, thì biến đó phải được khai báo với từ khóa `extern` trong file .c này.

Cú pháp:

**extern [data type] [name variable declared];**

Ví dụ:

Cả file `main.c` và `lib.c` đều cùng truy xuất đến một biến `lib_var`.

**lib.h**

```
#ifndef LIB_H
#define LIB_H

#include <stdio.h>

/*Declaration function*/
void update_var();

#endif
```

**lib.c**

```
#include "lib.h"

/*Declaration global variable*/
int lib_var;

void update_var()
{
    lib_var = 100;
    printf("Update: %d", lib_var);
}
```

**main.c**

```
#include <stdio.h>
#include "lib.h"

/*Extern global variable*/
extern int lib_var;

int main()
{
    lib_var = 10;
    printf("Value: %d\n", lib_var);
    update_var();
}
```

> **gcc main.c lib.c -o main.exe**

> .\main.exe

Value: 10

Update: 100

Khi khai báo một biến với từ khóa extern, trình biên dịch sẽ không cấp phát vùng nhớ cho biến này khi build thành file .o vì biến này sẽ được tìm thấy trong quá trình linking tại một file .o khác.

### 2.3.2. Static variable

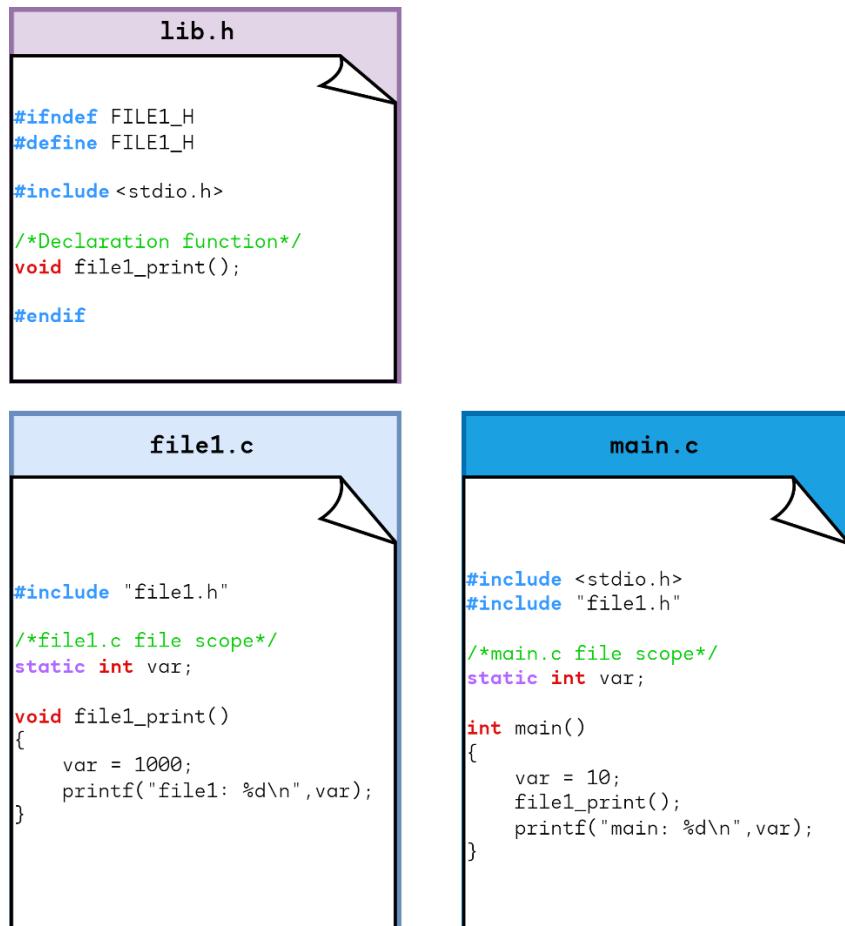
Cú pháp:

**static [data type] [name variable]; //declaration**

**static [data type] [name variable] = [value]; //initialization**

Khi khai báo biến static ở ngoài tất cả các hàm (global scope): scope của biến lúc này là file source .c đã khai báo biến. Các file khác không thể truy xuất đến biến này thông qua từ khóa extern nữa.

Ví dụ:



```

lib.h
#ifndef FILE1_H
#define FILE1_H

#include <stdio.h>

/*Declaration function*/
void file1_print();

#endif

file1.c
#include "file1.h"

/*file1.c file scope*/
static int var;

void file1_print()
{
    var = 1000;
    printf("file1: %d\n", var);
}

main.c
#include <stdio.h>
#include "file1.h"

/*main.c file scope*/
static int var;

int main()
{
    var = 10;
    file1_print();
    printf("main: %d\n", var);
}

```

> **gcc main.c file1.c -o main.exe**

> .\main.exe

file1: 1000

main: 10

Khi khai báo biến static trong một hàm thì scope của biến là hàm đó và giá trị của biến sẽ không bị mất đi khi hàm kết thúc.

**file2.h**

```
#ifndef FILE2_H
#define FILE2_H

#include <stdio.h>

/*Declaration function*/
void file2_update();

#endif
```

**file1.c**

```
#include "file1.h"

void file2_update()
{
    /*file2_update function scope*/
    static int count = 0;

    count++;
    printf("file2: %d\n", count);
}
```

**main.c**

```
#include <stdio.h>
#include "file2.h"

int main()
{
    file2_update();
    file2_update();
    file2_update();
}
```

```
> gcc main.c file2.c -o main.exe
```

```
> .\main.exe
```

```
file2: 1
file2: 2
file2: 3
```

### 2.3.3. Auto variable (recomment)

Từ khóa `auto` chỉ được dùng trong khai báo hoặc khởi tạo biến local trong một hàm. Biến có từ khóa `auto` sau khi khai báo có giá trị rác, sau khi thoát khỏi hàm thì giá trị biến mất đi. Nó giống với hành vi mặc định của một biến khi được khai báo trong hàm.

---

#### 2.3.4. Register variable

Các biến được khai báo với từ khóa `register` sẽ được trình biên dịch lưu trong core register ở CPU, các biến này nên là các biến được sử dụng nhiều trong chương trình và ta không thể dùng toán tử `địa chỉ` để tham chiếu đến `địa chỉ` của biến này.

Type Qualifier: hai từ khóa có thể thêm vào trước phần khai báo biến của mình để thay đổi cách truy cập các biến: `const`, `volatile`.

### 2.3.5. Constant variable

Khi khởi tạo một biến có chứa từ khóa `const`, biến đó sẽ được compiler xác định là `read-only`, giá trị của biến không thể bị thay đổi trong suốt quá trình chạy của chương trình. Bất kỳ hành động thay đổi giá trị bằng cách sử dụng tên định danh của biến được khai báo với từ khóa `const` sẽ được trình biên dịch báo lỗi.

Trong lập trình nhúng, các mảng dữ liệu có kích thước lớn và không cần thay đổi dữ liệu trong quá trình code vận hành thường được khởi tạo với từ khóa `const` để tiết kiệm RAM.

Cú pháp khởi tạo:

```
const [data type] [name constant variable] = [value];
```

```
[data type] const [name constant variable] = [value];
```

Thực chất các biến local `const` được khởi tạo trong một hàm vẫn có thể được thay đổi được giá trị thông qua con trỏ, vì các biến này khi chạy sẽ được khởi tạo trên RAM. Các biến global `const` không thể thay đổi được giá trị bằng bất kỳ hình thức nào.



```

1 #include <stdio.h>
2
3 const int b = 19;
4
5 int main()
6 {
7     const int a = 123;
8
9     int *ptrb = &b;
10    //ptrb = 20;           //compile error
11
12    int *ptra = &a;
13    *ptra = 12;
14
15    printf("a = %d\n", a);
16    printf("b = %d", b);
17 }
```

Trình biên dịch chỉ báo warning nhưng không báo lỗi. Kết quả trên terminal:

a = 12

b = 19

### 2.3.6. Volatile variable

Từ khóa volatile sẽ báo hiệu cho compiler biết rằng biến này có thể bị thay đổi giá trị bất cứ lúc nào bởi các lệnh mà compiler không thể tự nhận biết được. (Các thanh ghi thay đổi giá trị phù thuộc vào người dùng, quá trình hoạt động).

Trong lập trình nhúng, tất cả các biến được sử dụng trong hàm ngắt phải được khai báo với từ khóa volatile. Các biến dùng để truy xuất đến các giá trị thanh ghi phải được khai báo với từ khóa volatile. Nếu không, khi build với mức tối ưu thì trình biên dịch sẽ nhận thấy các biến trên không có sự thay đổi nào trong source code và sẽ tối ưu mất các biến đó.

[Mỗi một lần sử dụng phải load lại từ RAM vào register.]

Cú pháp khai báo:

**volatile [data type] [name variable];**

## 2.4. Function

### 2.4.1. External function

External function là hàm được khai báo với từ khóa extern và không được định nghĩa.

VD: **extern void function\_1();**

Cũng như khi được dùng với biến, extern khi được dùng với hàm sẽ giúp file code hiện tại có thể sử dụng hàm đã định nghĩa ở file code khác.

Việc khai báo hàm có từ khóa extern và không có từ khóa này đều như nhau.

### 2.4.2. Static function

Là hàm được khai báo hoặc định nghĩa với từ khóa static.

VD: **static void function\_1();**

Các hàm static là các hàm chỉ được sử dụng trong file khai báo nó, không thể sử dụng ở các file khác thông qua từ khóa extern.

### 2.4.3. Inline function

Là hàm được khai báo với từ khóa inline.

VD: **static inline void function\_1();**

Trong các phiên bản compiler mới và các trình biên dịch cho vi điều khiển ARM, ta bắt buộc phải khai báo inline function đi kèm với static hoặc extern để xác định scope hàm.

Tác dụng: khi gọi inline function thì thay vì thực hiện thao tác gọi hàm, mà các lệnh bên trong hàm sẽ được chèn trực tiếp vào vị trí gọi hàm trong file assembly.

Khi gọi hàm, hệ thống sẽ thực hiện push một đoạn prolog vào stack, sau đó thực hiện hàm trong stack, trước khi thoát khỏi hàm thì hệ thống thực hiện thêm đoạn epilog trong stack. Với inline function, quá trình gọi hàm (prolog và epilog) được lược bỏ đi, giúp cho quá trình vận hành code nhanh hơn. Tuy nhiên file code được load vào máy sẽ có dung lượng lớn hơn nếu hàm inline function được gọi nhiều lần.

Vì vậy ta sử dụng inline function cho các hàm chỉ gọi một lần trong suốt quá trình vận hành. Thông thường là các hàm khởi tạo.

#### 2.4.4. Function pointer

Function pointer là con trỏ trả về một hàm, ta có thể gọi hàm thông qua con trỏ đó.

Cú pháp khai báo:

```
[func_type] (*[func_ptr_name])([param_data_type]);
```

Ví dụ:

```
void(*func_ptr_1)();  
float(*func_ptr_2)(int, float);
```

Để gọi hàm bằng function pointer, ta sử dụng cú pháp như gọi hàm bình thường.



```
1 #include <stdio.h>  
2  
3 void print_hello()  
4 {  
5     printf("hello\n");  
6 }  
7  
8 int main(void)  
9 {  
10    void (*func_ptr)();  
11    func_ptr = print_hello;  
12    func_ptr();  
13 }
```



```
1 #include <stdio.h>  
2  
3 float mul(int a, float b)  
4 {  
5     return (a*b);  
6 }  
7  
8 int main(void)  
9 {  
10    float (*func_ptr)(int, float);  
11    func_ptr = mul;  
12    printf("%f\n", func_ptr(3, 0.3));  
13 }
```

Khi sử dụng function pointer, để rút gọn cấu trúc khi khai báo nhiều con trỏ hàm có cùng kiểu dữ liệu trả về và cùng đối số truyền vào, ta tạo ra một kiểu dữ liệu của function pointer bằng từ khóa **typedef**

Cú pháp:

```
typedef [func_type] (*[func_ptr_name_t])([ptr_data_type]);
```

Ví dụ: Tạo kiểu dữ liệu con trỏ hàm tên **callback\_func\_ptr\_t** có kiểu dữ liệu trả về là **void**, có tham số với kiểu dữ liệu lần lượt là **char \***, **int**.

```
typedef void (*callback_func_ptr_t)(char *, int);
```

Lúc này ta khai báo một function pointer bằng cách:

```
[func_ptr_name_t] [func_ptr_name];
```

Ví dụ:



```

1 #include <stdio.h>
2
3 typedef (*func_ptr_t)(int, float);
4
5 float mul(int a, float b)
6 {
7     return (a*b);
8 }
9
10 int main(void)
11 {
12     // float (*func_ptr)(int, float);
13     func_ptr_t func_ptr;
14     func_ptr = mul;
15     printf("%f\n", func_ptr(3, 0.3));
16 }
```

#### 2.4.5. Variadic function

Variadic function là hàm có số lượng đối số truyền vào không được xác định trong lúc định nghĩa hàm. Variadic function có thể xử lý bất kỳ số lượng đối số nào mà caller truyền vào.

Cú pháp: theo chuẩn ISO C thì ta cần truyền ít nhất một đối số được khai báo cố định, tiếp theo là bất kỳ đối số nào khác được khai báo với kí tự '...'.

**[return\_type] [name\_variadic\_func]([last\_required\_argument], ...){}**

Ví dụ: **int func(const char \*a, int b, ...)**

```

{
    /* body of variadic function */
}
```

**Caller:** func("hello", 12, 5, 6, 7, 8);

Để truy xuất đến các đối số đã truyền vào variadic function, ta thực hiện các bước sau khi định nghĩa hàm:

- Include thư viện `stdarg.h`
- Khởi tạo một argument pointer variable thuộc loại `va_list`.

- Sử dụng macro **va\_start(va\_list ap, last-required)** để khởi tạo phần tử đối số đầu tiên. **last-required** là đối số cuối cùng trong các đối số được truyền cố định vào hàm đã được khai báo.
- Truy xuất đến đối số tiếp theo được truyền vào hàm bằng macro **va\_arg(va\_list ap, type)** với **type** là kiểu dữ liệu của đối số muốn truy xuất đến.
- Khi đã hoàn tất quá trình truy xuất, dùng macro **va\_end(va\_list ap)**.

Ví dụ:

```
#include <stdio.h>
#include <stdarg.h>

int sum(int num,...)
{
    int result = 0;

    va_list args;
    va_start(args, num);
    for (int i = 0; i < num; i++)
    {
        result = result + va_arg(args, int);
    }
    va_end(args);
    return result;
}

int main(void)
{
    printf("%d\n", sum(5, 1, 2, 3, 4, 5));
    printf("%d\n", sum(2, 13, 2));
}
```

Result in terminal:      15  
                              15

Ví dụ: hàm lcd\_printf() có hoạt động giống hàm printf()

```
void lcd_printf(const char *string,...)
{
    char tringarr[16];

    va_list args;
    va_start(args, string);
    vsprintf(tringarr, string, args);
    va_end(args);

    /*lcd i2c transmit function, pass tringarr*/
}
```

## 2.5. Pointer

Pointer là biến dùng để lưu địa chỉ, được khai báo theo cú pháp:

**[data\_type] \*[pointer\_name];**

Để truy xuất giá trị ô nhớ mà pointer trả tới ta dùng toán tử '\*' đặt trước tên của biến pointer.



```

1 int a = 10;
2 int *ptr_a = &a;
3 *ptr_a = 12;

```

## 2.6. Array

Array là một tập hợp các biến có cùng kiểu dữ liệu sẽ được sắp xếp trên 1 vùng nhớ liên tiếp nhau.

Ví dụ:

```
int arr[5] = {1, 2, 3, 4, 5};
```

Trong đó:

`arr[0]`, `arr[1]`, `arr[2]`, `arr[3]`, `arr[4]` là các biến kiểu int.

`arr` và `&arr` cùng trả về địa chỉ của phần tử đầu tiên (`arr[0]`).

`arr` không phải pointer (vì không có ô nhớ nào lưu giá trị của arr nên nó không phải biến, nên cũng không phải là pointer).

Các trường hợp khi khai báo array:

- Không khai báo số lượng phần tử trong mảng, hệ thống sẽ dựa vào số phần tử được định nghĩa trị lúc khởi tạo để quyết định số phần tử của mảng.

Ví dụ: `int arr_1[] = {1, 2, 3, 4, 5};` → Mảng 5 phần tử.

`char arr_2[] = {'a', 'b', 'c'};` → Mảng 3 phần tử.

- Không khởi tạo đủ số giá trị của các phần tử, các phần tử không được định nghĩa giá trị sẽ có giá trị bằng 0.

Ví dụ: `int arr_1[4] = {1, 2, 3};` → `arr[3] = 0`.

`int arr_2[3] = {};` → Tất cả các phần tử có giá trị bằng 0.

**Không khởi tạo giá trị:** Nếu mảng là global sẽ có giá trị bằng 0, nếu mảng là local sẽ có giá trị rác.

## 2.7. String

Mảng ký tự là 1 mảng kiểu char, ngoài cách khởi tạo như bình thường ta có thể khởi tạo mảng ký tự theo cách sau:

```
char arr[] = "PIF";
```

Lúc này mảng arr sẽ tương đương với:

```
char arr[4] = {'P', 'I', 'F', '\0'};
```

Ta không thể gán giá trị các phần cho 1 mảng đã khởi tạo theo cách sau:

```
char arr[4];
```

```
arr = "PIF";
```

→ Error. Ta phải dùng hàm strcpy() hoặc strncpy().

Nhưng nếu làm như sau thì vẫn được:

```
char *str;
```

```
str = "PIF";
```

→ Lúc này giá trị của pointer str là địa chỉ của ô nhớ chứa ký tự 'P'.

## 2.8. Struct

Là 1 phuong thức để tạo 1 nhom các biến có kiểu dữ liệu khác nhau theo cú pháp:

```
struct [struct_name]
{
    [member_data_type] [member_name];
    . . . ;
};
```

Các biến member có thể là biến thông thường, biến con trỏ, con trỏ hàm, nhưng không thể khai báo nguyên mẫu hàm trong struct.

Để khai báo 1 biến có kiểu dữ liệu struct ta dùng cú pháp:

```
struct [struct_name] [variable_name];
```

Khởi tạo giá trị cho biến kiểu struct:



```

1 #include <stdio.h>
2
3 struct personal_t
4 {
5     char *name;
6     int age;
7 };
8
9 int main()
10 {
11     struct personal_t pers_1;
12     pers_1.name = "Tun";
13     pers_1.age = 19;
14 }

```

```

1 #include <stdio.h>
2
3 struct personal_t
4 {
5     char *name;
6     int age;
7 };
8
9 int main()
10 {
11     struct personal_t pers_1 = {"Tun", 19};
12 }

```



```

1 #include <stdio.h>
2
3 struct personal_t
4 {
5     char *name;
6     int age;
7 };
8
9 int main()
10 {
11     struct personal_t pers_1 =
12     {
13         .name = "Tun",
14         .age = 19
15     };
16 }

```

Để truy xuất tới biến thành phần bên trong 1 biến kiểu struct, ta dùng dấu chấm `.'



```

1 #include <stdio.h>
2
3 struct personal_t
4 {
5     char *name;
6     int age;
7 };
8
9 int main()
10 {
11     struct personal_t pers_1 =
12     {
13         .name = "Tun",
14         .age = 19,
15     };
16
17     printf("Name: %s\nAge:%d", pers_1.name, pers_1.age);
18 }

```

Nếu biến kiểu struct là 1 pointer thì ta dùng dấu `->'



```

1 #include <stdio.h>
2
3 struct personal_t
4 {
5     char *name;
6     int age;
7 };
8
9 int main()
10 {
11     struct personal_t pers_1 =
12     {
13         .name = "Tun",
14         .age = 19,
15     };
16
17     struct personal_t *ptr_pers;
18     ptr_pers = &pers_1;
19
20     printf("Name: %s\nAge:%d", ptr_pers->name, ptr_pers->age);
21 }

```

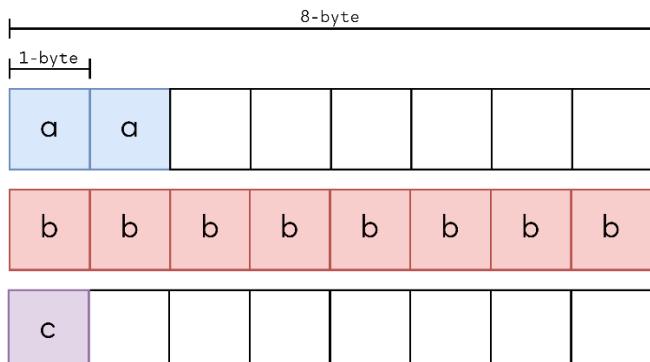
Trong struct, các biến thành phần sẽ được sắp xếp trong bộ nhớ theo thứ tự như lúc khai báo và sẽ tránh việc xếp 1 dữ liệu của 1 biến trên 2 ô nhớ khác nhau để quá trình truy xuất dữ liệu đạt tốc độ nhanh nhất.



```

1 struct struct_t
2 {
3     uint16_t a;
4     uint64_t b;
5     uint8_t c;
6 }

```



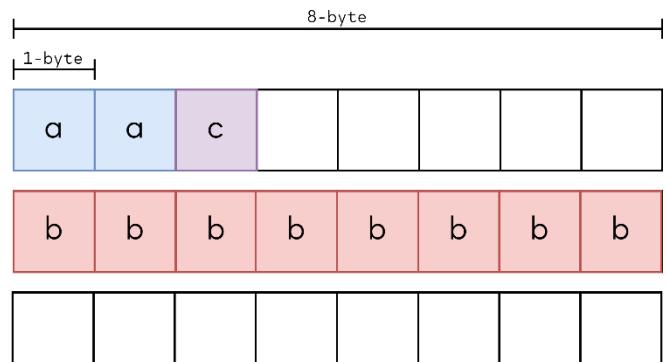
Kết quả: 24 bytes



```

1 struct struct_t
2 {
3     uint16_t a;
4     uint8_t c;
5     uint64_t b;
6 }

```



Kết quả: 16 bytes

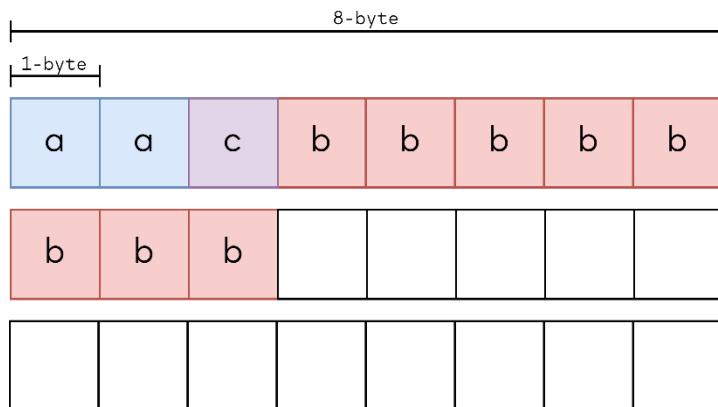
Vì 1 số lý do (dùng để truyền UART,...), ta có thể làm cho các vùng nhớ của các biến thành phần xếp sát nhau bằng cách:



```

1 struct struct_t
2 {
3     uint16_t a;
4     uint8_t c;
5     uint64_t b;
6 }__attribute__((packed));

```



Kết quả: 11 bytes

Bit field là một tính năng giúp ta có thể quy định mỗi biến thành phần bên trong struct sẽ sử dụng bao nhiêu bit bằng cách thêm ' : ' sau các biến thành phần và trước dấu ' ; ' - Lưu ý: số bit phải nhỏ hơn số bit kích thước của kiểu dữ liệu của biến thành phần.



```

1 struct struct_t
2 {
3     int a : 10;           /* a used 10-bit */
4     unsigned int b : 22;  /* b used 22-bit */
5     unsigned int c : 32;  /* c used 32-bit */
6 }__attribute__((packed));

```

Kết quả: 8-bytes

## 2.9. Union

Union là 1 phuong thức để tạo 1 nhóm các biến có kiểu dữ liệu khác nhau, nhưng vùng nhớ được cấp cho union sẽ bằng với kích thước của biến thành phần có kích thước lớn nhất. Cú pháp:

**union [union\_name]**

```
{
    [member_data_type] [member_name];
    . . . ;
}
```

Các biến thành phần có thể là biến thông thường, biến con trỏ, con trỏ hàm, nhưng không thể khai báo hàm trong union.

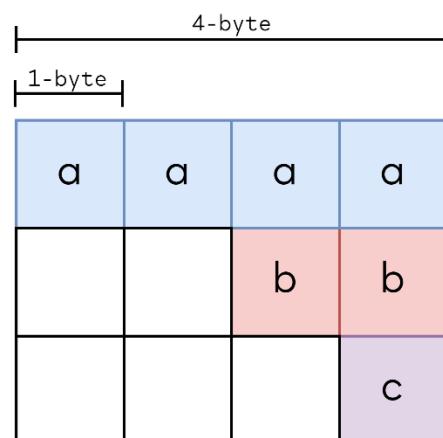
Để tạo 1 biến có kiểu dữ liệu struct dùng cú pháp:

```
union [union_name] [variable_name];
```

● ● ●

```

1 union union_t
2 {
3     uint32_t a;
4     uint16_t b;
5     uint8_t c;
6 }
```



Ngoài ra người ta còn dùng union trong truyền nhận thông tin, ví dụ khi ta cần truyền 1 biến kiểu float (4 byte) qua UART, ta tạo 1 union như sau:

● ● ●

```

1 union union_t
2 {
3     float var;
4     uint8_t arr[4];
5 };
6
7 int main()
8 {
9     union union_t data;
10    data.var = 4.9;
11    HAL_UART_Transmit(&huart1, data.arr, 4);
12 }
```

## 2.10. Cấp phát động

Trong quá trình chương trình vận hành, sẽ có những lúc ta cần cấp phát một ô nhớ để lưu dữ liệu không đoán định được tổng thước trong quá trình code, việc tạo

sẵn 1 vùng nhớ trong lúc code để dùng cho những trường hợp này sẽ gây lãng phí bộ nhớ hoặc vùng nhớ tạo sẵn không đủ, ta giải quyết các vấn đề này bằng cách sử dụng cấp phát động.

Vùng heap trong bộ nhớ sẽ được sử dụng cho việc cấp phát động.

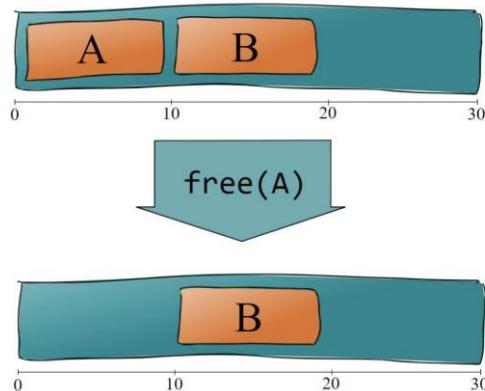
Để yêu cầu cấp phát vùng nhớ ta có thể dùng hàm: `malloc()`, `calloc()`, `realloc()`; các hàm này sẽ trả về địa chỉ ô nhớ đầu tiên của vùng nhớ được cấp phát, nếu không thể cấp phát sẽ trả về `NULL`.

Để giải phóng vùng nhớ đã được cấp phát khi không sử dụng nữa, ta dùng hàm `free()`.

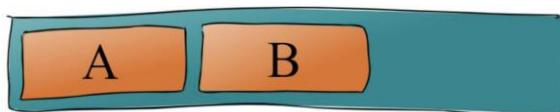
Vấn đề mất tham chiếu đến vùng nhớ - memory leak: không thể sử dụng vùng nhớ đã cấp phát được nữa do không còn biến lưu địa chỉ của vùng nhớ đó, dẫn đến việc không thể sử dụng hàm `free()` để giải phóng vùng nhớ đó.

Vấn đề double free: hàm `free()` được gọi đến 2 lần, lúc biên dịch thì chương trình sẽ không báo lỗi, lỗi chỉ xảy ra ngay lúc chạy chương trình, cụ thể là lúc chạy tới hàm `free()` thứ 2, lúc này chương trình sẽ bị treo.

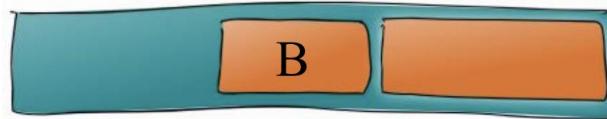
Phân mảnh vùng heap - Heap fragmentation: giả sử ta cấp phát động 2 vùng là A và B đều có kích thước 10 byte, sau đó giải phóng vùng A.



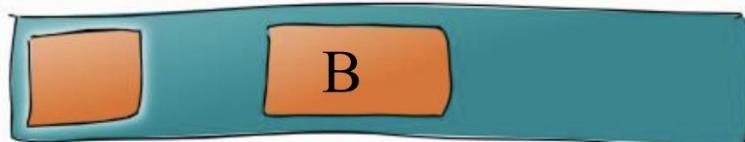
Nếu ở lần cấp phát động kế tiếp ta cần 1 vùng nhớ 10 byte bằng vùng A thì vùng A sẽ được sử dụng lại.



Nếu ở lần cấp phát kế ta cần 1 vùng nhớ lớn hơn vùng A, thì vùng A không thể dùng lại được mà ta phải mở rộng vùng heap ra thêm.



Nếu ở lần cấp phát kế ta cần 1 vùng nhớ nhỏ hơn vùng A, thì 1 phần của vùng A sẽ được dùng lại.



Các vùng nhớ sau khi được giải phóng chỉ được tận dụng lại tốt khi có yêu cầu cấp phát động vùng nhớ mới bằng với vùng được giải phóng.

Việc yêu cầu cấp động với kích thước khác nhau sẽ khiến cho bộ nhớ heap xuất hiện nhiều vùng nhớ không sử dụng, đây gọi là hiện tượng heap fragmentation.

→ Trong các hệ thống có bộ nhớ nhỏ như hệ thống nhúng, ta nên hạn chế dùng heap nếu không cần thiết

## 2.11. Macro

### 2.11.1. Object-like Macros

Object-like Macros là một mã định danh (identifier) đơn giản sẽ được thay thế bằng một đoạn code. Macros này được gọi là object-code vì nó giống một data object bên trong code. Ta sử dụng chỉ thị `#define` để tạo macro:

**#define [NAME\_MACRO] [replacement\_code]**

Ở preprocessing, tất cả các **[NAME\_MACRO]** sẽ được thay thế bằng phần **[replacement\_code]**.

Ví dụ: **#define BUFFER\_SIZE 1024**

```
foo = (char *) malloc (BUFFER_SIZE);
preprocessing foo = (char *) malloc (1024);
```

Object-like Macros có thể trải dài trên nhiều dòng, ở cuối mỗi dòng (ngoại trừ dòng cuối cùng) ta để thêm ký tự '`\`'.

Ví dụ:

```
#define NUMBERS 1, \
                2, \
```

```
int x[] = { NUMBERS };  
preprocessing → int x[] = { 1, 2, 3 };
```

Theo coding convention, tên của các macros phải viết hoa.

### 2.11.2. Function-like Macros

Function-like Macros là các macros thực hiện chức năng giống như việc gọi hàm. Để khởi tạo một function-like macros, ta sử dụng chỉ thị `#define` và phần tên có thêm cặp dấu ngoặc đơn, sau đó là một hàm.

Ví dụ:

```
#define lang_init() c_init()

lang_init();
preprocessing
→ c_init();
```

### 2.11.3. Macro Arguments

Function-like Macros cũng có thể truyền vào đối số như một hàm bình thường.

Ví dụ:

```
#define min(X,Y) ((X) < (Y) ? (X) : (Y))

x = min(a,b);
preprocessing
→ x = ((a) < (b) ? (a) : (b));

y = min(a+28,*p);
preprocessing
→ y = ((a+28) < (*p) ? (a+28) : (*p));
```

### 2.11.4. Stringizing

Các thông số của macros sẽ được preprocessor biến thành chuỗi nếu trước nó có ký tự '#.

Ví dụ:

```
#define WARN_IF(EXP) \
    do { if (EXP) \
        fprintf (stderr, "Warning: " #EXP "\n"); } \
    while (0)

WARN_IF (x == 0);
preprocessing
→ do { if (x == 0)
        fprintf (stderr, "Warning: " "x == 0" "\n"); }
    while (0);
```

### 2.11.5. Variadic Macros

Giống với variadic function, macros-like function cũng có thể được định nghĩa với số lượng đối số truyền vào chưa đoán định được. Tại phần tên của macros, ta truyền vào đối số là ký tự '...', phần được thay thế sẽ sử dụng token `__VA_ARGS__` để thay thế cho các đối số đã truyền vào.

Ví dụ:

```
#define eprintf(...) fprintf (stderr, __VA_ARGS__)  
eprintf ("%s:%d: ", input_file, lineno);  
preprocessing → fprintf (stderr, "%s:%d: ", input_file, lineno);
```

### 3. Embedded C

#### 3.1. Phân biệt C bare metal(super loop) với C trên hệ điều hành

#### 3.2. Compiler/Cross-compiler và Compiling process

##### 3.2.1. Compiler và Cross-compiler

Trình biên dịch là công cụ để chuyển đổi ngôn ngữ lập trình của con người (assembly, C, C++, ...) thành ngôn ngữ máy (nhị phân) cho máy tính vận hành.

Native compiler: là trình biên dịch biên dịch các file chương trình để vận hành trên chính máy tính đang dùng trình biên dịch. Ví dụ: máy tính Windows sử dụng trình biên dịch GCC để biên dịch một chương trình cho chính máy tính Windows.

Cross-compiler: là trình biên dịch biên dịch các file chương trình để vận hành trên nền tảng khác máy tính đang dùng trình biên dịch. Ví dụ: máy tính Windows sử dụng trình biên dịch GCC-ARM-NONE-EABI để biên dịch một chương trình vận hành trên vi điều khiển ARM-core.

##### 3.2.2. GCC (GNU Compiler Collection)

GNU C Compiler (GCC) ban đầu được phát triển bởi Richard Stallman, người sáng lập Dự án GNU vào năm 1984 nhằm tạo ra một hệ điều hành giống Unix hoàn chỉnh dưới dạng phần mềm miễn phí, nhằm thúc đẩy sự tự do và hợp tác giữa những người dùng máy tính và các lập trình viên.

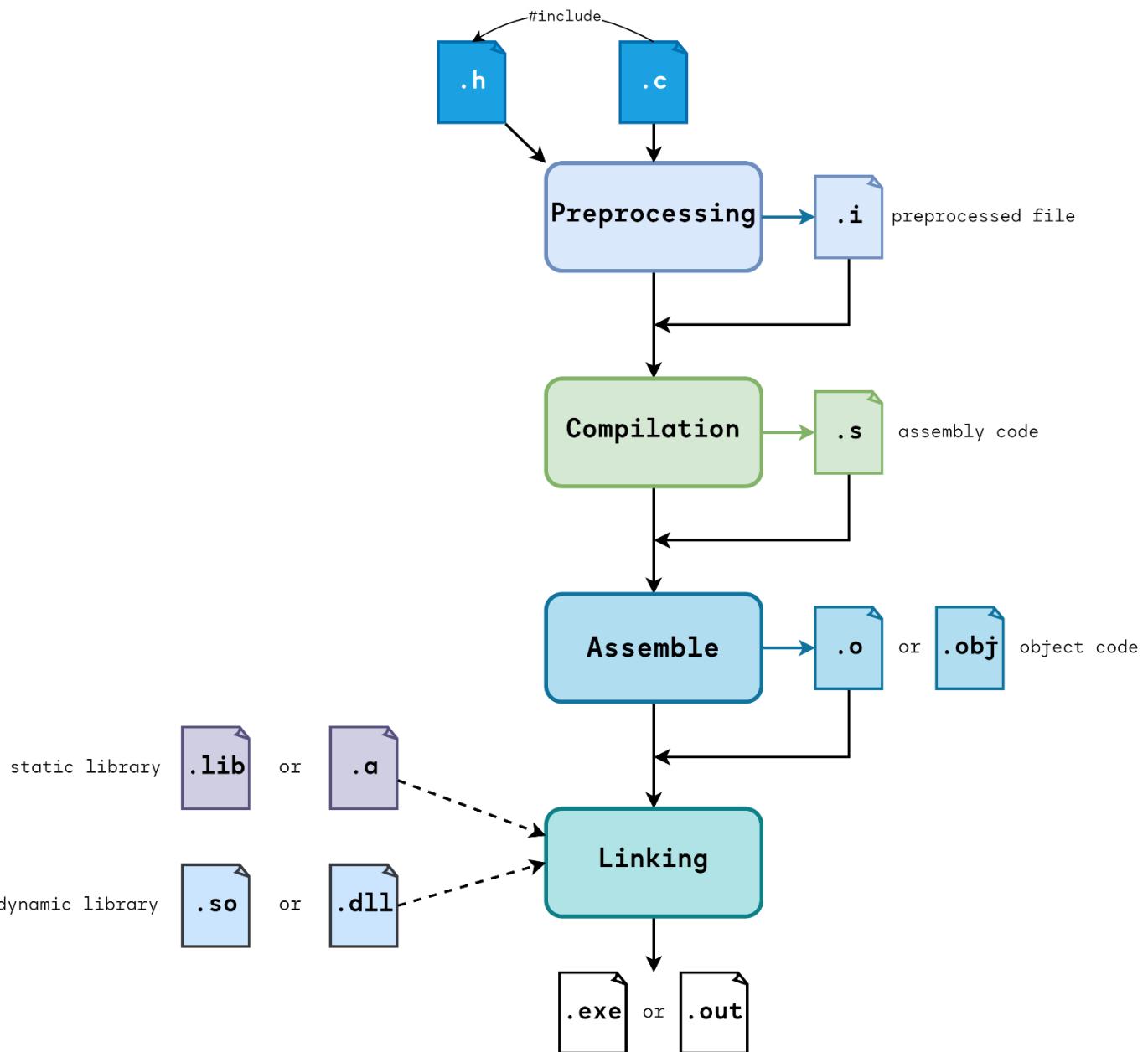
Sau thời gian phát triển, GNU C Compiler hỗ trợ cho rất nhiều ngôn ngữ khác nhau nên hiện tại được gọi GNU C Collection (GCC). Site gốc của GCC: <http://gcc.gnu.org/>

GCC ở hệ điều hành Unixes: là trình biên dịch tiêu chuẩn cho hầu hết các hệ điều hành Unix-like và sẵn có trong hệ điều hành.

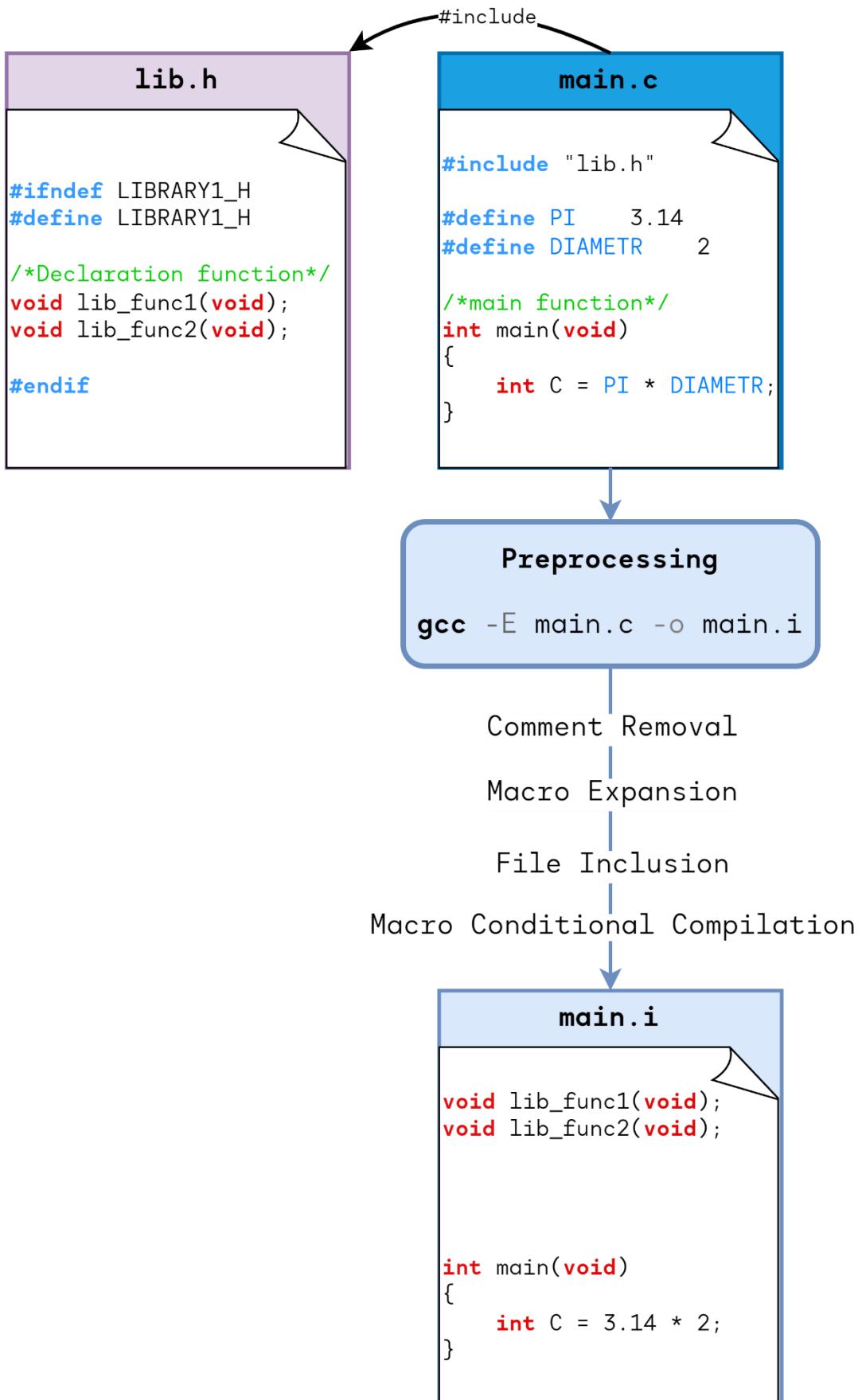
GCC ở hệ điều hành Windows: ta cần cài đặt Cygwin GCC, MinGW GCC or MinGW-W64 GCC:

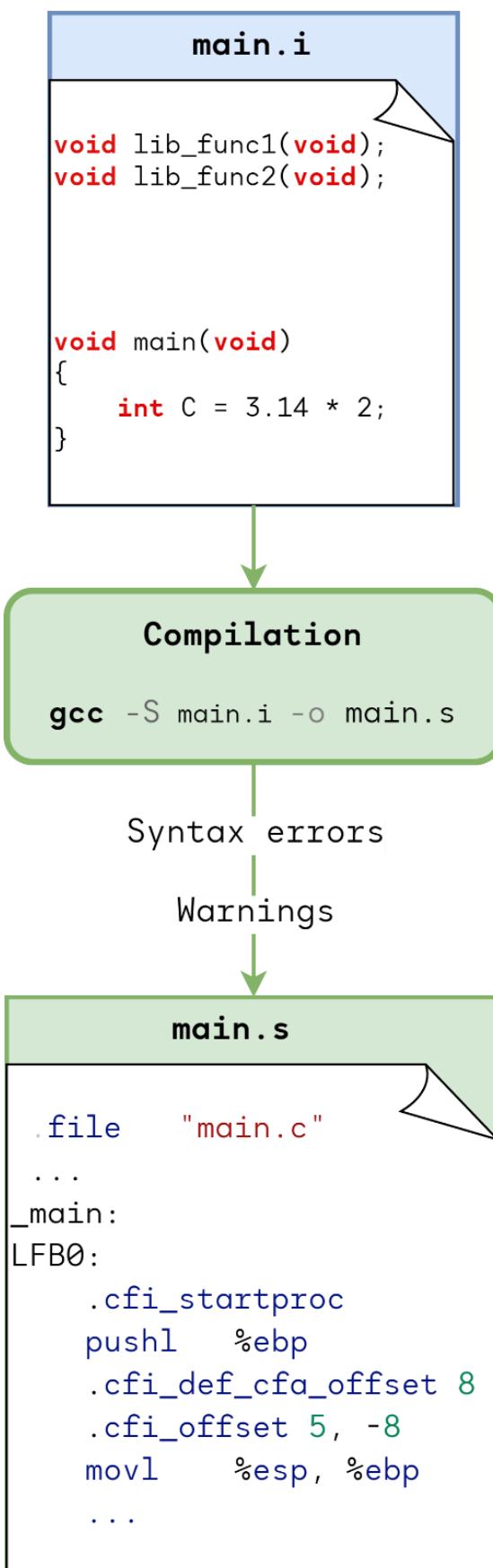
- Cygwin GCC: là một môi trường hỗ trợ command-line của Unix-like dành cho Microsoft Windows.
- MinGW (Minimalist GNU for Windows): là một cổng của GCC sử dụng cho Windows, bao gồm MSYS (Minimal System).
- MinGW-W64: là một nhánh của MinGW hỗ trợ cả Windows 32 bit và 64 bit.

### 3.2.3. Compiling process



## Preprocessing:



**Compilation:**

Assemble:

```
main.s  
file "main.c"  
...  
_main:  
LFB0:  
.cfi_startproc  
pushl %ebp  
.cfi_offset 8  
.cfi_offset 5, -8  
movl %esp, %ebp  
...
```

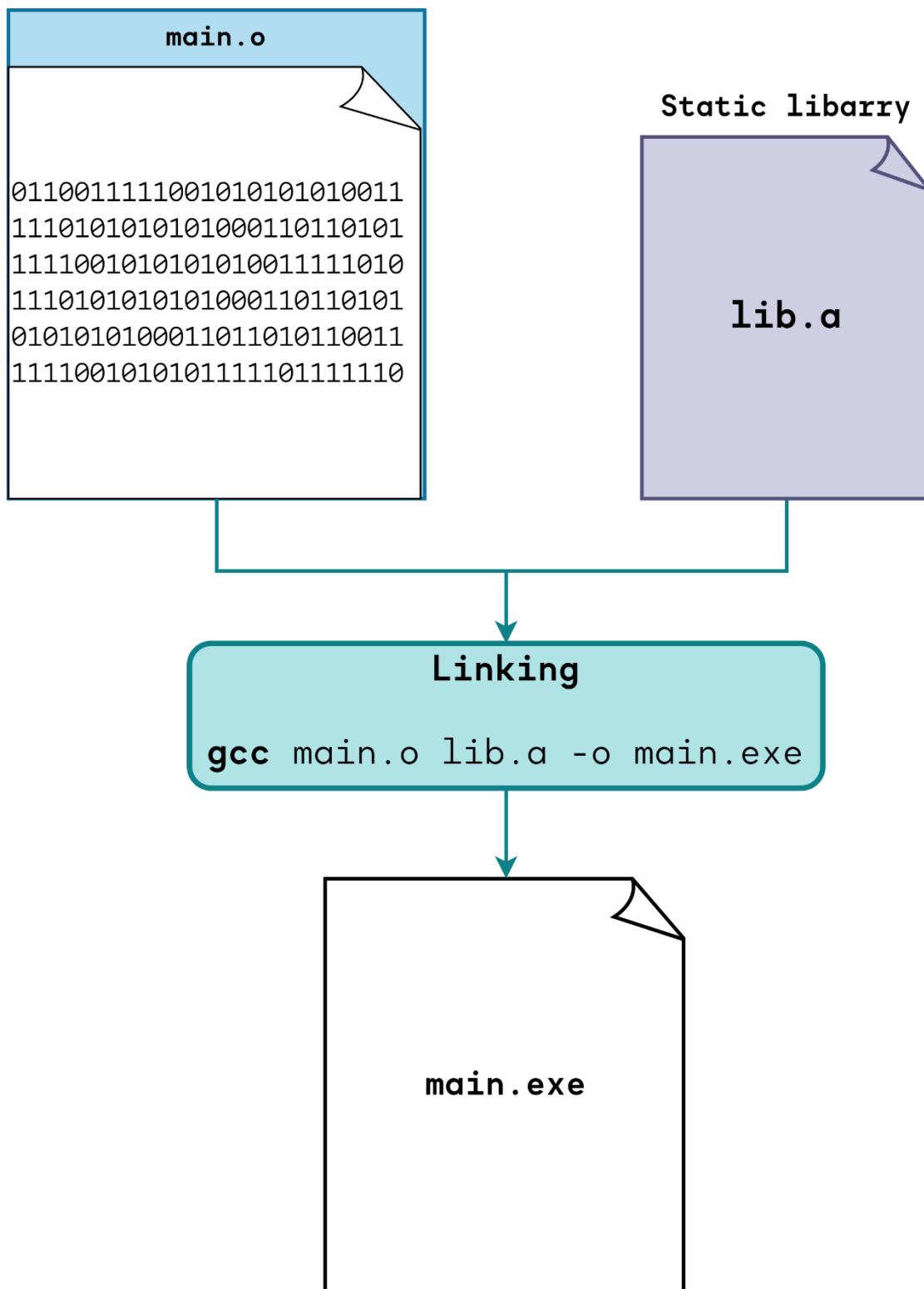
### Assemble

```
gcc -c main.s -o main.o
```

Translate assembly code  
into machine code

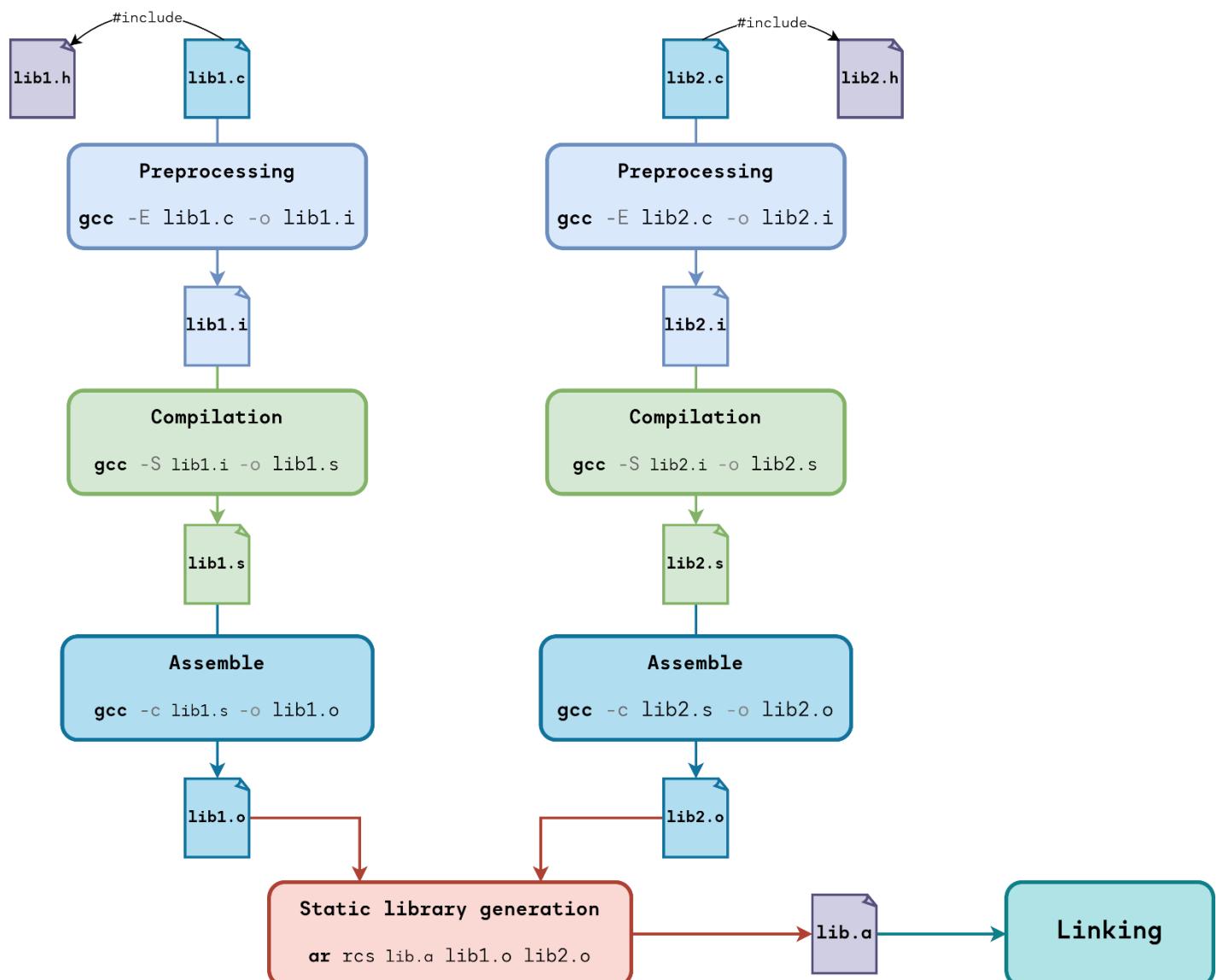
### main.o

```
011001111001010101010011  
1110101010101000110110101  
111100101010101010011111010  
1110101010101000110110101  
0101010100011011010110011  
1111001010101111101111110
```

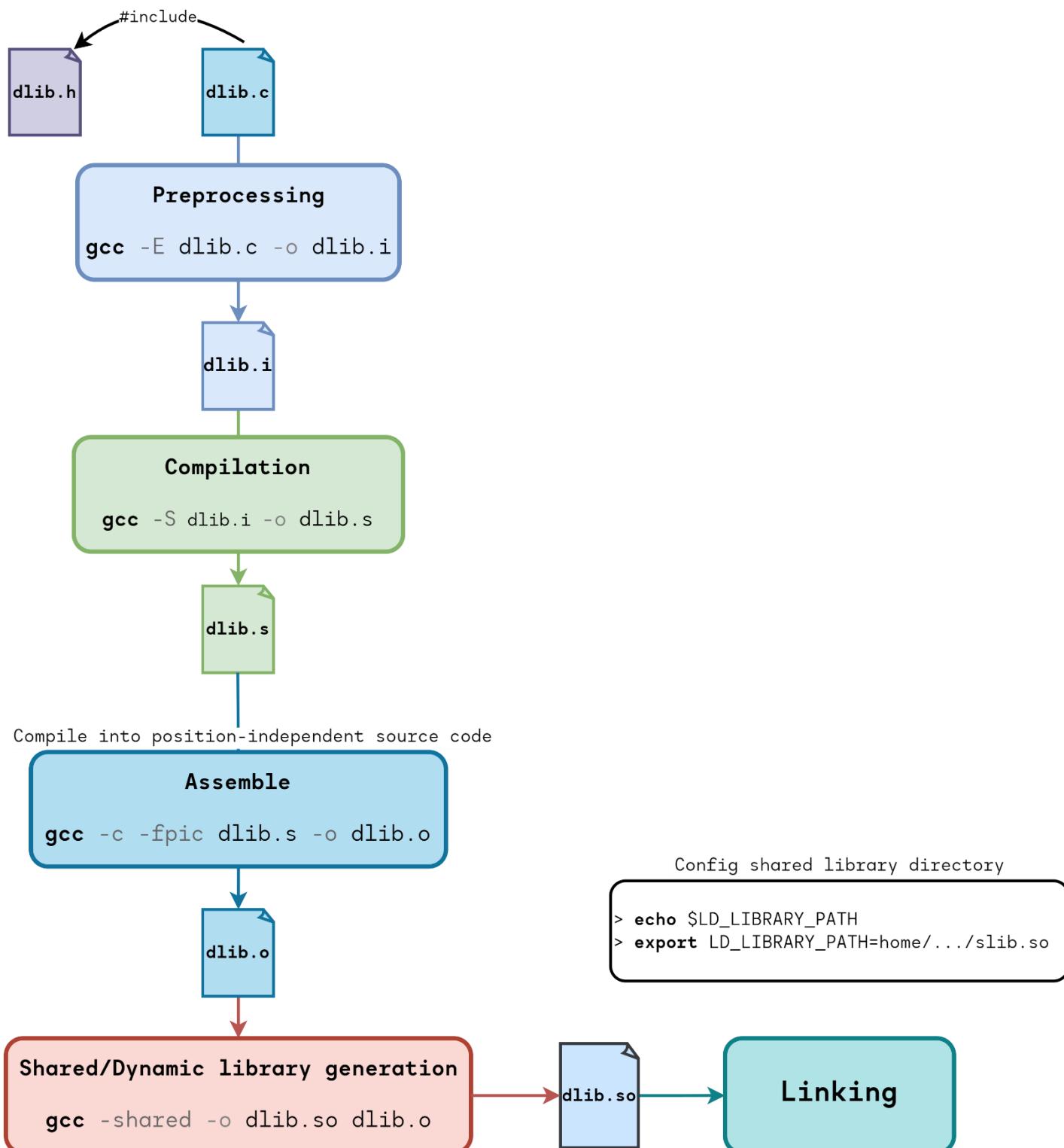
**Linking:**

### 3.2.4. Static library và Dynamic/Shared Library

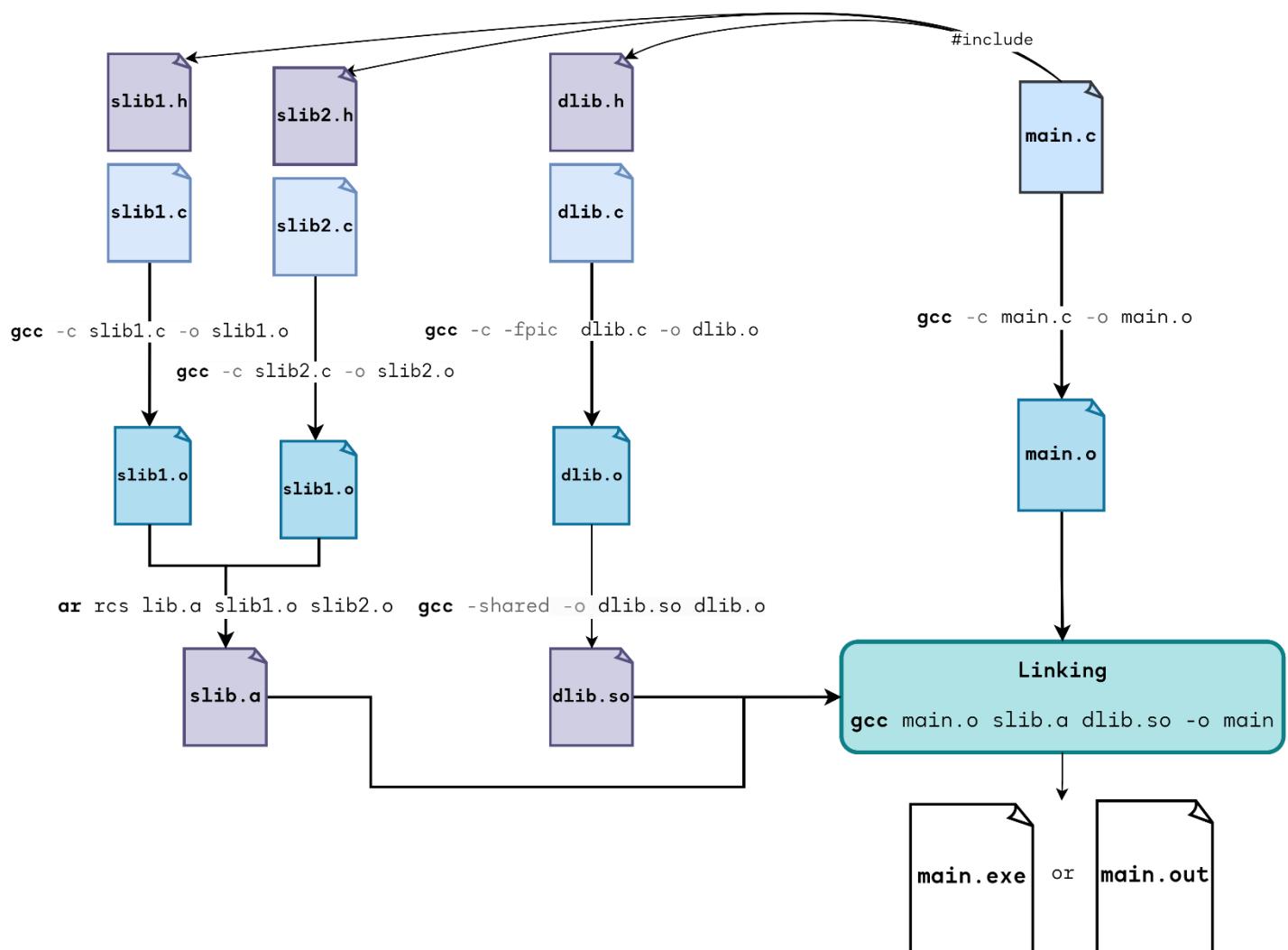
**Static library:** là các thư viện có đuôi .a hoặc .lib. Static library là file chứa các object .o được tạo từ source code. Ở linking, nếu các nội dung cần được thực thi nằm ở static library thì toàn bộ nội dung đó được copy vào file code cuối cùng.



**Dynamic library/Shared library:** là các thư viện có đuôi .so hoặc .dll. Dynamic library chứa các object được tạo từ source code. Ở linking, nếu nội dung cần được thực thi nằm ở dynamic library thì chỉ có phần địa chỉ tham chiếu đến nội dung được đưa vào file code cuối cùng. Trong quá trình chạy thì chương trình dựa vào tham chiếu đó để truy xuất đến nội dung chứa trong dynamic library.

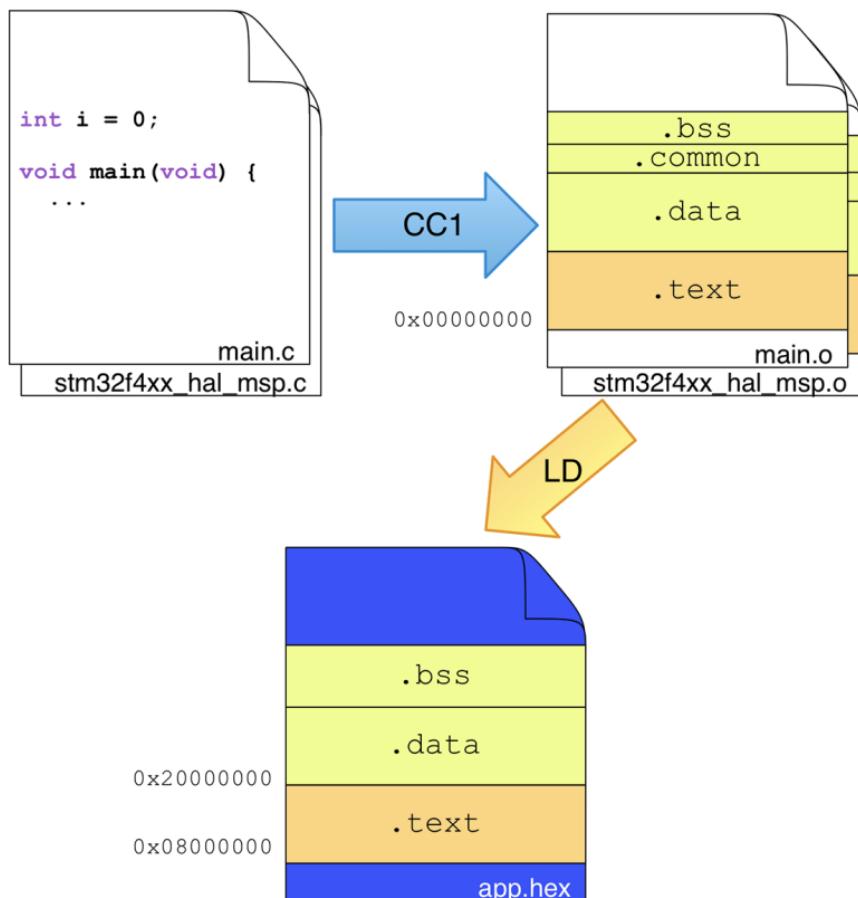


### 3.2.5. Compiling process của một project cơ bản sử dụng GCC



### 3.3. Memory Layout

3.3.1. Quá trình Biên dịch (Compilation) và Liên kết (Linking) của trình biên dịch ARM

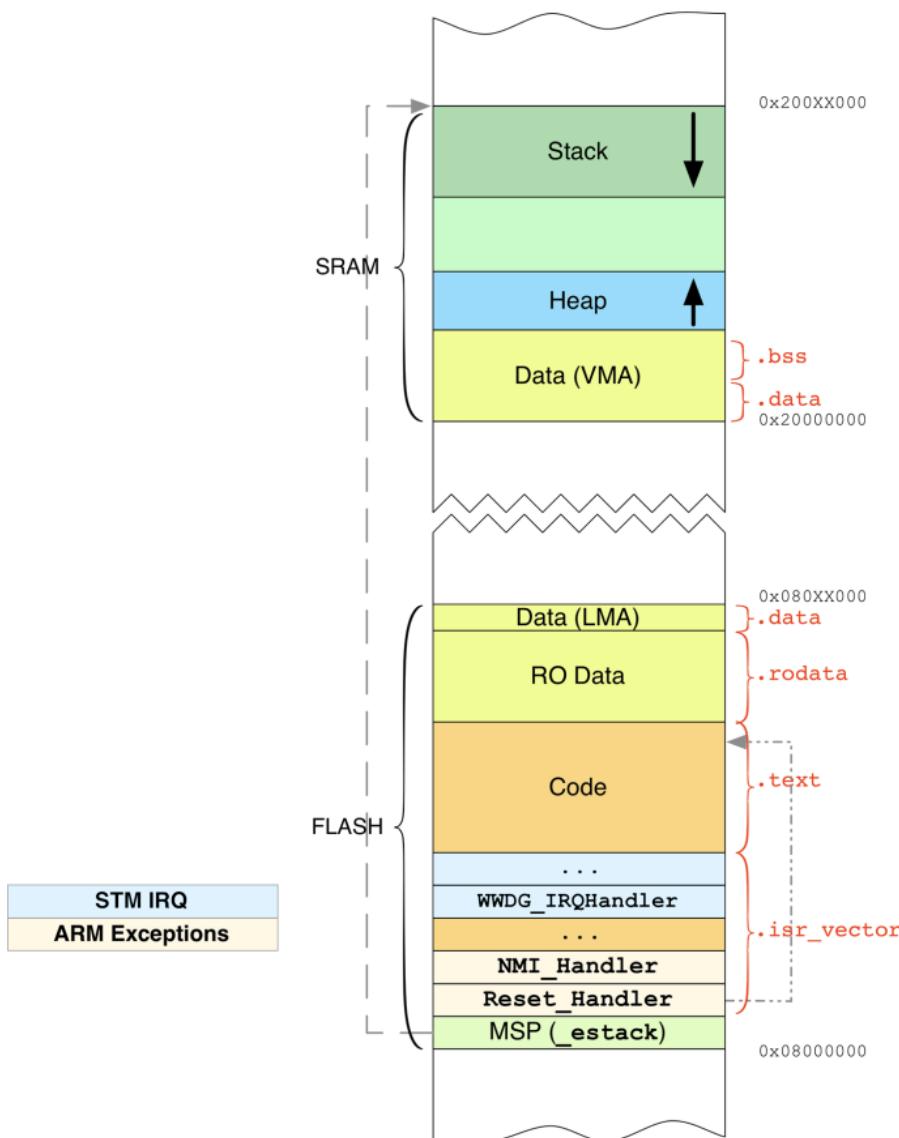


Quá trình biên dịch một C source file đến một file nhị phân đưa vào FLASH trên MCU bao gồm một số bước và công cụ đi kèm do trình biên dịch GCC cung cấp. Tất cả bắt đầu từ các C source file, chúng thường chứa các đối tượng sau:

- **Global variables:** biến toàn cục được chia thành hai loại: đã khởi tạo (initialized) và chưa khởi tạo (un-initialized). Cũng có thể được khai báo với từ khóa `static` cho biết phạm vi sử dụng của biến này chỉ ở trong file C hiện tại.
- **Local variables:** biến cục bộ được chia thành hai loại chính: simple local variable (còn được gọi là automatic variable) và static local variable (biến này có thời gian tồn tại suốt quá trình chạy của chương trình và giá trị sẽ không bị mất đi khi thoát hàm).
- **Const data:** có thể chia thành hai loại: const data types và string.
- **Routines:** chương trình sẽ được thực thi và chúng được dịch thành các assembly instruction.
- **External resources:** gồm global variable (được khai báo với từ khóa `extern`) và các routines được định nghĩa bởi các file source C khác.

Khi source file được biên dịch, tất cả các thành phần trên sẽ được đặt vào các sections của file nhị phân dựa vào file linker script.

Language structure	Binary file section	Memory region at run-time
Global un-initialized variables	.common	Data (SRAM)
Global initialized variables	.data	Data (SRAM+Flash)
Global static un-initialized variables	.bss	Data (SRAM)
Global static initialized variables	.data	Data (SRAM+Flash)
Local variables	<no specific section>	Stack or Heap (SRAM)
Local static un-initialized variables	.bss	Data (SRAM)
Local static initialized variables	.data	Data (SRAM+Flash)
Const data types	.rodata	Code (Flash)
Const strings	.rodata.1	Code (Flash)
Routines	.text	Code (Flash)



### 3.3.2. Virtual Memory Access (VMA) và Load Memory Access (LMA)

Virtual Memory Access là section nằm trong SRAM, sẽ được sinh ra khi firmware bắt đầu được thực thi.

---

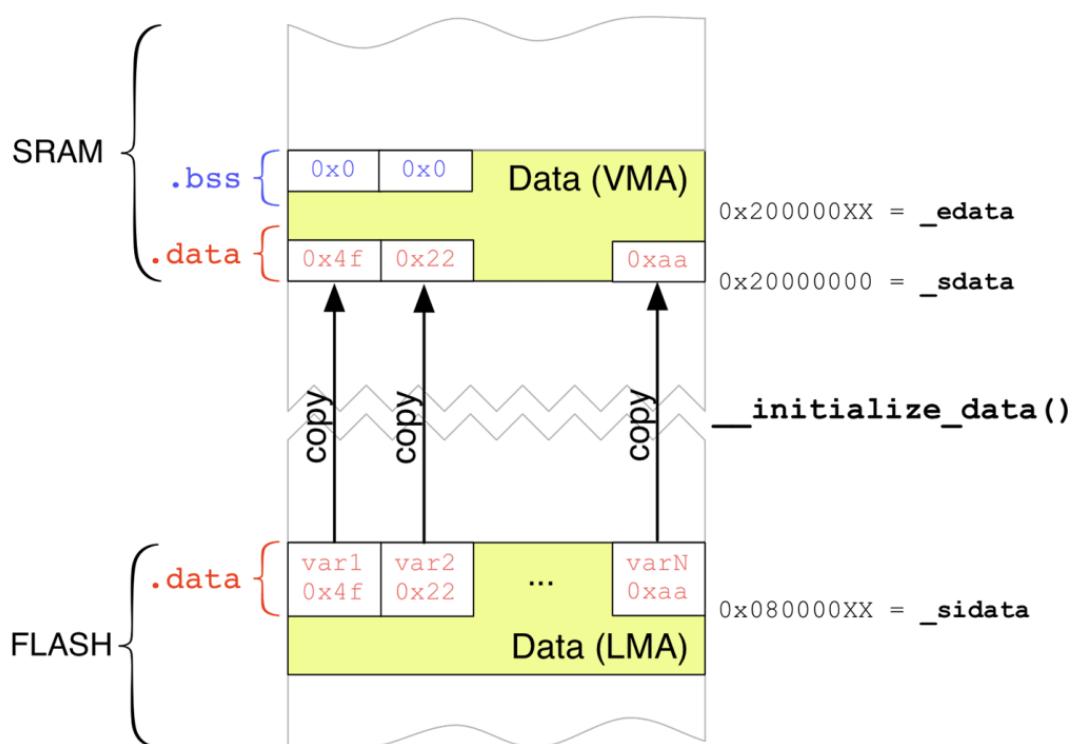
Load Memory Access là section nằm trong FLASH và sẽ được load vào VMA khi firmware bắt đầu được thực thi.

Sau khi chương trình chạy, .data section trong LMA sẽ được load vào VMA, đồng thời .bss section sẽ được khởi tạo trong VMA.

### 3.3.3. .data section

.data section chứa initialized variables: global initialized variables, global/local static initialized variables.

Ban đầu, .data section sẽ nằm được lưu trong FLASH tại LMA, khi bắt đầu thực thi thì .data section sẽ được copy vào VMA trong SRAM.



```

25 /* Used by the startup to initialize data */
26 _sidata = LOADADDR(.data);
27
28 .data : ALIGN(4)
29 {
30     . = ALIGN(4);
31     _sdata = .;           /* create a global symbol at data start */
32
33     *(.data)
34     *(.data*)
35
36     . = ALIGN(4);
37     _edata = .;           /* define a global symbol at data end */
38 } >SRAM AT>FLASH

```

Quá trình copy .data segment từ LMA ở FLASH sang VMA ở SRAM được thực hiện trong file startup trong hàm Reset Handler:

```
/* Copy the data segment initializers from flash to SRAM */
```

```
ldr r0, =_sdata
ldr r1, =_edata
ldr r2, =_sidata
movs r3, #0
b LoopCopyDataInit
```

**CopyDataInit:**

```
ldr r4, [r2, r3]
str r4, [r0, r3]
adds r3, r3, #4
```

**LoopCopyDataInit:**

```
adds r4, r0, r3
cmp r4, r1
bcc CopyDataInit
```

### 3.3.4. .bss (block starting symbol) section

.bss section chứa un-initialized variable: global un-initialized variables, global/local static un-initialized variable.

Section này được lưu tại VMA trong SRAM và nằm trên .data section, được hình thành khi firmward thực thi.

```

25  /* Uninitialized data section */
26  .bss : ALIGN(4)
27  {
28      /* This is used by the startup in order to initialize the .bss section */
29      _sbss = .;           /* define a global symbol at bss start */
30      *(.bss .bss*)
31      *(COMMON)
32
33      . = ALIGN(4);
34      _ebss = .;           /* define a global symbol at bss end */
35  } >SRAM AT>SRAM

```

Theo tiêu chuẩn của ANSI C, nội dung của .bss section phải được khởi tạo bằng 0 và điều này nằm ngoài khả năng của file linker script (do không có phân vùng FLASH nào chứa toàn bộ là 0). Vì vậy sau khi khởi tạo, ta sẽ gán 0 cho .bss section trong file startup tại hàm Reset Handler:

```

/* Zero fill the bss segment. */
ldr r2, =_sbss
ldr r4, =_ebss
movs r3, #0
b LoopFillZeroBSS

FillZeroBSS:
str r3, [r2]
adds r2, r2, #4

LoopFillZeroBSS:
cmp r2, r4
bcc FillZeroBSS

```

### 3.3.5. .rodata (read only data) section

.rodata section chứa các biến hằng số không thay đổi giá trị trong suốt quá trình hoạt động của chương trình: const data type, const string.

Section này được lưu trong FLASH (internal FLASH hoặc external FLASH được kết nối với MCU thông qua chuẩn giao tiếp Quad-SPI).

```
/* Constant data goes into flash */
.rodata : ALIGN(4)
{
    *(.rodata)          /* .rodata sections (constants) */
    *(.rodata*)         /* .rodata* sections (strings, etc.) */
} >FLASH
```

Ví dụ các định nghĩa sau sẽ được lưu trong .rodata section:

```
const char msg[] = "Hello World!";
const float vals[] = {3.14, 0.43, 1.414};
```

Phân biệt giữa hai loại định nghĩa string:

```
char *msg = "Hello World!";
...
```

đây là pointer to const array, hoàn toàn khác với:

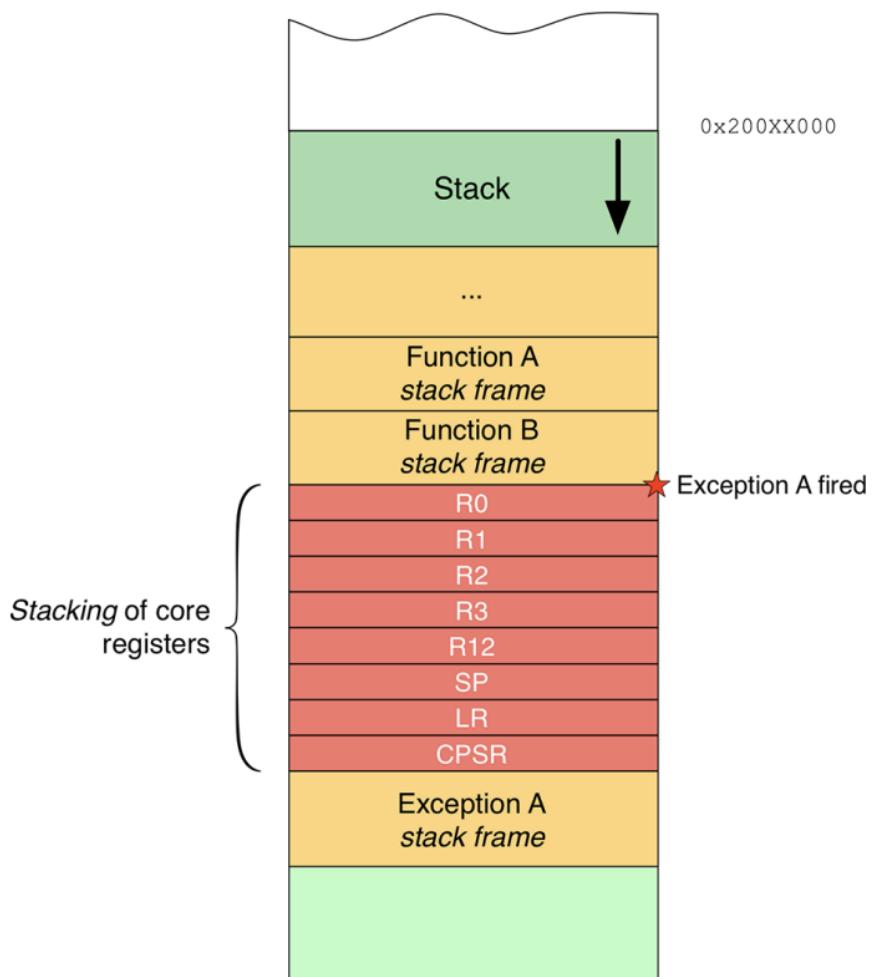
```
char msg[] = "Hello World!";
...
```

\*msg là một biến con trỏ được lưu trong .data section và đang trỏ đến địa chỉ của chuỗi "Hello World!" được lưu trong FLASH.

### 3.3.6. Stack region và Heap region

Stack là vùng nhớ nằm ở địa chỉ cao nhất của SRAM, đảm nhận vai trò vùng đệm thực thi chương trình chính, cụ thể là nơi thực hiện các routine trong thao tác gọi theo chuẩn calling convention của ARM.

```
_Min_Heap_Size = 0x200; /* required amount of heap */
_Min_Stack_Size = 0x400; /* required amount of stack */
/* User_heap_stack section, used to check that there is enough "RAM" Ram type
memory left */
._user_heap_stack :
{
    . = ALIGN(8);
    PROVIDE ( end = . );
    PROVIDE ( _end = . );
    . = . + _Min_Heap_Size;
    . = . + _Min_Stack_Size;
    . = ALIGN(8);
} >RAM
```



Vùng nhớ lưu trữ dữ liệu cấp phát động bởi các hàm như: malloc(), calloc(), free().