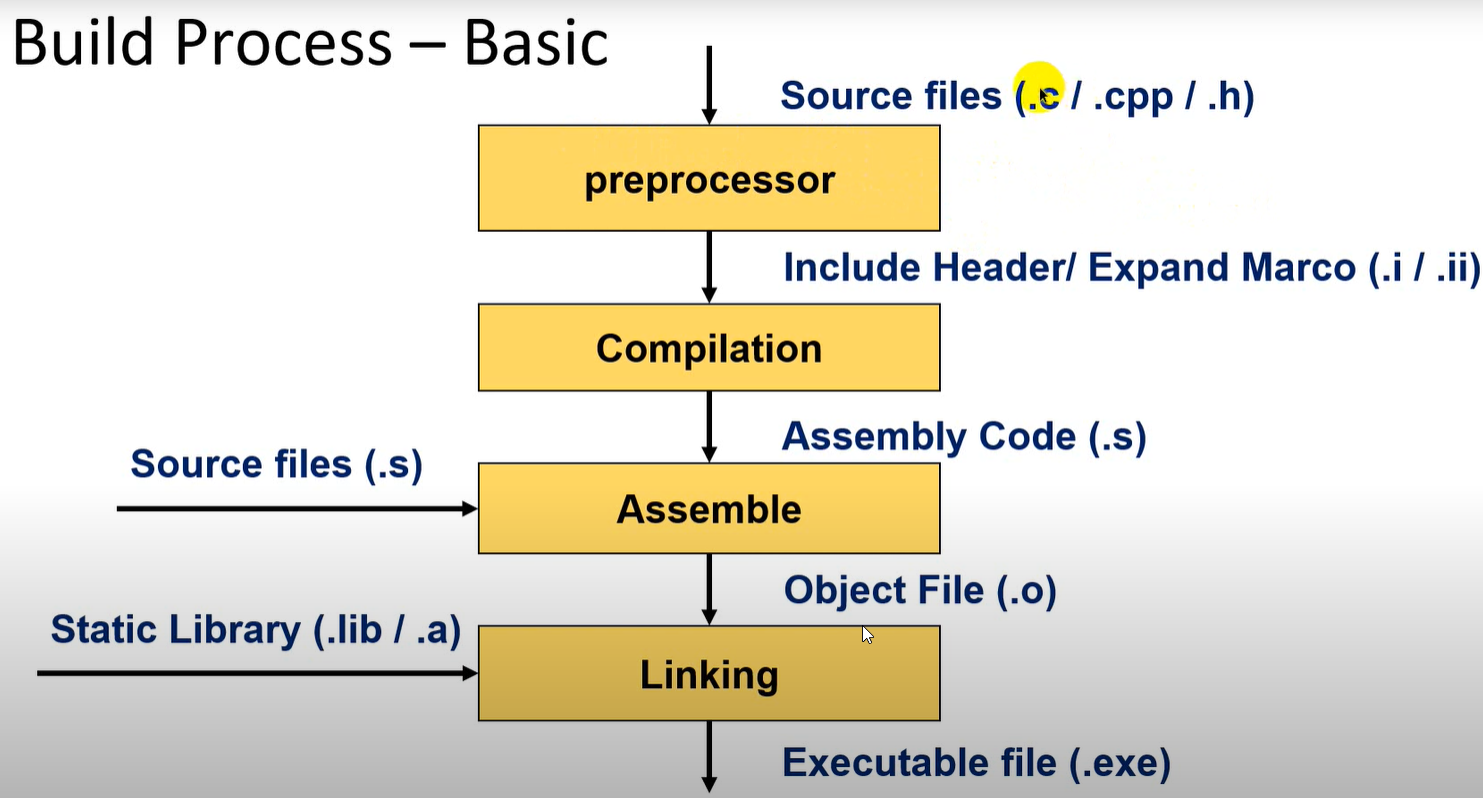
# BUILD PROCESS

## Build Process

### Khái niệm cơ bản



**Hình 1:** Build process cơ bản

Trong một chương trình C trên máy tính (window), các file source bao gồm .c, .h, .cpp sẽ trải qua các bước cơ bản - Tham khảo [**Build process with Command Line**](#_GCC_Command_for).

1. **Preprocessor**
2. Xử lý các chỉ thị tiền xử lý - thay thế các macro, thay thế nội dung file header vào các vị trí #include.
3. Xóa các comment
4. Đầu ra của quá trình này là file extended source (.i / .ii)
5. **Compilation**
   1. Các file source được Compile bởi Compiler tạo file Assembly (.asm).
   2. Compiler được sử dụng là các compiler thông dụng như GCC.
6. **Assemble**
   1. Các file .asm (từ bước trên / source file) tiếp tục được dịch bằng trình Assembler
   2. Đầu ra là các **file object (.o)**
7. **Linking**
   1. Các file .o được link với các file [**Thư viện tĩnh**](https://www.laptrinhdientu.com/2021/08/Library.html) (.a / .lib) bởi trình linker.
   2. Đầu ra là file .exe cuối cùng để chạy.

    👉 Nhưng trong một chương trình C nhúng (vi điều khiển) thì có một chút thay đổi

1. Compiler không compile ra file ASM của máy tính nữa, mà ở đây là Cross Compiler.  
   ***\*\* Cross Compiler:***Đọc về[***Cross Compiler***](#_Cross_Compiler)
2. 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**](https://www.laptrinhdientu.com/2021/08/LinkerScriptFile.html) để phân bố vùng nhớ cho vi điều khiển - Khác với máy tính thì IDE và Hệ điều hành đã tự động Locator và ẩn đi file Linker Script này.
3. 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 / .srec / .axf / ...** để nạp code và debug.

|  |
| --- |
| A diagram of a computer process  Description automatically generated |
| **Hình 1:** Quá trình compile các file source |

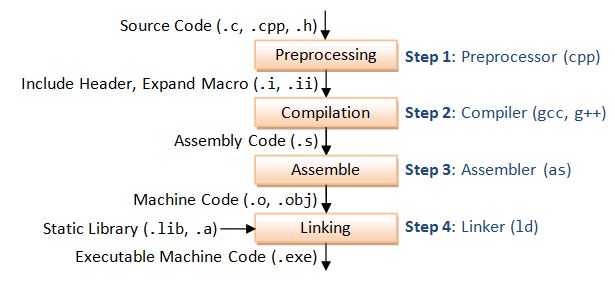
|  |
| --- |
| A diagram of a computer |
| **Hình 2:** Quá trình linking và locator |

## GCC Command for Build Process

### Build Process trong C?

[**Build Process**](#_Build_Process) là một quá trình xử lý với Input là các file source code (.c, .cpp, .h, .s, ...) và đầu ra là các file nhị phân để xử dụng (.exe, .elf, .hex, .bin, ...).

Build Process trong một chương trình C sẽ trải qua các bước sau:

[](https://lh3.googleusercontent.com/-1m64ZYzlqs0/YSOObRTgG5I/AAAAAAAAAqU/d_I0swanzZMSoWCXpomyL56UyX37EDMuwCLcBGAsYHQ/GCC_CompilationProcess.png)

### Trình biên dịch GCC

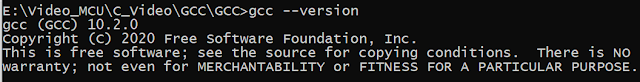
* GCC - GNU Compiler Collection là một tập hợp các trình biên dịch được thiết kế cho nhiều ngôn ngữ lập trình khác nhau. GCC là một thành phần quan trọng của GNU toolchain, và được hầu hết các hệ điều hành giống Unix, như Linux và Mac OS X chọn làm trình dịch tiêu chuẩn (Tham khảo Wiki).
* GCC là trình biên dịch chúng ta cần sử dụng cho **Step 2 - Compilation**, biến các file source code thành các file Assembly.

|  |
| --- |
| Trên thực tế thì GCC có thể dùng để giải quyết tất cả công việc trên 😁 |

### Sử dụng GCC để xử lý Build Process như thế nào?

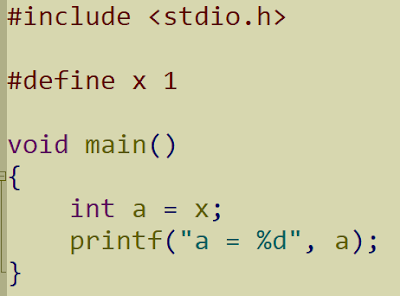
Cài đặt GCC trước. Nếu cài một số IDE như Codeblock, Dev C++ thì máy đã có sẵn GCC rồi. Còn không thì hãy thử search GCC download trên mạng nhé!

Cách kiểm tra GCC + phiên bản trên máy, các bạn mở cửa sổ **cmd**lên và gõ lệnh sau: ***gcc --version***

**[](https://lh3.googleusercontent.com/-5ELa6UKpmIc/YSOSrOxYjdI/AAAAAAAAAqc/NlOY4wdwBFkjL4jHCRVpdRPoEYVF4lp9QCLcBGAsYHQ/image.png)**

Sau khi đã có GCC thì ta sẽ bắt đầu từng Step như sơ đồ Build Process:

Đầu tiên mình có một file source code: **main.c.**

[](https://lh3.googleusercontent.com/-S5k_qN1TxzI/YSOTGaz9e5I/AAAAAAAAAqk/ZCUQXbCpQbYRBb90CV3UWSg9g1uF7otYQCLcBGAsYHQ/image.png)

Cần cd đến folder chứa Project của mình hoặc trỏ đến dường dẫn của folder đó và gõ cmd để hiển thị của sổ Command Line.

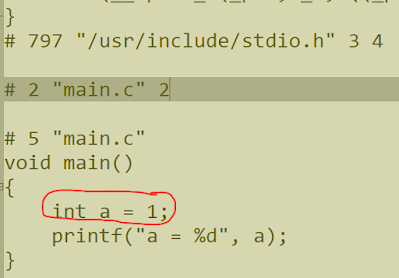
#### Step 1. Preprocessor

    Step này sẽ gộp các file source với file thư viện tương ứng (ở đây là stdio.h), cùng với đó là xử lý các chỉ thị tiền xử lý, ở đây là #define.

***Câu lệnh GCC cho Step này:***

|  |
| --- |
| **gcc -E main.c -o main.i** |

Output của Step này là Extended File - main.i như sau:

[](https://lh3.googleusercontent.com/-vhWbMLb9x8A/YSOVgwPIT_I/AAAAAAAAAq8/gyi4tsQcAQkQr2e2DW9his03w1kCHy5fACLcBGAsYHQ/image.png)

    File này rất nhiều nội dung, chủ yếu là nội dung của thư viện **stdio.h**. Cái chúng ta quan tâm là phần dưới cùng của file, phần tiền xử lý #define đã được thay thế.

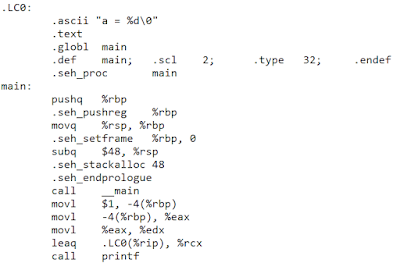
#### Step 2. Compilation

Quá trình này sẽ biến file mai.i trên thành file mã máy - Assembly.

***Câu lệnh GCC cho Step này:***

|  |
| --- |
| **gcc -S main.i -o main.s** |

Output là file **main.s** như này, nếu ai giỏi Assembly thì có thể đọc hiểu được:

[](https://lh3.googleusercontent.com/-4h5nfiPtNTk/YSOWvdo1A8I/AAAAAAAAArE/XsnfhvOrJFsmGQ00SFMb_pcdSIcKzAVAgCLcBGAsYHQ/image.png)

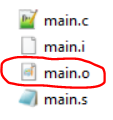
#### Step 3. Assembler

Step này sử dụng trình Assembler thay thì Compiler, nó gộp các file .s vừa tạo với các file .s có sẵn (nếu có) tạo thành các file object.

***Câu lệnh GCC cho Step này:***

|  |
| --- |
| **as main.s -o main.o** |

Output là file main.o, với nội dung và định dạng mà chúng ta không đọc được đâu 😂

**[](https://lh3.googleusercontent.com/-RkvW2BVgxDA/YSOYV1vmz5I/AAAAAAAAArU/GyVoFA2WiS05feCPqIA_AfEBXoXmOfKGwCLcBGAsYHQ/image.png)**

#### Step 4. Linking

    Sau khi có các file object, chúng sẽ được Link cùng với các [**static library**](https://www.laptrinhdientu.com/2021/08/Library.html) (nếu có).

***Câu lệnh GCC cho Step này:***

|  |
| --- |
| **gcc -v -o main main.o** |

Output của quá trình này là file **main.exe**dùng để run. Bây giờ chỉ cần gõ main, hoặc main.exe + Enter là chương trình sẽ chạy ra kết quả.

## Static Library - Shared Library

**Library**- thư viện là một tập hợp các file (chứa các hàm) để user dùng cho các chương trình khác nhau.

Thư viện bao gồm các**file header (.h)**chứa user-defined data types, macro, fucntion prototypes và các file.c tương ứng triển khai các hàm đó.

Một thư viện có thể bao gồm 1 hoặc nhiều object files, mang tính re-use và nhiều có nhiều lợi ích khác.

    👉 2 loại thư viện là static và shared sử dụng quá trình.

### Static Library

    Thư viện liên kết tĩnh - một file chứa các object được build từ source code (complier - time), các file objects này chứa các define của functions, variables.

    Những file Static Libraries có thể được linked với các file objects khác để tạo thành 1 file thư viện khác hoặc 1 file executable.

* Thường được cung cấp dưới dạng các file thư viện đã được build sẵn (.a, .lib) cùng với các file headers tương ứng.
* Các application sử dụng thư viện phải include header files và link tới các file thư viện.
* Nếu lib có sự thay đổi và ứng dụng đang sử dụng lib đó muốn cập nhật sự thay đổi thì bắt buộc phải build lại cả ứng dụng 🙁

[A diagram of a diagram of a library

Description automatically generated](https://blogger.googleusercontent.com/img/a/AVvXsEjc8d5qxt3XIrbGvGUBzLyN3RvcS_lwH2-izcd8yxnLN7ySKdVh_bXAOjxnCzpoBa1Nz2iRLlag5ifPNQnwppFatvErCwCMbHfLgU15H0jrChikMknp7hRdBigrK9Ap9ivLpINV8qHYnDB_KE823ZKKjE1x7oYgtTUQqeZZYF5rMoPoslrtqN9P7w10=s661)

Static Lib dùng trong quá trình build của 1 chương trình nhúng: [**Link truy cập**](https://www.laptrinhdientu.com/2021/08/Build.html).

Bên cạnh đó, nếu bạn mong muốn đóng gói một chương trình của mình thành một thư viện cho người khác sử dụng, nhưng lại không muốn họ biết source code của mình, thì Static Library là một phương án hữu hiệu. Một số người thường build library source của mình thành một Static Library (file .a) và cung cấp cho người dùng kèm các Header Files để họ sử dụng.

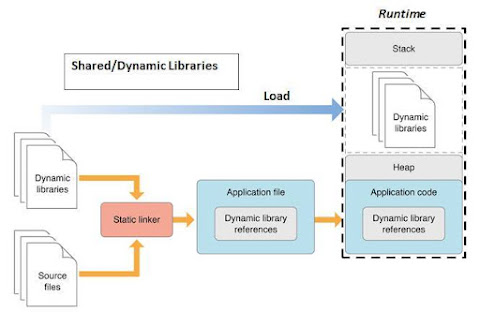
Thông thường thì các IDE sẽ hỗ trợ chúng ta Link các Lib này với các file object rồi, bằng cách trỏ đến folder chứa file .a. Nếu không chúng ta sẽ phải làm điều đó trong **[Makefile](https://www.laptrinhdientu.com/search/label/Makefile" \t "_blank)**.

Các bạn có thể tham khảo các tạo và sử dụng Static Library tại video này của mình:

### Shared Library / Dynamic Library

    Thư viện liên kết động là file chứa các objects được build từ source code, các file objects này chứa các define của functions, variables.

    👉 Cái này giống với static lib. Khác ở chỗ chúng được load vào file executable của application lúc runtime.

[](https://blogger.googleusercontent.com/img/a/AVvXsEjwAiWDsfjOKpJiS7ZRP1galmo3LxuB6oVQh3dREegwxlegrgii5lwbCZ4bF1M4csDsUi1IRF-ulkqOJiOIcMnxgiHDBHinweMLHoQsSakkM1IWm_ZOr02Xhk5do5ct94dIu2CbKlORLNIQB-DCmQ51u31i7L4cTWia1PtBuBcnXo1K1wwFdrlNb93B=s665)

    Vì nhiều chương trình có thể sử dụng chung các thư viện này nên nó được gọi là Shared Libraries.

    Khác với Static Lib, Shared Lib thường được dùng trong các ứng dụng máy tính nhiều hơn. Thường thì khi chơi game online chắc hẳn chúng ta sẽ thỉnh thoảng bắt gặp thiếu một file .dll nào đó. Đó chính là các Shared Library, thường thì chúng được build như các biến môi trường/các bản cập nhật/vá lỗi để game/ứng dụng hoạt động.

    Khi ta cài ứng dụng thì các file .dll này thường đi kèm với file .exe (Đúng với nguyên tắc khi ta chạy file .exe - runtime thì mấy file .dll này mới được). Nếu thiếu một số chúng thì ứng dụng sẽ không thể hoạt động được. Và dù đã run game một vài lần, thì việc xóa chúng đi cũng khiến lần sau chúng ta không thể chơi game được 😂😂😂 Rõ ràng rồi, vì khi run thì chương trình mới tìm đến mấy file này mà.

    Các bạn có thể tham khảo các tạo và sử dụng Shared Library tại video này của mình:

### So sánh Static Library vs Shared Library

    👉 (Hình ảnh) Shared Library được lưu trong bộ nhớ và sẽ được các process ánh xạ đến khi cần. Trong khi mỗi process dùng thư viện static cần có 1 bản copy của nó.

* Static library có ưu điểm hơn về mặt tốc độ thực thi nhưng shared library lại biên dịch nhanh hơn (xem lại phần static vs dynamic linking)
* Shared library có kích thước chương trình thực thi nhỏ hơn (vì chỉ cần copy 1 lần)
* Mỗi lần muốn update ứng dụng thì ứng dụng dùng static lib cần build lại cả chương trình, trong khi dùng shared lib chỉ cần nhớ build lại lib đó.

## Cross Compiler

    Hồi mới học về lập trình sau đó là lập trình vi điều khiển, mình nghĩ là việc biên dịch một chương trình C trên máy tính và Vi điều khiển là giống nhau, nhưng thực tế lại không phải như vậy @@ Chúng có một chút thay đổi để phù hợp với target mà file binary nạp xuống.

    👉 Lúc đó cầm Dev-C, Codeblocks build ra cái file .exe, rồi bên vi điều khiển là file .hex. Nghĩ là cái định dạng nó khác nhau thôi chứ nội dung file nhị phân vẫn thế.

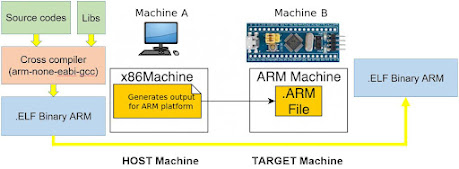
    👉 Thế rồi thấy người ta bảo học GCC mình cũng mày mò, ơ rồi ra nhị phân xong làm thế nào nữa? cho xuống vi điều khiển nó vẫn chạy đúng không?

    😅 Câu trả lời là không

    Nhìn sâu xuống dưới thì tập lệnh assembly của con 8051 với con STM32 nó khác nhau, chứng tỏ file nhị phân của cùng chương trình sẽ là khác nhau. Tức là mỗi con chip sẽ cần có những cách build chương trình ra file nhị phân khác nhau. Máy tính (của mình là chip Intel x86) nó cũng thế. Vậy chúng ta lập trình vi điều khiển trên máy tính sẽ cần dùng gì, thay vì cái GCC Compiler ???

    👉 Câu trả lời là cái ***Cross Compiler***, nó cho phép mình build chương trình trên một máy tính (chip khác - x86 Intel chẳng hạn), rồi nạp và chạy trên một chip khác (ARM).

Máy tính mà mình build chương trình được gọi là "host", trong khi môi trường máy tính (chip) chạy gọi là "target".

[](https://blogger.googleusercontent.com/img/a/AVvXsEjABQj36Ar06C2LGK44A6jWNVznyIW0xkcQcNTSCyCMqJrdZGJIUm_olsIbVc3O1E5UkcyukI7_hkuUSvG9Iuuetv0RHIv6SXaMbyjm80A9Zk7oD81ZpLlQZE0LRw7BrfNi4vsD7osynth9KMKZHvTZmLzxqBV2Ue6n9wKmDHWCsMkEvyd1PBmgI8p6=s1328)

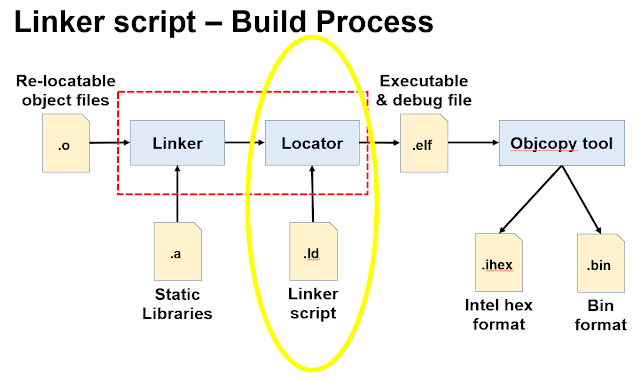
    😇😇😇

    Lan man quá, nhưng chip ARM (rất phổ biến) có cung cấp cái trình biên dịch riêng cho mình, gọi là **GNU Arm Embedded Toolchain**. Mọi người có thể tham khảo để sử dụng :)) Có thể mình sẽ review cách sử dụng ở những bài viết sau.

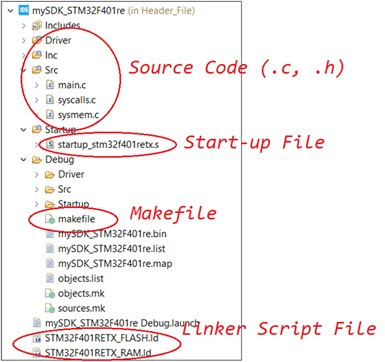
## Linker Script File

### Linker Script File là gì?

    Như ở bài **Build Process** mình đã giới thiệu về quá trình Linking  và Locator, thì quá trình locator là một quá trình rất quan trọng trong lập trình vi điều khiển, quá trình này được thể hiện bằng một ***Linker script file***.

[](https://lh3.googleusercontent.com/-E1Xxx1T8SMk/YSoY8xn8VyI/AAAAAAAAAu4/zrbDl6uHAW8g6S-eM_mRMawO5weE4pGtgCLcBGAsYHQ/image.png)

    Một file Linker script là một file không thể thiếu trong một project STM32, khi các bạn lập trình trên STM32CubeIDE, bạn có thể dễ dàng nhìn thấy các file **.ld** như sau <[**Bare - metal Example**](https://www.laptrinhdientu.com/2021/08/BareMetalExample.html)>

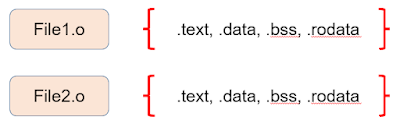
[](https://lh3.googleusercontent.com/-eSrQUrlJTRo/YSoafSYo-EI/AAAAAAAAAvA/yne9HJU4nnQD6lBoxv9Qu32Q3KXL4tuxQCLcBGAsYHQ/image.png)

    Một Linker Script File (.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).

    Ví dụ: Vùng flash bắt đầu từ đâu, độ dài bao nhiêu, vùng text, data trên đó chiếm bao nhiêu dung lượng, ... tất cả được quy định trên linker script file.

### Tại sao cần có Linker Script File?

    Như các bạn biết trong quá trình [**Build Process**](https://laptrinhcnhung.blogspot.com/2021/08/Build.html) thì các file **.o**sẽ được tạo ra sau quá trình Compile  và Assemble. Quá trình Compile này sẽ tạo ra địa chỉ tương đối cho mỗi file object, bao gồm text, data, bss, rodata, tức là mỗi file object đang bắt đầu từ địa chỉ 0x0000.0000.

[](https://lh3.googleusercontent.com/-GTkVd4ZFrcQ/YSocVGAYnyI/AAAAAAAAAvI/dRu0xq4U33w0tKigUZNIKY-GjdC6bcBOACLcBGAsYHQ/image.png)

    Như vậy, các file objects có thể bị conflict lẫn nhau. và địa chỉ này sẽ không phù hợp với địa chỉ mong muốn mà ta muốn nạp xuống vi điều khiển.

    Vì vậy, cần merge các file object trên với nhau, và phân bổ vào bộ nhớ vi điều khiển như người dùng mong muốn.

    Và file giúp làm điều đó chính là file Linker Script (.ld).

### Các lệnh trong Linker Script File

    Cấu trúc một file linker các bạn có thể xem trong một project stm32 bất kỳ. Về cơ bản thì nó bao gồm các lệnh.

#### ENTRY(\_symbol\_name\_) Command

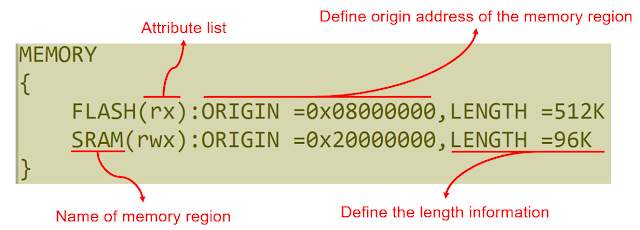
    Dùng để set thông tin *"Entry point address"*ở đầu file .elf. Debugger sẽ dùng entry point này để biết hàm nào được chạy đầu tiên.

    Ví dụ hay được dùng nhất là **Reset Handler**.

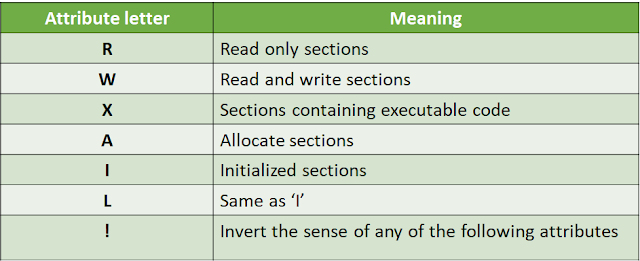
[Linker](https://lh3.googleusercontent.com/-AV-e30YP-uM/YSo9tE-BzWI/AAAAAAAAAvY/GiDG0Ru53QkMq6Z40KkpBE3ZmBK7A4uvwCLcBGAsYHQ/image.png)

#### MENORY Command

    Lệnh này dùng để set các phân vùng bộ nhớ, bao gồm thông tin địa chỉ bắt đầu & kích thước của các vùng nhớ tương ứng, cùng với đó là các Attribute (cho phép đọc/ghi).

[](https://lh3.googleusercontent.com/-gjixHX5A-gA/YSo-k7mQ73I/AAAAAAAAAvg/EBLKn_QR-HgvG9Lv3IPIOEZQlaoXb_aqQCLcBGAsYHQ/image.png)

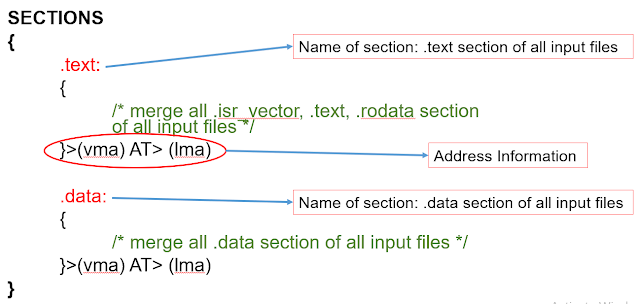
 Dưới đây là một số Attribute thường dùng:

[](https://lh3.googleusercontent.com/-jqerdBKRofE/YSo-9ARyQEI/AAAAAAAAAvs/JFwAQbvRkI8WIHg_rlxyQCohFCOBI0LFwCLcBGAsYHQ/image.png)

#### SECTION Command

    Các thành phần trong bộ nhớ sẽ được chia thành các Section, vì vậy lệnh SECTION trong linker file được dùng để định nghĩa các phân vùng nhớ này.

    Ví dụ: Các vùng .text, .data, .bss trong vùng nhớ FLASH.

[](https://lh3.googleusercontent.com/-sKrErN_umMA/YSo_noSp2GI/AAAAAAAAAv4/h8b0f6YucWE-8go6Sa4PKdfVulqLQ-MtACLcBGAsYHQ/image.png)

#### AT> Command

    Lệnh AT này được sử dụng sau mỗi section ở hình trên, được dùng để định nghĩa hai vùng nhớ trước và trong run-time.

    Ở ví dụ trên, vùng nhớ data trước khi run-time được phân bổ ở ***vma (virtual memory address)*** và được phân bổ trên ***lma (load memory address)*** khi run-time.

    Trên đây là một số kiến thức cơ bản về Linker Script File, để các bạn có thể đọc hiểu và sửa được file Linker, còn nếu muốn viết file Linker thì nên tìm hiểu thêm nữa.

# MAKEFILE

## Makefile Overview

### Tổng quan

Đối với các chương trình nhỏ, các bạn quen để tất cả hàm trong một file main.c, nhưng khi gặp 1 chương trình lớn, nhiều driver, nhiều người viết, ... thì việc quản lý các file trong nhiều folders khác nhau sẽ gặp rất nhiều vấn đề.

Trình biên dịch GCC cho phép build các file tuân thủ theo [**Build Process**](#_Build_Process_1). Vì vậy, khi build và link các file trong các folder khác nhau, thì mỗi lần build/link ta đều tuần theo một kịch bản cố định, dù có sửa nội dung source code.

👉 Người ta tạo ra một script file để tối ưu hóa các quá trình trên => ***Makefike.***

Makefile là một dạng script file chứa các thông tin:

* Project Structure (File, Dependencies).
* Các lệnh để tạo files (Object files, elf/bin/hex files, ...).
* Lệnh ***make*** để đọc nội dung makefile, hiểu cấu trúc project, và thực thi các lệnh.

### Tools hỗ trợ Makefile

**Tools hỗ trợ Makefile**: Cygwin/MingW/GNU Make/...

**[Tải Cygwin tại đây!](https://www.cygwin.com/" \t "_blank)**=> Khi cài các bạn nhớ chọn **make**rồi mới install.

[A black screen with white text

Description automatically generated](https://lh3.googleusercontent.com/-ifCufnLfrok/YRzcmqg9SAI/AAAAAAAAAe4/H1VXNKSSwJIiec_ehlyrcVHT_NAGRRKHACLcBGAsYHQ/image.png)

### Từ GCC Command đến Makefile

   Để học Makefile thì cần có kiến thức về GCC và Command line, những điều đơn giản cần nhớ:

* Lệnh **cd**, nhảy đến thư mục chỉ định
* Dấu **.**là chỉ chỗ mình đang đứng :D Còn **..**là back về một folder.

    Các bạn có thể đọc thêm về [**GCC Command tại đây!**](https://www.laptrinhdientu.com/2021/08/GCCforBuildProcess.html)

Bình thường cần chạy lệnh với của sổ cmd trên Window để tạo ra một file .exe từ một file source .c.

Ví dụ "Helloworld"

[A screen shot of a computer code

Description automatically generated](https://lh3.googleusercontent.com/-twRM8PdOTfY/YRzeTBvXkFI/AAAAAAAAAfI/9hx_n6D3dvQ_inhAphLUlU-u05QbvK85ACLcBGAsYHQ/image.png)

Công việc cần làm

[](https://lh3.googleusercontent.com/-dqtVHpGXbDI/YRzedxNar-I/AAAAAAAAAfM/z1Cs5d2PxdsS612D7FAsUu0E1B2QYK8dACLcBGAsYHQ/image.png)

* Nếu sử dụng GCC, một cách đơn giản là tuân thủ 3 bước: Tạo object file, sau đó tạo file .exe, rồi chạy file .exe này.
* Nếu project lớn, nhiều file, cần sửa đổi nhiều, thì sẽ phải chạy đi chạy lại các lệnh như vậy nhiều lần.

=> Phiền => Hãy thử dung **Makefile.**

[A close-up of a computer

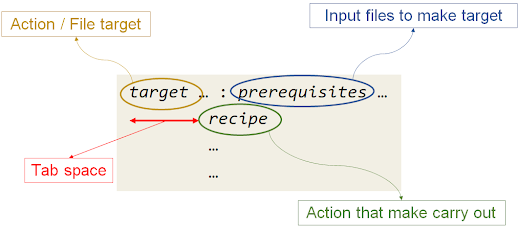
Description automatically generated](https://lh3.googleusercontent.com/-H6Bz9Ty4oSU/YRze4-tvpPI/AAAAAAAAAfY/i86cXQvO5eU7awqlDuLInOaa1meTk3DxQCLcBGAsYHQ/image.png)

Make file thực chất là tổng hợp các GCC Commands trong một script file (cấu trúc như hình trên).

Để chạy Makefile trên, chúng ta có thể mở cmd lên, cd đến makefile, và chạy thử lệnh **make** (lệnh này sẽ cho chạy cả 3 dòng màu xanh main.o, build, run) => Kết quả sẽ là "Hello"

Hoặc có thể chạy riêng từng dòng (từng option): **"make main.o", "make build", "make run"**. Mỗi option & Câu lệnh tương ứng như vậy gọi là một **Rule.**

### Cấu trúc một Rule đơn giản trong Makefile

[](https://lh3.googleusercontent.com/-22-3c1YM4H0/YRzcv1qxGMI/AAAAAAAAAe8/aVAKpIu6UJMZqBasABwOeOY5Nj9fGaCNQCLcBGAsYHQ/image.png)

Trong đó:

* **prerequisites**: Là các files cần thiết để tạo ra **target**
* **Target**: Là file được hình thành sau quá trình make được thực hiện, hoặc một **action name**: Là các hành động như build, run, ...
* **Action**: Là câu lệnh Compile để tạo ra **target** từ **prerequisites**, hoặc thực hiện target. Action được thụt lùi vào 1 tab so với **target**.

**VD 1:**

[A diagram of a program

Description automatically generated](https://lh3.googleusercontent.com/-wOaoLvWbqfo/YRzhWCNzw7I/AAAAAAAAAfg/HTeBkrfTlfsCV_5eSaTcHOK9lOmjWghFgCLcBGAsYHQ/image.png)

## Makefile Variables

Ở bài trước mình đã giới thiệu [**Tổng quan** **Makefile**](#_Makefile_Overview) và các Rule trong Makefile. Bài này sẽ giới thiệu về các Variables trong Makefile.

### Tại sao cần dùng Variable?

Một makefile có thể dùng để quản lý một project lớn, có thể bao gồm các folders: Source Files, Header Files, Object Files, Output Files, ... Các folder này có thể cố định, nhưng không điều gì đảm bảo các file source trong đó không thay đổi. Vì vậy, cần tổng quát hóa makefile qua các folders thay vì các files đơn lẻ.

👉 Người ta sử dụng **Variable** để tổng quát hóa Makefile, tránh sử dụng những tên file cụ thể.

#### Khởi tạo biến

Sử dụng dấu gán **:=** hoặc dùng dấu **=** để khởi tạo biến

**Cú pháp:**

|  |
| --- |
| Variable\_Name := Values |

**VD 1:**

[A close-up of a computer code

Description automatically generated](https://lh3.googleusercontent.com/-tjhSSf6pSeY/YRzqb8eQytI/AAAAAAAAAfw/mlgk8BsnImEaYPiP2NIhsL_T7cPEG-KbQCLcBGAsYHQ/image.png)

Nhìn chung, nó cũng khai báo giống như biến trong C, nhưng không có kiểu dữ liệu.

#### Lấy giá trị của biến

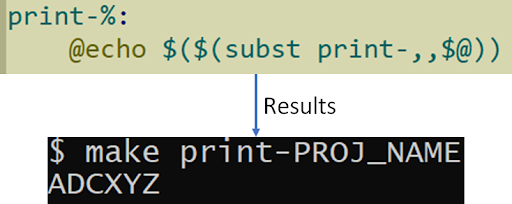
Để lấy giá trị của một **Variable**, ta sử dụng cú pháp:

|  |
| --- |
| $(Variable\_Name) |

[A math equation with words

Description automatically generated with medium confidence](https://lh3.googleusercontent.com/-XrUTXp-TaPY/YRzrmLWBTwI/AAAAAAAAAf4/EyaC5olWVJg5PyuXTdZnI4FA339cmBOrQCLcBGAsYHQ/image.png)

Ta cũng có thể kiểm tra tên của một biến bằng target **print** <Dùng lệnh **echo**>

[](https://lh3.googleusercontent.com/-atWNJYHL4fk/YRzsTdjOHQI/AAAAAAAAAgA/W9v2oxvRF4EBoFOKCYxpi8qX9lGGfWYlgCLcBGAsYHQ/image.png)

Ví dụ: **make print-PROJ\_NAME**

### Các phép toán của biến

#### Sử dụng dấu +=

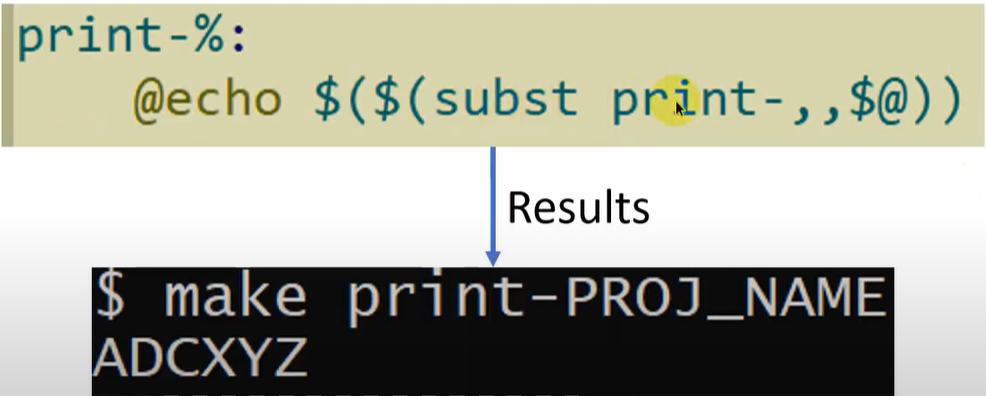
### In giá trị của biến

#### Sử dụng @

#### Sử dụng @echo

### Action printf

Sử dụng hàm subst



Tài liệu make file

### Automatic Variables

Để tổng quát hóa hơn nữa tránh sử dụng tên các file, người ta sử dụng các Automatic Variables:

|  |  |
| --- | --- |
| **Ký hiệu** | **Ý nghĩa** |
| $@ | Biểu thị cho tên file của **target** thuộc **rule** |
| $^ | Biểu thị cho tên của tất cả các **prerequisite (file input)** |
| $< | Biểu thị cho tên file đầu tiên trong **prerequisite** |

### Ví dụ sử dụng Automatic Variables

[A screenshot of a computer

Description automatically generated](https://lh3.googleusercontent.com/-MieHnklyvwI/YRzuy3_xxVI/AAAAAAAAAgY/8AA2hsqI_1kTYimXARlxs8w2d-nKNnKegCLcBGAsYHQ/image.png)

## Từ Khóa

### PHONY

## Makefile Functions

Các function trong makefile giúp việc truy xuất vào các files, folders trở nên dễ dàng hơn, thay vì sử dụng các đường dẫn trực tiếp. Điều này giúp tổng quát hóa các tệp giống như [**Variable mà mình đã nói ở bài trước**](https://www.laptrinhdientu.com/2021/08/Make.html).

Mình sẽ giới thiệu về một số Functions thường được sử dụng như wildcard, forech, notdir, ...

### "wildcard" Character

Ký tự wildcard được tạo ra bởi các ký tự **'\*', '?', '[...]'**.

Ví dụ, **\*.c** đại diện cho tất cả các file có đuôi là **.c**

[A screenshot of a computer

Description automatically generated](https://lh3.googleusercontent.com/-EKPGjzzyfeQ/YRzxYfsGkDI/AAAAAAAAAgg/ZjNNXeY3nVkTrEYLLRba4Da4_Pj_z4HvwCLcBGAsYHQ/image.png)

### foreach

Cú pháp: **$(foreach var, list, text)**

Function **foreach** có thể làm cho một đoạn văn bản (var) có thể sử dụng lặp lại nhiều lần (list) mỗi thời điểm nó sẽ đại diện cho một chuỗi khác nhau (list).

[A close up of a text

Description automatically generated](https://lh3.googleusercontent.com/-9vFiohcz-XA/YRz1fUT50OI/AAAAAAAAAgo/DYcxT3RlG0sQ6ejDFNyGdN1awHbfftABQCLcBGAsYHQ/image.png)

Một ví dụ tiêu biểu là dùng để get toàn bộ các file trong các folder a, b, c, d thuộc **dirs**.

Ở đây **var = dir** sẽ lần lượt được gán các giá trị **a, b, c, d** thuộc **dirs**. Giá trị này sẽ được thay thế bằng đoạn **$(wildcard $(dir)/\*).**

Ví dụ trên sẽ tương tự như việc dung **wildcard** như sau:

[](https://lh3.googleusercontent.com/-Fv2LRt6REVQ/YRz2soxvtXI/AAAAAAAAAg4/pFCKFq__9h04Uvdxb7jHPaSDzWHlUgeSQCLcBGAsYHQ/image.png)

### notdir

**notdir** viết tắt của not-direction, tức là không có đường dẫn. Dùng để lấy ra tên file cuối cùng mà bỏ qua đường dẫn của nó.

**Ví dụ**

[A screen shot of a computer

Description automatically generated](https://lh3.googleusercontent.com/-caqUkuv2LkU/YRz5KnPKRfI/AAAAAAAAAhI/yx-RXXWguqoF7C2dkrt_aThPXHe0LcxdwCLcBGAsYHQ/image.png)

