# MAKEFILE

## COMPILER

### Compiler là gì ?

Compiler hay còn gọi là **Trình biên dịch** có thể được hiểu là công việc dịch chuỗi câu lệnh được viết từ một ngôn ngữ lập trình thành chương trình tương đương dưới dạng ngôn ngữ máy tính, thường là ngôn ngữ ở cấp thấp hơn. Đơn giản dễ hiểu thì có thể tạm nói là nhờ Complier này mà file .c chúng ta viết mới được dịch thành file .hex .bin để nạp được xuống một MCU bất kỳ.

### A diagram of a computer system Description automatically generatedQuá trình biên dịch

**Hình 1.1** Quá trình biên dịch

Chúng ta có thể xem sơ đồ chi tiết các bước từ Code/Build/Run ở hình sau

A diagram of a computer program

Description automatically generated

**Hình 1.2** Quá trình Code/Build/Run

Thông thường nếu dùng chương trình để lập trình như Keil C chẳng hạn thì chỉ cần ấn một nút Build/Run xong là chúng ta chỉ việc ngồi chờ và chương trình được nạp trực tiếp vào chip luôn, nhưng ẩn đằng sau những nút này là một loạt hoạt động theo các bước như hình trên.

### Cross Compiler/ Toolchain là gì ?

Cross Compiler hay còn gọi là Toolchain có thể được hiểu là một source code được viết trên máy tính chạy trên chip Intel, sau khi thông qua một cross compiler sẽ cho ra file nhị phân có khả năng chạy được trên một nền tảng chip khác là ARM. Một ví dụ cơ bản nhất là mình đã dùng một máy tính hệ điều hành Ubuntu để build ra một file image có thể chạy trên Raspberry Pi

A diagram of a computer

Description automatically generated

Qúa trình tạo ra và sử dụng cross compiler/ tool chain có liên quan tới 3 đối tượng

* **Build**: hệ thống tạo ra tool chain, thường là các máy tính dùng chip Intel và hệ điều hành Linux hoặc Windows
* **Host**: hệ thống chạy tool chain để compile source code, host cũng giống build thường là các máy tính dùng chip Intel và Windows hoặc Linux là hệ điều hành.
* **Target**: là hệ thống chạy chương trình do host tạo ra, thường target là các máy tính nhúng dùng chip ARM, tuy nhiên nó cũng có thể là một máy tính bình thường dùng chip Intel.

A diagram of a computer system

Description automatically generated

Vậy các thành phần của Cross Compiler là gì ?

A diagram of a computer program

Description automatically generated with medium confidence

* **Binutils**: Là một tập các công cụ để tạo và quản lý file nhị phân (bin) của target CPU
* as : là assembler, nó sinh ra mã nhị phân (binary code) từ assembler source code
* ld : trình liên kết (linker)
* ar, ranlib : sinh ra file nén .a, sử dụng như là thư viện
* objdump, readelf, size, nm, strings: phân tích file nhị phân
* strip : để loại bỏ những phần thừa trong file nhị phân để giảm kích thước của chúng

Thông thường để cross-compiler một chương trình ta phải cài đặt biến môi trường mới có thể compile đúng được  
Ví dụ

$ export PATH=/path/to/compiler/bin:$PATH

$ export CROSS\_COMPILE=arm-none-linux-gnueabi-

$ export CC=${CROSS\_COMPILE}gcc

$ export CXX=${CROSS\_COMPILE}g++

$ export CPP=${CROSS\_COMPILE}cpp

$ export AR=${CROSS\_COMPILE}ar

$ export AS=${CROSS\_COMPILE}as

$ export LD=${CROSS\_COMPILE}ld

$ export RANLIB=${CROSS\_COMPILE}ranlib

$ export STRIP=${CROSS\_COMPILE}strip

Shell

* **C/C++ Library**  
  Library được dùng làm interface giữa applications và kernel, cung cấp các C API chuẩn để dễ dàng phát triển ứng dụng. Một số libb có thể kể đến như: glibc, uClibc, eglibc, dietlibc, newlib, …

A diagram of a computer application

Description automatically generated with medium confidence

* **Kernel header**  
  Cung cấp các API cần thiết cho Applications và C Library giao tiếp với Kernel.

A diagram of a computer application

Description automatically generated

* **GCC compiler**  
  gcc, c++, g++ : compiler  
  Trình biên dịch trong hệ thống Linux, compile cho rất nhiều ngôn ngữ và nhiều kiến trúc CPU khác nhau như ARM, MIPS, PowerPC, SuperH, x86; tuy nhiên mình chỉ đề cập đến ngôn ngữ C/C++ và kiến trúc CPU là ARM và x86.
* GDB Debugger: Trình gỡ rối, trợ giúp cho quá trình phát hiện lỗi khi develop application.

## Ví dụ với GCC Compiler

### Cài đặt GCC

Trước hết, mình thực hiện các bước với GCC trên máy tính dùng **Ubuntu**nhé

Thực hiện check version hiện có của gcc/g++ và cài đặt

$ gcc --version

$ g++ --version

$ sudo apt-get install gcc g++

Shell

Ví dụ

Xét một ví dụ cơ bản với chương trình C tính căn bậc 2 của 4 như sau

#include <stdio.h>

#include <math.h>

int main(int argc, char \*\*argv){

double x;

x = sqrt(4);

printf("x = %f \n", x);

return 1;

}

C

Để thực hiện thì chúng ta lưu code trên dưới dạng file là main.c, sau đó thực hiện gõ lệnh command sau trên ubuntu

$ export CFLAGS="-I./include -DDEBUG -Wall -g"

$ export LDFLAGS+=" -L./lib -lm"

$ gcc -c main.c ${CFLAGS} #tạo file object từ source

$ gcc -o prog main.o ${LDFLAGS} #tạo file chương trình nhị phân từ file object

$ ./prog #chạy chương trình

Shell

Kết quả

A computer screen with white text

Description automatically generated

Giải thích

Trên đây là một format cơ bản nhất của GCC

**CFLAGS**  
C compiler flags đưa các options vào trong compiler để thực hiện quá trình compile source code thành object sẽ bao gồm các thông tin:

* Đường dẫn các header bắt đầu với -I, ví dụ -I./include
* Các define được bắt đầu với -D, ví dụ -DDEBUG để define DEBUG
* Các option đặc biệt khác của compiler như -g để bật chức năng debug gdb của gcc compiler, -wall để trace các cảnh báo (warning) trong quá trình c

$ export CFLAGS="-I./include -DDEBUG -Wall -g"

Shell

**LDFLAGS**

Linker flags dùng trong quá trình linking các thư viện, nó bao gồm các thông tin:

* Đường dẫn tới thư viện, được bắt đầu bằng -L, ví dụ -L./lib
* Các thư viện bắt đầu với -l là viết tắt của lib, ví dụ: -lm tương ứng với libm, thư viện math có sẵn trong hệ thống

$ export LDFLAGS+=" -L./lib -lm"

Shell

**gcc**: Complier cho C source và **g++** là complier cho C++ source

Tiếp một ý bổ sung của anh Minatu

1. Có một thông số hơi quan trọng. Đó là sysroot, tức là đường dẫn đến thư mục root (/).Khi tìm file header lúc compile thì gcc sẽ làm đường dẫn base để tìm các file .h.Nó thường có 1 tập các thư mục mặc định như /usr/local/include,.. để tìm kiếm các file stdio.h, string… khi đó, nó sẽ tìm trên đường dẫn đầy là sysroot/usr/local/include.Khi linking cũng thế, nếu đường dẫn là /usr/lib thì đường dẫn đầy đủ khi tìm sẽ là sysroot/usr/lib.  
   Nếu biên dịch native thì sysroot chính là /. Còn nếu sử dụng cross-compiler đặc thì đường dẫn này thường khác.Có thể hiển thị cái này bằng tham số –print-sysroot cho gcc. Bạn có thể thao tác kiểm tra nhanh bằng lệnh gcc -print-sysroot
2. Có thể thêm -v vào câu lệnh biên dịch để thấy chính xác các tham số của gcc. Ví dụ như trong ví dụ trên ta thêm -v ở lệnh demo$ gcc -c main.c ${CFLAGS} -v

### Tạm kết

Thế là xong được những bước cơ bản đầu tiên với Crosscompiler, tìm hiểu được m

## GIỚI THIỆU

### Đặt vấn đề

#### Những vấn đề khi biên dịch

* Một chương trình đơn giản Chỉ có một vài file
* Một chương trình “không đơn giản”
* Nhiều dòng lệnh
* Nhiều module
* Nhiều người tham gia viết

#### Vấn đề xảy ra

* Khó quản lý một file lớn (cả người và máy).
* Mỗi thay đổi cần thời gian biên dịch lâu.
* Nhiều người lập trình không thể thay đổi cùng một file đồng thời
* Chương trình được phân ra thành nhiều module

#### Giải pháp

Chia project ra thành nhiều file.

#### Mục tiêu

* Chia project ra thành các modul một cách đúng đắn.
* Thời gian biên dịch phải ngắn nếu có sự thay đổi
* Dễ dàng bảo trì cấu trúc project

### Makefile

#### Makefile là gì?

* Makefile là một file dạng **script** chứa các thông tin
* Cấu trúc project (file, sự phụ thuộc)
* Các lệnh để tạo file
* Lệnh **make** sẽ đọc nội dung Makefile, hiểu kiến trúc của project và thực thi các lệnh

#### Cấu trúc project

* Cấu trúc và sự phụ thuộc của project có thể được biểu diễn bằng một DAG (Directed Acyclic Graph)

**Ví dụ:**

* Chương trình chứa 3 file: **main.c**, **sum.c**, **sum.h**
* File sum.h được dùng bởi cả 2 file **main.c** và **sum.c**
* File thực thi là **sum**

### Một số ví dụ cơ bản

#### Ví dụ với makefile cơ bản.

Chương trình đơn giản in ra dòng Hello makefiles được viết thành 3 file.

|  |  |
| --- | --- |
| hellomake.c | |
| 1  2  3  4  5  6  7  8 | #include <hellomake.h>  int main() {  // call a function in another file  myPrintHelloMake();  return(0);  } |

|  |  |
| --- | --- |
| hellofunc.c | |
| 1  2  3  4  5  6  7 | #include <stdio.h>  #include <hellomake.h>  void myPrintHelloMake(void) {  printf("Hello makefiles!\n");  return;  } |

|  |  |
| --- | --- |
| hellomake.h | |
| 1  2  3  4 | /\*  example include file  \*/  void myPrintHelloMake(void); |

Để bắt đầu chúng ta cần có 3 file bỏ chung vào 1 thư mục là:

* hellomake.c tương ứng chương trình chính.
* hellofunc.c là file hàm in thông báo.
* hellomake.h là file header khai báo hàm in.

Thông thường chúng ta có thể compile code và xem kết quả một cách đơn giản bằng lệnh sau:

|  |  |
| --- | --- |
| 1  2 | $ gcc -o hellomake hellomake.c hellofunc.c -I.  $ ./hellomake |

Lệnh này sẽ thực hiện compile 2 file .c. -I. có nghĩa là include gcc sẽ thực hiện tìm kiếm trong thư mục hiện tại(.) để thêm file hellomake.h.

Nếu không có makefile thì mỗi compile chúng ta lại phải mở terminal lên và gõ lệnh gcc -o … vào, điều này trở nên quá rắc rối, đặc biệt là khi chúng ta add thêm nhiều file .c khác vào trong chương trình hoặc khi chúng ta sửa lại nội dung code trong các file .c

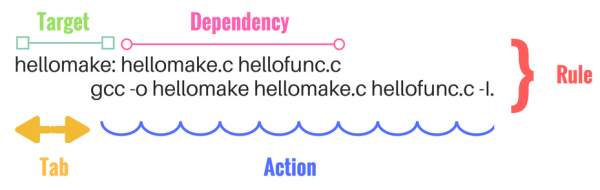
#### Vậy makefile sẽ khắc phục hạn chế ở trên như thế nào ?

##### Makefile 1

Trước tiên để dùng được makefile thì phải tạo file có tên là Makefile hoặc makefile trong thư mục chứa code hiện có với nội dung:

|  |  |
| --- | --- |
| 1  2  3  4  5 | CC=gcc  CFLAGS=-I.  hellomake: hellomake.c hellofunc.c  $(CC) -o hellomake hellomake.c hellofunc.c -I. |

Một cấu trúc make file cơ bản sẽ có dạng cơ bản như sau:



Trong đó:

* **Rule**: các rule cần thực hiện khi compile
* **Dependency**: là các file cần thiết để tạo ra target
* **Action**: là câu lệnh compile để tạo ra Target từ Dependency. Action được lùi vào 1 Tab so với Target
* **Target**: là file đích, nghĩa là file được hình thành sau khi quá trình make được thực hiện.

Vậy là trong thư mục chúng ta sẽ có 4 file: hellofunc.c hellomake.c hellomake.h và Makefile

Chạy chương trình bằng lệnh sau:

|  |  |
| --- | --- |
| 1  2 | $ make  $ ./hellomake |

##### Makefile 2

|  |  |
| --- | --- |
| 1  2  3  4  5 | CC=gcc  CFLAGS=-I.  hellomake: hellomake.o hellofunc.o  $(CC) -o hellomake hellomake.o hellofunc.o -I. |

Chúng ta tiếp tục phát triển thêm một chút bằng cách thêm CC và CFLAGS vào Makefile ở trên.

Trong đó:

* CC: là compiler C được sử dụng
* CFLAGS: là danh sách các flag của compiler.

Có một điểm khác nữa là thêm 2 file object là: hellomake.o và hellofunc.o trong dependency list và trong rule để make biết rằng đây là lần đầu tiên của quá trình biên dịch.

Với việc sử dụng makefile như trên thì đã có thể làm được các project nhỏ nhỏ rồi. Tuy nhiên vẫn còn thiếu dependency là các file include. Giả sử như ta có thay đổi trên file hellomake.h thì make sẽ không biên dịch lại file .c. Để khắc phục lỗi này thì ta cần phải thông báo cho make rằng tất cả các file .c đều bị phụ thuộc vào file .h, khi compile nhớ phải lưu ý nha.

##### Makefile 3

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | CC=gcc  CFLAGS=-I.  DEPS = hellomake.h  %.o: %.c $(DEPS)  $(CC) -c -o $@ $< $(CFLAGS)  hellomake: hellomake.o hellofunc.o  gcc -o hellomake hellomake.o hellofunc.o -I. |

Makefile này sẽ tạo ra một macro là DEPS để chỉ ra file .h mà các file .c phụ thuộc vào. Ngoài ra sẽ có một định nghĩa về rule áp dụng cho tất cả các file .o, rule này thông báo rằng các file .o phụ thuộc vào các file .c và .h được định nghĩa trong macro là DEPS (dòng 5). Rule sẽ tạo ra file .o, make được dùng C compile được định nghĩa trong CC để compile các file c.(dòng 6)

Có một số lưu ý:

* -c là tạo ra các object file
* $@ là tạo ra output của quá trình biên dịch trong tập tin bên trái dấu :
* $< là thành phần đàu tiên trong danh sách của dependency và CFLAGS là macro đã được định nghĩa ở dòng 2

Tiếp tục bước cuối cùng là sử dụng các macro đặc biệt như $@ và @^ để lấy thông tin bên trái và bên phải của dấu : để làm cho quá trình biên dịch được tổng quát hơn, cụ thể là các file include sẽ được đưa vào trong macro DEPS, tất cả các object file được đưa vào macro OBJ như makefile 4

##### Makefile 4

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | CC=gcc  CFLAGS=-I.  DEPS = hellomake.h  OBJ = hellomake.o hellofunc.o  %.o: %.c $(DEPS)  $(CC) -c -o $@ $< $(CFLAGS)  hellomake: $(OBJ)  gcc -o $@ $^ $(CFLAGS) |

Ta sẽ thấy một số file được sinh ra sau quá trình make như hình dưới:

Graphical user interface, application, Word

Description automatically generated