MAKEFILE

MỤC LỤC

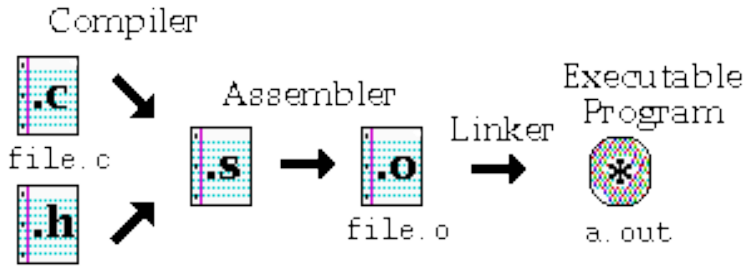
I. Mô tả

1. Câu lệnh Make

Câu lệnh **make** cho phép xử lý một chương trình lớn hay nhóm các chương trình. Khi bắt đầu viết một chương trình lớn, chúng ta sẽ chú ý đến việc compile lại chương trình sẽ tốn rất nhiều thời gian. Hơn nữa, chúng ta thường chỉ làm việc với một phần nhỏ của chương trình thôi(ví dụ một hàm đơn trong quá trình debug) và rất nhiều phần còn lại của chương trình vẫn giữ nguyên không đổi.

Chính vì vậy, chương trình **make** sẽ giúp chúng ta phát triển những chương trình lớn bằng cách bám sát những phần chương trình bị thay đổi thôi, và chỉ compile những phần này kể từ lần compile cuối cùng.

2. Một quá trình biên dịch đơn giản

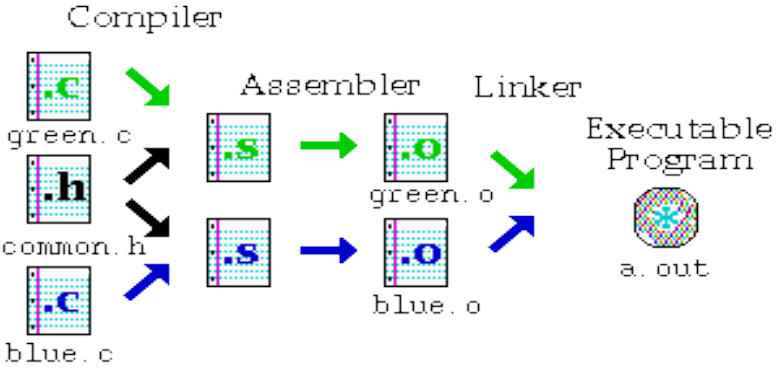


Biên dịch một chương trình C nhỏ đòi hỏi ít nhất 1 file .c và 1 file .h thích hợp. Chúng ta chỉ cần thực hiện bằng 1 lệnh đơn giản "*cc file.c*", nhưng có 3 bước cần thực hiện để xuất ra file thực thi:

a. Bước biên dịch(Compiler): Tất cả file code .c được chuyển đổi thành hợp ngữ (assembly) tạo ra file đuôi *.s*.
b. Bước dich hợp ngữ(Assembler): Các file code hợp ngữ sẽ được chuyển đổi thành file object code (*.o*) là những đoạn code mà máy tính có thể hiểu trực tiếp.
c. Bước liên kết(Linker): Bước cuối cùng khi compile chương trình là liên kết các file object code với file thư viện (chứa những hàm đã được định

nghĩa trước như *printf*). Sau bước này ta sẽ có được file thực thi *a.out* theo mặc định.

3. Biên dịch với nhiều files

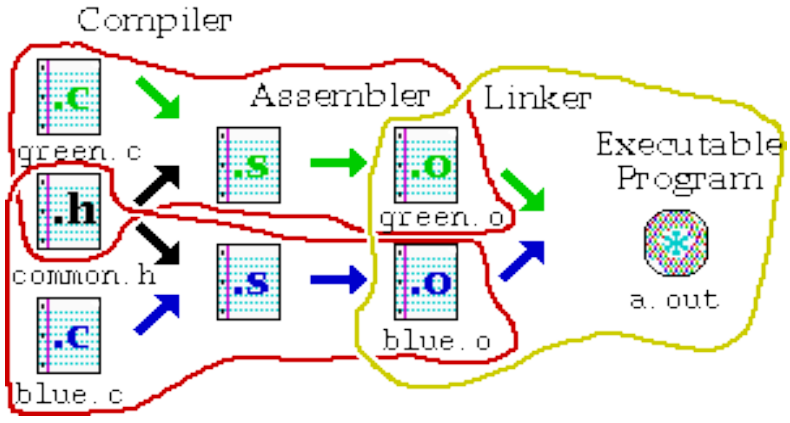


Khi chương trình trở nên lớn hơn, chúng ta cần phân chia source code ra thành nhiều file .c rời rạc để dễ quản lý. Hình vẽ trên mô tả quá trình biên dịch một chương trình cấu thành từ 2 file .c và 1 file .h chung. Câu lệnh sẽ là:

```
cc green.c blue.c
```

trong đó cả 2 file .c được đưa vào trình biên dịch Compiler. Quá trình Assembler thì giống ở trên nhưng bước cuối cùng thì có khác chút, đó là Linker sẽ liên kết 2 file .o để tạo thành 1 file thực thi duy nhất a.out .

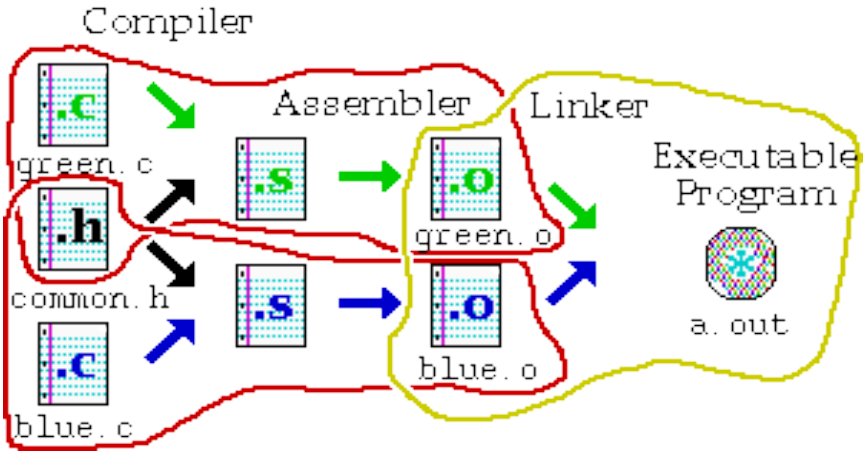
4. Biên dịch rời rạc



Như trên hình vẽ, ta thấy việc tạo thành file thực thi được chia làm 2 bước compiler/assembler riêng rẽ được khoanh đỏ, và một bước linker cuối cùng được khoanh vàng. 2 file *.o* được tạo ra rời rạc nhưng được kết hợp để tạo thành file thực thi.

Chúng ta có thể sử dụng tùy chọn **–c** với **cc** để tạo thành file object(*.o*) từ 1 file *.c*. Ví dụ: *cc –c green.c* không tạo thành 1 file thực thi đầu ra a.out, mà chỉ dừng ở bước Assembler tạo ra file green.o thôi.

5. Các bước biên dịch rời rạc



Có thể phân chia thành 3 bước sau:

- ✓ Biên dịch tạo green.o: cc –c green.c
- ✓ Biên dịch tạo blue.o: cc –c blue.c
- ✓ Liên kết các phần lại: cc green.o blue.o

Chúng ta nhận thấy rằng, để tạo thành file *green*.o thì đầu vào cần phải có 2 file *green.c* và *common.h*. Tương tự để tạo file *a.out* thì đầu vào cần 2 file *green.o* và *blue.o*.

6. Phân chia chương trình C

Khi phân chia chương trình C thành nhiều file, chúng ta cần chú ý các điểm sau:

- ✓ Không có 2 file nào sử dụng hàm với tên giống nhau cả.
- ✓ Chú ý khi sử dụng biến toàn cục(global) trong chương trình thì không có 2 file nào được định nghĩa cùng 1 biến cả.
- ✓ Đối với biến toàn cục, phải đảm bảo là chỉ có 1 file định nghĩa và khai báo biến đó trong file .h như sau:

extern int globalvar;

- ✓ Phải có ít nhất 1 file chứa hàm *main()*.

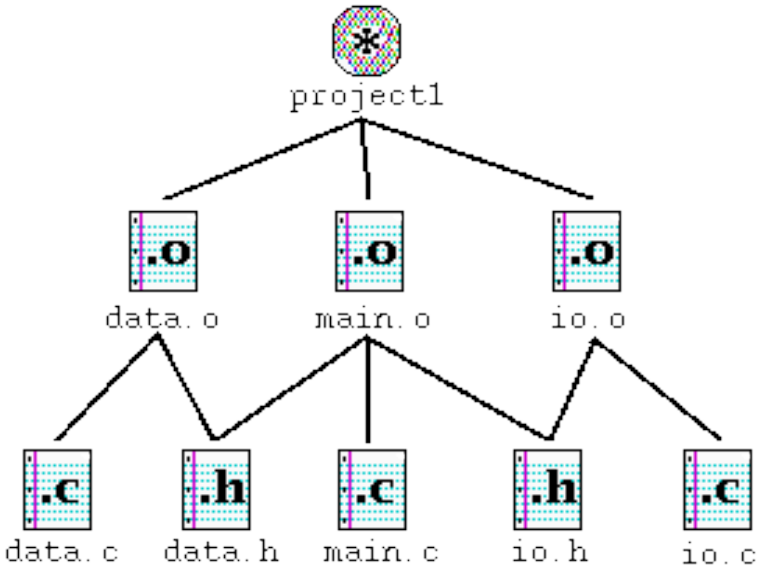
Chú ý: Khi chúng ta định nghĩa(define) 1 biến thì sẽ như sau: *int globalvar*. Còn khi khai báo(declare) 1 biến thì sẽ là *extern int globalvar*. Điểm khác biệt chính giữa định nghĩa và khai báo là định nghĩa là tạo ra biến còn khai báo ngụ ý rằng biến đó đã được khai báo ở đâu đó rồi!

II. Phụ thuộc(Dependencies)

Nguyên tắc mà **make** thực hiện là nó tạo chương trình thông qua các file phụ thuộc (dependencies). Ví dụ, để tạo ra file object *program.o* thì ít nhất phải có file *program.c*, ngoài ra có thể có các phụ thuộc khác như file .h.

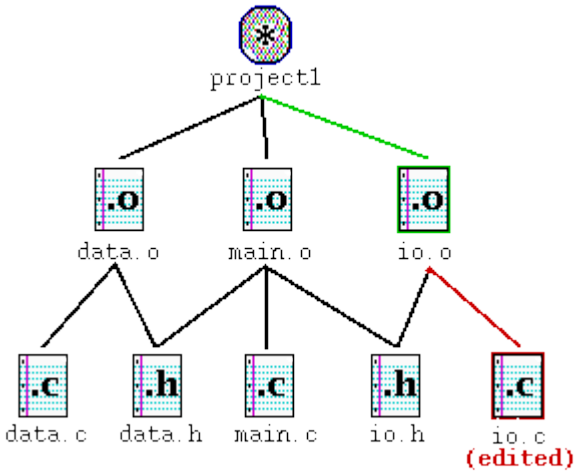
Phần này chúng ta sẽ nói đến sơ đồ phụ thuộc này để thuận tiện trong việc xây dựng **makefile**.

1. Sơ đồ phụ thuộc



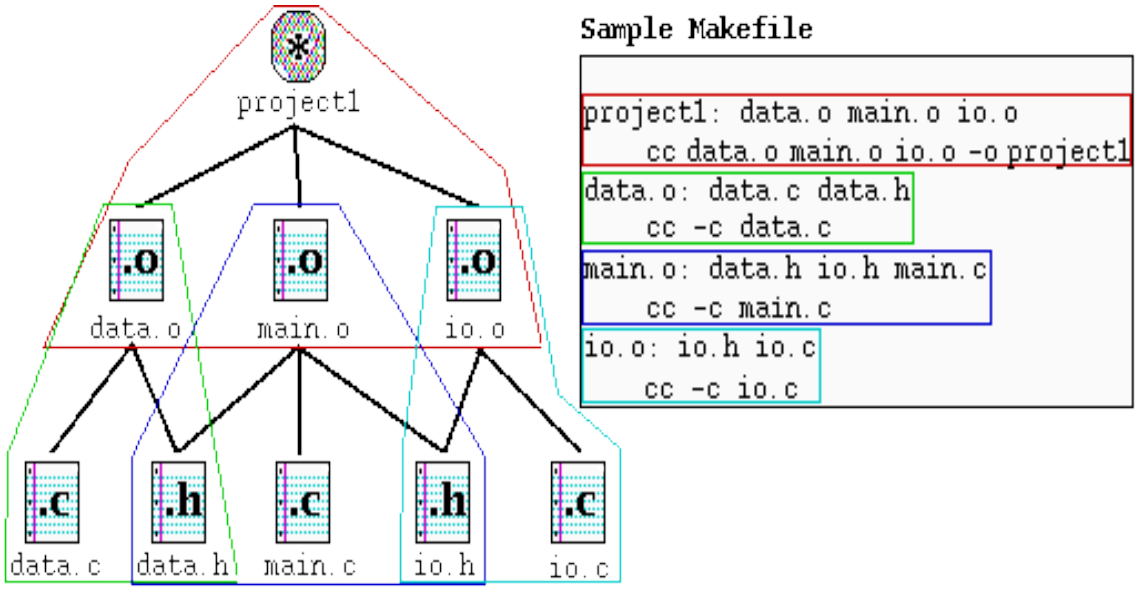
Sơ đồ trên là một chương trình tạo bởi 5 file source là *data.c, data.h, io.c, io.h, và main.c*. Đỉnh là file kết quả đầu ra project1. Những đường tỏa ra từ 1 file đến các file khác chính là những file mà nó phụ thuộc. Ví dụ như file *main.o* sẽ phụ thuộc 3 file *data.h, main.c, và io.h.*

2. Cách phụ thuộc hoạt động



Giả sử trong quá trình kiểm tra chương trình, chúng ta nhận thấy có 1 hàm trong file *io.c* có 1 lỗi. Chúng ta biên tập lại *io.c* để sửa lỗi và nhận thấy file *io.o* cần được cập nhật bởi vì file *io.c* mà nó phụ thuộc đã thay đổi. Tương tự, *io.o* đã thay đổi nên file *project1* cũng cần phải cập nhật.

3. Cách make thực hiện



Chương trình **make** tạo sơ đồ phụ thuộc dưới dạng text gọi là ***makefile*** hay ***Makefile*** được đặt cùng thư mục với các file mã nguồn(source code). **Make** sẽ kiểm tra quá trình sửa đổi của file; bất cứ khi nào 1 file được thay đổi, nó sẽ thực hiện quá trình biên dịch lại.

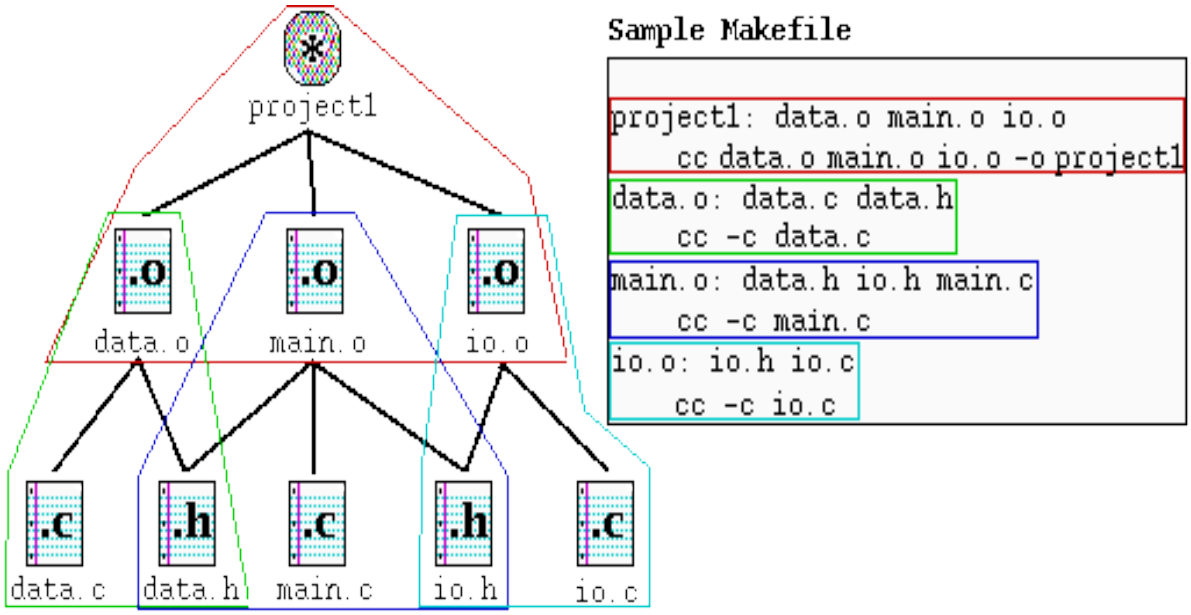
Ví dụ, khi file *io.c* bị thay đổi thì **make** phải chạy *cc –c io.c* để tạo file *io.o* mới hơn và sau đó chạy

cc data.o main.o io.o –o project1 để cập nhật file *project1*.

III. Makefile

Phần này sẽ mô tả chương trình **make** chi tiết hơn thông qua mô tả file mà nó sử dụng được gọi là **makefile** hay **Makefile**. File này quyết định mối quan hệ giữa file source, object và thực thi.

1. Chuyển đổi sơ đồ phụ thuộc



Mỗi sự phụ thuộc được thể hiện bằng hình vẽ khoanh tròn bởi màu tương ứng trong ***Makefile***, mỗi cái sẽ sử dụng định dạng sau(CÔNG THỨC):

Target : source file(s)
Command (phải được thụt vào bằng 1 Tab)

Một target(mục tiêu) trong ***Makefile*** là 1 file mà được tạo ra và cập nhật ngay khi bất kì source file nào bị thay đổi. Còn command(câu lệnh) trong những dòng còn lại được cách lùi đầu dòng bằng **1 Tab**, nó được thực thi để tạo file mục tiêu.

2. Liệt kê danh sách phụ thuộc

```
project1: data.o main.o io.o
        cc data.o main.o io.o -o project1
data.o: data.c data.h
        cc -c data.c
main.o: data.h io.h main.c
        cc -c main.c
io.o: io.h io.c
        cc -c io.c
```

Chú ý trong file ***Makefile*** ở trên thì có các file *.h* được liệt kê ra nhưng không có tham chiếu nào trong câu lệnh tương ứng cả. Đó là vì các file *.h* đã được tham chiếu đến file source *.c* thông qua *#include "file.h"* rồi. Nếu chúng ta không include các file này trong ***Makefile*** thì chương trình sẽ không được cập nhật nếu có thay đổi nào trong file header(*.h*).

3. Sử dụng Makefile với make

Makefile có thể xuất hiện dưới các tên sau đây: **Makefile** hay **makefile** đều được. Điều này cần được chú ý để khi sử dụng câu lệnh **make** thì có thể tìm đúng file để thực thi đúng. Ví dụ chúng ta chỉ cần đánh **make** trên terminal:

```
% make
        cc -c data.c
        cc -c main.c
        cc -c io.c
        cc data.o main.o io.o -o project1
%
```

IV. Shortcut cho make

Một trong những đặc điểm khá quan trọng của **make** là **macro.** Macros trong **make** hoạt động giống như macros trong lập trình C.

1. Macros trong make

Chương trình **make** cho phép bạn sử dụng macros, tương tự như biến để lưu trữ tên của file. Ví dụ như:

```
OBJECTS = data.o io.o main.o
```

Và khi muốn sử dụng biến này thì bạn cần gõ từ khóa `$(OBJECTS)`.

Đây là 1 chương trình có sử dụng macro:

```
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o: data.c data.h
        cc -c data.c
main.o: data.h io.h main.c
        cc -c main.c
io.o: io.h io.c
        cc -c io.c
```

Ngoài ra chúng ta có thể chỉ rõ giá trị macro khi chạy **make** như sau: **`make`**
`'OBJECTS=data.o newio.o main.o' project1`

Nó sẽ ghi đè giá trị của `OBJECTS` trong **Make**file.

2. Các luật(rule) được định nghĩa sẵn

Make sẽ tự hiểu để tạo ra file *.o* thì cần phải sử dụng câu lệnh cc –c đối với file *.c* tương ứng. Chính vì vậy mà chúng ta chỉ cần chỉ ra các file *.h* trong phần phụ thuộc(dependency) thì **make** sẽ tự hiểu là phải cần các file *.c* tương ứng. Ví dụ như:

```
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
```

```
data.o: data.h
main.o: data.h io.h
io.o: io.h
```

Tuy nhiên, khi chúng ta biên dịch chương trình trên **Wiliki** thì cần phải thêm macro `CFLAGS` ở đầu **Makefile** để sử dụng thư viện ANSI C chuẩn: `CFLAGS=-Aa -D_HPUX_SOURCE`.

3. Shortcut hỗn hợp

Nếu 1 file xuất hiện ở phần target nhiều hơn một lần thì tất cả source file phụ thuộc nó sẽ được biên dịch cùng lúc luôn.

```
CFLAGS = -Aa -D_HPUX_SOURCE
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o main.o: data.h
io.o main.o: io.h
```

Như ví dụ trên thì file *main.o* xuất hiện ở 2 vị trí nên khi compile thì **make** sẽ tìm tất cả phụ thuộc của nó bao gồm *data.h* và *io.h.*

V. Một vài tính năng cao cấp

1. Tùy biến đuôi file và các luật(rule)

Make cũng sử dụng một mục tiêu (target) đặc biệt như `.SUFFIXES` cho phép bạn định nghĩa các hậu tố của riêng bạn. Ví dụ như:

```
.SUFFIXES: .foo .bar
```

Câu trên có nghĩa là bạn có thể tạo ra file `.foo` từ file `.bar`.
Và định nghĩa luật(rule) của nó như sau:

```
.foo.bar:
        tr '[A-Z][a-z]' '[N-Z][A-M][n-z][a-m]' < $< > $@
.c.o:
        $(CC) $(CFLAGS) -c $<
```

Luật đầu tiên cho phép bạn tạo ra file `.bar` từ file `.foo`.

Còn luật thứ hai là luật mặc định cho phép **make** tạo ra file `.o` từ file `.c`.

2. Gọi Makefile khác để thực thi

Trong trường hợp chúng ta muốn sử dụng nhiều **Makefile** để thực thi công việc của mình thì phải sử dụng từ khóa @$(**MAKE**). Trong ví dụ sau đây,nếu chúng ta có 1 **makefile** tên là "*Makefile*" chỉ ra cách để **make** target 'foo'(và các target khác), chúng ta có thể viết 1 **makefile** khác tên là "*GNUmakefile*" chứa nội dung sau:

```
foo:
        frobnicate > foo

%: force
        @$(MAKE) -f Makefile $@
force: ;
```

Khi đó nếu chúng ta gọi '**make** foo' thì chương trình **make** sẽ tìm đến '*GNUmakefile'* và thấy rằng để **make** 'foo' thì nó phải thực thi câu lệnh

`frobnicate > foo` ,nhưng nếu gọi lệnh '**make** bar' thì **make** sẽ tìm đến câu lệnh '`make -f Makefile bar`'để thực thi, tức là nó sẽ gọi đến makefile '*Makefile*' để thực thi câu lệnh này.

VI. Một vài chú ý:

- **-I** : Để xác định thư mục include. Trong trường hợp, nếu source file không nằm trong thư mục hiện thời, mà cũng không chỉ rõ đường dẫn thì chương trình sẽ tự động tìm đến các thư mục được include trong tùy chọn '-I' hay --include-dir'; và các thư mục sau có thể được tìm đến: '/usr/local/include', '/usr/gnu/include', hay '/usr/include'.
- **-r** : Loại bỏ tất cả các luật(rule) ngầm định, tức là tùy chọn này sẽ xóa bỏ tất cả danh sách mặc định các hậu tố (.o, .c,…); tuy nhiên chúng ta có thể định nghĩa luật cho các hậu tố theo ý mình cho .SUFFIXES như đã trình bày ở trên.
- **--no-print-directory** : Vô hiệu hóa tính năng in cây thư mục đang làm việc khi bật option –w. Bởi vì trong option –w có 2 tùy chọn là –print-directory hay --no-print-directory.

- **?=** : Trong **makefile** có nghĩa là nếu chưa định nghĩa thì sẽ được định nghĩa. Ví dụ
  LINUX ?= /opt/brcm/linux
  Có nghĩa là nếu biến LINUX chưa được định nghĩa thì bây giờ sẽ được gán là /opt/brcm/linux.
- **-C dir** : Chuyển đến thư mục dir trước khi đọc **makefile**. Ví dụ

  make -C /opt/brcm/linux M=$(CWD) modules PLATFORM=$(NEXUS_PLATFORM)

Có nghĩa là trước khi đọc file **makefile** thì sẽ chuyển đến thư mục /opt/brcm/linux để build modules.

- **@echo**: Để in dòng thông tin ra màn hình console trong quá trình **make**. Ví dụ:
  @echo "[Install... nexus.ko]".

- **$@ :** Tên file target.
- **$% :** Tên của file target trong trường hợp target là một tập tin nén.
- **$< :** Tên của source file đầu tiên phụ thuộc.

Trên đây chỉ là một vài chú ý phổ biến khi viết **Makefile**, ngoài ra còn có nhiều chú ý, kí tự đại diện khác mà bạn có thể tìm thấy trong quyển **"GNU make.pdf".**

VII. Makefile để build kernel module

Chúng ta có một **make**file như sau để thực hiện build 1 module hello.ko:

```
obj-m += hellodriver.o
all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Có thể thấy rằng cách thức để viết **makefile** cho 1 module trên môi trường kernel (kernel space) hơi khác với cách viết **makefile** trên môi trường user (user space), tuy nhiên chúng cũng có những điểm chung và cùng một công thức cấu tạo bao gồm *target, sourcefile, và commands*.

Trong đó, với target là all câu lệnh obj-m += hellodriver.o chỉ ra rằng có 1 module được build từ file đối tượng (object) *hellodriver.o*. File module đầu ra sẽ được đặt tên là *hellodriver.ko* sau khi build file object trên.
Còn câu lệnh **make** -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules, đầu tiên thực hiện việc chuyển đến thư mục chứa mã nguồn của kernel được xác định sau tùy chọn –C là /lib/modules/$(shell uname -r)/build, với uname là tên của nhân linux được cài vào máy của chúng ta(có thể là linux-2.6;chúng ta có thể gõ lệnh **uname** vào shell để kiểm tra). Đồng thời nó sẽ tìm **makefile** của nhân ở trong thư mục đó. Còn tùy chọn **M=** buộc **makefile** chuyển về thư mục chứa mã nguồn module của chúng ta trước khi biên dịch module target. $(PWD) là câu lệnh trả về đường dẫn thư mục hiện tại.
Ngoài ra target clean cũng được viết tương tự all để xóa các file module vừa tạo.