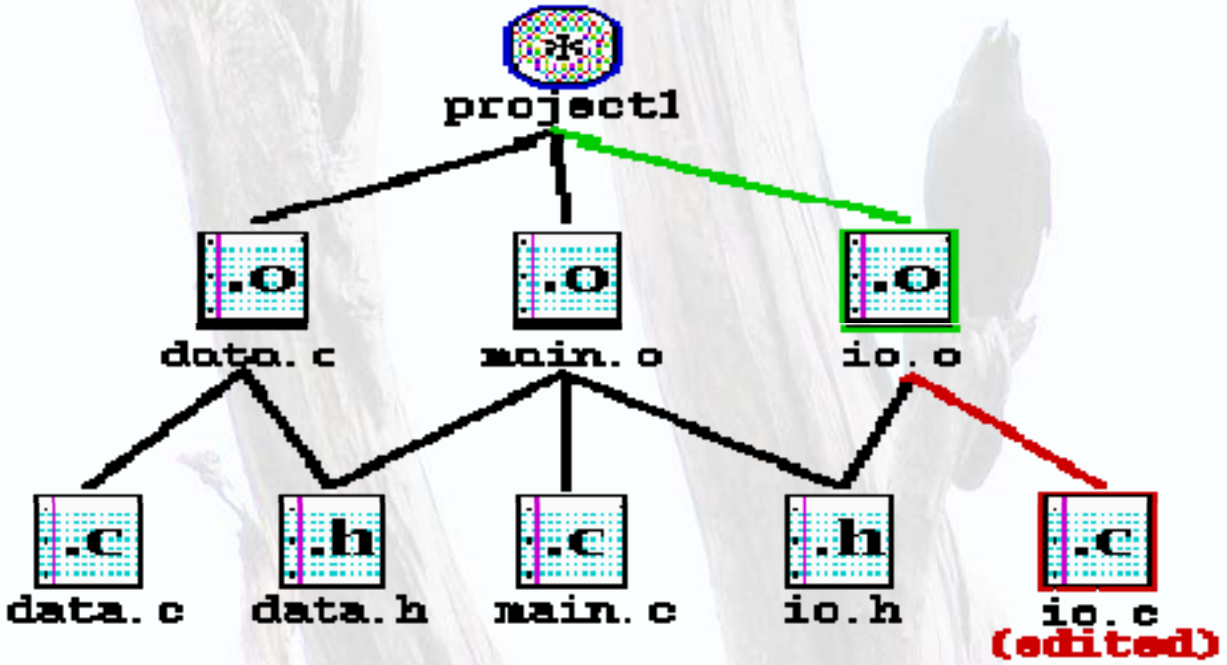
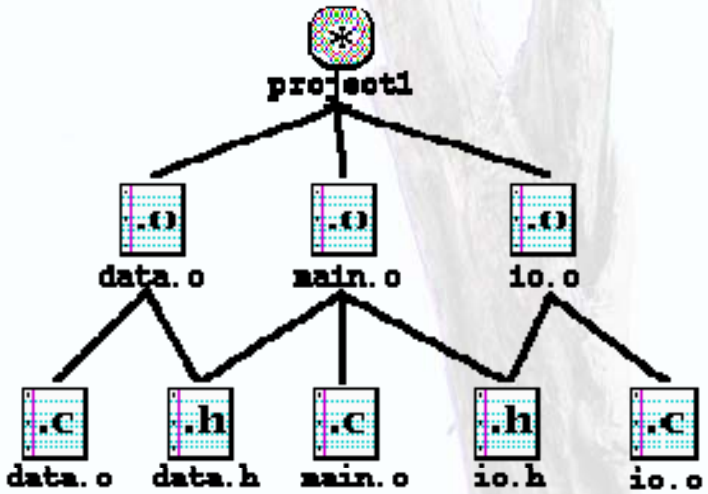
Makefile

Dependency graphs

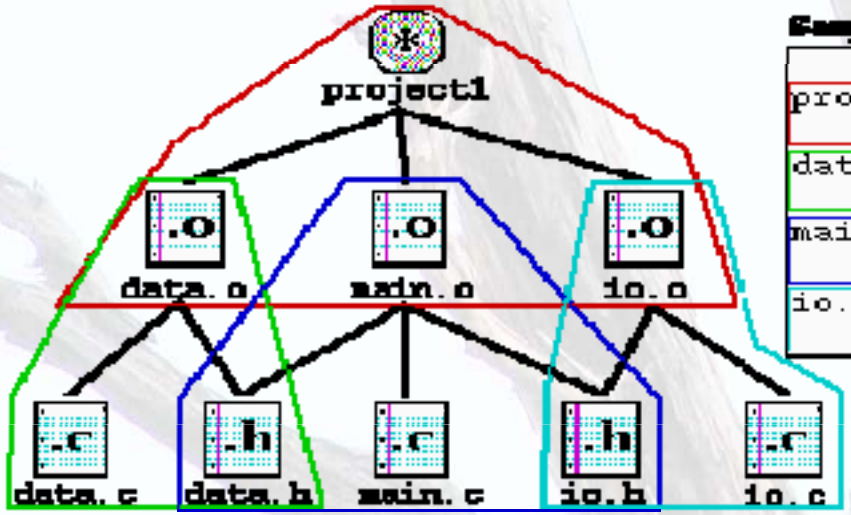


Dependency graphs



Sample Makefile

```
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
```



Sample Makefile

```
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
```

make程序

- Makefile規則的語法格式：

  目標文件列表：依賴文件列表

  <tab >命令列表

- 一個Makefile文件主要含有一系列的規則，每條規則包含以下內容。
  - “目標文件列表”，即make最終需要創建的文件，如可執行文件和目標文件；目標也可以是要執行的動作，如“clean”。
  - “依賴文件列表”，通常是編譯目標文件所需要的其他文件。
  - “命令列表”，是make執行的動作，通常是把指定的相關文件編譯成目標文件的編譯命令，每個命令占一行，且每個命令行的起始字符必須為TAB字符。
- 除非特別指定，否則make的工作目錄就是當前目錄。“目標文件列表”是需要創建的二進制文件或目標文件，依賴文件列表是在創建“目標文件列表”時需要用到的一個或多個文件的列表，命令序列是創建“目標文件列表”文件所需要執行的步驟，比如編譯命令。

make程序

■ 例如，有以下的Makefile文件：

```
# 一个简单的Makefile的例子
 test：prog.o code.o
        gcc  -o test prog.o code.o
 prog.o：prog.c prog.h code.h
        gcc  -c prog.c  -o prog.o
 code.o：code.c code.h
        gcc  -c code.c  -o code.o
 clean：
        rm  -f *.o
```

make程序

- 上面的Makefile文件中共定義了四個目標：test、prog.o、code.o和clean。目標從每行的最左邊開始寫，後面跟一個冒號（：），如果有與這個目標有依賴性的其他目標或文件，把它們列在冒號後面，並以空格隔開。然後另起一行開始寫實現這個目標的一組命令。
- 在Makefile中，可使用續行號（\）將一個單獨的命令行延續成幾行。但要注意在續行號（\）後面不能跟任何字符（包括空格和鍵）。
- 一般情況下，調用make命令可輸入：$make target
- target是Makefile文件中定義的目標之一，如果省略target，make就將生成Makefile文件中定義的第一個目標。對於上面Makefile的例子，單獨的一個 “make”命令等價於：

  $ make test
- 因為test是Makefile文件中定義的第一個目標，make首先將其讀入，然後從第一行開始執行，把第一個目標test作為它的最終目標，所有後面的目標的更新都會影響到test的更新。第一條規則說明只要文件test的時間戳比文件prog.o或code.o中的任何一個舊，下一行的編譯命令將會被執行。

make程序

- 在檢查文件prog.o和code.o的時間戳之前，make會在下面的行中尋找以prog.o和code.o為目標的規則，在第三行中找到了關於prog.o的規則，該文件的依賴文件是prog.c、prog.h和code.h。

- 同樣，make會在後面的規則行中繼續查找這些依賴文件的規則，如果找不到，則開始檢查這些依賴文件的時間戳，如果這些文件中任何一個的時間戳比prog.o的新，make將執行“gcc –c prog.c –o prog.o”命令，更新prog.o文件。

- 以同樣的方法，接下來對文件code.o做類似的檢查，依賴文件是code.c和code.h。當make執行完所有這些套嵌的規則後，make將處理最頂層的test規則。如果關於prog.o和code.o的兩個規則中的任何一個被執行，至少其中一個.o目標文件就會比test新，那麼就要執行test規則中的命令，因此make去執行gcc命令將prog.o和code.o連接成目標文件test。

make程序

- 在上面Makefile的例子中，還定義了一個目標clean，它是Makefile中常用的一種專用目標，即刪除所有的目標模塊。

- make做的工作：

  首先make按順序讀取makefile中的規則，

  然後檢查該規則中的依賴文件與目標文件的時間戳哪個更新，如果目標文件的時問戳比依賴文件還早，就按規則中定義的命令更新目標文件。如果該規則中的依賴文件又是其他規則中的目標文件，那麼依照規則鏈不斷執行這個過程，直到Makefile文件的結束，至少可以找到一個不是規則生成的最終依賴文件，獲得此文件的時間戳，然後從下到上依照規則鏈執行目標文件的時間戳比此文件時間戳舊的規則，直到最頂層的規則。

Makefile範例

```
#
# arch/arm/Makefile
#
LINKFLAGS           :=-p -X -T arch/arm/vmlinux.lds
OBJCOPYFLAGS        :=-O binary -R .note -R .comment -S
GZFLAGS             :=-9
CFLAGS              +=-Uarm -fno-common –pipe

CFLAGS_BOOT         :=$(apcs-y) $(arch-y) $(tune-y) -mshort-load-bytes -msoft-float
CFLAGS              +=$(apcs-y) $(arch-y) $(tune-y) -mshort-load-bytes -msoft-float
AFLAGS              +=$(apcs-y) $(arch-y) -mno-fpu -msoft-float

symlinks: include/asm-arm/.arch include/asm-arm/.proc

.PHONY: maketools
maketools: include/asm-arm/.arch include/asm-arm/.proc \
       include/asm-arm/constants.h include/linux/version.h checkbin
            @$(MAKETOOLS)
```

Makefile的語法規則

註解

```
#
# Make "config" the default target if there is no configuration file or
# "depend" the target if there is no top-level dependency information.
#
```

接續下行

```
DRIVERS-y += drivers/char/char.o \
             drivers/media/media.o
```

巨集

```
CFLAGS := $(CPPFLAGS) -Wall -Wstrict-prototypes -Wno-trigraphs -O2 \
              -fno-strict-aliasing -fno-common
```

法則

```
init/version.o: init/version.c include/linux/compile.h
             $(CC) $(CFLAGS) $(CFLAGS_KERNEL) -c init/version.c
```

自動變數 – $@ 與 $<

Demo.c

.c.o

```
$(CC) $(CFLAGS) –c –o $@ $<
```

$@

```
Demo
```

$<

```
Demo.c
```

Makefile常見巨集(1/2)

INCLUDE PATH

```
INCLUDE_PATH    = $(TOPDIR)/src/common
```

VPATH

```
VPATH           = $(TOPDIR)/src/lib
```

PREFIX

```
PREFIX          = /usr/local
```

CFLAGS

```
CFLAGS          = –g –I(INCLUDE_PATH)
```

AFLAGS

```
AFLAGS          = -mapcs-32 $(CFLAGS)
```

Makefile常見巨集(2/2)

LFLAGS

```
LFLAGS          = -Wl,-elf2flt,-M,-Map=$@.map
```

OBJS

```
OBJS            = demo.o sbrk.o driver.o
```

.C.O

```
$(CC) $(CFLAGS) –c –o $@ $<
```

.PHONY

```
.PHONY: clean
```

clean

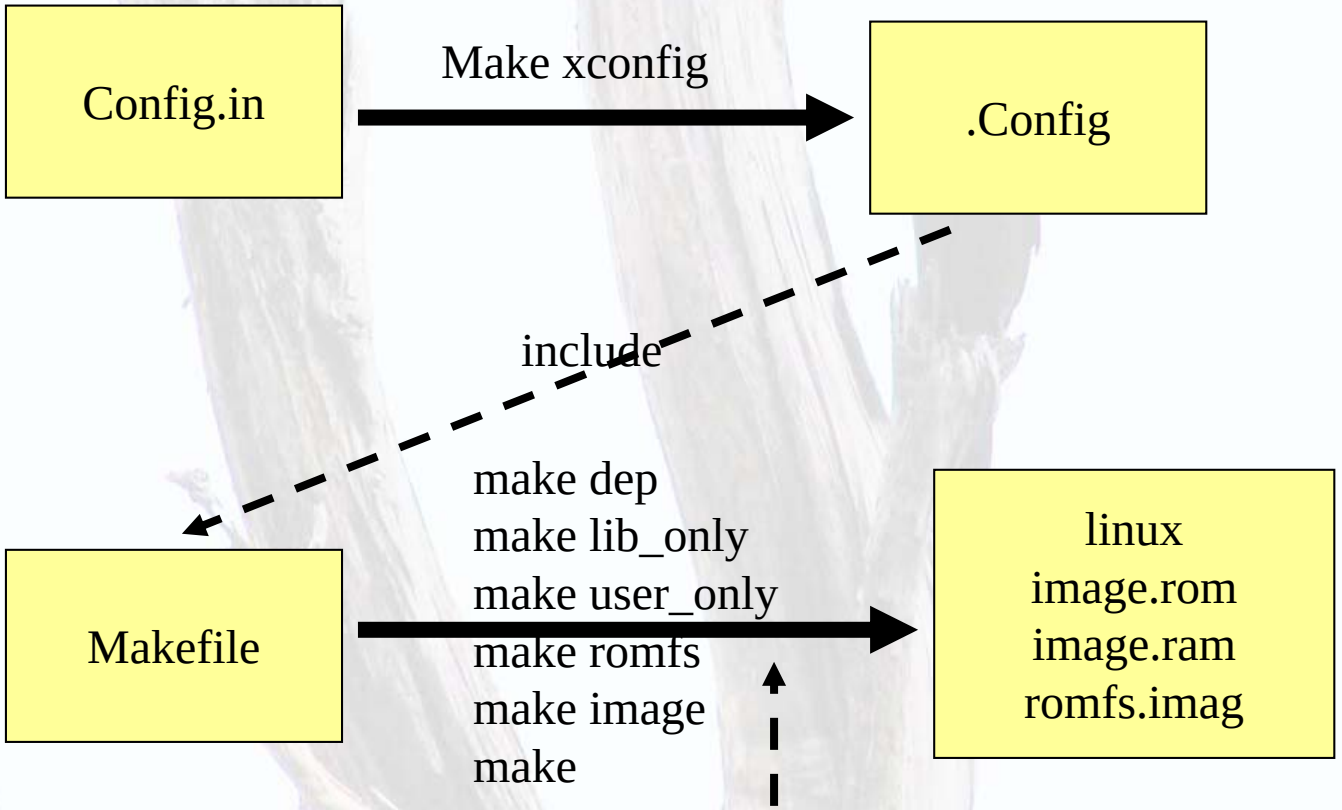
```
Rm *.bin *.axf *.o *.s
```

Make 內建參數

顯示內建的巨集參數

```
Make -p
```

Build Images



Example

```
CREATOR_PATH=/usr/var/creator/examples
INCLUDE_PATH1=$(CREATOR_PATH)/src/common
INCLUDE_PATH2=$(CREATOR_PATH)/src/gnu
VPATH = $(CREATOR_PATH)/src/common:$(CREATOR_PATH)/src/lib:$(CREATOR_PATH)/src/gnu

PREFIX=/usr/local
CC=$(PREFIX)/bin/arm-elf-gcc
LD=$(PREFIX)/bin/arm-elf-ld
CFLAGS= -nostartfiles -g -I$(INCLUDE_PATH1) -I$(INCLUDE_PATH2) -I. -O0
AFLAGS=-mapcs-32 -msoft-float -mno-fpu $(CFLAGS)
LFLAGS= -Wl,-elf2flt,-M,-Map=$@.map
OBJS=  demo.o sbrk.o driver.o

all : $(OBJS) head_ram.o head_rom.o
	$(CC)  -Tdemo.ld -Wl,-M,-Map=demo_ram.map -o "demo_ram.axf" $(OBJS) head_ram.o
	$(CREATOR_PATH)/lib/arm7_gnu_2953.a
	arm-elf-objcopy -O binary -S demo_ram.axf demo_ram.bin
	$(CC)  -Tdemo.ld -Wl,-M,-Map=demo_rom.map -o "demo_rom.axf" $(OBJS) head_rom.o
	$(CREATOR_PATH)/lib/arm7_gnu_2953.a
	arm-elf-objcopy -O binary -S demo_rom.axf demo_rom.bin

.c.o:
	$(CC) $(CFLAGS) -c -o $@ $<

head_ram.o : head.s
	$(CC) $(AFLAGS) head.s -c -o head_ram.o
head_rom.o : head.s
	$(CC) $(AFLAGS) -Wa,--defsym=LOADER=0 head.s -c -o head_rom.o

.PHONY : clean
clean:
	rm *.bin *.axf *.o *.s
```

- （1）**CREATOR_PATH=/usr/var/creator/examples**

  此列定義了我們相關檔案的路徑主目錄之置換名稱為CREATOR_PATH

- （2）**INCLUDE_PATH1=$(CREATOR_PATH)/src/common**
  INCLUDE_PATH2=$(CREATOR_PATH)/src/gnu

  同上，定義了兩組include路徑的置換名稱為INCLUDE_PATH1和INCLUDE_PATH2

  注意：這兩個定義均已使用了第一列所定義的CREATOR_PATH，其使用方式為$（置換名稱）

- （3）**VPATH = $(CREATOR_PATH) /src/common:**
  $(CREATOR_PATH) /src/lib: $(CREATOR_PATH)/src/gnu

  這其實也算是置換名稱，只不過VPATH已被內定為make尋找檔案時的路徑當makelife在make時預期所在路徑中找不到相關檔案時則會去VPATH中指定的路徑中尋找

- （4）**PREFIX=/usr/local**
  CC=$(PREFIX)/bin/arm-elf-gcc
  LD=$(PREFIX)/bin/arm-elf-ld

  定義compiler跟linker的路徑

- （5） **CFLAGS= -nostartfiles -g -I$(INCLUDE_PATH1)**
  I$(INCLUDE_PATH2) -I. -O0

  CFLAGS=

  設定給compiler使用的參數置換名稱

  -nostartfiles

  於link時不使用系統標準的startup檔

  -g

  產生OS原生格式的（stabs、COFF、XCOFF、DWARF）得除錯資訊，GDB需要這些資訊才能動作。大部分的系統都會使用stabs格式，-g會致能使用只有GDB能用的一些額外的除錯資訊，這些額外的資訊可讓GDB的除錯功能更強，但卻可能讓其他的debugger當掉或甚至無法讀取程式，你可用-gstab+，-gstab，-gxcoffe+， -gxcoffe…等參數來控制是否產生特定格式除錯資訊

-I

路徑，此參數用以指定header file的路徑

-O

最佳化的控制，-O0代表不最佳化，此為預設值

- （6）**AFLAGS=-mapcs-32 -msoft-float -mno-fpu $(CFLAGS)**

AFLAGS = -map

基本上，GCC Compiler出來的程式可支援相當多的平台，ARM只是其中一個，所以在參數設定中屬於ARM專用部份我們也刻意獨立起來置換為變數AFLAGS，以方便爾後平台的移植。

-mapcs-32

產生給有32bit程式計數器之處理器的程式碼

-msoft-float

產生含有浮點函式庫呼叫的輸出

-mno-fpu

同上，適用於SPARC

- （7）**LFLAGS= -Wl,-elf2flt,-M,-Map=$@.map**

  設定給linker使用的參數置換名稱

  -Wl

  選項參數將這些選項傳到linker，而且各選項間，以逗點隔開

- （8）**OBJS= demo.o sbrk.o driver.o**

  OBJS=

  定義主要的prerequisites（物件檔案）的置換名稱

- （9）

```
all : $(OBJS) head_ram.o head_rom.o
    $(CC)  -Tdemo.ld -Wl,-M,-Map=demo_ram.map -o "demo_ram.axf" $(OBJS) head_ram.o
$(CREATOR_PATH)/lib/arm7_gnu_2953.a
    arm-elf-objcopy -O binary -S demo_ram.axf demo_ram.bin
    $(CC)  -Tdemo.ld -Wl,-M,-Map=demo_rom.map -o "demo_rom.axf" $(OBJS)
    head_rom.o
$(CREATOR_PATH)/lib/arm7_gnu_2953.a
    arm-elf-objcopy -O binary -S demo_rom.axf demo_rom.bin
```

這隻demo程式除了定義在OBJS變數內的幾個檔案外，還必須包含我們所選擇的target的相關程式（可以稱之為驅動程式，一般均以Assembly撰寫），因為這幾個檔和target相關，因此我們將之獨立開來。

在前面我們曾經提過，一個makefile的第一個targer稱為goal，在這範例裡，也就是”all”了，而要達到這個goal，需要幾個prerequisites，也就是說demo，sbrk.o，driver.o，head-ram.o和

head-rom.o，這幾個物件檔，如規則所言，下一列會緊接command，而$（CC）在先前定義過

，為GNU的Compiler，而下一個檔案名稱用以指定linker description file（連結器描述檔）的檔名（再和makefile同一目錄下）而-Wl,-M,-Map=xxx.map則用以產生map檔

在本範例我們將以head-ram.o和head-rom.o（在接下來的說明中會向你交代此兩個檔案的產生方式）分別和共同的OBJS（demo.o，sbrk.o，driver.o）和函式庫arm7-gnu-2953.a 組合以”0”來產生demo-ram.axf和demo-rom.axf ，而另一個命令arm-elf-objcopy則在將.axf檔轉換成二進制的.bin檔

- （10）**.c.o:**

  $(CC) $(CFLAGS) -c -o $@ $<

  此為舊形式的suffix rules，用以定義make中內定的rule，此種rule因pattern rule較通用且清楚以漸漸被淘汰，而為了和舊的makelife相容故仍有支援，對make而言，.c.o為一-double-suffix rule.C表source suffix（也就是說副檔名為C的為source檔），-c表target suffix（也就是說target檔的副檔名為.o，-c為compiler，-o為輸出指定檔名），至於$@ $<為自動變數分別代表rule的target檔名和第一個prerequisite的名稱

變數 (Variables) 與隱性規則 (Implicit Rules)

- Make 也提供了一些預設的變數，或稱自動變數 (Automatic Variables)。同樣的，不同版本的 Make 工具，提供的預設變數也不盡相同，不見得互通。下列清單列出普遍的通用預設變數。

- 1. $*
- 目標檔之主檔名
- 2. $@
- 目標檔之全檔名
- 3. $<
- 成員檔中，比目標檔還新的檔案

- （11）

```
head_ram.o : head.s
        $(CC) $(AFLAGS) head.s -c -o head_ram.o
head_rom.o : head.s
        $(CC) $(AFLAGS) -Wa,--defsym=LOADER=0 head.s -c -o head_rom.o
```

此範例中我們用了點小技巧，將head.s分別組譯成head_ram.o和head_rom.o，其中-Wa意味將後面以逗點分開的參數送至Assembler，而依定義assembler在組譯輸入的檔案前先行將LOADER的值定義為0，而LOADER為0表其載入起始位址為0，以利未來我們所輸出的檔能燒錄至ROM中直接執行

- （12）

```
.PHONY : clean
clean:
        rm *.bin *.axf *.o *.s
```

定義clean這個target可於make執行特別指定執行（make clean）而rm為清除檔案的command

Lab.

- 何謂MakeFile及make處理MakeFile的程序為何?
- 請嘗試建立一MakeFile檔,假設一程式包含menu.c,menu.h,test.c,test.h,其中menu.c,menu.h,放在/usr/src/menu下,test.c,test.h放在/usr/src/menu/test下,程式完成後執行檔名為menu且放置於/usr/bin下,是完成上述條件之MakeFile
- 試判讀下頁MakeFile檔案.

```
.PHONY : all

PREFIX=/usr/local
CC=$(PREFIX)/bin/arm-elf-gcc
LD=$(PREFIX)/bin/arm-elf-ld
CFLAGS=-fno-builtin -nostdlib -g -I. -O0
AFLAGS=-mapcs-32 -msoft-float -mno-fpu $(CFLAGS)
LFLAGS=$(CFLAGS) -Wl,-elf2flt
OBJS= @OBJS@

.c.o:
	$(CC) $(CFLAGS) -c -o $@@ $<

.S.o:
	$(CC) $(AFLAGS) -c -o $@@ $<

all : $(OBJS)
	$(LD) -p -X -T Target1.ld -o "@TARGET_FILE@" $(OBJS)

clean:
	rm "@TARGET_FILE@" *.o
```