Keil 使用命令列附加預定義宏編譯

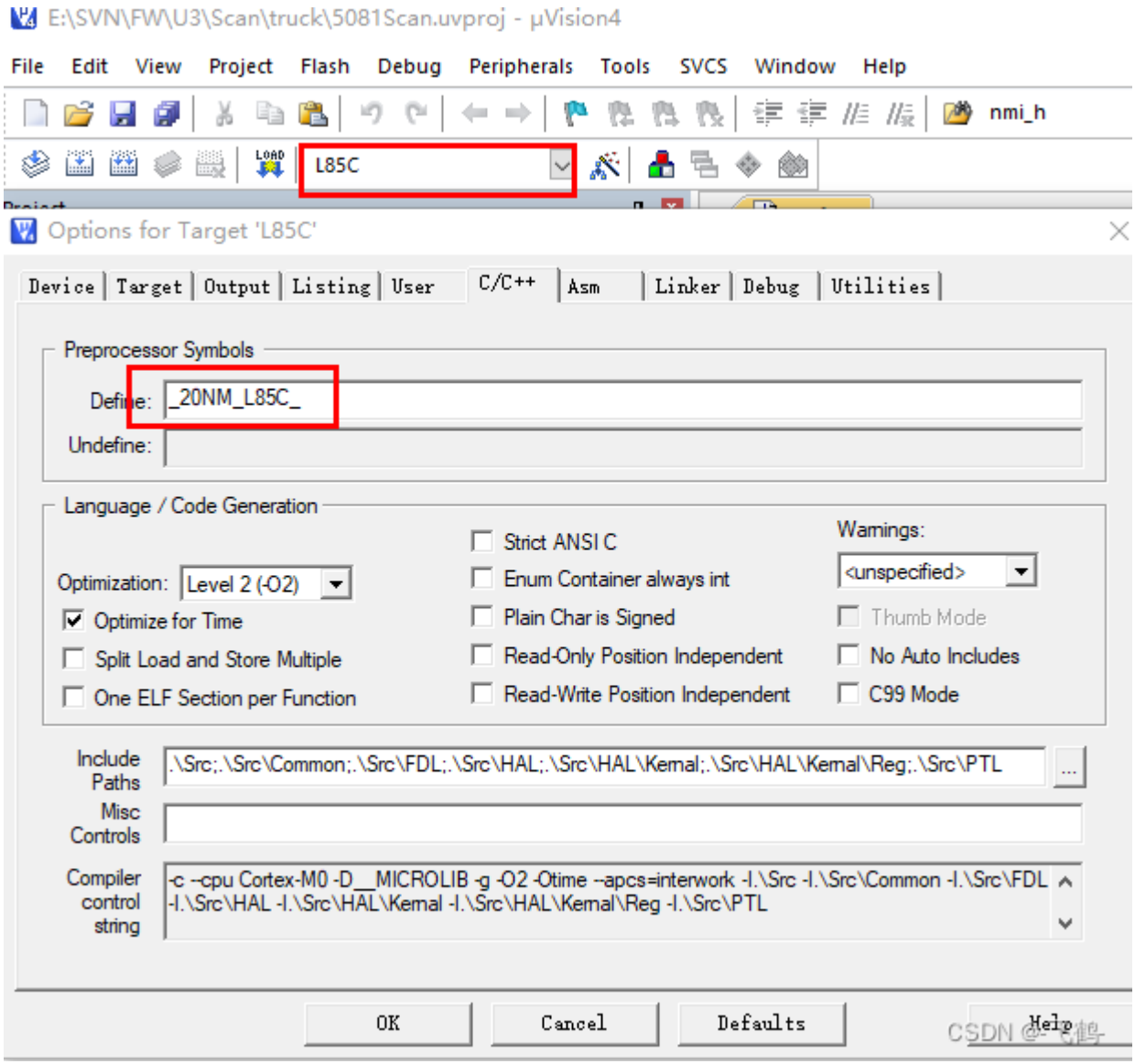
已於 2022-10-15 01:29:23 修改

分類專欄： 嵌入式 文章標籤： Keil makefile

嵌入式

1. 前言

很多時候，一份 Keil 工程程式碼可能需要滿足多個不同的應用場景。可以通過邏輯判斷，將多個不同的點整合在一份程式碼之中，但是嵌入式往往特別關注 RAM 空間，整合過多的邏輯判斷，RAM 空間可能就不夠用了。針對這種情況 Keil 提供了組態 Target，通過宏定義來分隔不同的功能，編譯生成不同的 bin 檔案。



但是有時，一個 Target 項目中，另外又有幾個子場景，子場景也需要用預定義宏來區分開。但是 Keil 並沒有提供這樣的功能。

另外，Keil 工程有非常多的 Target，一次修改，必須不停選擇不同 Target 來進行編譯，這種編譯方式非常不方便。

2. 方法

Keil 提供了命令列，可以編譯工程。另外，Keil 使用的 ArmCC 和 Armlink 來編譯和連結，這樣的話，便可以使用 makefile 來編譯 Keil 工程程式碼。

2.1. 命令列

2.1.1 用法

UV4 [command] [projectfile]

2.1.1.1. command

command 可以是下述列表之一。如果 command 沒有指定，則只是打開 Keil 工程。

- -b, 編譯指定的 Keil 工程默認的 Target。

  UV4 -b PROJECT1.uvprojx

- -c, 清除 Keil 工程中所有 Target 的臨時檔案。

  UV4 -c PROJECT1.uvprojx

- -cr, 清除所有 Target，然後重新編譯所有 Target。

  UV4 -cr PROJECT1.uvprojx

- -d, 啟動 Keil 的偵錯模式。

  UV4 -d PROJECT1.uvprojx

- -f, 下載程序到 flash 上，下載完成之後退出。

  UV4 -f PROJECT1.uvprojx -t "MCB2100 Board"

- -r, 重新編譯 Keil 工程的默認 Target，或用 -t 指定 Target。

  UV4 -r PROJECT1.uvprojx

UV4 -r PROJECT1.uvprojx -t "Simulator"

- -5, 轉換 Keil4 工程為 Keil5 工程。

  UV4 -5 myoldproject.uvproj -l log.txt

- -et, 匯出工程的 Target 的組態。
- -ep, 匯出工程所有 Target 組態。
- -X, 生成預處理符號檔案。
- -X1, 為所有 Target 生成預處理符號檔案。

2.1.3. Option

下面的選項是可選項。

- -j0, 隱藏 Keil 的 UI。
- -i, 建立一個新的工程，或通過 XML 檔案更新存在的工程。
- -l logfile，保存命令生成的輸出到 logfile 中。
- -n device_name, 建立一個指定 device_name 的工程。

  UV4 MyProject.uvprojx -n Device123

- -np device_name, 如果工程不存在，則指定 device_name 建立工程。如果工程存在，則更新所有 target 的 device_name。
- -o outputfile, 指定輸出的 Log 檔案。

  UV4 -r PROJECT1.uvprojx -o"output.log"

- -q, 重新編譯多工程指定的 Target。
- -t, 指定工程的 target。

  UV4 -r PROJECT1.uvprojx -t "MCB2100 Board"
- -x, 配合 Debug 模式命令-d 使用,返回完整的命令輸出。
- -y, 配合 Debug 模式命令-d 使用,返回通用的組態。
- -z, 重新編譯工程的所有 Target。

  UV4 -b PROJECT1.uvproj -z

2.1.4. ERRORLEVEL

Kiel 編譯完成後的錯誤碼

ERRORLEVEL	描述
0	無錯誤無敬告
1	僅有敬告
2	錯誤
3	重大錯誤
11	不能打開工程
12	給定的裝置不存在
13	寫工程檔案出錯

15	讀 XML 檔案出錯
20	轉換工程出錯

2.1.5. 指定多預定義宏

為一個 target 指定多個預定義宏，此處使用 shell 指令碼編寫，需要 git-bash 或 cygwin 來編譯。

新增一個 define.h 宏，Keil 工程引用此宏達到控制一個 Target 中有不同的預編譯宏。

```
#!/bin/bash

# Obtain Keil project
prj_name=$(find *.uvproj)

# Obtain all target name of uvproj
mapfile -t target_array < <(cat "${prj_name}" | grep "TargetName" | awk
-F '>' '{print $2}' | awk -F '<' '{print $1}')

# Macro
L85C_define=(__L85C1_ __L85C2__)

# Build all target
for name in "${target_array[@]}"; do
    if [[ "$name" == "L85C" ]]; then
        for define in "${L85C_define[@]}"; do
            echo "#define" $define>define.h

            # Code 中 include "define.h"

            "C:\Keil\UV4\UV4.exe" -r 5081Scan.uvproj -t"$name" -o
"$name""$define""Build.txt" -j0
        done
    else
        "C:\Keil\UV4\UV4.exe" -r 5081Scan.uvproj -t"$name" -o
"$name""Build.txt" -j0
```

```
    fi

    # If build error, stop build.
    if (( $? > 0 )); then
        echo Build error.
        break
    fi
done
```

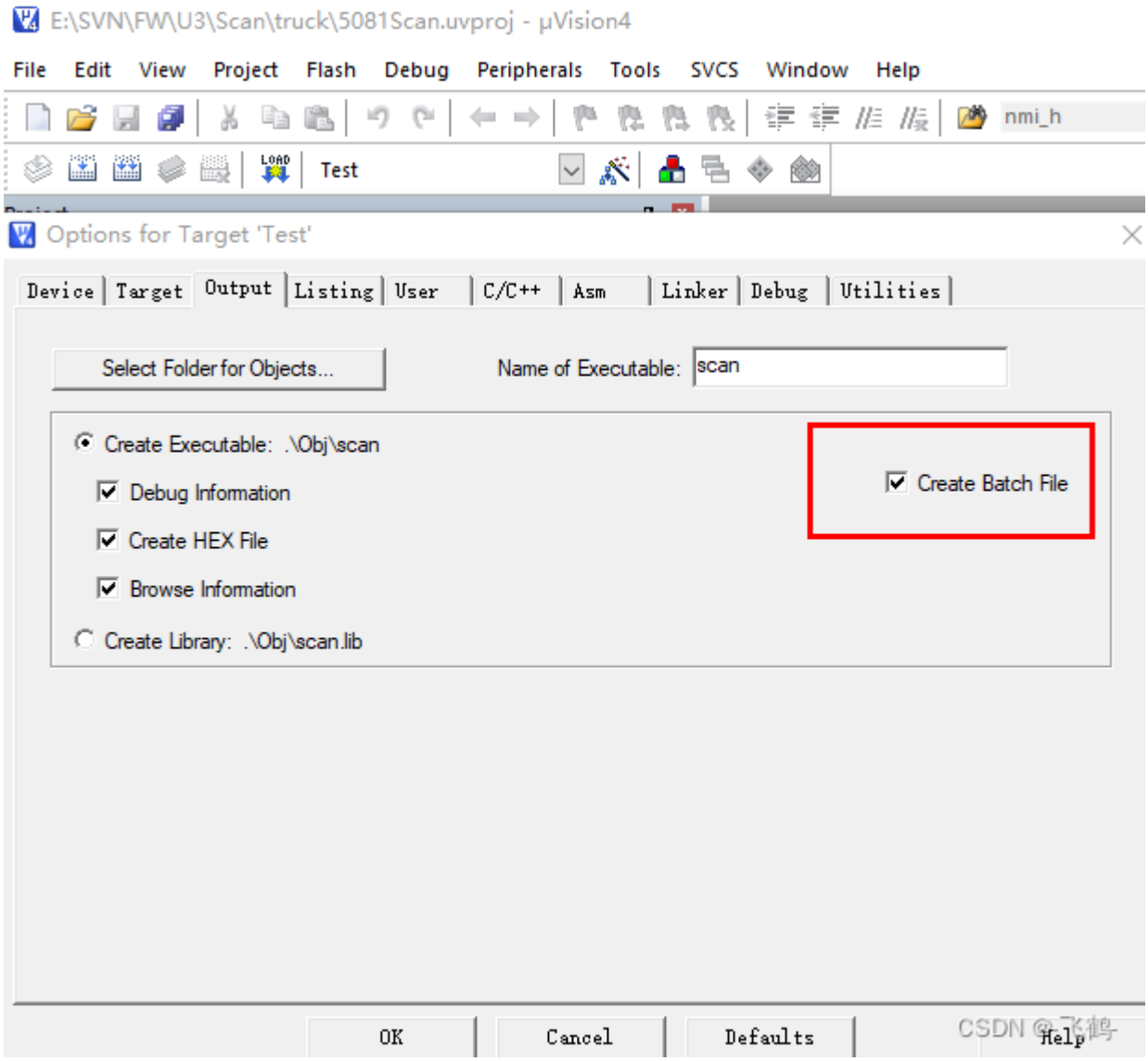
2.2. makefile

2.2.1. Keil 編譯過程

Keil 使用 ArmCC 編譯原始檔，使用 Armlink 連結目錄檔案生成 efl 檔案，然後呼叫 fromelf 轉換為 bin 檔案。

那麼 ArmCC 和 Armlink 如何使用呢？Keil 提供一種方法來展示如何使用 ArmCC 和 Armlink。

如下圖，勾選 Create Batch File，會將 ArmCC 和 Armlink 編譯連結的過程輸出到相應的檔案中。

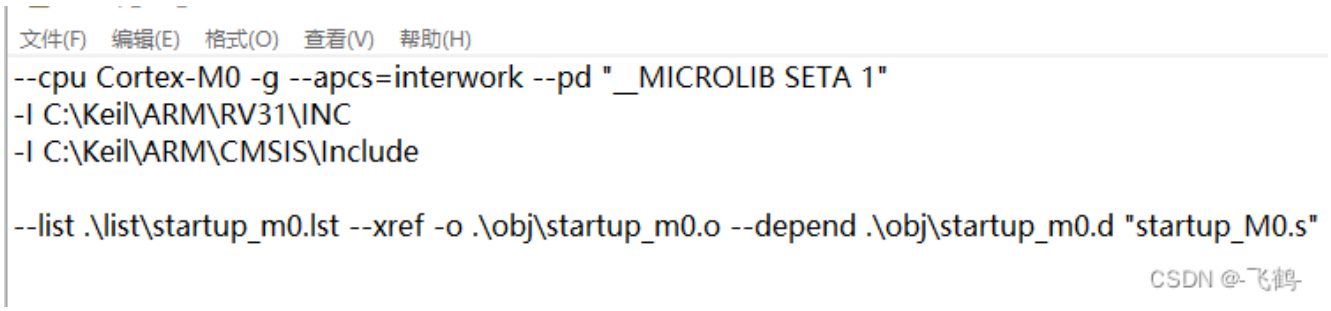


編譯當前工程指定的 Test，編譯連接的命令過程生成在 test.bat 檔案中，如下圖：

```
SET PATH=C:\Keil\ARM\ARMCC\bin\;C:\Users\feihe\AppData\Local\Programs\Python\Python
\Python310\Scripts\;C:\Python310\;
SET CPU_TYPE=Cortex-M0
SET CPU_VENDOR=ARM
SET UV2_TARGET=Test
SET CPU_CLOCK=0x00B71B00
"C:\Keil\ARM\ARMCC\bin\ArmAsm" --Via ".\obj\startup_m0._ia"
"C:\Keil\ARM\ARMCC\bin\ArmCC" --Via "list\main._ip"
"C:\Keil\ARM\ARMCC\bin\ArmCC" --Via ".\obj\main.__i"
"C:\Keil\ARM\ARMCC\bin\ArmLink" --Via ".\Obj\scan.lnp"
"C:\Keil\ARM\ARMCC\bin\fromelf.exe" ".\Obj\scan.axf"  --bin --output ".\Obj\scan.bin"
```

CSDN @-飞鹤

startup_mo_ia 檔案主要是編譯彙編檔案，內容為：

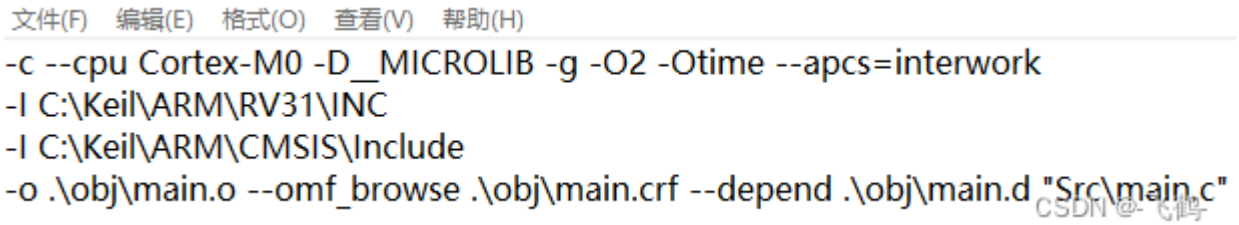


實際的呼叫命令為：

```
ArmAsm --cpu Cortex-M0 -g --apcs=interwork --pd “__MICROLIB SETA 1” -I
C:\Keil\ARM\RV31\INC
-I C:\Keil\ARM\CMSIS\Include --list .\list\startup_m0.lst --xref -
o .\obj\startup_m0.o --depend .\obj\startup_m0.d “startup_M0.s”
```

- main._ip，這個主要是生成彙編檔案，一般情況可以不使用。

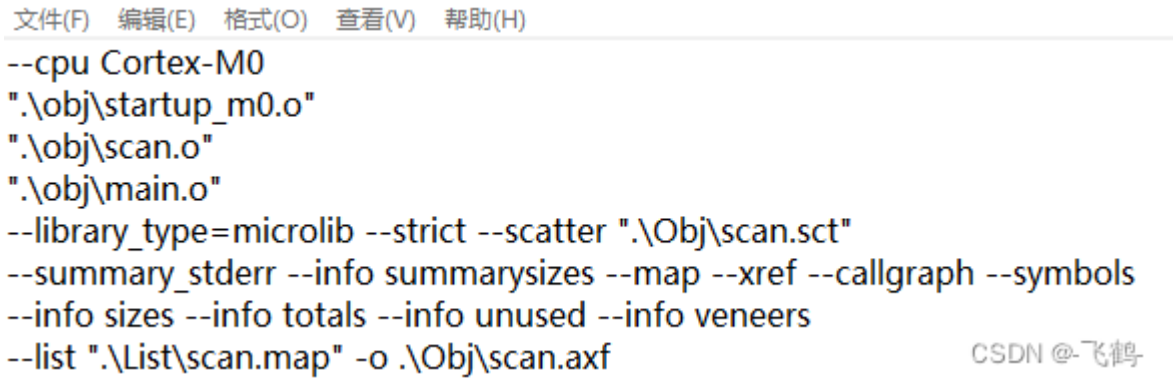
- main._i 檔案主要是編譯.c 檔案，內容為：



實際呼叫的過程為：

```
ArmCC -c --cpu Cortex-M0 -D__MICROLIB -g -O2 -Otime --apcs=interwork -I
C:\Keil\ARM\RV31\INC -I C:\Keil\ARM\CMSIS\Include -o .\obj\main.o --
omf_browse .\obj\main.crf --depend .\obj\main.d “Src\main.c”
```

- scan.lnp，主要是描述連接的過程。



實際的連接過程為：

```
ArmLink --cpu Cortex-M0".\obj\startup_m0.o" “.\obj\scan.o” “.\obj\main.o”
--library_type=microlib --strict --scatter “.\Obj\scan.sct” --
summary_stderr --info summarysizes --map --xref --callgraph --symbols --
info sizes --info totals --info unused --info veneers --list
“.\List\scan.map” -o .\Obj\scan.axf
```

2.2.2. makefile 內容

```
APP ?= .\output\scan.afx
MACRO ?=invlid_macro

OUTPUTCHAN ?= SEMIHOSTED

SHELL=$(windir)\system32\cmd.exe
RM_FILES = $(foreach file,$(1),if exist $(file) del /q $(file))
RM_DIRS = $(foreach dir,$(1),if exist $(dir) rmdir /s /q $(dir)$(EOL))

ifeq ($(QUIET),@)
PROGRESS = @echo Compiling $<...
endif

SRC_DIR = Src
ASM_DIR = .
INC_DIR = include
OBJ_DIR = obj
LINK = armlink
OUTPUT_DIR=output
```

```
INCLUDES := -I$(INC_DIR)
INCLUDES += -I C:\Keil\ARM\RV31\INC
INCLUDES += -I C:\Keil\ARM\CMSIS\Include

ARCH              := Cortex-M0
CPU               := ARM
LINK_GCC           := link.ld
GCC_LINKER_FILE       := -T link/$(LINK_GCC)

GCC_TOOLCHAIN   := C:\Keil\ARM\ARMCC\bin
ASM         := $(GCC_TOOLCHAIN)\ArmAsm
CC        := $(GCC_TOOLCHAIN)\ArmCC
LINKER_CSRC := $(GCC_TOOLCHAIN)\armlink
FROMELF := $(GCC_TOOLCHAIN)\fromelf

# GCC options
ASM_OPTS          := --cpu Cortex-M0 -g --apcs=interwork --pd "__MICROLIB
SETA 1"
CC_OPTS           := --cpu Cortex-M0 -D__MICROLIB -g -O2 -Otime --
apcs=interwork -D$(MACRO)
LINKER_FILE          := --cpu Cortex-M0 --library_type=microlib --strict -
-scatter ".\scan.sct"  --info summarysizes


APP_C_SRC := $(wildcard $(SRC_DIR)/*.c)
APP_S_SRC := $(wildcard $(ASM_DIR)/*.s)
OBJ_FILES := $(APP_C_SRC:$(SRC_DIR)/%.c=$(OBJ_DIR)/%.o)
OBJ_FILES += $(APP_S_SRC:$(ASM_DIR)/%.s=$(OBJ_DIR)/%.o)
DIRS= $(OUTPUT_DIR) $(OBJ_DIR)

.phony: all clean

all: FORCE $(APP)

FORCE:
    @echo being

$(APP): $(DIRS) $(OBJ_FILES)
```

```
    @echo Linking $@
    $(LINKER_CSRC) $(LINKER_FILE) $(OBJ_FILES) -o $@
    $(FROMELF) $@ --i32combined --output ".\output\scan.hex"
    $(FROMELF) $@  --bin --output ".\output\scan.bin"
    @echo Done.

clean:
    $(call RM_DIRS,$(OBJ_DIR))
    $(call RM_DIRS,$(OUTPUT_DIR))
    $(call RM_FILES,$(APP))


$(DIRS):
    mkdir $@
    @echo $(OBJ_FILES)

$(OBJ_DIR)/%.o : $(SRC_DIR)/%.c
    @echo begin to comple c source
    $(CC) -c $(CC_OPTS) -o $@ $<

$(OBJ_DIR)/%.o : $(ASM_DIR)/%.s
    @echo begin to comple asm
    $(ASM) -c $(ASM_OPTS)  -o $@ $<

# Make sure everything is rebuilt if this makefile is changed
$(OBJ_FILES) $(APP): makefile
```

2.2.3. 多預編譯宏

利用批處理，新增 MACRO 參數控制預編譯宏。

```
echo off
set list=__AAA_ __BBB__  __CCCC__ __DDDD__

for %%n in (%list%) do (
    make clean
```

```
    make MACRO=%%n )
```

3. 其他

3.1. 總結

命令列的方式使用簡單，但是靈活度不夠，例如一個 Target 不能指定宏。Makefile 比較靈活，但是前期建構工程會比較複雜，但是如果 makefile 建構完成之後，使用也非常方便。

3.2. 參考

https://developer.arm.com/documentation/101407/0537/Command-Line