GNU ld的linker script簡介

Wen Liao

Disclaimer

投影片資料為作者整理資料及個人意見，沒有經過嚴謹確認，請讀者自行斟酌

目標

簡介在GNU ld 吃的linker script 語法

測試環境: OS

```
$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 14.04.1 LTS
Release:        14.04
Codename:       trusty
```

假設你是天龍國內糊區的企業家。決定把手上不同地區的德國豬腳、萬巒豬腳、里港豬腳中央廚房合併成超級豬腳工廠，你會怎麼處理？

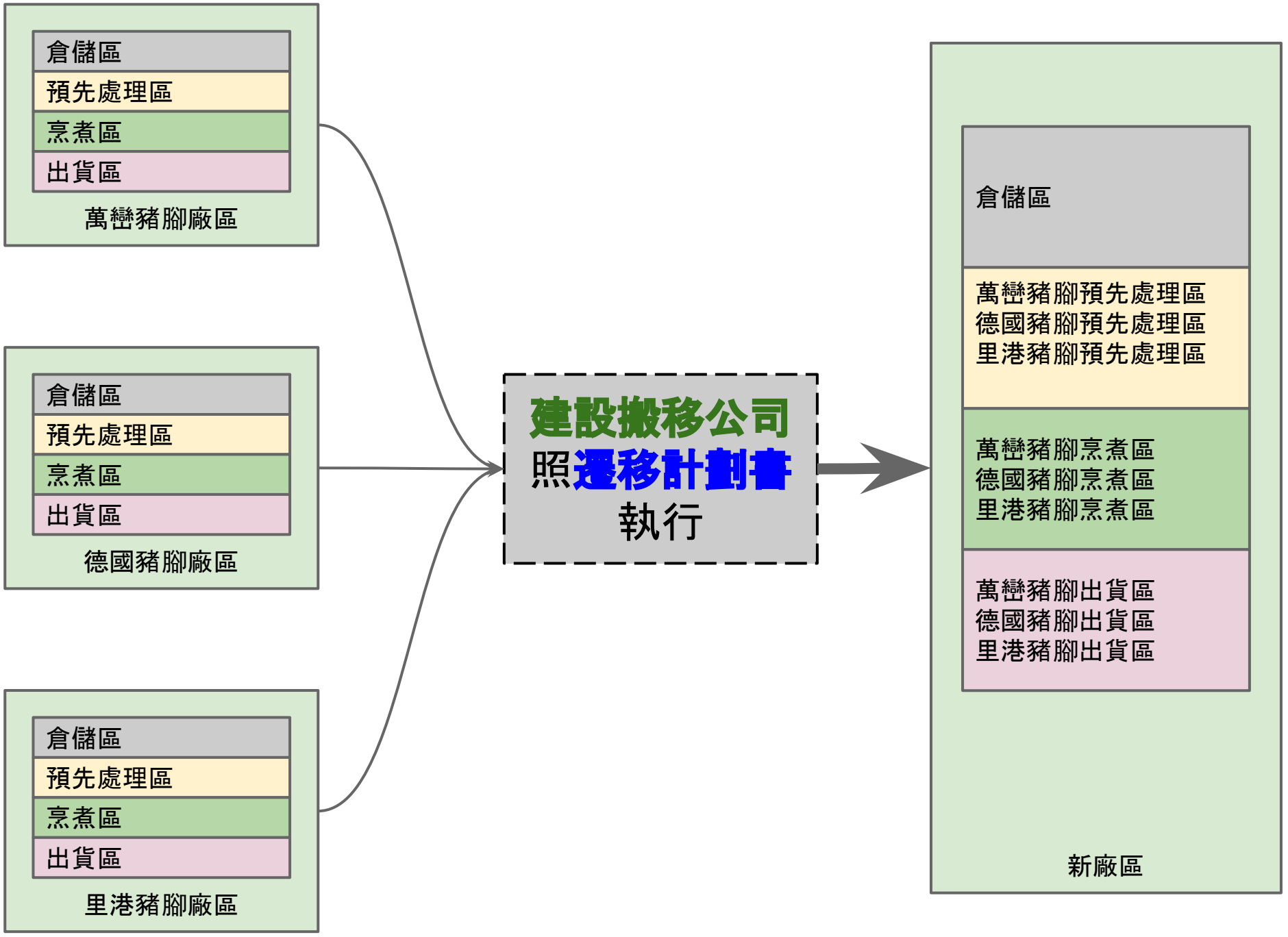
假設豬腳處理方式

- 取得原料
- 預先處理(醃漬、除毛等)
- 烹煮
- 出貨

如何合併？

廢話！當然是把相同功能的區塊規劃放在同一個場所

- 豬腳存放在同一個倉庫
- 預先處理在同一個廠區
- 建立烹煮區，切割成德國豬腳、萬巒豬腳、里港豬腳三個子區塊
- …


倉儲區
預先處理區
烹煮區
出貨區
萬巒豬腳廠區
倉儲區
預先處理區
烹煮區
出貨區
德國豬腳廠區
倉儲區
預先處理區
烹煮區
出貨區
里港豬腳廠區
建設搬移公司
照遷移計劃書
執行
倉儲區
萬巒豬腳預先處理區
德國豬腳預先處理區
里港豬腳預先處理區
萬巒豬腳烹煮區
德國豬腳烹煮區
里港豬腳烹煮區
萬巒豬腳出貨區
德國豬腳出貨區
里港豬腳出貨區
新廠區

恭喜!你已經知道linker在幹啥了

三小！？

你寫的程式不是只有描述行為，而是描述處理**資料**的**行為**！

- 上帝的歸上帝、凱薩的歸凱薩
- linker
  - 行為的歸行為，資料的歸資料、除錯的歸除錯、xx的歸xx

GNU ld 是啥？問問男人吧

man ld

...

ld combines a number of object and archive files, relocates their data and ties up symbol references. Usually the last step in compiling a program is to run ld.

...

ld combines a number of object and archive files, relocates their data and ties up symbol references. Usually the last step in compiling a program is to run ld.

...

英文！眼睛！我的眼睛！

不要擔心，聽眾是來聽分享，不是來學英文。當然內容儘量用中文，翻譯米糕啟動！

PS: 專用術語或重要意義會保留原文

名詞解釋1

- Object檔
  - relocatable 的機械碼
- Archive
  - 把一個或多個object檔壓成一個檔案

Demo 或是你的練習時間

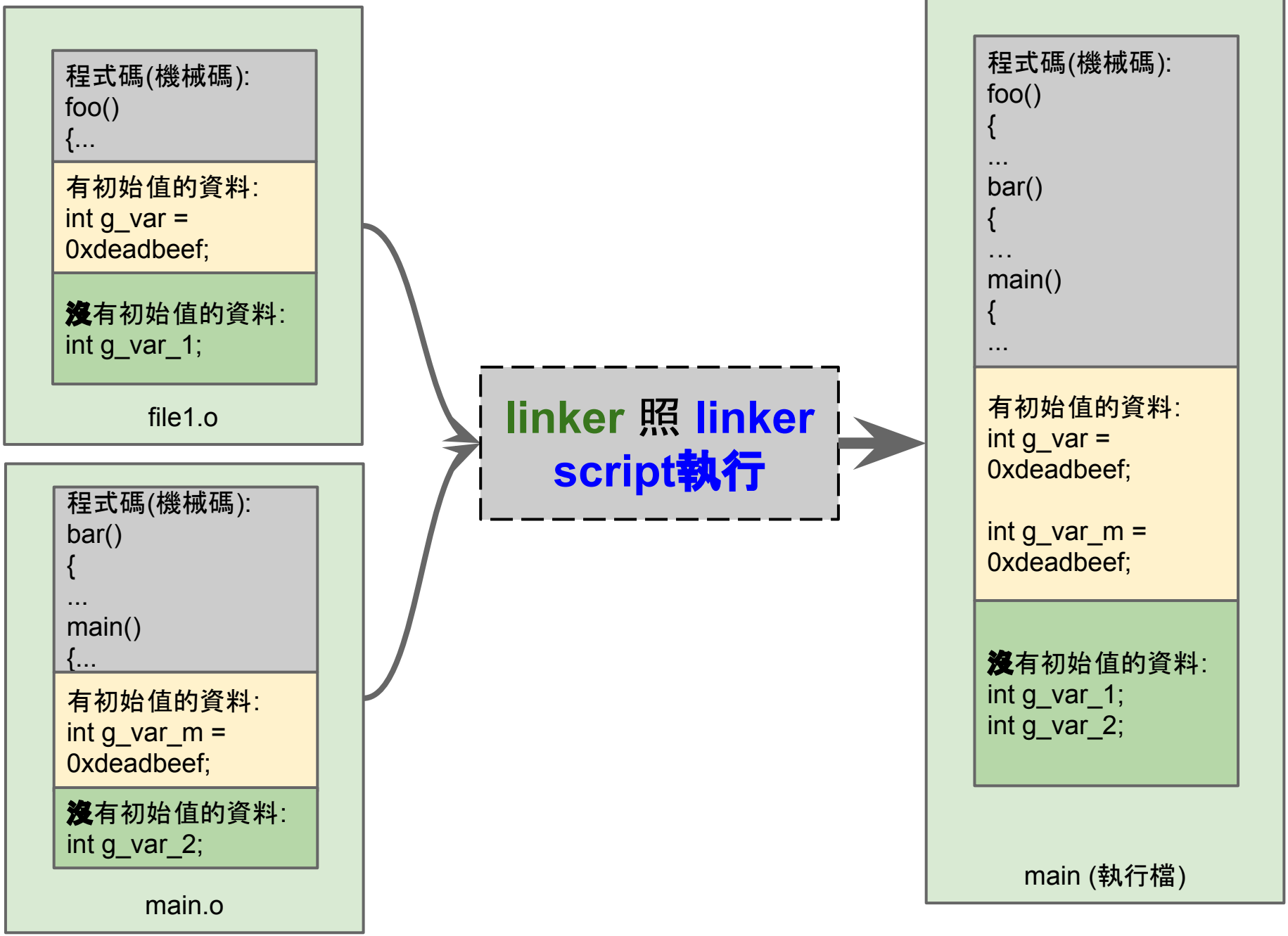
```
$ cd /tmp ; ar xv /usr/lib/x86_64-linux-gnu/libc.a
x - init-first.o
x - libc-start.o
...
x - errno.o
x - ctype.o
x - fprintf.o
x - printf.o
x - fscanf.o
x - scanf.o
x - strlen.o
x - read.o
x - write.o
```

和主題無關作業

- 找出write和printf的關係
- 找出為何需要printf而不用write

GNU ld

- 吃object 檔和archive檔，把他們的資料合併，連結symbol後，輸出成另外一個object檔
- 通常linker用在產生執行檔的最後一個步驟
- ld 要怎麼知道那塊資料放在那邊？當然是要有人告訴他


程式碼(機械碼):
foo()
{...
有初始值的資料:
int g_var =
0xdeadbeef;
沒有初始值的資料:
int g_var_1;
file1.o
程式碼(機械碼):
bar()
{
...
main()
{...
有初始值的資料:
int g_var_m =
0xdeadbeef;
沒有初始值的資料:
int g_var_2;
main.o
linker 照 linker
script執行
程式碼(機械碼):
foo()
{
...
bar()
{
…
main()
{
...
有初始值的資料:
int g_var =
0xdeadbeef;
int g_var_m =
0xdeadbeef;
沒有初始值的資料:
int g_var_1;
int g_var_2;
main (執行檔)

Demo 或是你的練習時間

```
$ ld --verbose
GNU ld (GNU Binutils for Ubuntu) 2.24
using internal linker script:
...
SECTIONS
{
  PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x400000));
  . = SEGMENT_START("text-segment", 0x400000) + SIZEOF_HEADERS;
  .interp         : { *(.interp) }
  ...
  .text           :
  {
    *(.text.unlikely .text.*_unlikely .text.unlikely.*)
...
 .data           :
  {
    *(.data .data.* .gnu.linkonce.d.*)
...
  .bss            :
  {
```

很複雜對不對？
我也懶得搞懂這是三小朋友

重點是,
ld會用到linker script

Linker script

- 每次link的時候，都會依照特定的命令去產生新的object檔。而這些命令就是linker script

- 換句話說，linker script提供一連串的命令讓linker照表操課

Linker script的目的

- 還記得豬腳的故事？
- 每個object檔都會有共通的區塊
- linker要透過script才知道把輸入object檔案哪個section的資料放在輸出object 檔的section, 以及最後放在記憶體的那個位置

因為很重要, 再講一次

- Linker script命令可以區分為
  - 平台記憶體長什麼樣子
  - 要把輸入object檔

名詞解釋2

- object 檔格式
  - 格式輸入檔案和輸出檔案所遵循的格式
- object 檔案
  - linker處理時讀入除了linker script外的輸入檔案和將結果存放的輸出檔案
- executable
  - ld輸出的檔案，有時候會這樣稱呼

名詞解釋3

- 每個object檔案都有好幾個section
  - input section：輸入object檔案中的section
  - output section：輸出object檔案中的section

- 常用section
  - .bss
    - 存放**沒有**初始值全域變數的地方 ex: int g_var;
  - .text
    - 存放編譯過的執行機械碼的地方
  - .data
    - 存放**有**初始值全域變數的地方 ex: int g_var = 0xdeadbeef;

名詞解釋4

- locale counter
  - 代表目前輸出object檔案位置的最後端，表示符號為.
- region
  - 執行平台實體的記憶體區塊。
    - 如0x1000~0x1999是ROM, 0x5000~0x9999是RAM。那麼這個平台就可以設定成有兩個region
    - 要注意RAM和ROM的差別唷

名詞解釋5

- Section
  - object存放檔案的區塊
    - 可能是資料，可能是程式碼
  - 內容
    - 名稱
    - 長度
    - 要放到平台記憶體的那個位址 (VMA)
    - 要從那塊記憶體載入 (LMA)
    - 檔案中存放的offset
    - alignment
    - 資料內容

名詞解釋6

- Section狀態
  - LOAD
    - 表示這個section需要從檔案載入到記憶體
  - DATA
    - 表示這個section存放資料，不可以被執行
  - READONLY
    - 可以望文生義吧？

名詞解釋7

- Section狀態 (接關）
  - ALLOC
    - 表示該section會吃記憶體，你可能會想說廢話，section不放記憶體放檔案是放心酸的嘛？還真的有，例如放除錯的section
  - CONTENTS
    - 表示這個section是執行程式所需要的資訊，如程式碼或是資料

Demo 或是你的練習時間

```
$ objdump -h /bin/ls

/bin/ls:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000400238  0000000000400238  00000238  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
...
 12 .text         0000f65a  00000000004028a0  00000000004028a0  000028a0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
...
 23 .data         00000254  000000000061a3a0  000000000061a3a0  0001a3a0  2**5
                  CONTENTS, ALLOC, LOAD, DATA
...
 24 .bss          00000d60  000000000061a600  000000000061a600  0001a5f4  2**5
                  ALLOC
```

終於回到主題了

- 這次要介紹的兩個主要指令
  - MEMORY
    - 描述平台記憶體區塊，還記得region嘛？
  - SECTIONS
    - 描述**輸出object檔案** section有幾個，裏面每個section該和哪些輸入object檔案合體。以及自訂symbol。

MEMORY

```
MEMORY
{
    name [(attr)] : ORIGIN = origin, LENGTH = len
    ...
}
```

MEMORY 欄位說明

- name
  - 你給這塊記憶體取的名稱，也就是說前面一直講的region(以下以region稱呼)。這個名稱不可以和同個linker script中以下的名稱相同
    - symbol名稱
    - section名稱
    - 檔案名稱

MEMORY 欄位說明

- attr
  - optional
  - 告訴linker這塊記憶體有什麼值得注意的地方，一個region可以有多個屬性，列出如下
    - R: Read only
    - W: 可讀寫
    - X: executable
    - A: 可allocate
    - I和L: Initialized section，據說是link後就用不到的section，所以不需要存到輸出object檔案中
    - !: 將該符號後面所有的屬性inverse

MEMORY 欄位說明

- ORIGIN
  - 一個expression，表示該region的起始位址
    - expression懶得講，請看參考資料
- LENGTH
  - region 大小，單位為byte

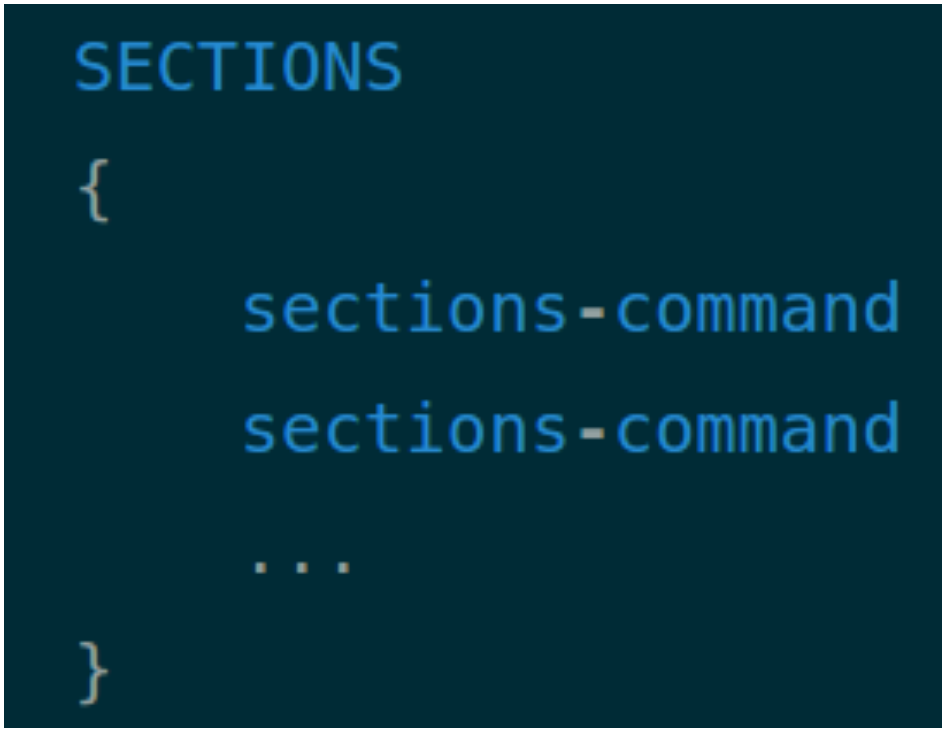
範例

- 唯讀、可執行
- 起始位址為0
- 長度為256k

```
MEMORY
{
    rom (rx)   : ORIGIN = 0, LENGTH = 256K
    ram (!rx) : org = 0x40000000, l = 4M
}
```

- 非唯讀、不可執行
- 起始位址為0x40000000
- 長度為4M
- 使用了縮寫，縮寫規則不想翻，請自己看參考資料

SECTIONS



Section commands?

- 主要的兩個用途
  - 設定symbol
  - 描述**輸出object檔案** section有幾個，裏面每個section該和哪些輸入object檔案體。

設定symbol

```
symbol = expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
symbol <<= expression ;
symbol >>= expression ;
symbol &= expression ;
symbol |= expression ;
```

範例

- 計算結果為數字
  - 大部分情況代表記憶體位置
  - 但是還是有可能不是記憶體位置
- 這些assignment有發生時間由上往下

```
. = 0x2000
_sdata = .
… (中間actions)
_edata = .
data_size = _edata - _sdata
_estack = ORIGIN(RAM) + LENGTH(RAM);
```

. 代表目前輸出locale 位置，所以_data是0x2000

輸出object檔案section描述格式

```
section [address] [(type)] :
    [AT(lma)]
    [ALIGN(section_align) | ALIGN_WITH_INPUT]
    [SUBALIGN(subsection_align)]
    [constraint]
    {
        output-section-command
        output-section-command
        ...
    } [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
```

因為是Optional，所以我只挑簡單我想講的部份

指定從特定位址將section載入記憶體

- 情境模擬
  - ROM裏面放有初始值的全域變數section,
  - 程式要去更動全域變數 => GG
  - 解法
    - 把這些section內的資料複製到RAM裏面
- LMA (load memory address)
- VMA (virtual memory address)
- 上面的情境模擬哪個是LMA，那個是VMA?

指定從特定位址將section載入記憶體

- AT(LMA)
  - 告訴linker這個section應該要去哪個LMA載入資料到VMA
- AT>lma_region
  - region, memory 指定的區塊
  - 告訴linker這個section LMA的資料放在那個section

指定該section要放在哪個region

- >region
  - 不解釋

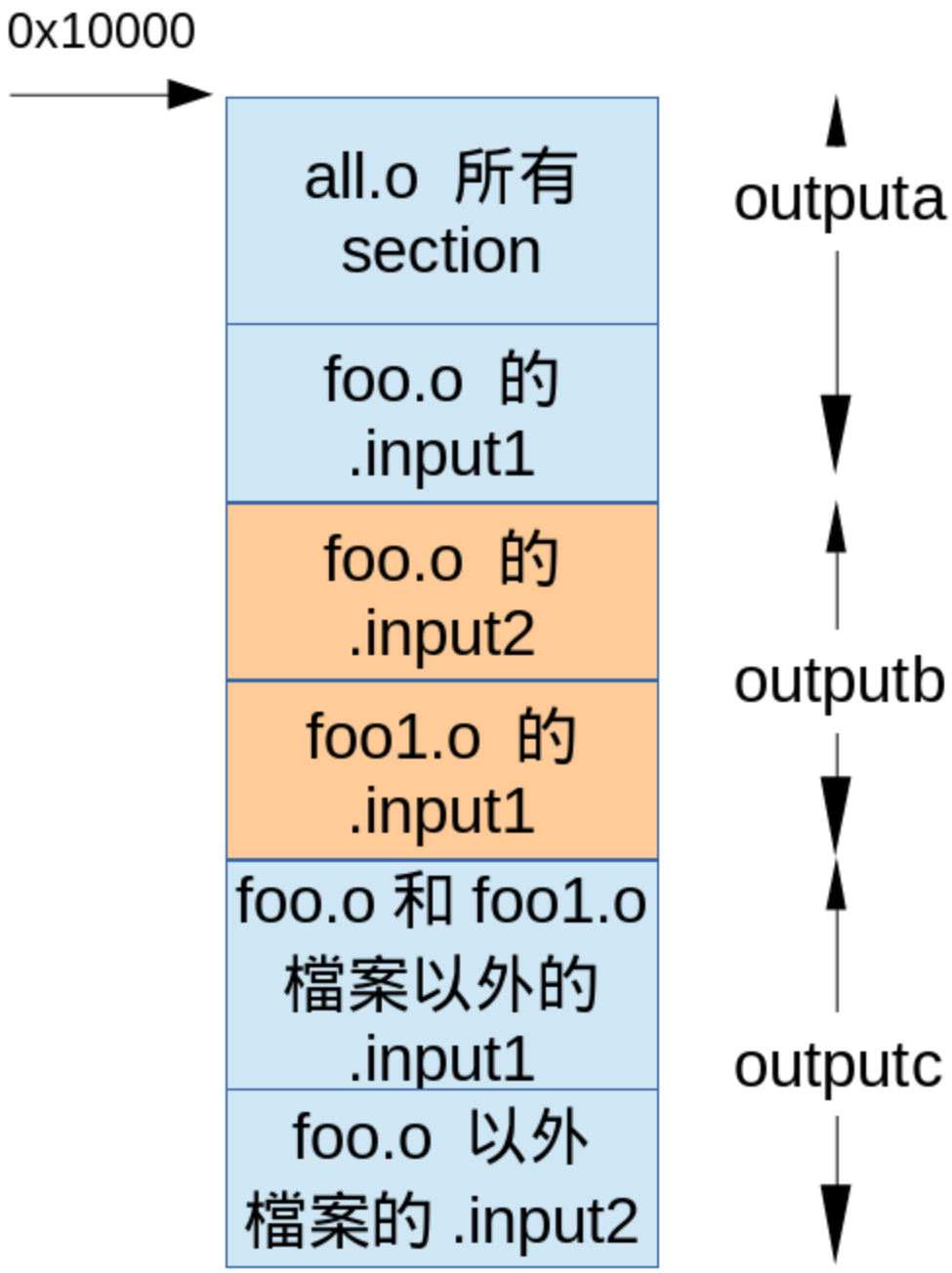
output-section-command

- 有很多，挑簡單我想講的
  - 設定symbol，前面講過，跳過。
    - 設定symbol可以在linker script的任何地方。
  - 輸入object 檔案的section應該要放到輸出object檔案的那個section

輸入object 檔案的section應該要放到輸出object檔案的那個section

- 格式 : 檔案(section1 section2 ...)
  - 檔案支援萬用字元
- 範例
  - *(.text)
    - 所有輸入object檔案的.text就放目前的section

```
SECTIONS {
    outputa 0x10000 :
    {
        all.o
        foo.o (.input1)
    }
    outputb :
    {
        foo.o (.input2)
        foo1.o (.input1)
    }
    outputc :
    {
        *(.input1)
        *(.input2)
    }
}
```


0x10000
all.o 所有 section
foo.o 的 .input1
foo.o 的 .input2
foo1.o 的 .input1
foo.o 和 foo1.o 檔案以外的 .input1
foo.o 以外檔案的 .input2
outputa
outputb
outputc

最後範例:rtenv的linker script

- rtenv
  - ARM CM3的RTOS
  - 台灣國立成功大學資訊工程系學生課堂作業
  - URL
    - https://github.com/southernbear/rtenv.git

```
ENTRY(main)
MEMORY
{
  FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 128K
  RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 20K
}
```

- 程式起始點是main
- 使用MEMORY指令設了兩個region，分別為FLASH和RAM
  - **請對照CM3 規格裏面的memory map和這邊的設定的數值！**

```
SECTIONS
{
    .text :
    {
        KEEP(*(.isr_vector))
        *(.text)
        ...
        _sromdev = .;
        *(.rom.*)
        _eromdev = .;
        _sidata = .;
    } >FLASH
```

- SECTIONS，描述輸出object檔案有幾個section
- .text section有會存放**所有**輸入object檔案的
  - .isr_vector, .text, .rom.開頭的等section
- .text 要放在FLASH的region
- symbol有_sromdev, eromdev, _sidata。它們有用處的，請自己下載rtenv trace 程式碼

```
.data : AT (_sidata)
{
  _sdata = .;
  *(.data)         /* Initialized data */
  *(.data*)
  _edata = .;
} >RAM
```

- .data的LMA (載入記憶體位址)是_sidata，就是.text結束的地方。另外這邊你要自己搬，有興趣請查原始碼
- .data要放在RAM的region
- 所有輸入object檔案的.data和所有.data開頭的section
- symbol _sdata和_edata分別代表section起始和結束位置

```
.bss : {
    _sbss = .;
    *(.bss)
    _ebss = .;
} >RAM
```

- .bss要放在RAM的region
- 所有輸入object檔案的.bss會放入這個輸出object檔案section
- 有_sbss和_ebss代表.bss的開始和結束位址

```
    _estack = ORIGIN(RAM) + LENGTH(RAM);
}
```

- 這個symbol沒放在section命令中
- 位址是RAM的開頭位址加上RAM的size
- 有印象程式使用的stack是由記憶體最後面往前面長的嘛？沒印象？那就估狗linux, stack, text的圖片吧

總結

- 本投影片簡單的介紹linker script的用處以及部份指令。最後以台灣成功大學資訊工程系課堂作業開發的小型RTOS裏面的linker script為範例做結尾。

- 部份指令就是說很多都省略掉，請自行參考附錄的參考資料

Q & A

參考資料

- GNU linker ld: Linker Scripts
  - https://sourceware.org/binutils/docs/ld/Scripts.html#Scripts
- GNU LD 手冊略讀 (0): 目錄和簡介
  - http://wen00072-blog.logdown.com/posts/246068-study-on-the-linker-script-0-table-of-contents
- rtenv的linker script解釋
  - http://wen00072-blog.logdown.com/posts/247207-rtenv-linker-script-explained