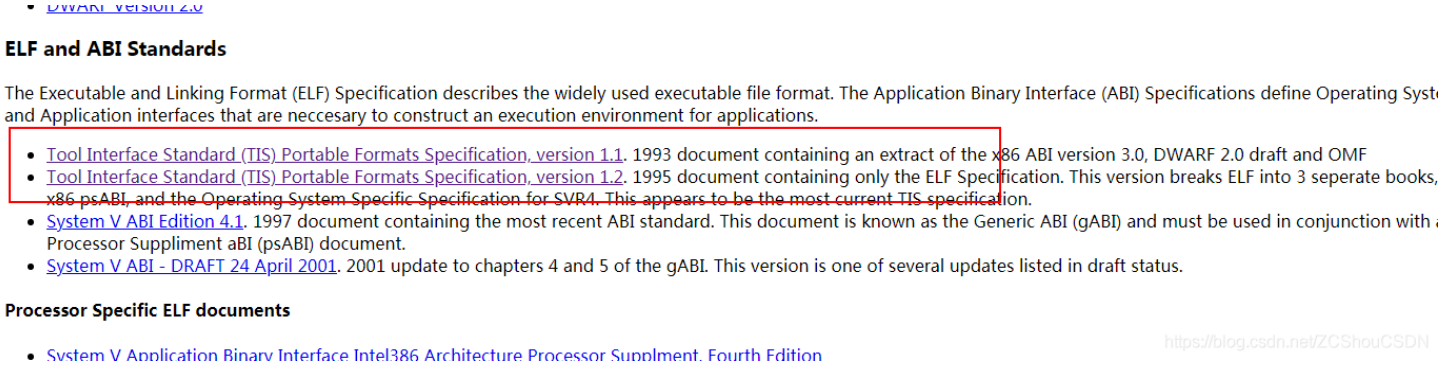
ARM 之 AXF 文件

ARM 之一 ELF 文件、鏡像（Image）文件、可執行文件、對象文件 詳解

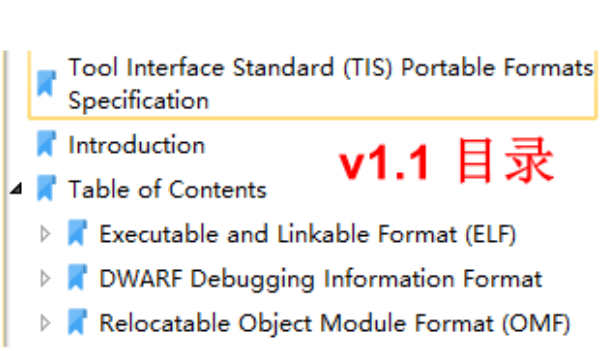
原創　　ZCShouEXP　　2019-08-25 13:22

ELF 文件規範

ELF（Executable and Linking Format）是一個二進制文件規範。用於定義不同類型的對象文件（Object files）中都放了什麼東西、以及都以什麼樣的格式去放這些東西。
現在流行的二進制可執行文件格式 (Executable File Format)，主要是 Windows 下的 PE（Portable Executable）和 Linux 的 ELF（Executable and Linking Format）可執行和鏈接格式)。他們都是 COFF（Common Object File Format）的變種。ARM 體系中採用的也是 ELF 文件格式。
COFF 是在 Unix System V Release 3 時由 UNIX 系統實驗室（UNIX System Laboratories, USL）首先提出並且使用的文件規範，後來微軟公司基於 COFF 格式，制定了 PE 格式標準，並將其用於當時的 Windows NT 系統。在 System V Release 4 時，UNIX 系統實驗室在 COFF 的基礎上，開發和發佈了 ELF 格式，作為應用程序二進制接口（Application Binary Interface，ABI）。
此後，工具接口標準委員會（Tool Interface Standard Committee，TISC）選擇了正在發展中的 ELF 標準作為工作在 32 位 INTEL 體繫上不同操作系統之間可移植的二進制文件格式。可以從 這裡 找到詳細的標準文檔。如下圖：



TISC 共出過兩個版本（v1.1 和 v1.2）的標準文檔。兩個版本內容上差不多，但 v1.2 版本重新組織了原本在 v1.1 版本中的內容。可讀性更高。兩個版本的目錄如下所示：

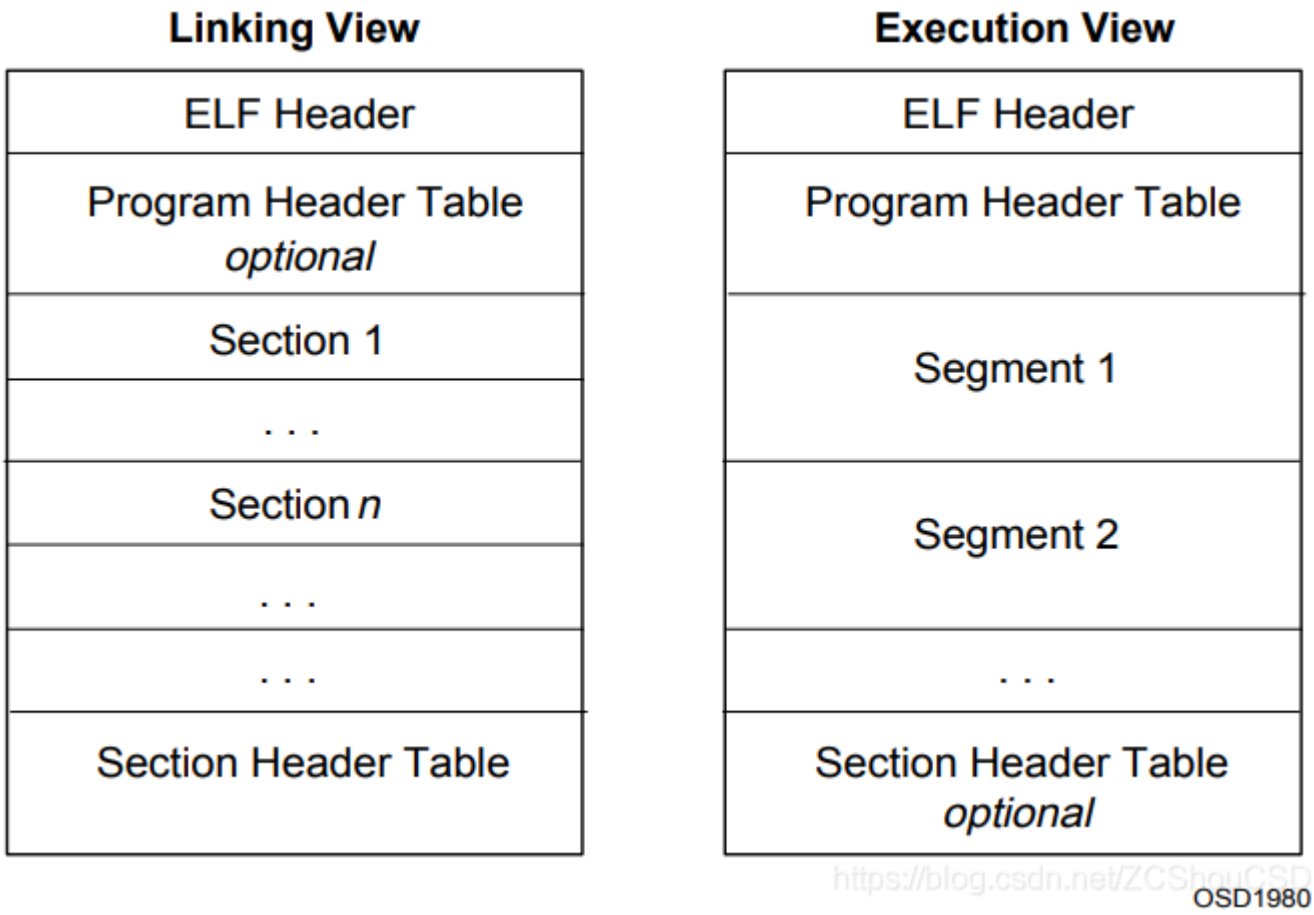


在 ELF 文件規範中，把系統中採用 ELF 格式的文件（規範中稱為**對象文件（Object File）**）歸類為以下三種：

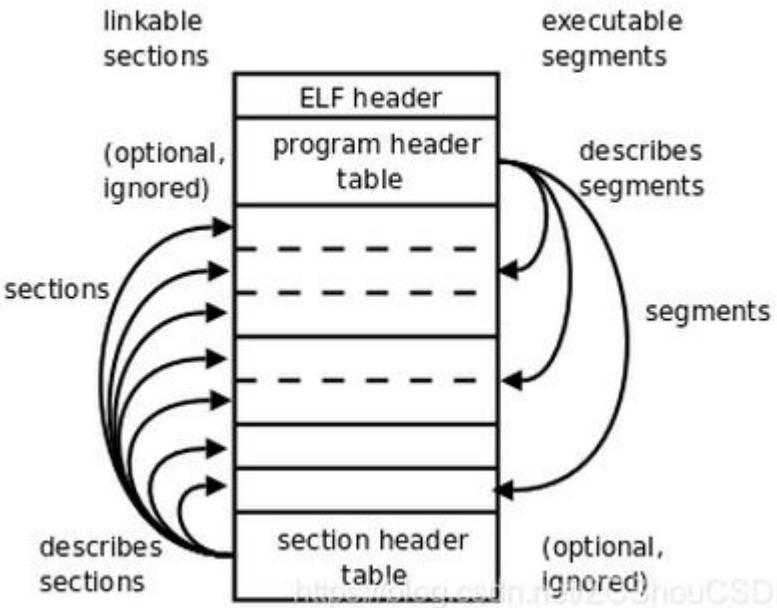
- **可重定位文件（Relocatable File ）：** 這類文件包含代碼和數據，可用來連接成可執行文件或共享對象文件（Object File），靜態鏈接庫歸為此類，對應於 Linux 中的.o ；Windows 的 .obj.
- **可執行文件（Executable File ）：** 這類文件包含了可以直接執行的程序，它的代表就是 ELF 可執行文件。Linux 下，他們一般沒有擴展名，比如 /bin/bash；Windows 下的 .exe
- **共享對象文件（Object File）（Shared Object File ）：** 這種文件包含代碼和數據，鏈接器可以使用這種文件跟其他可重定位文件的共享對象文件（Object File）鏈接，產生新的對象文件（Object File）。另外是動態鏈接器可以將幾個這種共享對象文件（Object File）與可執行文件結合，作為進程鏡像文件來運行。對應於 Linux 中的 .so，Windows 中的 DLL

在 Linux 系統中，還有一類文件，被稱為 **核心轉儲文件（Core Dump File）** ，當進程意外終止，系統可以將該進程地址空間的內容及終止時的一些信息轉存到核心轉儲文件。 對應 Linux 下的 core dump。

對象文件參與程序鏈接（構建程序）和程序執行（運行程序）。 為了方便和高效，對象文件（Object File）格式提供文件內容的並行視圖，反映了這些活動的不同需求。 下圖顯示了對象文件（Object File）的組織。



其中，各部分的含義都是規範定義好的！



數據表示法

對象文件（Object File）格式支持具有 8 位字節和 32 位體系結構的各種處理器。 然而，它旨在可擴展到更大（或更小）的體系結構。 因此，對象文件（Object File）用一種與機器無關的格式表示一些控制數據，從而可以識別對象文件（Object File）並以通用方式解釋它們的內容。 目標處理器中的剩餘數據使用目標處理器的編碼，而不管創建文件的機器如何。出於可移植性的原因，ELF 不使用位字段。

Name	Size	Alignment	Purpose
Elf32_Addr	4	4	Unsigned program address
Elf32_Half	2	2	Unsigned medium integer
Elf32_Off	4	4	Unsigned file offset
Elf32_Sword	4	4	Signed large integer
Elf32_Word	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

對象文件格式定義的所有數據結構都遵循相關類的自然大小和對齊準則。如果需要，數據結構包含顯式填充，以確保 4 字節對象的 4 字節對齊，強制結構大小為 4 的倍數，以此類推。數據從文件開始也有適當的對齊。因此，例如，包含 Elf32 Addr 成員的結構將在文件中的 4 字節邊界上對齊。

字符表示法

ELF 中對於符號的字符編碼也有一定的要求。當 ELF 接口文檔提到字符常量時，例如'/'或'\n'，它們的數值應遵循 7 位 ASCII 準則。 對於先前的字符常量，單字節值分別為 47 和 10。根據字符編碼，在 0 到 127 範圍之外的字符值可以佔用一個或多個字節。 應用程序可以根據需要使用不同語言的不同字符集擴展來控制自己的字符集。 儘管 TIS-一致性 不限制字符集，但它們通常應遵循一些簡單的指導原則：

- 0 到 127 之間的字符值應對應於 7 位 ASCII 代碼。 也就是說，編碼大於 127 的字符集應包含 7 位 ASCII 碼作為子集。
- 值大於 127 的多字節字符編碼應僅包含值在 0 到 127 範圍之外的字節。也就是說，每個字符使用多個字節的字符集不應嵌入類似於 7 位 ASCII 字符的字節。 一個多字節，非 ASCII 字符。

- 多字節字符應該是自我識別的。 例如，這允許在任何一對多字節字符之間插入任何多字節字符，而不改變字符的解釋。

關於 ELF 文件規範這裡就不多做詳細介紹了，感興趣的可以去 Linux 基金會的官方網站下載規範來看看！

ARM ELF 文件格式

ARM 體系中，所有文件均採用的 ELF 文件格式。我們可以在 ARM 的官網找到 ARM 關於 ARM ELF 文件格式的說明文檔。後文參考部分的下載中是目前可以從 ARM 官網找到的所有和 ARM ELF 相關的 PDF 文檔。
目前，我們可以找到的 ARM ELF 相關的文檔主要有 4 個：《ARM ELF File Format》、《ELF for the ARM® Architecture》、《ARM ELF》以及 ARM 的鏈接器手冊。其中，《ARM ELF File Format》是比較早期的文檔，針對於 ARM SDT 時代的 ELF 文件，有點過時了；後者三個則是最新的介紹文檔，《ELF for the ARM® Architecture》 僅僅是對 ARM ELF 取值的一些特殊說明，是在讀者先瞭解 ELF 文件規範的基礎上進行的說明。
ARM 中的各種源文件（包括彙編文件，C 語言程序及 C++ 程序等）經過 ARM 編譯器編譯後生成 ELF 格式的對象文件（Object File）（.o 文件）。這些對象文件（Object File）和相應的 C/C++ 運行時用到的庫經過 ARM 連接器處理後，生成 ELF 格式的鏡像文件（image），這種 ELF 格式的映像文件是一種可執行文件，可被寫入嵌入式設備的 ROM 中。
在 ARM 體系中，所有的二進制文件均被稱為 **對象文件**。其中，鏈接器最終生成的 ELF 格式的可執行文件又被稱為**鏡像文件（Image file）**。ARM ELF 鏡像文件或者對象文件由**輸入節（Input Sections）**、**輸出節（Output Sections）**、**域（Regions）** 和 **段（Segments）** 組

成，每個鏈接階段都有不同的鏡像視圖。如下圖所示：

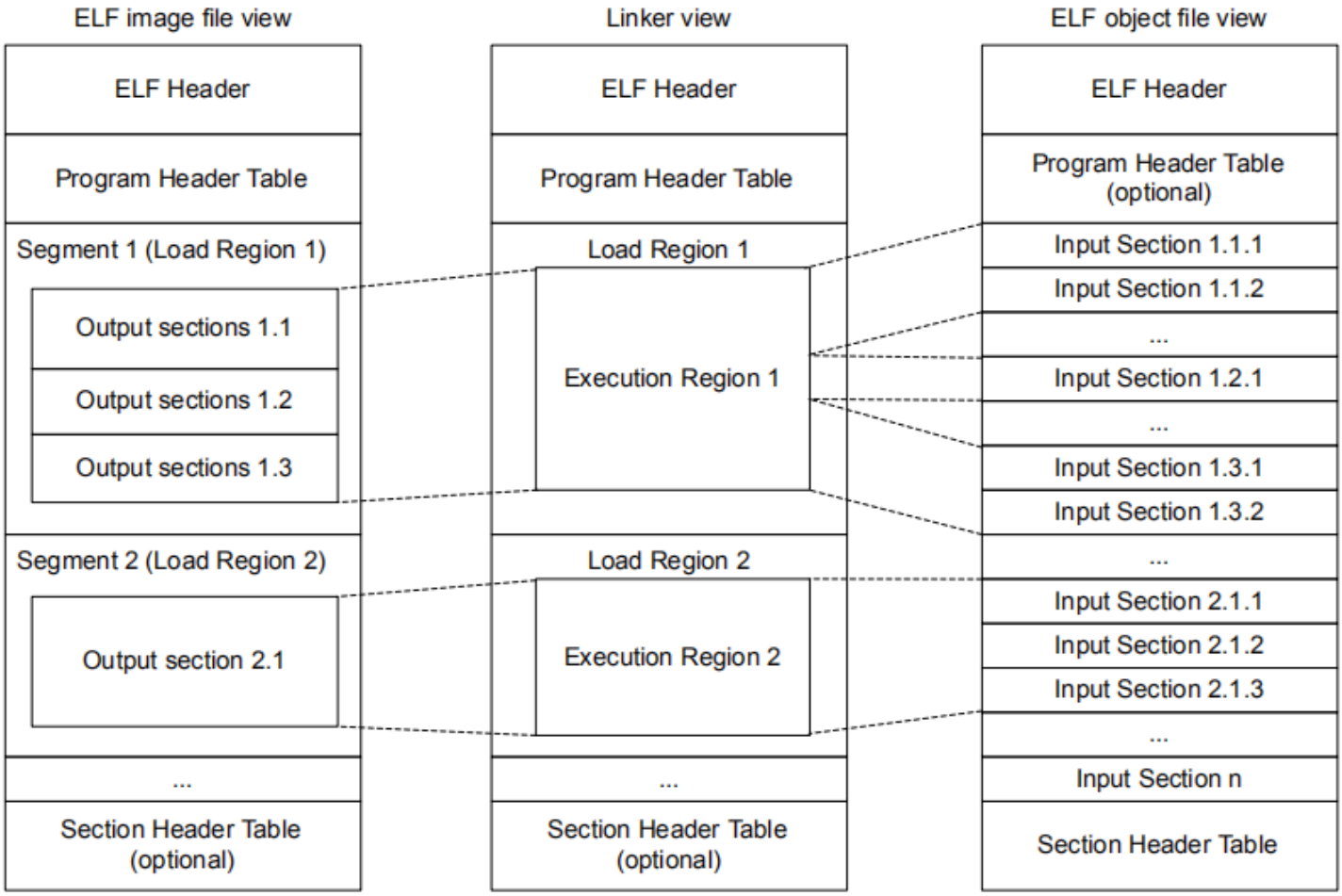


Figure 3-1 Relationship between sections, regions, and segments

- **ELF object file view (linker input)： ELF 對象文件視圖**由輸入節組成。 ELF 對象文件可以是：
  - 一個可重定位文件，包含適合與其他對象文件（Object File）鏈接的代碼和數據，以創建可執行文件或共享對象文件。
  - 包含代碼和數據的共享對象文件。
- **Linker view： 連接器視圖**針對程序地址空間會有兩個視圖。並且這兩個視圖在存在重疊，位置無關和可重定位的程序片段（代碼或數據）時變得不同：
  - 程序片段的加載地址是鏈接器期望外部代理（例如程序加載器，動態鏈接器或調試器）從 ELF 文件複製片段的目標地址。 這可能不是片段執行的地址。
  - 程序片段的執行地址是目標地址，其中鏈接器期望片段在參與程序的執行時駐留。

  如果片段與位置無關或可重定位，則其執行地址在執行期間可能會有所不同。

- **ELF image file view (linker output)： ELF 鏡像文件視圖**由程序段和輸出節組成：
  - 一個加載域對應於一個程序段。

- 一個執行域包含一個或多個以下輸出節：
  - RO section.
  - RW section.
  - XO section.
  - ZI section.

一個或多個執行域組成一個加載域。

When describing a memory view:

1. The term root region means a region that has the same load and execution addresses.
2. Load regions are equivalent to ELF segments.

輸入節 Input section

一個輸入節就是是輸入對象文件中的一個獨立的部分。 它包含代碼，初始化數據，或著是描述未初始化或必須在鏡像文件執行前設置為零的內存片段。 這些屬性由 RO，RW，XO 和 ZI 等屬性表示。 armlink 使用這些屬性將輸入節分組為更大的構建塊，稱為輸出節和域。

輸出節 Output section

一個輸出節就是一組輸入節的組合，它們具有相同的 RO，RW，XO 或 ZI 屬性，並且由鏈接器連續放置在存儲器中。 輸出節與組成它的輸入節具有相同的屬性。 在輸出節中，輸入節根據節放置規則進行排序。

域 Region

一個域最多包含四個輸出節，具體取決於內容和具有不同屬性的節的數量。 默認情況下，域中的輸出節根據其屬性進行排序。 首先是 XO 屬性的輸出節，然後是 RO 屬性的輸出節，再然後是 RW 屬性的輸出節，最後是 ZI 屬性的輸出節。 域通常會映射到物理存儲設備，例如 ROM，RAM 或外圍設備。 您可以使用分散加載文件來更改輸出節的順序。

程序段 Program segment

一個程序段對應於一個加載域，並且包含執行域。 程序段包含文本和數據等信息。

存在 XO（ execute-only）節時的注意事項

1. 您可以在同一執行域中混合 XO 和 非 XO 節。 但是，輸出的結果是一個 RO 節。
2. 如果輸入文件具有一個或多個 XO 節，則鏈接器將生成單獨的 XO ELF 段。 在最終鏡像中，除非使用分散加載文件或 --xo-base 選項另有指定，否則 XO 段緊接在 RO 段之前。

鏡像的加載視圖和執行視圖

鏡像的域在加載時放置在系統存儲器映射中。 內存中域的位置可能會在執行期間發生變化。在執行鏡像之前，可能必須將鏡像的某些域移動到其執行地址並創建 ZI 輸出節。 例如，初始化的 RW 數據可能必須從其 ROM 中的加載地址複製到 RAM 中的執行地址。鏡像的內存映射具有以下不同視圖：

加載視圖 Load view

根據鏡像加載到內存中時所處的地址，即鏡像執行開始前的位置，描述每個鏡像的域和節。

執行視圖 Execution view

根據鏡像執行期間所在的地址描述每個鏡像的域和節。

下圖顯示了沒有僅執行（XO）節的鏡像的這些視圖：

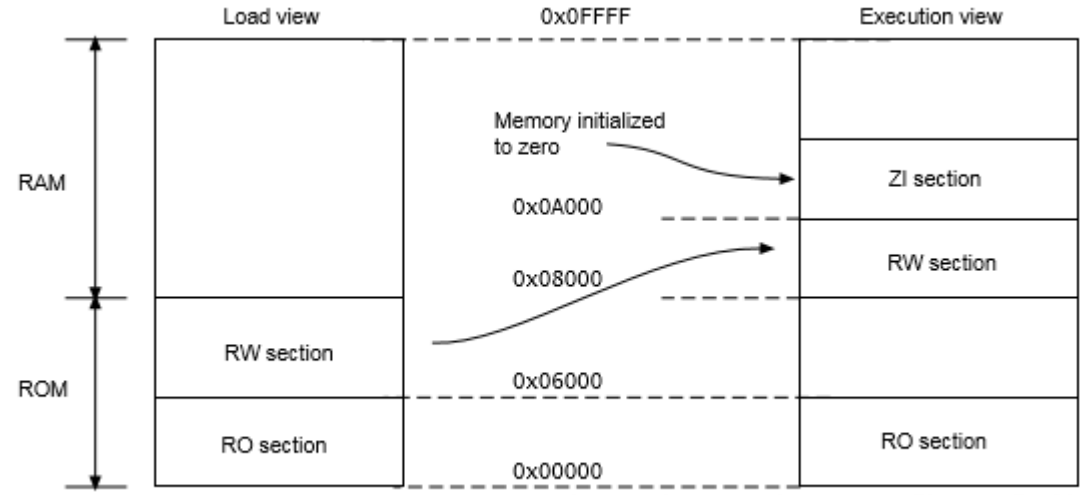


Figure 3-2 Load and execution memory maps for an image without an XO section

下圖顯示了具有 XO 節的鏡像的加載和執行視圖：

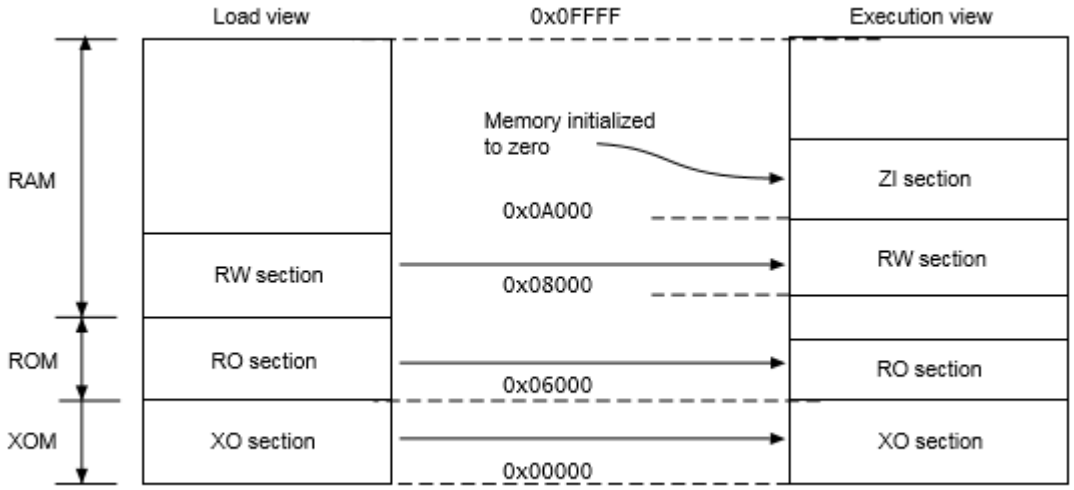


Figure 3-3 Load and execution memory maps for an image with an XO section

Image entry points

鏡像中的入口點就是鏡像中的一個位置（地址），該位置（地址）會被加載到 PC 寄存器。 它是程序執行開始的位置。　　雖然鏡像中可以有多個入口點，但在鏈接時只能指定一個入口點。
並非每個 ELF 文件都必須有入口點。 不允許在單個 ELF 文件中存在多個入口點。
對於嵌入式 Cortex-M 核的程序，程序的執行是從復位向量所在的位置（地址）開始執行。復位向量會被加載到 PC 寄存器中，且復位向量的位置（地址）並不固定。 通常，復位向量指向 CMSIS Reset_Handler 函數。
有兩種不同類型的入口點：

- 初始化入口點：鏡像的初始入口點是存儲在 ELF 頭文件中的單個值。 對於那些需要由操作系統或引導加載程序加載到 RAM 中的程序，加載程序通過將控制轉移到鏡像中的初始入口點來啓動鏡像執行。一個鏡像只能有一個初始化入口點。初始入口點可以是 ENTRY 指令設置的入口點之一，但不是必需的。
- ENTRY 指令指定的入口點：可以為鏡像從多個可能的入口點中選擇其中一個。每個鏡像只能有一個入口點。您可以在彙編程序文件中使用 ENTRY 指令在對象中創建入口點。 在嵌入式系統中，該指令的典型用途是標記進入處理器異常向量（例如 RESET，IRQ 和 FIQ）的代碼。該指令使用 ENTRY 關鍵字標記輸出代碼部分，該關鍵字指示鏈接器在執行未使用的部分消除時不刪除該部分。對於 C/C++ 程序，C 庫 中的 __main 就是入口點。

如果加載程序要使用嵌入式的映像，則它必須在標頭中指定一個初始入口點。 使用--entry 命令行選項選擇入口點。

ARM ELF 文件實例

與標準的 ELF 文件相比，ARM ELF 的某些值比較特殊，下面以實際文件來說明一下每個部分。編譯工具如下圖：

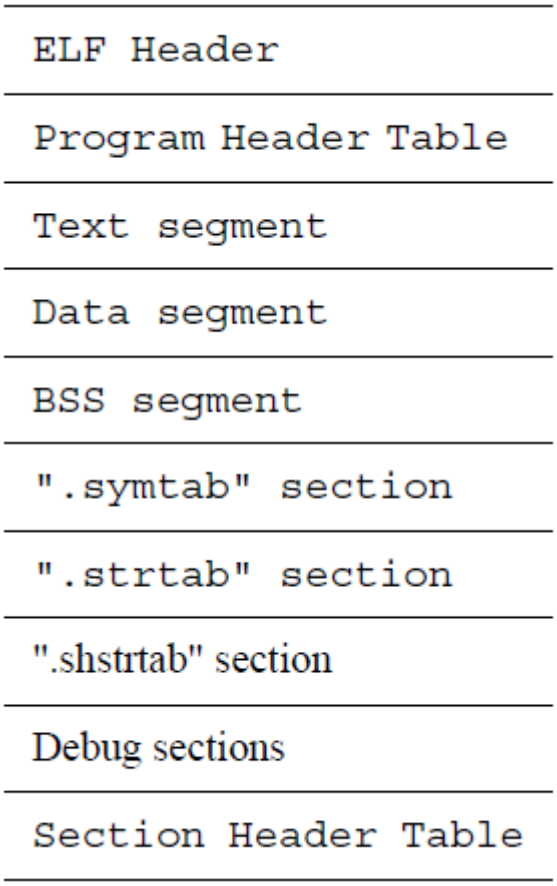
編譯後，會在對應目錄下生成 .o 文件和 .axf 文件，為了分析 ELF 文件，我們將使用 readelf 工具。在詳細解析之前，先用 Winhex 直接打開生成的 .o 文件，可以看到文件開頭有 ELF 字樣。表明它是一個 ELF 文件。如下：

Prog1.o

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	7F	45	4C	46	01	01	01	00	00	00	00	00	00	00	00	00	ELF
00000010	01	00	28	00	01	00	00	00	00	00	00	00	00	00	00	00	(
00000020	DC	04	00	00	00	00	00	05	34	00	00	00	00	00	28	00	Ü 4 (
00000030	0E	00	0B	00	00	10	A0	E1	2C	00	8F	E2	FE	FF	FF	EA	á, âþÿÿê
00000040	28	00	9F	E5	10	40	2D	E9	00	20	90	E5	04	00	90	E5	(Ÿå @-é å å
00000050	02	00	80	E0	01	00	80	E2	00	10	81	E0	08	00	8F	E2	€à €â à â
00000060	FE	FF	FF	EB	00	00	A0	E3	10	80	BD	E8	25	64	0A	00	þÿÿë ã €½è%d
00000070	00	00	00	00	00	00	00	00	43	6F	6D	70	6F	6E	65	6E	Componen

注意：.o 不是 ARM 的可執行文件！axf 為可執行文件。以下用兩種程序作對比。

一個簡單的可執行 ARM ELF 文件的概念佈局如下圖所示。請注意，文件中各部分的實際排序可能與下圖中的順序不同，因為只有 ELF Header 在文件中具有固定位置。



注意，針對目前最新版本的 ARM ELF，上圖有點過時！

ELF Header

ELF Header 描述了體系結構和操作系統等基本信息，並指出 Section Header Table 和 Program Header Table 在文件中的什麼位置。實際文件中，只有 ELF Header 位置是絕對的，且只能在最開始，其他部分部分的位置順序並不一定完全相同。

Program Header Table 在彙編和鏈接過程中沒有用到，所以在重定位文件中可以沒有；Section Header Table 中保存了所有 Section 的描述信息，Section Header Table 在加載過程中沒有用到，對於可執行文件，可以沒有該部分。當然，對於某些類型的文件來說，可以同時擁有 Program header table 和 Section Header Table，這樣 load 完後還可以重定位。（例如：shared objects）
ELF Header 可以使用如下數據結構表示：

```
#define EI_NIDENT 16

typedef struct {
    unsigned char   e_ident[EI_NIDENT]; // Magic
    Elf32_Half      e_type;            // Type
    Elf32_Half      e_machine;         // Machine
    Elf32_Word      e_version;         // Version
    Elf32_Addr      e_entry;           // Entry point address
    Elf32_Off       e_phoff;           // Start of program headers
    Elf32_Off       e_shoff;           // Start of section headers
    Elf32_Word      e_flags;           // Flags
    Elf32_Half      e_ehsize;          // Size of this header
    Elf32_Half      e_phentsize;       // Size of program headers
    Elf32_Half      e_phnum;           // Number of program headers
    Elf32_Half      e_shentsize;       // Size of section headers
    Elf32_Half      e_shnum;           // Number of section headers
    Elf32_Half      e_shstrndx;        // Section header string table index
} Elf32_Ehdr;
```

下面兩幅圖分別顯示了不同文件的 ELF Header。以上數據結構中的註釋，即對應於下圖中的各部分字段。
.o 文件 ELF Header 如下圖所示：

```
C:\Users\ZCShou\Desktop\ArmTest\MDK-ARM\Objects>readelf -h gpioconfig.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          428124 (bytes into file)
  Flags:                             0x5000000, Version5 EABI
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         253
  Section header string table index: 250
```


.axf 文件 ELF Header 如下圖所示：

```
C:\Users\ZCShou\Desktop\ArmTest\MDK-ARM\Objects>readelf -h ArmTest.axf
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x8000195
  Start of program headers:          485688 (bytes into file)
  Start of section headers:          485720 (bytes into file)
  Flags:                             0x5000402, Version5 EABI, hard-float ABI, <unknown>
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         1
  Size of section headers:           40 (bytes)
  Number of section headers:         16
  Section header string table index: 15
```


下面對以上兩幅圖中的內容做一下詳細介紹：

- 第 1 行 ELF Header：指名 ELF 文件頭開始。
- 第 2 行 Magic：用來指名該文件是一個 ELF 對象文件（Object File），對應於 Elf32_Ehdr 數據結構中的 unsigned char e_ident[EI_NIDENT];，使用以下宏值進行索引：

名稱	取值	意義

名稱	取值	意義
EI_MAG0	0	文件標識
EI_MAG1	1	文件標識
EI_MAG2	2	文件標識
EI_MAG3	3	文件標識
EI_CLASS	4	文件類
EI_DATA	5	數據編碼
EI_VERSION	6	文件版本
EI_PAD	7	補齊字節開始處
EI_NIDENT	16	e_ident[]大小

- e_ident[EI_MAG0] ~ e_ident[EI_MAG3]：包含了 ELF 文件的魔數，依次是 0x7f 和 ‘E’、‘L’、‘F’ 的 ASCII。
  e_ident[EI_CLASS]：取值如下

名稱	取值	意義
ELFCLASSNONE	0	非法類別
ELFCLASS32	1	32
ELFCLASS64	2	64

- ARM ELF 文件應包含 ELFCLASS32。
  e_ident[EI_DATA]：

名稱	取值	意義
ELFDATANONE	0	非法數據編碼
ELFDATA2LSB	1	高位在前
ELFDATA2MSB	2	低位在前

- 選擇將由執行環境中的默認數據順序控制。 在以 BE8 模式運行的 Architecture v6 處理器上，所有的指令均為小端格式。 適合在此模式下操作的可執行鏡像將在 e_flags 字段中設置 EF_ARM_BE8。
  e_ident[EI_VERSION]：指定 ELF 頭部的版本，當前必須為 1。
  e_ident[7]~e_ident[15]：是填充符，通常是 0
- 第 3 行 Class：該值就是 e_ident[EI_CLASS]。
- 第 4 行 Data：該值就是 e_ident[EI_DATA]。
- 第 5 行 Version：該值就是 e_ident[EI_VERSION]

- 第 6 行 OS/ABI：該值應該是 e_ident 的擴展部分。操作系統類型，ABI 是 Application Binary Interface 的縮寫。除非文件使用具有 OS 特定含義的標誌（例如，使用 SHN_LOOS 通過 SHN_HIOS 的段索引），否則該字段應為零。 目前，該字段有一個特定於處理器的值，如下。

取值	意義
ELFOSABI_ARM_AEABI (64)	該對象包含符號版本控制擴展，如§3.1.1 符號版本控制中所述。

- 第 7 行 ABI Version： 該值應該是 e_ident 的擴展部分。版本號，當前為 0 。
- 第 8 行 Type：表示該對象文件（Object File）類型。（上圖中的類型省略了 ET_）。

名稱	取值	意義
ET_NONE	0	未知對象文件（Object File）格式
ET_REL	1	可重定位文件
ET_EXEC	2	可執行文件
ET_DYN	3	共享對象文件（Object File）
ET_CORE	4	Core 文件（轉儲格式）
ET_LOPROC	0xff00	特定處理器文件 ET_LOPROC 和 ET_HIPROC 之間的取值用來標識與處理器相關的文件格式。
ET_HIPROC	0xffff	特定處理器文件

- 目前沒有特定於 ARM 的對象文件類型。 ET_LOPROC 和 ET_HIPROC 之間的所有值都保留給本規範的未來版本。
- 第 9 行 Machine：機器平臺類型。ARM 架構為 EM_ARM

Name	Value	Meaning
EM_NONE	0	No machine
EM_M32	1	AT&T WE 32100
EM_SPARC	2	SPARC
EM_386	3	Intel Architecture
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_860	7	Intel 80860
EM_MIPS	8	MIPS RS3000 Big-Endian
EM_MIPS_RS4_BE	10	MIPS RS4000 Big-Endian
	……	

Name	Value	Meaning
EM_ARM	40	ARM/Thumb Architecture

- 第 10 行 Version：當前對象文件（Object File）的版本號。

名稱	取值	意義	說明
EV_NONE	0	Invalid version	
EV_CURRENT	1	Current version	該項的取值可根據需要改變

- 第 11 行 Entry point address：程序的虛擬地址入口點。在 ARM 中：
  - 在可執行 ELF 文件中，e_entry 是鏡像唯一入口點的虛擬地址，如果鏡像沒有唯一入口點，則為 0。
  - 在可重定位 ELF 文件中，e_entry 是被 SHF_ENTRYSECT 所標記的段的入口點的偏移量，若沒有入口點，則為 0。
  - Bit[0] = 1，表示 Thumb 指令；Bit[0:1] = 00，表示 ARM 指令；Bit[0:1] = 10，保留；
    平臺標準可以指定可執行文件總是具有入口點，在這種情況下，e_entry 指定入口點，即使為零。
- 第 12 行 Start of program headers：程序頭的起始地址，.o 文件沒有 Program Headers 。
- 第 13 行 Start of section headers：節頭的起始地址。圖 4 的 486388 是十進制，即：表示節頭是從地址偏移 0x76BF4 處開始。
- 第 14 行 Flags：是一個與處理器相關聯的標誌。

名稱	意義
EF_ARM_ABIMASK (0xFF000000) (current version is 0x05000000)	此 ELF 文件符合的 ARM EABI 的版本，該值為一個 8 比特的掩碼。 當前 EABI 是版本 5。0 表示未知符合。
EF_ARM_BE8 (0x00800000)	ELF 文件包含適合在 ARM Architecture v6 處理器上執行的 BE-8 代碼。 該標誌只能在可執行文件上設置。
EF_ARM_GCCMASK (0x00400FFF)	gcc-arm-xxx 生成的舊版代碼(ABI 版本 4 及更早版本）可能會使用這些位。
EF_ARM_ABI_FLOAT_HARD (0x00000400) (ABI version 5 and later)	設置可執行文件頭（e_type = ET_EXEC 或 ET_DYN）以標註可執行文件的構建是為了符合硬件浮點過程調用標準。 與舊版（預版本 5）兼容，gcc 用作 EF_ARM_VFP_FLOAT
EF_ARM_ABI_FLOAT_SOFT (0x00000200) (ABI version 5	設置在可執行文件頭（e_type = ET_EXEC 或 ET_DYN）中明確標註可執行文件的構建符合軟件浮點過程調用

名稱	意義
and later)	標準（基準標準）。 如果 EF_ARM_ABI_FLOAT_XXXX 位都清零，則默認符合基本過程調用標準。 與舊版（預版本 5）兼容，gcc 用作 EF_ARM_SOFT_FLOAT。

- 注意：以上部分與 ARM 早期文檔是有區別的，很多值已經不同
- 第 15 行 Size of this header：ELF 文件頭的字節數。
- 第 16 行 Size of program headers：Program Headers 大小。.o 文件大小為 0。
- 第 17 行 Number of program headers：Program Headers 的數量（可以有多個）。
- 第 18 行 Size of section headers：sections header 的大小
- 第 19 行 Number of section headers：sections header 的數量。
- 第 20 行 Section header string table index：節頭部表格中與節名稱字符串表相關的表項的索引。如果文件沒有節名稱字符串表，此參數可以為 SHN_UNDEF。

注意：實際文件中，每一部分的位置順序並不一定完全相同，只有 ELF Header 位置是絕對的，且只能在最開始。

Section Header（節頭）

節頭表提供了對 ELF 文件中所有節的訪問。節中包含對象文件（Object File）中的所有信息，除了：ELF 頭部、程序頭部表格、節頭部 表格。節滿足以下條件：

1. 對象文件（Object File）中的每個節都有對應的節頭部描述它，反過來，有節頭部不意 味著有節。
2. 每個節佔用文件中一個連續字節域（這個區域可能長度為 0）。
3. 文件中的節不能重疊，不允許一個字節存在於兩個節中的情況發生。
4. 對象文件（Object File）中可能包含非活動空間（INACTIVE SPACE）。這些區域不屬於任何 頭部和節，其內容未指定。

ELF 頭部中，e_shoff 成員給出從文件頭到節頭部表格的偏移字節數；e_shnum 給出表格中條目數目；e_shentsize 給出每個項目的字節數。從這些信息中可以確切地定位節的具體位置、長度。節頭部表格中比較特殊的幾個下標如下：

名稱	取值	說明
SHN_UNDEF	0	標記未定義的、缺失的、不相關的，或者沒有含義的節引用
SHN_LORESERVE	OxFF00	保留索引的下界
SHN_LOPROC	0xFF00	SHN_HIPROC 0XFF1F 保留給處理器特殊的語義
SHN_ABS	1	包含對應引用量的絕對取值。這些值不會被重定位所 影響

名稱	取值	說明
SHN_COMMON	2	相對於此節定義的符號是公共符號。如 FORTRAN 中 COMMON 或者未分配的 C 外部變量。
SHN_HIRESERVE		保留索引的上界

介於 SHN_LORESERVE 和 SHN_HIRESERVE 之間的表項不會出現在節頭部表中。

.o 文件 Section Header(部分)

```
E:\ARM\STM32F302RE\Project\Objects>readelf -S gpioconfig.o
There are 253 section headers, starting at offset 0x68044:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .rev16_text       PROGBITS        00000000 000034 000004 00  AX  0   0  4
  [ 2] .revsh_text       PROGBITS        00000000 000038 000004 00  AX  0   0  4
  [ 3] .rrx_text         PROGBITS        00000000 00003c 000006 00  AX  0   0  4
  [ 4] .emb_text         PROGBITS        00000000 000044 000006 00  AX  0   0  4
  [ 5] i.CarrierWaveRese PROGBITS        00000000 00004c 00003c 00  AX  0   0  4
  [ 6] i.CarrierWaveSet  PROGBITS        00000000 000088 000024 00  AX  0   0  4
  [ 7] i.GPIOUsedInit    PROGBITS        00000000 0000ac 000130 00  AX  0   0  4
  [ 8] i.GPIO_CarrierWav PROGBITS        00000000 0001dc 0000bc 00  AX  0   0  4
  [ 9] i.GPIO_ClockInit  PROGBITS        00000000 000298 00001c 00  AX  0   0  2
  [10] i.GPIO_IWDGInit   PROGBITS        00000000 0002b4 000028 00  AX  0   0  2
  [11] i.IWDG_Feed       PROGBITS        00000000 0002dc 00002a 00  AX  0   0  2
  [12] i.SysTickConfig   PROGBITS        00000000 000308 000064 00  AX  0   0  4
  [13] .data             PROGBITS        00000000 00036c 000018 00  WA  0   0  4
  [14] .debug_info       PROGBITS        00000000 000384 0000cc 00      0   0  1
                                 .......
[242] .rel.debug_info    REL             00000000 06584c 0000d0 08     168 135  4
[243] .rel.debug_info    REL             00000000 06591c 0000e0 08     168 139  4
[244] .rel.debug_info    REL             00000000 0659fc 000068 08     168 143  4
[245] .rel.debug_info    REL             00000000 065a64 000038 08     168 147  4
[246] .rel.debug_info    REL             00000000 065a9c 000018 08     168 151  4
[247] .rel.debug_info    REL             00000000 065ab4 000020 08     168 155  4
[248] .rel.debug_info    REL             00000000 065ad4 000018 08     168 159  4
[249] .rel.debug_info    REL             00000000 065aec 000020 08     168 163  4
[250] .shstrtab          STRTAB          00000000 065b0c 00064f 00      0   0  1
[251] .strtab            STRTAB          00000000 06615b 001e61 00      0   0  1
[252] .ARM.attributes    ARM_ATTRIBUTES  00000000 067fbc 000087 00      0   0  1
ey to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
```


.axf 文件 Section Header

```
C:\Users\ZCShou\Desktop\ArmTest\MDK-ARM\Objects>readelf -S ArmTest.axf
There are 16 section headers, starting at offset 0x76958:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] ER_IROM1          PROGBITS        08000000 000034 002170 00  AX  0   0  4
  [ 2] RW_IRAM1          PROGBITS        20000000 0021a4 000080 00  WA  0   0  4
  [ 3] RW_IRAM1          NOBITS          20000080 002224 000f60 00  WA  0   0  8
  [ 4] .debug_abbrev     PROGBITS        00000000 002224 0005c4 00      0   0  1
  [ 5] .debug_frame      PROGBITS        00000000 0027e8 000af4 00      0   0  1
  [ 6] .debug_info       PROGBITS        00000000 0032dc 00a748 00      0   0  1
  [ 7] .debug_line       PROGBITS        00000000 00da24 0038c4 00      0   0  1
  [ 8] .debug_loc        PROGBITS        00000000 0112e8 000f58 00      0   0  1
  [ 9] .debug_macinfo    PROGBITS        00000000 012240 05b658 00      0   0  1
  [10] .debug_pubnames   PROGBITS        00000000 06d898 000c57 00      0   0  1
  [11] .symtab           SYMTAB          00000000 06e4f0 002230 10     12 315  4
  [12] .strtab           STRTAB          00000000 070720 00234c 00      0   0  1
  [13] .note             NOTE            00000000 072a6c 00001c 00      0   0  4
  [14] .comment          PROGBITS        00000000 072a88 003e14 00      0   0  1
  [15] .shstrtab         STRTAB          00000000 07689c 00009c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (noread), p (processor specific)
```


上圖中的表頭可以用如下數據結構描述（對應關係見註釋）：

```
typedef struct {
    Elf32_Word sh_name;         // name
    Elf32_Word sh_type;         // Type
    Elf32_Word sh_flags;        // Flg
    Elf32_Addr sh_addr;         // Addr
    Elf32_Off sh_offset;        // Off
    Elf32_Word sh_size;         // Size
    Elf32_Word sh_link;         // Lk
    Elf32_Word sh_info;         // Inf
    Elf32_Word sh_addralign;    // Al
    Elf32_Word sh_entsize;      // ES
} Elf32_Shdr;
```

- sh_name：給出節名稱。是節頭部字符串表節（Section Header String Table Section）的索引。名字是一個 NULL 結尾的字符串。ELF 文件規定一些標準節的名字，例如.text、.data、.bss。此外，如上圖中，許多節名字都是 ARM 自己擴展的。
- sh_type：為節的內容和語義進行分類。ARM ELF 只使用了其中的一部分。參見下表（部分）。

名稱	取值	含義
SHT_NULL	0	此值標誌節頭部是非活動的，沒有對應的節。此節頭部中的其他成員取值無意義。
SHT_PROGBITS	1	此節包含程序定義的信息，其格式和含義都由程序來解釋釋。
SHT_SYMTAB	2	此節包含一個符號表。目前對象文件（Object File）對每種類型的節都只能包含一個，不過這個限制將來可能發生變化。一般，SHT_SYMTAB 節提供用於鏈接編輯（指 ld 而言） 的符號，儘管也可用來實現動態鏈接。
SHT_STRTAB	3	此節包含字符串表。對象文件（Object File）可能包含多個字符串表節。
SHT_RELA	4	此節包含重定位表項，其中可能會有補齊內容（addend），例如 32 位對象文件（Object File）中的 Elf32_Rela 類型。對象文件（Object File）可能擁有多個重定位節。
SHT_HASH	5	此節包含符號哈希表。所有參與動態鏈接的目標都必須包含一個符號哈希表。目前，一個對象文件（Object File）只能包含一個哈希表，不過此限制將來可能會解除。
SHT_DYNAMIC	6	此節包含動態鏈接的信息。目前一個對象文件（Object File）中只能包含一個動態節，將來可能會取消這一限制。
SHT_NOTE	7	此節包含以某種方式來標記文件的信息。
SHT_NOBITS	8	這種類型的節不佔用文件中的空間，其他方面和 SHT_PROGBITS 相似。儘管此節不包含任何字節，成員 sh_offset 中還是會包含概念性的文件偏移
SHT_REL	9	此節包含重定位表項，其中沒有補齊（addends），例如 32 位對象文件（Object File）中的 Elf32_rel 類型。對象文件（Object File）中可以擁有多個重定位節。

- 除了以上標準節類型外，ARM 架構下，還有以下特殊的類型：

名稱	取值	含義
SHT_ARM_EXIDX	0x70000001	異常索引表
SHT_ARM_PREEMPTMAP	0x70000002	BPABI DLL 動態鏈接搶佔地圖
SHT_ARM_ATTRIBUTES	0x70000003	對象文件兼容性屬性
SHT_ARM_DEBUGOVERLAY	0x70000004	
SHT_ARM_OVERLAYSECTION	0x70000005	

- sh_flags：字段定義了一個節中包含的內容是否可以修改、是否可以執行等信息。如果一個標誌比特位被設置，則該位取值為 1。未定義的各位都設置為 0。

名稱	取值	含義
SHF_WRITE	0x1	節包含進程執行過程中將可寫的數據
SHF_ALLOC	0x2	此節在進程執行過程中佔用內存。某些控制節並不出現於目標 文件的內存映像中，對於那些節，此位應設置為 0
SHF_EXECINSTR	0x4	節包含可執行的機器指令
SHF_MASKPROC	0xF0000000	所有包含於此掩碼中的四位都用於處理器專用的語義

- ARM 中的特殊取值如下：

Name	Value	Purpose
SHF_ARM_NOREAD	0x20000000	本節的內容不應由程序執行者讀取

- sh_addr：如果節將出現在進程的內存鏡像中，此成員給出節的第一個字節應處的位置。否則，此字段為 0。
- sh_link 和 sh_info：根據節類型的不同，sh_link 和 sh_info 的具體含義也有所不同。ARM 取值如下：

sh_type	sh_link	sh_info
SHT_SYMTAB, SHT_DYNSYM	相關聯的字符串表的節頭部索引	最後一個局部符號（綁定 STB_LOCAL）的符號表索引值加一
SHT_DYNAMIC	此節中條目所用到的字符串表格 的節頭部索引	0
SHT_HASH	此哈希表所適用的符號表的節頭部索引	0
SHT_REL、SHT_RELA	相關符號表的節頭部索引	重定位所適用的節的節頭部索引
其它	SHN_UNDEF	0

- sh_addralign：節沒有最小對齊要求。 但是，包含 thumb 代碼的部分必須至少為 16 位對齊，並且包含 ARM 代碼的部分必須至少為 32 位對齊。具有 SHF_ALLOC 屬性的任何節必須滿足 sh_addralign >= 4。其他節可根據需要對齊。 例如，調試表通常沒有對齊要求。並且輸入到靜態鏈接器的數據段可以自然對齊。
  平臺標準可能會限制他們可以保證的最大對齊（通常是頁面大小）。
- sh_entsize：某些節中包含固定大小的項目，如符號表。對於這類節，此成員給出每個表項的長度字節數。如果節中並不包含固定長度表項的表格，此成員取值為 0。

- sh_size：此成員給出本節的長度（字節數）。除非節的類型是 SHT_NOBITS，否則節佔用文件中的 sh_size 字節。類型為 SHT_NOBITS 的節長度可能非零，不過卻不佔用文件中的空間。
- sh_offset：此成員的取值給出節的第一個字節與文件頭之間的偏移。不過，SHT_NOBITS 類型的節不佔用文件的空間，因此其 sh_offset 成員給出的是其概念性的偏移。

注意：

1. 保留給處理器體系結構的節名稱一般構成為：**處理器體系結構名稱簡寫 + 節名稱**。且處理器名稱應該與 e_machine 中使用的名稱相同。例如：圖 5 最後的 .ARM.attributes
2. 對象文件（Object File）中也可以包含多個名字相同的節。
3. 上圖節名 ER_IROM1、RW_IRAM1、RW_IRAM 是由連接器的分散加載文件指定的名稱。可以根據需要自行修改。

ARM 節名稱是以下面列出的具有預定義含義的標準前綴之一開始的名稱，或者是包含美元（$）字符的名稱。 在 ARM EABI 下沒有其他具有特殊意義的段名稱。

節前綴名	節類型	節屬性	解釋
.bss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE	本節保存有助於程序內存映像的未初始化數據。 根據定義，當程序開始運行時，系統將使用零初始化數據。 該部分不佔用文件空間，如段類型 SHT_NOBITS 所示。
.comment	SHT_PROGBITS	None	本節包含版本控制信息
.data	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE	這些部分保存有助於程序內存映像的已初始化數據
.data1	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE	
.debug…	SHT_PROGBITS	None	本節保存符號調試信息。 內容未指定。具有前綴.debug 的所有段名保留供將來使用
.dynamic	SHT_DYNAMIC	SHF_ALLOC [+SHF_WRITE]	本節保存動態鏈接信息，並具有 SHF_ALLOC 和 SHF_WRITE 等屬性。 操作系統和處理器確定 SHF_WRITE 位是否被置位
.hash	SHT_HASH	[SHF_ALLOC]	本節包含一個符號哈希表。
.line	SHT_PROGBITS	None	本節保存符號調試的行號信息，其中描述了源程序和機器代碼之間的對應關係。內容未指定

節前綴名	節類型	節屬性	解釋
.rodata	SHT_PROGBITS	SHF_ALLOC	這些部分保存通常有助於過程映像中的不可寫段的只讀數據
.rodata1	SHT_PROGBITS	SHF_ALLOC	
.rel name .rela name	SHT_REL SHT_RELA	[SHF_ALLOC]	這些節中包含了重定位信息。如果文件中包含可加載的段，段中有重定位內容，節的屬性將包含 SHF_ALLOC 位，否則該位置 0。傳統上 name 根據重定位所適用的節 區給定。例如 .text 節的重定位節名字，將是:.rel.text 或者 .rela.text。
.shstrtab	SHT_STRTAB	None	本節保存節名稱。
.strtab	SHT_STRTAB	[SHF_ALLOC]	此節包含字符串，通常是代表與符號表項相關的名稱。如果文件擁有一個可加載的段，段中包含符號串表，節的屬性將包含 SHF_ALLOC 位，否則該位為 0。
.symtab	SHT_SYMTAB	[SHF_ALLOC]	此節包含一個符號表。如果文件中包含一個可加載的段，並且該段中包含符號表，那 麼節的屬性中包含 SHF_ALLOC 位，否則 該位置為 0。
.text	SHT_PROGBITS	SHF_ALLOC+ SHF_EXECINSTR	本節包含程序的文本或可執行指令

除了以上標準節外，ARM 架構下，還有以下特殊的節：

節前綴名	節類型	節屬性	說明
.ARM.exidx*	SHT_ARM_EXIDX	SHF_ALLOC + SHF_LINK_ORDER	以.ARM.exidx 開頭的節包含部分展開的索引條目。
.ARM.extab*	SHT_PROGBITS	SHF_ALLOC	以.ARM.extab 開頭的節包含異常展開信息的名稱部分。
.ARM.preemptmap	SHT_ARM_PREEMPTMAP	SHF_ALLOC	以.ARM.preemptmap 開頭的節包含一個 BPABI DLL 動態鏈接優先地圖。
.ARM.attributes	SHT_ARM_ATTRIBUTES	none	包含構建屬性
.ARM.debug_overlay	SHT_ARM_DEBUGOVERLAY	none	
.ARM.overlay_table	SHT_ARM_OVERLAYSECTION	See DBGOVL for	

節前綴名	節類型	節屬性	說明
			details

這裡需要注意一下 Debug Sections。Debug Sections 僅在調試時使用，稍微複雜一些。ARM 可執行 ELF 文件的調試節中包含多種類型的調試信息，ELF 可執行文件的使用者（如 armlink）可以通過檢查可執行文件的節表來區分這些種類型的調試信息。
ARM 系列的開發工具在不同的發展時期，採用的調試信息是有區別的，後來統一採用 DWARP。目前採用的應該是 3.0 版本。具體如下：

- ASD debugging tables：
  These provide backwards compatibility with ARM’s Symbolic Debugger. ASD debugging information is stored in a single Section in the executable named .asd.
- DWARP version 1.0
  When DWARF 1.0 debugging information is included by the linker in the ELF executable, the file contains the following ELF Sections, each of which has a Section Header Table entry:

Section name	Contents
.debug	debugging entries
.line	fileinfo entries
.debug_pubnames	table for accelerated access to debug items
.debug_aranges	address ranges for compilation units

- DWARF version 2.0
  When DWARF 2.0 debugging information is included by the linker in the ELF executable, the file contains the following ELF sections, each of which has a Section Header Table entry:

Section name	Contents
.debug_info	debugging entries
.debug_line	fileinfo statement program
.debug_pubnames	table for accelerated access to debug items
.debug_aranges	address ranges for compilation units
.debug_macinfo	macro information (#define / #undef)
.debug_frame	call frame information
.debugj_abbrev	abbreviation table

Section name	Contents
.debug_str	debug string table

關於 DWARF 調試標準詳見：http://www.dwarfstd.org/。目前最新版本是 The DWARF Debugging Standard Version 5

Program Headers（程序頭）

可執行文件或者共享對象文件（Object File）的程序頭部是一個結構數組，每個結構描述了一個段或者係統準備程序執行所必需的其它信息。對象文件（Object File）的"段"包含一個或者多個"節"，也就是"段內容（Segment Contents）"。程序頭部僅對於可執行文件和共享對象文件（Object File）有意義。

圖 7 Program Header

```
C:\Users\ZCShou\Desktop\ArmTest\MDK-ARM\Objects>readelf -l gpioconfig.o

There are no program headers in this file.

C:\Users\ZCShou\Desktop\ArmTest\MDK-ARM\Objects>readelf -l ArmTest.axf

Elf file type is EXEC (Executable file)
Entry point 0x8000195
There are 1 program headers, starting at offset 485688

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000034 0x08000000 0x08000000 0x021f0 0x03150 RWE 0x8

 Section to Segment mapping:
  Segment Sections...
   00     ER_IROM1                    https://blog.csdn.net/ZCShouCSDN
```

程序頭可以使用如下數據結構來表示（對應關係見註釋）：

```
typedef struct {
    Elf32_Word    p_type;       // Type
    Elf32_Off     p_offset;     // Offset
    Elf32_Addr    p_vaddr;      // VirtAddr
    Elf32_Addr    p_paddr;      // PhyAddr
    Elf32_Word    p_filesz;     // FileSiz
    Elf32_Word    p_memsz;      // MemSiz
    Elf32_Word    p_flags;      // Flg
    Elf32_Word    p_align;      // Align
} Elf32_Phdr;
```

- p_type：這個成員告訴這個數組元素描述什麼樣的段，或者如何解釋數組元素的信息。類型值及其含義如下圖所示。

名稱	取值	意義
PT_NULL	0	數組元素未使用; 其他成員的值是未定義的。 此類型使程序頭表已忽略條目。
PT_LOAD	1	數組元素指定由 p_filesz 和 p_memsz 描述的可加載段。
PT_DYNAMIC	2	數組元素指定動態鏈接信息。
PT_INTERP	3	數組元素指定要作為解釋器調用的以空值結尾的路徑名的位置和大小。
PT_NOTE	4	數組元素指定輔助信息的位置和大小。
PT_SHLIB	5	該段類型是保留的，但具有未指定的語義。
PT_PHDR	6	數組元素（如果存在）指定程序頭表本身的位置和大小。
PT_ARM_ARCHEXT	0x70000000	Platform architecture compatibility information
PT_ARM_EXIDX PT_ARM_UNWIND	0x70000001	Exception unwind tables

- p_offset：此成員給出從文件頭到該段第一個字節的偏移
- p_vaddr：此成員給出段的第一個字節將被放到內存中的虛擬地址。
- p_paddr：此成員僅用於與物理地址相關的系統中。因為 System V 忽略所有應用程序的物理地址信息，此字段對與可執行文件和共享對象文件（Object File）而言，具體內容是未指定的。
- p_filesz：此成員給出段在文件鏡像中所佔的字節數。可以為 0。
- p_memsz： 此成員給出段在內存鏡像中佔用的字節數。可以為 0。
- p_flags：此成員給出與段相關的標誌。

名稱	取值	意義
PF_X	1	可執行的段
PF_W	2	可寫的段
PF_R	4	可讀的段
PF_MASKPROC	0xf0000000	保留

- p_align：可加載的進程段的 p_vaddr 和 p_offset 取值必須合適，相對於對頁面大小的取模而言。此成員給出段在文件中和內存中如何 對齊。數值 0 和 1 表示不需要對齊。否則 p_align 應該是個 正整數，並且是 2 的冪次數，p_vaddr 和 p_offset 對 p_align 取模後應該相等。

Symbol table（符號表）

一個對象文件的符號表保存了定位和重定位所在程序的符號定義和引用所需的信息。符號表以數組的下標進行索引。0 指定表中的第一個條目，並用作未定義的符號索引。ARM 結構中，符號表與標準的 ELF 文件沒有任何區別。

圖 12 .o 文件 Symbol table（部分）

```
E:\ARM\STM32F302RE\Project\Objects>readelf -s gpioconfig.o

Symbol table '.symtab' contains 226 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 NOTYPE  LOCAL  DEFAULT    1 $t
     2: 00000000     0 NOTYPE  LOCAL  DEFAULT    2 $t
    18: 00000000     0 NOTYPE  LOCAL  DEFAULT   13 $d.realdata
    19: 00000000     0 NOTYPE  LOCAL  DEFAULT  166 $d.realdata
    20: 00000000     0 FILE    LOCAL  DEFAULT  ABS ..\\User\\src\\GPIOConfig
    21: 00000000     4 SECTION LOCAL  DEFAULT    1 .rev16 text
   208: 00000000     0 OBJECT  GLOBAL HIDDEN    81 __ARM_grp..debug_pubnames
   209: 00000000     0 OBJECT  GLOBAL HIDDEN   115 __ARM_grp..debug_pubnames
   210: 00000001    52 FUNC    GLOBAL HIDDEN     5 CarrierWaveReset
   218: 00000000     0 FUNC    GLOBAL HIDDEN   UND GPIO_Init
   219: 00000000     0 FUNC    GLOBAL HIDDEN   UND GPIO_ReadInputDataBit
   220: 00000000     0 FUNC    GLOBAL HIDDEN   UND GPIO_SetBits
   221: 00000000     0 FUNC    GLOBAL HIDDEN   UND GPIO_WriteBit
   222: 00000000     0 FUNC    GLOBAL HIDDEN   UND RCC_AHBPeriphClockCmd
   223: 00000000     0 OBJECT  GLOBAL HIDDEN   UND SystemCoreClock
   224: 00000000     0 FUNC    WEAK   HIDDEN   UND __aeabi_lib$$Request$$armlib
```


在 C 語言中，符號表保存了程序實現或使用的所有全局變量和函數，如果程序引用一個自身代碼未定義的符號，則稱之為未定義符號，這類引用必須在靜態鏈接期間用其他目標模塊或庫解決，或在加載時通過動態鏈接解決。

符號表可以使用以下數據結構表示：

```
typedef struct {
    Elf32_Word      st_name;    // Name
    Elf32_Addr      st_value;   // Value
    Elf32_Word      st_size;    // Size
    unsigned char   st_info;    //
    unsigned char   st_other;
    Elf32_Half      st_shndx;   // Ndx
} Elf32_Sym;
```

- st_name：該成員將對象文件（Object File）的符號字符串表中的索引保存在符號名稱的字符表示中
- st_value：該成員給出相關聯的符號的值。 根據上下文，這可能是絕對值，地址等等；不同對象文件（Object File）類型的符號表條目對 st_value 成員的解釋略有不同。
    - 在可重定位文件中，st_value 保持其索引為 SHN_COMMON 的符號的對齊約束。
    - 在可重定位文件中，st_value 包含已定義符號的節偏移量。 也就是說，st_value 是 st_shndx 標識的部分開頭的偏移量。
    - 在可執行文件和共享對象文件中，st_value 包含虛擬地址 1。 為了使這些文件的符號對動態鏈接器更有用，段偏移（文件解釋）讓位於與段號無關的虛擬地址（存儲器解釋）。
- st_size：許多符號具有相關尺寸。 例如，數據對象的大小是對象中包含的字節數。 如果符號沒有大小或未知的大小，該成員將保持 0。
- st_info：該成員指定符號的類型和綁定屬性。 值和值的列表如下面兩個表格所示。 以下代碼顯示瞭如何操作這些值。
- ```
  #define ELF32_ST_BIND(i) ((i)>>4)
  ```
- ```
  #define ELF32_ST_TYPE(i) ((i)&0xf)
  ```
- ```
  #define ELF32_ST_INFO(b,t) (((b)<<4)+((t)&0xf))
  ```

A symbol’s binding determines the linkage visibility and behavior.

Name	Value	Meaning
STT_NOTYPE	0	The symbol’s type is not specified.
STT_OBJECT	1	The symbol is associated with a data object, such as a variable, an array, and so on.
STT_FUNC	2	The symbol is associated with a function or other executable code.
STT_SECTION	3	The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have STB_LOCAL binding.
STT_FILE	4	A file symbol has STB_LOCAL binding, its section index is SHN_A BS, and it precedes the other STB_LOCAL symbols for the file, if it is present.
STT_LOPROC	13	Values in this inclusive range are reserved for processor-specific semantics. If a symbol’s value refers to a specific location within a section, its section index member, st_shndx, holds an index into the section header table. As the section moves during
  ```

Name	Value	Meaning
		relocation, the symbol’s value changes as well, and references to the symbol continue to point to the same location in the program. Some special section index values give other semantics.
STT_HIPROC	15	

In each symbol table, all symbols with STB_LOCAL binding precede the weak and global symbols. A symbol’s type provides a general classification for the associated entity. Figure 3-17, Symbol Types, ELF32_ST_TYPE

Name	Value	Meaning
STT_NOTYPE	0	The symbol’s type is not specified.
STT_OBJECT	1	The symbol is associated with a data object, such as a variable, an array, and so on.
STT_FUNC	2	The symbol is associated with a function or other executable code.
STT_SECTION	3	The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have STB_LOCAL binding.
STT_FILE	4	A file symbol has STB_LOCAL binding, its section index is SHN_A BS, and it precedes the other STB_LOCAL symbols for the file, if it is present.
STT_LOPROC	13	Values in this inclusive range are reserved for processor-specific semantics. If a symbol’s value refers to a specific location within a section, its section index member, st_shndx, holds an index into the section header table. As the section moves during relocation, the symbol’s value changes as well, and references to the symbol continue to point to the same location in the program. Some special section index values give other semantics.
STT_HIPROC	15	

- st_other：該成員目前只有 0，沒有定義。
- st_shndx：每個符號表條目與某些部分有關"定義"; 該成員保存相關部分標題表索引。如上圖 3-7 和 3.3.1 節所述，一些段索引表示特殊含義。

The symbols in ELF object files convey specific information to the linker and loader. See section 4, ARM- and Thumb-Specific Definitions, for a description of the actual linking model used in the system.

- SHN_ABS：The symbol has an absolute value that will not change because of relocation.
- SHN_COMMON：The symbol labels a common block that has not yet been allocated. The symbol's value gives alignment constraints, similar to a section's sh_addralign member. That is, the link editor will allocate the storage for the symbol at an address that is a multiple of st_value. The symbol's size tells how many bytes are required.
- SHN_UNDEF：This section table index means the symbol is undefined. When the link editor combines this object file with another that defines the indicated symbol, this file's references to the symbol will be linked to the actual definition.

As mentioned above, the symbol table entry for index 0 (STN_UNDEF) is reserved. It is shown in Figure 3-18. Figure 3-18, Symbol Table Entry: Index 0

Name	Value	Note
st_name	0	No name
st_value	0	Zero value
st_size	0	No size
st_info	0	No type, local binding
st_other	0	
st_shndx	SHN_UNDEF	No section

String table（字符串表）

字符串表節包含以 NULL（ASCII 碼 0）結尾的字符序列，通常稱為字符串。ELF 對象文件（Object File）通常使用字符串來表示符號和節名稱。對字符串的引用通常以字符串在字符 串表中的下標給出。ARM 結構中，字符串表與標準的 ELF 文件沒有任何區別。

axf 文件

axf 文件是 ARM 的調試文件，其格式符合上一節講的對象文件（Object File）格式（ELF）。其中除了包含了完整的 bin 文件外，還附加了其他的調試信息。在調試的時候，這些調試信息是不必下到 RAM 中去的，真正下到 RAM 中的信息僅僅是可執行代碼。下圖為 axf 文件的

頭部。

通過直接查看完整的 axf 文件可以看出，axf 中絕大多數都是和調試相關的內容。真正的 Bin 只是其中的一小部分。Bin 的結尾處在 axf 文件中也很容易找到，再次就不在贅述。
既然前面我們說了，axf 文件就是 ELF 文件格式，那麼我們可以使用 readelf 工具，具體查看一下 axf 文件。下圖是一個 axf 文件的節表

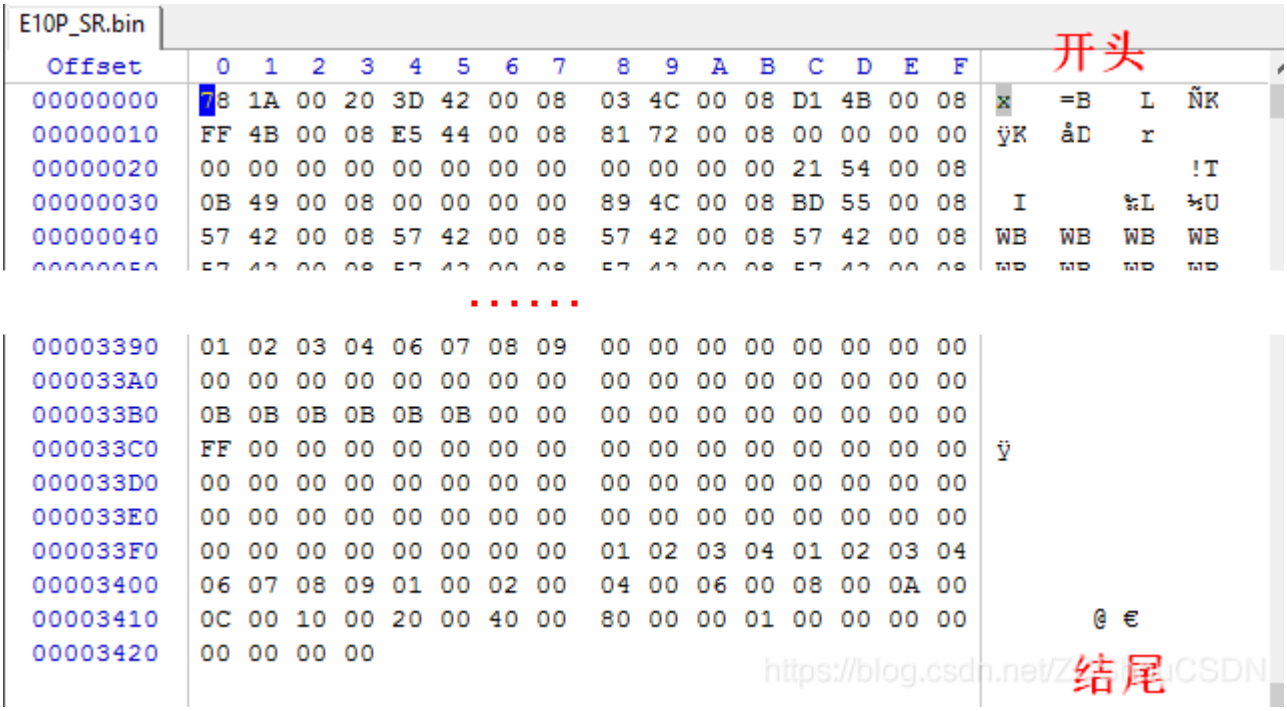
```
E:\ARM>readelf -S E10P_SR.axf
There are 16 section headers, starting at offset 0x7c054:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] ER_IROM1          PROGBITS        08004000 000034 003384 00  AX  0   0  4
  [ 2] RW_IRAM1          PROGBITS        20000000 0033b8 0000a0 00  WA  0   0  4
  [ 3] RW_IRAM1          NOBITS          200000a0 003458 0019d8 00  WA  0   0  8
  [ 4] .debug_abbrev     PROGBITS        00000000 003458 0005c4 00      0   0  1
  [ 5] .debug_frame      PROGBITS        00000000 003a1c 000d30 00      0   0  1
  [ 6] .debug_info       PROGBITS        00000000 00474c 00bee0 00      0   0  1
  [ 7] .debug_line       PROGBITS        00000000 01062c 0047a8 00      0   0  1
  [ 8] .debug_loc        PROGBITS        00000000 014dd4 0015b4 00      0   0  1
  [ 9] .debug_macinfo    PROGBITS        00000000 016388 05bb84 00      0   0  1
  [10] .debug_pubnames   PROGBITS        00000000 071f0c 000f79 00      0   0  1
  [11] .symtab           SYMTAB          00000000 072e88 002870 10     12 376  4
  [12] .strtab           STRTAB          00000000 0756f8 0026bc 00      0   0  1
  [13] .note             NOTE            00000000 077db4 00001c 00      0   0  4
  [14] .comment          PROGBITS        00000000 077dd0 0041c8 00      0   0  1
  [15] .shstrtab         STRTAB          00000000 07bf98 00009c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```


Bin 文件

bin 文件是 ARM 的可執行文件，是最純粹的二進制機器代碼。與 HEX 文件包括地址信息的不同，BIN 文件格式只包括了數據本身。在燒寫或下載 HEX 文件的時候，一般都不需要用戶指定地址，因為 HEX 文件內部的信息已經包括了地址。而燒寫 BIN 文件的時候，用戶是一定需要指定地址信息的。
ARM 的 Bin 文件就是 axf 的精華部分（掐掉 ELF 頭，去掉 .symtab、.debug 和.symtab 區

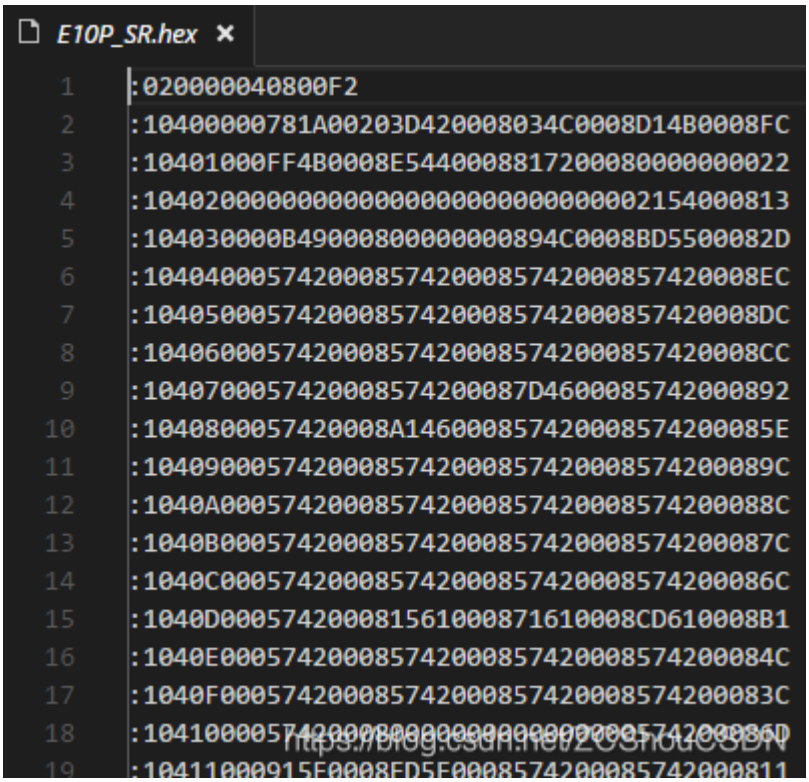
裡的信息）。下圖是筆者使用 Winhex 截取的 ARM 的 Bin 文件的開頭和結尾的示意圖。



hex 文件

首先，hex 文件最初由 Intel 提出。在 Intel HEX 文件中，每一行是一個 HEX 記錄，由十六進制數組成的機器碼或者數據常量，Intel HEX 文件經常被用於將程序或數據傳輸存儲到 ROM、EPROM，大多數編程器和模擬器使用 Intel HEX 文件。

hex 文件全部由可打印的 ASCII 字符組成。如下圖就是 ARM-MDK5.22 生成的一個 hex 文件（部分）



從上圖不難看出，hex 文件就是一個個的十六進制的字符串。實際上，一個 Intel HEX 文件可以包含任意多的十六進制記錄，每條記錄有五個域，每條記錄都由一個冒號":"打頭。一個數據記錄以一個回車和一個換行結束。其格式如下：

:CCAAAARR[DD...]ZZ

其中：

- CC：本條記錄中數據(dd)的字節數目
- AAAA：本條記錄中的數據在存儲區中的起始地址
- RR：記錄類型：
    - 00 數據記錄 (data record)
    - 01 文件結束記錄 (end record)
    - 02 擴展段地址記錄 (paragraph record)
    - 03 擴展線性地址記錄 (transfer address record)
- DD… ：數據域。表示一個字節的數據，一個記錄可能有多個數據字節，字節數目可以 查看 ll 域的說明
- ZZ：效驗和域，表示記錄的效驗和，計算方法是將本條記錄冒號開始的所有字母對所表示的十六進制數字都加起來然後模除 256 得到的餘數最後求出餘數的補碼即是本效驗字節 cc。

舉例如下：

:10400000781A00203D420008034C0008D14B0008FC

- 10： 長度 16

- 4000： 起始地址
- 00： 表示數據記錄
- 78 ~ 08: 數據
- FC：校驗和

參考

1. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2
2. ARM® Compiler v5.06 for µVision® Version 5 armlink User Guide
3. ARM® Compiler v5.06 for µVision® Version 5 armcc User Guide
4. ARM ELF File Format ARM DUI 00101-A
5. ARM ELF Development Systems Business Unit Engineering Software Group
6. ELF for the ARM® Architecture