ARM 之十 ARMCC（Keil） map 文件（映射文件）詳解

https://blog.csdn.net/ZCShouCSDN/article/details/100049644

在看這篇文章之前

1. 需要對 ARM ELF 文件有一定的瞭解。瞭解什麼是域（Region）、節（Section，也稱為節區）、段（Segment）、鏡像（Image）、鏡像文件（Image File）等概念
2. 需要對編譯、連接過程有一定的瞭解

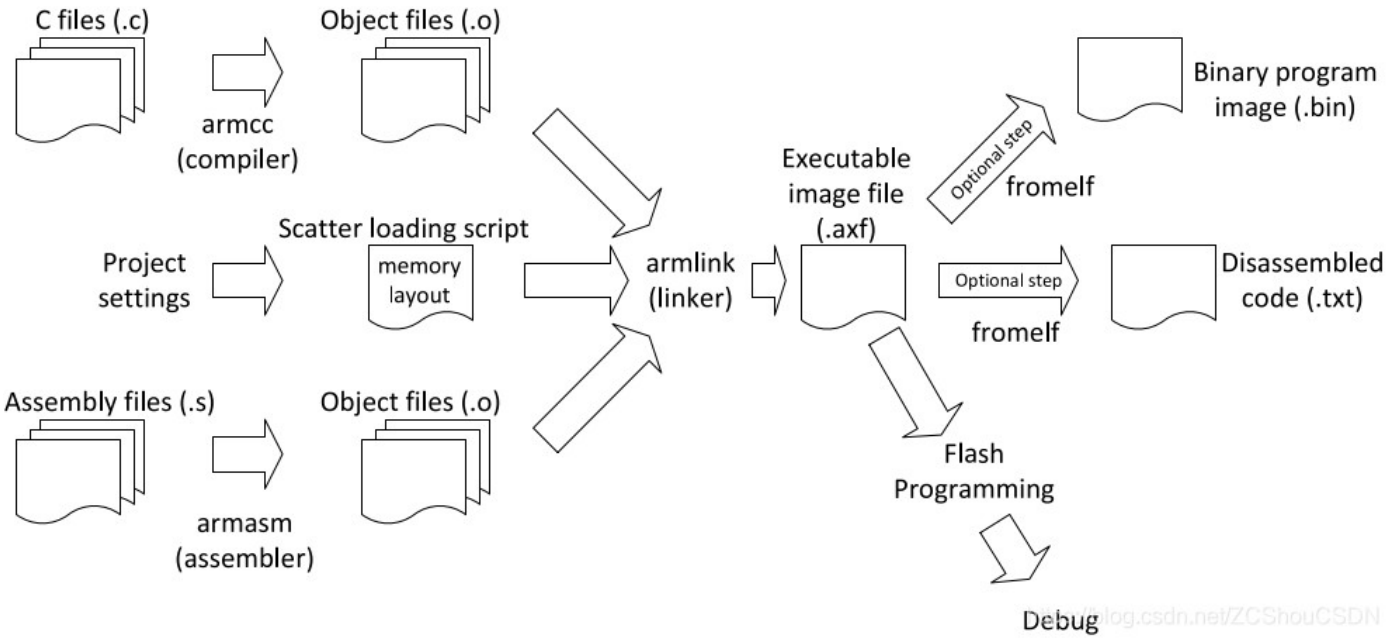
map 文件是什麼

map 文件對應的中文名應該是 **映射文件**，用來展示（映射）項目構建的鏈接階段的細節。通常包含程序的全局符號、交叉引用和內存映射等等信息。目前，大多數編譯套件（主要是其中的鏈接器）都可以生成 Map 文件。常見的 GCC、VC、IAR 都可以輸出 map 文件（PC 平台的 map 文件與 ARM 平台的差別較大）。

在 ARM 的官方文檔中，並沒有找到有關於 ARM 內核的 map 文件的介紹文檔。不過倒是有個 C51 生成的 map 文件的說明文檔：Listing (MAP) File。但是 C51 的 map 文件和 ARM 核的 map 文件差別比較大，也沒啥參考價值！

map 文件就是用來展示鏈接器工作過程的東西。想要瞭解 map 文件還需要對 ARM ELF 文件有一定的瞭解（可以參考博文 ARM 之一 ELF 文件、鏡像（Image）文件、可執行文件、對象文件 詳解）以及需要對於編譯過程有一定的瞭解。下圖顯示了鏈接器在軟件開發過程中的角色。鏈接器接受幾種類型的文件作為輸入，包括對象文件、命令文件、庫和部分鏈接的文件，

創建一個可執行對象模塊。



map 文件從哪來

map 文件是由編譯套件中的鏈接器產生的。ARM 的編譯套件中鏈接器為 armlink，map 文件中的各信息均由 armlink 的各參數（–info topic、–map、–symbols 等）控制輸出（由 --list=filename 文件名輸出到文件）。關於 ARM 編譯套件的詳細信息，參考博文 ARM 之七 主流編譯器（armcc、iar、gcc for arm）詳細介紹 中關於 ARM 鏈接器的詳細參數。下面是 armlink 的和 map 文件有關的參數介紹

--list=filename

將診斷輸出重定向到指定名字為 filename 文件，即輸出 filename.map 文件。

語法

--list=filename。其中 filename 是用於保存診斷輸出的文件。 文件名可以包含路徑。

使用

1. 將參數 --info， --map， --symbol， --verbose， --xref， --xreffrom 和 --xrefto 的診斷信息輸出重定向到文件。
2. 輸出診斷信息時將創建指定的文件。 如果已存在同名文件，則會覆蓋該文件。 但是，如果未輸出診斷，則不會創建文件。 在這種情況下，具有相同名稱的任何現有文件的內容保持不變。如果指定了 filename 而沒有路徑，則會放在與鏈接器生成的可執行文件的相同目錄下。

--info=topic[,topic,…]

打印有關指定主題的信息。通常與上面的 `--list=file` 一起使用，將輸出內容寫入指定的文本文件中。

語法

`--info=topic[,topic,…]`。其中 `topic` 是以下關鍵字以逗號分隔的組合（也可以每次只用一個關鍵字，然後寫多個 `--info=`關鍵字）：

- any：對於使用 `.ANY` 模塊選擇器的節區，列出：
  - 排序順序
  - 放置算法
  - The sections that are assigned to each execution region in the order they are assigned by the placement algorithm.
  - 有關每個域使用的應急空間和策略的信息。

  在分散加載文件中使用執行域屬性 `ANY_SIZE` 時，此關鍵字還會顯示其他信息。

- architecture：通過列出處理器，FPU 和字節順序來歸納鏡像文件架構。
- common：列出從鏡像文件中刪除的所有公共節區。使用這個選項意味著：`--info=common,totals`
- compression：提供有關 `RW` 壓縮過程的更多信息。
- debug：Lists all rejected input debug sections that are eliminated from the image as a result of using --remove. Using this option implies --info=debug,totals.
- exceptions：Gives information on exception table generation and optimization.
- inline：列出由鏈接器內聯的所有函數，並且如果使用了 `--inline`，則列出內聯的總數。
- inputs：列出輸入符號、對象和庫
- libraries：Lists the full path name of every library automatically selected for the link stage.You can use this option with --info_lib_prefix to display information about a specific library.
- merge：Lists the const strings that are merged by the linker. Each item lists the merged result, the strings being merged, and the associated object files.
- sizes：列出鏡像中每個輸入對象和庫成員的代碼和數據（RO 數據，RW 數據，ZI 數據和調試數據）大小。 使用此選項意味著 – info=sizes,totals。
- stack：列出所有函數的堆棧使用情況
- summarysizes：Summarizes the code and data sizes of the image.
- summarystack：Summarizes the stack usage of all global symbols.
- tailreorder：Lists all the tail calling sections that are moved above their targets, as a result of using --tailreorder.

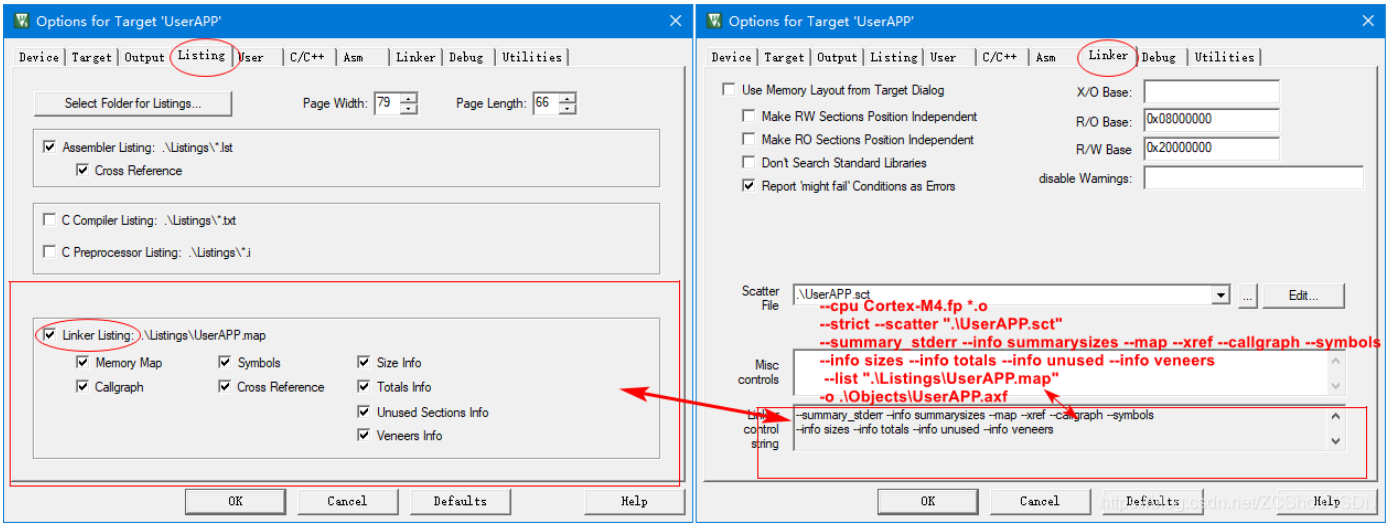
- totals：列出輸入對象和庫的代碼和數據（RO 數據，RW 數據，ZI 數據和調試數據）大小的總和。
- unused：列出由於使用 --remove 而從用戶代碼中刪除的所有未使用的部分。 它不會列出從 ARM C 庫加載的任何未使用的部分。
- unusedsymbols：Lists all symbols that have been removed by unused section elimination.
- veneers：列出鏈接器生成的膠合代碼。
- veneercallers：Lists the linker-generated veneers with additional information about the callers to each veneer. Use with --verbose to list each call individually.
- veneerpools：Displays information on how the linker has placed veneer pools.
- visibility：Lists the symbol visibility information. You can use this option with either --info=inputs or --verbose to enhance the output.
- weakrefs：Lists all symbols that are the target of weak references, and whether or not they were defined.

用法

1. --info=sizes,totals 的輸出始終包含輸入對象和庫的總計中的填充值。
2. 如果使用 RW 數據壓縮（默認值），或者使用 --datacompressor=id 選項指定了壓縮器，則 --info=sizes,totals 的輸出包括 Grand Totals 下的條目以反映真實鏡像大小。
3. 列表中的關鍵字之間不允許有空格。

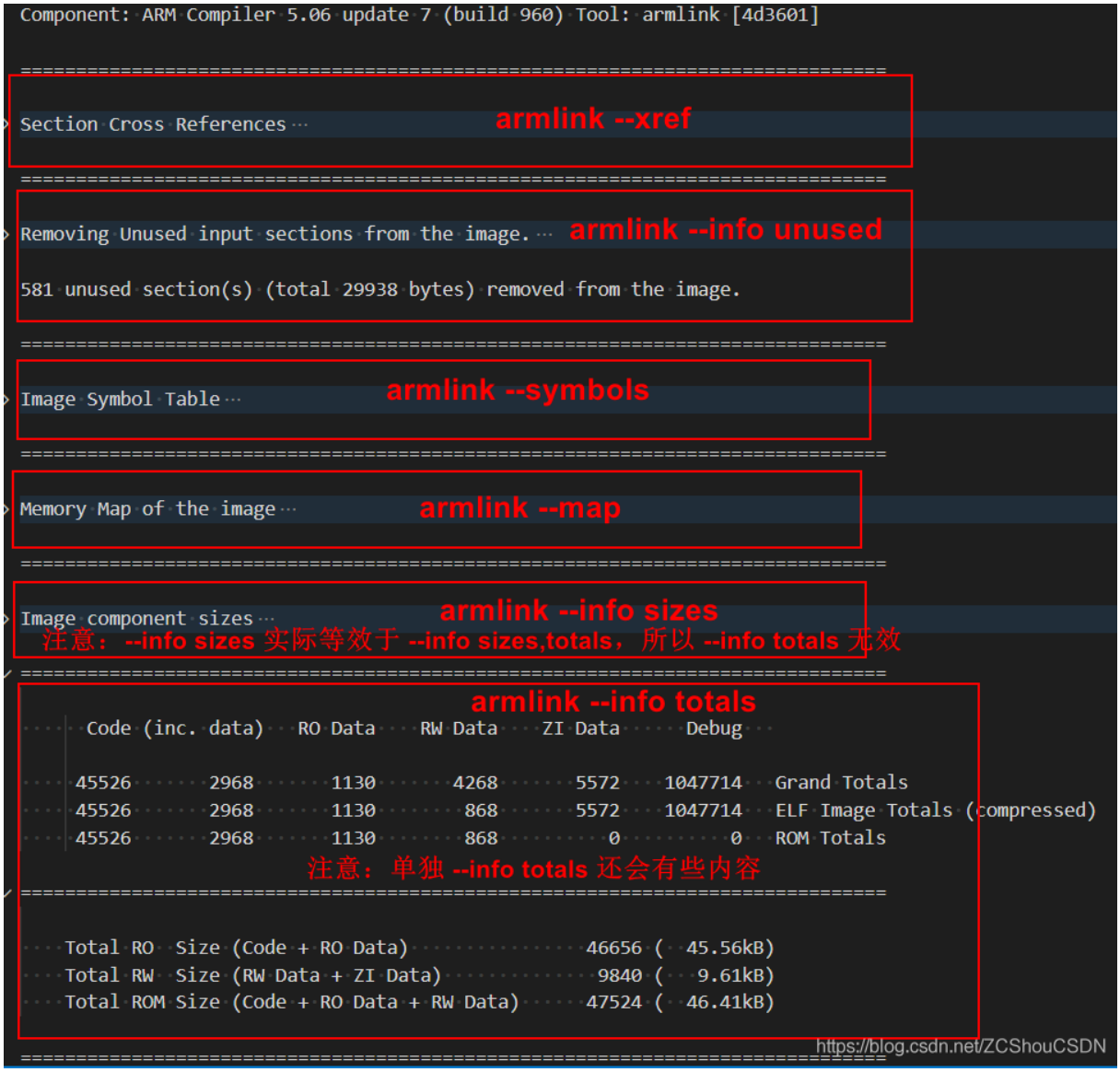
Keil 配置

鏈接器列表文件或 Map 文件包含有關鏈接/定位過程的大量信息。在 Keil 中，需要通過 Project -> Options for Target -> Listing 界面如下的配置才可以輸出 map 文件：



其中，各選項的基本功能如下：

- **Select Folder for Listings…**： 選擇存放清單文件的文件夾
- **Page Width**： 為清單文件指定每行字符數
- **Page Length**： 為清單文件指定每頁的行數
- **Assembler Listing**： 為匯編源文件創建列表文件，對應產生 **源文件名.lst** 的文件
    - **Cross Reference**： 列出有關符號的交叉引用信息，包括它們的定義位置以及宏的內部和外部的使用位置
- **C Compiler Listing**： 為 C 源文件創建列表文件，對應產生 **源文件名.txt** 的文件 和 **源文件名.lst** 的文件
- **C Preprocessor Listing**： 指示編譯器生成預處理文件。 宏調用將被展開並且注釋將被刪除 對應產生 **源文件名.i** 的文件
- **Linker Listing**： 讓鏈接器為目標項目創建映射文件（map 文件）。對應的 armlink 參數為 `--list=filename` ，如果不選擇則不會生成文件，對應生成 **用戶指定名.map** 的文件。生成的 MAP 文件如下圖所示：



通常需要配合以下參數一起使用：

- **Memory Map：** 包含一個內存映射，其中包含鏡像中每個加載區，執行區和輸入節的地址和大小，包括調試和鏈接器生成的輸入節。對應的 `armlink` 參數為 `--map`
- **Callgraph：** 以 HTML 格式創建函數的靜態調用圖文件。 調用圖給出了鏡像中所有函數的定義和參考信息。對應的 `armlink` 參數為 `--callgraph` 。該項會獨立生成一個 **配置的輸出名.htm** 的文件。如下圖所示：

Static Call Graph for image .\Objects\UserIAP.axf

#<CALLGRAPH># ARM Linker, 5060422: Last Updated: Thu Aug 15 08:42:16 2019

Maximum Stack Usage = 232 bytes + Unknown(Cycles, Untraceable Function Pointers)

Call chain for Maximum Stack Depth:

SX1276Process ⇒ SX1276FskProcess ⇒ __hardfp_round ⇒ __aeabi_dadd ⇒ _double_epilogue ⇒ _double_round

Mutually Recursive functions

- ADC_IRQHandler ⇒ ADC_IRQHandler

Function Pointers

- ADC_IRQHandler from startup_stm32f411xe.o(.text) referenced from startup_stm32f411xe.o(RESET)
- BusFault_Handler from stm32f4xx_it.o(i.BusFault_Handler) referenced from startup_stm32f411xe.o(RESET)
- DMA1_Stream0_IRQHandler from startup_stm32f411xe.o(.text) referenced from startup_stm32f411xe.o(RESET)

其中顯示了詳細的調用關系。最重要的是，其中還有使用的棧的大小！

- **Symbols：** 列出本地，全局和鏈接器生成的符號以及符號值。對應的 armlink 參數為 --symbols。注意：該參數不包含映射符號，下文我們會詳細介紹！
- **Cross Reference：** 列出輸入節之間的所有交叉引用。對應的 armlink 參數為 --xref
- **Size Info：** 給出鏡像中每個輸入對象和庫成員的代碼和數據（RO 數據，RW 數據，ZI 數據和調試數據）的大小的列表。對應的 armlink 參數為 --info sizes
- **Totals Info：** 提供輸入對象和庫的代碼和數據（RO 數據，RW 數據，ZI 數據和調試數據）大小的總和。對應的 armlink 參數為 --info totals
- **Unused Section Info：** 列出從鏡像文件中刪除的所有未使用的部分。對應的 armlink 參數為 --info unused
- **Veneers Info：** 提供鏈接器生成的 Thumb/ARM 膠合代碼的詳細信息。對應的 armlink 參數為 --info veneers

map 文件有啥用

map 文件對於分析問題是非常有用的！

1. 分析問題
2. 優化程序
3. 學習瞭解連接過程

map 文件中的符號

在 map 文件中，有很多符號是編譯套件的開發商預定義好的，用戶的符號不能與編譯套件的開發商預定義好的符號沖突。以下內容來自於 ARM 的鏈接器手冊！關於如何導入這些符號鏈接器的手冊有專門的章節來介紹！

映射符號

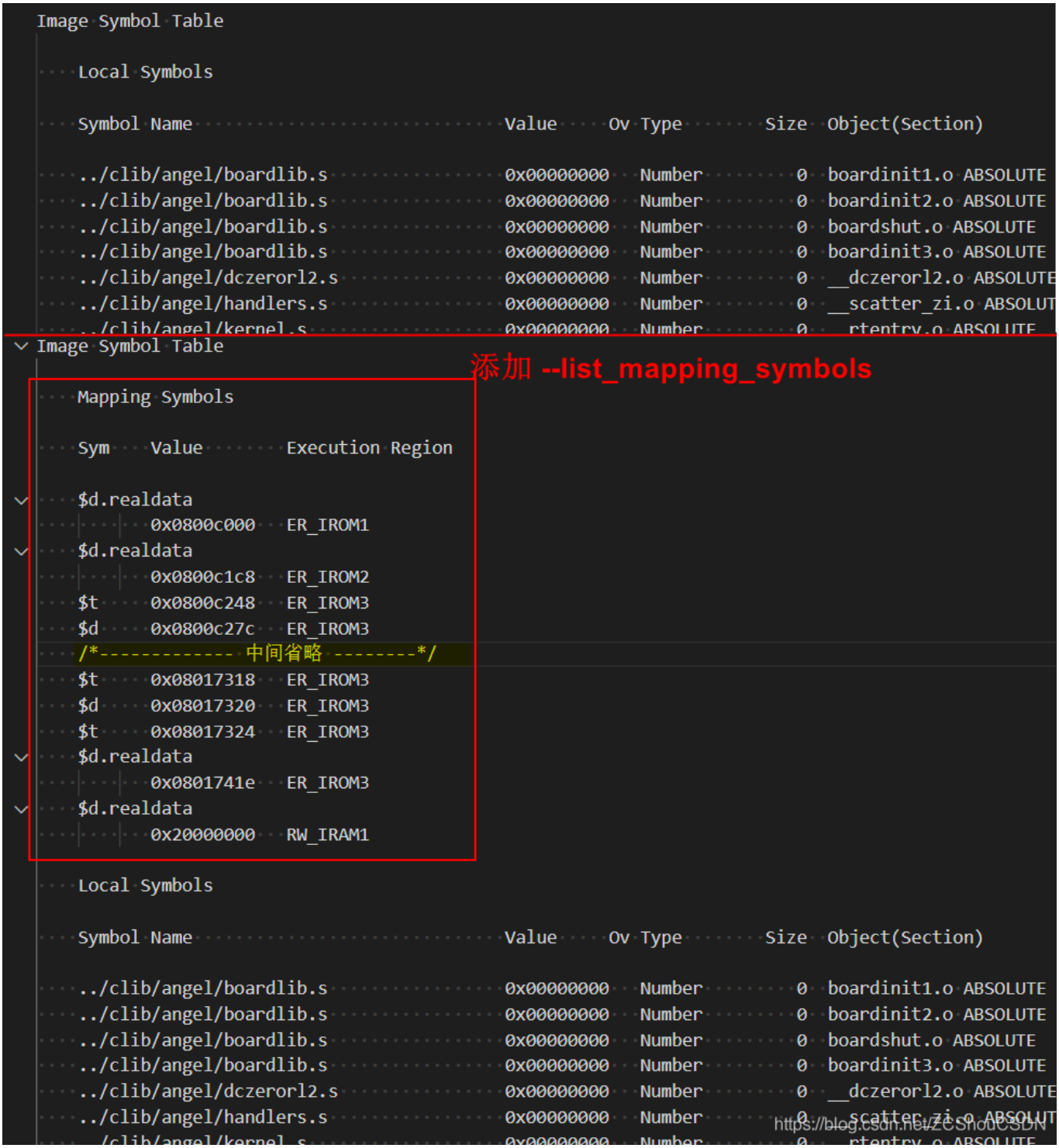
映射符號由編譯器和匯編器生成，以識別文字池邊界處的代碼和數據之間的內聯轉換，以及 ARM 代碼和 Thumb 代碼之間的內聯轉換。例如 ARM/Thumb 交互操作膠合代碼。映射符號有如下這些：

- $a：一系列 ARM 指令的開始
- $t：一系列 Thumb 指令的開始
- $t.x：一系列 ThumbEE 指令的開始
- $d：一系列數據項的開始，如文字池

1. **文字池** 是代碼段中存放常量數據的區域。因為沒有一條指令可以生成一個 4 字節的常量，因此編譯器將這些常量放到文字池中，然後生成從文字池加載這些常量的代碼。
2. ARM/Thumb 交互（ARM/Thumb interworking）是指對匯編語言和 C/C++ 語言的 ARM 和 Thumb 代碼進行連接的方法，它進行兩種狀態（ARM 和 Thumb）間的切換。
3. 膠合代碼（Veneer）：在進行 ARM/Thumb 交互時，有時需使用額外的代碼，這些代碼被稱為 膠合代碼（Veneer）。
4. AAPCS 定義了 ARM 和 Thumb 過程調用的標准。

armlink 會生成 $d.realdata 映射符號，以告訴 fromelf 該數據是來自非可執行節區。因此，fromelf -z 輸出的代碼和數據大小與 armlink --info sizes 的輸出相同。例如：

在以上的示例中，y 標記為 $d，RO Data 標記為 $d.realdata。如果啟用了 --list_mapping_symbols，則會在 map 文件中有體現，如下圖：



請注意：

1. 以字符 `$v` 開頭的符號是與 VFP 相關的映射符號，在使用 VFP 構建目標時可能會輸出。 避免在源代碼中使用以 `$v` 開頭的符號。
2. 使用 `fromelf --elf --strip=localsymbols` 命令修改可執行鏡像會從鏡像中刪除所有映射符號。
3. 其必須由 `armlink` 的參數 `--list_mapping_symbols` 和 `--no_list_mapping_symbols` 分別來控制顯示與不顯示。在默認情況下為 `--no_list_mapping_symbols`，即不顯示這部分符號。

鏈接器定義的符號

當鏈接器創建鏡像文件時，它會創建一些 ARM 預定義的與域或者節相關的符號。這些符號就代表了鏈接器創建創建鏡像的依據。下面我們就重點來瞭解一下這些符號。

鏈接器定義了一些 ARM 保留的符號，我們可以在需要時訪問這些符號。 ***這些符號是包含 $$ 字符序列的符號以及所有其他包含 $$ 字符序列的外部名稱。*** 您可以導入這些符號地址，並將它們作為匯編語言程序的可重定位地址使用，或者將它們作為 C 或 C++ 源代碼中的 extern 符號來引用。

- 如果使用 --strict 編譯器命令行選項，則編譯器不接受包含 $ 的符號名稱。 要重新啟用支持，請在編譯器命令行中包含 --dollar 選項。
- ***鏈接器定義的符號只有在代碼引用它們時才會生成。***
- 如果存在僅執行（XO）節，則鏈接器定義的符號受以下約束：
    - 不能對沒有 XO 節的域或者空域定義 XO 連接器定義符號
    - 不能對僅包含 RO 節的域定義 XO 連接器定義符號
    - 對於僅包含 XO 節的域，不能定義 RO 連接器定義符號

引入到 C/C++

可以通過 **值引用** 或 **地址引用** 這兩種方式將鏈接器定義的符號導入到的 C 或 C++ 源代碼中來供我們使用：

- 值引用：extern unsigned int symbol_name;
- 地址引用：extern void *symbol_name;

注意，如果將符號聲明為 int 類型的值引用，則必須使用尋址操作符（&）來獲得正確的值，如下例所示：

引入到 匯編

可以使用指令 IMPORT 將連接器定義的符號引入到 ARM 匯編文件中來供我們使用：

域相關的符號

鏈接器為鏡像文件中的每個域生成不同類型的與域相關的符號，我們可以根據需要訪問這些符號。域相關的符號主要有以下兩種：

- Image$$ 或者 Load$$ 開頭的符號，用於各執行域
- Load$$LR$$ 開頭的符號，用於各加載域

如果未使用分散加載文件，則會以默認的 region 名稱來生成域相關的符號。鏈接器默認的域名稱如下：

- **ER_XO** ： 用於僅執行屬性的執行域（如果存在）。
- **ER_RO** ： 用於只讀執行域。
- **ER_RW** ： 用於可讀寫執行域。
- **ER_ZI** ： 用於零初始化的執行域。

  可以將這些名稱插入 **Image$$** 和 **Load$$** 中以獲取所需的地址，例如：**Load$$ER_RO$$Base** 就是只讀域的基地址。

使用分散加載時，連接器將使用分散加載文件中的名稱來生成各種域相關的符號。分散加載文件可以實現以下功能：

- 命名鏡像中的所有執行域，並提供他們的加載和執行地址。
- 定義堆棧和堆。 鏈接器還會生成特殊的棧和堆符號。

1. 鏡像的 **ZI** 輸出節不是靜態創建的，而是在運行時自動動態創建的。 因此，**ZI** 輸出節沒有加載地址符號。
2. 符號 **Load$$region_name** 僅適用於執行域。**Load$$LR$$load_region_name** 符號僅適用於加載域。

執行域符號 Image$$

鏈接器為鏡像中存在的每個執行域生成符號 **Image$$**。下表列出了鏈接器為鏡像中存在的每個執行域生成的符號。 初始化 C 庫後，所有符號都指向執行地址。

Symbol	Description
Image$$region_name$$Base	域的執行地址
Image$$region_name$$Length	執行域長度（以字節為單位），不包括 ZI 的長度。
Image$$region_name$$Limit	超出執行域中非 ZI 部分末尾的字節的地址
Image$$region_name$$RO$$Base	域中的輸出節 RO 的執行地址
Image$$region_name$$RO$$Length	Length of the RO output section in bytes.
Image$$region_name$$RO$$Limit	Address of the byte beyond the end of the RO output section in the execution region.
Image$$region_name$$RW$$Base	Execution address of the RW output section in this region.
Image$$region_name$$RW$$Length	Length of the RW output section in bytes.
Image$$region_name$$RW$$Limit	Address of the byte beyond the end of the RW output section in the execution region.
Image$$region_name$$XO$$Base	Execution address of the XO output section in this region.
Image$$region_name$$XO$$Length	Length of the XO output section in bytes.
Image$$region_name$$XO$$Limit	Address of the byte beyond the end of the

Symbol	Description
	XO output section in the execution region.
Image$$region_name$$ZI$$Base	Execution address of the ZI output section in this region.
Image$$region_name$$ZI$$Length	Length of the ZI output section in bytes.
Image$$region_name$$ZI$$Limit	Address of the byte beyond the end of the ZI output section in the execution region.

執行域符號 Load$$

鏈接器為鏡像中存在的每個執行域生成符號 Load$$。下表列出了鏈接器為鏡像中存在的每個 Load$$ 執行域生成的符號。 初始化 C 庫後，所有符號都指向加載地址。

Symbol	Description
Load$$region_name$$Base	Load address of the region.
Load$$region_name$$Length	Region length in bytes.
Load$$region_name$$Limit	Address of the byte beyond the end of the execution region.
Load$$region_name$$RO$$Base	Address of the RO output section in this execution region.
Load$$region_name$$RO$$Length	Length of the RO output section in bytes.
Load$$region_name$$RO$$Limit	Address of the byte beyond the end of the RO output section in the execution region.
Load$$region_name$$RW$$Base	Address of the RW output section in this execution region.
Load$$region_name$$RW$$Length	Length of the RW output section in bytes.
Load$$region_name$$RW$$Limit	Address of the byte beyond the end of the RW output section in the execution region.
Load$$region_name$$XO$$Base	Address of the XO output section in this execution region.
Load$$region_name$$XO$$Length	Length of the XO output section in bytes.
Load$$region_name$$XO$$Limit	Address of the byte beyond the end of the XO output section in the execution region.
Load$$region_name$$ZI$$Base	Load address of the ZI output section in this execution region.
Load$$region_name$$ZI$$Length	Load length of the ZI output section in bytes. The Load Length of ZI is zero unless region_name has the ZEROPAD scatter-loading keyword set. If ZEROPAD is set then:

Symbol	Description
	Load Length = Image$$region_name$$ZI$$Length
Load$$region_name$$ZI$$Limit	Load address of the byte beyond the end of the ZI output section in the execution region.

初始化 C 庫之前，此表中的所有符號均指加載地址。請注意以下事項：

- 這些符號是絕對的，因為相對於節的符號只能有執行地址。
- 這些符號考慮了 RW 壓縮。
- 從 RW 壓縮執行域引用的鏈接器定義的符號必須是在應用 RW 壓縮之前可解析的符號。
- 如果鏈接器檢測到從 RW 壓縮域到依賴於 RW 壓縮的鏈接器定義符號的重定位，則鏈接器將禁用當前域的壓縮。
- Limit 和 Length 值影響寫入文件的任何零初始化數據。 使用 ZEROPAD 分散加載關鍵字時，零初始化數據將寫入文件。

加載域符號 Load$$LR$$

鏈接器為鏡像中存在的每個加載區生成符號 Load$$LR$$。一個 Load$$LR$$ 加載域可以包含許多執行域，因此沒有單獨的 $$RO 和 $$RW 部分。下表顯示了鏈接器為鏡像中存在的每個 Load$$LR$$ 加載域生成的符號。

Symbol	Description
Load$$LR$$load_region_name$$Base	Address of the load region.
Load$$LR$$load_region_name$$Length	Length of the load region.
Load$$LR$$load_region_name$$Limit	Address of the byte beyond the end of the load region.

節相關的符號

與節相關的符號是鏈接器在創建沒有使用分散加載文件的鏡像時生成的符號。鏈接器會為輸出和輸入節生成不同類型的與節相關的符號：

- **鏡像符號（Image symbols）**（如果不使用分散加載來創建簡單的鏡像文件）。 簡單的鏡像文件具有多達四個輸出節（XO，RO，RW 和 ZI），用於生成相應的執行域。
- **輸入節符號（Input section symbols）** 鏡像中存在的每個輸入節的輸入節符號（Input section symbols）

鏈接器首先按屬性 RO，RW 或 ZI 對執行域內的節進行排序，然後按名稱排序。 例如，所有 .text 節都放在一個連續的塊中。 具有相同屬性和名稱的連續塊部分稱為合並節。

1. ARM 建議優先使用與域相關的符號，而不是與節相關的符號。

鏡像符號

當您不使用分散加載文件來創建簡單鏡像時，鏡像符號將由鏈接器生成。我們常用的 Keil 會默認生成分散加載文件的，所以基本沒有不使用分散加載文件的情況。下表顯示了鏡像符號：

Symbol	Section type	Description
Image$$RO$$Base	Output	Address of the start of the RO output section.
Image$$RO$$Limit	Output	Address of the first byte beyond the end of the RO output section.
Image$$RW$$Base	Output	Address of the start of the RW output section.
Image$$RW$$Limit	Output	Address of the byte beyond the end of the ZI output section. (The choice of the end of the ZI region rather than the end of the RW region is to maintain compatibility with legacy code.)
Image$$ZI$$Base	Output	Address of the start of the ZI output section.
Image$$ZI$$Limit	Output	Address of the byte beyond the end of the ZI output section.

如果存在 XO 節，那麼還包含符號 Image$$XO$$Base 和 Image$$XO$$Limit

如果使用了分散加載文件，則上面這些鏡像符號都將稱為未定義的。 如果在代碼中訪問這些符號中的任何一個，則必須將它們視為 **弱引用**。__user_setup_stackheap() 的標准實現中就使用 Image$$ZI$$Limit 中的值，因此，如果您使用的是分散加載文件，則必須手動設置堆棧和堆。 方法主要有以下兩種：

- 在分散文件中使用下列方法之一
  - 定義名為 ARM_LIB_STACK 和 ARM_LIB_HEAP 的單獨的棧和單獨的堆域。
  - 定義包含堆棧和堆的組合域，名為 ARM_LIB_STACKHEAP。
- 通過重新實現 __user_setup_stackheap() 來設置堆和堆棧邊界。（在我們的項目中的 .s 啟動文件中，是這種方法）

具體見博文 [ARM 之十三 armlink（Keil） 分散加載機制詳解 及 分散加載文件的編寫]

輸入節符號

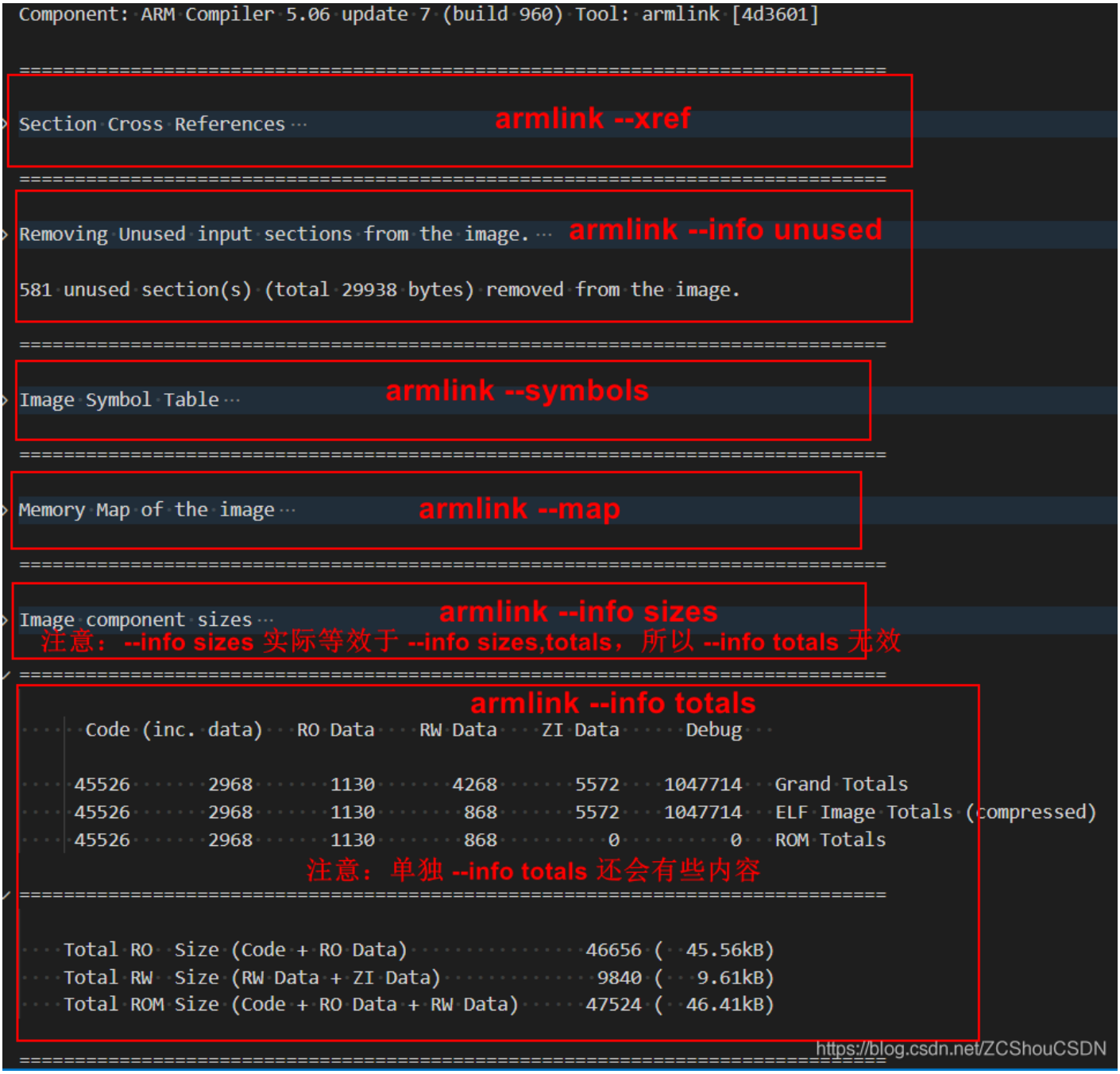
鏈接器為鏡像中存在的每個輸入節生成輸入節符號。下表顯示了鏈接器定義的輸入節符號：

Symbol	Section type	Description
SectionName$$Base	Input	Address of the start of the consolidated section called SectionName.
SectionName$$Length	Input	Length of the consolidated section called SectionName (in bytes).
SectionName$$Limit	Input	Address of the byte beyond the end of the consolidated section called SectionName.

如果在的代碼引用輸入節符號，則表示希望將鏡像中具有相同名稱的所有輸入節都連續放置在鏡像內存映射中。如果分散加載文件不連續地放置輸入節，則鏈接器會發出錯誤。 這是因為在非連續存儲器上將導致 Base 符號和 Limit 符號是不明確的。

map 文件示例

根據選擇的參數不同，map 文件中的內容肯定是有變化的！下面以 Keil 中全選以上所說的參數後生成的 map 文件為例來進行說明。map 文件中，有如下圖所示的六個大部分：



其中，有些符號需要我們知道其含義：

- `.data`：這些部分保存有助於程序內存映像的已初始化數據
- `.text`：本節包含程序的文本或可執行指令
- `.bss`：本節保存有助於程序內存映像的未初始化數據
- `.ARM.exidx*`：以`.ARM.exidx`開頭的節包含部分展開的索引條目
- `i.xxxx`： `i` 是 `interface` 的意思，`i.xxxx` 就表示 `xxx` 接口（一般就是指函數名）

更信息的請參考博文 [ARM 之一 ELF 文件、鏡像（Image）文件、可執行文件、對象文件 詳解]

Section Cross References

該部分顯示了節區之間的交叉引用，指的是各個源文件生成的模塊之間相互引用的關系，是由 armlink 的參數 --xref 生成的。主要分為以下幾種情況：

- **用戶代碼間接口的相互引用：** 例如：stm32f4xx_ll_flash.o(i.LL_FLASH_EraseChip) refers to stm32f4xx_ll_flash.o(i.LL_FLASH_IsActiveFlag_BSY) for LL_FLASH_IsActiveFlag_BSY 表示 stm32f4xx_ll_flash.o 中的函數 LL_FLASH_EraseChip 引用了 stm32f4xx_ll_flash.o 中的 函數 LL_FLASH_IsActiveFlag_BSY，其中的 stm32f4xx_ll_flash.o 是由用戶代碼 stm32f4xx_ll_flash.c 生成的模塊
- **用戶接口引用用戶數據**：主要有兩類，例如：lora.o(i.LoRaRcvProcess) refers to lora.o(.data) for Radio 表示 lora.o 中的接口 LoRaRcvProcess 引用了 lora.o 中的已初始化數據 Radio；lora.o(i.LoRaRcvProcess) refers to lora.o(.bss) for ProcLoRa 表示 lora.o 中的接口 LoRaRcvProcess 引用了 lora.o 中的未初始化默認初始化為 0 數據 ProcLoRa
- **用戶數據引用接口：** 這個要是函數指針的使用。例如：pnwz.o(.data) refers to pnwzuif.o(i.PNWZ_USER_RespGetLife) for PNWZ_USER_RespGetLife
- **用戶接口引用 C 庫接口：** 例如：pnwzuif.o(i.PNWZ_USER_ReqGetVerHW) refers to strlen.o(.text) for strlen
- **代碼引用接口：** 主要發生於 C 庫之間（ARM C 庫是不開源的，僅提供二進制文件），例如：``
- **C 庫接口之間的引用：** 主要發生於 C 庫之間（ARM C 庫是不開源的，僅提供二進制文件），例如：__2sprintf.o(.text) refers to _sputc.o(.text) for _sputc 及 __printf_flags_ss_wp.o(.text) refers to __printf_wp.o(i._is_digit) for _is_digit
- **C 庫引用連接器符號：** 例如：exit.o(.text) refers to rtexit.o(.ARM.Collect$$rtexit$$00000000) for __rt_exit
- **符號之間的引用：** 例如：retnan.o(xfplretnan) refers to trapv.o(xfpltrapveneer) for __fpl_cmpreturn

在 ARM 編譯套件中，所有的 C 庫由工具 armar 來管理，位於 ARM 編譯器目錄 lib 下。使用 armar 即可從指定的庫文件中解壓出 __main.o 等模塊。至於如何操作，參見博文 ARM 之九 Cortex-M/R 內核啟動過程 / 程序啟動流程（基於 ARMCC、Keil）。

Removing Unused input sections from the image

這部分列出了鏈接器移除的我們源碼中實際未使用的數據和函數。其中包含移除數據的大小。例如 Removing flash.o(.rrx_text), (6 bytes). 表示移除 flash.o 中的 6 字節的代碼；Removing virtualuart.o(i.VirtualUartBufClear), (92 bytes). 表示移除 virtualuart.o 中的函數 VirtualUartBufClear，共 92 字節。
需要注意的是，被移除的函數在調試時將無法進行調試。如果不注意，在調試時很容易造成

困擾。如下圖所示：

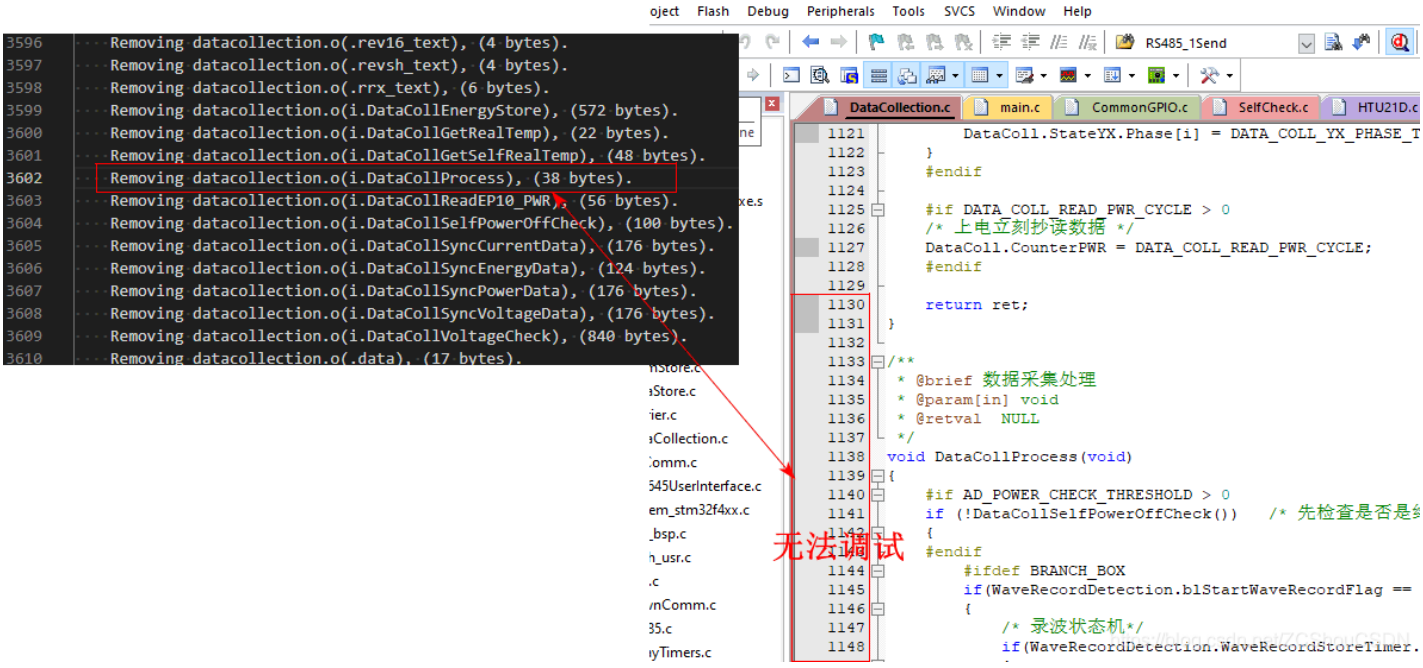
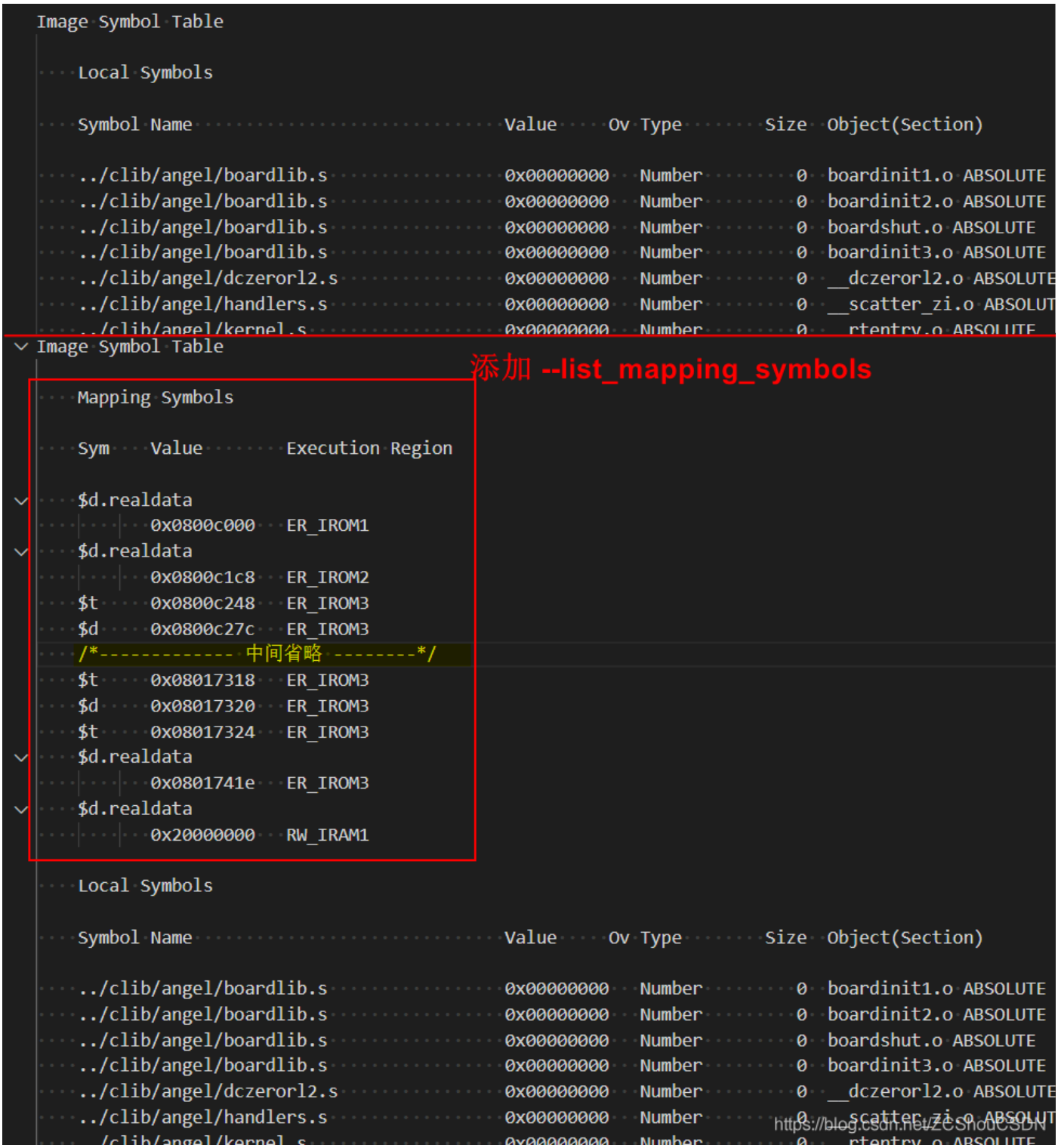


Image Symbol Table

鏡像符號映射表。就是每個符號（這裡的符號可以指模塊、變量、函數）實際的地址等信息。分為以下三部分：

Mapping Symbols

這一部分只有在指定了連接器參數 `--list_mapping_symbols` 才會有！下面是一個對比：



各列的含義如下：

- **Sym**： 連接器定義的各種符號
- **Value**： 符號表示的地址
- **Execution Region**： 符號所在的執行域

Local Symbols

不知道 ARMCC 是怎麼分的 Local 和 Global。各列含義如下：

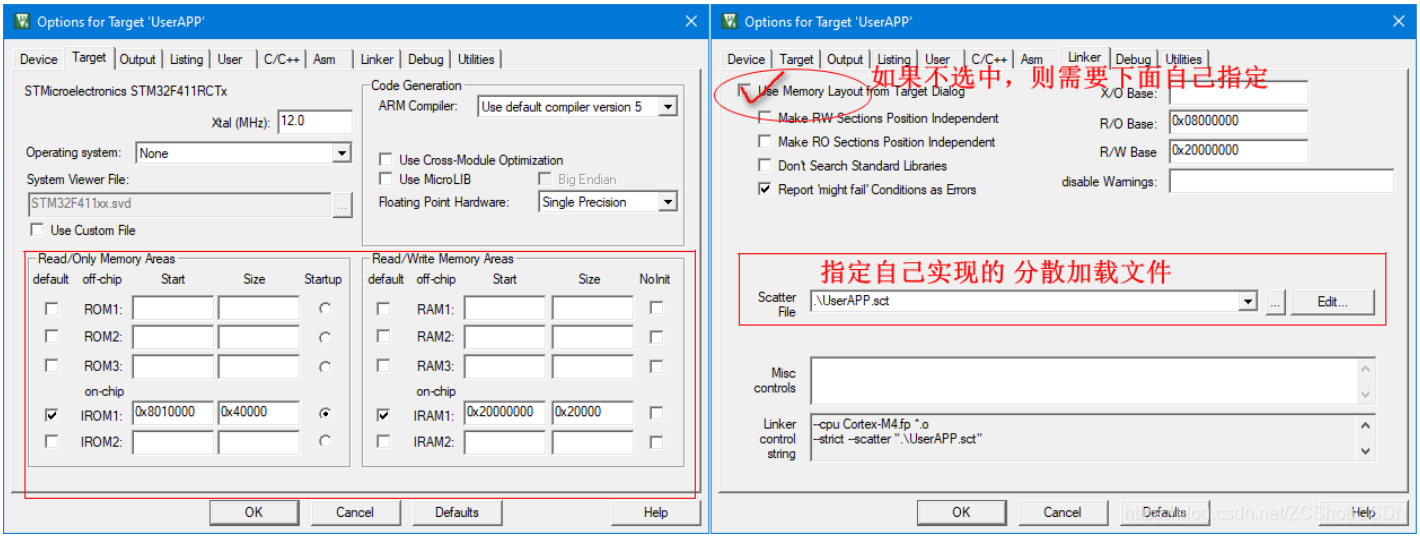
- Symbol Name
- Value
- Ov Type
- Size
- Object(Section)

Global Symbols

不知道 ARMCC 是怎麼分的 Local 和 Global。

Memory Map of the image

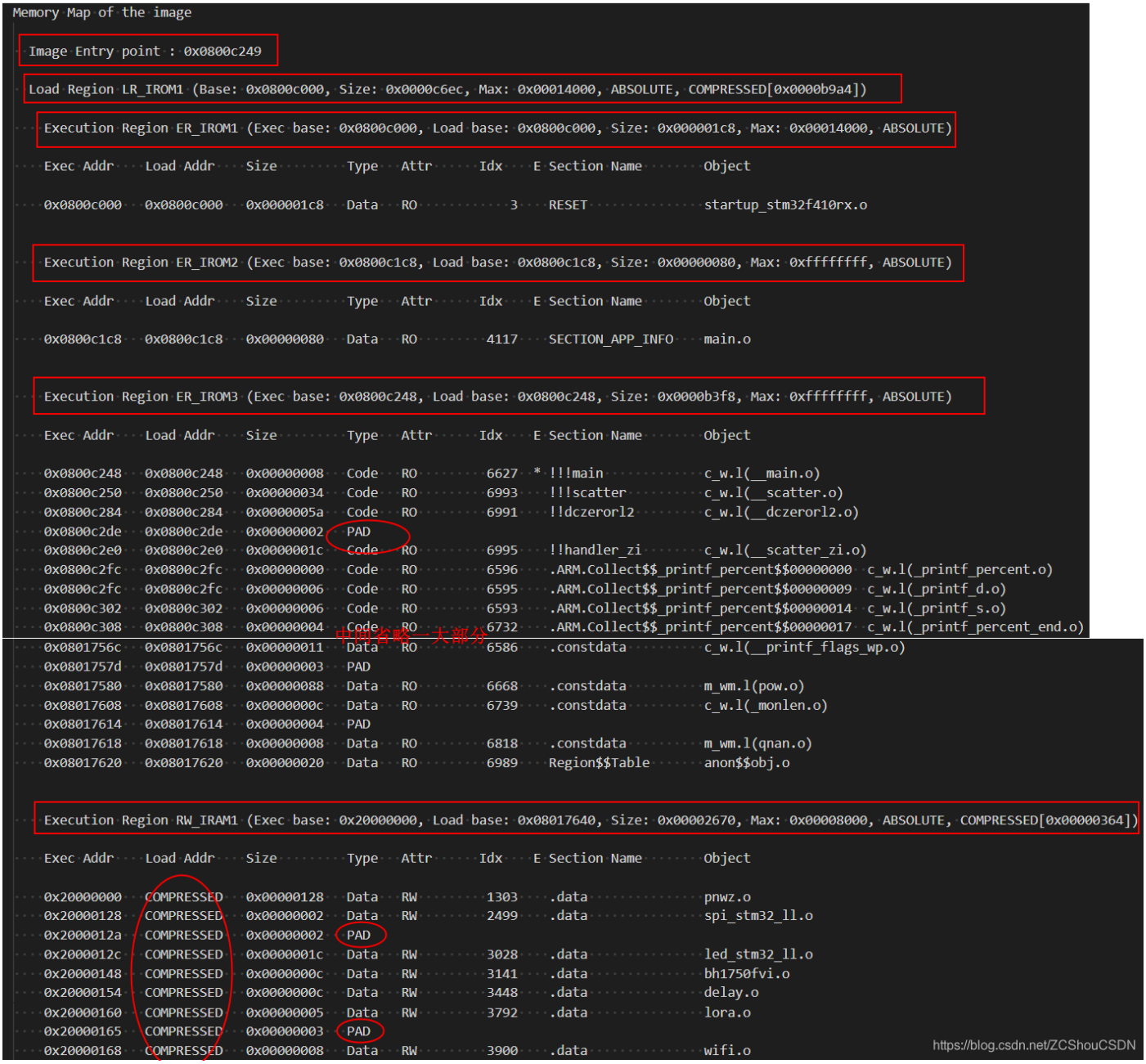
該映射包含鏡像文件中每個加載域，執行域和輸入節（包括連接器生成的輸入節）的地址和大小。由連接器 armlink 通過參數 --map 生成。這部分與 **分散加載文件（Scatter File）** 有密切的關系。如果使用 Keil，則在 Keil 的配置界面中有如下配置：



這裡的設置就是對應的分散加載文件中的內容。如果我們在鏈接器的配置頁面不選擇 Use Memory Layout from Taget Dialog，則需要如上圖自己指定一個分散加載文件。其實，如果選擇 Use Memory Layout from Taget Dialog，**Keil 會根據我們左邊的配置自行生成一個分散加載文件來給鏈接器使用（這個文件就在我們的編譯輸出指定的目錄中）**。上圖的示例就是沒有使用 Keil 默認，而是通過自己指定的分散加載文件生成鏡像文件。

以上的 Keil 配置，最終是通過鏈接器的參數 --scatter=filename 來讓鏈接器使用該文件的。分散加載文件一次性描述了我們的鏡像文件怎麼佈局。瞭解鏈接器的應該知道，鏈接器還有一些獨立使用的和鏡像文件生成有關的參數：--first, --last, --partial, --reloc, --ro_base, --ropi,--rosplit, --rw_base, --rwpi, --split, --startup, --xo_base, and --zi_base.，如果使用了 --scatter=filename，則以上參數就不可再用了！ Keil 就是直接使用的 --scatter=filename。（默認下，Keil 根據配置界面的配置，會成一個分散加載文件）。

接下來以一個示例來看看 `map` 文件中關於鏡像內存映射的內容，如下圖：



```
Memory Map of the image

  Image Entry point : 0x0800c249

  Load Region LR_IROM1 (Base: 0x0800c000, Size: 0x0000c6ec, Max: 0x00014000, ABSOLUTE, COMPRESSED[0x0000b9a4])

    Execution Region ER_IROM1 (Exec base: 0x0800c000, Load base: 0x0800c000, Size: 0x000001c8, Max: 0x00014000, ABSOLUTE)

    Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x0800c000   0x0800c000   0x000001c8   Data   RO            3    RESET               startup_stm32f410rx.o


    Execution Region ER_IROM2 (Exec base: 0x0800c1c8, Load base: 0x0800c1c8, Size: 0x00000080, Max: 0xffffffff, ABSOLUTE)

    Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x0800c1c8   0x0800c1c8   0x00000080   Data   RO         4117    SECTION_APP_INFO    main.o


    Execution Region ER_IROM3 (Exec base: 0x0800c248, Load base: 0x0800c248, Size: 0x0000b3f8, Max: 0xffffffff, ABSOLUTE)

    Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x0800c248   0x0800c248   0x00000008   Code   RO         6627  * !!!main             c_w.l(__main.o)
    0x0800c250   0x0800c250   0x00000034   Code   RO         6993    !!!scatter          c_w.l(__scatter.o)
    0x0800c284   0x0800c284   0x0000005a   Code   RO         6991    !!dczerorl2         c_w.l(__dczerorl2.o)
    0x0800c2de   0x0800c2de   0x00000002   PAD
    0x0800c2e0   0x0800c2e0   0x0000001c   Code   RO         6995    !!handler_zi        c_w.l(__scatter_zi.o)
    0x0800c2fc   0x0800c2fc   0x00000000   Code   RO         6596    .ARM.Collect$$_printf_percent$$00000000  c_w.l(_printf_percent.o)
    0x0800c2fc   0x0800c2fc   0x00000006   Code   RO         6595    .ARM.Collect$$_printf_percent$$00000009  c_w.l(_printf_d.o)
    0x0800c302   0x0800c302   0x00000006   Code   RO         6593    .ARM.Collect$$_printf_percent$$00000014  c_w.l(_printf_s.o)
    0x0800c308   0x0800c308   0x00000004   Code   RO         6732    .ARM.Collect$$_printf_percent$$00000017  c_w.l(_printf_percent_end.o)
    0x0801756c   0x0801756c   0x00000011   Data   RO         6586    .constdata          c_w.l(__printf_flags_wp.o)
    0x0801757d   0x0801757d   0x00000003   PAD
    0x08017580   0x08017580   0x00000088   Data   RO         6668    .constdata          m_wm.l(pow.o)
    0x08017608   0x08017608   0x0000000c   Data   RO         6739    .constdata          c_w.l(_monlen.o)
    0x08017614   0x08017614   0x00000004   PAD
    0x08017618   0x08017618   0x00000008   Data   RO         6818    .constdata          m_wm.l(qnan.o)
    0x08017620   0x08017620   0x00000020   Data   RO         6989    Region$$Table       anon$$obj.o


    Execution Region RW_IRAM1 (Exec base: 0x20000000, Load base: 0x08017640, Size: 0x00002670, Max: 0x00008000, ABSOLUTE, COMPRESSED[0x00000364])

    Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x20000000   COMPRESSED   0x00000128   Data   RW         1303    .data               pnwz.o
    0x20000128   COMPRESSED   0x00000002   Data   RW         2499    .data               spi_stm32_ll.o
    0x2000012a   COMPRESSED   0x00000002   PAD
    0x2000012c   COMPRESSED   0x0000001c   Data   RW         3028    .data               led_stm32_ll.o
    0x20000148   COMPRESSED   0x0000000c   Data   RW         3141    .data               bh1750fvi.o
    0x20000154   COMPRESSED   0x0000000c   Data   RW         3448    .data               delay.o
    0x20000160   COMPRESSED   0x00000005   Data   RW         3792    .data               lora.o
    0x20000165   COMPRESSED   0x00000003   PAD
    0x20000168   COMPRESSED   0x00000008   Data   RW         3900    .data               wifi.o
```

中間省略一大部分

- **Image Entry point：** 這個是鏡像的入口點。就是鏡像在被執行時，開始的位置（地址）。
- **Load Region LR_IROM1：** 表示一個叫做 `LR_IROM1` 加載域
  - **Base：0x0800c000** 這個表示 `LR_IROM1` 的基地址
  - **Size：0x0000c6ec** 表示 `LR_IROM1` 的大小為 `0x0000c6ec`，單位字節。
  - **Max：0x00014000** 表示 `LR_IROM1` 的最大大小，單位字節。這個是由分散加載文件中指定，如果不顯示指出，默認為 `0xFFFFFFFF`
  - **ABSOLUTE** 表示地址為絕對地址
  - **COMPRESSED[0x0000b9a4]** 表示壓縮之後的大小為 `0x0000b9a4`，單位字節。

- **Execution Region ER_IROM1、Execution Region ER_IROM2、Execution Region ER_IROM3：** 這是 3 個執行域，分別叫做 ER_IROM1、ER_IROM2、ER_IROM3。下面以 ER_IROM1 為例，來說說每個列（字段）的含義：
  - **Exec base：0x0800c000** 這個表示 ER_IROM1 執行時的基地址
  - **Load base：0x0800c000** 該執行域對應的加載域地址是 0x0800c000
  - **Size: 0x000001c8** 表示該執行域的大小為 0x000001c8，單位字節
  - **Max: 0x00014000** 表示該執行域的最大大小為 0x00014000，單位字節。這個是由分散加載文件中指定，如果不顯示指出，默認為 0xFFFFFFFF
  - **ABSOLUTE** 表示地址為絕對地址

  為什麼有 3 個？因為我本身使用了自己寫的分散加載文件。手動指定了三個執行域。我的分散加載文件如下

- 在這三個執行域中，有很多類型為 PAD 的行，並且這些行沒有節名字也沒有所屬的模塊。這些其實是一些鏈接器自己添加的對齊。除了對齊沒有其他作用。關於對齊本文之前的章節有介紹。除了 PAD 之外，剩下的就全是 Code 了。
- **Execution Region RW_IRAM1：** 一個名為 RW_IRAM1 的執行域。括號中的內容含義與上面的相同

Exec Addr	Load Addr	Size	Type	Attr	Idx	E	Section Name	Object
0x20000000	COMPRESSED	0x00000128	Data	RW	1303		.data	pnwz.o
0x20000128	COMPRESSED	0x00000002	Data	RW	2499		.data	spi_stm32_ll.o
0x2000012a	COMPRESSED	0x00000002	PAD					
0x2000012c	COMPRESSED	0x0000001c	Data	RW	3028		.data	led_stm32_ll.o
0x200010ac	-	0x00000080	Zero	RW	1302		.bss	pnwz.o
0x2000112c	-	0x00000810	Zero	RW	2247		.bss	uart_stm32_ll.o
0x2000193c	-	0x00000210	Zero	RW	3791		.bss	lora.o
0x20001b4c	COMPRESSED	0x00000004	PAD					
0x20001b50	-	0x00000018	Zero	RW	4244		.bss	datacoll.o
0x20001b68	-	0x00000014	Zero	RW	4296		.bss	init.o
0x20001b7c	-	0x00000018	Zero	RW	5004		.bss	radio.o
0x20001b94	-	0x00000070	Zero	RW	5058		.bss	sx1276.o
0x20001c04	-	0x00000100	Zero	RW	5199		.bss	sx1276-fsk.o
0x20001d04	-	0x00000100	Zero	RW	5535		.bss	sx1276-lora.o
0x20001e04	-	0x0000020c	Zero	RW	6047		.bss	esp8266.o
0x20002010	-	0x00000060	Zero	RW	6834		.bss	c_w.l(libspace.o)
0x20002070	-	0x00000200	Zero	RW	2		HEAP	startup_stm32f410rx.o
0x20002270	-	0x00000400	Zero	RW	1		STACK	startup_stm32f410rx.o

這個就對應我們的內存部分，存放我們的代碼中用到的各種變量數據（常量數據在以上的 ER_IROM 中）。同樣，該部分也有些對齊，除此之外全部是 Data、Zero、HEAP（堆）、STACK（棧）。還有一點就是，.bss 數據沒有加載域地址

最後在說明一下每一列（字段）的具體含義如下：：

Base Addr Size Type Attr Idx E Section Name Object

節的基地址 節的大小 類型 屬性 索引 節的名字 對象（節所屬的文件模塊）

Image component sizes

該部分列出了組成鏡像的各部分內容的大小等詳細信息。主要有三部分組成：用戶文件大小信息、庫文件大小信息、匯總信息。

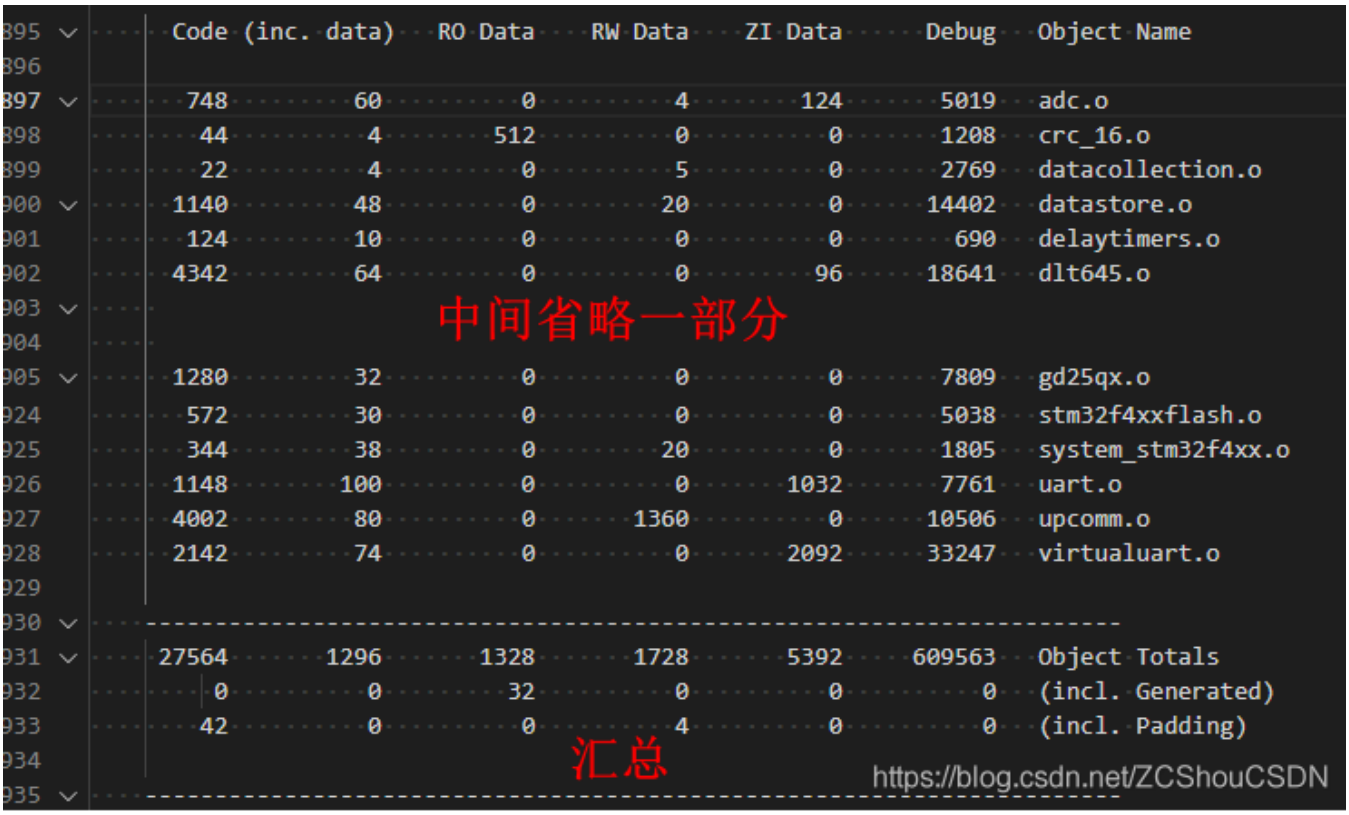
```
==============================================================================

Image component sizes

    Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

         700         40          0         12          0       7118   bh1750fvi.o
          44          4          0        512          0       1228   crc_16.o
         104         14          0          0         24       2793   datacoll.o
          68         14          0         12          0       2486   delay.o
         216         18          0          0          0      26667   delaytimers_stm32_ll.o
         854        110          0         12        524      10970   esp8266.o
         316         20         56          0          0       7886   flash.o
          40          6          0          0          0       8413   stm32f4xx_ll_utils.o
        2016        214          0         90        256      35110   sx1276-fsk.o
         618         60          0          0          0      14676   sx1276-hal.o
        2120        184        204         64        256      22171   sx1276-lora.o
        1088        112          0          0          0       8742   sx1276-loramisc.o
         418         50          0          2        112       4830   sx1276.o
          32         10         24          4          0       1191   system_stm32f4xx.o
         100          0          0          0          0       3265   uart.o
        2678        252          8          0       2064      99580   uart_stm32_ll.o
        3016         88          0       3204          0       9492   upcomm.o
         764         44          0          0          0      24804   virtuali2c1_ll.o
          16          0          0          0          0       1010   wdg.o
         624         42          0          8          0      11996   wifi.o

       34906       2226        950       4268       5476    1085202   Object Totals
           0          0         32          0          0          0   (incl. Generated)
          58          4          2         13          4          0   (incl. Padding)

用户文件

--------------------------------------------------------------------------------

    Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Member Name

          56          6          0          0          0         88   __2snprintf.o
          90          0          0          0          0          0   __dczerorl2.o
           8          0          0          0          0         68   __main.o
         312          4         17          0          0         92   __printf_flags_wp.o
          14          0          0          0          0         68   __printf_wp.o
           0          0          0          0          0          0   usenofp.o
         164         44          0          0          0        620   dunder.o
          24          0          0          0          0        124   fabs.o
          48          0          0          0          0        124   fpclassify.o
         248          0          0          0          0        152   poly.o
        3152        296        136          0          0        352   pow.o
           0          0          8          0          0          0   qnan.o
         224         30          0          0          0        172   round.o
         110          0          0          0          0        148   sqrt.o

       10620        742        180          0         96       8348   Library Totals
          26          0          7          0          0          0   (incl. Padding)

库文件

--------------------------------------------------------------------------------

    Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Name

        3204        124         29          0         96       3564   c_w.l
        3420        248          0          0          0       3092   fz_wm.l
        3970        370        144          0          0       1692   m_wm.l

       10620        742        180          0         96       8348   Library Totals

--------------------------------------------------------------------------------

==============================================================================

汇总
```

每一行表示一個模塊，其中各列的具體含義如下：

- **Code (inc. data)：** 這對應兩列數據，分別表示代碼佔用的字節數和代碼中內聯數據佔用的字節數。inc. data 是內聯數據（inline data）的縮寫。 內聯數據包括文字池和短字符串等。例如，上圖中第一行：表示 bh1750fvi.o 中代碼有 700 字節，其中內聯數據 40 字節。
- **RO Data：** 模塊中 RO 數據佔用的字節數。這是除去 Code（inc. data）列中 inc. data 數據外的只讀數據的字節數。 例如，我們定義的 const 數組！
- **RW Data：** 模塊中 RW 數據佔用的字節數。
- **ZI Data：** 模塊中 ZI 數據佔用的字節數。
- **Debug：** 模塊中調試數據佔用的字節數。例如，調試用的輸入節以及符號和字符串表。
- **Object Name：** 對象文件的名字。
- 特殊行的含義如下：
    - **Object Totals：** 它所在的行就是對各列數據的匯總。
    - **(incl. Generated)：** armlink 在生成鏡像文件時，可能會產生一些額外數據（interworking veneers, and input sections such as region tables）。如果存在這些額外數據，那麼他們就位於該行中顯示。
    - **Library Totals：** 顯示當前用戶代碼使用的庫文件中的各成員佔用的字節數。
    - **(incl. Padding)：** armlink 會插入填充以強制部分對齊。 如果 Object Totals 行中包含此類數據，則會在相關的（incl. Padding）行中顯示出 armlink 添加的對齊佔用的字節數。 同樣，如果 Library Totals 行中包含此類數據，則會在其關聯的行中顯示。
    - **Grand Totals：** 鏡像文件中所有模塊的每一列數據的總大小。
    - **ELF Image Totals：** 如果使用 RW 數據壓縮（默認值）來優化 ROM 大小，則最終鏡像的大小會發生變化，這會反映在 --info 的輸出中。 比較 Grand Totals 和 ELF Image Totals 下的字節數，以查看壓縮效果。
    - **ROM Totals：** 顯示包含鏡像所需的 ROM 的最小大小。 這其中不包括未存儲在 ROM 中的 ZI 數據和調試信息。

用戶文件大小信息

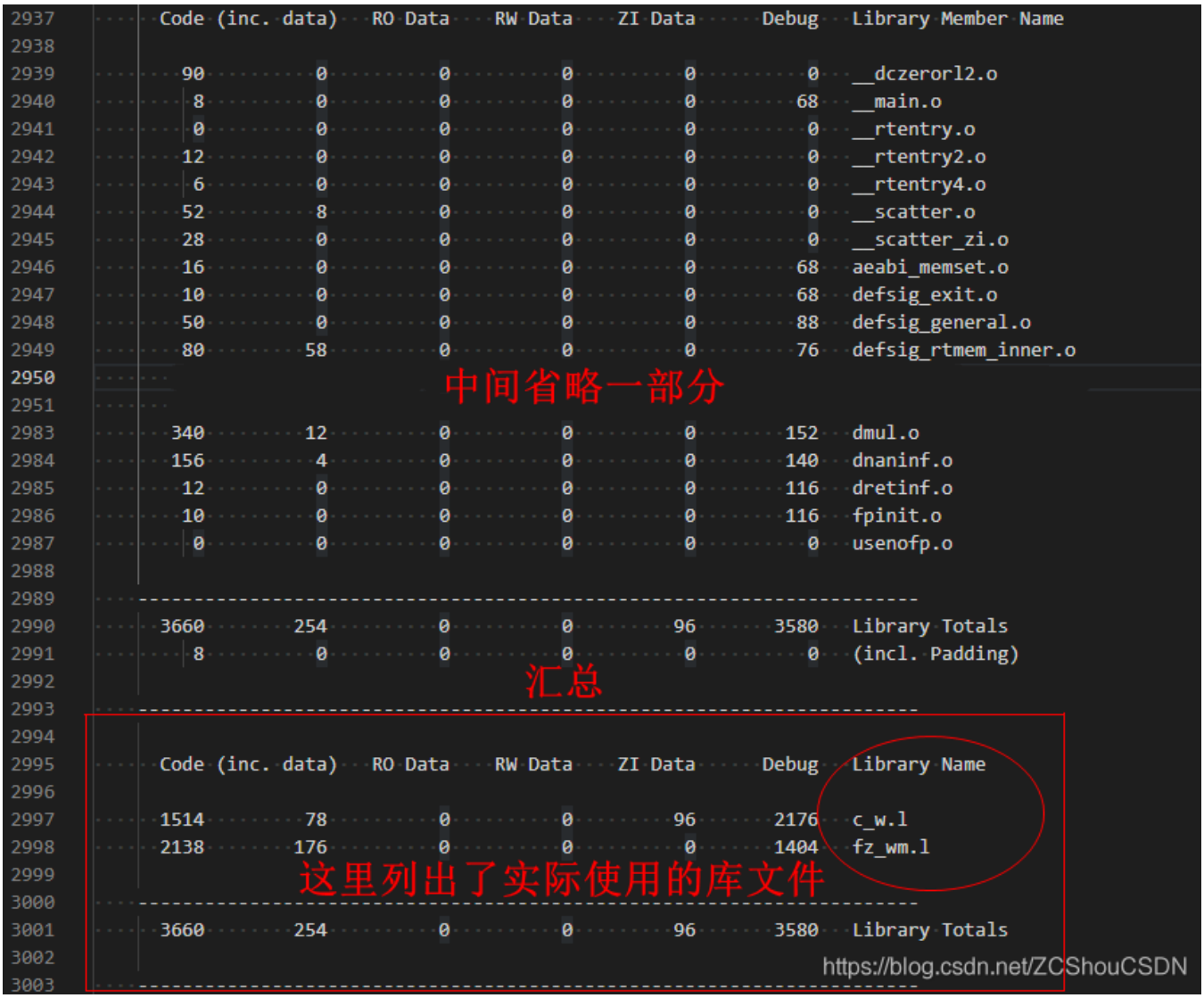
第一部分就是用戶源碼各模塊的大小信息！如下圖所示：



Code (inc. data)	RO Data	RW Data	ZI Data	Debug	Object Name
748	60	0	4	124	5019	adc.o
44	4	512	0	0	1208	crc_16.o
22	4	0	5	0	2769	datacollection.o
1140	48	0	20	0	14402	datastore.o
124	10	0	0	0	690	delaytimers.o
4342	64	0	0	96	18641	dlt645.o
中间省略一部分					
1280	32	0	0	0	7809	gd25qx.o
572	30	0	0	0	5038	stm32f4xxflash.o
344	38	0	20	0	1805	system_stm32f4xx.o
1148	100	0	0	1032	7761	uart.o
4002	80	0	1360	0	10506	upcomm.o
2142	74	0	0	2092	33247	virtualuart.o
27564	1296	1328	1728	5392	609563	Object Totals
0	0	32	0	0	0	(incl. Generated)
42	0	0	4	0	0	(incl. Padding)
汇总					

庫文件大小信息

第二部分用戶源碼中使用的各 C 庫模塊的大小信息以及使用的 C 庫文件名！如下圖所示：



這部分中，除了列出了庫文件的獨立單元模塊的大小，還列出了我們的源碼中實際使用的庫文件。這個會根據我們源碼中引用的庫函數的不同而變化。

匯總信息

主要就是各部分數據的匯總大小，如下圖所示

```
==============================================================================

    Code (inc. data)   RO Data    RW Data    ZI Data      Debug

   31224       1550       1328       1728       5488     588047   Grand Totals
   31224       1550       1328         88       5488     588047   ELF Image Totals (compressed)
   31224       1550       1328         88          0          0   ROM Totals

==============================================================================

  Total RO  Size (Code + RO Data)                32552 (  31.79kB)
  Total RW  Size (RW Data + ZI Data)              7216 (   7.05kB)
  Total ROM Size (Code + RO Data + RW Data)      32640 (  31.88kB)

==============================================================================
```

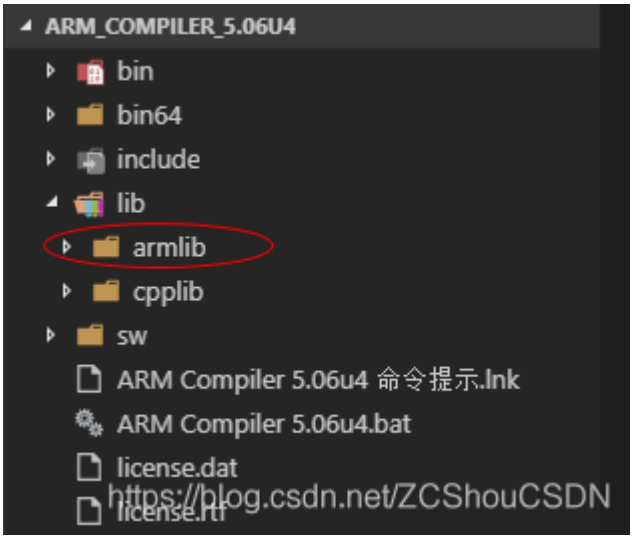

其中最下面的三行數據是匯總的再匯總，以方便我們的使用

- **Total RO Size：** 就是我們的可執行程序中常量數據（代碼和只讀數據）的大小。
- **Total RW Size：** 就是我們的可執行程序中需要佔用的內存的大小。
- **Total ROM Size：** 就是我們的可執行程序本身的大小。這個大小就等於我們的可執行文件的大小。

需要特殊注意的是， `armlink` 輸出的是 `.axf` 文件，這個文件中包含調試信息，並不是我們需要使用的可執行文件，我們使用 `fromelf` 工具從中提取的文件才是真正的可執行文件。這裡的匯總大小指定是實際使用的可執行文件的大小。關於這部分，參考博文 [ARM 之一 ELF 文件、鏡像（Image）文件、可執行文件、對象文件 詳解]。

ARM 庫文件

我們可以在 ARM 編譯套件的目錄下找到這兩個文件，路徑如下圖所示：



下面我們使用 ARM 編譯套件中相應的工具來看看具體文件。關於編譯套件的詳細使用說明可以參考博文《ARM 之 主流編譯器（armcc、iar、gcc for arm）詳細介紹》。具體使用的工具就是 armar.exe，這是 ARM 的庫文件管理工具。

從中我們可以看到有__main.o 等文件，接下來我們可以使用 armar -x 命令將 c_w.l 解壓出以上全部文件，然後使用 fromelf 來查看__main.o 的詳細信息，這裡就不一一嘗試了！

參考

1. ARM 的鏈接器用戶手冊：ARM® Compiler v5.06 for µVision® armlink User Guide
2. ARM 的編譯器用戶手冊：ARM® Compiler v5.06 for µVision® armcc User Guide