MTK 的 Scatter file

因为我的图片文件较大，容量超过了 14M，所以改用 32MRom＋8M　Ram，scatter 文件我只改了 3 个地方：

1.　ROM 总大小，即把　ROM 0x00000000 0x00e00000　改成了　ROM 0x00000000 0x01e00000（由 14M－＞30M）；

2.　把 ROM　的 4 个分区都改成了 8M，即：

　　分区 1：ROM　　0x00000000 0x00400000　改成了　ROM 0x00000000 0x00800000（4M　－＞8M）；

　　分区 2：ROM2　　＋0x0　　0x00400000　改成了　ROM2　　＋0x0　0x00800000（4M　－＞8M）；

　　分区 3：ROM3　　＋0x0　　0x00400000　改成了　ROM3　　＋0x0　0x00800000（4M　－＞8M）；

　　分区 4：ROM4　　＋0x0　　0x00400000　改成了　ROM4　　＋0x0　0x00800000（4M　－＞8M）；

3.　把内存总大小改成了 8M，原来才 4M：

　　我把 DUMMY_END 0x08400000 0x04　改成了　DUMMY_END 0x08800000 0x04

　　这样改后，我的 16M　bin 档烧进去后，还是开不了机。超过 14M 原因是图片较大。请教各位高手，我这样改 Satter 文件有错没有？要怎样改呀，请指点！

请按以下我说的去分析这个问题.

1.首先，使用大的 FLASH，改 scatter 文件只改结束地址就 OK，所以只要做你所说的第一步。后面的是按照偏移量自动去算的，除非是加特殊的第三方软件，一般是不用修改的，dummy end 也不用去改，不影响。你用的 FLASH 一般是由 NOR+SDRAM 组成。我们这里只讨论 NOR 的部分。NOR 里确定 code region 是最优先的。你的情况是需要腾出一个大于 14M 的空间。那么我们假如用 20M。那结束地址就是 00001400000.

2.然后再说为什么开不了机的问题。因为你的 FAT 和 NVRAM 及 Z 盘的配置有问题。你用去 30M 给 BIN，那就余下 2M 给 NVRAM+Z+FAT.你认为够么。你得去 custom_memory...这只文件下查一下，你的 partion_sector 是多少.1024 代表 512K。不出意外的话，你留给 NVRAM+Z 的空间只有 1.5M 左右。这是远远不够的。

3.而你用的这个 FLASH，按我的猜想，应该是 toshiba 的，如果是 spansion 的，那你还有不分区的可能，直接分为 28+4，28 作 BIN，4M 做 NVRAM+FAT，有可能能开机。但如果是 TOSHIBA 的，那多半是 16 个 2M 的 bank 结构的。那就没办法了。必须去打开 makefile 里的 enhance_single_bank...这只开关。然后在 xls 表里对最后一个 bank 进行扩容。在表里的 last bank 下改其大小。必须按大的 block 的整数倍增加。

按照上面的去试下，应该可以开机。

Scatter file (分散加载描述文件)用于 armlink 的输入参数，他指定映像文件内部各区域的 download 与运行时位置。Armlink 将会根据 scatter file 生成一些区域相关的符号，他们是全局的供用户建立运行时环境时使用。

（注意：当使用了 scatter file 时将不会生成以下符号：

Image$$RW$$Base,

Image$$RW$$Limit,

Image$$RO$$Base,

Image$$RO$$Limit,

Image$$ZI$$Base,

Image$$ZI$$Limit）

二 什么时候使用 scatter file

当然首要的条件是你在利用 ADS 进行项目开发，下面我们看看更具体的一些情况。

1 存在复杂的地址映射：例如代码和数据需要分开放在在多个区域。

2 存在多种存储器类型：例如包含 Flash,ROM,SDRAM,快速 SRAM。我们根据代码与数据的特性把他们放在不同的存储器中，比如中断处理部分放在快速 SRAM 内部来提高响应速度，而把不常用到的代码放到速度比较慢的 Flash 内。

3 函数的地址固定定位：可以利用 Scatter file 实现把某个函数放在固定地址，而不管其应用程序是否已经改变或重新编译。

4 利用符号确定堆与堆栈：

5 内存映射的 IO：采用 scatter file 可以实现把某个数据段放在精确的地指处。

因此对于嵌入式系统来说 scatter file 是必不可少的，因为嵌入式系统采用了 ROM，RAM，和内存映射的 IO。

三 scatter file 实例

1 简单的内存映射

LOAD_ROM 0x0000 0x8000

{

EXEC_ROM 0x0000 0x8000

{

*(+RO)

}

RAM 0x10000 0x6000

{

*(+RW, +ZI)

}

}

LOAD_ROM(下载区域名称) 0x0000(下载区域起始地址) 0x8000(下载区域最大字节数)

{

EXEC_ROM(第一执行区域名称) 0x0000(第一执行区域起始地址) 0x8000(第一执行区域最大字节数)

{

*(+RO(代码与只读数据))

}

RAM(第二执行区域名称) 0x10000(第二执行区域起始地址) 0x6000(第二执行区域最大字节数)

{

*(+RW(读写变量), +ZI(未初始化变量))

}

}

2 复杂内存映射

```
LOAD_ROM_1 0x0000
{
EXEC_ROM_1 0x0000
{
program1.o(+RO)
}
DRAM 0x18000 0x8000
{
program1.o (+RW, +ZI)
}
}
LOAD_ROM_2 0x4000
{
EXEC_ROM_2 0x4000
{
program2.o(+RO)
}
SRAM 0x8000 0x8000
{
program2.o (+RW, +ZI)
}
}
```

```
LOAD_ROM_1 0x0000(下载区域一起始地址)
{
EXEC_ROM_1 0x0000(第一执行区域开始地址)
{
program1.o(+RO) (program1.o 内的 Code 与 RO data  放在第一执行区域)
}
```

DRAM 0x18000(第二执行区域开始地址) 0x8000(第二执行区域最大字节数)

{

program1.o (+RW, +ZI) (program1.o 内的 RW data 与 ZI data 放在第二执行区域)

}

}

LOAD_ROM_2 0x4000(下载区域二起始地址)

{

EXEC_ROM_2 0x4000

{

program2.o(+RO) (program2.o 内的 Code 与 RO data 放在第一执行区域)

}

SRAM 0x8000 0x8000

{

program2.o (+RW, +ZI) (program2.o 内的 RW data 与 ZI data 放在第二执行区域)

}

}

2.1 BNF 符号与语法

"：由引号赖标示的符号保持其字面原意，如 A”+”B 标示 A+B。

A ::= B ： 定义 A 为 B。

[A]：标示可选部分，如 A[B]C 用来标示 ABC 或 AC。

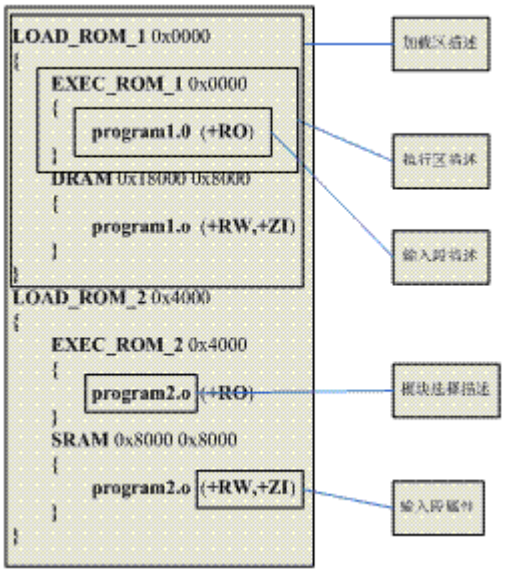
A+：用来标示 A 可以重复任意次，如 A+可标示 A,AA,AAA, …

A*：同 A+。

A | B：用来标示选择其一，不能全选。如 A|B 用来标示 A 或者 B。

(A B)：标示一个整体，当和|符号或复杂符号的多次重复一起使用时尤其强大，如（AB）+(C|D)标示 ABC,ABD,ABABC,ABABD, …

2.2 分散加载文件各部分描述



如图 2.1 所示为一个完整的分散加载脚本描述结构图。下面我们对图示中各个部分进行讲述。

2.2.1 加载区描述

每个加载区有：

名称：供连接器确定不同下载区域

基地址：相对或绝对地址

属性：可选

最大字节数：可选

执行区域列：确定执行时各执行区域的类型与位置

load_region_name (base_address | ("+" offset)) [attribute_list] [max_size]

"{"

execution_region_description+

"}"

load_region_name：下载区域名称，最大有效字符数 31。（并不像执行区域段名用于 Load$$region_name，而是仅仅用于标示下载区域）。

base_address：本区域内部目标被连接到的地址（按字对齐）。

+offset：相对前一个下载区域的偏移量（4 的整数倍，如果为第一个区域）。

2.2.2 执行区描述

每个执行区有：

名称：供连接器确定不同下载区域

基地址：相对或绝对地址

属性：确定执行区域的属性

最大字节数：可选

输入段：确定放在该执行区域的模块

exec_region_name (base_address | "+" offset) [attribute_list] [max_size]

"{"

input_section_description+

"}"

exec_region_name：执行区域名称，最大有效字符数 31。

base_address：本执行区域目标要被联接到的位置，按字对齐。

+offset：相对于前一个执行区域结束地址的偏移量，4 的整数倍；如果没有前继之能够行区域（本执行区域为该下载区域的第一个执行区域），则该偏移量是相对于该下载区域的基址偏移量。

attribute_list：PI，OVERLAY，ABSOLUTE，FIXED，UNINIT。

PI: 位置独立。

OVERLAY: 覆盖。

ABSOLUTE: 绝对地址。

FIXED: 固定地址，下载地址与执行地址具有该地址指示确定。

UNINIT: 未初始化数据。

RELOC：无法明确指定执行区域具有该属性，而只能通过继承前一个执行区或父区域获得。

对于 PI，OVERLAY，ABSOLUTE，FIXED，我们只能选择一个，缺省属性为 ABSOLUTE。一个执行区域要么直接继承其前面的执行区域的属性或者具有属性为 ABSOLUTE。

具有 PI，OVERLAY，RELOC 属性的执行区域允许其地址空间重叠，对于 BSOLUTE，FIXED 属性执行区域地址空间重叠 Armlink 会报错。

max_size：可选，他用于指使 Armlink 在实际分配空间大于指定值时报错。

input_section_description：指示输入段的内容。

基本语法 2

2.2.3 输入段描述

输入段：

模块名：目标文件名，库成员名，库文件名。名称可以使用通配符。

输入段名，或输入段属性(READ-ONLY,CODE)。

module_select_pattern

["("

("+" input_section_attr | input_section_pattern)

([","] "+" input_section_attr | "," input_section_pattern))*

")"]

2.2.3.1

module_select_pattern：选择的模块名称（目标文件，库文件成员，库文件），模块名可以使用通配符（*匹配任意多个字符，？匹配任意一个字符），名称不区分字母大小写，它是供选择的样本。

例 1：*libtx.a (+RO)

libtx.a 为 threadX 库文件。

例 2：tx_ill.o (INIT)

tx_ill.o 为 threadX 中断向量目标文件。

2.2.3.2

input_section_attr：输入段属性选择子，每个选择子以”+”开头，选择子不区分大小写字符。

选择子可选：

RO-CODE，

RO-DATA，

RO（ selects both RO-CODE and RO-DATA），

RW-DATA，

RW-CODE，

RW（ selects both RW-CODE and RW-DATA），

ZI，

ENTRY（ that is a section containing an ENTRY point）。

以下同义词可以选择：

CODE (for RO-CODE)，

CONST(for RO-DATA)，

TEXT (for RO)，

DATA (for RW)，

BSS (for ZI)。

还有两个伪属性：FIRST，LAST。如果各段的先后顺序比较重要时，可以使用 FIRST，LAST 标示一个执行区域的第一个和最后一个段。

例 1：os_main_init.o (INIT ,+FIRST)

FIRST 表示放于本执行区域的开始处。

例 2：*libtx.a (+RO)

RO 表示*libtx.a 的只读部分。

2.2.3.3

input_section_pattern：输入段名。

例 1：os_main_init.o (INIT ,+FIRST)

INIT 为 os_main_init.o 的一个段。

例 2：os_stackheap.o (heap)

heap 为 os_stackheap.o 的一个段。

例 3：os_stackheap.o (stack)

stack 为 os_stackheap.o 的一个段。

提高篇

3.1 在 scatter file 中指定胶合段

胶合段用于实现 ARM 代码到 Thumb 代码的切换或者实现代码的长转移。使用 scatter file 可以指定怎样放置胶合输入段。通常，在 scatter file 中一个执行区域可以拥有胶合段选择 *(Venner$$Code)。

Armlink 把胶合输入段放到拥有段选择符*(Veneer$$Code)的执行区域中，这样做是安全的。

可能由于地址范围问题或者受执行区域大小限制无法完成把胶合段分配个某个执行区

域。如果当发生胶合段无法加到指定区域时，他将会被加到那些包含了生成胶合段的可重载输入段的执行区域。

3.2 创建根执行区域

根执行区域就是指那些执行与加载时地址相同的区域。

当你为映像文件指定初始化入口点或者由于你只使用一个 ENTRY 导向符而使得连接器创建初始化入口位置时，你就必须确保该入口点位于根执行区域。如果入口点不在根执行区域，连接将会失败，连接器会报错。

如：ENTRY point (0x00000000) lies within non-root region ER_ROM

可以通过以下方式实现在 scatter file 中指定根执行区域。

① 显示或缺省的指定执行区的属性为 ABSOLUTE，同时使得加载区域与第一个执行区域具有相同的地址。

② 使用 FIXED 属性使得执行区域的加载地址与其执行时地址保持不变。

3.3 创建根执行区域

可以通过在 scatter file 中为某个执行区域指定 FIXED 属性来实现该区域加载于运行时地址保持不变。

FIXED 可以用来在一个加载区中创建多个根执行区域。因此我们可以通过它实现把某个函数或一段数据放到目标地址，从而可以通过指针方便地访问该地址。比如，我们可以实现把常量表和 checksum 放到 ROM 上的固定地址处。

注意：

① 为了使得代码更加易于维护和调试，请尽量少使用 scatter file 来指定放置位置，而是应该尽可能多地让连接器来确定函数和数据的位置。

②

3.3.1 怎样把函数或数据放到指定地址

通常，编译器处理来自单个源文件的 RO,RW,和 ZI 段。这些区域包括源文件的代码与数据。如果打算把单个函数或数据项放到某个固定地址，我们就必须让编译器单独处理这些函数和数据。

我么可以通过以下方式处理单个目标：

① 把函数和数据放到其源文件。

② 使用编译选项 –zo 为每个函数单独生成一个目标文件。(参看 ARM Compiler Guide)

③ 在 C,C++源文件内利用 #pragma arm section 来生成多命名段。

④ 在汇编源文件内利用 AREA 导向符来生成可重载段。

3.3.2 怎样放置单个目标文件的内容

3.3.3 怎样使用 ARM 的 section pragma

通常把函数和数据放到其源代码文件，然后放到其目标文件的相应段中。然而，我们也可以#pragma 和 scatter file 实现单独处理某个命名段。

```
// file adder.c
int x1 = 5; // in.data
int y1[100]; // in.bss
int const z1[3] = {1,2,3}; // in.constdata
int sub1(int x) // in.text
{
return x-1;
}
#pragma arm section rwdata = "foo", code ="foo"
int x2 = 5; // in foo (data part of region)
char *s3 = "abc"; // s3 in foo, "abc" in .constdata
int add1(int x)
{
return x+1;
} // in foo (.text part of region)
#pragma arm section code, rwdata // return to default placement

FLASH 0x24000000 0x4000000
{
```

```
FLASH 0x24000000 0x4000000
{
init.o (Init, +First) ;place code from init.o first
* (+RO) ;sub1(), z1[]
}
32bitRAM 0x0000
{
vectors.o (Vect, +First)
* (+RW,+ZI) ;x1, y1
}
ADDER 0x08000000
{
adder.o (foo) ;x2, string s3, and add1()
}
}
```

从 MTK 的 scat 文件谈 ROM 和 RAM 的分配，管理和问题解决

mtk25 平台默认的是 128Mbit flash memory 和 32Mbit SRAM，因为 1BYTE 等于 8BIT，所以就是我们通常所说的是 16M ROM 和 4M RAM，不过由于文件系统占用 2M，这 2M 一般又被分为系统盘和用户盘，系统盘存储 NV 文件和 MMS 相关文件，对用户不可见，用户盘用户连上电脑就可以看到，但由于一些原因，有不少手机是不设用户盘的，用户盘过大，会导致彩信等一些模块不稳定，所以很多使用 NORFALSH 时不设用户盘，但 NAND FALSH 一般都会设置一定的用户盘。如果需要设置，只要修改宏 PARTITION_SIZE 值就可以控制。所以我们能够使用的就只剩下 14，这一点可以从 BUILD 目录下的 scatWINGTECH25_GEMINI.txt 文件的声明部分看到，在 SCAT 文件中，有一行是 SCHEME　: external 14MB flash memory and 4MB SRAM，就是说 14MROM 和 4MRAM。

由于用户需求不同，有些时候我们会修改 RAM 和 ROM 大小，一般就要相应的修改 SCAT 文件。在 SCAT 文件中，我们可以看到行 ROM 0x00000000 0x00e00000，就是说可以使用的 ROM 从 0x00000000 开始，到 0x00e00000 结束，共计 0x00e00000 字节，在下面又可以看到这些 ROM 被分成四个 4M 的

段使用。在 SCAT 的行 EXTSRAM_LARGEPOOL_NORMAL 0x08000000 处我们可以看到 RAM 的使用情况，地址从 0x08000000 开始， 到 0x08400000 结束，共计 0x00400000BYTE,即 4MBYTE，如果你是 32MROM,8MRAM,就要修改 ROM 0x00000000 0x00e00000 为 ROM 0x00000000 0x01c00000，修改 DUMMY_END 0x08400000 0x04 为 DUMMY_END 0x08800000 0x04，这样的修改，现在的 ROM 和 RAM 大小都为以前默认的 2 倍。

其实有时 RAM 紧张时不一定非要采用增加 RAM 来实现，这样成本较大，可以采用复用内存也可以节约大量内存。在 SCAT 文件中，很多时候,我们可以看到关健字 overlay，这是一些手机的应用中为节省内存使用的复合内存，如 INTSRAM_MULTIMEDIA 0x40000000 0xC000，声明了 MED 复用内存的起始地址，只要不冲突，这几乎是最好的解决内存紧张的方法。如果 ROM 超过了，可能会比较麻烦一些，去掉不必要的图片，音乐，减小图片的质量，去掉一些不必要的功能，把宏函数转为普通函数都可以节约一部分 ROM。

如果 RAM 或者 ROM 编绎到最后出错，提示 ROM 超了或者 RAM 超过了，这时就要精确计算超出部分的大小，然后再根据计算的大小寻找解决办法。计算的方法是打开 LIS 文件，把 RAM 或者 ROM 加起来，减去 14 或者 4，超过的字节数，就是需要调整的内存大小

SCAT 文件的规范可以查看 ADS 的帮助文档

---

mtk25 平台默认的是 128Mbit flash memory 和 32Mbit SRAM，因为 1BYTE 等于 8BIT，所以就是我们通常所说的是 16M ROM 和 4M RAM，不过由于文件系统占用 2M，这 2M 一般又被分为系统盘和用户盘，系统盘存储 NV 文件和 MMS 相关文件，对用户不可见，用户盘用户连上电脑就可以看到，但由于一些原因，有不少手机是不设用户盘的，用户盘过大，会导致彩信等一些模块不稳定，所以很多使用 NORFALSH 时不设用户盘，但 NAND FALSH 一般都会设置一定的用户盘。如果需要设置，只要修改宏 PARTITION_SIZE 值就可以控制。所以我们能够使用的就只剩下 14，这一点可以从 BUILD 目录下的 scatWINGTECH25_GEMINI.txt 文件的声明部分看到，在 SCAT 文件中，有一行是 SCHEME : external 14MB flash memory and 4MB SRAM，就是说 14MROM 和 4MRAM。

由于用户需求不同，有些时候我们会修改 RAM 和 ROM 大小，一般就要相应的修改 SCAT 文件。在 SCAT 文件中，我们可以看到行 ROM 0x00000000 0x00e00000，就是说可以使用的

ROM 从 0x00000000 开始，到 0x00e00000 结束，共计 0x00e00000 字节，在下面又可以看到这些 ROM 被分成四个 4M 的段使用。在 SCAT 的行 EXTSRAM_LARGEPOOL_NORMAL 0x08000000 处我们可以看到 RAM 的使用情况，地址从 0x08000000 开始，到 0x08400000 结束，共计 0x00400000BYTE,即 4MBYTE，如果你是 32MROM,8MRAM,就要修改 ROM 0x00000000 0x00e00000 为

ROM 0x00000000 0x01c00000，修改 DUMMY_END 0x08400000 0x04 为 DUMMY_END 0x08800000 0x04，这样的修改，现在的 ROM 和 RAM 大小都为以前默认的 2 倍。

其实有时 RAM 紧张时不一定非要采用增加 RAM 来实现，这样成本较大，可以采用复用内存也可以节约大量内存。在 SCAT 文件中，很多时候,我们可以看到关健字 overlay，这是一些手机的应用中为节省内存使用的复合内存，如 INTSRAM_MULTIMEDIA 0x40000000 0xC000，声明了 MED 复用内存的起始地址，只要不冲突，这几乎是最好的解决内存紧张的方法。如果 ROM 超过了，可能会比较麻烦一些，去掉不必要的图片，音乐，减小图片的质量，去掉一些不必要的功能，把宏函数转为普通函数都可以节约一部分 ROM。

如果 RAM 或者 ROM 编绎到最后出错，提示 ROM 超了或者 RAM 超过了，这时就要精确计算超出部分的大小，然后再根据计算的大小寻找解决办法。计算的方法是打开 LIS 文件，把 RAM 或者 ROM 加起来，减去 14 或者 4，超过的字节数，就是需要调整的内存大小

---

scatter 文件的写法

很多朋友对分散加载不是很理解，其实它的原来很简单，这些加载的原理都源自生活。

由于现在的嵌入式技术发展比较快，各类存储器也层出不穷，但是它们在容量、成本和速度上有所差异，嵌入式系统又对成本比较敏感，那么合理的选择存储器和充分的利用存储器资源成为一个必要解决的问题。咋们工程师最喜欢的就是发掘问题，然后解决问题，基于嵌入式系统对存储器的敏感，那么要合理的利用存储器资源，就必须找到一种合理的方式。工程师们发现，可以把运行的程序放在不同成本的存储器中来寻找这个成本的支点，比如把没有运行的但是较为庞大的程序放在容量大、成本低、速度也较低的FLASH存储器中，要用的时候再去拿。但是，这里面又有一个问题，嵌入式本身就对信号的处理速度有较高的要求，这点在实时操作系统的定义上上有所体现。所以那些经常要用的程序段如果要保证其高速的运行那么就得放在一个在高速的存储器中，不过这是有代价的：较高成本，小容量。但

是，相信由于技术的发展这个问题终将被解决，到时候寻找平衡点的问题也就不存在了。好了，说了多了点。切入正题。

程序总有两种状态：运行态和静止态。当系统掉电的时候程序需要被保存在非易失性的存储器中，且这个时候程序的排放是按照地址依次放的，换句话说：我才懒得管它怎么放，只要不掉就行。当系统上电后，CPU 就要跑起来了，CPU 属于高速器件，存储器总是不怎么能跟得上，既然跟不上那么我们就尽量缩短它们之间的差距，那留下一条路，那就是尽量提高存储器的读取速度，存储器类型决定其速度的水平，那么尽量放在速度高的存储器就成为首选解决方案。那么我们就把要执行的程序暂时拿到速度较快的 RAM 中。那么拿的过程就牵涉到程序的加载了。这就是要解决的问题。

一个映像文件由域（region）、输出段(output sections)和输入段（input sections）组成。不要想得太复杂，其实他们之间就是包含与被包好的关系。具体关系是这样的：

映像文件 > 域 > 输出段 > 输入段

输入段：

输入段就是我们写的代码+初始化的数据+应该被初始化为 0 的数据+没有初始化的数据，用英文表示一下就是：RO（ReadOnly），RW （ReadWrite），ZI（ZeroInitialized）,NOINIT（Not Initialized）。ARM 连接器根据各个输入段不同的属性把相同的拿再一起组合一下就成为了输出段。

请看看平时写的东东：

```
AREA    RESET, CODE, READONLY

AREA    DSEG1, DATA, READWRITE

AREA    HEAP, NOINIT, READWRITE
```

看出其属性没？

输出段：

为了简化编译过程和更容易取得各种段的地址，那么把多个同属性的输入段按照一定的规律组合在一起，当然这个输出段的属性就和它包含的输入段的属性一样咯。输入段的排放规律就是：最先排放 RO 属性的输入段，然后是 RW 属性段，最后是 ZI 或 NOINIT 段。

域：

为什么还要加一层域，我的理解是由于代码的功能不同，那么我们有必要把不同功能的代码分类放。我们可以把需要高速执行的代码放在一起、把对速度要求不高的放在一起、把执行频率高的放在一起，把执行频率低的放在一起...那么按照这种方式放的代码就可以根据其具体需要放在不同的存储器中了。这样可以提高程序执行速度。一个域中包含 1~3 个输出段。

映像文件：

我暂时把映像文件理解成烧到存储器中的文件，由 N 个域组成。这些域其实可以看做是独立的模块，只是他们按照一定的顺序（这个顺序还是：RO+RW+ZI）被捆绑在一起，这样才方便烧写到非易失存储器中去。

好了，了解了映像文件的组成，那么来看看映像文件是怎么跑起来的。

映像文件就是有 N 节车厢的火车，车厢（域）里装着要送到不同站（不同类型的存储器）的货物。到相应的站了，那么就把相应的车厢拿下来。指挥拿这个的就是 scatter 文件。拿下货物车厢后，我们就解开它，把里面的品牌为 RO 的货物提取出来，按照 scatter 的指示发给某个地址，然后再先后把品牌为 RW 和 ZI 的货物发到 scatter 指定的地址。

看看这个加深理解：

```
LOAD_ROM1     0X00000000    ; 从火车上取出来时的地址（如：成都站）
{
        EXEC_ROM1       0x40000000
        {
            PROGRAM.O（+RO） ;把品牌 RO 的货物发给 0x40000000 去
             RAM1          0x80000000
             {
             PROGRAM.O(+RW,+ZI)；把品牌 RW,ZI 的货物依次发给 0x80000000
              }
         }
         ......
}
```

其他的段也可以这样依葫芦画瓢。

scatter 的原理就介绍这样，其中的语法和规则要多写多把代码的地址拖出来看才能体会。不过都是很简单的，生活中的小常识就能解决这些问题。为什么？因为设计这些规则的工程师的灵感就是源自生活。嘿嘿...享受把代码随处放的乐趣吧，...enjoy...

---

ads1.2 scatter 文件分析

；总共三个分散加载文件 mem_a.scf，mem_b.scf，mem_c.scf，区别是加载地址不一样

；具体加载哪个，在 DebugInExram->ARM Linker->Scatter 定义，链接类型选择 Scattered

；image entry point 一定要跟 ROM_LOAD 值一样

；ROM_LOAD 为加载区的名称，其后面的 0x00000000 表示加载区的起始地址（存放程序代码的起始地址）

```
ROM_LOAD 0x0
{
；ROM_EXEC 描述了执行区的地址，放在第一块位置定义
    ROM_EXEC 0x00000000
    {
；从起始地址开始放置向量表(即 Startup.o(vectors, +First)，其中 Startup.o 为 Startup.s 的目标文件)
；+First 表示 Vector 段放在最前面
        Startup.o (vectors, +First)
；接着放置其它代码(即* (+RO))，* 是通配符，类似 WINDOW 下搜索用的通配符
        * (+RO)
    }
；变量区 IRAM 的起始地址为 0x40000000
    IRAM 0x40000000
    {
；放置 Startup.o (MyStacks)
        Startup.o (MyStacks)
    }
```

```
；+0 表示接着上一段，UNINIT 表示不初始化
    STACKS_BOTTOM +0 UNINIT
    {
；放置 AREA    StackBottom, DATA, NOINIT
      Startup.o (StackBottom)
    }
；接着从 0x40004000 开始，放置 AREA    Stacks, DATA, NOINIT，UNINIT 表示不初始化
    STACKS 0x40004000 UNINIT
    {
      Startup.o (Stacks)
    }
；外部 RAM 从 0x80000000 开始为变量区
；如果片外 RAM 起始地址不为 0x8000 0000，则需要修改 mem_.scf 文件
    ERAM 0x80000000
    {
      * (+RW,+ZI)
    }
；+0 表示接着上一段，UNINIT 表示不初始化
    HEAP +0 UNINIT
    {
；放置堆底，  AREA    Heap, DATA, NOINIT
      Startup.o (Heap)
    }
；接着在外部 0x80080000 放置堆顶
；这个地址是片外 RAM 的结束地址，根据实际情况修改
    HEAP_BOTTOM 0x80080000 UNINIT
    {
      Startup.o (HeapTop)
    }
}
```

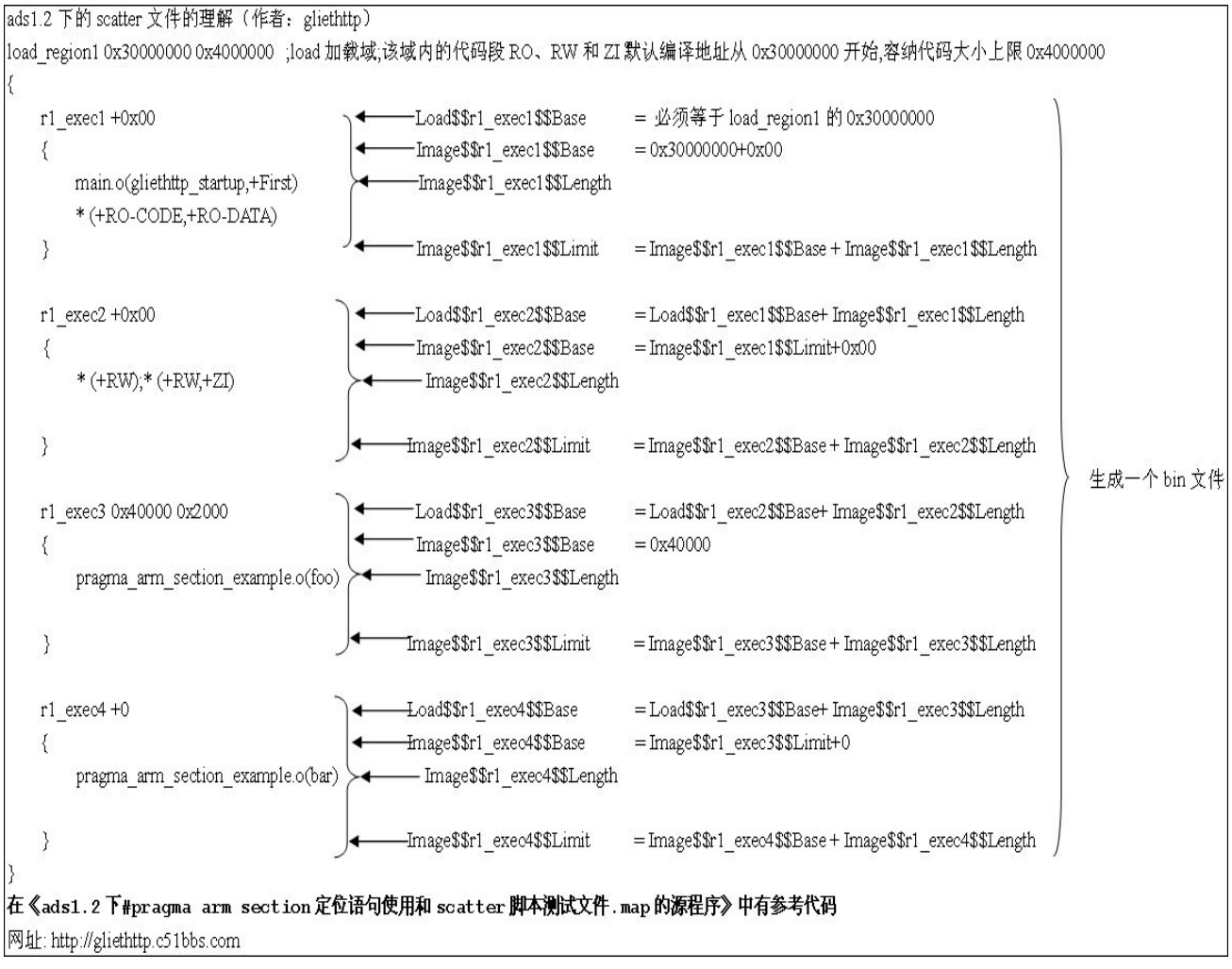
；重定向__user_initial_stackheap 函数

；分配新的 bottom_of_heap 地址等，R0-R3 是函数必须的返回值，返回 bottom_of_heap 的值

；通过分散加载描述文件，重定向其位置，bottom_of_heap 等已经在 Startup.s 中定义为 DATA 类型

__user_initial_stackheap

```
    LDR    r0,=bottom_of_heap
;    LDR    r1,=StackUsr
    LDR    r2,=top_of_heap
    LDR    r3,=bottom_of_Stacks
    MOV    pc,lr
```



复用问题

scatter 文件,个人认为主要功能就是重定向.按要求重新堆放 rom,ram. 1.Nor flash 的特性决定了 scat 的首地址是 0x08000000 . 前面谈到,26 是 128mbit,即为 16Mbyte.共分成 4 块,每块 4m. 第一块主要方系统内核相关的,L1,l4,等内容.第二块则为,wap,java 相关. 剩下两块,则堆放其它剩余的内容. 2.版本编译中,很可能出现 rom 超出的版本,一般就是首先计算出超出量,然后修改资源数量,大小等方法; 3.ram 堆放差不多.

但是 ram 是可以读写的,决定了该空间可以重用.利用关键字 overlay.如下: WAP_SRAM 0x87A7800 OVERLAY 0x58800 { wap_mem.obj(LARGEPOOL_ZI) } MYSRAM 0x087E0000 OVERLAY 0x020000 { myapp.obj (LARGEPOOL_ZI) } 由于 wap 中的这个数组和 myapp.obj 中的 zi 型数组,在使用中,不会冲突,则可以重用. 就像一个容器,只要不同时适用,两个人则可以共用这个容器. 在上面的例子中,为了更加安全,是从后往前共用的. 0x087E0000 +0x20000=0x87A7800 +0x58800