Main 函数运行前的分析（原创，转载请注明出处）

一、 启动文件的介绍

在 MDK 的启动文件 startup_stm32f10x_md_vl 中，该文件分别定义了栈段、堆段、存放中断向量表的数据段、还有一个代码段

大小为 0x400 的栈段定义如图 1-1：

```
Stack_Size      EQU     0x00000400

                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp
```

图 1-1

大小为 0x200 的堆段如图 1-2：

```
Heap_Size       EQU     0x00000200

                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit
```

图 1-2

由其定义属性可知，栈和堆都未初始化，该过程由后面的_user_initial_stackheap 来完成。

存放中断向量的数据段，如图 1-3 所示：

```
; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size
```

图 1-3

10 个系统异常过程段和在同一地址的外部中断过程段，下面我们就详细介绍上电复位的代码段，如图 1-4 所示:

```
                AREA    |.text|, CODE, READONLY

; Reset handler
Reset_Handler    PROC
                 EXPORT  Reset_Handler             [WEAK]
     IMPORT  __main
     IMPORT  SystemInit
                 LDR     R0, =SystemInit
                 BLX     R0
                 LDR     R0, =__main
                 BX      R0
                 ENDP
```

图 1-4

二、Reset_Handler 段分析

1. _systeminit 函数分析

STM32 上电启动，首先从 0x0000 0000 处初始化 sp 的值，然后从 0x0000 0004 处取得复位中断处理的地址 0x0800 1F6D，程序跳转

```
                Reset_Handler:
0x08001F6C 4809      LDR      r0,[pc,#36]   ; @0x08001F94
```

图 2-1

但是 Reset_Handler 的地址为 0x0800 1F6C，这是因为 Cortex-M3 使用的是 thumb-2 指令集，其最低位必须为 1；如果为 0，则会出现异常，如图 2-1 所示。

查看反汇编窗口，如图 2-2 所示：

```
                Reset_Handler:
0x08001F6C 4809      LDR      r0,[pc,#36]   ; @0x08001F94
0x08001F6E 4780      BLX      r0
0x08001F70 4809      LDR      r0,[pc,#36]   ; @0x08001F98
0x08001F72 4700      BX       r0
```

图 2-2

可以知道先取得 SystemInit 函数的地址，该地址是多少，我们可以跳转到 0x08001F94 看看，如下所示，该地址保存了值为 0x0800045D 的数，如图 2-3：

```
0x08001F94 045D      DCW      0x045D
0x08001F96 0800      DCW      0x0800
0x08001F98 0121      DCW      0x0121
0x08001F9A 0800      DCW      0x0800
```

图 2-3

然后我们跳转到 0x0800045D，发现该处正是我们需要的 SystemInit 函数入口地址，如图 2-4：

```
                SystemInit:
0x0800045C B510      PUSH     {r4,lr}
0x0800045E 4836      LDR      r0,[pc,#216]  ; @0x08000538
0x08000460 6800      LDR      r0,[r0,#0x00]
0x08000462 F0400001  ORR      r0,r0,#0x01
0x08000466 4934      LDR      r1,[pc,#208]  ; @0x08000538
```

图 2-4

该函数首先保存跳转前的有关状态，然后根据使用的芯片，进行相应的初始化操作，函数最后重新映射了中断向量的存放地址，如图 2-5：

```
#ifdef VECT_TAB_SRAM
  SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; /* Vector Table Relo
#else
  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Rel
#endif
```

图 2-5

查看反汇编窗口，如图 2-6：

```
0x080004A4 F04F6000   MOV      r0,#0x8000000
0x080004A8 4926       LDR      r1,[pc,#152]   ; @0x08000544
0x080004AA 6008       STR      r0,[r1,#0x00]
0x080004AC BD10       POP      {r4,pc}
```

图 2-6

根据分析可知，该段代码是把 0x0800 0000 存放到地址为 0xE000 ED08 处，查找 Cortex_M3 手册如图 2-7 所示，该地址是向量表偏移寄存器，也就是说，这条语句把中断向量表重新映射到地址为 0x0800 0000。

Vector Table Offset Register

Use the Vector Table Offset Register to determine:

- if the vector table is in RAM or code memory
- the vector table offset.

The register address, access type, and Reset state are:

Address	0xE000ED08
Access	Read/write
Reset state	0x00000000

图 2-7

2. _main 函数分析

进行完相应的初始化，函数跳转到_main 函数，_main 函数的入口地址从 0x08001F98 取得，通过查找，发现_main 函数的地址为 0x08000121,跳转到 0x0800120,证明此处就是我们要找的_main 函数入口，如图 2-8

```
                __main:
0x08000120 F000F802   BL.W     __scatterload (0x08000128)
0x08000124 F000F83A   BL.W     __rt_entry (0x0800019C)
                __scatterload:
0x08000128 A00A       ADR      r0,{pc}+4   ; @0x08000154
0x0800012A E8900C00   LDM      r0,{r10-r11}
0x0800012E 4482       ADD      r10,r10,r0
0x08000130 4483       ADD      r11,r11,r0
0x08000132 F1AA0701   SUB      r7,r10,#0x01
                scatterload null:
```

图 2-8

_main 函数到底进行了哪些操作呢，下面我们就一一逐步分析过去。

◆ _scatterload 函数分析

首先是一条跳转指令，跳转到_scatterload 函数，该函数的第一条指令是把地址 0x08000154 赋值给 r0（但是此处 PC+4 取得值应该是 0x0800012C，为什么会是 0x08000154 呢？根据 ARMv7-M Architecture Reference Manual，ADR 的二进制码，相差 0x28），第二条指令是分别把 0x08000154、0x08000158 存放的 0x0000 1ECC、0x0000 1EEC 给 r10、r11，如图 2-9。经过该函数的第三、第四条指令，我们可以得到 r10=0x08002020,r11=0x08002040,r7=0x0800201F，

```
0x08000154 1ECC      DCW      0x1ECC
0x08000156 0000      DCW      0x0000
0x08000158 1EEC      DCW      0x1EEC
0x0800015A 0000      DCW      0x0000
```

图 2-9

查看.map 文件，知道 0x08002020 称为 Region$$Table$$Base，0x08002040 称为 Region$$Table$$Limit。

◆ _scatterload_null 函数分析

如图 2-10 所示，程序继续执行到_scatterload_null 函数，首先比较 r10、r11 是否相等，如果不等则跳转到 0x0800013E。很明显不等，程序跳转，第一条指令是把 0x08000137f 赋值给 lr，其实就是保存_scatterload_null 的入口地址；第二条指令是把 r10=0x08002020 为起始，0x08002030 为终止的地址内容分别赋值给 r0-r3，最后我们得到 r0=0x08002040, r1=0x20000000, r2=0x00000034, r3=0x0800 015C,r10=0x08002030；第三条指令是判断 r3 是否为 1，很明显不为 1，IT 指令等效于 if-then 模式，最后几条指令可以得到 r3=0x0800 015D，程序跳转

```
                __scatterload_null:
0x08000136 45DA      CMP      r10,r11
0x08000138 D101      BNE      0x0800013E
0x0800013A F000F82F  BL.W     __rt_entry (0x0800019C)
0x0800013E F2AF0E09  ADR.W    lr,{pc}-0x07  ; @0x08000137
0x08000142 E8BA000F  LDM      r10!,{r0-r3}
0x08000146 F0130F01  TST      r3,#0x01
0x0800014A BF18      IT       NE
0x0800014C 1AFB      SUBS     r3,r7,r3
0x0800014E F0430301  ORR      r3,r3,#0x01
0x08000152 4718      BX       r3
```

图 2-10

◆ MAP 文件分析

0x0800 015D 地址是函数_scatterload_copy 的入口，该函数到底 copy 了什么值呢？在此之前我们先要熟悉一下.map 文件

.map 文件是值包括了映像文件信息图和其它信息的一个映射文件，该文件包含了：

(1) 从映像文件中删除的输入段中未使用段的统计信息，对应参数-remove;

(2) 域符号映射和全局、局部符号及生成符号映射统计信息，对应参数 -symbol;

(3) 映射文件的信息图，对应参数-map,该信息中包含映像文件中的每个加载域、运行域和输入段的大小和地址，如 2-11 图、2-12 图所示：

```
Memory Map of the image

  Image Entry point : 0x08000121

  Load Region LR_IROM1 (Base: 0x08000000, Size: 0x00002074, Max: 0x00010000, ABSOLUTE)

    Execution Region ER_IROM1 (Base: 0x08000000, Size: 0x00002040, Max: 0x00010000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x08000000   0x00000120   Data   RO          340    RESET               startup stm32f10x md vl.o
```

图 2-11

```
Execution Region RW_IRAM1 (Base: 0x20000000, Size: 0x00000698, Max: 0x00005000, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x20000000   0x00000004   Data   RW          2    .data               stm32f10x_it.o
0x20000004   0x00000005   Data   RW        108    .data               main.o
```

图 2-12

(4) 映像文件的每个输入文件或库的 RO、RW、ZI 等统计信息，对应参数-info sizes，示例如图 2-13：

```
--------------------------------------------------------------------------------

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

   266         16          0          0         96        576   c_w.l

--------------------------------------------------------------------------------
   272         16          0          0        100        576   Library Totals

--------------------------------------------------------------------------------
```

图 2-13

(5) 文件对象类和库总的 RO、RW、ZI 等下大小，对应参数-info totals,示例如图 2-14：

```
Total RO  Size (Code + RO Data)                 8256 (   8.06kB)
Total RW  Size (RW Data + ZI Data)              1688 (   1.65kB)
Total ROM Size (Code + RO Data + RW Data)       8308 (   8.11kB)
```

图 2-14

由此，根据.map 文件我们得到

RW=0x34,
ZI=0x0644,
Code+RO=0x2040,
RW+ZI=0x0698

现在我们回到_scatterload_copy 中，看看到底 copy 了什么，其代码如图 2-15 所示：

```
                __scatterload_copy:
0x0800015C 3A10      SUBS     r2,r2,#0x10
0x0800015E BF24      ITT      CS
0x08000160 C878      LDM      r0!,{r3-r6}
0x08000162 C178      STM      r1!,{r3-r6}
0x08000164 D8FA      BHI      __scatterload_copy (0x0800015C)
0x08000166 0752      LSLS     r2,r2,#29
0x08000168 BF24      ITT      CS
0x0800016A C830      LDM      r0!,{r4-r5}
0x0800016C C130      STM      r1!,{r4-r5}
0x0800016E BF44      ITT      MI
0x08000170 6804      LDR      r4,[r0,#0x00]
0x08000172 600C      STR      r4,[r1,#0x00]
0x08000174 4770      BX       lr
0x08000176 0000      MOVS     r0,r0
```

图 2-15

◆ _scatterload_copy 函数分析

经过_scatter_null 函数我们得到 r2=0x00000034，这正是需要初始化全局变量的大小，由此我们猜测 copy 的值是全局变量。

第一条指令是得 r2=0x0000 0024,然后判断标志位 C 是否为 1，如果是 1，则执行下面两条语句:

```
LDM     r0! , (r3-r6);
STM     r1! , (r3-r6);
```

r0=0x0800 2040, r1=0x2000 0000,而 0x20000000 正好是 SRAM 的起始地址，所以上面两条语句是把以 0x8002040 为起始的地址复制到以 0x2000 0000 为起始的地址，该循环类似我们的 for(r0=0x0800 2040, r1=0x2000 0000;r2=r2-0x10;r2>0), 直到 r2 为负数

```
LSLS    r2,r2,#29
ITT     CS              如果 r2 除以 16 是 8 的整数，则复制该 8 字节
LDM     r0!,{r4-r5}
STM     r1!,{r4-r5}
ITT     MI              如果 r2 除以 16 是 4 的整数，则复制 4 或者 12 字节
LDR     r4,[r0,#0x00]
STR     r4,[r1,#0x00]
```

这样_scatterload_copy 完成了需要初始化全局变量 RW 的装载过程，最后函数返回到_scatterload_null。

◆ _scatterload_zeroinit 函数分析

然后判断 r10 是否与 r11 相等，很明显不等,r3=0x0800 0178,函数跳转，进入到_scatterload_zeroinit 函数，此时

r0=0x0800 2074,r1=0x20000034, r2=0x0000 0664

_scatterload_zeroinit 函数代码如图 2-16：

```
                __scatterload_zeroinit:
0x08000178 2300      MOVS     r3,#0x00
0x0800017A 2400      MOVS     r4,#0x00
0x0800017C 2500      MOVS     r5,#0x00
0x0800017E 2600      MOVS     r6,#0x00
0x08000180 3A10      SUBS     r2,r2,#0x10
0x08000182 BF28      IT       CS
0x08000184 C178      STM      r1!,{r3-r6}
0x08000186 D8FB      BHI      0x08000180
0x08000188 0752      LSLS     r2,r2,#29
0x0800018A BF28      IT       CS
0x0800018C C130      STM      r1!,{r4-r5}
0x0800018E BF48      IT       MI
0x08000190 600B      STR      r3,[r1,#0x00]
0x08000192 4770      BX       lr
```

图 2-16

通过_scatterload_copy 我们可以猜测_scatterload_zeroinit 是一个清零过程，但是对什么需要清零呢？当然是 ZI 段，由 r2=0x00000664 这正是 ZI 的大小，所以该过程是以 0x20000034 为起始地址，大小为 r2=0x00000664 的清零过程，具体分析和_scatterload_copy 类似，不再重复。

程序最后返回到_scatterload，接着跳转到_rt_entry,如图 2-17

```
                __main:
0x08000120 F000F802  BL.W      __scatterload (0x08000128)
0x08000124 F000F83A  BL.W      __rt_entry (0x0800019C)
```

图 2-17

2. _rt_entry 函数分析

_rt_entry 的第一条指令又是一条跳转指令，程序再次跳转到_user_setup_stackheap，如图 2-18

```
                __user_setup_stackheap:
0x08001FB2 4675      MOV       r5,lr
0x08001FB4 F000F828  BL.W      __user_libspace (0x08002008)
0x08001FB8 46AE      MOV       lr,r5
0x08001FBA 0005      MOVS      r5,r0
0x08001FBC 4669      MOV       r1,sp
0x08001FBE 4653      MOV       r3,r10
0x08001FC0 F0200007  BIC       r0,r0,#0x07
0x08001FC4 4685      MOV       sp,r0
0x08001FC6 B018      ADD       sp,sp,#0x60
0x08001FC8 B520      PUSH      {r5,lr}
0x08001FCA F7FFFFDD  BL.W      __user_initial_stackheap (0x08001F88)
0x08001FCE E8BD4020  POP       {r5,lr}
0x08001FD2 F04F0600  MOV       r6,#0x00
0x08001FD6 F04F0700  MOV       r7,#0x00
0x08001FDA F04F0800  MOV       r8,#0x00
0x08001FDE F04F0B00  MOV       r11,#0x00
0x08001FE2 F0210107  BIC       r1,r1,#0x07
0x08001FE6 46AC      MOV       r12,r5
0x08001FE8 E8AC09C0  STM       r12!,{r6-r8,r11}
0x08001FEC E8AC09C0  STM       r12!,{r6-r8,r11}
0x08001FF0 E8AC09C0  STM       r12!,{r6-r8,r11}
```

图 2-18

_user_setup_stackheap 函数的第一条指令是保存函数的返回地址，此处为什么没有用 PUSH？因为此时堆栈还没有初始化好。第二条指令是跳转到_user_libspace 进行一些微库的初始化工作，后面的几条语句是建立一个大小为 90 字节的临时栈，然后程序跳转到_user_inital_stackheap 进行用户栈的初始化，这也就是启动文件 startup_stm32f10x_md_vl 中初始化堆栈段的那些语句，如图 2-19

```
__user_initial_stackheap

                LDR     R0, =  Heap_Mem
                LDR     R1, =(Stack_Mem + Stack_Size)
                LDR     R2, = (Heap_Mem +  Heap_Size)
                LDR     R3, = Stack_Mem
                BX      LR

                ALIGN
```

图 2-19

经过该函数处理得：r0=0x2000 0098,r1=0x2000 0698,r2=0x2000 0298,r3=0x2000 0298。最后用户栈顶被设置成 0x2000 0698，完成了堆栈的初始化工作，程序返回到 rt_entry_main

```
                __rt_entry_main:
0x080001A6 F000F829  BL.W     main (0x080001FC)
0x080001AA F001FF27  BL.W     exit (0x08001FFC)
```

三、 总结

最终函数终于跳转到我们的 main 函数执行我们写的代码。

总结启动文件的整个过程，分为如下：

（1） 系统初始化，包括对中断向量表的重新映射；
（2） 加载 RW 段；
（3） ZI 段清零；
（4） 初始化用户堆栈；
（5） 初始化微库（具体干什么我也不知道，屏蔽此处函数好像也能正常运行）；
（6） 调用 main 函数。

错误之处：敬请指正，邮箱 duanzhipingysu2011@gmail.com,谢谢！