Technical Note

STM32 启动文件浅析

正点原子团队编写

技术文档

修订历史

版本	日期	原因
V1.0	2020/4/27	第一次发布
V1.1	2020/6/15	改善系统启动流程小节的描述
V1.2	2021/8/27	修改图 3.3 地址错误

目录

1. STM32 启动文件简介

STM32 启动文件由 ST 官方提供，在官方的固件包里。启动文件由汇编编写，是系统上电复位后第一个执行的程序。

启动文件主要做了以下工作：

1、初始化堆栈指针 SP = _initial_sp
2、初始化程序计数器指针 PC = Reset_Handler
3、设置堆和栈的大小
4、初始化中断向量表
5、配置外部 SRAM 作为数据存储器（可选）
6、配置系统时钟，通过调用 SystemInit 函数（可选）
7、调用 C 库中的 _main 函数初始化用户堆栈，最终调用 main 函数

1.1 启动文件中的一些指令

指令名称	作用
EQU	给数字常量取一个符号名，相当于 C 语言中的 define
AREA	汇编一个新的代码段或者数据段
ALIGN	编译器对指令或者数据的存放地址进行对齐，一般需要跟一个立即数，缺省表示 4 字节对齐。要注意的是，这个不是 ARM 的指令，是编译器的，这里放到一起为了方便。
SPACE	分配内存空间
PRESERVE8	当前文件堆栈需要按照 8 字节对齐
THUMB	表示后面指令兼容 THUMB 指令。在 ARM 以前的指令集中有 16 位的 THUMBM 指令，现在 Cortex-M 系列使用的都是 THUMB-2 指令集，THUMB-2 是 32 位的，兼容 16 位和 32 位的指令，是 THUMB 的超级版。
EXPORT	声明一个标号具有全局属性，可被外部的文件使用
DCD	以字节为单位分配内存，要求 4 字节对齐，并要求初始化这些内存
PROC	定义子程序，与 **ENDP** 成对使用，表示子程序结束
WEAK	弱定义，如果外部文件声明了一个标号，则优先使用外部文件定义的标号，如果外部文件没有定义也不会出错。要注意的是，这个不是 ARM 的指令，是编译器的，这里放到一起为了方便。
IMPORT	声明标号来自外部文件，跟 C 语言中的 extern 关键字类似
LDR	从存储器中加载字到一个存储器中
BLX	跳转到由寄存器给出的地址，并根据寄存器的 LSE 确定处理器的状态，还要把跳转前的下条指令地址保存到 LR
BX	跳转到由寄存器/标号给出的地址，不用返回
B	跳转到一个标号
IF,ELSE,ENDIF	汇编条件分支语句，跟 C 语言的类似
END	到达文件的末尾，文件结束

表 1.1.1 启动文件的汇编指令

上表，列举了 STM32 启动文件的一些汇编和编译器指令，关于其他更多的 ARM 汇编

指令，我们可以通过 MDK 的索引搜索工具中搜索找到。打开索引搜索工具的方法：MDK->Help->uVision Help，如图 1.1.1 所示。

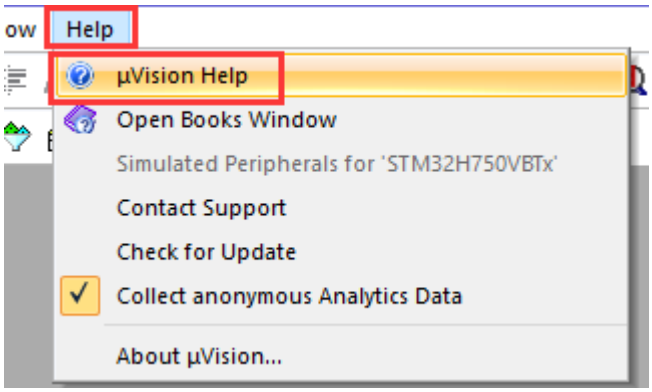

图 1.1.1 打开索引搜索工具的方法

打开之后，我们以 **EQU** 为例，演示一下怎么使用，如图 1.1.2 所示。

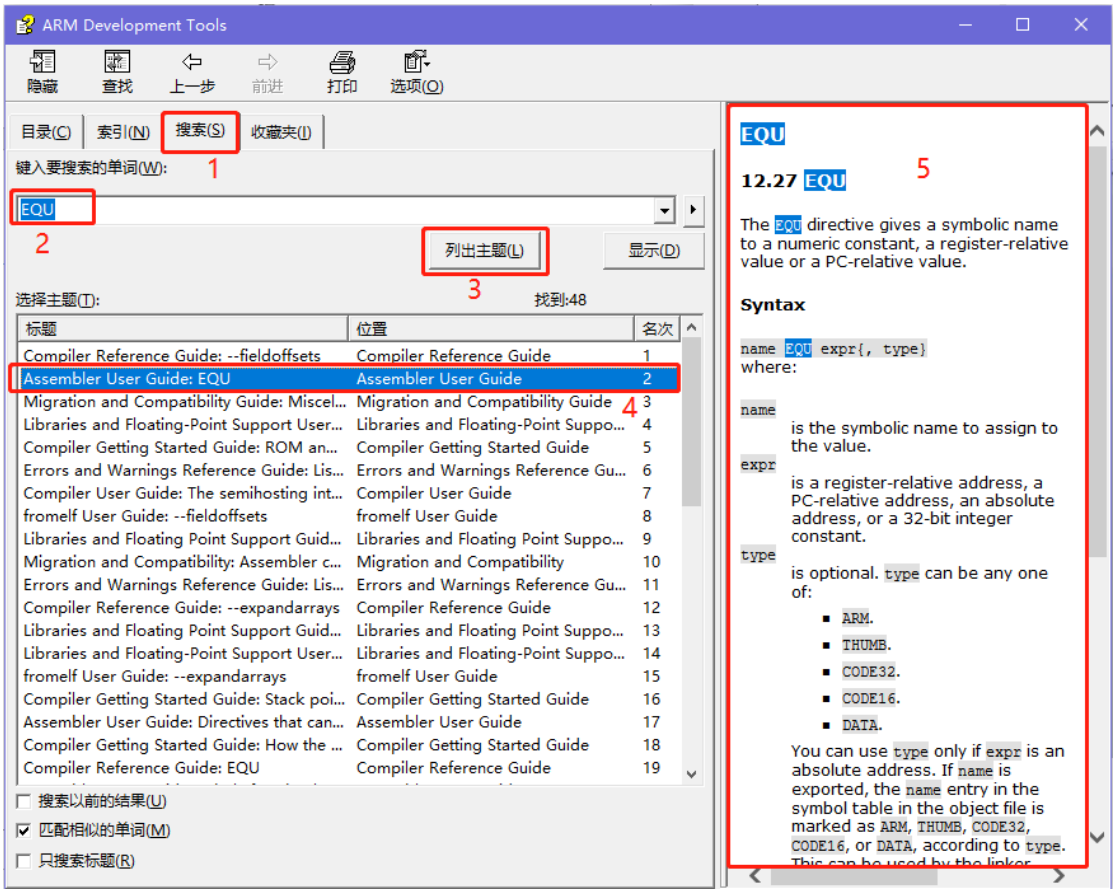

图 1.1.2 搜索 EQU 汇编指令

搜索到的标题有很多，我们只需要看 Assembler User Guide 这部分即可。

2. 启动文件代码详解

下面，我们以 STM32F103 的启动代码为例讲解，版本是：STM32Cube_FW_F1_V1.8.0，启动文件名称是：startup_stm32f103xe.s。把启动代码分成几个功能段进行详细的讲解，详情如下。

2.1 栈空间的开辟

栈空间的开辟，源码如图 2.1.1 所示：

```
33  Stack_Size      EQU     0x00000400
34
35                  AREA    STACK, NOINIT, READWRITE, ALIGN=3
36  Stack_Mem       SPACE   Stack_Size
37  __initial_sp
```

图 2.1.1 栈空间的开辟

33 行 EQU：宏定义的伪指令，给数字常量取一个符号名，类似与 C 中的 define。定义栈大小为 0x00000400 字节，即 1024B（1KB），常量的符号是 Stack_Size。

35 行 AREA 汇编一个新的代码段或者数据段。段名为 STACK，段名可以任意命名；NOINIT 表示不初始化； READWRITE 表示可读可写；ALIGN=3，表示按照 2^3 对齐，即 8 字节对齐。

36 行 SPACE 分配内存指令，分配大小为 Stack_Size 字节连续的存储单元给栈空间。

37 行__initial_sp 紧挨着 SPACE 放置，表示栈的结束地址，栈是从高往低生长，所以结束地址就是栈顶地址。

栈主要用于存放局部变量，函数形参等，属于编译器自动分配和释放的内存，栈的大小不能超过内部 SRAM 的大小。如果工程的程序量比较大，定义的局部变量比较多，那么就需要在启动代码中修改栈的大小，即修改 Stack_Size 的值。如果程序出现了莫名其妙的错误，并进入了 HardFault 的时候，你就要考虑下是不是栈空间不够大，溢出了的问题。

关于栈顶地址，我们先编译战舰开发板 HAL 库例程的实验 1 跑马灯实验工程，然后可以通过.map 文件查看，方法如图 2.1.2 所示。关于.map 文件的详细介绍，大家可以查看正点原子团队编写的：《MAP 文件浅析.pdf》这个文档 。

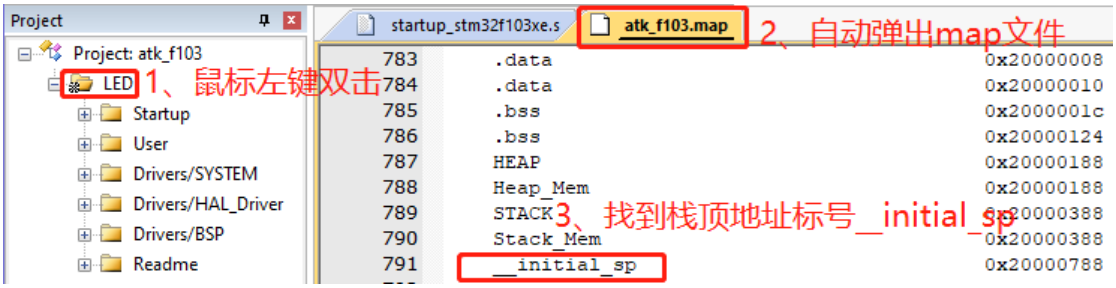

图 2.1.2 通过.map 文件查看栈顶地址

我们定义 Stack_Size 的大小是 0x00000400，图 2.1.2 中可以看到栈顶地址__initial_sp 的地址是 0x20000788，那栈低地址是多少呢？从图中可以知道是 0x20000388。所以栈顶地址 0x20000788 到栈低地址 0x20000388 的内存大小刚好就是 Stack_Size 的大小。栈是从高往低生长，所以每使用一个栈空间地址，栈顶地址__initial_sp 就减一。

2.2 堆空间的开辟

堆空间的开辟，源码如图 2.2.1 所示：

```
43  Heap_Size       EQU     0x00000200
44
45                  AREA    HEAP, NOINIT, READWRITE, ALIGN=3
46  __heap_base
47  Heap_Mem        SPACE   Heap_Size
48  __heap_limit
```

图 2.2.1 堆空间的开辟

堆空间开辟代码跟栈空间开辟代码是类似的了。这部分代码的意思就是：开辟堆的大小为 0x00000200（512 字节），段名为 HEAP，不初始化，可读可写，8 字节对齐。__heap_base 表示堆的起始地址，__heap_limit 表示堆的结束地址。堆和栈的生长方向相反的，堆是由低向高生长，而栈是从高往低生长。

堆主要用于动态内存的分配，像 malloc()、calloc()和 realloc()等函数申请的内存就在堆上面。堆中的内存一般由程序员分配和释放，若程序员不释放，程序结束时可能由操作系统回收。

接下来是 PRESERVE8 和 THUMB 指令两行代码。如图 2.2.2 所示。

```
50                  PRESERVE8
51                  THUMB
```

图 2.2.2 PRESERVE8 和 THUMB 指令

PRESERVE8：指示编译器按照 8 字节对齐。

THUMB：指示编译器之后的指令为 THUMB 指令。

注意：由于正点原子提供了独立的内存管理实现方式（mymalloc，myfree 等），并不需要使用 C 库的 malloc 和 free 等函数，也就用不到堆空间，因此我们可以设置 Heap_Size 的大小为 0，以节省内存空间。

2.3 中断向量表定义（简称：向量表）

中断向量表定义代码，如图 2.3.1 所示：

```
55                  AREA    RESET, DATA, READONLY
56                  EXPORT  __Vectors
57                  EXPORT  __Vectors_End
58                  EXPORT  __Vectors_Size
```

图 2.3.1 中断向量表定义代码

定义一个数据段，名字为 RESET, READONLY 表示只读。EXPORT 表示声明一个标号具有全局属性，可被外部的文件使用。这里是声明了__Vectors、__Vectors_End 和__Vectors_Size 三个标号具有全局性，可被外部的文件使用。

当内核响应了一个发生的异常后，对应的异常服务例程(ESR)就会执行。为了决定 ESR 的入口地址， 内核使用了向量表查表机制。向量表其实是一个 WORD（32 位整数）数组，每个下标对应一种异常，该下标元素的值则是该 ESR 的入口地址。向量表在地址空间中的位置是可以设置的，通过 NVIC 中的一个重定位寄存器来指出向量表的地址。在复位后，该寄存器的值为 0。因此，在地址 0 （即 FLASH 地址 0） 处必须包含一张向量表，用于初始时的异常分配。表 2.3.1 是 F103 的向量表。

位置	优先级	优先级类型	名称	说明	地址
	-	-	-	保留	0x0000_0000
	-3	固定	Reset	复位	0x0000_0004
	-2	固定	NMI	不可屏蔽中断 RCC时钟安全系统(CSS)联接到NMI向量	0x0000_0008
	-1	固定	硬件失效(HardFault)	所有类型的失效	0x0000_000C
	0	可设置	存储管理(MemManage)	存储器管理	0x0000_0010
	1	可设置	总线错误(BusFault)	预取指失败，存储器访问失败	0x0000_0014
	2	可设置	错误应用(UsageFault)	未定义的指令或非法状态	0x0000_0018
	-	-	-	保留	0x0000_001C ~0x0000_002B
	3	可设置	SVCall	通过SWI指令的系统服务调用	0x0000_002C
	4	可设置	调试监控(DebugMonitor)	调试监控器	0x0000_0030
	-	-	-	保留	0x0000_0034
	5	可设置	PendSV	可挂起的系统服务	0x0000_0038
	6	可设置	SysTick	系统嘀嗒定时器	0x0000_003C
0	7	可设置	WWDG	窗口定时器中断	0x0000_0040
1	8	可设置	PVD	连到EXTI的电源电压检测(PVD)中断	0x0000_0044
2	9	可设置	TAMPER	侵入检测中断	0x0000_0048
3	10	可设置	RTC	实时时钟(RTC)全局中断	0x0000_004C
4	11	可设置	FLASH	闪存全局中断	0x0000_0050
中间部分省略，详细请参考《STM32中文参考手册》第九章 中断和事件 中断和异常向量					
56	63	可设置	DMA2通道1	DMA2通道1全局中断	0x0000_0120
57	64	可设置	DMA2通道2	DMA2通道2全局中断	0x0000_0124
58	65	可设置	DMA2通道3	DMA2通道3全局中断	0x0000_0128
59	66	可设置	DMA2通道4_5	DMA2通道4和DMA2通道5全局中断	0x0000_012C

表 2.3.1 F103 的向量表

举个例子，如果发生了异常 SVCall，则 NVIC 会计算出偏移移量是 11x4=0x2C，然后从那里取出服务例程的入口地址并跳入。要注意的是这里有个另类：地址 0x0000 0000 并不是什么入口地址，而是给出了复位后 MSP 的初值。更详细的向量表，可以参考《STM32 中文参考手册》第九章-中断和事件-中断和异常向量。

F103 的向量表格中灰色部分是系统内核异常。表格中位置 0 到 59 是外部中断，CM3 内核的芯片最大支持 240 个外部中断，具体使用多少个由芯片厂家设计决定。如这个表格中的 103 芯片只是使用了 60 个。这里说的外部中断是相对内核而言。

代码中的中断向量表是与 F103 的向量表对应的，如图 2.3.2 所示。

```
60  __Vectors       DCD     __initial_sp               ; Top of Stack   （栈顶地址）
61                  DCD     Reset_Handler              ; Reset Handler  （复位程序地址）
62                  DCD     NMI_Handler                ; NMI Handler
63                  DCD     HardFault_Handler          ; Hard Fault Handler
64                  DCD     MemManage_Handler          ; MPU Fault Handler
65                  DCD     BusFault_Handler           ; Bus Fault Handler
66                  DCD     UsageFault_Handler         ; Usage Fault Handler
67                  DCD     0                          ; Reserved
68                  DCD     0                          ; Reserved
69                  DCD     0                          ; Reserved
70                  DCD     0                          ; Reserved
71                  DCD     SVC_Handler                ; SVCall Handler
72                  DCD     DebugMon_Handler           ; Debug Monitor Handler
73                  DCD     0                          ; Reserved
74                  DCD     PendSV_Handler             ; PendSV Handler
75                  DCD     SysTick_Handler            ; SysTick Handler
76
77                  ; External Interrupts （外部中断）
78                  DCD     WWDG_IRQHandler            ; Window Watchdog
79                  DCD     PVD_IRQHandler             ; PVD through EXTI Line detect
80                  DCD     TAMPER_IRQHandler          ; Tamper
81                  DCD     RTC_IRQHandler             ; RTC
82                  DCD     FLASH_IRQHandler           ; Flash
83                  ;中间篇幅太长，省略掉，代码向量表与STM32F103的向量表对应
84                  DCD     DMA2_Channel1_IRQHandler   ; DMA2 Channel1
85                  DCD     DMA2_Channel2_IRQHandler   ; DMA2 Channel2
86                  DCD     DMA2_Channel3_IRQHandler   ; DMA2 Channel3
87                  DCD     DMA2_Channel4_5_IRQHandler ; DMA2 Channel4 & Channel5
88  __Vectors_End
89
90  __Vectors_Size  EQU   __Vectors_End - __Vectors
```

图 2.3.2 中断向量表定义代码

__Vectors 为向量表起始地址， __Vectors_End 为向量表结束地址，__Vectors_Size 为向量表大小， __Vectors_Size = __Vectors_End - __Vectors。

DCD：分配一个或者多个以字为单位的内存，以四字节对齐，并要求初始化这些内存。

中断向量表被放置在代码段的最前面。例如：当我们的程序在 FLASH 运行时，那么向量表的起始地址是：0x0800 0000。结合图 2.3.2 可以知道，地址 0x0800 0000 存放的是栈顶地址。DCD：以四字节对齐分配内存，也就是下个地址是 0x0800 0004，存放的是 Reset_Handler 中断函数入口地址。

从代码上看，向量表中存放的都是中断服务函数的函数名，所以 C 语言中的函数名对芯片来说实际上就是一个地址。

2.4 复位程序

接下来是定义只读代码段，如图 2.4.1 所示：

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

图 2.4.1 定义只读代码段

定义一个段命为.text，只读的代码段，在 CODE 区。

复位子程序代码，如图 2.4.2 所示：

```
144  ; Reset handler
145  Reset_Handler    PROC
146                   EXPORT  Reset_Handler             [WEAK]
147                   IMPORT  __main
148                   IMPORT  SystemInit
149                   LDR     R0, =SystemInit
150                   BLX     R0
151                   LDR     R0, =__main
152                   BX      R0
153                   ENDP
```

图 2.4.2 复位子程序代码

利用 PROC、ENDP 这一对伪指令把程序段分为若干个过程，使程序的结构加清晰。

145 行子程序开始

146 行声明复位中断向量 Reset_Handler 为全局属性，这样外部文件就可以调用此复位中断服务。WEAK：表示弱定义，如果外部文件优先定义了该标号则首先引用外部定义的标号，如果外部文件没有声明也不会出错。这里表示复位子程序可以由用户在其他文件重新实现，这里并不是唯一的。

147 行和 148 行 IMPORT 表示该标号来自外部文件。这里表示 SystemInit 和__main 这两个函数均来自外部的文件。

149 行 LDR 表示从存储器中加载字到一个存储器中。SystemInit 是一个标准的库函数，在 system_stm32f1xx.c 文件中定义，主要作用是配置系统时钟、还有就是初始化 FSMC/FMC 总线上外挂的 SRAM(可选)，前面说配置外部 SRAM 作为数据存储器（可选）就是这个。

150 行 BLX 表示跳转到由寄存器给出的地址，并根据寄存器的 LSE 确定处理器的状态，还要把跳转前的下条指令地址保存到 LR。

151 行把__main 的地址给 R0。__main 是一个标准的 C 库函数，主要作用是初始化用户堆栈和变量等，最终调用 main 函数去到 C 的世界。这就是为什么我们写的程序都有一个 main 函数的原因，如果不调用__main，那么程序最终就不会调用我们 C 文件里面的 main，也就无法正常运行。

152 行 BX 表示跳转到由寄存器/标号给出的地址，不用返回。这里表示切换到__main 地址，最终调用 main 函数，不返回，进入 C 的世界。

153 行 ENDP 表示子程序结束。

LDR、BLX、BX 是内核的指令，可在《CM3 权威指南 CnR2》第四章-指令集里面查询到。

2.4.1 对于 weak 的理解

weak 顾名思义是“弱”的意思，在汇编中，在函数名称后面加[WEAK]来表示，而在 C 语言中，在函数名称前面加上__weak 修饰符来表示，这样的函数我们称为“弱函数”。

被[WEAK]或__weak 声明的函数，我们可以在自己的文件中重新定义一个同名函数，最终编译器编译的时候，会选择我们定义的函数，如果我们没有重新定义这个函数，

那么编译器就会执行[WEAK]或__weak 声明的函数，并且编译器不会报错。

举个例子：打开战舰开发板 HAL 库例程实验 1 跑马灯实验。我们用 HardFault_Handler 中断函数举例。在启动文件的 161 行到 165 行，定义了 HardFault_Handler 中断函数，且声明为“弱函数”，如图 2.4.1.1 所示。

```
161  HardFault_Handler\
162                  PROC
163                  EXPORT  HardFault_Handler          [WEAK]
164                  B       .
165                  ENDP
```

图 2.4.1.1 HardFault_Handler 中断函数（弱定义）

同样我们打开 stm32f1xx_it.c 文件的 60 行到 66 行也定义了 HardFault_Handler 中断函数，如图 2.4.1.2 所示。

```
60  void HardFault_Handler(void)
61  {
62    /* Go to infinite loop when Hard Fault exception occurs */
63    while (1)
64    {
65    }
66  }
```

图 2.4.1.2 HardFault_Handler 中断函数

在 stm32f1xx_it.c 文件定义了 HardFault_Handler 中断函数的情况下，当 HardFault_Handler 中断来到的时候，代码会运行到 stm32f1xx_it.c 文件的 HardFault_Handler 中断函数，且进入 while(1)。

下面，我们注释掉 stm32f1xx_it.c 的 HardFault_Handler 中断函数，然后进行编译，发现不会报错。这时候当 HardFault_Handler 中断来到的时候，代码会运行到启动文件的“弱函数”中，即在启动文件中 164 行代码，进行原地跳转（即无限循环） 。

2.4.2 对于_main 函数的分析

当看到__main 函数时，估计有不少人认为这个是 main 函数的别名或是编译之后的名字，否则在启动代码中再也无法找到和 main 相关的字眼了。可事实是，_main 和 main 是两个完全不同的函数。_main 代码是编译器自动创建的，因此无法找到_main 代码。MDK 文档中有一句说明：it is automatically craated by the linker when it sees a definition of main() 。大体意思可以理解为：当编译器发现定义了 main 函数，那么就会自动创建_main。

程序经过汇编启动代码，执行到__main()后，可以看出有两个大的函数：

__scatterload()：负责把 RW/RO 输出段从装载域地址复制到运行域地址，并完成了 ZI 运行域的初始化工作。

__rt_entry():负责初始化堆栈，完成库函数的初始化，最后自动跳转向 main()函数。

我们以战舰开发板 HAL 库例程的实验 1 跑马灯实验为例，进行仿真，在汇编窗口中运行__scatterload()函数进行分析，如图 2.4.2.1 所示。

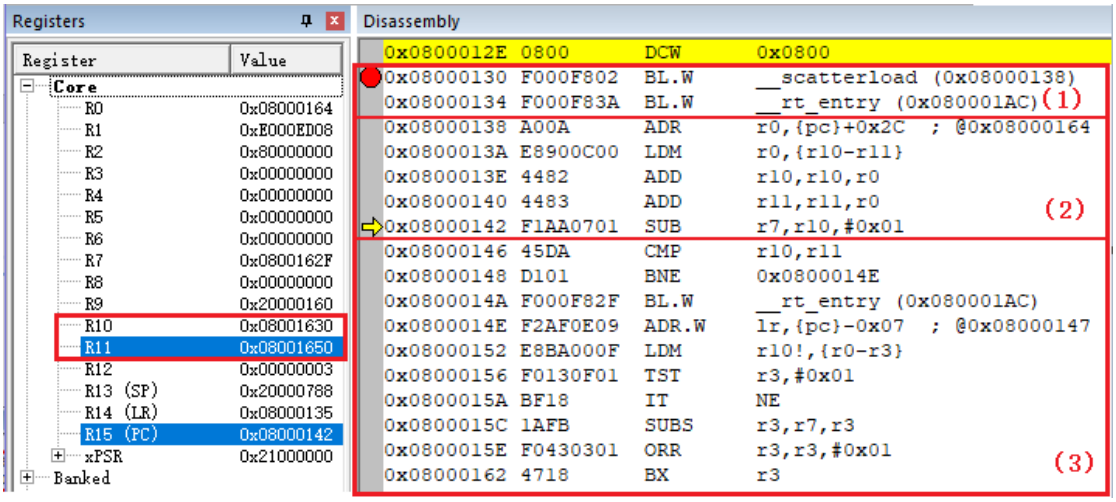

图 2.4.2.1 __scatterload()函数运行分析

图 2.4.2.1 中，(1)段是__main 函数，(2)段是__scatterload 函数，(3)段是__scatterload_null 函数。当程序运行到__main 函数，先跳转到__scatterload 函数运行，执行完__scatterload 函数后，R10 和 R11 会被赋值，如图 2.4.2.1 所示。可以通过.map 文件找到对应的 symbol，如图 2.4.2.2 所示。

```
Region$$Table$$Base                      0x08001630   Number         0  anon$$obj.o(Region$$Table)
Region$$Table$$Limit                     0x08001650   Number         0  anon$$obj.o(Region$$Table)
```

图 2.4.2.2 .map 文件找到对应的 symbol

执行完__scatterload 函数后，就接着执行__scatterload_null 函数。__scatterload_null 函数第 1、2 行比较 r10、r11 是否相等，如果不等则跳转到 0x0800014E。明显两个值不等，所以程序跳转到 0x0800014E，第 4 行是把 0x08000147 赋值给 lr，即是保存_scatterload_null 的入口地址；第 5 行是把 r10 对应地址存放的 4 个字复制到 r0-r3 中，执行后得到 r0=0x08001650， r1=0x20000000， r2=0x0000001C， r3=0x0800016C，r10=0x08001640，如图 2.4.2.3 所示。

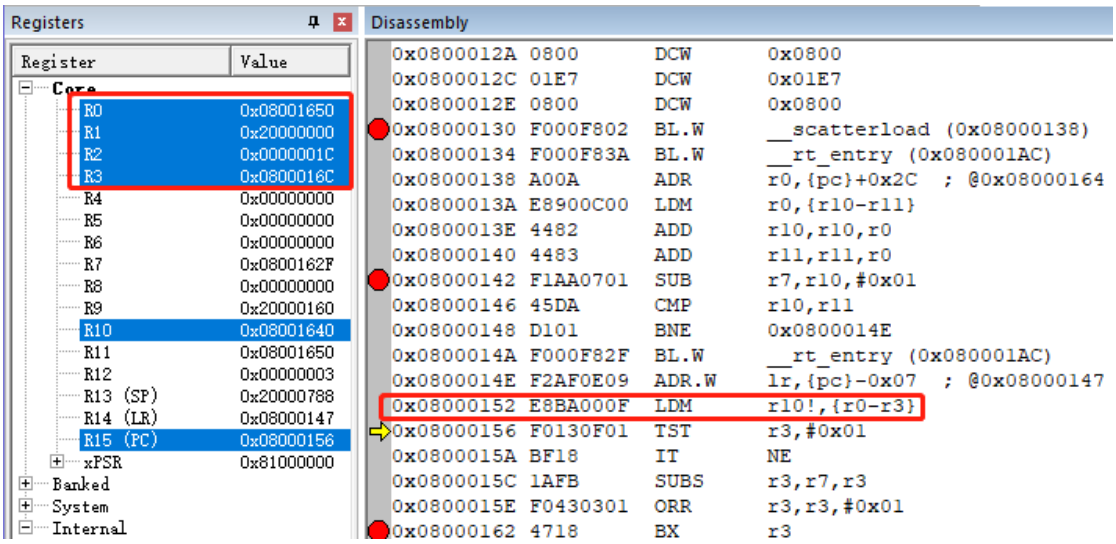

图 2.4.2.3 r10 对应地址存放的 4 个字复制到 r0-r3

R0: 0x08001650 表示的是加载域起始地址。

R1: 0x20000000 为运行域地址。

R2: 0x0000001C 为要复制的 RW Data 大小，也可以在 map 文件查找得知。

R3: 0x0800016C 是_scatterload_copy 函数的起始地址。

通过__scatterload_null 函数的最后一行跳转到_scatterload_copy 函数，其代码如图 2.4.2.4 所示。

```
0x0800016C 3A10      SUBS     r2,r2,#0x10
0x0800016E BF24      ITT      CS
0x08000170 C878      LDM      r0!,{r3-r6}
0x08000172 C178      STM      r1!,{r3-r6}
0x08000174 D8FA      BHI      __scatterload_copy (0x0800016C)
0x08000176 0752      LSLS     r2,r2,#29
0x08000178 BF24      ITT      CS
0x0800017A C830      LDM      r0!,{r4-r5}
0x0800017C C130      STM      r1!,{r4-r5}
0x0800017E BF44      ITT      MI
0x08000180 6804      LDR      r4,[r0,#0x00]
0x08000182 600C      STR      r4,[r1,#0x00]
0x08000184 4770      BX       lr
```

图 2.4.2.4 _scatterload_copy 代码

_scatterload_copy 复制好 RW Data 后，最后跳转回到__scatterload_null。回到__scatterload_null 函数后同样是先判断 r10 和 r11 是否相等，明显也是不等的，代码继续运行，最后跳转到 r3 寄存器存的地址。此时是循环回来再执行完__scatterload_null 函数后，即将进入了__scatterload_zeroinit 函数，先来看一下 r0 到 r3 的值变化。

R0: 0x0800166C 表示的是加载域结束地址。

R1: 0x2000001C 为 ZI 段的起始地址。

R2: 0x0000076C 为 ZI 段大小，即 ZI Data 大小，也可以在 map 文件查找得知。

R3: 0x08000189 是__scatterload_zeroinit 函数的起始地址。

然后就跳转到__scatterload_zeroinit 代码，如图 2.4.2.5 所示。

```
0x08000188 2300      MOVS     r3,#0x00
0x0800018A 2400      MOVS     r4,#0x00
0x0800018C 2500      MOVS     r5,#0x00
0x0800018E 2600      MOVS     r6,#0x00
0x08000190 3A10      SUBS     r2,r2,#0x10
0x08000192 BF28      IT       CS
0x08000194 C178      STM      r1!,{r3-r6}
0x08000196 D8FB      BHI      0x08000190
0x08000198 0752      LSLS     r2,r2,#29
0x0800019A BF28      IT       CS
0x0800019C C130      STM      r1!,{r4-r5}
0x0800019E BF48      IT       MI
0x080001A0 600B      STR      r3,[r1,#0x00]
0x080001A2 4770      BX       lr
```

图 2.4.2.5 __scatterload_zeroinit 代码

__scatterload_zeroinit 代码其实就是对 ZI 段清零的过程，从 ZI 段的起始地址 0x2000001C 开始，大小为 0x0000076C，进行清零操作。最后跳转回__scatterload 函数，再接着就跳转到了__rt_entry 函数。

如图 2.4.2.6，__rt_entry 函数开始就先调用__user_setup_stackheap 函数来建立堆栈。

```
                __rt_entry:
0x080001AC F000F833  BL.W     __user_setup_stackheap (0x08000216)
0x080001B0 4611      MOV      r1,r2
```

图 2.4.2.6 __rt_entry 代码

跳转到__user_setup_stackheap 函数，其代码如图 2.4.2.7 所示。

```
0x08000216 4675      MOV      r5,lr
0x08000218 F000F82C  BL.W     __user_libspace (0x08000274)
0x0800021C 46AE      MOV      lr,r5
0x0800021E 0005      MOVS     r5,r0
0x08000220 4669      MOV      r1,sp
0x08000222 4653      MOV      r3,r10
0x08000224 F0200007  BIC      r0,r0,#0x07
0x08000228 4685      MOV      sp,r0
0x0800022A B018      ADD      sp,sp,#0x60
0x0800022C B520      PUSH     {r5,lr}
0x0800022E F7FFFFDB  BL.W     __user_initial_stackheap (0x080001E8)
0x08000232 E8BD4020  POP      {r5,lr}
0x08000236 F04F0600  MOV      r6,#0x00
0x0800023A F04F0700  MOV      r7,#0x00
0x0800023E F04F0800  MOV      r8,#0x00
0x08000242 F04F0B00  MOV      r11,#0x00
0x08000246 F0210107  BIC      r1,r1,#0x07
0x0800024A 46AC      MOV      r12,r5
0x0800024C E8AC09C0  STM      r12!,{r6-r8,r11}
0x08000250 E8AC09C0  STM      r12!,{r6-r8,r11}
0x08000254 E8AC09C0  STM      r12!,{r6-r8,r11}
0x08000258 E8AC09C0  STM      r12!,{r6-r8,r11}
0x0800025C 468D      MOV      sp,r1
0x0800025E 4770      BX       lr
```

图 2.4.2.7 __user_setup_stackheap 代码

__user_setup_stackheap 函数的第一条指令是保存函数的返回地址。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临时栈，然后程序跳转到__user_inital_stackheap 进行用户栈的初始化。__user_inital_stackheap 代码如图 2.4.2.8 所示。

```
                __user_initial_stackheap:
0x080001E8 4804     LDR      r0,[pc,#16]  ; @0x080001FC
0x080001EA 4905     LDR      r1,[pc,#20]  ; @0x08000200
0x080001EC 4A05     LDR      r2,[pc,#20]  ; @0x08000204
0x080001EE 4B06     LDR      r3,[pc,#24]  ; @0x08000208
0x080001F0 4770     BX       lr
```

图 2.4.2.8 __user_inital_stackheap 代码

这段__user_inital_stackheap 代码就是启动文件中初始化堆栈的代码，如图 2.4.2.9 所示。

```
342  __user_initial_stackheap
343
344                  LDR      R0, =  Heap_Mem
345                  LDR      R1, =(Stack_Mem + Stack_Size)
346                  LDR      R2, = (Heap_Mem +  Heap_Size)
347                  LDR      R3, = Stack_Mem
348                  BX       LR
349
350                  ALIGN
```

图 2.4.2.9 启动文件的__user_inital_stackheap 代码

__user_inital_stackheap 函数执行完后得到 r0=0x20000188，r1=0x20000788，r2=0x20000388， r3=0x20000388。最后就执行完__rt_entry 代码，用户栈顶地址为设置成 0x20000788，完成了堆栈的初始化，最后就运行到__rt _entry_main，去到 C 的世界， 就这样终于执行到我们自己写的程序了。__rt _entry_main 代码如图 2.4.2.10 所示。

```
                    __rt_entry_main:
0x080001B6 F001FA01  BL.W      main (0x080015BC)
0x080001BA F000F851  BL.W      exit (0x08000260)
```

图 2.4.2.10 __rt _entry_main 代码

__main 函数就讲到这里，总的来说，__main 是编译系统提供的一个函数，负责完成库函数的初始化和初始化应用程序执行环境，最后自动跳转到 main()。

2.5 中断服务程序

接下来就是中断服务程序了，如图 2.5.1 所示：

```
156 ;系统异常中断
157 NMI_Handler     PROC
158                 EXPORT  NMI_Handler                 [WEAK]
159                 B       .             ;原地跳转（即无限循环）
160                 ENDP
161 HardFault_Handler\
162                 PROC
163                 EXPORT  HardFault_Handler           [WEAK]
164                 B       .
165                 ENDP
166 ;中间代码太长，已经省略掉
167 SysTick_Handler PROC
168                 EXPORT  SysTick_Handler             [WEAK]
169                 B       .
170                 ENDP
171 ;外部中断
172 Default_Handler PROC
173
174                 EXPORT  WWDG_IRQHandler             [WEAK]
175                 EXPORT  PVD_IRQHandler              [WEAK]
176                 EXPORT  TAMPER_IRQHandler           [WEAK]
177                 EXPORT  RTC_IRQHandler              [WEAK]
178                 EXPORT  FLASH_IRQHandler            [WEAK]
179 ;中间代码太长，已经省略掉
180 DMA2_Channel1_IRQHandler
181 DMA2_Channel2_IRQHandler
182 DMA2_Channel3_IRQHandler
183 DMA2_Channel4_5_IRQHandler
184                 B       .
185
186                 ENDP
```

图 2.5.1 中断服务程序

可以看到这些中断服务函数都被[WEAK]声明为弱定义函数，如果外部文件声明了一个标号，则优先使用外部文件定义的标号，如果外部文件没有定义也不会出错。

这些中断函数分为系统异常中断和外部中断，外部中断根据不同芯片有所变化。B 指令是跳转到一个标号，这里跳转到一个‘.’，表示无限循环。

在启动文件代码中，已经把我们所有中断的中断服务函数写好了，但都是声明为弱定义，所以真正的中断服务函数需要我们在外部实现。

如果我们开启了某个中断，但是忘记写对应的中断服务程序函数又或者把中断服务函数名写错，那么中断发生时，程序就会跳转到启动文件预先写好的弱定义的中断服务程序中，

并且在 B 指令作用下跳转到一个‘.’中，无限循环。

这里的系统异常中断是内核的，外部中断是外设的。

2.6 用户堆栈初始化

ALIGN 指令，如图 2.6.1 所示：

```
326                             ALIGN
```

图 2.6.1 ALIGN 指令

ALIGN 表示对指令或者数据的存放地址进行对齐，一般需要跟一个立即数，缺省表示 4 字节对齐。要注意的是，这个不是 ARM 的指令，是编译器的。

接下就是启动文件最后一部分代码，用户堆栈初始化代码，如图 2.6.2 所示：

```
331                     IF      :DEF:__MICROLIB
332
333                     EXPORT  __initial_sp
334                     EXPORT  __heap_base
335                     EXPORT  __heap_limit
336
337                     ELSE
338
339                     IMPORT  __use_two_region_memory
340                     EXPORT  __user_initial_stackheap
341                     |
342 __user_initial_stackheap
343
344                     LDR     R0, =  Heap_Mem
345                     LDR     R1, =(Stack_Mem + Stack_Size)
346                     LDR     R2, = (Heap_Mem +  Heap_Size)
347                     LDR     R3, = Stack_Mem
348                     BX      LR
349
350                     ALIGN
351
352                     ENDIF
353
354                     END
```

图 2.6.2 用户堆栈初始化代码

IF, ELSE, ENDIF 是汇编的条件分支语句。

331 行判断是否定义了__MICROLIB。关于__MICROLIB 这个宏定义，我们是在 KEIL 里面配置，具体方法如图 2.6.3 所示。

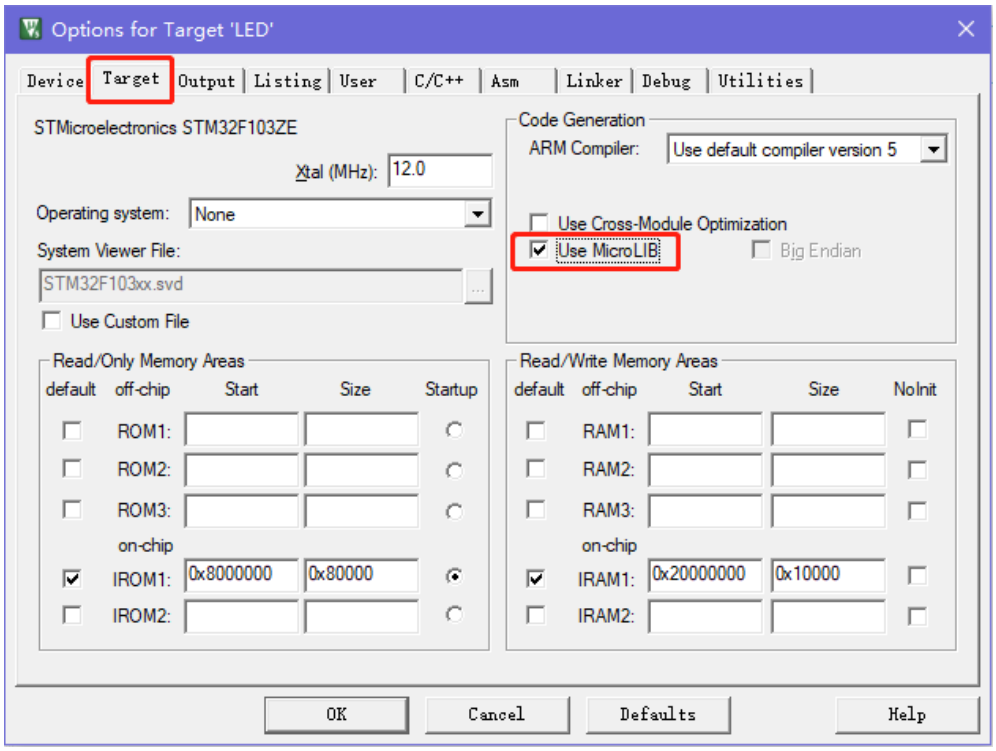

图 2.6.3 __MICROLIB 定义方法

勾选了 Use MicroLIB 就代表定义了__MICROLIB 这个宏。

333 行到 335 行如果定义__MICROLIB，声明__initial_sp、__heap_base 和__heap_limit 这三个标号具有全局属性，可被外部的文件使用。__initial_sp 表示栈顶地址，__heap_base 表示堆起始地址，__heap_limit 表示堆结束地址。

337 行没有定义__MICROLIB，实际的情况就是我们没有定义__MICROLIB，所以使用默认的 C 库运行。堆栈的初始化由 C 库函数__main 来完成。

339 行 IMPORT 声明__use_two_region_memory 标号来自外部文件。

340 行 EXPORT 声明__user_initial_stackheap 具有全局属性，可被外部的文件使用。

342 行标号__user_initial_stackheap，表示用户堆栈初始化程序入口。

接下来进行堆栈空间初始化，堆是从低到高生长，栈是从高到低生长，是两个互相独立的数据段，并且不能交叉使用。

344 行保存堆起始地址。

345 行保存栈大小。

346 行保存堆大小。

347 行保存栈顶指针。

348 行跳转到 LR 标号给出的地址，不用返回。

354 行 END 表示到达文件的末尾，文件结束。

Use MicroLIB

MicroLIB 是 MDK 自带的微库，是缺省 C 库的备选库，MicroLIB 进行了高度优化使得其代码变得很小，功能比缺省 C 库少。MicroLIB 是没有源码的，只有库。

关于 MicroLIB 更多知识可以看官方介绍 http://www.keil.com/arm/microlib.asp 。

3. 系统启动流程

在以前 ARM7/ARM9 内核的控制器在复位后，CPU 会从存储空间的绝对地址

0x00000000 取出第一条指令执行复位中断服务程序的方式启动，即固定了复位后的起始地址为 0x00000000（PC = 0x00000000），同时中断向量表的位置也是固定的。而 Cortex-M3 内核复位后的起始地址和中断向量表的位置可以被重映射。充映射的方法是通过启动模式的选择，有以下 3 种情况:

1、通过 boot 引脚设置可以将中断向量表定位于 SRAM 区，即起始地址为 0x2000000，同时复位后 PC 指针位于 0x2000000 处；

2、通过 boot 引脚设置可以将中断向量表定位于 FLASH 区，即起始地址为 0x8000000，同时复位后 PC 指针位于 0x8000000 处；

3、 通过 boot 引脚设置可以将中断向量表定位于内置 Bootloader 区，本文不对这种情况做论述。

Cortex-M3 内核规定，起始地址必须存放堆顶指针，而第二个地址则必须存放复位中断入口向量地址，这样在 Cortex-M3 内核复位后，会自动从起始地址的下一个 32 位空间取出复位中断入口向量，跳转执行复位中断服务程序。

下面将结合《Cortex-M3 权威指南(中文)》chpt03-复位序列的内容进行讲解。

启动模式不同，启动的起始地址是不一样的，下面我们以代码下载到内部 FLASH 的情况举例，即代码从地址 0x0800 0000 开始被执行。

我们知道的复位方式有三种：上电复位，硬件复位和软件复位。当产生复位，并且离开复位状态后，CM3 内核做的第一件事就是读取下列两个 32 位整数的值：

（1）从地址 0x0800 0000 处取出堆栈指针 MSP 的初始值，该值就是栈顶地址。

（2）从地址 0x0800 0004 处取出程序计数器指针 PC 的初始值，该值指向复位后执行的第一条指令。下面用示意图表示，如图 3.1 所示。

图 3.1 复位序列

我们看看战舰开发板 HAL 库例程的实验 1 跑马灯实验中，取出的 MPS 和 PC 的值是多少，方法如图 3.2 所示。

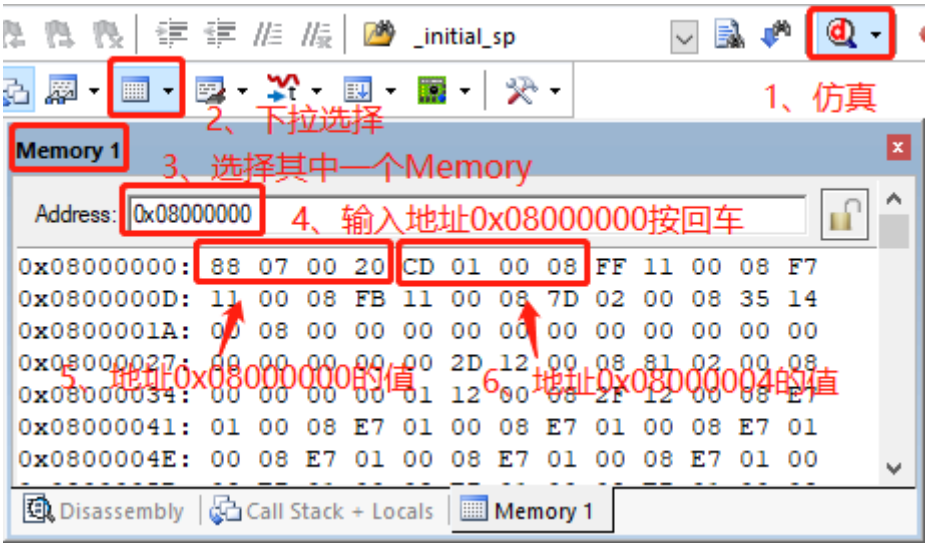

图 3.2 取出的 MPS 和 PC 的值

由图 3.2 可以知道地址 0x08000000 的值是 0x20000788，地址 0x08000004 的值是

0x080001CD，即堆栈指针 SP = 0x20000788，程序计数器指针 PC = 0x080001CD。因为 CM3 内核是小端模式，所以倒着读。

请注意，这与传统的 ARM 架构不同——其实也和绝大多数的其它单片机不同。传统的 ARM 架构总是从 0 地址开始执行第一条指令。它们的 0 地址处总是一条跳转指令。而在 CM3 内核中，0 地址处提供 MSP 的初始值，然后就是向量表（向量表在以后还可以被移至其它位置）。向量表中的数值是 32 位的地址，而不是跳转指令。向量表的第一个条目指向复位后应执行的第一条指令，就是 Reset_Handler 这个函数。下面继续以战舰开发板 HAL 库例程的实验 1 跑马灯实验为例，代码从地址 0x0800 0000 开始被执行，讲解一下系统启动，初始化堆栈、MSP 和 PC 后的内存情况。

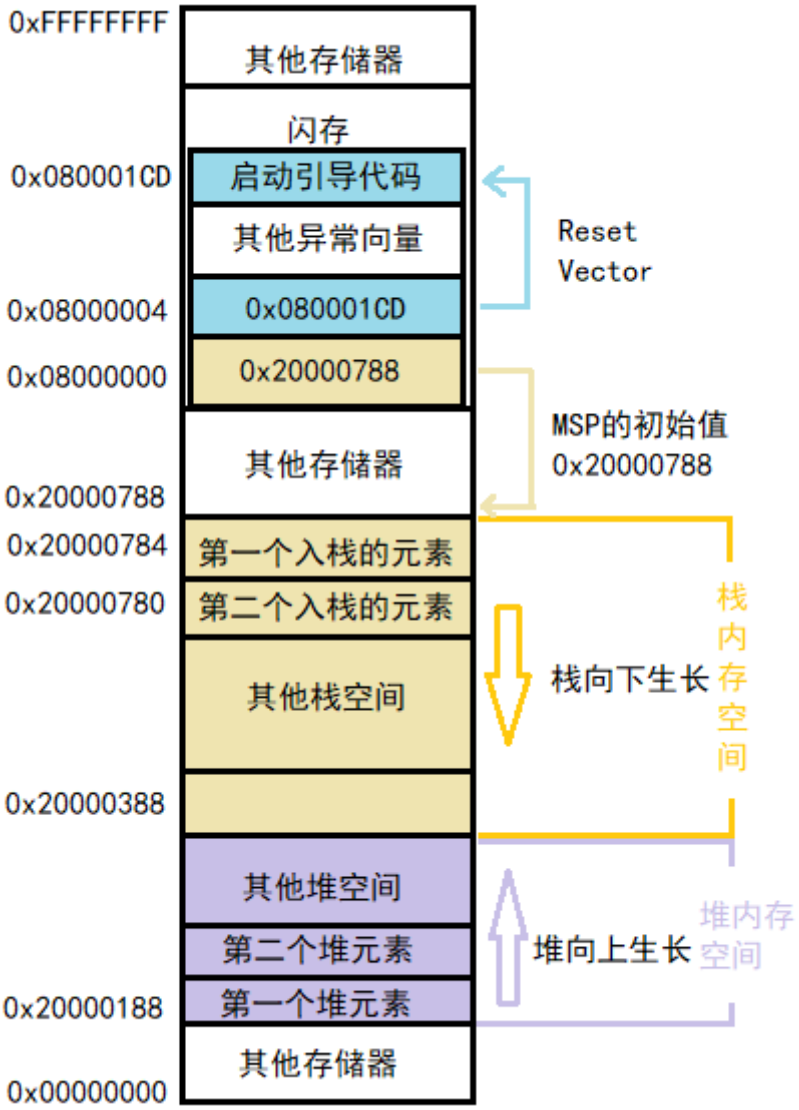

图 3.3 初始化堆栈、MSP 和 PC 后的内存情况

因为 CM3 使用的是向下生长的满栈，所以 MSP 的初始值必须是堆栈内存的末地址加 1。举例来说，如果你的栈区域在 0x20000388 - 0x20000787 之间，那么 MSP 的初始值就必须是 0x20000788。

向量表跟随在 MSP 的初始值之后——也就是第 2 个表目。

R15 是程序计数器，在汇编代码中，可以使用名字“PC”来访问它。ARM 规定：PC 最低两位并不表示真实地址，最低位 LSB 用于表示是 ARM 指令（0）还是 Thumb 指令（1），

因为 CM3 主要执行 Thumb 指令，所以这些指令的最低位都是 1（都是奇数）。因为 CM3 内部使用了指令流水线，读 PC 时返回的值是当前指令的地址+4。比如说：

```
0x1000:  MOV R0,  PC ;  R0 = 0x1004
```

如果向 PC 写数据，就会引起一次程序的分支（但是不更新 LR 寄存器）。CM3 中的指令至少是半字对齐的，所以 PC 的 LSB 总是读回 0。然而，在分支时，无论是直接写 PC 的值还是使用分支指令，都必须保证加载到 PC 的数值是奇数（即 LSB=1），表明是在 Thumb 状态下执行。倘若写了 0，则视为转入 ARM 模式，CM3 将产生一个 fault 异常。

正因为上述原因，图 3.3 中使用 0x080001CD 来表达地址 0x080001CC。当 0x080001CD 处的指令得到执行后，就正式开始了程序的执行（即去到 C 的世界）。所以在此之前初始化 MSP 是必需的，因为可能第 1 条指令还没执行就会被 NMI 或是其它 fault 打断。MSP 初始化好后就已经为它们的服务例程准备好了堆栈。

STM32 启动文件的解析就给大家介绍到这里。

4. 其他

1、 购买地址：

官方店铺 1：https://openedv.taobao.com

官方店铺 2：https://zhengdianyuanzi.tmall.com

2、 资料下载

模块资料下载地址：http://www.openedv.com/docs/index.html

3、 技术支持

公司网址：www.alientek.com

技术论坛：www.openedv.com

在线教学：www.yuanzige.com

传真：020-36773971

电话：020-38271790

正点原子