STM32启动详细流程之__main

__main -> main

1.前言

上一篇博客详细地讲述了一个流程：

cpu执行第一条用户代码 -> 调用__main函数

这篇博客着重讲述了STM32启动文件中一些需要注意的细节，对于STM32启动文件的内容没有过多的讲解，因为我的第一篇博客讲述的就是STM32启动文件的解释。

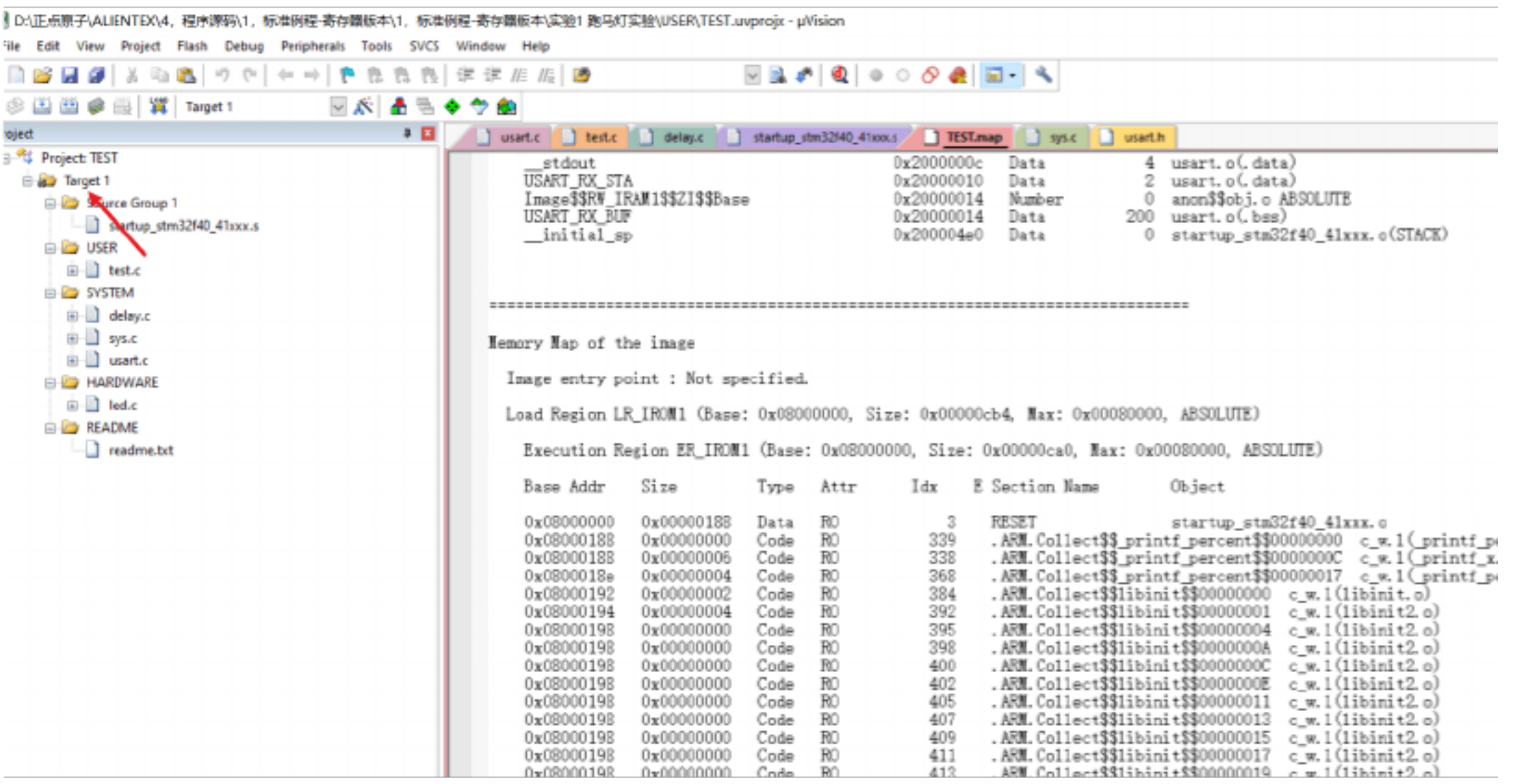
而本篇博客将要详细地描述一个流程：

_ _main函数 -> __rt_entry -> main函数

这里再次声明一下：__main函数是c库中的一个函数，和用户编写的main函数是有区别的

2.必备知识

必备知识中主要是用到了.map文件，双击红色箭头所指向的区域就可以打开



2.1. 用户程序在FLASH中的组织架构

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

Memory Map of the image

  Image entry point : Not specified.

  Load Region LR_IROM1 (Base: 0x08000000, Size: 0x00000cb4, Max: 0x00080000, ABSOLUTE)

    Execution Region ER_IROM1 (Base: 0x08000000, Size: 0x00000ca0, Max: 0x00080000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x08000000   0x00000188   Data   RO            3    RESET                startup_stm32f40_41xxx.o
    0x08000188   0x00000000   Code   RO          339    .ARM.Collect$$_printf_percent$$00000000  c_w.l(_printf_percent.o)
    0x08000188   0x00000006   Code   RO          338    .ARM.Collect$$_printf_percent$$0000000C  c_w.l(_printf_x.o)
    0x0800018e   0x00000004   Code   RO          368    .ARM.Collect$$_printf_percent$$00000017  c_w.l(_printf_percent_end.o
    0x08000192   0x00000002   Code   RO          384    .ARM.Collect$$libinit$$00000000  c_w.l(libinit.o)
    0x08000194   0x00000004   Code   RO          392    .ARM.Collect$$libinit$$00000001  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          395    .ARM.Collect$$libinit$$00000004  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          398    .ARM.Collect$$libinit$$0000000A  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          400    .ARM.Collect$$libinit$$0000000C  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          402    .ARM.Collect$$libinit$$0000000E  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          405    .ARM.Collect$$libinit$$00000011  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          407    .ARM.Collect$$libinit$$00000013  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          409    .ARM.Collect$$libinit$$00000015  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          411    .ARM.Collect$$libinit$$00000017  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          413    .ARM.Collect$$libinit$$00000019  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          415    .ARM.Collect$$libinit$$0000001B  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          417    .ARM.Collect$$libinit$$0000001D  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          419    .ARM.Collect$$libinit$$0000001F  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          421    .ARM.Collect$$libinit$$00000021  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          423    .ARM.Collect$$libinit$$00000023  c_w.l(libinit2.o)
    0x08000198   0x00000000   Code   RO          425    .ARM.Collect$$libinit$$00000025  c_w.l(libinit2.o)
    0x08000428   0x0000000c   Code   RO          377    .text                c_w.l(exit.o)
    0x08000434   0x000000b4   Code   RO          124    i.Ex_NVIC_Config     sys.o
    0x080004e8   0x00000040   Code   RO          125    i.GPIO_AF_Set        sys.o
    0x08000528   0x000000de   Code   RO          126    i.GPIO_Set           sys.o
    0x08000606   0x00000004   Code   RO          127    i.INTX_DISABLE       sys.o
    0x0800060a   0x00000004   Code   RO          128    i.INTX_ENABLE        sys.o
    0x0800060e   0x00000002   PAD
    0x08000610   0x00000040   Code   RO          268    i.LED_Init           led.o
    0x08000650   0x00000078   Code   RO          129    i.MY_NVIC_Init       sys.o
    0x080006c8   0x00000028   Code   RO          130    i.MY_NVIC_PriorityGroupConfig  sys.o
    0x080006f0   0x00000010   Code   RO          131    i.MY_NVIC_SetVectorTable  sys.o
    0x08000700   0x00000068   Code   RO          132    i.Stm32_Clock_Init   sys.o
    0x08000768   0x0000010c   Code   RO          133    i.Sys_Clock_Set      sys.o
    0x08000874   0x00000010   Code   RO          134    i.Sys_Soft_Reset     sys.o
    0x08000884   0x0000004c   Code   RO          135    i.Sys_Standby        sys.o
    0x080008d0   0x00000080   Code   RO          219    i.USART1_IRQHandler  usart.o
    0x08000950   0x00000004   Code   RO          136    i.WFI_SET            sys.o
    0x08000954   0x00000048   Code   RO           12    i.__main             test.o
    0x0800099c   0x00000006   Code   RO          220    i._sys_exit          usart.o
    0x080009a2   0x00000002   PAD
    0x080009a4   0x00000040   Code   RO           83    i.delay_init         delay.o
    0x080009e4   0x00000038   Code   RO           84    i.delay_ms           delay.o
    0x08000a1c   0x0000003c   Code   RO           85    i.delay_us           delay.o
    0x08000a58   0x0000003c   Code   RO           86    i.delay_xms          delay.o
    0x08000a94   0x0000001c   Code   RO          221    i.fputc              usart.o
    0x08000ab0   0x000000bc   Code   RO           13    i.main               test.o
    0x08000b6c   0x00000100   Code   RO          222    i.uart_init          usart.o
    0x08000c6c   0x0000000a   Code   RO          446    x$fpl$fpinit         fz_wm.l(fpinit.o)
```

上面两张图截取了镜像文件在FLASH上的内存分布。

从上面两张图可以知道，在程序的最开始处，存储的是数据段，这个数据段就是中断向量表，里面存储这所有中断函数的入口地址。

紧跟着的就是代码段，代码段包含了自己编写的用户代码和库函数。

之后有跟着数据段，这个数据段有个专有的名称，叫做代码常量区，也就是你定义的const类型的全局变量（记住不是const类型的局部变量，const类型的局部变量还是存储在栈区）会存储在这个区域。

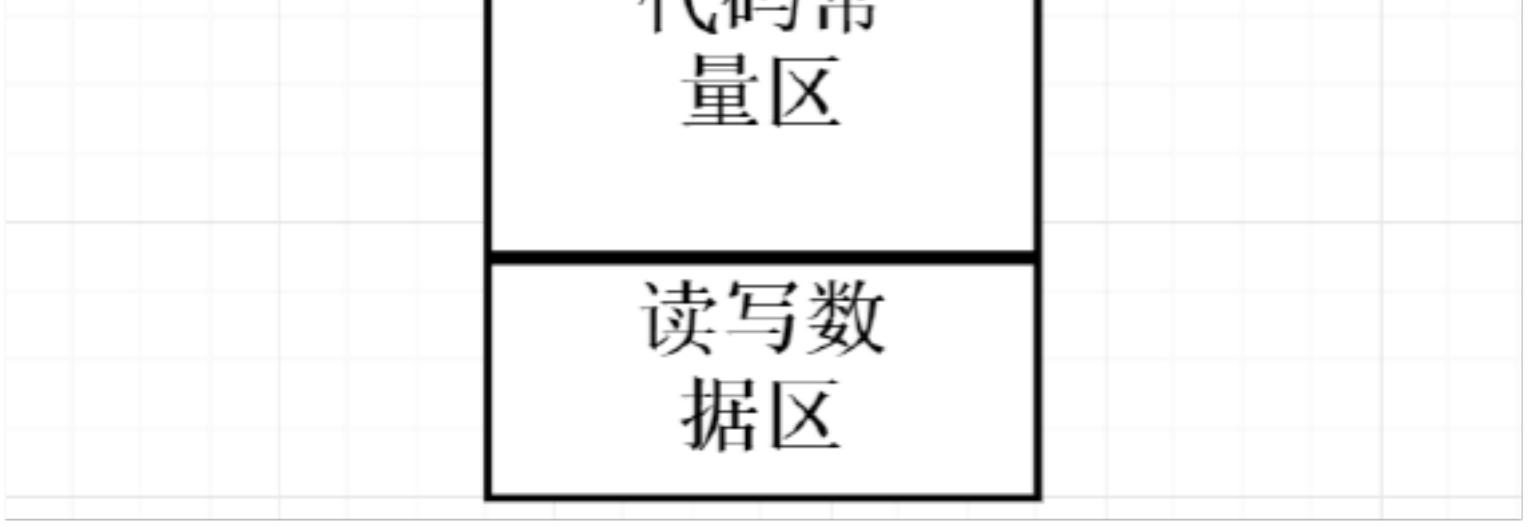
特别注意，非常重要的知识点：

在代码常量区后面还有一个区，叫做读写数据区，这个区域中的数据最终要被拷贝到SRAM中去，因为FLASH只能读不能写（事实上可以进行写操作，只不过需要密钥而已，参考手册中有说明）而SRAM中的数据是可读可写的。

但是，.map文件中并没有提到，也就是说你从.map文件中是找不到这个区的，

0x08000a94	0x0000001c	Code	RO	221	i.fputc	usart.o
0x08000ab0	0x000000bc	Code	RO	13	i.main	test.o
0x08000b6c	0x00000100	Code	RO	222	i.uart_init	usart.o
0x08000c6c	0x0000000a	Code	RO	446	x$fp1$fpinit	fz_wm.l(fpinit.o)

你能看到的最后一项就是代码常量区，因此这个地方一般情况下很难发现到，只有深入__main函数之后才可以知道。



值得注意的是：

0x08000434	0x000000b4	Code	RO	124	i.Ex_NVIC_Config	sys.o
0x080004e8	0x00000040	Code	RO	125	i.GPIO_AF_Set	sys.o
0x08000528	0x000000de	Code	RO	126	i.GPIO_Set	sys.o
0x08000606	0x00000004	Code	RO	127	i.INTX_DISABLE	sys.o
0x0800060a	0x00000004	Code	RO	128	i.INTX_ENABLE	sys.o
0x0800060e	0x00000002	PAD				
0x08000610	0x00000040	Code	RO	268	i.LED_Init	led.o
0x08000650	0x00000078	Code	RO	129	i.MY_NVIC_Init	sys.o
0x080006c8	0x00000028	Code	RO	130	i.MY_NVIC_PriorityGroupConfig	sys.o

在代码区中，不仅有Code、Data类型的数据，还有PAD

PAD就是padding的意思，中文翻译过来就是填充的意思

> 作用：进行4字节对齐，提高cpu的取指速率

也就是说，无论是指令还是数据，在内存中都要4个字节对齐，所表现出来的特征就是：

地址的最低两位都为0，换成16进制来说，就是最后一个字母只能为0、4、8、c。

2.2. 用户数据在SRAM中的组织架构

```
Execution Region RW_IRAM1 (Base: 0x20000000, Size: 0x000004e0, Max: 0x00020000, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x20000000   0x00000008   Data   RW          14    .data               test.o
0x20000008   0x00000004   Data   RW          87    .data               delay.o
0x2000000c   0x00000006   Data   RW         224    .data               usart.o
0x20000012   0x00000002   PAD
0x20000014   0x000000c8   Zero   RW         223    .bss                usart.o
0x200000dc   0x00000004   PAD
0x200000e0   0x00000000   Zero   RW           2    HEAP                startup_stm32f40_41xxx.o
0x200000e0   0x00000400   Zero   RW           1    STACK               startup_stm32f40_41xxx.o
```

在SRAM中，第一个区域叫做全局区，也有人叫静态区。你定义的全局变量（有初始值），静态变量都存放在这个区域当中。

这里需要说明一下一个特例：

比如你定义了一个全局变量：`int a;`

没有初始化的全局变量默认为0，但要注意，并不是说没有初始化的全局变量就属于.bss段（网上有很多的博客都说错了），它还是属于全局区，它的值是编译器赋值给它的

（视频需要进行验证）

紧跟着的就是.bss段。

> 注意：.bss段不被包含在可执行文件当中

定义的未初始化全局数组，未初始化的静态全局数组等等保存在.bss段。

接下来就是堆和栈，因为堆向上生长，栈向下生长，因此堆在栈的前面。

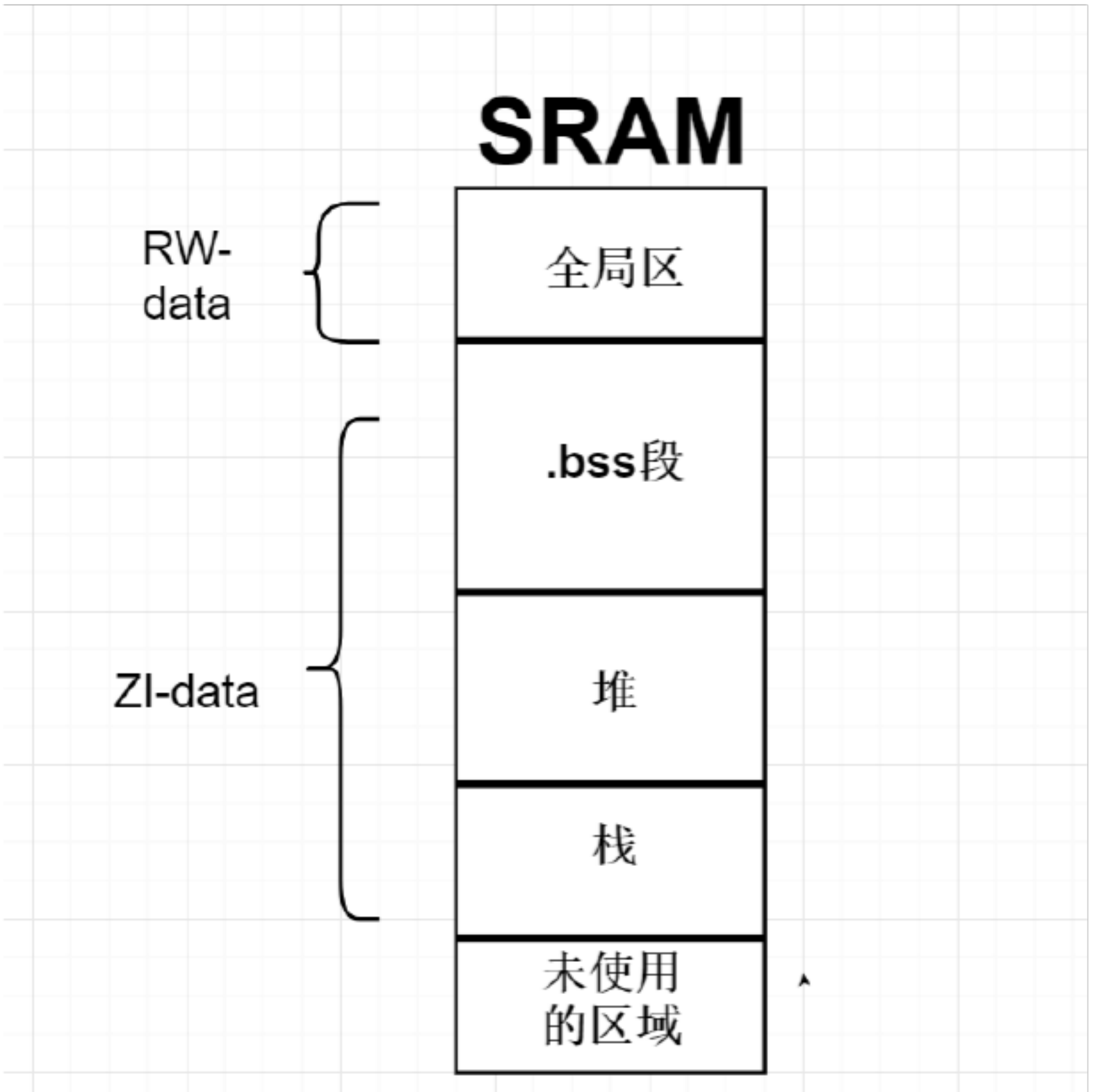
此时，我们得到一个非常重要的结论：

栈顶指针的值 = RW-data + ZI-data

大家可以想一下，为什么。

还有，由于当一个程序生成可执行文件之后，栈顶指针的值就确定了。

那也就是说，从栈顶指针处，到SRAM最后一个存储单元都处于未使用状态，也就是说，有一部分内存我们是没有使用的，这里需要注意



2.3. 2.加载地址 链接地址 运行地址 存储地址

加载地址：将指令或数据从地址A拷贝到地址B，地址A就是加载地址

链接地址：由链接脚本文件指出，链接的时候确定

运行地址：程序在内存中运行时候的地址

存储地址：指令或数据在flash中存放的存储地址，就是存储地址

这里需要说明一下：

> 链接地址是静态的，在程序链接的时候确定。
> 运行地址是动态的，因为当你使用位置无关码（后面会提到）将程序从A地址拷贝到B地址处，那么运行地址就发生了改变。
> 存储地址就是加载地址，没有区别

2.3.4.代码重定向

程序或数据的链接地址要和运行地址一致，但往往程序或数据的存储地址（加载地址）和运行地址不一样，因此需要代码重定向。

代码重定向：使用位置无关码将用户程序或数据从存储地址拷贝到运行地址

用一句很精确的话来描述代码重定向：

使逻辑地址与实际物理地址一一对应的过程

这篇博客非常详细地描述了代码重定向的过程，读者特别需要注意的就是:

MCU和MPU代码重定向的区别

2.3.4.1.位置无关码

当程序或数据的链接地址和运行地址不一样的时候，此时只有位置无关码才能够正确被执行

位置无关码：依赖于程序当前运行的PC值，进行相对的跳转，导致的结果就是，无论代码在哪，总能达到指令正常运行的目的，因此是位置无关的。

位置有关码：不依赖当前PC值，是绝对跳转，只有程序运行在链接地址处时，才能达到指令的正常目的，因此是位置有关系的。

3.__main函数

作用：Initialization of the execution environment and execution of the application

You can customize execution intialization by defining your own __main that branches to __rt_entry.

The entry point of a program is at __main in the C library where library code:

1. Copies non-root (RO（不会拷贝，官方提供和实际实践有出入） and RW) execution regions from their load addresses to their execution addresses. Also, if any data sections are compressed, they are decompressed from the load address to the execution address.
2. Zeroes ZI regions.
3. Branches to __rt_entry.

If you do not want the library to perform these actions, you can define your own __main that branches to __rt_entry.（我们后面会自己实现__main函数）

> 注意：__main函数不会将RO段数据拷贝到执行地址处，虽然官方说明了

4._rt_entry函数

4.1.procedure

The library function `__rt_entry()` runs the program as follows:

1. Sets up the stack and the heap by one of a number of means that include calling `__user_setup_stackheap()`, calling `__rt_stackheap_init()`, or loading the absolute addresses of scatter-loaded regions.
2. Calls `__rt_lib_init()` to initialize referenced library functions, initialize the locale and, if necessary, set up argc and argv for `main()`.This function is called immediately after`__rt_stackheap_init()` and is passed an initial chunk of memory to use as a heap. This function is the standard ARM C library initialization function and it must not be reimplemented.
3. Calls `main()`, the user-level root of the application.

   From `main()`, your program might call, among other things, library functions.
4. Calls `exit()` with the value returned by `main()`.

entry的是ARM汇编语法中程序的入口地址，GNU Assember语法中start是程序的入口地址

__rt_lib库函数是没有源文件，都已经编译完成了。

The symbol __rt_entry is the starting point for a program using the ARM C library.

Control passes to __rt_entry after all scatter-loaded regions have been relocated to their execution addresses.

4.2.Usage

The default implementation of __rt_entry:

1. Sets up the heap and stack.
2. Initializes the C library by calling __rt_lib_init.(ARMc库里面全面都是.b .l形式的库，没有源码)
3. Calls `main()`.
4. Shuts down the C library, by calling __rt_lib_shutdown.
5. Exits.

__rt_entry must end with a call to one of the following functions:

`exit()`

Calls `atexit()-` registered functions and shuts down the library.

`__rt_exit()`

Shuts down the library but does not call `atexit()` functions.

`_sys_exit()`

Exits directly to the execution environment. It does not shut down the library and does not call `atexit()` functions.

5.自己实现__main函数

视频链接如下：

需要源码的和我说

5.1.消除警告

提示：程序的首地址并不和程序的入口地址等效

注意：ARM汇编语法entry是一个程序的入口地址，GNU汇编语法start是一个程序的入口地址

我们已自己实现__main 函数，ENTRY 已没有实质作用， 但为了避免 KEIL 警告，这里加上。

6.自己实现__rt_entry函数

你觉得你行吗？你知道要多少行代码吗，并且，没必要

7.问题思考

7.1.为什么我们可以自己编写__main和__rt_entry

因为库函数里面的 ___main函数 和 ____rt_entry函数是弱函数

```
73  __weak void __rt_entry(void)
74  {
75
76
```

弱函数定义时需要写红色箭头所指向的关键字。

7.2.当一个用户程序运行完以后，会出现什么情况

结论：具体情况我也不知，有大神知道可以告诉我吗

半主机模式介绍:

8.本系列文章总结

这篇文章是本系列博客文章的结尾，主要就是描述了:

_ _main函数 -> __rt_entry函数 -> main函数

本系列文章流程:

可执行程序 -> cpu执行第一条用户代码的流程 -> _ _main函数 -> __rt_entry函数 -> main函数

编写用户代码到如何生成可执行文件并没有解释，如果需要，可以安排

详细地阐述了可执行文件是如何被加载到FLASH上，以及编写的用户程序（main函数）被调用之前经历了哪些步骤。

如果你对这些步骤了然于胸的时候，那么恭喜你，你已经很强了，大部分人是学不到这么深的，就算工作了很多年

希望本系列的博文能够对你有所帮助

最后，希望大家能够学有所成，未来可期!