

CORTEX M4 进入 main 之前干的那些事

目录

版本控制.....	i
1 前言	2
2 资源下载	3
3 CORTEX M4 核的简单介绍	4
3.1 资料来源	4
3.2 CORTEX M4 核寄存器和存储器模型	4
3.2.1 寄存器	4
3.2.2 存储器模式	6
3.3 C 语言如何到可执行文件	7
3.4 存储器中的各个段	7
3.5 其他预备知识	8
3.6 小结	8
4 SES 启动代码分析	9
4.1 可执行文件的存储器使用信息	9
4.2 ses_nRF_Startup.s 分析	11
4.3 thumb_crt0.s 分析	18
4.4 ses_nrf52_Vectors.s 分析	23
5 工程中的符号	26
5.1 工程设置符号	26
5.2 section 符号	27
6 XML 文件详解(Section Placement file)	28
6.1 name 名字属性	28
6.2 start 起始地址属性	28
6.3 size 段具体大小属性	28
6.4 address_symbol 地址开始符号属性	28
6.5 end_symbol 地址结束符号属性	28
6.6 size_symbol 段大小符号属性	29
6.7 alignment 访问对齐属性	29
6.8 fill 填充属性	29
6.9 inputsections 输入哪些文件到 name 段	29
6.10 keep 保持属性	30
6.11 load 加载属性	30
6.12 place_from_segment_end 段末尾开始放数据属性	30
6.13 runin 属性	30
6.14 runoffset 属性	30
7 本文工程 XML 分析	32
8 工程的链接文件	35
9 本文工程所有符号	40
10 参考文档	49

版本控制

版本	修改内容	修改时间	修改人	联系方式
V1.0	初稿	2018/6/16	刘权	450547566@qq.com

1 前言

已经写了 8 年的嵌入式代码，但是一直没有想过一个问题：对于一个 Cortex M4 核的单片机芯片，ARM 公司的做了什么？芯片厂商做了什么？程序员做了什么？当拿到一个不同核的 mcp，对于程序员基本可以使用同样的方法将其运行起来，因为芯片厂商为工程师屏蔽了必须了解 MCP 架构的要求。然而正是因为这样的屏蔽，导致了许多问题都没有正在的去研究。

许多时候想去学一个东西时，会发现总是找不到自己想要的资料，随着你对在你想要学习的东西越来越理解的时候，会发现原来你想学习的东西到处都是资料，可能这个资料一开始就在你眼前，但是却不知道这个就是你想要的资料。这是我在学习过程中发现的一个我的现象。

这里要讲的就是一个很简单的问题：单片机(这里讲 CORTEX M4 核的处理器)到底是如何进行工作的，更进一步是：CORTEX M4 核的处理器是如何从上电跑到主函数运行用户代码的。其实简单一点的话：

- a、上电从 0 地址处得到 SP 地址；
- b、从 4 地址处得到复位地址；
- c、复位中断服务函数进行全局未赋值变量 bss 段和全局变量赋值变量 data 段的拷贝；
- d、跳转到用户 main 函数运行。

就 4 点完成了进入用户程序之前的工作。但是会不会有些疑问：

- 1、上面 4 点有没有官方的文档支持，因为或许是我自己猜测的；
- 2、SP 指针是怎么确定的，地址是哪？
- 3、MSP 主栈和 PSP 进程栈是什么关系，怎么用的？
- 4、0 地址处放的是啥？---中断向量表，那中断向量表是怎么放到 0 地址的？
- 5、中断向量表是不是可以偏移，偏移是由 CORTEX M4 核设计的还是由芯片厂商定义的。
- 6、bss 和 data 段为什么要复制？从哪复制到哪？复制的地址是什么？怎么确定的？
- 7、既然是拷贝，那么一定是有拷贝代码的，一般是用汇编，那么可不可以换成 c 语言实现，即便 c 语言比汇编慢，但是这个 c 拷贝也只是在上电时运行一次而已。
- 8、浮点运算支持是由 CORTEX M4 核设计的还是由厂商设计？

以上问题中，有许多是可以从宋岩译的《Cortex-M3 权威指南(中文)》中找到答案，这就是前面我提到的这个资料其实一直在我面前，但是如果没有去真正阅读，我是不知道这个文章能帮我解决困扰我多年的问题的。当然我还是喜欢找到最官方的资料来论证我的观点。所以这里也建议新手可以先阅读《Cortex-M3 权威指南(中文)》再来阅读本文，当然高手请忽略本文，同时作者能力实属有限，难免出现一些不同等级的错误，也请读者联系作者 450547566@qq.com 进行更正。

本文使用 Segger Embedded Studio(SES)作为 IDE，使用的编译工具是 GCC，之所以使用这个 IDE 是因为很多问题可以很直观的看出来，而 KEIL 的话已经偷偷的帮程序员做了很多的事情。本文还会涉及一些链接文件以及 SES 工具的 Section Placement 文件。

2 资源下载

相关的环境搭建的下载地址：

SES 下载地址：<https://www.segger.com/downloads/embedded-studio>

Jlink 下载地址：<https://www.segger.com/downloads/jlink#>

代码使用的是 nordic 的 SDK 代码进行分析的，下载地址：

http://developer.nordicsemi.com/nRF5_SDK/ 下载 SDK14.1 以上的 SDK，里面自带
有 SES 的工程。

本文工程文件下载地址：

<https://github.com/Bluetooth-BLE/RT-Thread-3.1.0-NORDIC-SES>

本文也在 RT-Thread-3.1.0-NORDIC-SES\nordic_example\documentation 中

3 CORTEX M4 核的简单介绍

因为是针对 cortex M4 核的启动讲解，所以需要对它又基本的认识，首先介绍资料的参考来源。

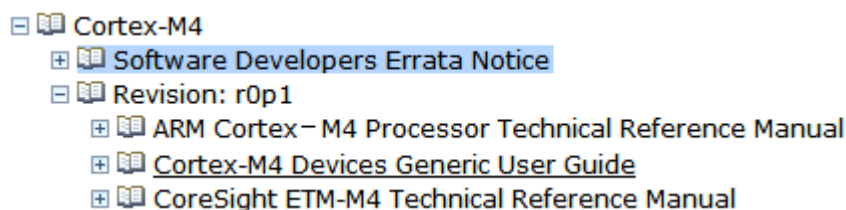
3.1 资料来源

至所以我每次写文章都会有一节资料的来源，一方面是授之以渔，另一方面是告诉读者文中有很大一部分都是有据可依的。

中文的话可以看宋岩译的《Cortex-M3 权威指南(中文)》，这个文档可以从很多地方下载到。

ARM 官方文档：

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.cortexm.m4/index.html>

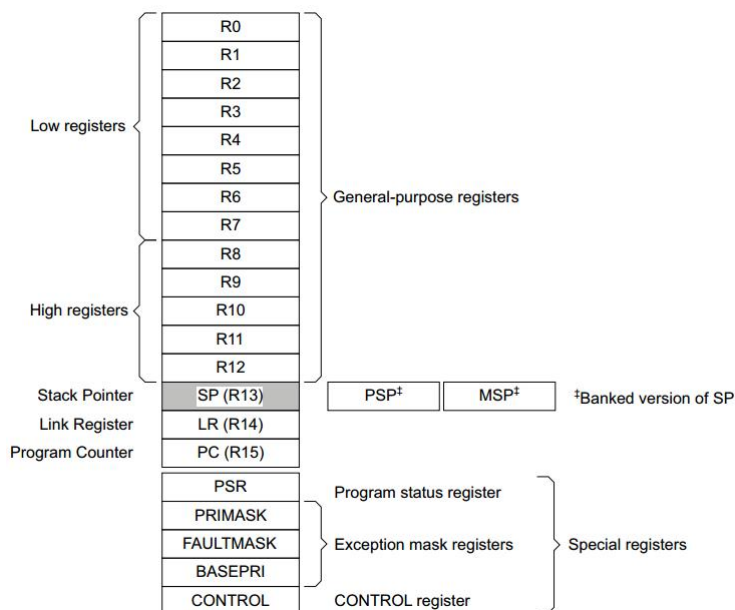


3.2 CORTEX M4 核寄存器和存储器模型

这里主要谈 cortex m4 核内部的特殊寄存器和存储器模型。对于核内有寄存器所有的嵌入式的工作者都知道，但是并不是所有的工程师都使用了这些寄存器，同时我们总是说 xxx 地址，但是从来没有听说 R0 的地址是多少，那么地址是针对什么的？这里就要讲到 cortex m4 核的存储器模式。

3.2.1 寄存器

核内的寄存器是没有地址的概念的，地址都指存储器模式中的外围。



名称	访问类型	级别(用户和特权)	复位值	描述
R0-R12	RW	任意	Unknown	通用寄存器
MSP ¹	RW	特权	从 0 地址加载	主栈寄存器 R13
PSP	RW	任意	Unknown	进程栈寄存器 R13
LR	RW	任意	0xFFFFFFFF	链接寄存器 R14
PC ²	RW	任意	从 4 地址加载	程序计数器 R15
PSR	RW	特权	0x01000000	程序状态寄存器
APSR	RW	任意	Unknown	应用程序状态寄存器
IPSR	RO	特权	0x00000000	中断程序状态寄存器
EPSR	RO	特权	0x01000000	执行程序状态寄存器
PRIMASK	RW	特权	0x00000000	屏蔽所有可配置优先级中断的寄存器
FAULTMASK	RW	特权	0x00000000	屏蔽所有 fault 中断寄存器
BASEPRI	RW	特权	0x00000000	屏蔽大于这个寄存器值的所有中断
CONTROL	RW	特权	0x00000000	特权控制寄存器

注 1: Cortex™-M4 Devices Generic User Guide 第 17 页

Stack Pointer

The *Stack Pointer* (SP) is register R13. In Thread mode, bit[1] of the CONTROL register indicates the stack pointer to use:

- 0 = *Main Stack Pointer* (MSP). This is the reset value.
- 1 = *Process Stack Pointer* (PSP).

On reset, the processor loads the MSP with the value from address 0x00000000.

注 2:

ProGRAM Counter

The *ProGRAM Counter* (PC) is register R15. It contains the current progRAM address. On reset, the processor loads the PC with the value of the reset vector, which is at address 0x00000004.

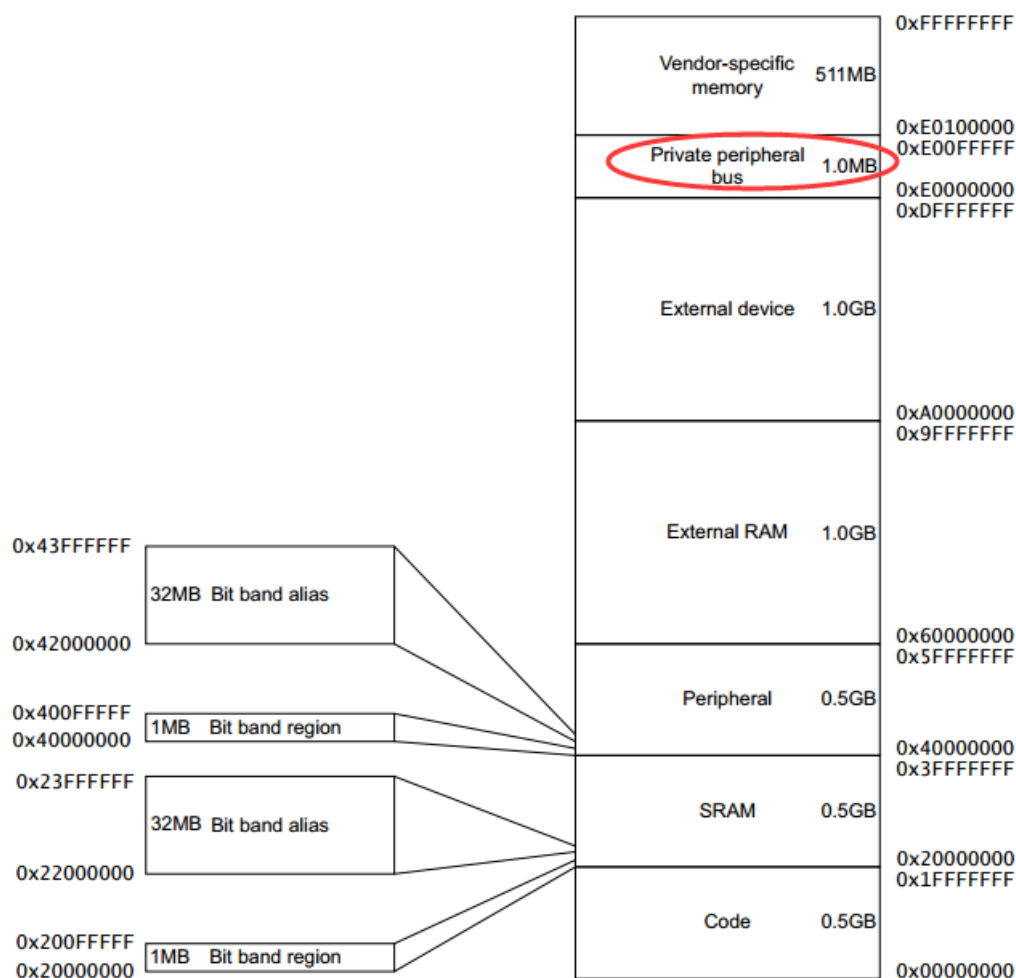
Bit[0] of the value is loaded into the EPSR T-bit at reset and must be 1.

这里主要讲 CONTROL 寄存器，寄存器位描述如下：

位	名称	功能
[31:3]	—	保留
[2]	FPCA	这个位来决定在处理异常时是否保留浮点状态 0=没有浮点处于活动状态 1=有浮点处于活动状态
[1]	SPSEL	堆栈指针选择 0=选择主堆栈指针 MSP（复位后缺省值） 1=选择进程堆栈指针 PSP 在线程或基础级（没有在响应异常——译注），可以使用 PSP。在 handler 模式下，只允许使用 MSP，所以此时不得往该位写 1。
[0]	nPRIV	0=特权级的线程模式 1=用户级的线程模式 Handler 模式永远都是特权级的。

3.2.2 存储器模式

对于核内的寄存器只有符号，没有地址，而地址都是针对内核总线上能访问到的地址空间的地址，对于 cortex m4 核来说，无论外面挂什么，对核来说都是挂的地址，就像 linux 的驱动模型一样，一切皆文件。cortex M4 能访问的地址空间范围是 4G，这 4G 的存储器地址，M4 核有强制规定，如下：



为了统一所有厂商以及软件或者操作系统能在各个厂商的芯片上能跑起来，一些重要的外围 M4 核也进行了强制规定的，强制规定在存储模型上的访问地址以及各个寄存器的定义都有详细说明，也就是上图中圈出来的部分，1M 空间的 Private peripheral bus，这总线的地址范围是 0xE0000000-0xE00FFFFFFF。在这个地址范围上，M4 核进行了如下外设的定义：

地址	核心外围	描述
0xE000E008-0xE000E00F	SyStem Control Block	系统控制块
0xE000E010-0xE000E01F	System timer	系统时钟
0xE000E100-0xE000E4EF	Nested Vectored Interrupt Controller	嵌套中断向量控制器
0xE000ED00-0xE000ED3F	System Control Block	系统控制块
0xE000ED90-0xE000ED93	MPU Type Register	读出为 0 表示没有 MPU 实施
0xE000ED90-0xE000EDB8	Memory Protection Unit	内存保护单元
0xE000EF00-0xE000EF03	Nested Vectored Interrupt Controller	嵌套中断向量控制器
0xE000EF30-0xE000EF44	Floating Point Unit	浮点运算控制单元

也就是说，所有厂商只要用 M4 的核，那么上面的外围控制的地址是不允许厂商自己进行定义的。

当系统复位时，中断向量表固定在地 0x00000000。在特权模式下软件可以写 VTOR 寄存器(中断向量偏移寄存器)将向量表的起始地址重新定位到一个不同的内存位置，只要在 0x00000080 到 0x3FFFFFF80 范围内。

原文如下：

On system reset, the vector table is fixed at address 0x00000000. Privileged software can write to the VTOR to relocate the vector table start address to a different memory location, in the range 0x00000080 to 0x3FFFFFF80

3.3 C 语言如何到可执行文件

这里简单提一下 c 语言如何到可执行文件的，共经历四个阶段，依次为：预处理、编译、汇编、链接。

预处理	编译	汇编	链接
a、预处理指令(#include) b、删除注释 c、#define 宏替换 d、条件编译	a、语法分析 b、词法分析 c、语义分析 d、符号汇总	a、形成符号表 b、汇编语言变为机器指令	a、合并段 b、合并符号表 c、重定位符号表
形成.i 文件	形成.s 文件	形成.o 文件	形成可执行文件
		每个.o 文件中都可能包含可执行代码、全局赋值变量、常量以及全局未赋值的变量。	这里需要将.o 文件中的所有同类型的可执行代码(text 段)、全局赋值变量(data 段)、常量(rodata 段)和全局未赋值的变量都集中的放在 flash 或者 RAM 的位置。

在链接的过程中可以得到 flash 中的 data 段有多大，当可执行文件运行时，这个段需要复制到 RAM 中；也可以得到 bbs 段需要多大的 RAM 空间（bbs 段是不占用 flash 空间的），也就是说没有赋值的全局变量在代码段中只有这个变量在 RAM 中的地址以及这个变量所用到的 RAM 的大小。所以 RAM 空间必须大于所需要的 data 和 bbs 空间，否则编译器会报错。

3.4 存储器中的各个段

➤ Flash 中的段：

vector 段：这个段一般被忽略了，这个是在 flash 的 0 地址处

text 段：代码段，包括进入 main 函数前的代码 init 段以及代码段

rodata 段：由 const 修饰的变量的值以及字符串，这个是不拷贝到 RAM 中的，当

然如果需要拷贝也是可以做到的。

data 段：全局赋值了的变量和静态变量，需要拷贝到 RAM 中

➤ RAM 中的段

data 段：将 flash 中的数据段拷贝到 RAM 中形成 RAM 中的数据段

bss 段：程序中没有赋值的全局变量，bss 段的长度是右所有的文件

heap 段：堆段

stack 段：栈段（一般是主栈段，进程栈一般不会使用这里的内存空间）

3.5 其他预备知识

这些预备知识例如：

- 怎么进入特权模式
 - MSR,MRS 访问特殊功能寄存器
 - When changing the stack pointer, software must use an ISB instruction immediately after the MSR instruction. This ensures that instructions after the ISB instruction execute using the new stack pointer.
 -
- 这些内容，阅读宋岩译的《Cortex-M3 权威指南(中文)》吧！或者官方原文《Cortex-M4 Devices Generic User Guide》

3.6 小结

- M4 核里面的寄存器没有地址一说，地址针对的是 M4 核总线能访问的地址
- M4 核规定了某些特定的外围控制块的地址
- 复位后中断向量表在 0 地址，可以通过设置中断向量偏移寄存器将中断向量表偏移
- SP 指针分为主栈指针和程序栈指针
- 主堆栈指针(MSP)：复位后缺省使用的堆栈指针，用于操作系统内核以及异常处理例程(包括中断服务例程)；进程堆栈指针(PSP)：由用户的应用程序代码使用或者操作系统的线程。
- 栈指针切换后，必须使用 ISB 指令进行强制命令隔离，即等待所有的操作完成后，才能使用新的栈指针
- 编译器连接的过程其实就是将各个.o 文件中的相同的段进行合并，计算 bbs 需要的 RAM 大小，计算 data 需要的 RAM 大小
- bbs 段只存在于 RAM，而 bbs 中的地址是 falsh 代码段中的全局未赋值变量的地址
- rodata 段可以不拷贝到 RAM 中，falsh 中的 data 段需要拷贝到 RAM 中。

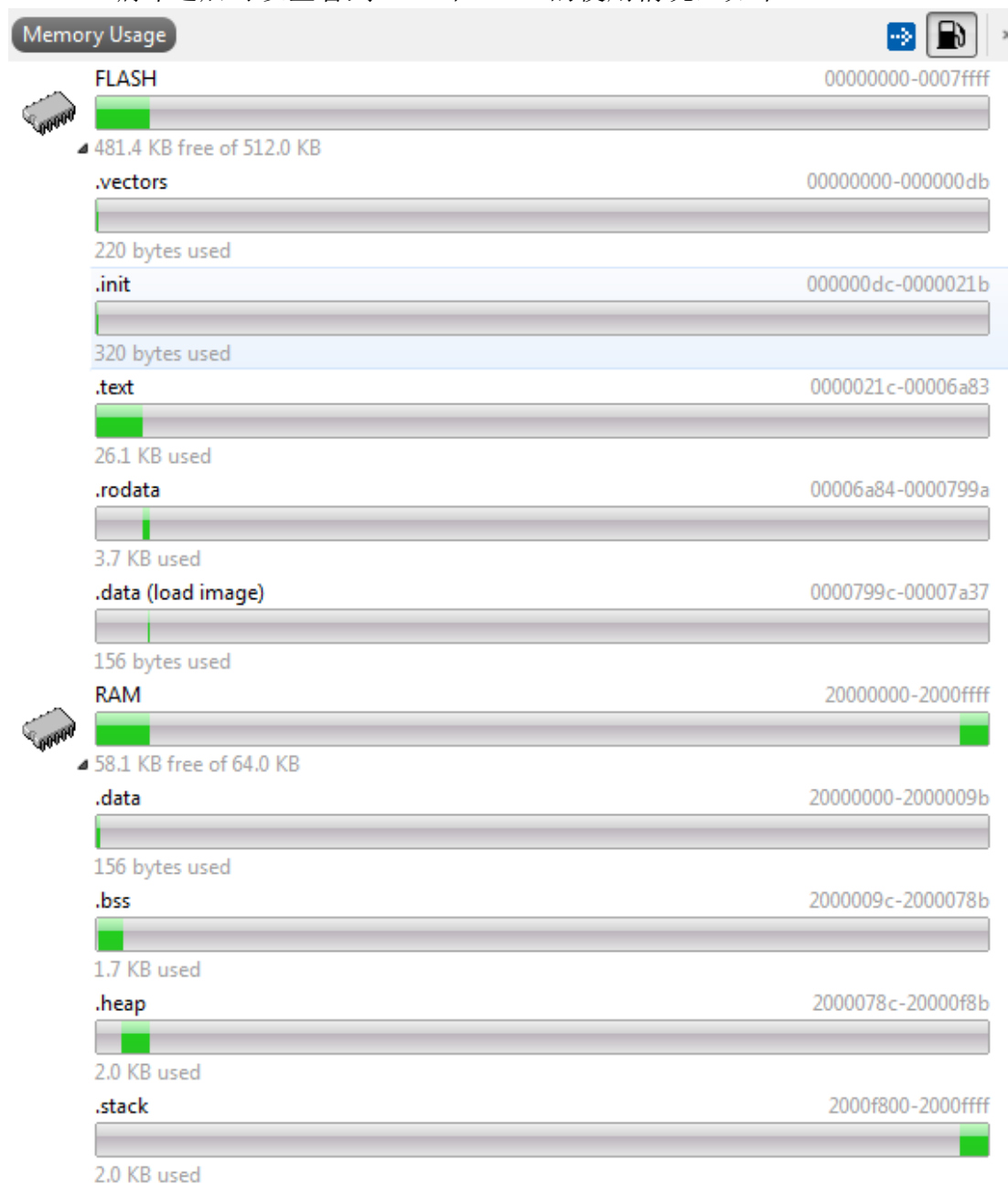
4 SES 启动代码分析

文中的工程使用的 IDE 是 Segger Embedded Studio，编译工具链是 GCC，硬件平台是 Nordic nrf52832(官方 PCA10040 开发板)，操作系统是 RT-Thread 3.1.0。本文中只讲解启动以及可能涉及到的 xml 文件以及链接文件等，不讨论 SES 的使用方法(见“Segger Embedded Studio 和 MDK keil 类比”文章)，也不讨论 RT-Thread 的使用方法。本文的工程作用是 nordic SDK 工程中的 blank 裸机工程，作用是使 PCA10040 板子上的 4 个灯每隔 500ms 进行流水亮灭。

为了便于分析，将启动文件尽可能简化了。

4.1 可执行文件的存储器使用信息

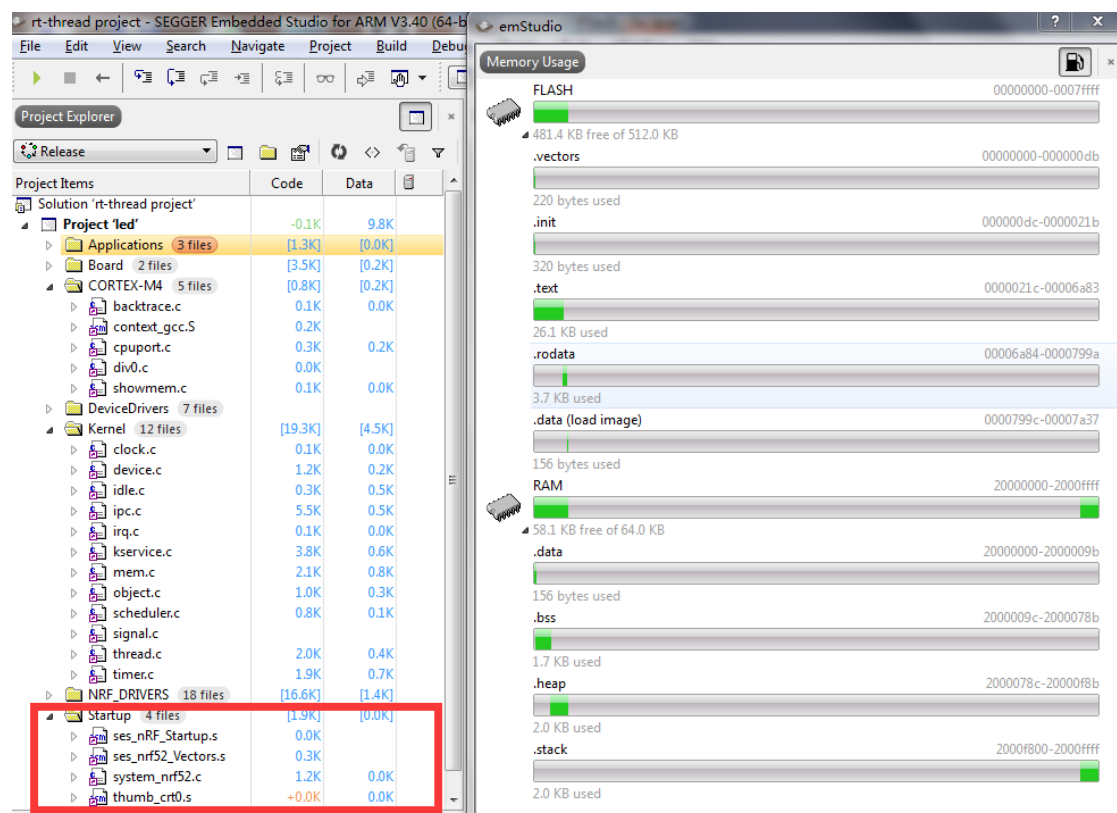
SES 编译之后可以查看到 flash 和 RAM 的使用情况，如下：



Name	Value	Range	Size
(No section)			
.bss	2000009c	2000009c-2000078b	1,776
.data	20000000	20000000-2000009b	156
.data_ram	20000000	20000000-2000009b	156
.heap	2000078c	2000078c-20000f8b	2,048
.init	000000dc	000000dc-0000021b	320
.rodata	00006a84	00006a84-0000799a	3,863
.stack	2000f800	2000f800-2000ffff	2,048
.text	0000021c	0000021c-00006a83	26,728
.vectors	00000000	00000000-000000db	220

上图中可以看到 flash 和 RAM 的使用情况，这形象的展示可能更加具有冲击感，接下来直接分析启动文件，分析的过程中会引出问题，同时再去解决问题。

启动文件在 SES 中被分为了 4 个文件，ses_nrf_Startup.s、ses_nrf52_Vectors.s、system_nrf52.c 以及 thumb_crt0.s 文件，这里主要分析 3 个汇编文件。



注：SES 的汇编文件可以使用 c 语言中的//和/* */进行注释。

4.2 ses_nRF_Startup.s 分析

```
.syntax unified //是一个指示,说明下面的指令是 ARM 和 THUMB 通用格式

.global Reset_Handler //声明本文件定义的一个符号,在其他文件可以使用这个符号
#ifdef INITIALIZE_USER_SECTIONS
.global InitializeUserMemorySections //用户需要在 main 之前拷贝的段
#endif

.extern _vectors //引用一个外部文件定义的符号_vectors

.section .init, "ax" //指定下面的代码属于.init 段,且该段具有可执行属性
.thumb_func //Thumb 模式下运行的函数必须在函数前使用该伪指令

.equ VTOR_REG, 0xE000ED08 //VTOR_REG = 0xE000ED08 这个地址是 cortex-m4 处理器强制规定的地址,所有使 m4 核的单片机都是这个地址。

.equ FPU_CPACR_REG, 0xE000ED88 //FPU_CPACR_REG= 0xE000ED88 FPU 的寄存器地址也是核固定了的
```

GUN 相关的链接: <https://sourceware.org/binutils/>和 <http://tigcc.ticalc.org/doc/>

<https://sourceware.org/binutils/docs-2.30/as/Global.html#Global>

<https://sourceware.org/binutils/docs-2.30/as/Extern.html#Extern>

<http://tigcc.ticalc.org/doc/gnuasm.html#SEC119>

<http://tigcc.ticalc.org/doc/gnuasm.html#SEC119>

.equ 参考 <https://sourceware.org/binutils/docs-2.22/as/Equ.html#Equ>

ARM 官方文档介绍下载:

http://infocenter.arm.com/help/topic/com.arm.doc.100166_0001_00_en/arm_cortexm4_processor_trm_100166_0001_00_en.pdf

4.1 System control registers

0xE000ED08	VTOR	RW	0x00000000	Vector Table Offset Register
------------	------	----	------------	------------------------------

0xE000ED88	CPACR	RW	-	Coprocessor Access Control Register
------------	-------	----	---	-------------------------------------

7.3.1 Floating Point system registers

Summary of the FP system registers in the Cortex-M4 processor, if your implementation includes the FPU.
All Cortex-M4 FPU registers are described in the *ARM®v7-M Architecture Reference Manual*.

Table 7-4 Cortex-M4 Floating Point system registers

Address	Name	Type	Reset	Description
0xE000EF34	FPCCR	RW	0xC0000000	FP Context Control Register
0xE000EF38	FPCAR	RW	-	FP Context Address Register
0xE000EF3C	FPDSCR	RW	0x00000000	FP Default Status Control Register
0xE000EF40	MVFR0	RO	0x10110021	Media and VFP Feature Register 0, MVFR0
0xE000EF44	MVFR1	RO	0x11000011	Media and VFP Feature Register 1, MVFR1

7.3.2 Enabling the FPU

Example code sequence for enabling the FPU in both privileged and user modes. The processor must be in privileged mode to read from and write to the CPACR.

Enabling the FPU

```
; CPACR is located at address 0xE000ED88

LDR.W  R0, 0xE000ED88

; Read CPACR

LDR    R1, [R0]

Set bits 20-23 to enable CP10 and CP11 coprocessors

ORR    R1, R1, #0xF << 20

; Write back the modified value to the CPACR

STR    R1, [R0]
```

具体寄存器介绍：
http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DUI0553A_cortex_m4_dgug.pdf


```
#ifndef STACK_INIT_VAL    //栈的初始值，这里如果没有定义栈的值
#define STACK_INIT_VAL __RAM_segment_end__ //就会使用__RAM_segment_end__符号
#endif

Reset_Handler: //复位函数入口
#ifndef NO_STACK_INIT //如果没有定义 NO_STACK_INIT 就执行下面的代码初始化主栈 MSP
    /* Initialise main stack */
    ldr r0, =STACK_INIT_VAL //将 RAM 的末地址给 cpu 的寄存器 R0。
    ldr r1, =0x7            //立即数 0x07 存到 r1
    bics r0, r1             //将 R0 的低 3 位清 0，目的是保证栈顶地址大于等于 8 字节。
    mov sp, r0             //将栈顶地址赋值给 SP
```

Coprocessor Access Control Register

The CPACR register specifies the access privileges for coprocessors. See the register summary in *Cortex-M4F floating-point system registers* for its attributes. The bit assignments are:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								CP11		CP10		Reserved																			

Table 4-50 CPACR register bit assignments

Bits	Name	Function
[31:24]	-	Reserved. Read as Zero, Write Ignore.
[2n+1:2n] for n values 10 and 11	CPn	Access privileges for coprocessor n. The possible values of each field are: 0b00 = Access denied. Any attempted access generates a NOCP UsageFault. 0b01 = Privileged access only. An unprivileged access generates a NOCP fault. 0b10 = Reserved. The result of any access is Unpredictable. 0b11 = Full access.
[19:0]	-	Reserved. Read as Zero, Write Ignore.

问题1 符号 **RAM segment end** 是哪里定义的?? ---->这个符号是 RAM 的结束地址
得到一个结论: **栈指针地址可以由代码进行修改。**

实际上这里不初始化 SP 也没有问题的，因为 MCU 会在上电复位自动从 0 地址获取值传给 SP 指针，所以要保证 0 地址处存放的地址是有效的栈地址就可以。

栈必须8字节对齐，但是为什么是8字节，不是4字节？这个问题我还没有理解


```
#endif

#ifndef NO_SYSTEM_INIT
    /* Initialise system */

    ldr r0, =SystemInit //这里是调用 system_nrf52.c 中的 SystemInit()函数初始化 nordic 的
                        //时钟和内存，保证能够在任何 C 启动之前进行初始化完成

    blx r0
#endif

#ifndef NO_VTOR_CONFIG //这里设置中断向量表的偏移量
/* Configure vector table offset register */
ldr r0, =VTOR_REG //将中断向量偏移寄存器赋值给 R0

#ifdef VECTORS_IN_RAM
ldr r1, =_vectors_ram
#else
ldr r1, =_vectors //将中断向量处理符号入口地址赋值给 r1 这里_vectors =0
#endif

str r1, [r0] //将 R1 中的 32 位数据写入以 R1 为地址的存储器中，也就是将实际的中断
向量表的入口地址赋值给中断向量偏移寄存器

#endif

#if (defined(__ARM_ARCH_FPV4_SP_D16__) || defined(__ARM_ARCH_FPV5_D16__))
&& !defined(NO_FPU_ENABLE)

/* Enable FPU */

ldr r0, =FPU_CPACR_REG //FPU 寄存器地址给 R0
```

这里有一个疑惑：[为什么 SystemInit 没有声明，可以直接被这里直接引用？](#)

符号	条件	关系到的标志位
EQ	相等(EQual)	Z==1
NE	不等（ NotEqual）	Z==0
CS/HS	进位(CarrySet) 无符号数高于或相同	C==1
CC/LO	未进位(CarryClear) 无符号数低于	C==0
MI	负数(MInus)	N==1
PL	非负数	N==0
VS	溢出	V==1
VC	未溢出	V==0
HI	无符号数大于	C==1 && Z==0
LS	无符号数小于等于	C==0 Z==1
GE	带符号数大于等于	N==V
LT	带符号数小于	N!=V
GT	带符号数大于	Z==0 && N==V
LE	带符号数小于等于	Z==1 N!=V
AL	总是	

<pre>ldr r1, [r0] //将 FPU 的寄存器地址内容给 R1 orr r1, r1, #(0xF << 20) //将 R1 的第 20 21 22 23 位设置为 1 str r1, [r0] //将 R1 中的数据写入 FPU 寄存器中 dsb //数据同步隔离.仅当所有在它前面的存储器访问操作都执行完 毕后, 才执行在它后面的指令 isb //指令同步隔离.最严格: 它会清洗流水线, 以保证所有它前面的 指令都执行完毕之后, 才执行它后面的指令 /* Jump to program start */ b _start</pre>	<table border="1"> <thead> <tr> <th>指令</th><th>功能描述</th></tr> </thead> <tbody> <tr> <td>DMB</td><td>数据存储器隔离。DMB 指令保证： 仅当所有在它前面的存储器访问都执行完毕后，才提交(commit)在它后面的存储器访问动作。</td></tr> <tr> <td>DSB</td><td>数据同步隔离。比 DMB 严格： 仅当所有在它前面的存储器访问都执行完毕后，才执行它在后面的指令（亦即任何指令都要等待）</td></tr> <tr> <td>ISB</td><td>指令同步隔离。最严格：它会清洗流水线，以保证所有它前面的指令都执行完毕之后，才执行它后面的指令</td></tr> </tbody> </table>	指令	功能描述	DMB	数据存储器隔离。DMB 指令保证： 仅当所有在它前面的存储器访问都执行完毕后，才提交(commit)在它后面的存储器访问动作。	DSB	数据同步隔离。比 DMB 严格： 仅当所有在它前面的存储器访问都执行完毕后，才执行它在后面的指令（亦即任何指令都要等待）	ISB	指令同步隔离。最严格：它会清洗流水线，以保证所有它前面的指令都执行完毕之后，才执行它后面的指令
指令	功能描述								
DMB	数据存储器隔离。DMB 指令保证： 仅当所有在它前面的存储器访问都执行完毕后，才提交(commit)在它后面的存储器访问动作。								
DSB	数据同步隔离。比 DMB 严格： 仅当所有在它前面的存储器访问都执行完毕后，才执行它在后面的指令（亦即任何指令都要等待）								
ISB	指令同步隔离。最严格：它会清洗流水线，以保证所有它前面的指令都执行完毕之后，才执行它后面的指令								

如果只是运行到用户函数这个文件可以简洁到如下：

<pre>.syntax unified .global Reset_Handler .section .init, "ax" .thumb_func Reset_Handler: /* Jump to program start */ b _start</pre>

这个文件主要做了以下几件事：

- SP 指针赋值
- 为进入用户代码设置系统时钟以及内存初始化
- 中断向量表偏移寄存器设置
- FPU 寄存器设置

这里主要想说明以下几个问题：

- CORTEX M4 内核规定了一些重要的外围设备的地址，例如：中断控制、浮点运算控制、系统时钟控制等等

- 系统时钟即便没有设置，
- 栈地址可以由程序员进行设置位置以及大小
- 栈地址如果没有被设置默认就是 **RAM** 的末地址
- 栈大小如果没有设置，默认栈地址依然是 **RAM** 的末地址，也就是栈顶和栈底相等，但是应用程序依然是可以运行的

4.3 thumb_crt0.s 分析

<pre> #ifndef APP_ENTRY_POINT #define APP_ENTRY_POINT main //用户 main 函数 #endif .syntax unified .global _start .extern APP_ENTRY_POINT .section .init, "ax" .code 16 .align 2 .thumb_func _start: /* Copy initialized memory sections into RAM (if necessary).*/ ldr r0, =__data_load_start__ //从 flash 中拷贝全局赋值变量到 ram, flash 的起始地址 ldr r1, =__data_start__ //RAM 中的起始地址 ldr r2, =__data_end__ //RAM 中的结束地址 bl memory_copy //跳转到内存拷贝 /* Zero the bss. 清 bss 段*/ ldr r0, =__bss_start__ //bss 段在 RAM 中的起始地址 </pre>	<p>用户程序 main 函数</p> <p>.init 段具有可执行属性</p> <p>全局赋值了的变量是必须要进行复制到 RAM 的。那么：</p> <p><u>data load start 在 flash 的地址是怎么来的？</u></p> <p><u>data start 在 RAM 中的起始地址是怎么来的？</u></p> <p><u>data end 在 RAM 中的结束地址是怎么来的？</u></p> <p>bbs 段可以不进行清理，那么：</p> <p><u>bss start 在 RAM 中的地址是怎么得到的？</u></p>
--	---

<pre> ldr r1, __bss_end__ //bss 段在 RAM 中的结束地址 movs r2, #0 //清零 bl memory_set //跳转到设置函数 .type start, function //start 为一个函数 start: /* Jump to application entry point */ movs r0, #0 //将 main 函数的入口参数清零 movs r1, #0 //将 main 函数的入口参数清零 ldr r2, =APP_ENTRY_POINT // blx r2 //跳转到 main 函数运行 .thumb_func memory_copy: cmp r0, r1 //比较 R0 和 R1 beq 2f //相等就跳转到符号 2 subs r2, r2, r1 //R2 = R2 - R1 即 R2 放的是长度 beq 2f //相等就跳转到符号 2 如果长度等于为 0 就不复制 1: ldrb r3, [r0] //加载 R0 地址中的 8bit 到 R3 adds r0, r0, #1 //将 R0 的地址加 1 源地址加 1 strb r3, [r1] //将 R3 的内容赋值给 R1 所指的地址 adds r1, r1, #1 //R1=R1+1 地址加 1 目的地址加 1 subs r2, r2, #1 //R2 = R2-1 长度减 1 </pre>	<p><u>bss end 在 RAM 中的地址是怎么得到的?</u></p> <p>这里 memory_set 函数是由汇编实现的，可以换成 C 语言实现</p> <p>跳转到用户代码运行</p> <p>内存复制函数，可以使用 c 语言代替这个函数： 3 个入口参数分别由 R0,R1,R2 传到这个函数。</p> <pre> void memory_copy(uint32_t fadd,uint32_t raddstart,uint32_t raddend) { uint32_t flash_add = fadd; uint32_t ram_add_start = raddstart; uint32_t ram_add_end = raddend; //uint8_t datas; uint16_t len = ram_add_end - ram_add_start; for(uint16_t i = 0;i<len;i++) { *((uint8_t*)ram_add_start) = *((uint8_t*)flash_add); flash_add++; } } </pre>
---	---

<pre> bne 1b //如果 R2 等 0 说明赋值完成，不等于 0 就继续赋值 长度=0 说 明复制完毕 2: bx lr .thumb_func memory_set: //内存设置 cmp r0, r1 beq 1f //如果 R0=R1 就返回 strb r2, [r0] //将 R2 内容赋值给 R0 所指向的地址 adds r0, r0, #1 //R0 的地址加 1 b memory_set 1: bx lr </pre>	<pre> ram_add_start++; } } 将内存设置，可以使用 c 语言函数进行替代： 2 个入口参数分别由 R0,R1 传到这个函数。 void memory_set(uint32_t raddstart,uint32_t raddend) { uint32_t ram_add_start = raddstart; uint32_t ram_add_end = raddend; //uint8_t datas; uint16_t len = ram_add_end - ram_add_start; for(uint16_t i = 0;i<len;i++) { *((uint8_t*)ram_add_start) = 0; ram_add_start++; } } </pre>
--	---

这个文件本并不是只有这么一点点代码，还有许多其他操作，我将其进行了删除，做了减法，以便能看到想要的东西。如果将内存设置和内存拷贝的函数用 c 语言进行代替，那么以上文件可以简化为：

<pre> .syntax unified .global _start .extern APP_ENTRY_POINT .section .init, "ax" </pre>	
--	--

```
.code 16

.align 2

.thumb_func

_start:

/* Copy initialized memory sections into RAM (if necessary).*/
ldr r0, =__data_load_start__
ldr r1, =__data_start__
ldr r2, =__data_end__
bl memory_copy

/* Zero the bss. */
ldr r0, =__bss_start__
ldr r1, =__bss_end__
movs r2, #0
bl memory_set

.type start, function

start:

/* Jump to application entry point */
movs r0, #0
movs r1, #0
ldr r2, =APP_ENTRY_POINT
blx r2
```

全局赋值变量进行拷贝

BBS 段进行清理，这个清除操作都可以免去

跳转到用户 main 函数

这个文件主要做了以下几件事：

- 全局赋值变量拷贝到 **RAM** 中
- **BBS** 段进行清零
- 跳转到用户函数运行

这里主要想说明以下几个问题：

- 全局赋值变量是必须要拷贝到 **RAM** 中的
- **BBS** 段可以不进行清零，但是一般建议清零
- 启动文件中的汇编语言函数也可以用 c 语言同功能函数代替

4.4 ses_nrf52_Vectors.s 分析

这个文件主要是中断向量表的建立，以及中断函数的弱定义。

<pre> .syntax unified .code 16 .section .init, "ax" .align 0 /***** ** * Default Exception Handlers * *****/ .thumb_func .weak NMI_Handler NMI_Handler: b . .thumb_func .weak HardFault_Handler HardFault_Handler: b . </pre>	<pre> .section Syntax: .section name[, "flags"] or .section name[, subsegment] Use the .section directive to assemble the following code into a section named name. If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized: b: bss section (uninitialized data) n: section is not loaded w: writable section d: data section r: read-only section x: executable section m: mergeable section (TIGCC extension, symbols in the section are considered mergeable constants) u: unaligned section (TIGCC extension, the contents of the section need not be aligned) s: shared section (meaningful for PE targets, useless for TIGCC) a: ignored (for compatibility with the ELF version) If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to be loaded and writable. Note the n and w flags remove </pre>
---	--

<pre> .ooothumb_func .weak FPU_IRQHandler FPU_IRQHandler: b . /***** ** * Vector Table * ** *****/ .section .vectors, "ax" .align 0 .global _vectors .extern __stack_end__ .extern Reset_Handler _vectors: .word __stack_end__ </pre>	<p>attributes from the section, rather than adding them, so if they are used on their own it will be as if no flags had been specified at all. If the optional argument to the .section directive is not quoted, it is taken as a subsegment number (see Sub-Sections).</p> <p>以下为.vectors 段，具有可执行属性</p> <p><u>__stack_end__</u>这个符号即为栈指针，这符号怎么来的？</p>
---	--

```
.word Reset_Handler
.word NMI_Handler
.word HardFault_Handler
.word MemoryManagement_Handler
.word BusFault_Handler

.....

.....

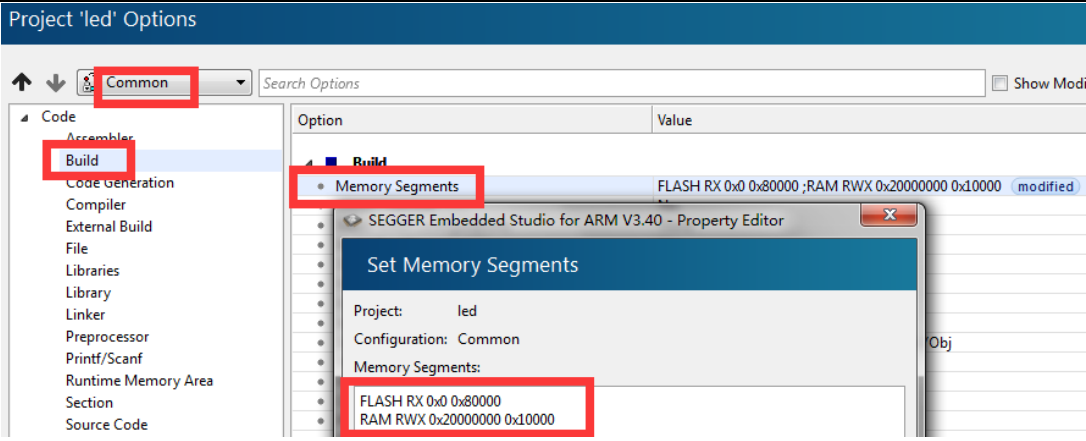
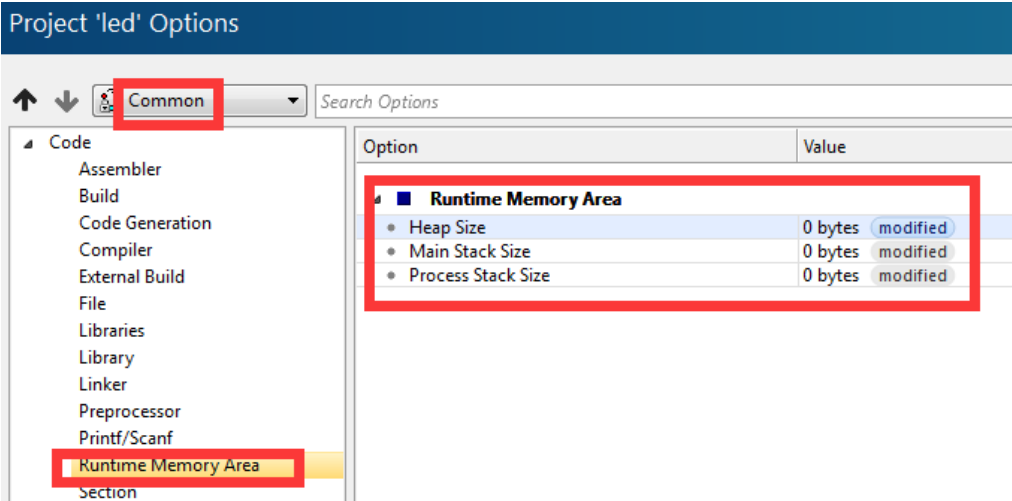
.word I2S_IRQHandler
.word FPU_IRQHandler
_vectors_end:
```

这个文件很简单，主要了解什么是函数弱定义，也就是说，其他地方可以将中断函数使用同样的中断函数名再次进行函数定义，调用时便会调用程序员定义的中断函数，如果没有定义，就使用这个文件中弱定义的中断函数

5 工程中的符号

整个工程中有许多的符号，这些符号来源于：工程设置中的预处理宏、汇编代码定义、应用代码定义、工程设置、链接文件定义以及工程特定的文件，例如 SES 是 XML 文件，MDK KEIL 分散 SCT 文件都会带来符号，这些符号可能在不同的地方以不同的名称显示，但是**这符号最终都是需要化为地址(flash 和 ram 的地址)**，这样可能符号的名称不同，但是地址可以是一样的，同样有些符号的地址也可能是用另外的符号计算得到的地址。

5.1 工程设置符号

来源	符号	地址(16 进制)	说明
	__FLASH_segment_end__	0x00080000	
	__FLASH_segment_size__	0x00080000	
	__FLASH_segment_start__	0x00000000	
	__FLASH_segment_used_end__	0x000074b0	
	__FLASH_segment_used_size__	0x000074b0	
	__RAM_segment_end__	0x20010000	
	__RAM_segment_size__	0x00010000	
	__RAM_segment_start__	0x20000000	
	__RAM_segment_used_end__	0x20010000	
	__RAM_segment_used_size__	0x00010000	
工程设置	__HEAPSIZE__	0x00000000	
	__STACKSIZE_PROCESS__	0x00000000	
	__STACKSIZE__	0x00000000	

5.2 section 符号

下面的符号基本都是由 XML 文件字节定义或者演变而成的，起地址都是在 XML 文件中定义的。

段	符号	地址	说明
.bss	__bss_end__	0x20000788	代码中全局没有被初始化的变量
	__bss_load_end__	0x20000788	
	__bss_start__	0x20000098	
.data	__data_end__	0x20000098	代码中全局被初始化的变量
	__data_start__	0x20000000	
	__data_load_start__	0x00007418	
.data_ram	__data_ram_end__	0x20000098	拷贝代码中全局被初始化的变量预留的 RAM
	__data_ram_load_end__	0x20000098	
	__data_ram_start__	0x20000000	
.rodata	__rodata_end__	0x00007417	字符串以及 const 修饰的变量
	__rodata_load_end__	0x00007417	
	__rodata_start__	0x00006500	
.stack	__stack_end__	0x20010000	栈，这个地址默认指向的是 ram 的末地址，大小是由工程设置 __STACKSIZE__ 决定的。这个默认位置也是可以修改的。
	__stack_load_end__	0x20010000	
	__stack_start__	0x20010000	
.text	__text_end__	0x00006500	代码段的起始地址是紧跟着中断向量表的末尾的
	__text_load_end__	0x00006500	
	__text_start__	0x00000174	
.init	__init_end__	0x00000174	这个 init 段起始就是在复位后到 main 之间的代码，起始也属于代码段。它的起始地址和 .text 的地址一样
	__init_load_end__	0x00000174	
	__init_start__	0x000000dc	
.vectors	__vectors_end__	0x000000dc	中断向量表，它的起始地址也是在 XML 文件中进行定义的
	__vectors_load_end__	0x000000dc	
	__vectors_start__	0x00000000	

6 XML 文件详解(Section Placement file)

SES 使用的 XML 语法作为段定位文件。参考 https://www.segger.com/downloads/embedded-studio/EmbeddedStudio_Manual 中附件的 **Section Placement file format** 章节

文件的第一行为文档类型: `<!DOCTYPE Linker_Placement_File>`

第二行为文件根, 一个段定位文件只能有一个根: `<Root name="Flash Section Placement">`, Root 有一个名字属性。

在这个根下可以有多个内存段(MemorySegment), 每个内存段下可有多程序段(ProgramSection), ProgramSection 的有一些属性, 都有一个名字属性。

ProgramSection 共有 *name*、*start*、*address_symbol*、*end_symbol*、*alignment*、*fill*、*inputsections*、*keep*、*load*、*place_from_segment_end*、*runin*、*runoffset*、*size*、*size_symbol* 和 *start* 等属性, 当然这些属性有些是可选的, 有些是必须的。

6.1 name 名字属性

名字属性在 Root、MemorySegment 以及 ProgramSection 都有的一个元素, 且这个属性是必须的, 使用方法是 `name="xxx"`, 这个名字需要唯一, 在一个段定位文件中。双引号中不能以数组开头, 一般以字母或者.xxx。例如:

```
<Root name="Flash Section Placement">
  <MemorySegment name="FLASH" start="$ (FLASH_PH_START)" size="$ (FLASH_PH_SIZE)" >
    <ProgramSection alignment="0x100" load="Yes" name=".vectors" start="$ (FLASH_START)" />
```

6.2 start 起始地址属性

start 属性是一个 MemorySegment 以及 ProgramSection 的属性, 后面双引号中的值必须是以 0x 开头的 16 进制数据。表示这个 name 属性段的起始地址是多少。例如 `start="0x20010000"`, 或者 `start="$ (FLASH_PH_START)"`, 而 FLASH_PH_START 是一个 16 进制的宏。

6.3 size 段具体大小属性

这个用来表明 name 段具体的大小, 以字节为单位。它的值必须是以 0x 开头的 16 进制数据。

6.4 address_symbol 地址开始符号属性

address_symbol 为地址开始符号, 注意这里是符号, 和上面 start 的属性的不同在于, start 给是固定的地址, 而这里的 address_symbol 是表示 name 属性段的起始地址的符号, 这个符号是在编译链接的过程中用到, 并确定这个符号的值。例如: `address_symbol="__StackLimit"`

6.5 end_symbol 地址结束符号属性

end_symbol 和上面的 address_symbol 相对应, 也就是是 name 属性段的接收地址的

符号。例如：end_symbol="__StackTop"

6.6 size_symbol 段大小符号属性

这个用来表示一个段大小的符号，也就是 end_symbol - address_symbol 的值的符号。

6.7 alignment 访问对齐属性

alignment 对齐方式，以字节对齐，强制规定 name 属性段的以多数个字节对齐，使用方法：alignment="0x100"，双引号中的值必须是以 0x 开头的 16 进制数据。

6.8 fill 填充属性

这是一个可选的属性，用于填充内存中没有指定范围的值，使用方法：fill="0xff"，双引号中必须是以 0x 开头的 16 进制数据。

6.9 inputsections 输入哪些文件到 name 段

这个属性一般是不用声明的，当 SES 编译时会用到这个 XML 文件中的符号，而在链接时会生成连接文件，链接文件其实就是将各个文件按照编译出来的东西(代码、常量/赋值了的全局变量、为初始化的全局变量)，把这些文件进行组合放到指定的地方。而 inputsections 的作用是告诉链接文件，name 段里存放的只有特定的文件后缀或特定的文件名称，也就是通过 inputsections 来缩小存放到 name 段的文件。

一般情况先.text, .ctors, .ctors, .data, .rodata, 或 bss 是不允许使用 inputsections 进行属性限制的，这些段可放的文件名一般是*(.name.name.*)。例如：

```
__init_load_start__ = ALIGN(__vectors_end__ , 4);
__init ALIGN(__vectors_end__ , 4) : AT(ALIGN(__vectors_end__ , 4))
{
    __init_start__ = .;
    *(.init .init.*)
}
```

(.init .init.) 的第一个*表示所有的.O 文件，括号里面的表示这个段中存放的是所有.O 文件中 section 名是.init 或者代码中带有.init.*(*也为通配符)的符号都放到这个段中。

这里的*(.init .init.*)就是编译器默认存放.init 段名的代码或者数据。假设我只想存放一个特定的段名或者特定的符号到相应的段中呢？？这个时候 inputsections 就起作用了，例如：

没有使用 inputsections 说明属性段生成的连接件。

```
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".fs_data" runin=".fs_data_run"/>
__fs_data_load_start__ = ALIGN(__nrf_sections_end__ , 4);
__fs_data ALIGN(__nrf_sections_run_end__ , 4) : AT(ALIGN(__nrf_sections_end__ , 4))
{
    __fs_data_start__ = .;
    KEEP(*(.fs_data .fs_data.*))
}
```

使用 inputsections 说明属性段生成的连接件。

```
<ProgramSection alignment="4" keep="Yes" load="Yes" name=".fs_data" inputsections="*(.fs_data*)" runin=".fs_data_run"/>
```

```
__fs_data_load_start__ = ALIGN(__nrf_sections_end__, 4);
__fs_data ALIGN(__nrf_sections_run_end__, 4) : AT(ALIGN(__nrf_sections_end__, 4))
{
    __fs_data_start__ = .;
    KEEP(*(__fs_data*))
}
```

也就是指定了输入到这个段的段文件名为*(.fs_data*)。

6.10 keep 保持属性

keep 如果等于"YES"的作用是 name 属性段中的符号即使没有别程序代码调用，也将这个 ProgramSection 属性保留。否则 NO 就是没有使用就丢弃，不在连接文件中体现。

6.11 load 加载属性

如果 load="YES"，这个段将会被加载到 RAM，所以这个属性只能是 flash 可以设置为 YES，如果是 RAM 内的程序段必须将这个属性设置为 NO。

6.12 place_from_segment_end 段末尾开始放数据属性

这个其实就是为栈 stack 准备的属性，ARM 一般都是满减栈，也就是栈地址向下生长。值是 YES 或者 NO，默认其实就 NO。例如：

```
<ProgramSection alignment="8" size="__STACKSIZE__" load="No" place_from_segment_end="Yes"
name=".stack" address_symbol="__StackLimit" end_symbol="__StackTop"/>
```

6.13 runin 属性

这个用来将 flash 中 runin 定义的段名字拷贝到对应的 RAM 中的同名字的名字段。例如：data 段拷贝到.data_run 段。

```
<!DOCTYPE Linker_Placement_File>
<Root name="Flash Section Placement">
  <MemorySegment name="FLASH" start="$(FLASH_PH_START)" size="$(FLASH_PH_SIZE)">
    .....
    <ProgramSection alignment="4" load="Yes" runin=".data_run" name=".data" />
    .....
  </MemorySegment>
  <MemorySegment name="RAM" start="$(RAM_PH_START)" size="$(RAM_PH_SIZE)">
    .....
    <ProgramSection alignment="4" load="No" name=".data_run" />
    .....
  </MemorySegment>
</Root>
```

6.14 runoffset 属性

这个属性的作用是 name 段加载到 ram 中时，加载的地址不是从 name 段的起始地址进行加载，而是从起始地址的偏移量 runoffset 设置值进行偏移加载。这个值必须是

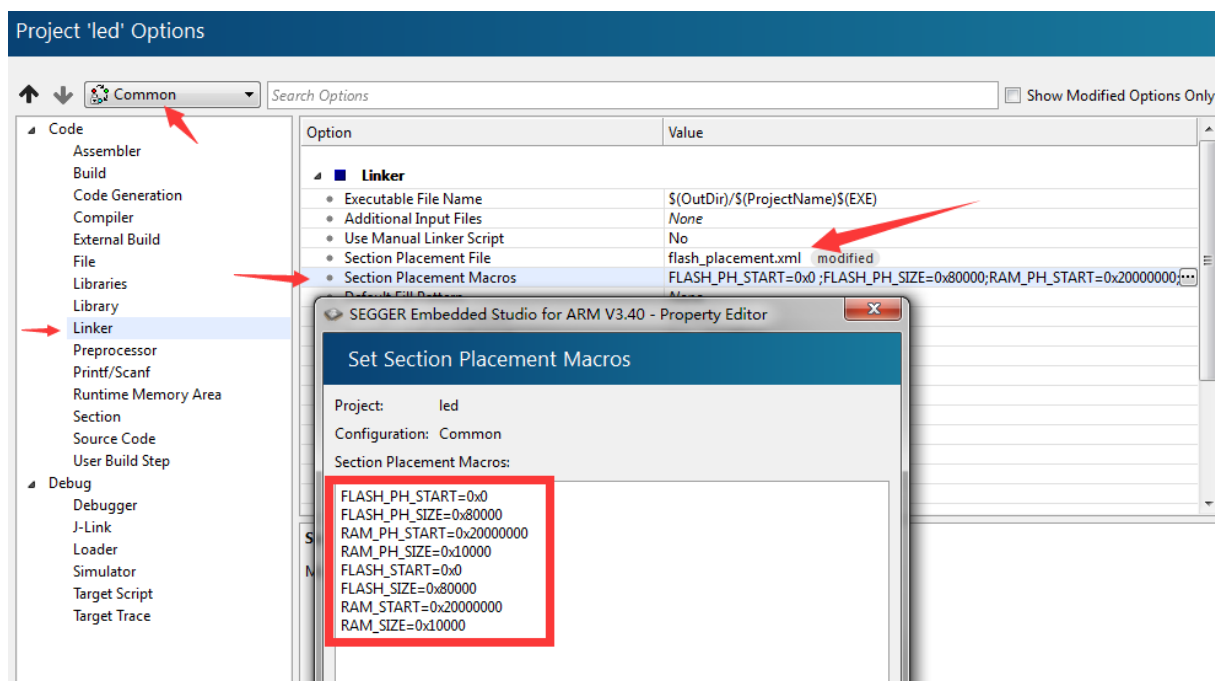
以 0x 开头的 16 进制数据。

7 本文工程 XML 分析

先贴一张本文工程 XML 文件的内容：

```
<!DOCTYPE Linker_Placement_File>
<Root name="Flash Section Placement">
  <MemorySegment name="FLASH" start="$ (FLASH_PH_START)" size="$ (FLASH_PH_SIZE)">
    <ProgramSection alignment="0x100" load="Yes" name=".vectors" start="$ (FLASH_START)" />
    <ProgramSection alignment="4" load="Yes" name=".init" />
    <ProgramSection alignment="4" load="Yes" name=".text" />
    <ProgramSection alignment="4" load="Yes" name=".rodata" />
    <ProgramSection alignment="4" load="Yes" runin=".data_ram" name=".data" />
  </MemorySegment>
  <MemorySegment name="RAM" start="$ (RAM_PH_START)" size="$ (RAM_PH_SIZE)">
    <ProgramSection alignment="4" load="No" name=".data_ram" />
    <ProgramSection alignment="4" load="No" name=".bss" />
    <ProgramSection alignment="8" size="__STACKSIZE__" load="No" place_from_segment_end="Yes" name=".stack"/>
  </MemorySegment>
</Root>
```

上图中有一些符号，是编辑器进行设置的，如下图：



下面对其注释：

```
<!DOCTYPE Linker_Placement_File> //文件类型
<Root name="Flash Section Placement"> //根开始
  <MemorySegment name="FLASH" start="$(FLASH_PH_START)" size="$(FLASH_PH_SIZE)">
    //第一个存储段，名字是 flash，起始地址是符号 FLASH_PH_START，大小为符号 FLASH_PH_SIZE
    <ProgramSection alignment="0x100" load="Yes" name=".vectors" start="$(FLASH_START)" />
    //中断向量段，256 字节对齐，这里强制指定起始地址为 FLASH_START，也就是 falsh 的起始地址，这里需要加载，
    //实际上是无法加载的，因为 RAM 中没有对应的段空间给他，
    //在代码中对应的符号是 __vectors_start__、__vectors_end__ 和 __vectors_load_end__
    <ProgramSection alignment="4" load="Yes" name=".init" />
    //初始化段，起始地址是紧跟在 __vectors_end__ 之后，4 字节对齐，虽然需要加载，但是同样的加载会失败，
    //在代码中对应的符号有 __init_start__、__init_end__ 和 __text_load_end__
    <ProgramSection alignment="4" load="Yes" name=".text" />
    //代码段，起始地址紧跟着 __init_end__，
    //代码中对应的符号有 __text_start__、__text_end__ 和 __text_load_end__
    <ProgramSection alignment="4" load="Yes" name=".rodata" />
    //只读数据段、起始地址紧跟着 __text_end__，
    //代码中对应的符号有 __rodata_start__、__rodata_end__ 和 __rodata_load_end__
    <ProgramSection alignment="4" load="Yes" runin=".data_ram" name=".data" />
    //数据段、起始地址紧跟着 __rodata_end__，并且这个能成功加载到 RAM 中，
    //并且会加载到 RAM 中名为 ".data_ram" 的段，
    //在代码中对应的符号有 __data_start__、__data_load_start__ 和 __data_end__
  </MemorySegment> //flash 段结束
  <MemorySegment name="RAM" start="$(RAM_PH_START)" size="$(RAM_PH_SIZE)">
    //第二个存储段，名字为 RAM，起始地址是符号 RAM_PH_START，大小是符号 RAM_PH_SIZE
    <ProgramSection alignment="4" load="No" name=".data_ram" />
    //数据段，用来存放从 falsh 中拷贝过来的数据段，不需要加载
    //在代码中对应的符号有 __data_ram_start__、__data_ram_end__ 和 __data_ram_load_end__
```

```

<ProgramSection alignment="4" load="No" name=".bss" />
//BBS 段，这段内存用来存放全局未定义的变量，在进入 main 函数之前这段内存进行清零，不需要加载
//在代码中对应的符号有 __bss_start__、__bss_end__ 和 __bss_load_end__
<ProgramSection alignment="8" size="__STACKSIZE__" load="No" place_from_segment_end="Yes" name=".stack"/>
//栈段，8 字节对齐，长度为符号 __STACKSIZE__，不需要加载，同时访问从结束地址开始，
//在代码中对应的符号有 __stack_start__、__stack_end__ 和 __stack_load_end__
</MemorySegment>
</Root>
<!DOCTYPE Linker_Placement_File> //文件类型
<Root name="Flash Section Placement"> //根开始
<MemorySegment name="FLASH" start="$(FLASH_PH_START)" size="$(FLASH_PH_SIZE)">
//第一个存储段，名字是flash，起始地址是符号FLASH_PH_START，大小为符号FLASH_PH_SIZE
<ProgramSection alignment="0x100" load="Yes" name=".vectors" start="$(FLASH_START)" />
//中断向量段，256字节对齐，这里强制指定起始地址为FLASH_START，也就是falsh的起始地址，这里需要加载，
//实际上是无法加载的，因为RAM中没有对应的段空间给他，
//在代码中对应的符号是 __vectors_start__、__vectors_end__ 和 __vectors_load_end__
<ProgramSection alignment="4" load="Yes" name=".init" />
//初始化段，起始地址是紧跟在 __vectors_end__ 之后，4字节对齐，虽然需要加载，但是同样的加载会失败，
//在代码中对应的符号有 __init_start__、__init_end__ 和 __text_load_end__
<ProgramSection alignment="4" load="Yes" name=".text" />
//代码段，起始地址紧跟着 __init_end__，
//代码中对应的符号有 __text_start__、__text_end__ 和 __text_load_end__
<ProgramSection alignment="4" load="Yes" name=".rodata" />
//只读数据段、起始地址紧跟着 __text_end__，
//代码中对应的符号有 __rodata_start__、__rodata_end__ 和 __rodata_load_end__
<ProgramSection alignment="4" load="Yes" runin=".data_ram" name=".data" />
//数据段、起始地址紧跟着 __rodata_end__，并且这个能成功加载到RAM中，
//并且会加载到RAM中名为".data_ram"的段，
//在代码中对应的符号有 __data_start__、__data_load_start__ 和 __data_end__
</MemorySegment> //flash段结束
<MemorySegment name="RAM" start="$(RAM_PH_START)" size="$(RAM_PH_SIZE)">
//第二个存储段，名字为RAM，起始地址是符号RAM_PH_START，大小是符号RAM_PH_SIZE
<ProgramSection alignment="4" load="No" name=".data_ram" />
//数据段，用来存放从falsh中拷贝过来的数据段，不需要加载
//在代码中对应的符号有 __data_ram_start__、__data_ram_end__ 和 __data_ram_load_end__
<ProgramSection alignment="4" load="No" name=".bss" />
//BBS段，这段内存用来存放全局未定义的变量，在进入main函数之前这段内存进行清零，不需要加载
//在代码中对应的符号有 __bss_start__、__bss_end__ 和 __bss_load_end__
<ProgramSection alignment="8" size="__STACKSIZE__" load="No" place_from_segment_end="Yes" name=".stack"/>
//栈段，8字节对齐，长度为符号 __STACKSIZE__，不需要加载，同时访问从结束地址开始，
//在代码中对应的符号有 __stack_start__、__stack_end__ 和 __stack_load_end__
</MemorySegment> //ram段结束
</Root>

```

8 工程的链接文件

```
MEMORY
{
    UNPLACED_SECTIONS (wx) : ORIGIN = 0x100000000, LENGTH = 0
    RAM (wx) : ORIGIN = 0x20000000, LENGTH = 0x00010000
    FLASH (wx) : ORIGIN = 0x00000000, LENGTH = 0x00080000
}

SECTIONS
{
    __RAM_segment_start__ = 0x20000000;
    __RAM_segment_end__ = 0x20010000;
    __RAM_segment_size__ = 0x00010000;
    __FLASH_segment_start__ = 0x00000000;
    __FLASH_segment_end__ = 0x00080000;
    __FLASH_segment_size__ = 0x00080000;

    __HEAPSIZE__ = 0;
    __STACKSIZE_PROCESS__ = 0;
    __STACKSIZE__ = 0;

    __vectors_load_start__ = 0x0;
    .vectors 0x0 : AT(0x0)
    {
        __vectors_start__ = .;
    }
}
```

```

    *(.vectors.vectors.*)
}
__vectors_end__ = __vectors_start__ + SIZEOF(.vectors);
__vectors_size__ = SIZEOF(.vectors);
__vectors_load_end__ = __vectors_end__;

. = ASSERT(__vectors_start__ == __vectors_end__ || (__vectors_end__ >= __FLASH_segment_start__ && __vectors_end__ <=
__FLASH_segment_end__), "error: .vectors is too large to fit in FLASH memory segment");

__init_load_start__ = ALIGN(__vectors_end__, 4);
.init ALIGN(__vectors_end__, 4) : AT(ALIGN(__vectors_end__, 4))
{
    __init_start__ = .;
    *(.init.init.*)
}
__init_end__ = __init_start__ + SIZEOF(.init);
__init_size__ = SIZEOF(.init);
__init_load_end__ = __init_end__;

. = ASSERT(__init_start__ == __init_end__ || (__init_end__ >= __FLASH_segment_start__ && __init_end__ <=
__FLASH_segment_end__), "error: .init is too large to fit in FLASH memory segment");

__text_load_start__ = ALIGN(__init_end__, 4);
.text ALIGN(__init_end__, 4) : AT(ALIGN(__init_end__, 4))
{
    __text_start__ = .;
    *(.text.text.*.glue_7t.glue_7.gnu.linkonce.t.*.gcc_except_table.ARM.extab*.gnu.linkonce.armextab.*)

```

```

}
__text_end__ = __text_start__ + SIZEOF(.text);
__text_size__ = SIZEOF(.text);
__text_load_end__ = __text_end__;

. = ASSERT(__text_start__ == __text_end__ || (__text_end__ >= __FLASH_segment_start__ && __text_end__ <=
__FLASH_segment_end__), "error: .text is too large to fit in FLASH memory segment");

__rodata_load_start__ = ALIGN(__text_end__, 4);
.rodata ALIGN(__text_end__, 4) : AT(ALIGN(__text_end__, 4))
{
    __rodata_start__ = .;
    *(.rodata .rodata.* .gnu.linkonce.r.*)
}
__rodata_end__ = __rodata_start__ + SIZEOF(.rodata);
__rodata_size__ = SIZEOF(.rodata);
__rodata_load_end__ = __rodata_end__;

. = ASSERT(__rodata_start__ == __rodata_end__ || (__rodata_end__ >= __FLASH_segment_start__ && __rodata_end__ <=
__FLASH_segment_end__), "error: .rodata is too large to fit in FLASH memory segment");

__data_load_start__ = ALIGN(__rodata_end__, 4);
.data ALIGN(__RAM_segment_start__, 4) : AT(ALIGN(__rodata_end__, 4))
{
    __data_start__ = .;
    *(.data .data.* .gnu.linkonce.d.*)
}

```

```

__data_end__ = __data_start__ + SIZEOF(.data);
__data_size__ = SIZEOF(.data);
__data_load_end__ = __data_load_start__ + SIZEOF(.data);

__FLASH_segment_used_end__ = ALIGN(__rodata_end__, 4) + SIZEOF(.data);
__FLASH_segment_used_size__ = __FLASH_segment_used_end__ - __FLASH_segment_start__;

. = ASSERT(__data_load_start__ == __data_load_end__ || (__data_load_end__ >= __FLASH_segment_start__ &&
__data_load_end__ <= __FLASH_segment_end__), "error: .data is too large to fit in FLASH memory segment");

.data_ram ALIGN(__RAM_segment_start__, 4) (NOLOAD) :
{
    __data_ram_start__ = .;
    . = MAX(__data_ram_start__ + SIZEOF(.data), .);
}
__data_ram_end__ = __data_ram_start__ + SIZEOF(.data_ram);
__data_ram_size__ = SIZEOF(.data_ram);
__data_ram_load_end__ = __data_ram_end__;

. = ASSERT(__data_ram_start__ == __data_ram_end__ || (__data_ram_end__ >= __RAM_segment_start__ && __data_ram_end__
<= __RAM_segment_end__), "error: .data_ram is too large to fit in RAM memory segment");

__bss_load_start__ = ALIGN(__data_ram_end__, 4);
.bss ALIGN(__data_ram_end__, 4) (NOLOAD) : AT(ALIGN(__data_ram_end__, 4))
{
    __bss_start__ = .;
    *(.bss .bss.* .gnu.linkonce.b.*) *(COMMON)

```



```

}
__bss_end__ = __bss_start__ + SIZEOF(.bss);
__bss_size__ = SIZEOF(.bss);
__bss_load_end__ = __bss_end__;

. = ASSERT(__bss_start__ == __bss_end__ || (__bss_end__ >= __RAM_segment_start__ && __bss_end__ <=
__RAM_segment_end__), "error: .bss is too large to fit in RAM memory segment");

__stack_load_start__ = __RAM_segment_end__ - 0;
.stack __RAM_segment_end__ - 0 (NOLOAD) : AT(__RAM_segment_end__ - 0)
{
    __stack_start__ = .;
    *(.stack.stack.*)
    . = ALIGN(MAX(__stack_start__ + __STACKSIZE__, .), 8);
}
__stack_end__ = __stack_start__ + SIZEOF(.stack);
__stack_size__ = SIZEOF(.stack);
__stack_load_end__ = __stack_end__;

__RAM_segment_used_end__ = __RAM_segment_end__ - 0 + SIZEOF(.stack);
__RAM_segment_used_size__ = __RAM_segment_used_end__ - __RAM_segment_start__;

. = ASSERT(__stack_start__ == __stack_end__ || (__stack_end__ >= __RAM_segment_start__ && __stack_end__ <=
__RAM_segment_end__), "error: .stack is too large to fit in RAM memory segment");
. = ASSERT(__bss_end__ <= __stack_start__, "error: section .bss overlaps absolute placed section .stack");
}

```

9 本文工程所有符号

text 的符号删除了，否则这个附件太大。

(No section)	地址		段	代码大小	data 大小	rodata 大小
NVIC_INT_CTRL	0xe000ed04					
NVIC_PENDSVSET	0x10000000					
NVIC_PENDSV_PRI	0x00ff0000					
NVIC_SYSPRI2	0xe000ed20					
SCB_VTOR	0xe000ed08					
__bss_load_start__	0x20000098					
__bss_size__	0x000006f0					
__data_load_end__	0x000074b0					
__data_load_start__	0x00007418					
__data_ram_size__	0x00000098					
__data_size__	0x00000098					
__FLASH_segment_end__	0x00080000					
__FLASH_segment_size__	0x00080000					
__FLASH_segment_start__	0x00000000					
__FLASH_segment_used_end__	0x000074b0					
__FLASH_segment_used_size__	0x000074b0					
__HEAPSIZE__	0x00000000					
__init_load_start__	0x000000dc					
__init_size__	0x00000098					
__RAM_segment_end__	0x20010000					
__RAM_segment_size__	0x00010000					

__RAM_segment_start__	0x20000000					
__RAM_segment_used_end__	0x20010000					
__RAM_segment_used_size__	0x00010000					
__rodata_load_start__	0x00006500					
__rodata_size__	0x00000f17					
__STACKSIZE_PROCESS__	0x00000000					
__STACKSIZE__	0x00000000					
__stack_load_start__	0x20010000					
__stack_size__	0x00000000					
__text_load_start__	0x00000174					
__text_size__	0x0000638c					
__vectors_load_start__	0x00000000					
__vectors_size__	0x000000dc					
__vfprintf	0x00000000					
__vscanf	0x00000000					
.bss	0x20000098	1776			1776	
heap_end	0x200003e4	4	.bss		4	
heap_ptr	0x200003e0	4	.bss		4	
heap_sem	0x200003ec	28	.bss		28	
idle	0x20000144	120	.bss		120	
lfree	0x200003e8	4	.bss		4	
max_mem	0x20000410	4	.bss		4	
mem_size_aligned	0x20000408	4	.bss		4	
m_cb	0x200000ac	88	.bss		88	
m_cb	0x20000104	8	.bss		8	
m_cb	0x2000010c	48	.bss		48	

m_clock_cb	0x20000098	20	.bss		20	
m_in_critical_region	0x2000013c	4	.bss		4	
m_tick_overflow_count	0x20000730	4	.bss		4	
random_nr.6122	0x2000071c	4	.bss		4	
rt_assert_hook	0x20000354	4	.bss		4	
rt_current_priority	0x20000470	1	.bss		1	
rt_current_thread	0x2000046c	4	.bss		4	
rt_exception_hook	0x2000072c	4	.bss		4	
rt_free_hook	0x200003dc	4	.bss		4	
rt_interrupt_enter_hook	0x20000340	4	.bss		4	
rt_interrupt_from_thread	0x20000720	4	.bss		4	
rt_interrupt_leave_hook	0x20000344	4	.bss		4	
rt_interrupt_nest	0x20000348	1	.bss		1	
rt_interrupt_to_thread	0x20000724	4	.bss		4	
rt_log_buf.6313	0x20000358	128	.bss		128	
rt_malloc_hook	0x200003d8	4	.bss		4	
rt_object_attach_hook	0x20000414	4	.bss		4	
rt_object_detach_hook	0x20000418	4	.bss		4	
rt_object_put_hook	0x20000424	4	.bss		4	
rt_object_take_hook	0x20000420	4	.bss		4	
rt_object_trytake_hook	0x2000041c	4	.bss		4	
rt_scheduler_hook	0x20000480	4	.bss		4	
rt_scheduler_lock_nest	0x20000428	2	.bss		2	
rt_soft_timer_list	0x20000498	8	.bss		8	
rt_thread_defunct	0x20000478	8	.bss		8	
rt_thread_idle_hook	0x2000033c	4	.bss		4	

rt_thread_init_hook	0x2000048c	4	.bss		4	
rt_thread_priority_table	0x2000042c	64	.bss		64	
rt_thread_ready_priority_group	0x20000474	4	.bss		4	
rt_thread_resume_hook	0x20000488	4	.bss		4	
rt_thread_stack	0x200001bc	384	.bss		384	
rt_thread_suspend_hook	0x20000484	4	.bss		4	
rt_thread_switch_interrupt_flag	0x20000728	4	.bss		4	
rt_tick	0x20000140	4	.bss		4	
rt_timer_list	0x20000490	8	.bss		8	
rt_timer_timeout_hook	0x20000718	4	.bss		4	
timer_thread	0x200004a0	120	.bss		120	
timer_thread_stack	0x20000518	512	.bss		512	
used_mem	0x2000040c	4	.bss		4	
working_cfg	0x20000784	4	.bss		4	
_console_device	0x20000350	4	.bss		4	
_serial0_0	0x20000734	80	.bss		80	
__bss_end__	0x20000788		.bss			
__bss_load_end__	0x20000788		.bss			
__bss_start__	0x20000098		.bss			
__rt_errno	0x2000034c	4	.bss		4	
.data	0x20000000	152			152	
rt_object_container	0x20000000	112	.data		112	
uart0	0x20000070	20	.data		20	
_uart_ops	0x20000084	20	.data		20	
__data_end__	0x20000098		.data			
__data_start__	0x20000000		.data			

.data_ram	0x20000000	152			152	
__data_ram_end__	0x20000098		.data_ram			
__data_ram_load_end__	0x20000098		.data_ram			
__data_ram_start__	0x20000000		.data_ram			
.init	0x000000dc	152		152		
BusFault_Handler	0x000000e6		.init			
CCM_AAR_IRQHandler	0x00000112		.init			
COMP_LPCOMP_IRQHandler	0x0000011a		.init			
DebugMon_Handler	0x000000ec		.init			
Dummy_Handler	0x000000f2		.init			
ECB_IRQHandler	0x00000110		.init			
FPU_IRQHandler	0x0000013c		.init			
I2S_IRQHandler	0x0000013a		.init			
MemoryManagement_Handler	0x000000e4		.init			
MWU_IRQHandler	0x00000130		.init			
NFCT_IRQHandler	0x000000fe		.init			
NMI_Handler	0x000000e0		.init			
PDM_IRQHandler	0x0000012e		.init			
PWM1_IRQHandler	0x00000132		.init			
PWM2_IRQHandler	0x00000134		.init			
QDEC_IRQHandler	0x00000118		.init			
RADIO_IRQHandler	0x000000f6		.init			
Reset_Handler	0x000000dc		.init			
RNG_IRQHandler	0x0000010e		.init			
RTC0_IRQHandler	0x0000010a		.init			
RTC2_IRQHandler	0x00000138		.init			

SPIM0_SPIS0_TWIM0_TWIS0_SPI0_TWI0_IRQHandler	0x000000fa		.init			
SPIM1_SPIS1_TWIM1_TWIS1_SPI1_TWI1_IRQHandler	0x000000fc		.init			
SPIM2_SPIS2_SPI2_IRQHandler	0x00000136		.init			
start	0x00000154		.init			
SVC_Handler	0x000000ea		.init			
SWI0_EGU0_IRQHandler	0x0000011c		.init			
SWI1_EGU1_IRQHandler	0x0000011e		.init			
SWI2_EGU2_IRQHandler	0x00000120		.init			
SWI3_EGU3_IRQHandler	0x00000122		.init			
SWI4_EGU4_IRQHandler	0x00000124		.init			
SWI5_EGU5_IRQHandler	0x00000126		.init			
SysTick_Handler	0x000000f0		.init			
TEMP_IRQHandler	0x0000010c		.init			
TIMER0_IRQHandler	0x00000104		.init			
TIMER1_IRQHandler	0x00000106		.init			
TIMER2_IRQHandler	0x00000108		.init			
TIMER3_IRQHandler	0x00000128		.init			
TIMER4_IRQHandler	0x0000012a		.init			
UsageFault_Handler	0x000000e8		.init			
WDT_IRQHandler	0x00000114		.init			
_start	0x00000140		.init			
__init_end__	0x00000174		.init			
__init_load_end__	0x00000174		.init			
__init_start__	0x000000dc		.init			
.rodata	0x00006500	3863				3863
large_digits.6203	0x000068d0	17	.rodata			17

m_board_led_list	0x00006520	4	.rodata			4
small_digits.6202	0x000068e4	17	.rodata			17
__FUNCTION__.5992	0x00006598	15	.rodata			15
__FUNCTION__.6015	0x000065a8	15	.rodata			15
__FUNCTION__.6020	0x000065b8	16	.rodata			16
__FUNCTION__.6034	0x000065c8	16	.rodata			16
__FUNCTION__.6066	0x0000660c	22	.rodata			22
__FUNCTION__.6079	0x00006bb4	11	.rodata			11
__FUNCTION__.6085	0x00006dfc	16	.rodata			16
__FUNCTION__.6088	0x000066f0	12	.rodata			12
__FUNCTION__.6088	0x00006bc0	20	.rodata			20
__FUNCTION__.6089	0x00006ce8	26	.rodata			26
__FUNCTION__.6094	0x00006d04	26	.rodata			26
__FUNCTION__.6094	0x00006ebc	14	.rodata			14
__FUNCTION__.6097	0x00006bd4	10	.rodata			10
__FUNCTION__.6097	0x00006e0c	15	.rodata			15
__FUNCTION__.6099	0x00006ecc	16	.rodata			16
__FUNCTION__.6104	0x00006e1c	18	.rodata			18
__FUNCTION__.6105	0x00006c6c	15	.rodata			15
__FUNCTION__.6110	0x00006c7c	17	.rodata			17
__FUNCTION__.6112	0x000066fc	12	.rodata			12
__FUNCTION__.6119	0x00006c90	19	.rodata			19
__FUNCTION__.6123	0x00006edc	15	.rodata			15
__FUNCTION__.6124	0x00006ca4	17	.rodata			17
__FUNCTION__.6128	0x00006be0	8	.rodata			8
__FUNCTION__.6128	0x00006cb8	26	.rodata			26

__FUNCTION__. 6136	0x00006e30	16	.rodata			16
__FUNCTION__. 6140	0x00006eec	14	.rodata			14
__FUNCTION__. 6146	0x00006efc	17	.rodata			17
__FUNCTION__. 6160	0x00006e40	18	.rodata			18
__FUNCTION__. 6165	0x00006e54	17	.rodata			17
__FUNCTION__. 6170	0x00006e68	18	.rodata			18
__FUNCTION__. 6284	0x0000729c	19	.rodata			19
__FUNCTION__. 6287	0x000071a8	16	.rodata			16
__FUNCTION__. 6290	0x0000733c	19	.rodata			19
__FUNCTION__. 6292	0x000072b0	19	.rodata			19
__FUNCTION__. 6297	0x000071d8	16	.rodata			16
__FUNCTION__. 6299	0x000072c4	19	.rodata			19
__FUNCTION__. 6300	0x00007350	19	.rodata			19
__FUNCTION__. 6308	0x00007188	15	.rodata			15
__FUNCTION__. 6315	0x00007364	18	.rodata			18
__FUNCTION__. 6321	0x000071c8	15	.rodata			15
__FUNCTION__. 6327	0x00007378	19	.rodata			19
__FUNCTION__. 6329	0x000070e8	29	.rodata			29
__FUNCTION__. 6338	0x00007108	29	.rodata			29
__FUNCTION__. 6344	0x00007128	29	.rodata			29
__FUNCTION__. 6351	0x00007198	15	.rodata			15
__FUNCTION__. 6370	0x00007148	15	.rodata			15
__FUNCTION__. 6377	0x00007158	15	.rodata			15
__FUNCTION__. 6387	0x00007168	16	.rodata			16
__FUNCTION__. 6400	0x00007178	15	.rodata			15
__FUNCTION__. 6408	0x000071b8	16	.rodata			16

__FUNCTION__. 6416	0x000071e8	18	.rodata			18
__FUNCTION__. 6431	0x000071fc	22	.rodata			22
__FUNCTION__. 6440	0x00007214	17	.rodata			17
__FUNCTION__. 9685	0x000073e8	10	.rodata			10
__FUNCTION__. 9696	0x000073f4	11	.rodata			11
__FUNCTION__. 9710	0x00007400	11	.rodata			11
__FUNCTION__. 9719	0x0000740c	11	.rodata			11
__lowest_bit_bitmap	0x000067d0	256	.rodata			256
__rodata_end__	0x00007417		.rodata			
__rodata_load_end__	0x00007417		.rodata			
__rodata_start__	0x00006500		.rodata			
.stack	0x20010000					
__stack_end__	0x20010000		.stack			
__stack_load_end__	0x20010000		.stack			
__stack_start__	0x20010000		.stack			
.text	0x00000174	25484		25484		
.....		
__text_end__	0x00006500		.text			
__text_load_end__	0x00006500		.text			
__text_start__	0x00000174		.text			
.vectors	0x00000000	220		220		
_vectors	0x00000000		.vectors			
_vectors_end	0x000000dc		.vectors			
__vectors_end__	0x000000dc		.vectors			
__vectors_load_end__	0x000000dc		.vectors			
__vectors_start__	0x00000000		.vectors			

10 参考文档

《Cortex-M3 权威指南(中文)》

《Cortex™-M4 Devices Generic User Guide》

《ARM® Cortex®-M4 Processor Technical Reference Manual》

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.cortexm.m4/index.html>

<http://tigcc.ticalc.org/doc/>

<http://re-eject.gbadev.org/files/GasARMRef.pdf>

《Embedded Studio for ARM Reference Manual》

https://www.segger.com/downloads/embedded-studio/EmbeddedStudio_Manual