**一、Bootloader 简介**

**1.1、bootloader基本概念**

在嵌入式操作系统中，BootLoader是在操作系统内核运行之前运行，用来初始化硬件设备，建立内存空间映射图，以便为操作系统的启动做准备。在嵌入式系统中，通常没有像BIOS那样的的固定程序，因此整个系统的加载启动任务就完全由BootLoader来完成。在嵌入式系统中，系统在上电或复位时通常从地址0x0000出开始执行，BootLoader即从该地址开始执行，即上电或复位后执行的第一段代码。

BootLoader是启动程序，主要作用就是做某些初始化动作和升级应用程序。根据升级方案的不同，可以分为带临时存储区域的和不带临时存储区域的。

带临时存储区域的升级流程里面，编程芯片将接收的到的升级文件以数据的形式放进临时区域，升级文件校验通过后，再将升级文件复制到应用程序区域。

不带临时存储区域的升级流程里面，编程芯片将接收的升级文件直接放到应用程序区域，升级文件校验。

**1.2、方案设计模型**

该设计模块采用 **boot + 单bank模式**，boot加载flash驱动，负责一切升级活动。 （bank也可称partitions）

例如还有一些别的设计：

boot + 单bank （options ：备份区）

boot(UBL) + 单bank（options ：备份区）,UBL也分为PBL（**Primary Bootloader**）和SBL（**Secondary Bootloader**） ,链接：<https://blog.csdn.net/u011029104/article/details/136539766>

英飞凌TC3XX芯片之A/B Swap机制，链接：<https://bbs.21ic.com/icview-3384826-1-1.html>

**二、分区介绍**

APM32E103 芯片，**flash 最小单位sector（页）：2KB**



分区根据芯片pflash 地址**用户自行定义**，分布如下：

起始地址占64KB 运行bootloader程序

占2个sector（扇区）存放bootflag(升级标志)

占2个sector（扇区）存放Integrity app flag(app完整性标志)

规划设计给APP区占用256KB内存大小

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Address | 0x8000000(64Kb) | ~~0x8010000(2Kb)~~ | 0x8010800(2Kb) | 0x8011000(256Kb) |
| 分区 | bootloader | ~~bootflag~~ | Integrity app flag | APP |

拓展：在设计bootloader的时候，我们可以对flash进行ecc检查。链接：<https://blog.csdn.net/beyehz/article/details/40356781>

目前： bootflag 变更存放在不初始化sram的区域当作

**三、Boot flag分区设计目的**

boot flag的设计目的，在bootloader程序运行的时候，会判断该标志位是否有升级请求，决定是否停留在bootloader还是跳转到APP程序当中，关于boot flag存放的地方有几点考虑：

该芯片支持非易失性RAM，也就是有特殊ram区可以存放flag，并掉电不会丢失。类似NXP、英飞凌高级芯片会支持

该芯片支持Data flash，也是特定芯片支持

外部挂载Eeprom芯片

外部挂载SDRAM芯片

最终结论：当把升级当作平台开发，最好是把boot flag放在主控芯片的pflash中。

**四、启动过程**

**4.1、contex-M 启动地址详解**

问题：为什么contex-M的芯片是从0x08000000启动？

**4.1.1、背景**

ARM官方规定，M3，M4内核芯片上电复位后，要固定从0x0000 0000地址读取中断向量表，获取复位中断服务程序的入口地址后，进入复位中断服务程序，其中0x0000 0000是栈顶地址，0x0000 0004存的是复位中断服务程序地址。

**4.1.2、内存重映射功能**

既然ARM规定了M3，M4内核要从地址0x0000 0000读取中断向量表，而STM32设置Flash地址到0x0800 0000怎么办？

STM32支持了个内存重映射功能，将地址0x0800 0000开始的内容重映射到首地址0x0000 0000中，这样就解决了从0x0000 0000读取中断向量表的问题。

**4.1.3、中断向量表是否能改变**

; Vector Table Mapped to Address 0 at Reset

AREA RESET, DATA, READONLY

EXPORT \_\_Vectors

EXPORT \_\_Vectors\_End

EXPORT \_\_Vectors\_Size

这个名字很重要，MDK对应的xxx.sct分散加载里面通过下面这句将这个RESET段放在了0x0800 0000优先存储。

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; \*\*\* Scatter-Loading Description File generated by uVision \*\*\*

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//定义一个加载域，域地址0x08000000，域大小为0x00080000

//load region size\_region 所有代码需要下载到0x08000000，开始的区域中，且这个区域大小只有0x00080000

LR\_IROM1 0x08000000 0x00080000 { ; load region size\_region 注释：加载域，基地址 空间大小

//load address = = execution address 第一个运行时域必须和加载域起始地址相同，大小也想通

ER\_IROM1 0x08000000 0x00080000 { ; load address = execution address 注释：加载地址 = 执行地址

\*.o (RESET, +First) //启动代码的首次执行地址，RO执行域名称为ER\_IORM1

//将 RESET 段最先加载到本域的起始地址外

//首次执行的地址为RESET标号所表示的地址,RESET 存储的是向量表

//对应启动文件中的AREA RESET, CODE, READONLY

\*(InRoot$$Sections)

.ANY (+RO) //加载所有匹配目标文件的只读属性数据，包含：Code、 RW-Code、 RO-Data。

.ANY (+XO)

}

//再定义一个运行时域，域基址0x20000000

//RW data 执行域是以0x20000000 开始的长度为0x00020000 一段区域

RW\_IRAM1 0x20000000 0x00020000 { ; RW data

.ANY (+RW +ZI)

}

}

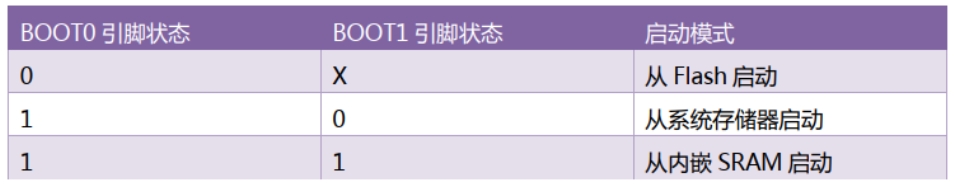
保证中断向量表存到0x0800 0000，这个涉及到分散加载的一个小知识，以MDK为例，如果大家看xxx.S启动文件，里面通过AREA定义了一个名叫RESET的段，这段存的就是中断向量表。

这样我们就解决了0x0800 0000首地址存储中断向量表，一旦程序开始运行后，我们就可以随意设置中断向量表的位置了。比如想将中断向量表存到内部SRAM，我们就可以操作寄存器SCB->VTOR 重新安排，然后将0x0800 0000的内容复制到设置的地址内即可。

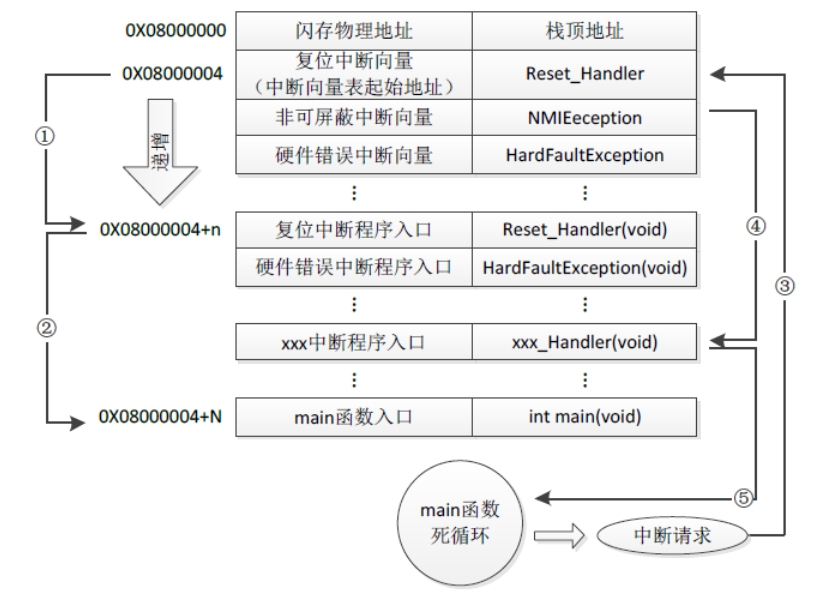
**4.1.4、contex-M限制**

既然设置到0x0800 0000这么麻烦，为什么不直接使用0x0000 0000？

这是因为STM32不仅可以从内部Flash启动，还可以从系统存储器（可以实现串口ISP，USB DFU等程序下载方式，这个程序是ST固化好的程序代码）和从内部SRAM启动， 我们将内部Flash安排到0x0000 0000显然是不行的。这样会导致系统存储器或者内部SRAM无法重映射到0x0000 0000了。



**4.2、APM32正常运行流程**



从图中可以看到，在0x08000000处存放着程序的中断向量表，它由栈顶地址和各中断的入口地址组成。在上电复位后，芯片会由硬件控制回到0x08000000处，先将0x08000000地址开始的4字节栈顶指针送给sp寄存器，然后将0x08000004地址开始存放的复位中断的入口地址给程序计数器pc，从此开始程序跳转到复位中断服务函数。

跳转到复位中断后，它做的只有两件事，一是系统时钟初始化，二是跳转到我们编写的main函数：

; Reset handler

Reset\_Handler PROC

EXPORT Reset\_Handler [WEAK]

IMPORT \_\_main

IMPORT SystemInit

LDR R0, =SystemInit

BLX R0

LDR R0, =\_\_main

BX R0

ENDP

当我们进入到main函数时，程序会按照我们编写的逻辑运行。当有中断请求时，芯片会从中断向量表中获取相应的中断服务函数入口地址，跳转到相应的中断程序。并且中断向量表的位置是可以重定义的，它由SCB->VTOR寄存器控制。Bootloader 和 APP 的程序设计中，APP需要偏移中断向量：

SCB->VTOR = FMC\_BASE | VECT\_TAB\_OFFSET;

**4.3、跳转APP前注意事项**

bootloader在跳转APP的程序代码需要注意以下几点：

检查栈顶地址是否合法，也就是APP存放的起始地址是否合法

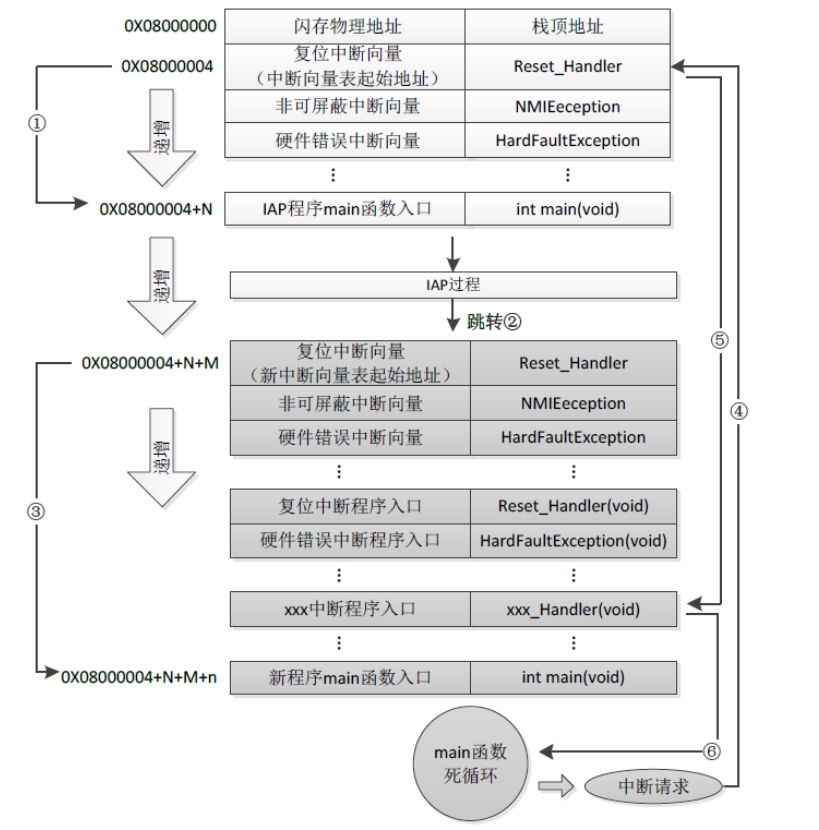
if(((\*(\_\_IO uint32\_t\*)APP\_FLASH\_ADDR) & 0x2FFE0000) == 0X20000000)

屏蔽总中断，防止程序跳转过程中，中断干扰出现异常

获取APP程序起始地址，代码区第二个字（起始地址 +4 Flash 位置存储的数据）

初始化堆栈指针（用于代码区的第一个字用于存放栈顶地址）

利用APP起始地址，转化为函数指针类型，执行跳转



Contex-M启动文件详解链接：<https://blog.csdn.net/Yuanghxb/article/details/127274317>

**4.4、APU项目启动流程**

[bootloader流程图](note://WEB0cc2fff183cd25a4a3677b88e79fbe71)

**五、升级流程服务**

**5.1、默认会话**

应用运行时，会话等级处于默认会话（CMD：A1）

会话等级流程图：<https://note.youdao.com/s/CJZ9m1cM>

**5.2、编程会话**

收到该指令，下位机将boot flag置为0xAA，跳转成功后应答（CMD：0xA2）

会话等级流程图：<https://note.youdao.com/s/CJZ9m1cM>

**5.3、扩展会话**

进入扩展会话等级，某些服务需要在扩展会话下支持，进入准备升级阶段（CMD:0xA3）

会话等级流程图：<https://note.youdao.com/s/CJZ9m1cM>

**5.4、停止设备通信服务**

停止对外的通信，包括485、CAN、串口等输出（CMD:0xA4）

基于扩展会话等级下

**5.5、请求擦除**

目前设计把hex文件进行识别，会下发地址进行擦除，擦除整片APP区域，包括Integrity app flag（CMD：0xA5）

**5.6、请求下载**

将偏移量置清零，准备开始写入（CMD:0xA6）

**5.7、数据传输**

上位机会识别hex文件划分整片APP的分包，从第一包下发到最后一包，传输有效数据大小最大值为512个字节（CMD:0xA7）

**5.8、传输退出**

计算整个固件CRC16值（CMD:0xA8）

**5.9、检查应用完整性**

上位机下发CRC16值与下位机计算的CRC16值比较，成功则将Integrity app flag置为0xAA（CMD:0xA9）

**5.10、软复位指令**

应答后调用软件复位函数（CMD:0xAA）

**5.11、会话保持服务**

保持当前设备的会话等级（CMD:0x3E）

**六、数据协议规定**

**6.1、帧格式**

**6.1.1、串口数据帧格式**

上位机：0xEE（1byte）+ ID （1byte）+ DLC(2byte)+ DATA(n)+CRC16（2byte）

下位机：0xEE（1byte）+ ID （1byte）+ DLC(2byte)+ DATA(n)+CRC16（2byte）

**6.2、协议说明**

**6.2.1、串口升级协议规定**

**采用一发一收模式**

**上位机：**

0xEE：升级协议头

ID：数据id

DLC：有效数据长度

DATA：有效数据

CRC16：采用MODBUS CRC16计算

**下位机：**

0xEE：升级协议头

ID：数据id

DATA：有效数据

CRC16：采用MODBUS CRC16计算

正响应：

上位机：EE 11 00 01 01 + CRC16

下位机：EE 22 00 01 41 + CRC16

负响应:(0x01 ,默认会话只需要一个字节长度，多发55字节长度不对。)

上位机：EE 11 00 02 01 55+ CRC16

下位机：EE 22 00 03 7F 01 + N\_ACK + CRC16

**6.2.2、汽车的CAN-TP协议（扩展内容）**

CAN-TP帧类型分为 单帧 和 多帧。

单帧（SF）：single Frame

多帧可分为：

首帧（FF）：First Frame

流控帧（FC）: Flow Control

连续帧（CF）: Consecutive Frame

学习链接：<https://blog.csdn.net/kian9one/article/details/140085589>

**七、上位机制作**

**7.1、开发环境**

采用QT去制作我们的上位机，因为Qt Creator是一个用于Qt开发的轻量级跨平台集成开发环境。

提供首个专为支持跨平台开发而设计的集成开发环境 (IDE)，并确保首次接触Qt框架的开发人员能迅速上手和操作。

即使不开发Qt应用程序，Qt Creator也是一个简单易用且功能强大的IDE。

**7.2、设计框架**

**7.2.1、上位机功能设计框架**

<https://note.youdao.com/s/A5Gc50Mh>

**7.2.2、ui设计框架**

<https://note.youdao.com/s/Q9QLUkVO>

**7.3、驱动开发层（目前加载串口驱动）**

使用QT提供的uart驱动，也可以加载其他第三方驱动库。

**7.4、数据解析层**

进行模块化设计，与驱动进行解耦，并映射到app服务上。

**7.5、相关笔记和文档**

<https://note.youdao.com/s/I3vVFJBu>

<https://note.youdao.com/s/FuRhEAe2>