

# STM32 IAP升级程序设计详解-IAR环境

原创

置顶

GWen9

于 2020-11-07 18:36:01 发布

1023

收藏

7

版权


分类专栏：

单片机

 文章标签：

单片机

嵌入式

 单片机 专栏收录该内容

1 订阅   3 篇文章   

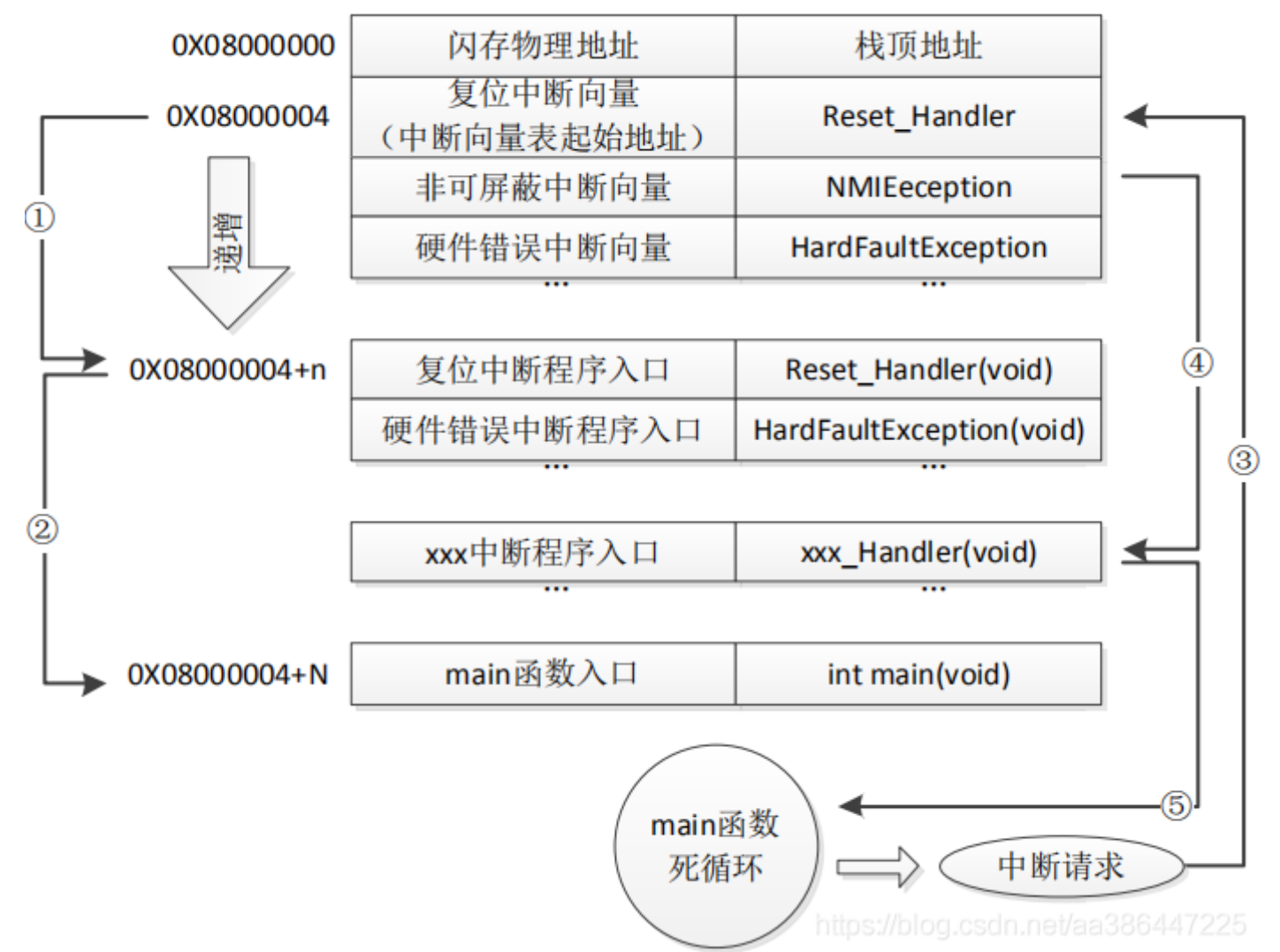
订阅专栏

本文可与另外一篇文章做对比参考：STM8 IAP升级程序设计详解 - IAR环境

## 一 STM32 IAP 原理分析

STM32 IAP的实现原理与STM8类似，只是STM32可以设置中断向量表的偏移，而STM8不能设置偏移只能通过中断向量表的 **重定向** 来实现APP程序中中断的使用。但是同样还是需要设计两个程序，在Bootloader程序通过某种通信方式，如 USB、USART接收APP程序数据，并写入Flash中，然后跳转到APP程序的首地址，开始运行第二个程序。

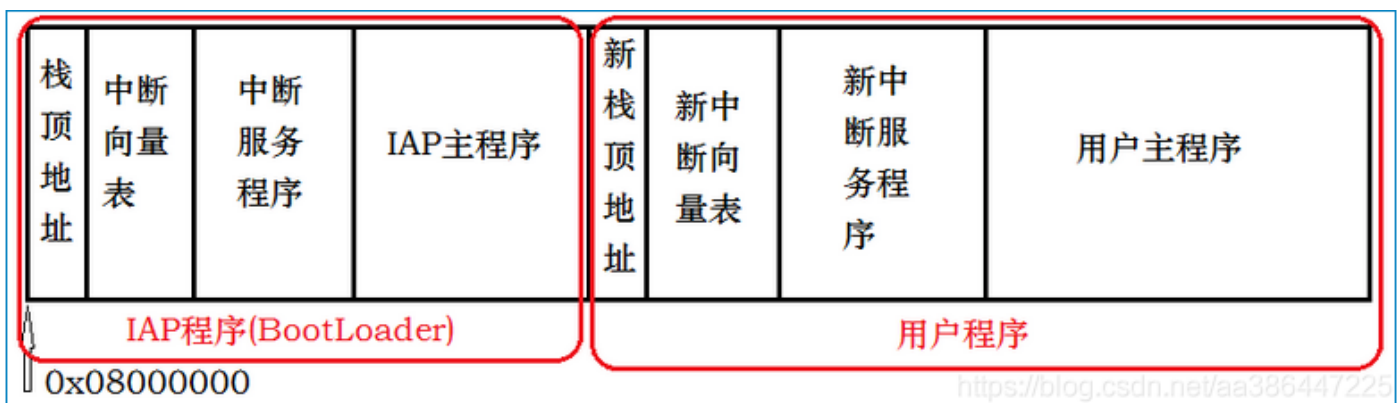
首先我们来分析只有一个APP程序的时候STM32的运行流程图：



以STM32L4系列单片机为例，在官方手册中可以查到Flash的起始地址为0x0800 0000，通过上图可以看到在Flash的起始地址存放的是栈顶指针，紧接着就是中断向量表，中断向量表中的第一个中断向量就是复位中断向量。此时STM32的运行流程可以大致分为以下几步：

1. 芯片上电复位后，PC指针硬件强制指向复位中断向量，跳转执行复位中断函数；
2. 执行完复位中断函数后再跳转到用户main函数；
3. 在main函数中如果发生了其他中断，硬件会强制PC指针指向对应的中断向量；
4. 然后再跳转到相应中断函数中执行；
5. 执行完成后继续返回main主循环，并重复步骤3，4，5。

当加入Bootloader程序之后Flash中的内容变成了这样:



此时程序的运行流程发生了一些变化:

1. 在BootLoader中程序的运行流程与Flash中只有一个APP程序时一致。
2. BootLoader接收完APP程序后写入Flash，并跳转执行，进入APP的main函数。
3. 但是在APP的main函数中发生中断请求时，硬件仍然会强制PC指针指向BootLoader程序的中断向量表处（这是硬件决定的），然后跳转执行BootLoader程序中的中断服务函数。

此时会出现一个问题：APP程序中的中断服务程序无法执行。

note: 在STM8中可以通过重定向中断向量表，通过修改BootLoader的中断向量表中的内容，当PC指针指向第一个中断向量表（BootLoader的中断向量表）时，我们把中断向量表中原本应该跳转到中断服务函数的指令，修改为跳转到新的APP的中断向量表的指令，然后再跳转到APP的中断服务函数。这样就能在APP程序中使用中断了，但是这样存在的问题是，无法在BootLoader中使用中断了。

显然，STM32仍然可以使用这种办法，只是对于STM32来说有更好的解决方案，STM32提供了中断向量表的偏移机制。只需要在APP程序的启动文件中设置好中断向量向量表的偏移地址，当在APP程序中产生中断请求时，PC指针会强制指向：**原中断向量地址+偏移地址**，这样就能在APP程序中

使用中断了，而只要我们不在BootLoader中设置偏移，当BootLoader中产生中断请求时PC指针还是会指向它自己的中断向量表地址。这样无论在BootLoader中还是在APP中我们都能够使用中断了。

note:另外可以设置中断向量表的偏移还有另外一个好处，就是可以使用多个APP，因为只需要在APP程序中设置不同的中断向量表的偏移就行了，而如果用中断向量表的重定向就只能有一个APP应用程序了。

## 二 STM32 IAP升级设计流程

有了以上对升级原理分析我们就可以来实现自己的IAP升级程序了。前面也已经说过了这里需要我们设计两个程序：

- 1. BootLoader 程序
- 2. 用户APP

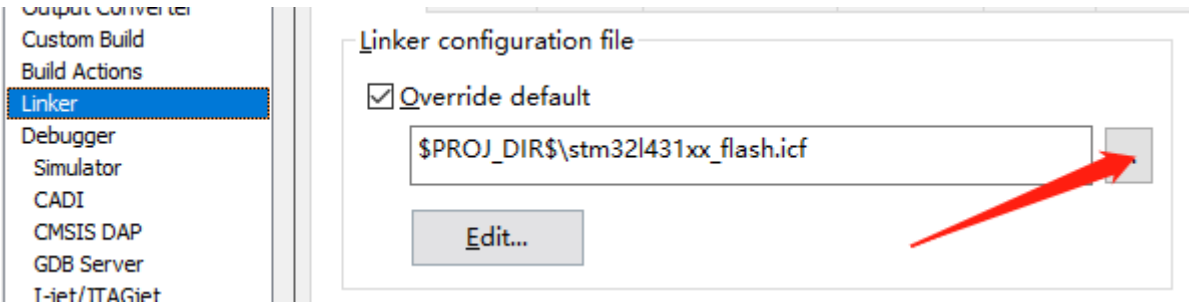
STM32 IAP升级设计流程如下：

- 1. 修改icf链接文件，并且设置新的链接文件为当前工程的链接文件；
- 2. 设置APP程序中中断向量表的偏移地址；
- 3. 自己编写或者下载一个bin文件的分包发送工具；
- 4. 接收下载的内容写入到 Flash 中；
- 5. 跳转执行app程序.

### 1.修改链接文件

由于BootLoader和APP程序需要烧录到不同的地址所以这里我们需要通过链接文件来对其的烧写地址进行修改。

a. 首先打开工程对应的\*.icf链接文件：



b. 这里我们以Flash中的程序为例，所以选择第一个icf文件，右键编辑：

Debug	2020/11/7 15:25	文件夹	
settings	2020/11/7 15:25	文件夹	
stm32l431xx_flash	2020/10/15 16:06	ICF 文件	2 KB
stm32l431xx_sram	2020/10/15 16:06	ICF 文件	2 KB

c.在Memory Regions栏可以看到该款芯片的ROM地址范围为0x0800 0000 - 0x0803 FFFF (256K)，即Flash大小为256K。

```
/*-Memory Regions-*/
define symbol _ICFEDIT_region_ROM_start_ = 0x08000000;
define symbol _ICFEDIT_region_ROM_end_   = 0x0803FFFF;
define symbol _ICFEDIT_region_RAM_start_  = 0x20000000;
define symbol _ICFEDIT_region_RAM_end_    = 0x2000FFFF;
```

d.对于BootLoader程序可以根据自己的需求自己设置大小，例如这里我们设置为32K。

```
/*-Memory Regions-*/
define symbol _ICFEDIT_region_ROM_start_ = 0x08000000;
define symbol _ICFEDIT_region_ROM_end_   = 0x08008000;
define symbol _ICFEDIT_region_RAM_start_  = 0x20000000;
define symbol _ICFEDIT_region_RAM_end_    = 0x2000FFFF;
```

e.APP程序直接使用剩下的空间即可（共224K）。

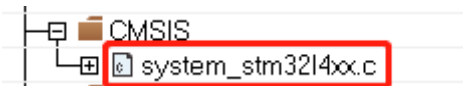
```
/*-Memory Regions-*/
define symbol _ICFEDIT_region_ROM_start_ = 0x08008000;
define symbol _ICFEDIT_region_ROM_end_   = 0x0803FFFF;
define symbol _ICFEDIT_region_RAM_start_  = 0x20000000;
define symbol _ICFEDIT_region_RAM_end_    = 0x2000FFFF;
```

e.最后APP的链接文件中还需要重新设置中断向量表的起始地址。

```
/*-Specials-*/
define symbol _ICFEDIT_intvec_start_ = 0x08008000;
/*-Memory Regions-*/
define symbol _ICFEDIT_region_ROM_start_ = 0x08008000;
define symbol _ICFEDIT_region_ROM_end_   = 0x0803FFFF;
define symbol _ICFEDIT_region_RAM_start_  = 0x20000000;
define symbol _ICFEDIT_region_RAM_end_    = 0x2000FFFF;
```

## 2. 设置中断向量表的偏移地址（APP）

首先我们找到STM32的系统启动文件，这里以STM32L4为例：



在系统启动文件中找到SystemInit函数，并在最后两行可以看到如下代码：

```
/* Configure the Vector Table location add offset address -----*/
#ifdef VECT_TAB_SRAM
  SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal SRAM */
#else
  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal FLASH */
#endif
```

这里可以设置SRAM和Flash中断向量表的偏移，这里以Flash为例所以需要检查VECT\_TAB\_SRAM是否被定义，如果定义了则需要取消，然后再设置宏VECT\_TAB\_OFFSET的值。在该文件找到VECT\_TAB\_OFFSET宏，可以发现的初始值为0

```
/*----- Miscellaneous Configuration -----*/
/*!< Uncomment the following line if you need to relocate your vector Table in
Internal SRAM. */
/* #define VECT_TAB_SRAM */
#define VECT_TAB_OFFSET 0x00 /*!< Vector Table base offset field.
This value must be a multiple of 0x200. */
```

由于我们之前设置了BootLoader的大小为32K，所以这里需要把VECT\_TAB\_OFFSET宏的值修改为0x8000

```

/***** Miscellaneous Configuration *****/
/*< Uncomment the following line if you need to relocate your vector Table in
   Internal SRAM. */
/* #define VECT_TAB_SRAM */
#define VECT_TAB_OFFSET 0x8000 /*< Vector Table base offset field.
   This value must be a multiple of 0x200. */
/*****

```

note: 这里只需要设置APP程序的中断向量表的偏移, BootLoader不需要进行设置

### 3. 编写Bootloader升级程序

## 1. 升级条件判断

通过指定判断条件判断是否需要升级（这里笔者通过的是判断是否有USB插入）。如果需要升级则进入接收程序，不需要则直接跳转到APP程序。

## 2. 下载升级文件

传输bin文件可以通过很多方式，这里笔者自己用QT写了一个串口bin文件的发送工具，将bin文件分包发送给单片机，单片机接收到数据后直接写入Flash即可。

这里需要注意的是，普通的串口调试助手虽然能够发送bin文件，但是无法分包，如果APP程序不是特别大的话（RAM空间能将程序全部储存下来），可以直接用串口助手。接收完成后再一次性全部写入Flash中。如果RAM空间不足以存下所有的APP程序就需要自己写一个分包发送工具，或者使用可以设置发送延时的串口工具或超级终端都行。

### 3. 写入flash

设置好写入的地址，可以一次性写入，也可以分包写入。

#### 4. 跳转到app地址执行

注意：如果在bootloader中启用了某些外设的中断，需要在跳转之前关闭否则无法在app中使用中断。  
参考代码如下：

```
typedef void (*PFN_Reset)(void);           //定义函数指针类型
void vIapLoadApp(uint32_t xAppAddr)
{
    uint32_t nMSP, xJumpAddr;              /*栈顶指针*/
    PFN_Reset vResetHandler = NULL;        /*复位中断函数指针*/

    /*app起始位置4个字节储存的是栈顶指针*/
    nMSP = *((__IO uint32_t*)(xAppAddr));
    /*取出复位中断函数的地址*/
    xJumpAddr = *((__IO uint32_t*)(xAppAddr+4));
    /*复位中断函数指针赋值*/
    vResetHandler = (PFN_Reset)(xJumpAddr);
    /*检测栈顶指针是否合法*/
    if((nMSP&0x2FFE0000) == 0x20000000)
    {
```

[illegible]

*/\*初始化APP堆栈指针(用户代码区的第一个字用于存放栈顶地址)\*/*

`__set_MSP(nMSP);`

*/\*跳转到APP (从APP复位中断向量处取指令执行)\*/*

`vResetHandler();`

`}`

`}`