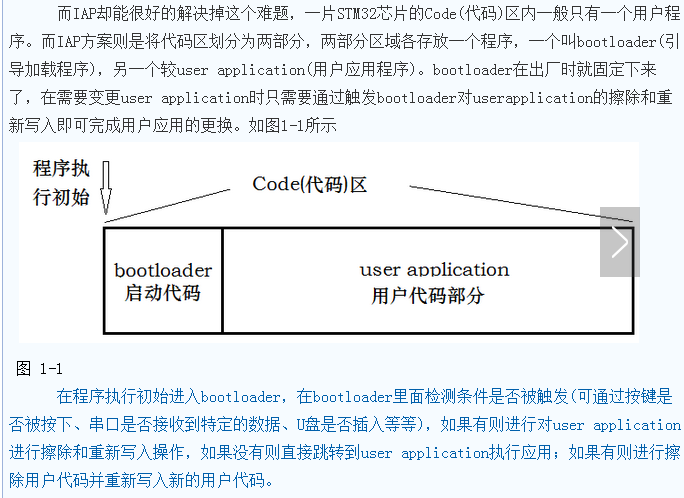
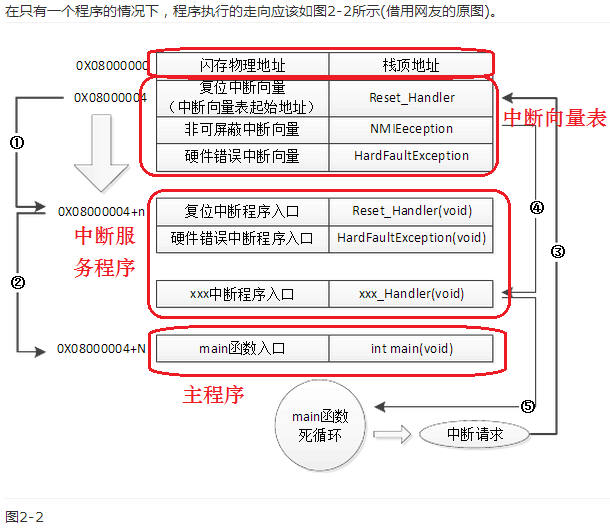
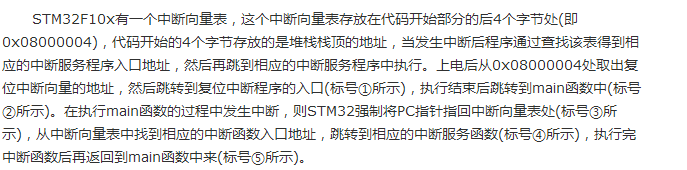
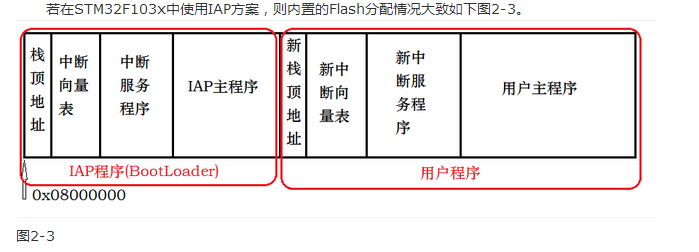
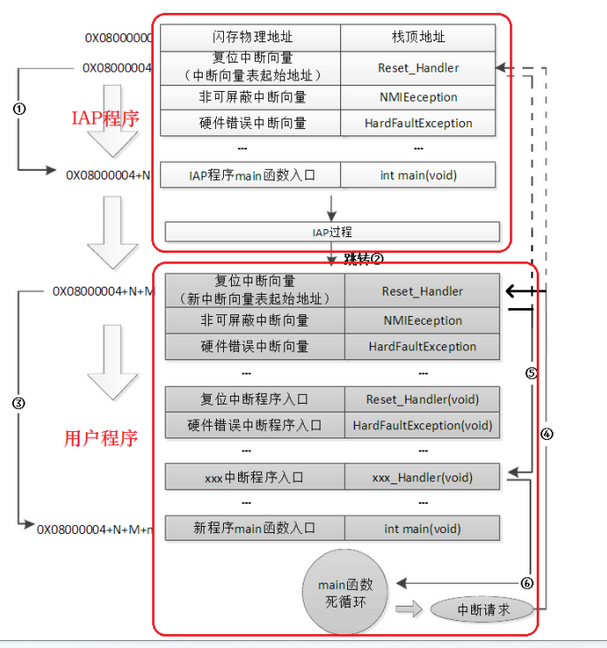
# STM32+IAP方案 实现网络升级应用固件

2015年12月09日 10:33:31 [目标向前](https://me.csdn.net/guozhongwei1) 阅读数：13455

关注了这个概念有些日子了，这段时间总算有机会实战==网络升级应用固件，这里记录下遇到的问题，及解决方案。   
原理与网上流传的串口作为传输手段 一致；不同之处，无非我这里使用了网络设备传输。==(lwip)TFTP客户端的应用.   
参考：   
[IAR环境下STM32+IAP方案的实现](http://www.51hei.com/stm32/4315.html)   
[STM32浅谈之IAP.pdf](http://pan.baidu.com/s/1czCWxS)   
[基于IAP和Keil MDK的远程升级设计](http://blog.csdn.net/zhzht19861011/article/details/6008702)   
[keil MDK中如何生成\*.bin格式的文件](http://blog.csdn.net/zhzht19861011/article/details/5927924)

概况：

* 什么是IAP，为什么要IAP
* 可实现的原理
* 实现过程
* 细节及实现   
  以上基本都可以从【IAR环境下STM32+IAP方案的实现】中找到答案。这里只是贴图，醒目：   
  IAP框架布局：   
     
  STM32F103ZET6的启动方式有三种：内置FLASH启动、内置SRAM启动、系统存储器ROM启动，通过BOOT0和BOOT1引脚的设置可以选择从哪中方式启动，这里选择内置的FLASH启动。其FLASH的地址为0x08000000—0x0807ffff，共512KB，这些都能从芯片数据手册中直接得到。而这里首要的一个问题是中断的问题。正常情况下发生中断的过程为：发生中断(中断请求)，到中断向量表查找中断函数入口地址，跳转到中断函数，执行中断函数，中断返回。也就是说在STM32的内置的Flash中有一个中断向量表来存放各个中断服务函数的入口地址，内置Flash的分配情况大致如下图2-1。   
     
     
     
     
  在内置的Flash里面添加一个BootLoader程序，BootLoader程序和user application各有一个中断向量表，假设BootLoader程序占用的空间为N+M字节，则程序的走向应该如图2-2所示(借用网友的原图并做改动，其中虚线部分为原图步骤④⑤的走向，本人改为指向灰色部分)。   
     
  上电初始程序依然从0x08000004处取出复位中断向量地址，执行复位中断函数后跳转到IAP的main(标号①所示)，在IAP的main函数执行完成后强制跳转到0x08000004+N+M处(标号②所示)，最后跳转到新的main函数中来(标号③所示)，当发生中断请求后，程序跳转到新的中断向量表中取出新的中断函数入口地址，再跳转到新的中断服务函数中执行(标号④⑤所示)，执行完中断函数后再返回到main函数中来(标号⑥所示)。   
  对于步骤④⑤，网友认为是：“在main执行的过程中，如果CPU得到一个中断请求，PC指针仍强制跳转到地址0x08000004中断向量表处，而不是新的中断向量表，如图标号④所示，程序再根据我们设置的中断向量表偏移量，跳转到对应中断源新的中断服务程序中，如图标号⑤所示”。我对此的理解是：“当发生中断后，程序从0x08000004(旧)处的中断向量表中得到相应的中断服务函数入口地址，继而跳转到相应的中断服务程序”。但是旧的中断向量列表里边存放的是IAP程序中断函数的入口地址，它是如何得到user程序中断函数的入口地址呢？所以我觉得此种说法是错误的。“当发生中断时PC指针强制会跳转到0x08000004处”这种说法并没有错，只是忽略了后续的一些知识要点而导致这个说法出现矛盾。   
  对于步骤④⑤我认为的是，在main函数的执行过程中，如果CPU得到一个中断请求，PC指针本来应该跳转到0x08000004处的中断向量表，由于我们设置了中断向量表偏移量为N+M，因此PC指针被强制跳转到0x08000004+N+M处的中断向量表中得到相应的中断函数地址(待求证)，再跳转到相应新的中断服务函数，执行结束后返回到main函数中来。

IAP流程描述：

**1、IAP的bootloader引导程序**

IAP在应用中编程，可以拓展成远程网络更新应用固件。   
片内的flash，至少划分成2个分区，对应至少两个完整的程序；   
低地址分区端推荐放入IAP程序==bootloader引导程序（这里边的手段可以是串口、网络等不同的方式），高地址分区端推荐烧写app固件。   
关键点提及：   
IAP程序中，当满足跳转条件（被触发）时，执行跳转代码到app应用固件程序，跳转代码流程：   
至少需要设定跳转目的地的app应用固件 栈顶指针，：

/\* Initialize user application's Stack Pointer \*/

**\_\_set\_MSP(\*(\_\_**IO uint32*\_t\*) USER\_*FLASH*\_FIRST\_*PAGE\_ADDRESS);

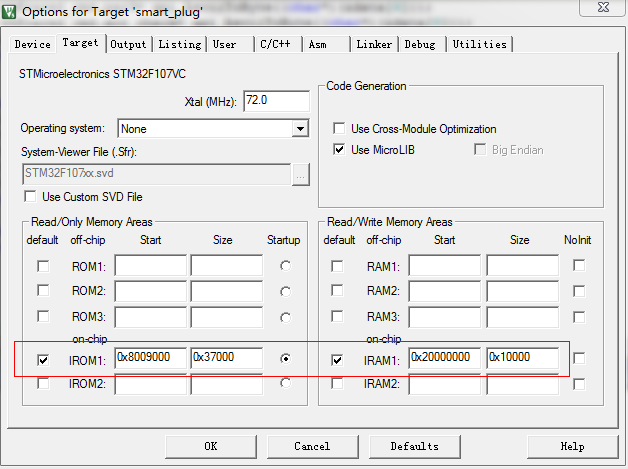
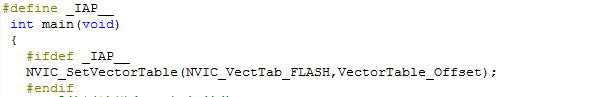
* 1
* 2

其中，app应用固件的分区地址：

#define USER\_FLASH\_FIRST\_PAGE\_ADDRESS 0x08009000

* 1

**2、app应用固件**

需要两处的更改，不然错误未知   
IROM设置如图：   
   
中断向量表偏移:   


NVIC\_SetVectorTable(NVIC\_VectTab\_FLASH,VectorTable\_Offset);

* 1

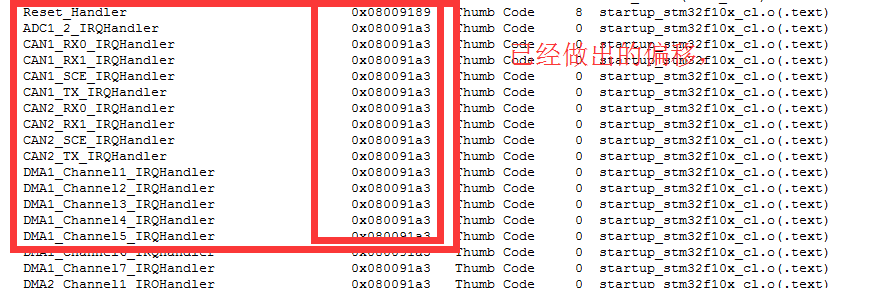
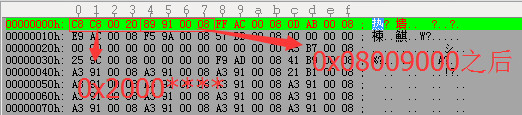
其中：

#define NVIC\_VectTab\_FLASH ((uint32\_t)0x08000000)

#define VectorTable\_Offset 0x9000

* 1
* 2

查错：

如果做了上边的工作，IAP依然无法顺利执行跳转至app应用程序，可以查看.map和.bin文件，确定是否如实的改变的中断向量表的偏移和栈顶指针，如图：   
   
.bin文件：   
   
可以看到，主栈顶MSP地址=0x2000C8C8、reset\_handler地址=0x08009189   
如此，才能生效，否则，可能原因：   
修改后的向量表偏移，在之后的程序中，又再次被还原，通过如下的函数：

void NVIC\_SetVectorTable(uint32\_t NVIC\_VectTab, uint32\_t Offset);

void SystemInit (void);

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

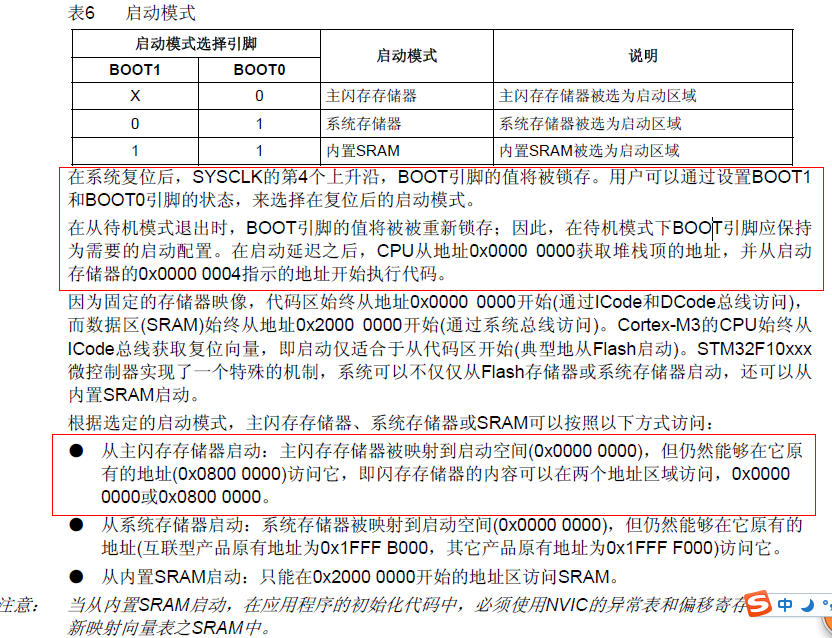
* 1
* 2
* 3

附：

1、如需要.hex文件转.bin，参见上边的文章   
当然，就算使用.hex文件，同样可以升级，只是需要修改IAP中判定已经升级的文件是否有效，文件条件部分的代码，

if((*(\*(\_\_IO uint32\_t\*)*USER\_FLASH\_FIRST\_PAGE\_ADDRESS) & 0x2FFE0000 ) == 0x20000000)

* 1

2、地址偏移后的app应用程序，是否能够独立的运行？   
不能，理由：   
   
可知，开机上电并不能够找到我们指定的偏移后的向量表。

# IAR环境下STM32+IAP方案的实现

作者:佚名   来源:本站原创   点击数: 19209   更新时间：2014年09月15日   【字体：[大](javascript:ContentSize(16)) [中](javascript:ContentSize(14)) [小](javascript:ContentSize(13))】

|  |  |
| --- | --- |
|  |  |

**--基于STM32F103ZET6的UART通讯实现**

**一、什么是IAP，为什么要IAP**

      IAP即为In Application Programming(在应用中编程)，一般情况下，以STM32F10x系列芯片为主控制器的设备在出厂时就已经使用J-Link仿真器将应用代码烧录了，如果在设备使用过程中需要进行应用代码的更换、升级等操作的话，则可能需要将设备返回原厂并拆解出来再使用J-Link重新烧录代码，这就增加了很多不必要的麻烦。站在用户的角度来说，就是能让用户自己来更换设备里边的代码程序而厂家这边只需要提供给用户一个代码文件即可。

      而IAP却能很好的解决掉这个难题，一片STM32芯片的Code(代码)区内一般只有一个用户程序。而IAP方案则是将代码区划分为两部分，两部分区域各存放一个程序，一个叫bootloader(引导加载程序)，另一个较user application(用户应用程序)。bootloader在出厂时就固定下来了，在需要变更user application时只需要通过触发bootloader对userapplication的擦除和重新写入即可完成用户应用的更换。如图1-1所示

|  |
| --- |
| <http://www.51hei.com/UpFiles/up/0/49151122166046.jpg> |
| 图 1-1 |

      在程序执行初始进入bootloader，在bootloader里面检测条件是否被触发(可通过按键是否被按下、串口是否接收到特定的数据、U盘是否插入等等)，如果有则进行对user application进行擦除和重新写入操作，如果没有则直接跳转到user application执行应用；如果有则进行擦除用户代码并重新写入新的用户代码。

**二、STM32F103ZET6硬件条件**

      STM32F103ZET6的启动方式有三种：内置FLASH启动、内置SRAM启动、系统存储器ROM启动，通过BOOT0和BOOT1引脚的设置可以选择从哪中方式启动，这里选择内置的FLASH启动。其FLASH的地址为0x08000000—0x0807ffff，共512KB，这些都能从芯片数据手册中直接得到。而这里首要的一个问题是中断的问题。正常情况下发生中断的过程为：发生中断(中断请求)à到中断向量表查找中断函数入口地址à跳转到中断函数à执行中断函数à中断返回。也就是说在STM32的内置的Flash中有一个中断向量表来存放各个中断服务函数的入口地址，内置Flash的分配情况大致如下图2-1。

|  |
| --- |
| <http://www.51hei.com/UpFiles/up/0/491511221656373.jpg> |
| 图2-1 |

在只有一个程序的情况下，程序执行的走向应该如图2-2所示(借用网友的原图)。

|  |
| --- |
| <http://www.51hei.com/UpFiles/up/0/491511221618834.jpg> |
| 图2-2 |

      STM32F10x有一个中断向量表，这个中断向量表存放在代码开始部分的后4个字节处(即0x08000004)，代码开始的4个字节存放的是堆栈栈顶的地址，当发生中断后程序通过查找该表得到相应的中断服务程序入口地址，然后再跳到相应的中断服务程序中执行。上电后从0x08000004处取出复位中断向量的地址，然后跳转到复位中断程序的入口(标号①所示)，执行结束后跳转到main函数中(标号②所示)。在执行main函数的过程中发生中断，则STM32强制将PC指针指回中断向量表处(标号③所示)，从中断向量表中找到相应的中断函数入口地址，跳转到相应的中断服务函数(标号④所示)，执行完中断函数后再返回到main函数中来(标号⑤所示)。

      若在STM32F103x中使用IAP方案，则内置的Flash分配情况大致如下图2-3。

|  |
| --- |
| <http://www.51hei.com/UpFiles/up/0/491511231663247.jpg> |
| 图2-3 |

在内置的Flash里面添加一个BootLoader程序，BootLoader程序和user application各有一个中断向量表，假设BootLoader程序占用的空间为N+M字节，则程序的走向应该如图2-2所示(借用网友的原图并做改动，其中虚线部分为原图步骤④⑤的走向，本人改为指向灰色部分)。

|  |
| --- |
| <http://www.51hei.com/UpFiles/up/0/491511231667910.jpg> |
| 图2-2 |

      上电初始程序依然从0x08000004处取出复位中断向量地址，执行复位中断函数后跳转到IAP的main(标号①所示)，在IAP的main函数执行完成后强制跳转到0x08000004+N+M处(标号②所示)，最后跳转到新的main函数中来(标号③所示)，当发生中断请求后，程序跳转到新的中断向量表中取出新的中断函数入口地址，再跳转到新的中断服务函数中执行(标号④⑤所示)，执行完中断函数后再返回到main函数中来(标号⑥所示)。

      对于步骤④⑤，网友认为是：“在main执行的过程中，如果CPU得到一个中断请求，PC指针仍强制跳转到地址0x08000004中断向量表处，而不是新的中断向量表，如图标号④所示，程序再根据我们设置的中断向量表偏移量，跳转到对应中断源新的中断服务程序中，如图标号⑤所示”。我对此的理解是：“当发生中断后，程序从0x08000004(旧)处的中断向量表中得到相应的中断服务函数入口地址，继而跳转到相应的中断服务程序”。但是旧的中断向量列表里边存放的是IAP程序中断函数的入口地址，它是如何得到user程序中断函数的入口地址呢？所以我觉得此种说法是错误的。“当发生中断时PC指针强制会跳转到0x08000004处”这种说法并没有错，只是忽略了后续的一些知识要点而导致这个说法出现矛盾。

      对于步骤④⑤我认为的是，在main函数的执行过程中，如果CPU得到一个中断请求，PC指针本来应该跳转到0x08000004处的中断向量表，由于我们设置了中断向量表偏移量为N+M，因此PC指针被强制跳转到0x08000004+N+M处的中断向量表中得到相应的中断函数地址(待求证)，再跳转到相应新的中断服务函数，执行结束后返回到main函数中来。

**三、实现过程**

      STM32F103ZET6的Flash地址为0x08000000—0x0807ffff共512KB，把这512KB的空间分为两块，第一块大小为32KB存放BootLoader程序，剩余的空间存放用户程序(根据实际情况分配这两块空间的大小，BootLoader程序占用的空间越小越好，则BootLoader地址为0x08000000—0x08007fff，用户程序地址为0x08008000—0x0807ffff。BootLoader流程图大致应该如下：

1、初始化时钟。

2、初始化中断向量表地址。

3、初始化按键。      (使用按键触发方式，上电时如果按键被按下则进行用户程序更新操作)

4、初始化串口。

5、检测按键是否被按下，是则执行步骤6，否则执行步骤10。

6、擦除用户程序(擦除0x08008000—0x0807ffff地址空间Flash)。

7、从串口读取新的用户代码数据，把代码写入用户程序空间。

8、检测串口数据接收完毕？是则执行步骤9，否则跳回步骤7。

9、用户程序更新完毕，等待重新上电或硬件复位。

10、跳转到用户程序(强制将PC指针跳转到0x08008000+4处)。

**到这里首先要解决的问题就有：**

1、如何进行对STM32的Flash进行擦除和写入操作。

2、中断向量表偏移如何设置。

3、如何改变代码存放的地址空间(因为BootLoader要存放在0x08000000处，用户程序要存放在0x08008000处,而默认的代码存放的地址空间为0x08000000)。

4、怎么进行PC指针的强制跳转，跳转时需要做些什么。

5、串口接收的用户代码数据是什么样的代码数据，是一种什么样的文件。

**问题的解决：**

1、使用STM32的固件库函数，只需调用几个库函数即可轻松解决，使用的固件库为stm32f10x\_flash.c文件，对Flash的操作过程简要为：Flash解锁àFlash擦除àFlash写入àFlash上锁。(对Flash编程的更详细操作参考STM32F10xxx闪存编程手册)

①解锁：

FLASH\_Unlock();            //解锁Flash

FLASH\_SetLatency(FLASH\_Latency\_2);            //因为系统时钟为72M所以要设置两个时钟周期的延时

②擦除：

for(i=0;i<240;i++)

{

  if(FLASH\_ErasePage(FLASH\_ADDR+i\*2048) != FLASH\_COMPLETE)      //一定要判断是否擦除成功

    return ERROR;

}

说明：FLASH\_ErasePage(uint32\_t Page\_Address)即为Flash擦除操作，按页擦除，每页2KB，Page\_Address为页的起始地址，如0x08000000是第一页起始地址，0x08000800为第二页起始地址，这里的操作擦除了0x08008000—0x0807ffff地址空间的Flash。

③写入：

unsigned char buf[1024];            //假设待写入的代码数据

unsigned short temp;            //临时数据

for(i=0;i<512;i++)

{

      temp = (buf[2\*i+1]<<8) | buf[2\*i];            //2个字节整合为1个半字

      if(FLASH\_ProgramHalfWord(ADDR,temp) != FLASH\_COMPLETE)      //判断是否写入成功

      {

            Return ERROR;

      }

ADDR +=2;      //地址要加2，因为每次写入的是2个字节(1个半字)

}

说明：因为STM32的Flash写入为双字节(1个半字)写入，FLASH\_ProgramHalfWord(uint32\_t Address, uint16\_t Data)函数即为对地址为Address写入1个半字的Data，每次写入完成后地址要加2。

④上锁：

FLASH\_Lock();      //Flash 上锁，一个固件库函数即可实现。

2、关于中断向量表的偏移设置，对于BootLoader程序只需设置中断向量表的指向在0x08000000处，对于用户程序需要设置中断向量表的指向在0x08008000处即可。

①在BootLoader程序的中断向量表指向设置中应有这么一句：

NVIC\_SetVectorTable(NVIC\_VectTab\_FLASH, 0x0);      //设置中断向量表指向

其中NVIC\_VectTab\_FLASH是个宏定义，的值为0x08000000。

②在用户程序的中断向量表指向设置用应有这么一句：

NVIC\_SetVectorTable(NVIC\_VectTab\_FLASH, 0x8000);      //设置中断向量表指向

3、确认代码存放的地址空间，在IAR和在Keil中的设置是不同的，网上有在Keil中设置的方法，设立介绍在IAR软件环境下的设置方法。

①在固件库目录\STM32F10x\_StdPeriph\_Lib\_V3.5.0\Project\STM32F10x\_StdPeriph\_Template\EWARM下找到一个stm32f10x\_flash.icf文件，将其复制到工程目录中来，在打开IAR工程，将配置文件添加到工程中，如下图3-2所示

|  |
| --- |
| <http://www.51hei.com/UpFiles/up/0/491511241656668.jpg> |
| 图3-1 |

②在工程中打开stm32f10x\_flash.icf该文件，修改两个参数即可改变代码存放的地址空间，图下图3-2所示。

|  |
| --- |
| <http://www.51hei.com/UpFiles/up/0/491511241694601.jpg> |
| 图3-2 |

4、关于PC指针的强制跳转，想在BootLoader程序中将PC指针跳转到用户代码处，可选择下面的操作

typedef        void (\*pFunction)(void);

pFunction       Jump\_To\_Application;

uint32\_t       JumpAddress;

#define       ApplicationAddress       0x08008000

if (((\*(\_\_IO uint32\_t\*)ApplicationAddress) & 0x2FFE0000 ) == 0x20000000)       //--------①

{

  JumpAddress = \*(\_\_IO uint32\_t\*) (ApplicationAddress + 4);                  //--------②

  Jump\_To\_Application = (pFunction) JumpAddress;                              //--------③

  \_\_set\_MSP(\*(\_\_IO uint32\_t\*) ApplicationAddress);                             //--------④

  Jump\_To\_Application();                                                                 //--------⑤

}

①因为用户程序开始位置(0x08008000处)的前4个字节存放的是堆栈的地址，堆栈地址必定是指向RAM空间的，而STM32的RAM空间起始地址为0x20000000，所以要进行判断。

②程序跳转地址的确认，前面已经说过0x08008004处的4个字节存放的是复位函数的入口地址，该句的意思为获得(ApplicationAddress + 4)地址处的数据，即为获得新的复位函数入口地址。

③令Jump\_To\_Application这个函数指针指向复位函数入口地址。

④堆栈的初始化，重新设定栈顶代地址，把栈顶地址设置为用户代码指向的栈顶地址。

⑤跳转到新的复位函数。

5、通过串口来接收代码数据，就是PC机通过串口将代码数据发送到STM32中去。这里就涉及到两个问题：

①数据怎么得来。

②数据传输的过程需要遵循的协议，什么时候开始，什么时候结束。

解决①：一般我们就将\*.hex文件使用JFlash-ARM打开再通过Jlink仿真器烧录到STM32芯片中，但是\*.hex文件里边包含的数据不纯粹是代码数据还有一些别的东西，而\*.bin文件数据就全部是代码数据。

在IAR软件环境中打开一个用户工程，先设置好中断向量表偏移和代码存放的地址空间后(前面已介绍过这两种方法)。设置工程如下图3-3所示，确认后重新编译工程，在工程的\Debug\Exe目录下会相应生成一个xxx.bin文件，这就是所需要的代码文件。

|  |
| --- |
| <http://www.51hei.com/UpFiles/up/0/491511241639740.jpg> |
| 图3-3 |

②数据通过串口来传输文件常用的协议有XModem、YModem、ZModem这三种协议，在PC端使用这些协议传输文件只需要PC的超级终端或者终端工具SecureCRT即可，但是在STM32这边的编程会增加一些困难(因为要先去读懂、解析这些协议，在通过编程来实现)。也可选择自己定义一套简单的传输协议，但同样会有一些困难(因为要在PC端进行文件和串口编程)。总之不管通过什么办法都行，只要能将xxx.bin文件数据通过串口全部发送到STM32并且STM32能够全部接收到这些数据并写入Flash即可(我选择后者，自定义传输协议并用VC进行文件和串口编程)。

**四、结束语**

      总的来说STM32的IAP方案实现需要在进行用户程序之前加一段Bootloader程序，BootLoader程序的作用就是：

①什么都不做，直接跳转到用户程序。

②删除原有的用户程序，读取\*.bin文件数据并将数据重新写入新的用户程序。

对于用户程序相比普通的编程只需要做三步改动即可

①改变中断向量表。

②改变代码存放的地址空间

③修改生成\*.bin文件

   使用通过UART的IAP方案并不是很好的选择，这只是IAP方案的一个机制，因为能使用PC机通过串口升级程序，同样能通过Jlink烧写程序，并且自定义的串口通讯协议在没有校CRC校验的情况下不能及时发现数据传输过程发生的错误。这里推荐使用SD卡(或U盘)进行用户程序更新，将\*.bin文件复制到SD卡(或U盘)中，STM32再通过读取SD卡(或U盘)的\*.bin文件进行用户程序更新，这也避免了STM32与PC笨重的通讯，只需插一个SD卡(或U盘)更显得人性化一些，但需要去弄懂STM32如何与SD卡(或U盘)的通讯。

# 基于IAP和Keil MDK的远程升级设计

2010年11月14日 12:03:00 [zhzht19861011](https://me.csdn.net/zhzht19861011) 阅读数：22894 标签： [flash](http://so.csdn.net/so/search/s.do?q=flash&t=blog)[编程](http://so.csdn.net/so/search/s.do?q=%E7%BC%96%E7%A8%8B&t=blog)[存储](http://so.csdn.net/so/search/s.do?q=%E5%AD%98%E5%82%A8&t=blog)[hex](http://so.csdn.net/so/search/s.do?q=hex&t=blog)[编译器](http://so.csdn.net/so/search/s.do?q=%E7%BC%96%E8%AF%91%E5%99%A8&t=blog) 更多

个人分类： [keil MDK](https://blog.csdn.net/zhzht19861011/article/category/724257)

版权声明：本文为博主原创文章，未经博主允许不得转载。联系邮箱:zhzhchang@126.com https://blog.csdn.net/zhzht19861011/article/details/6008702

      写在前面：三周之前，我突然想写一个远程升级的程序。那个时候我只是大概知道IAP的意思是在应用编程，但怎么编，我还一无所知。我给自己定下一个个阶段目标，从最基础的代码一点点写起，解决一个又一个的问题。三个周之后，我用自己设计的方法实验了50多次，无一例升级失败。

      三个周来，遇到了很多的不解、困惑，甚至是想放弃，但我现在想说的是：很多未知的困难会挡在我们面前，我们会感觉毫无头绪甚至觉得毫无出路忍不住要放弃，但多坚持一下，那些困难不但能烟消云散还能带给我们进步。

      本设计是基于LPC2114和Keil MDK（V4.10），但所有支持IAP的处理器都可借鉴本方案，重要的是思想，而不是用什么。

**0 引言**

      在应用编程(IAP)技术为系统在线升级和远程升级提供了良好的解决方案，也为数据存储和现场固件的升级都带来了极大的灵活性。通常可利用芯片的串行口接到计算机的RS232口、通过现有的Internet或、无线网络或者其他通信方式很方便地实现在线以及远程升级和维护。

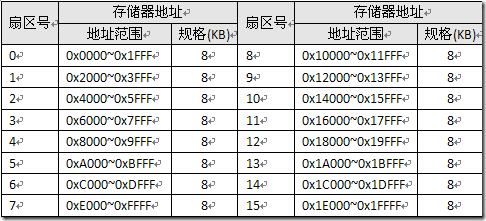
      本文以NXP的LPC2114 ARM微处理器为平台，以Keil MDK为开发工具，阐述IAP的原理、Flash的划分、分散加载机制、中断重映射以及在线升级的实现方案及其优化。本方案使用多种校验技术，最大限度的保障传输数据的正确性；使用bootloader机制，即使因意外事件（断电，编程Flash失败等）造成升级失败后，程序也能返回到升级前的状态。

**1 LPC2114的Flash规划**

**1.1 扇区描述**

      LPC2114共有128KB片内Flash，共分为16个扇区，分别为0扇区～15扇区，每个扇区为8KB存储空间。其中第15扇区出厂时被固化为Boot Block区，控制复位后的初始化操作，并提供实现Flash 编程的方法。所以用户可用的Flash空间只有120KB。IAP程序固化于Boot Block中，IAP操作是以扇区为单位，并占用片内RAM的高32字节。下表列出LPC2114器件所包含的扇区数和存储器地址.

表**1.1 LPC2114 Flash**器件中的扇区

[](http://hi.csdn.net/attachment/201011/14/0_1289736221q200.gif)

**1.2 Flash的扇区划分**

      本设计将Flash划分为四个区，扇区0存放跳转程序和升级引导程序（Bootloader）。分站上电后执行跳转程序，跳转到用户程序处。用户程序运行过程中，如果接收到升级指令，会从用户程序跳转到引导程序区（Bootloader），接收新程序数据包，完成Flash编程并跳转到新程序区执行程序。扇区1～扇区7为程序存储低区；扇区8～扇区13为程序存储高区；扇区14存放当前程序运行区域标志，如果当前程序运行在高区，该标志区的最低四个字节为0x00010000，如果当前程序运行在低区，该标志区的最低四个字节为0x00008000。

**2 IAP的原理与软件设计**

**2.1 IAP的原理**

      IAP函数是固化在微处理器内部flash上的一些函数代码，最终的用户程序可以直接通过调用这些函数来对内部flash进行擦除和编程操作。LPC2114微处理器的内部flash有一个块称为Boot Block，位于flash的顶端，可供调用的IAP函数就位于该块中。上电后Boot Block被映射到内部地址空间的顶端，同样IAP函数人口地址也被映射到地址0x7ffffff0处。用户可通过跳转到该地址来调用相应的lAP函数。

**2.2 IAP 命令**

      对于在应用编程来说，应当通过寄存器r0 中的字指针指向存储器(RAM)包含的命令代码和参数来调用IAP 程序。IAP 命令的结果返回到寄存器r1 所指向的返回表。用户可通过传递寄存器r0 和r1 中的相同指针重用命令表来得到结果。参数表应当大到足够保存所有的结果以防结果的数目大于参数的数目。参数传递见图2-1。参数和结果的数目根据IAP命令而有所不同。参数的最大数目为5，由“将RAM 内容复制到Flash”命令传递。结果的最大数目为2，由“扇区查空”命令返回。命令处理程序在接收到一个未定义的命令时发送状态代码INVALID\_COMMAND。IAP 程序是thumb 代码，位于地址0x7FFFFFF0。

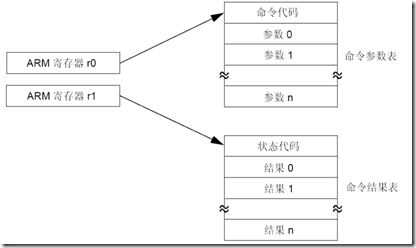
[](http://hi.csdn.net/attachment/201011/14/0_1289736222Q0WR.gif)

图2-1 IAP的参数传递

表2-1描述了IAP的命令。

表2-1 IAP 命令汇总

|  |  |  |
| --- | --- | --- |
| IAP命令 | 命令代码 | 描述 |
| 准备编程扇区 | 50 | 该命令必须在执行“将 RAM 内容复制到Flash”或“擦除扇区”命令之前执行。这两个命令的成功执行会导致相关的扇区再次被保护。该命令不能用于boot 扇区。要准备单个扇区，可将起始和结束扇区号设置为相同值。 |
| 将RAM内容复制到Flash | 51 | 该命令用于编程 Flash 存储器。受影响的扇区应当先通过调用“准备写操作的扇区”命令准备。当成功执行复制命令后，扇区将自动受到保护。该命令不能写boot 扇区。 |
| 擦除扇区 | 52 | 该命令用于擦除片内 Flash 存储器的一个或多个扇区。boot 扇区不能由该命令擦除。要擦除单个扇区可将起始和结束扇区号设定为相同值。 |
| 扇区查空 | 53 | 该命令用于对片内 Flash 存储器的一个或多个扇区进行查空。要查空单个扇区可将起始和结束扇区号设定为相同值。 |
| 读器件ID | 54 | 该命令用于读取器件的 ID 号。 |
| 读Boot版本 | 55 | 该命令用于读取 boot 代码版本号。 |
| IAP比较 | 56 | 该命令用来比较两个地址单元的存储器内容。当源或目标地址包含从地址**0**开始的前**64**字节中的任意一个时，比较的结果不一定正确。前**64**字节重新映射到**Flash boot**扇区。 |

**2.3 IAP 编程函数接口**

      IAP 功能可用下面的C 代码来调用。

      定义 IAP 程序的入口地址。由于IAP 地址的第0 位是1，因此，当程序计数器转移到该地址时会引起Thumb 指令集的变化。

#define IAP\_LOCATION 0x7ffffff1

      定义数据结构或指针，将IAP 命令表和结果表传递给IAP 函数

1. unsigned long command[5];
2. unsigned long result[2];

      定义函数类型指针，函数包含2 个参数，无返回值。注意：IAP 将函数结果和R1 中的表格基址一同返回。

1. typedef void (\*IAP) (unsigned int [ ] , unsigned int [ ]);
2. IAP iap\_entry;

      设置函数指针

iap\_entry=(IAP) IAP\_LOCATION;

    使用下面的语句来调用IAP。

iap\_entry (command , result);

      Flash 存储器在写或擦除操作过程中不可被访问。执行Flash 写/擦除操作的IAP 命令使用片内RAM 顶端的32 个字节空间。如果应用程序中允许IAP 编程，那么用户程序不应使用该空间。

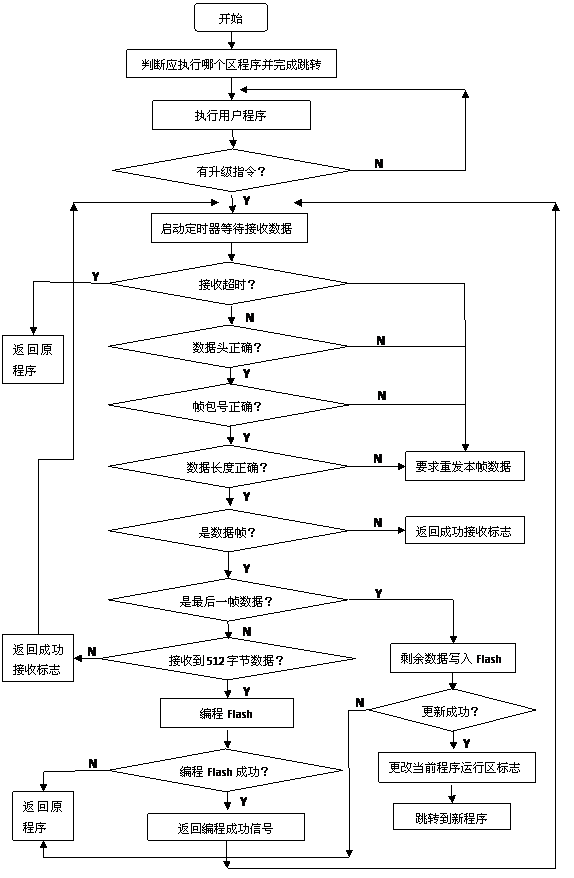
**3 LPC2114升级实现过程**

      由于在升级程序软件设计中，分散加载机制、中断向量的重映射、软中断等的实现还与所使用的编译器紧密相关，因此，本文结合Keil MDK（V4.10）编译工具，来详细阐述升级程序的实现过程。

**3.1 总体思路**

      分站上电后，首先运行位于Flash 0x000～0x3FF中的跳转程序。跳转程序会读取位于14扇区的当前程序运行标志，如果该扇区的最低四个字节为0x00010000，表示当前程序运行在高区，跳转程序会跳转到Flash的0x00010000处执行用户程序；如果该标志区的最低四个字节为0x00008000，表示当前程序运行在低区，跳转程序会跳转到Flash的0x00002000处执行用户程序。用户程序正常执行后，会按照设计进行正常的程序采集、数据处理传送。当接收到升级命令后，用户程序会跳转到Flash的0x00000400处的Bootloader处进行升级的一些操作。当升级成功后，Bootloader程序更新当前程序运行区标志，程序跳转到新程序处运行，如果升级不成功，返回升级前的程序。

      流程图如下所示：

[](http://hi.csdn.net/attachment/201011/14/0_1289736223ZlUq.gif)

**3.2 跳转程序的设计**

      跳转程序是分站上电后最先运行的程序，根据当前程序运行区标志，跳转到相应的用户程序区执行。本段程序占用Flash的最低1K字节空间，与Bootloader同在第0扇区。

      跳转程序的启动代码仅初始化堆栈，不使用PLL和存储加速功能。下面描述了跳转程序的主要启动代码。

1. ; Enter User Mode and set its Stack Pointer
2. MSR CPSR\_c, #Mode\_USR
3. MOV SP, R0
4. SUB SL, SP, #USR\_Stack\_Size
5. ; Enter the C code
6. IMPORT \_\_main
7. LDR R0, =\_\_main
8. BX R0

      当跳转程序确定要跳转到高区用户程序或者低区用户程序后，使用函数指针跳转到0x00010000处（高区用户函数入口地址）或0x00002000处（低区用户函数入口地址）。

      定义函数指针：

void (\*UserProgram)() ;

      指定入口地址：

1. UserProgram = (void (\*)()) (0x00010000)；
2. UserProgram = (void (\*)()) (0x00002000)；

      实现跳转：

(\*UserProgram)() ；

      要将用户代码精确定位到Flash的0x00010000处（高区用户函数入口地址）或0x00002000处（低区用户函数入口地址），需要使用编译器的分散加载机制，将在Bootloader中详细描述实现过程。

      另外，跳转程序还在烧录代码的同时初始化当前程序运行区标志，即对Flash的0x0001C000地址处写入0x00008000，表示当前用户程序在低区。主要使用了编译器的**\_\_at**关键字：精确定位变量。需要注意的是，使用该关键字必须包含头文件absacc.h。

const uint32 x \_\_at(0x0001C000)=0x00008000; *//初始化用户程序标志区,默认运行低区*

**3.3 升级程序Bootloader的设计**

      升级程序的好坏，在很大程度上取决于Bootloader设计的好坏。

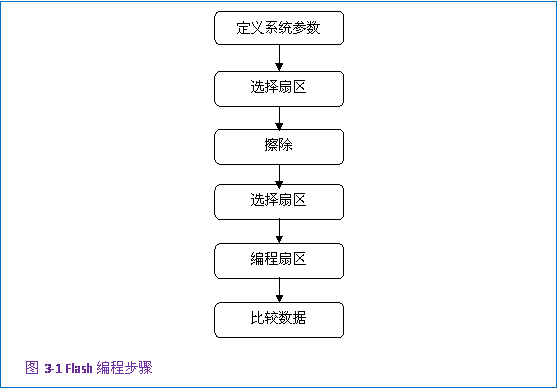
      一个优秀的IAP升级Bootloader，必须做好升级中出现故障等异常的处理。保证系统不会崩溃，即使升级失败，也能返回升级前的程序。

* 有升级指令，进行初始化工作（串口、定时器、看门狗）
* 接收升级数据包，检测帧头、长度、帧号、数据区校验，最大程度的保证升级数据的完整性、正确性。
* 实时检测接收状态，10 S内没有接收到数据或接收到的数据包都是错的，则退出升级，返回原程序。
* 接收的数据按照512字节一组写入Flash，写入后再读出与原数据进行对比校验，校验成功后，本次编程Flash成功。允许连续3次编程Flash，三次都不成功，退出升级程序，执行原程序。
* 升级成功后，更新当前程序运行区标志，跳转到新程序，同时原程序保存。

      本设计的Bootload位于Flash的0x400开始的扇区0存储区内，使用分散加载机制，将程序的入口地址定位到0x00000400处。当用户程序接收到升级指令后，就会使用函数指针跳转到这个入口处。

**3.3.1 使用IAP**

      图3-1 描述了使用IAP编程Flash所必须的步骤。

[](http://hi.csdn.net/attachment/201011/14/0_1289736224b93q.gif)

**3.3.1.1 定义系统参数**

      在使用IAP前，需要定义一些系统参数，比如系统时钟、IAP中断入口、输入输出缓存。

1. #define IAP\_CLK 11059200UL
2. #define IAP\_LOCATION 0x7FFFFFF1
3. typedef void(\*IAP)(uint32 [],uint32 []); *//定义函数类型指针*
4. IAP iap\_entry=(IAP)IAP\_LOCATION; *//设置函数指针*
5. unsigned long command[5] = {0,0,0,0,0};
6. unsigned long result[2]= {0,0};

**3.3.1.2 选择扇区**

      在任何擦除和编程Flash之前，必须选中扇区，可以选中一个或多个。

1. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
2. *\* 名称：SelSector()*
3. *\* 功能：IAP操作扇区选择，命令代码50。*
4. *\* 入口参数：sec1 起始扇区*
5. *\* sec2 终止扇区*
6. *\* 出口参数：IAP返回值(paramout缓冲区) CMD\_SUCCESS,BUSY,INVALID\_SECTOR*
7. *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
8. void SelSector(uint8 sec1, uint8 sec2)
9. {
10. paramin[0] = IAP\_SELSECTOR; *// 设置命令字*
11. paramin[1] = sec1; *// 设置参数*
12. paramin[2] = sec2;
13. iap\_entry(paramin, paramout); *// 调用IAP服务程序*
14. }

**3.3.1.3 擦除扇区**

      在编程Flash前必须执行擦除操作，如果某个扇区已经擦除，就不需要再次擦除。可以一次擦除一个或多个扇区。

1. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
2. *\* 名称：EraseSector()*
3. *\* 功能：扇区擦除，命令代码52。*
4. *\* 入口参数：sec1 起始扇区*
5. *\* sec2 终止扇区*
6. *\* 出口参数：IAP返回值(paramout缓冲区) CMD\_SUCCESS,BUSY,INVALID\_SECTOR \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
7. void EraseSector(uint8 sec1, uint8 sec2)
8. {
9. paramin[0] = IAP\_ERASESECTOR; *// 设置命令字*
10. paramin[1] = sec1; *// 设置参数*
11. paramin[2] = sec2;
12. paramin[3] = Fosc/1000; *// 当不使用PLL功能时，Fcclk=Fosc*
13. iap\_entry(paramin, paramout); *// 调用IAP服务程序*
14. }

**3.3.1.4 编程扇区**

      通过这个过程，数据可以从RAM中编程到片内Flash中。

**注：**

1. **数据只能从片内SRAM编程到片内Flash。**
2. **片内Flash的地址必须512字节对齐。**
3. **片内RAM应位于局部总线，即USB或以太网的SRAM不可以使用。**
4. **每一次编程字节应该是512、1024、4096、8192中的一个。**
5. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
6. *\* 名称：RamToFlash()*
7. *\* 功能：复制RAM的数据到FLASH，命令代码51。*
8. *\* 入口参数：dst 目标地址，即FLASH起始地址。以512字节为分界*
9. *\* src 源地址，即RAM地址。地址必须字对齐*
10. *\* no 复制字节个数，为512/1024/4096/8192*
11. *\* 出口参数：IAP返回值(paramout缓冲区) CMD\_SUCCESS,SRC\_ADDR\_ERROR,DST\_ADDR\_ERROR,*
12. *SRC\_ADDR\_NOT\_MAPPED,DST\_ADDR\_NOT\_MAPPED,COUNT\_ERROR,BUSY,未选择扇区*
13. *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
14. void RamToFlash(uint32 dst, uint32 src, uint32 no)
15. {
16. paramin[0] = IAP\_RAMTOFLASH; *// 设置命令字*
17. paramin[1] = dst; *// 设置参数*
18. paramin[2] = src;
19. paramin[3] = no;
20. paramin[4] = Fosc/1000; *// 当不使用PLL功能时，Fcclk=Fosc*
21. iap\_entry(paramin, paramout); *// 调用IAP服务程序*
22. }

**3.3.1.5 比较数据**

      通过这个函数，可以检查写入Flash中的数据和RAM中的是否相同。

**注意源地址、目标地址和字节数必须是4的倍数。可使用Keil MDK提供的关键字\_\_align(n) 来指定n字节对齐。**

1. */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
2. *\* 名称：Compare()*
3. *\* 功能：校验数据，命令代码56。*
4. *\* 入口参数：dst 目标地址，即RAM/FLASH起始地址。地址必须字对齐*
5. *\* src 源地址，即FLASH/RAM地址。地址必须字对齐*
6. *\* no 复制字节个数，必须能被4整除*
7. *\* 出口参数：IAP返回值(paramout缓冲区) CMD\_SUCCESS,COMPARE\_ERROR,ADDR\_ERROR*
8. *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
9. void Compare(uint32 dst, uint32 src, uint32 no)
10. {
11. paramin[0] = IAP\_COMPARE; *// 设置命令字*
12. paramin[1] = dst; *// 设置参数*
13. paramin[2] = src;
14. paramin[3] = no;
15. iap\_entry(paramin, paramout); *// 调用IAP服务程序*
16. }

**3.3.2 IAP编程期间的中断管理**

      LPC2114片上Flash在擦除/编程期间绝不可被中断打断。但Bootloader中定时和串口接收又使用了中断，因此必须在擦除/编程之前禁止总中断，待操作完成后再使能总中断。Bootloader运行在用户模式下，不具有禁止/使能中断的权力，所以在本设计中使用软中断禁止/使能总中断。Keil MDK提供了关键字\_\_svc来触发软中断。

      软中断函数声明：

1. \_\_svc(0x00) void EnableIrq(void); *//使能中断,软中断0*
2. \_\_svc(0x01) void DisableIrq(void); *//禁止中断,软中断1*

      软中断函数代码：

1. */\**
2. *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
3. *\* 功 能:禁止中断*
4. *\* 描 述:利用软中断实现在用户模式下调用函数关中断*
5. *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*
6. void DisableIrqFunc(void)
7. {
8. int temp;
9. \_\_asm
10. {
11. MRS temp,SPSR
12. ORR temp,temp,#0x80
13. MSR SPSR\_c,temp
14. }
15. }
16. */\**
17. *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
18. *\* 功 能:使能中断*
19. *\* 描 述:利用软中断实现在用户模式下调用函数开中断*
20. *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
21. *\*/*
22. void EnableIrqFunc(void)
23. {
24. int temp;
25. \_\_asm
26. {
27. MRS temp,SPSR
28. BIC temp,temp,#0x80
29. MSR SPSR\_c,temp
30. }
31. }

      更改启动代码，挂接软中断入口：

1. ;软中断入口
2. EXPORT SWI\_Handler
3. extern EnableIrq1
4. extern DisableIrq1
5. SWI\_Handler
6. STMFD SP!, {R0,R12,LR} ；入栈
7. LDR R0, [LR,#-4] ；取软中断指令，软中断号就包含其中
8. BIC R0,R0,#0xFF000000
9. CMP R0,#0 ；判断是否软中断0
10. BLEQ EnableIrqFunc
11. BLNE DisableIrqFunc
12. LDMFD SP!,{R0,R12,PC}^

        在程序中,如果想禁止中断,只需使用DisableIrq()；若是能中断，只需使用EnableIrq()。

**3.3.3 使用分散加载机制精确定位入口地址**

      应用程序接收到升级指令后，会跳转到0x00000400处执行Bootloader升级程序。因此Bootloader程序的入口地址必须精确定位到0x00000400处。这可以使用Keil MDK提供的分散加载机制来完成。

      分散加载代码见代码3-8.

1. ; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
2. ; \*\*\* Scatter-Loading Description File generated by uVision \*\*\*
3. ; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
4. LR\_IROM1 0x00000400 0x00001C00 { ; load region size\_region
5. ER\_IROM1 0x00000400 0x00001C00 { ; load address = execution address
6. \*.o (RESET, +First)
7. \*(InRoot$$Sections)
8. .ANY (+RO)
9. }
10. RW\_IRAM1 0x40000040 0x00003FA0 { ; RW data
11. .ANY (+RW +ZI)
12. }
13. }

      这段代码显示出Bootloader程序从0x00000400处开始执行，最多占用0x1C00字节的Flash空间。另外，该程序的RAM从0x40000040开始，长度为0x3FA0个字节。这样RAM的低64字节保留给中断向量映射使用，高32字节保留给IAP编程使用。

**3.3.4 中断向量的重映射**

      Bootloader的起始地址位于0x00000400，中断向量也从这一地址开始存储。但默认情况下ARM发生异常时，会跳转到0x00000000处的64字节中断向量区域执行相应操作，所以为了使Bootloader能相应中断，必须将位于0x00000400开始的64字节中断向量表重映射到RAM的低区。LPC2114使用向寄存器MEMMAP写入0x02来完成这一过程。

      代码3-9 描述了中断向量重映射的过程。

1. ; Copy Exception Vectors to Internal RAM
2. ADR R8, Vectors ; 源地址
3. LDR R9, =RAM\_BASE ; 目标地址，这里是0x40000000
4. LDMIA R8!, {R0-R7} ; 装载向量表
5. STMIA R9!, {R0-R7} ; 存储向量表
6. LDMIA R8!, {R0-R7} ; 装载处理程序地址
7. STMIA R9!, {R0-R7} ; 存储处理程序地址
8. ; Memory Mapping (when Interrupt Vectors are in RAM)
9. MEMMAP EQU 0xE01FC040 ; Memory Mapping Control
10. IF :DEF:REMAP
11. LDR R0, =MEMMAP
12. IF :DEF:EXTMEM\_MODE
13. MOV R1, #3
14. ELIF :DEF:RAM\_MODE
15. MOV R1, #2
16. ELSE
17. MOV R1, #1
18. ENDIF
19. STR R1, [R0]
20. ENDIF

      由于Keil MDK提供的启动代码中使用条件编译指令，所以，要想正确的执行中断向量重映射，还需要在Keil MDK编译器工程设置Options for target“你的工程目标名”下的Asm标签中找到Define编辑框，在编辑框中键入“REMAP RAM\_MODE”。如图3-2所示

[](http://hi.csdn.net/attachment/201011/14/0_1289736224HfGe.gif)

图3-2

**注意：在擦除/编程Flash的时候还应该禁止PLL、存储器加速模块。**

**3.4 用户程序的设计**

      用户程序运行在高区（扇区8～13）或者低区（扇区1～7），用于实现数据的采集、处理和上传等等，用户程序除本身功能的要求外，还需要注意：

* 使用分散加载机制，将程序入口精确定位到0x00010000（高区）或0x00008000（低区）。
* 进行中断向量重映射，映射到RAM最底处。

**4 通讯协议与上位机软件**

**4.1 Intel的hex格式**

      Intel hex文件是记录文本行的ASCII文本文件，在Intel HEX文件中，每一行是一个HEX记录，由十六进制数组成的机器码或者数据常量。一个数据记录以一个回车和一个换行结束。

      一个Intel HEX文件可以包含任意多的十六进制记录,每条记录有五个域,下面是一个记录的格式.

                        : LL AAAA TT [DD...] CC

      每一组字母是独立的一域,每一个字母是一个十六进制数字,每一域至少由两个十六进制数字组成,下面是字节的描述.

* :冒号 是每一条Intel HEX记录的开始
* LL 是这条记录的长度域,他表示数据(dd)的字节数目.
* AAAA 是地址域,他表示数据的起始地址
* TT 这个域表示这条HEX记录的类型,他有可能是下面这几种类型
  + 00 ----数据记录
  + 01 ----文件结束记录
  + 02 ----扩展段地址记录
  + 04 ----扩展线性地址记录
* DD 是数据域,表示一个字节的数据,一个记录可能有多个数据字节,字节数目可以查看LL域的说明。
* CC 是效验和域,表示记录的效验和,计算方法是将本条记录冒号开始的所有字母（包括校验字节）相加之后等于0x00。

      一个Intel HEX文件必须有一个文件结束记录,这个记录的类型域必须是01,

      一个EOF记录总是这样:

                      :00000001FF

* 00是记录中数据字节的数目
* 0000这个地址对于EOF记录来说无任何意义
* 01记录类型是01(文件结束记录标示)

**4.2 对上位机软件的要求**

* 上位机具备解析重组Intel HEX文件的能力.
* 上位机软件应能识别分站发来的应答信号并做出正确的响应。
* 上位机应能够检验代码的完整性。
* 上位机能根据分站发出的程序所在高区或低区标志，自动判别当前升级程序是否和升级区域相对应。

**5 实验数据**

      为验证升级程序的稳定性，对分站进行重上电、复位、远程升级等一些列实验，实验记录及如下。

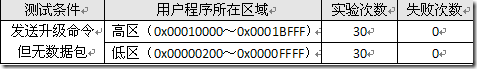
**5.1 测试程序跳转功能.**

      程序在上电或复位之后，应顺利跳转到用户程序。

[](http://hi.csdn.net/attachment/201011/14/0_1289736226T9Qi.gif)

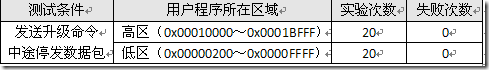
**5.2 测试Bootloader（一）**

      上位机发送升级命令但不发送升级数据包，程序应能进入Bootloader并发送当前程序所在的区域（高区或者低区代号），10S后程序应跳转到用户程序。

[](http://hi.csdn.net/attachment/201011/14/0_1289736226Z006.gif)

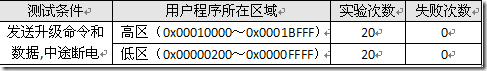
**5.3 测试Bootloader（二）**

      上位机发送升级命令，发送升级数据包，但发送到一半时停止发送。程序在10S后应能跳转到用户程序区。

[](http://hi.csdn.net/attachment/201011/14/0_1289736227h64J.gif)

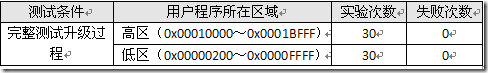
**5.4 测试Bootloader（三）**

      上位机发送升级命令，发送升级数据包，但发送中途给分站断电，重新上电后，应还能执行原来的程序。

[](http://hi.csdn.net/attachment/201011/14/0_1289736227UoMv.gif)

**5.5 测试Bootloader（四）**

      上位机发送升级命令，发送完成升级数据包。程序应能接收升级数据包并编程Flash，完成用户程序的更新，更新用户程序后，跳转到新的用户程序。

[](http://hi.csdn.net/attachment/201011/14/0_1289736228K7c8.gif)

**6.总结**

      本次升级方案虽然是以LPC2114为基础的，但任何具有IAP功能的单片机、ARM都可使用本设计方案。

      设计的重点在于如何保证升级的安全性，分站采取了一些列校验、超时处理以及看门狗等措施，一是保障升级数据包的正确传送，二是即使升级失败也能退回原升级程序。上位机的校验措施需相关部门配合。从实验数据来看，进行了几十次的远程升级，未有一例失败，安全性能可以得到保证。

**7.参考文献**

1. 周立功等 ARM微控制器基础与实战（第二版） 北京航空航天大学出版社 2005
2. LPC2114/2124/2212/2214 使用指南.Pdf 广州周立功单片机发展有限公司
3. 韦文祥 朱志杰 车琳娜 郭宝泉 基于LPC21 24的一个远程系统软件升级方案 单片机与嵌入式系统应用 2006第三期
4. 许文杰 丁志冈 张 泉基于ARM 处理器的IAP设计及应用 计算机应用与软件 2009第3期
5. 姜晓梅 李祥和 任朝荣 姚明基于ARM的IAP在线及远程升级技术 计算机应用 2008第二期
6. RealView 编译工具-编译器参考指南.pdf ARM Limited 2009.3
7. RealView Compilation Tools（连接器用户指南）.pdf ARM Limited 2009.3
8. RealView编译工具-编译器用户指南.pdf ARM Limited 2009.1
9. **RealView**编译工具-链接器参考指南.pdf ARM Limited 2008.9
10. Intel HEX文件格式
11. LPC2000 secondary bootloader for code update using IAP NXP Semiconductors 2009.5.26

**8.后记**

**8.1 后记1**

      分散加载文件，软中断，中断向量表重映射，变量对齐，精确定位变量等等这些东西的详细讲解在我的参考资料上都能找的到，发现问题并能解决它，是件很美妙的事情，所以我没打算也没时间详细写这些东西的用法。

      需要说的是，我在设计的时候走了一个弯路，现在想想还觉得挺可笑。我以为上面讲的东西要在一个工程里面实现才好，这样才能生成一个.hex可烧录文件，可以一次性的将用户程序、Bootloader程序烧写进处理器，我想弯了。正确的做法是建四个工程：跳转程序、Bootloader、用户低区程序、用户高区程序。如果你懂了.hex文件的格式，就完全可以将跳转程序、Bootloader和用户低区程序（或者跳转程序、Bootloader和用户高区程序）这三个工程生成的.hex文件合成一个。灵活多变的处理问题，这是我最大的收获。

**8.2 后记2**

      很多同学看完后都希望得到源码,这种心情我是理解的,最初的时候我也希望有一套别人的源码的,毕竟这样可以进行的快点.所以我将一个远程升级的例子放在下面的链接里,大家想看看的就去下载吧,这个不是我产品中用到的,因为我的代码毕竟是含有公司的一些信息.如果我最近有时间,会把其中的英文文档翻译一下的.

      链接:<http://download.csdn.net/detail/zhzht19861011/3618966>

**8.3 后记3**

      补充一个隐含的Bug，从Bootloader跳转到应用程序前，应该使处理器进入特权模式，否则用户程序可能出现堆栈错误。----2012.12.10

**8.4 后记4**

      重新编排了文章格式。本来这篇文章是在Word中书写的，排版十分清晰，但是从Word复制csdn博客，很多word格式csdn并不支持，因此版面很是凌乱，这一凌乱一下就持续了四年多。

      以前写博客是为了给自己看，每当有个心得，记下来以便以后查阅。后来发现自己写的很多文章会有比较高的访问量，很多网友会留言说“xx文章对我帮助很大”，这让我意识到原来我可以帮助很多人。既然有人看，那么对我而言，有两个方面需要特别考虑：一是要准确，二是要清晰。准确源于平时积累，书写成文字必须多考证；清晰不仅要求文章内容逻辑上要清晰，而且文章排版也要清晰。这是我重新排版的原因。

      因为各种原因，我的文章可能不总是准确，如果读者发现错误，请正面留言告诉我。CSDN博客有个功能， 如果你觉得这篇文章很好，可以在文章最后点“顶”，反之你觉得这篇文章误导了你，你可以点“踩”。这篇文章就被人点了“踩”，但并没有反馈为什么点踩。如果你不告诉我点踩的原因，比如里面有错误，那么不仅我没有进步，别人也可能会继续看到错误的东西。所以，如果你发现了任何错误或者不满，正面告诉我，对我以及以后看文章的网友都是件好事。

# keil MDK中如何生成\*.bin格式的文件

2010年10月08日 18:18:00 [zhzht19861011](https://me.csdn.net/zhzht19861011) 阅读数：62657 标签： [工具](http://so.csdn.net/so/search/s.do?q=%E5%B7%A5%E5%85%B7&t=blog)[motorola](http://so.csdn.net/so/search/s.do?q=motorola&t=blog)[编译器](http://so.csdn.net/so/search/s.do?q=%E7%BC%96%E8%AF%91%E5%99%A8&t=blog)[file](http://so.csdn.net/so/search/s.do?q=file&t=blog)[user](http://so.csdn.net/so/search/s.do?q=user&t=blog)[嵌入式](http://so.csdn.net/so/search/s.do?q=%E5%B5%8C%E5%85%A5%E5%BC%8F&t=blog)更多

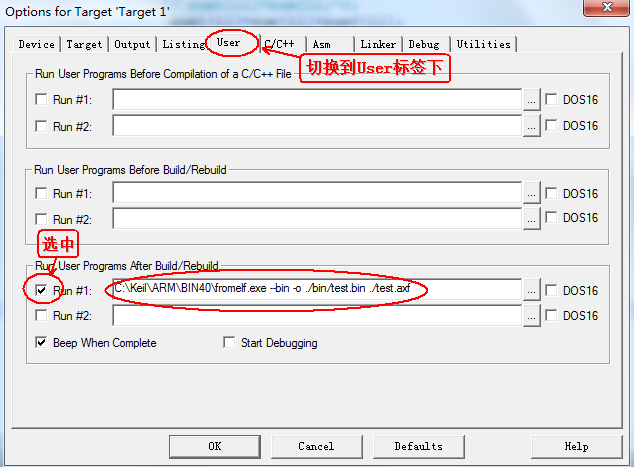
个人分类： [keil MDK](https://blog.csdn.net/zhzht19861011/article/category/724257)

版权声明：本文为博主原创文章，未经博主允许不得转载。联系邮箱:zhzhchang@126.com https://blog.csdn.net/zhzht19861011/article/details/5927924

      在Realview MDK的集成开发环境中，默认情况下可以生成\*.axf格式的调试文件和\*.hex格式的可执行文件。虽然这两个格式的文件非常有利于ULINK2仿真器的下载和调试，但是ADS的用户更习惯于使用\*.bin格式的文件，甚至有些嵌入式软件开发者已经拥有了\*.bin格式文件的调试或烧写工具。为了充分地利用现有的工具，同时发挥Realview MDK集成开发环境的优势，将\*.axf格式文件或\*.hex格式文件转换成\*.bin格式的文件是十分自然的想法。本文将详细的探讨这种转换方法。

      在Realview MDK中，生成\*.bin格式文件的工具是ARM公司的RVCT开发套件中的fromelf.exe。默认安装下，它位于C/：Keil/ARM/BIN40文件夹下。

      下面将介绍它在Keil MDK中的使用方法：  
      1. 新建一个工程，例如test.uvproj；  
      2. 打开Options for Target ‘target 1’对话框，选择User标签页；  
      3. 构选Run User Programs After Build/Rebuild框中的Run #1多选框，在后边的文本框中输入C:/Keil/ARM/BIN31/fromelf.exe --bin -o ./bin/test.bin ./test.axf命令行；步骤2、3可见下图所示。

  
      4. 重新编译文件，在./bin/文件夹下生成了test.bin文件。  
经过上述4步的操作以后，将得到我们希望的test.bin格式的文件。

ps：步骤3中的"C:/Keil/ARM/BIN31/fromelf.exe ”也可以通过点击下图红色标注出进入keil安装目录下选择。

http://hi.csdn.net/attachment/201010/8/0_1286533031Rn3n.gif

        以上是很久前写的,对于如何使用fromelf工具,通过上面的示例,想必都能很轻松的生成bin文件,今天补写一下fromelf工具的基本命令:

        --bin:输出二进制文件

        --i32:Intel 32位Hex

        --m32：Motorola 32位Hex

         --output <file>:file为输出文件名

         -o<file>:这个是armcc编译器命令,也可用于这里,指定输出文件的名字

其它命令可以在Dos中运行fromelf.exe工具,会列出详细命令和用法.

[](https://me.csdn.net/libsong)

窗体顶端



窗体底端

* [](https://me.csdn.net/a454672135)

[a454672135：](https://me.csdn.net/a454672135) 博主，您不觉得应该是反斜杠吗？&quot;\&quot;,&quot;/&quot;区别很大。(2年前#9楼)

* [](https://me.csdn.net/a454672135)

[a454672135：](https://me.csdn.net/a454672135) 博主，您不觉得应该是反斜杠吗？&quot;\&quot;,&quot;/&quot;区别很大。(2年前#8楼)

* [](https://me.csdn.net/x232323h)

[思之于心：](https://me.csdn.net/x232323h) 收藏了，才开始上手(3年前#7楼)

* [varding](https://me.csdn.net/varding)

[varding：](https://me.csdn.net/varding) 好用，谢谢 顺便说下mdk 4.73的fromelf在：C:\Keil\ARM\ARMCC\bin下(4年前#6楼)