第五十三章 串口 IAP 实验

IAP，即在应用编程。很多单片机都支持这个功能，STM32 也不例外。在之前的 FLASH 模拟 EEPROM 实验里面，我们学习了 STM32 的 FLASH 自编程，本章我们将结合 FLASH 自编程的知识，通过 STM32 的串口实现一个简单的 IAP 功能本章分为如下几个部：

53.1 IAP 简介

53.2 硬件设计

53.3 软件设计

53.4 下载验证

53.1 IAP 简介

IAP（In Application Programming）即在应用编程，IAP 是用户自己的程序在运行过程中对 User Flash 的部分区域进行烧写，目的是为了在产品发布后可以方便地通过预留的通信口对产品中的固件程序进行更新升级。 通常实现 IAP 功能时，即用户程序运行中作自身的更新操作，需要在设计固件程序时编写两个项目代码，第一个项目程序不执行正常的功能操作，而只是通过某种通信方式(如 USB、USART)接收程序或数据，执行对第二部分代码的更新；第二个项目代码才是真正的功能代码。这两部分项目代码都同时烧录在 User Flash 中，当芯片上电后，首先是第一个项目代码开始运行，它作如下操作：

1）检查是否需要对第二部分代码进行更新

2）如果不需要更新则转到 4）

3）执行更新操作

4）跳转到第二部分代码执行

第一部分代码必须通过其它手段，如 JTAG 或 ISP 烧入；第二部分代码可以使用第一部分代码 IAP 功能烧入，也可以和第一部分代码一起烧入，以后需要程序更新是再通过第一部分 IAP 代码更新。

我们将第一个项目代码称之为 Bootloader 程序，第二个项目代码称之为 APP 程序，他们存放在 STM32 FLASH 的不同地址范围，一般从最低地址区开始存放 Bootloader，紧跟其后的就是 APP 程序（注意，如果 FLASH 容量足够，是可以设计很多 APP 程序的，本章我们只讨论一个 APP 程序的情况）。这样我们就是要实现 2 个程序：Bootloader 和 APP。

STM32 的 APP 程序不仅可以放到 FLASH 里面运行，也可以放到 SRAM 里面运行，本章，我们将制作两个 APP，一个用于 FLASH 运行，一个用于 SRAM 运行。

我们先来看看 STM32 正常的程序运行流程，如图 53.1.1 所示：

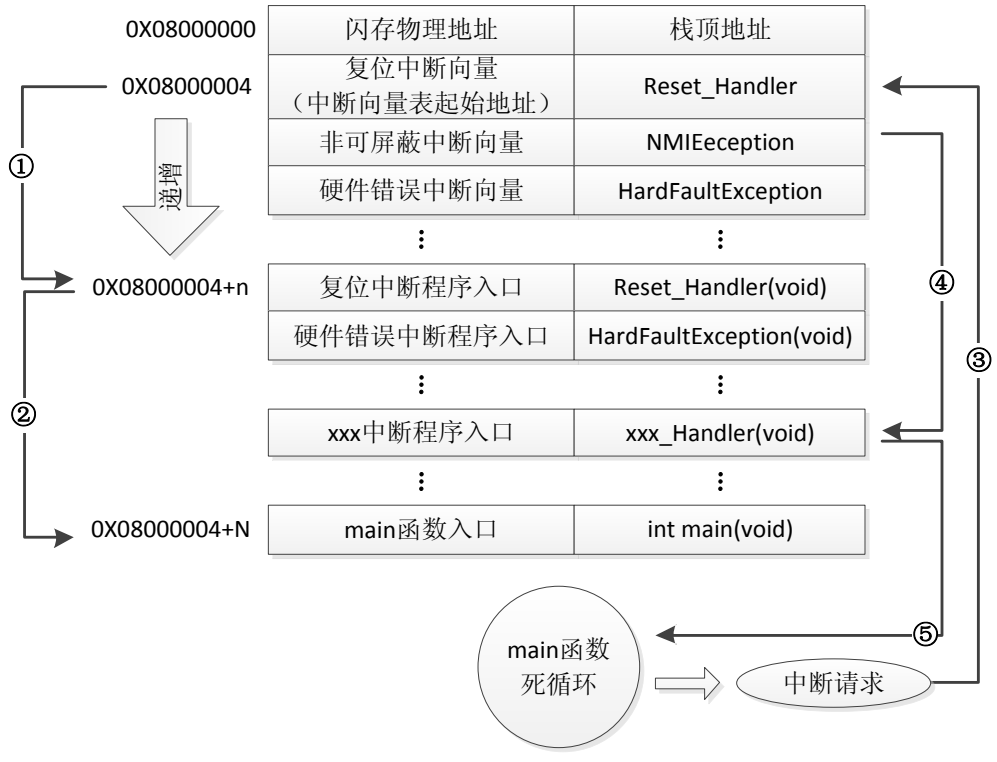


图 53.1.1 STM32 正常运行流程图

STM32 的内部闪存（FLASH）地址起始于 0x08000000，一般情况下，程序文件就从此地址开始写入。此外 STM32 是基于 Cortex-M3 内核的微控制器，其内部通过一张“中断向量表”来响应中断，程序启动后，将首先从“中断向量表”取出复位中断向量执行复位中断程序完成启动，而这张“中断向量表”的起始地址是 0x08000004，当中断来临，STM32 的内部硬件机制亦会自动将 PC 指针定位到“中断向量表”处，并根据中断源取出对应的中断向量执行中断服务程序。

在图 53.1.1 中，STM32 在复位后，先从 0X08000004 地址取出复位中断向量的地址，并跳转到复位中断服务程序，如图标号①所示；在复位中断服务程序执行完之后，会跳转到我们的 main 函数，如图标号②所示；而我们的 main 函数一般都是一个死循环，在 main 函数执行过程中，如果收到中断请求（发生重中断），此时 STM32 强制将 PC 指针指回中断向量表处，如图标号③所示；然后，根据中断源进入相应的中断服务程序，如图标号④所示；在执行完中断服务程序以后，程序再次返回 main 函数执行，如图标号⑤所示。

当加入 IAP 程序之后，程序运行流程如图 53.1.2 所示：

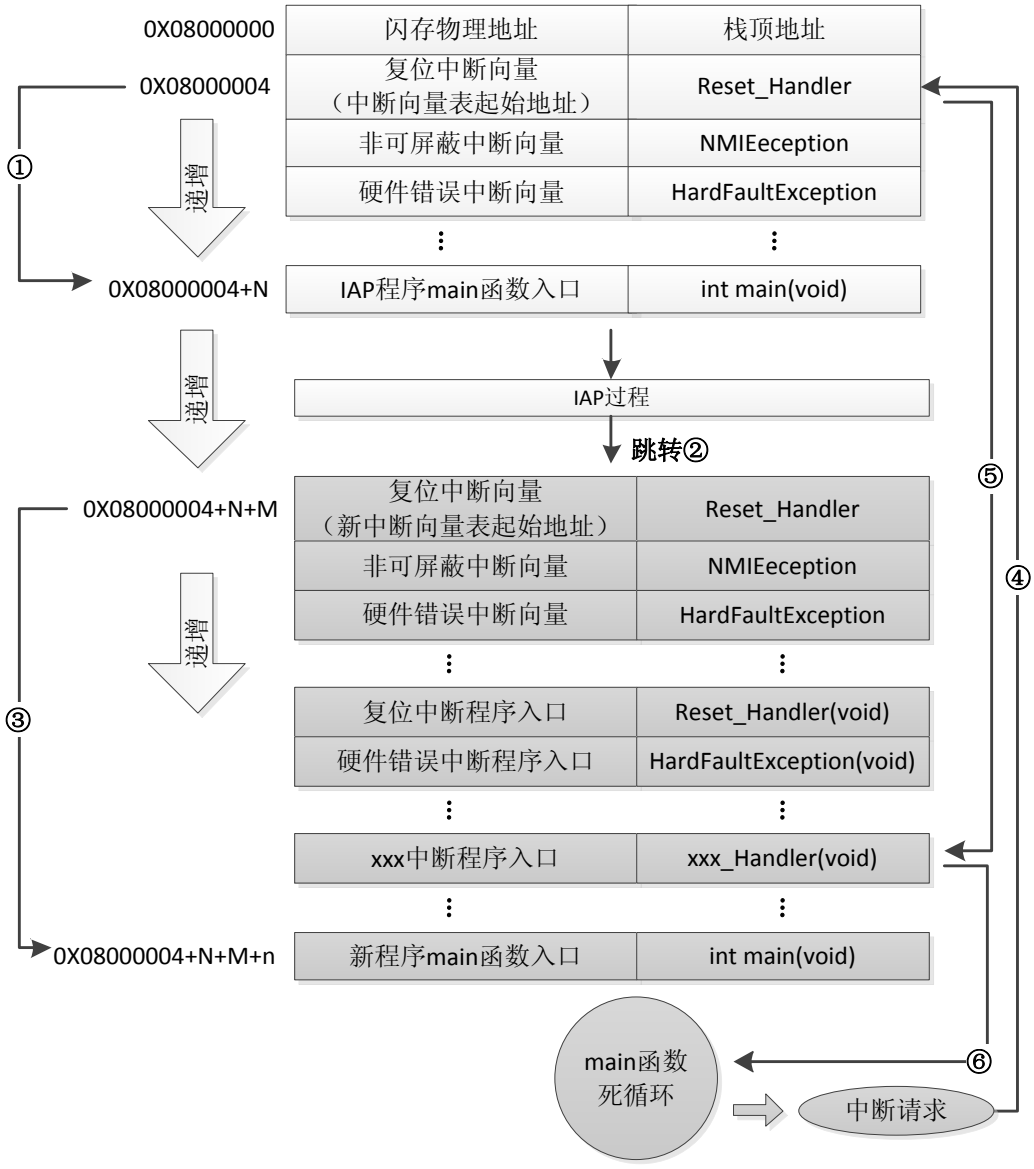


图 53.1.2 加入 IAP 之后程序运行流程图

在图 53.1.2 所示流程中，STM32 复位后，还是从 0X08000004 地址取出复位中断向量的地址，并跳转到复位中断服务程序，在运行完复位中断服务程序之后跳转到 IAP 的 main 函数，如图标号①所示，此部分同图 53.1.1 一样；在执行完 IAP 以后（即将新的 APP 代码写入 STM32 的 FLASH，灰底部分。新程序的复位中断向量起始地址为 0X08000004+N+M），跳转至新写入程序的复位向量表，取出新程序的复位中断向量的地址，并跳转执行新程序的复位中断服务程序，随后跳转至新程序的 main 函数，如图标号②和③所示，同样 main 函数为一个死循环，并且注意到此时 STM32 的 FLASH，在不同位置上，共有两个中断向量表。

在 main 函数执行过程中，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，而不是新程序的中断向量表，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；在执行完中断服务程序后，程序返回 main 函数继续运行，如图标号⑥所示。

通过以上两个过程的分析，我们知道 IAP 程序必须满足两个要求：

1） 新程序必须在 IAP 程序之后的某个偏移量为 x 的地址开始；

2） 必须将新程序的中断向量表相应的移动，移动的偏移量为 x；

本章，我们有 2 个 APP 程序，一个为 FLASH 的 APP，程序在 FLASH 中运行，另外一个位 SRAM 的 APP，程序运行在 SRAM 中，图 53.1.2 虽然是针对 FLASH APP 来说的，但是在 SRAM 里面运行的过程和 FLASH 基本一致，只是需要设置向量表的地址为 SRAM 的地址。

1.APP 程序起始地址设置方法

随便打开一个之前的实例工程，点击 Options for Target→Target 选项卡，如图 53.1.3 所示：

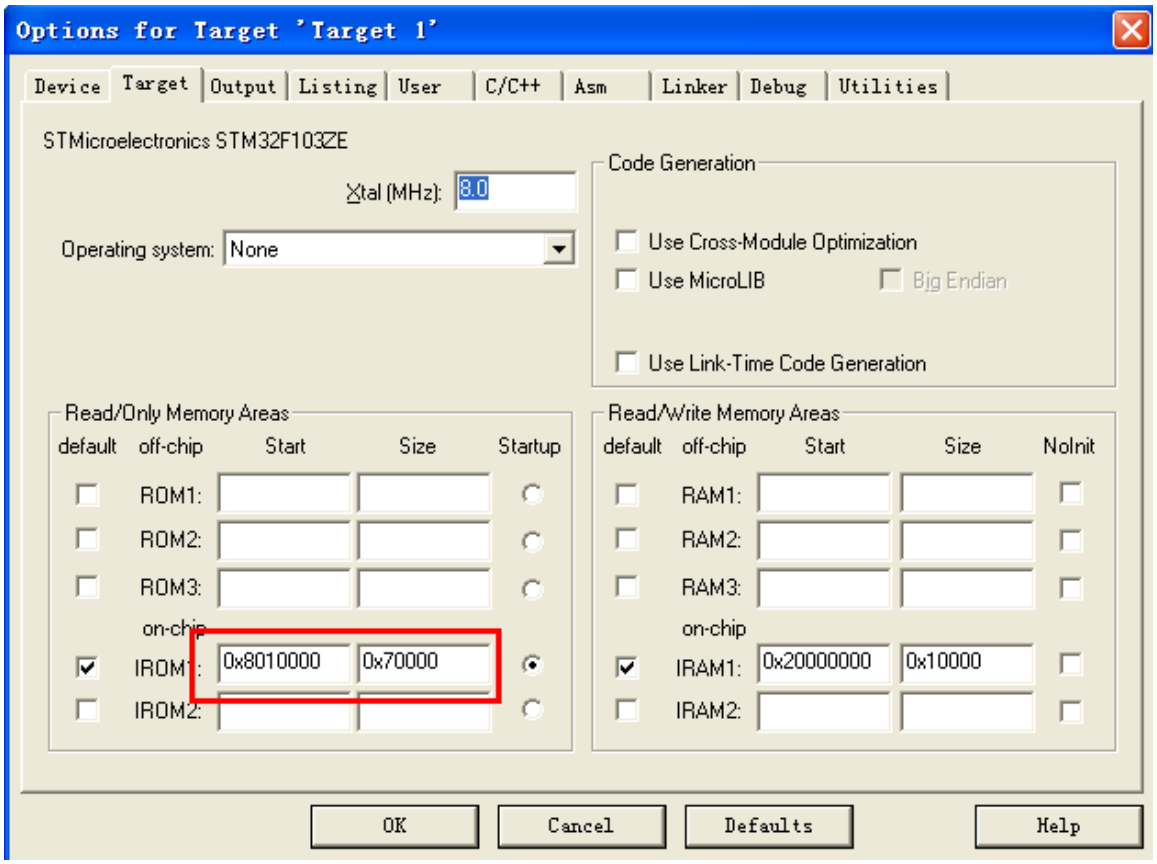


图 53.1.3 FLASH APP Target 选项卡设置

默认的条件下，图中 IROM1 的起始地址（Start）一般为 0X08000000，大小（Size）为 0X80000，即从 0X08000000 开始的 512K 空间为我们的程序存储（因为我们的 STM32F103ZET6 的 FLASH

大小是 512K）。而图中，我们设置起始地址（Start）为 0X08010000，即偏移量为 0X10000（64K 字节），因而，留给 APP 用的 FLASH 空间（Size）只有 0X80000-0X10000=0X70000（448K 字节）大小了。设置好 Start 和 Szie，就完成 APP 程序的起始地址设置。

这里的 64K 字节，需要大家根据 Bootloader 程序大小进行选择，比如我们本章的 Bootloader 程序为 22K 左右，理论上我们只需要确保 APP 起始地址在 Bootloader 之后，并且偏移量为 0X200 的倍数即可（相关知识，请参考：http://www.openedv.com/posts/list/392.htm）。这里我们选择 64K（0X10000）字节，留了一些余量，方便 Bootloader 以后的升级修改。

这是针对 FLASH APP 的起始地址设置，如果是 SRAM APP，那么起始地址设置如图 53.1.4 所示：

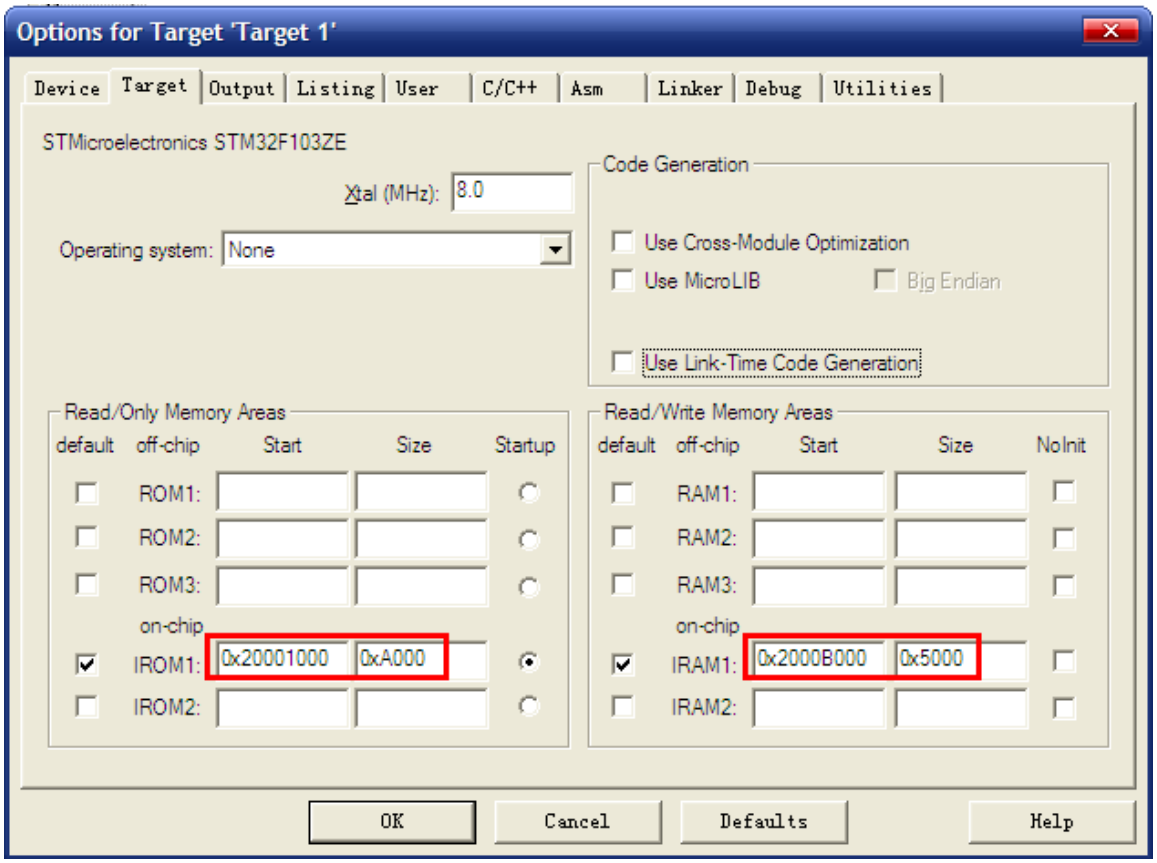


图 53.1.4 SRAM APP Target 选项卡设置

这里我们将 IROM1 的起始地址（Start）定义为：0X20001000，大小为 0XA000（40K 字节），即从地址 0X20000000 偏移 0X1000 开始，存放 APP 代码。因为整个 STM32F103ZET6 的 SRAM 大小为 64K 字节，所以 IRAM1（SRAM）的起始地址变为 0X2000B000（0x20001000+0xA000=0X2000B000），大小只有 0X5000（20K 字节）。这样，整个 STM32F103ZET6 的 SRAM 分配情况为：最开始的 4K 给 Bootloader 程序使用，随后的 40K 存放 APP 程序，最后 20K，用作 APP 程序的内存。这个分配关系大家可以根据自己的实际情况修改，不一定和我们这里的设置一模一样，不过也需要注意，保证偏移量为 0X200 的倍数（我们这里为 0X1000）。

2.中断向量表的偏移量设置方法

之前我们讲解过，在系统启动的时候，会首先调用 systemInit 函数初始化时钟系统，同时 systemInit 还完成了中断向量表的设置，我们可以打开 systemInit 函数，看看函数体的结尾处有这样几行代码：

```
#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
```

从代码可以理解，VTOR 寄存器存放的是中断向量表的起始地址。默认的情况 VECT_TAB_SRAM 是没有定义,所以执行 SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; 对于 FLASH APP，我们设置为 FLASH_BASE+偏移量 0x10000,所以我们可以在 FLASH APP 的 main 函数最开头处添加如下代码实现中断向量表的起始地址的重设：

```
    SCB->VTOR = FLASH_BASE | 0x10000;
```

以上是 FLASH APP 的情况，当使用 SRAM APP 的时候，我们设置起始地址为：SRAM_bASE+0x1000,同样的方法，我们在 SRAM APP 的 main 函数最开始处，添加下面代码：

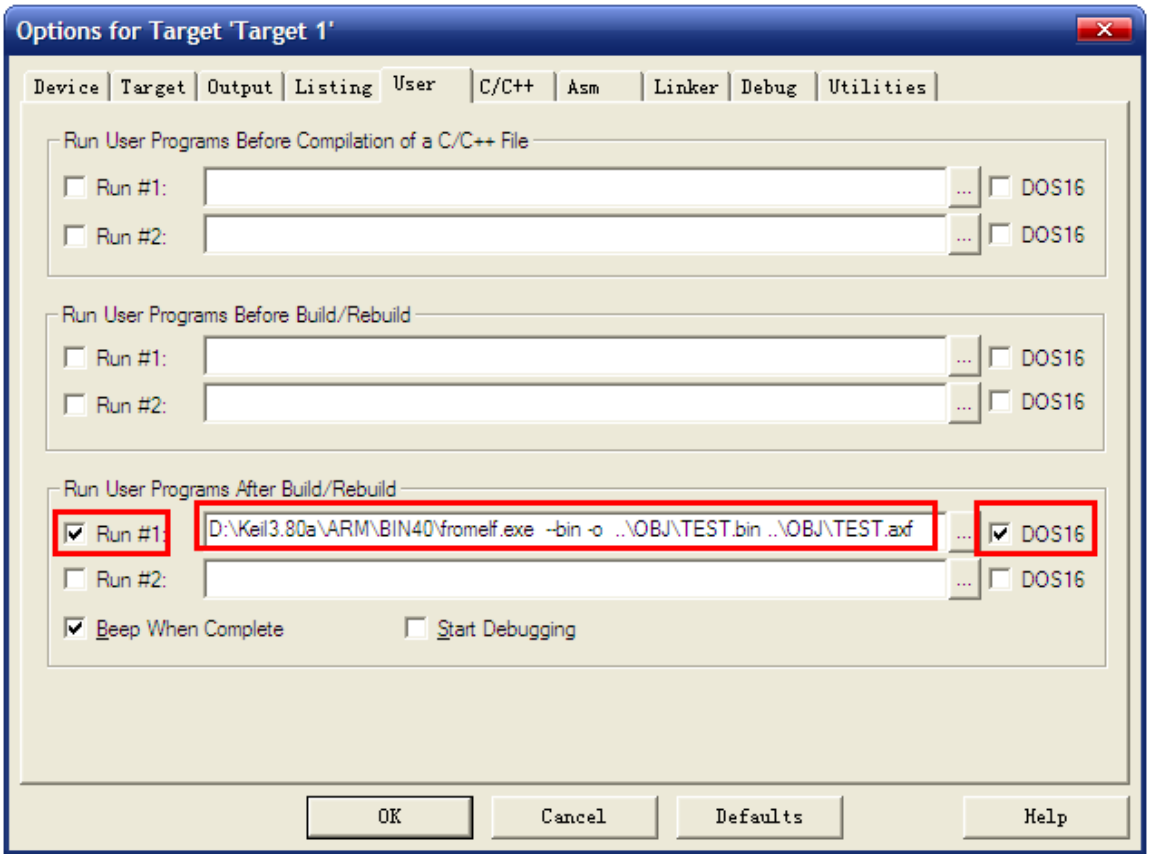
```
    SCB->VTOR = SRAM_BASE | 0x1000;
```

这样，我们就完成了中断向量表偏移量的设置。

通过以上两个步骤的设置，我们就可以生成 APP 程序了，只要 APP 程序的 FLASH 和 SRAM 大小不超过我们的设置即可。不过 MDK 默认生成的文件是.hex 文件，并不方便我们用作 IAP 更新，我们希望生成的文件是.bin 文件，这样可以方便进行 IAP 升级（至于为什么，请大家自行百度 HEX 和 BIN 文件的区别！）。这里我们通过 MDK 自带的格式转换工具 fromelf.exe，来实现.axf 文件到.bin 文件的转换。该工具在 MDK 的安装目录\ARM\BIN40 文件夹里面。

fromelf.exe 转换工具的语法格式为：fromelf [options] input_file。其中 options 有很多选项可以设置，详细使用请参考光盘《mdk 如何生成 bin 文件.pdf》.

本章，我们通过在 MDK 点击 Options for Target→User 选项卡，在 Run User Programs After Build/Rebuild 栏，勾选 Run#1 和 DOS16，并写入：D:\Keil3.80a\ARM\BIN40\fromelf.exe --bin -o ..\OBJ\TEST.bin ..\OBJ\TEST.axf ，如图 53.1.6 所示：



通过这一步设置，我们就可以在 MDK 编译成功之后，调用 fromelf.exe（注意，我的 MDK 是安装在 D:\Keil3.80A 文件夹下，如果你是安装在其他目录，请根据你自己的目录修改 fromelf.exe 的路径），根据当前工程的 TEST.axf（如果是其他的名字，请记住修改，这个文件存放在 OBJ 目录下面，格式为 xxx.axf），生成一个 TEST.bin 的文件。并存放在 axf 文件相同的目录下，即工程的 OBJ 文件夹里面。在得到.bin 文件之后，我们只需要将这个 bin 文件传送给单片机，即可执行 IAP 升级。

最后再来 APP 程序的生成步骤：

1） 设置 APP 程序的起始地址和存储空间大小

对于在 FLASH 里面运行的 APP 程序，我们可以按照图 53.1.3 的设置。对于 SRAM 里面运行的 APP 程序，我们可以参考图 53.1.4 的设置。

2） 设置中断向量表偏移量

这一步按照上面讲解，重新设置 SCB->VTOR 的值即可。

3） 设置编译后运行 fromelf.exe，生成.bin 文件.

通过在 User 选项卡，设置编译后调用 fromelf.exe，根据.axf 文件生成.bin 文件，用于 IAP 更新。

以上 3 个步骤，我们就可以得到一个.bin 的 APP 程序，通过 Bootlader 程序即可实现更新。大家可以打开我们光盘的两个 APP 工程，熟悉这些设置。

53.2 硬件设计

本章实验（Bootloader 部分）功能简介：开机的时候先显示提示信息，然后等待串口输入接收 APP 程序（无校验，一次性接收），在串口接收到 APP 程序之后，即可执行 IAP。如果

是 SRAM APP，通过按下 KEY0 即可执行这个收到的 SRAM APP 程序。如果是 FLASH APP，则需要先按下 WK_UP 按键，将串口接收到的 APP 程序存放到 STM32 的 FLASH，之后再按 KEY2 既可以执行这个 FLASH APP 程序。通过 KEY1 按键，可以手动清除串口接收到的 APP 程序。DS0 用于指示程序运行状态。

本实验用到的资源如下：

1） 指示灯 DS0

2） 四个按键（KEY0/KEY1/KEY2/WK_UP）

3） 串口

4） TFTLCD 模块

这些用到的硬件，我们在之前都已经介绍过，这里就不再介绍了。

53.3 软件设计

本章，我们总共需要 3 个程序：1，Bootloader；2，FLASH APP；3）SRAM APP；其中，我们选择之前做过的 RTC 实验（在第二十章介绍）来做为 FLASH APP 程序（起始地址为 0X08010000），选择触摸屏实验（在第三十一章介绍）来做 SRAM APP 程序（起始地址为 0X20001000）。Bootloader 则是通过 TFTLCD 显示实验（在第十八章介绍）修改得来。本章，关于 SRAM APP 和 FLASH APP 的生成比较简单，我们就不细说，请大家结合光盘源码，以及 53.1 节的介绍，自行理解。本章软件设计仅针对 Bootloader 程序。

打开本实验工程，可以看到我们增加了 IAP 组，在组下面添加了 iap.c 文件以及其头文件 isp.h。

打开 iap.c， 代码如下：

```
#include "sys.h"
#include "delay.h"
#include "usart.h"
#include "stmflash.h"
#include "iap.h"
iapfun jump2app;
u16 iapbuf[1024];
//appxaddr:应用程序的起始地址
//appbuf:应用程序 CODE.
//appsize:应用程序大小(字节).
void iap_write_appbin(u32 appxaddr,u8 *appbuf,u32 appsize)
{
    u16 t;
    u16 i=0;
    u16 temp;
    u32 fwaddr=appxaddr;//当前写入的地址
    u8 *dfu=appbuf;
    for(t=0;t<appsize;t+=2)
    {
        temp=(u16)dfu[1]<<8;
        temp+=(u16)dfu[0];
```

```
            dfu+=2;//偏移 2 个字节
            iapbuf[i++]=temp;
            if(i==1024)
            {
                i=0;
                STMFLASH_Write(fwaddr,iapbuf,1024);
                fwaddr+=2048;//偏移 2048  16=2*8.所以要乘以 2.
            }
        }
        if(i)STMFLASH_Write(fwaddr,iapbuf,i);//将最后的一些内容字节写进去.
    }
    //跳转到应用程序段
    //appxaddr:用户代码起始地址.
    void iap_load_app(u32 appxaddr)
    {
        if(((*(vu32*)appxaddr)&0x2FFE0000)==0x20000000)     //检查栈顶地址是否合法.
        {
            jump2app=(iapfun)*(vu32*)(appxaddr+4);
            //用户代码区第二个字为程序开始地址(复位地址)
            MSR_MSP(*(vu32*)appxaddr);
            //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)
            jump2app();     //跳转到 APP.
        }
    }
```

该文件总共只有 2 个函数，其中，iap_write_appbin 函数用于将存放在串口接收 buf 里面的 APP 程序写入到 FLASH。iap_load_app 函数，则用于跳转到 APP 程序运行，其参数 appxaddr 为 APP 程序的起始地址，程序先判断栈顶地址是否合法，在得到合法的栈顶地址后，通过 MSR_MSP 函数（该函数在 sys.c 文件）设置栈顶地址，最后通过一个虚拟的函数（jump2app）跳转到 APP 程序执行代码，实现 IAP→APP 的跳转。

打开 iap.h 代码如下：

```
#ifndef __IAP_H__
#define __IAP_H__
#include "sys.h"
typedef    void (*iapfun)(void);     //定义一个函数类型的参数.
#define FLASH_APP1_ADDR            0x080010000
//第一个应用程序起始地址(存放在 FLASH)
//保留 0X08000000~0X0800FFFF 的空间为 Bootloader 使用
void iap_load_app(u32 appxaddr);                                    //跳转到 APP 程序执行
void iap_write_appbin(u32 appxaddr,u8 *appbuf,u32 applen);    //在指定地址开始,写入 bin
#endif
```

这部分代码比较简单，。本章，我们是通过串口接收 APP 程序的，我们将 usart.c 和 usart.h 做了稍微修改，在 usart.h 中，我们定义 USART_REC_LEN 为 55K 字节，也就是串口最大一次可以接收 55K 字节的数据，这也是本 Bootloader 程序所能接收的最大 APP 程序大小。然后新增

一个 USART_RX_CNT 的变量，用于记录接收到的文件大小，而 USART_RX_STA 不再使用。打开 usart.c，可以看到我们修改 USART1_IRQHandler 部分代码如下：

```
//串口 1 中断服务程序
//注意,读取 USARTx->SR 能避免莫名其妙的错误
u8 USART_RX_BUF[USART_REC_LEN] __attribute__ ((at(0X20001000)));
//接收缓冲,最大 USART_REC_LEN 个字节,起始地址为 0X20001000.
//接收状态
//bit15， 接收完成标志
//bit14， 接收到 0x0d
//bit13~0， 接收到的有效字节数目
u16 USART_RX_STA=0;                 //接收状态标记
u16 USART_RX_CNT=0;                 //接收的字节数
void USART1_IRQHandler(void)
{
    u8 res;
#ifdef OS_CRITICAL_METHOD
//如果 OS_CRITICAL_METHOD 定义了,说明使用 ucosII 了.
    OSIntEnter();
#endif
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)//接收到数据
    {
        res=USART_ReceiveData(USART1);
        if(USART_RX_CNT<USART_REC_LEN)
        {
            USART_RX_BUF[USART_RX_CNT]=res;
            USART_RX_CNT++;
        }
    }
#ifdef OS_CRITICAL_METHOD
//如果 OS_CRITICAL_METHOD 定义了,说明使用 ucosII 了.
    OSIntExit();
#endif
}
```

这里，我们指定 USART_RX_BUF 的地址是从 0X20001000 开始，该地址也就是 SRAM APP 程序的起始地址！然后在 USART1_IRQHandler 函数里面，将串口发送过来的数据，全部接收到 USART_RX_BUF，并通过 USART_RX_CNT 计数。代码比较简单，我们就不多说了。

最后我们看看 main 函数如下：

```
int main(void)
{
    u8 t;
    u8 key;
    u16 oldcount=0;    //老的串口接收数据值
    u16 applenth=0;    //接收到的 app 代码长度
```

```
u8 clearflag=0;
uart_init(256000);  //串口初始化为 256000
delay_init();               //延时初始化
LCD_Init();
LED_Init();                 //初始化与 LED 连接的硬件接口
KEY_Init();                 //按键初始化
POINT_COLOR=RED;//设置字体为红色
LCD_ShowString(60,50,200,16,16,"Warship STM32");
LCD_ShowString(60,70,200,16,16,"IAP TEST");
LCD_ShowString(60,90,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(60,110,200,16,16,"2012/9/24");
LCD_ShowString(60,130,200,16,16,"WK_UP:Copy APP2FLASH");
LCD_ShowString(60,150,200,16,16,"KEY1:Erase SRAM APP");
LCD_ShowString(60,170,200,16,16,"KEY0:Run SRAM APP");
LCD_ShowString(60,190,200,16,16,"KEY2:Run FLASH APP");
POINT_COLOR=BLUE;
//显示提示信息
POINT_COLOR=BLUE;//设置字体为蓝色
while(1)
{
    if(USART_RX_CNT)
    {
        if(oldcount==USART_RX_CNT)
        //新周期内,没有收到任何数据,认为本次数据接收完成.
        {
            applenth=USART_RX_CNT;
            oldcount=0;
            USART_RX_CNT=0;
            printf("用户程序接收完成!\r\n");
            printf("代码长度:%dBytes\r\n",applenth);
        }else oldcount=USART_RX_CNT;
    }
    t++; delay_ms(10);
    if(t==30)
    {
        LED0=!LED0; t=0;
        if(clearflag)
        {
            clearflag--;
            if(clearflag==0)LCD_Fill(60,210,240,210+16,WHITE);//清除显示
        }
    }
    key=KEY_Scan(0);
```

```
if(key==KEY_UP)
{
    if(applenth)
    {
        printf("开始更新固件...\r\n");
        LCD_ShowString(60,210,200,16,16,"Copying APP2FLASH...");
        if(((*(vu32*)(0X20001000+4))&0xFF000000)==0x08000000)
        //判断是否为 0X08XXXXXX.
        {
            iap_write_appbin(FLASH_APP1_ADDR,USART_RX_BUF,
            applenth); //更新 FLASH 代码
            LCD_ShowString(60,210,200,16,16,"Copy APP Successed!!");
            printf("固件更新完成!\r\n");
        }else
        {
            LCD_ShowString(60,210,200,16,16,"Illegal FLASH APP!    ");
            printf("非 FLASH 应用程序!\r\n");
        }
    }else
    {
        printf("没有可以更新的固件!\r\n");
        LCD_ShowString(60,210,200,16,16,"No APP!");
    }
    clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示
}
if(key==KEY_DOWN)
{
    if(applenth)
    {

        printf("固件清除完成!\r\n");
        LCD_ShowString(60,210,200,16,16,"APP Erase Successed!");
        applenth=0;
    }else
    {
        printf("没有可以清除的固件!\r\n");
        LCD_ShowString(60,210,200,16,16,"No APP!");
    }
    clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示

}
if(key==KEY_LEFT)
{
```

```
                printf("开始执行 FLASH 用户代码!!\r\n");
                if(((*(vu32*)(FLASH_APP1_ADDR+4))&0xFF000000)==0x08000000)
                //判断是否为 0X08XXXXXX.
                {
                    iap_load_app(FLASH_APP1_ADDR);//执行 FLASH APP 代码
                }else
                {
                    printf("非 FLASH 应用程序,无法执行!\r\n");
                    LCD_ShowString(60,210,200,16,16,"Illegal FLASH APP!");
                }
                clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示
            }
            if(key==KEY_RIGHT)
            {
                printf("开始执行 SRAM 用户代码!!\r\n");
                if(((*(vu32*)(0X20001000+4))&0xFF000000)==0x20000000)
                //判断是否为 0X20XXXXXX.
                {
                    iap_load_app(0X20001000);//SRAM 地址
                }else
                {
                    printf("非 SRAM 应用程序,无法执行!\r\n");
                    LCD_ShowString(60,210,200,16,16,"Illegal SRAM APP!");
                }
                clearflag=7;//标志更新了显示,并且设置 7*300ms 后清除显示
            }

        }
    }
```

该段代码，实现了串口数据处理，以及 IAP 更新和跳转等各项操作。Bootloader 程序就设计完成了，但是一般要求 bootloader 程序越小越好（给 APP 省空间嘛），所以，本章我们把一些不需要用到的.c 文件全部去掉，最后得到工程截图如图 53.3.1 所示：

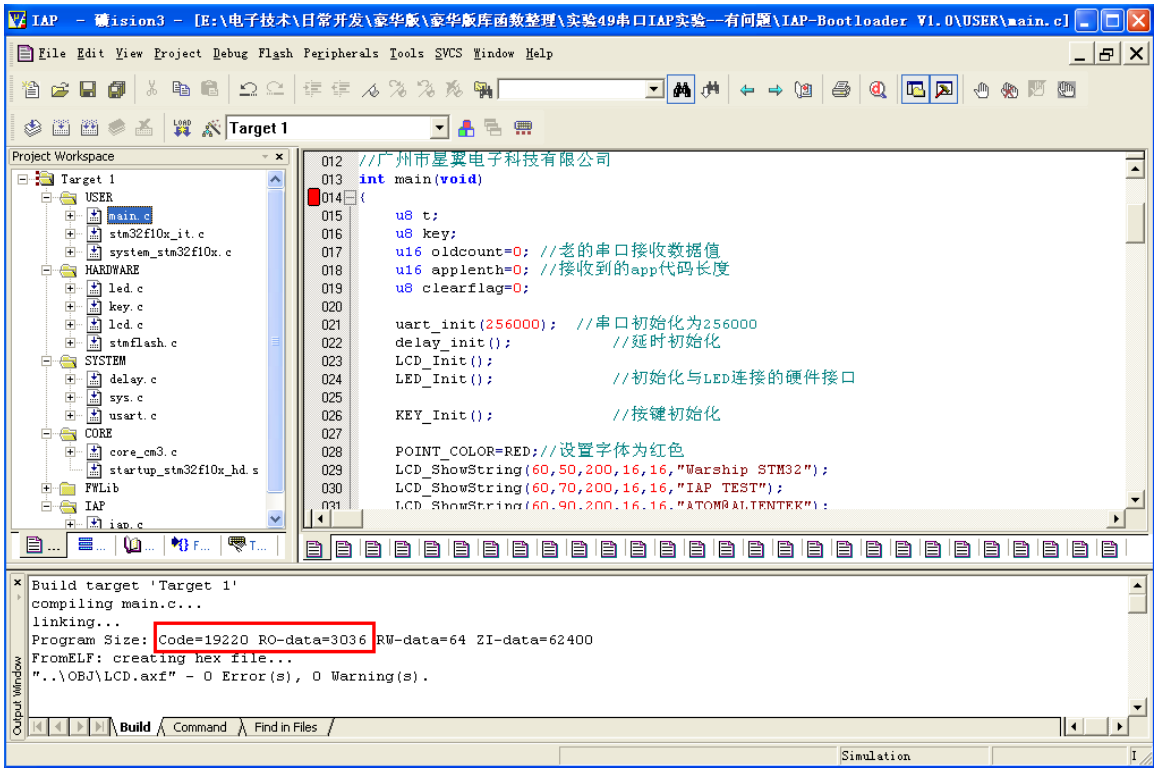


图 53.3.1 Bootloader 工程截图

从上图可以看出，虽然去掉了一些不用的.c 文件，但是 Bootloader 大小还是有 22K 左右，比较大，主要原因是液晶驱动和 printf 占用了比较多的 flash，如果大家想进一步删减，可以去掉 LCD 显示和 printf 等，不过我们在本章为了演示效果，所以保留了这些代码。

至此，本实验的软件设计部分结束。

FLASH APP 和 SRAM APP 两部分代码，我们在实验目录下提供了两个实验供大家参考，不过要提醒大家，根据我们的设置，FLASH APP 的起始地址必须是 0X08010000，而 SRAM APP 的起始地址必须是 0X20001000。

53.4 下载验证

在代码编译成功之后，我们下载代码到 ALIENTEK 战舰 STM32 开发板上，得到，如图 53.4.1 所示：

图 53.4.1 IAP 程序界面

此时，我们可以通过串口，发送 FLASH APP 或者 SRAM APP 到战舰 STM32 开发板，如图 53.4.2 所示：

图 53.4.2 串口发送 APP 程序界面

先用串口调试助手的打开文件按钮（如图标号 1 所示），找到 APP 程序生成的.bin 文件，然后设置波特率为 256000（为了提高速度，Bootloader 程序将波特率被设置为 256000 了），最后点击发送文件（图中标号 3 所示），将.bin 文件发送给战舰 STM32 开发板。

在收到 APP 程序之后，我们就可以通过 KEY0/KEY2 运行这个 APP 程序了（如果是 FLASH APP，则先需要通过 WK_UP 将其存入对应 FLASH 区域）。