STM32F103 实现 IAP 在线升级应用程序

作者：DS小龙哥

2022-08-09 重庆

本文字数：4268 字 阅读完需约 17 分钟

一、环境介绍

MCU: STM32F103ZET6

编程 IDE: Keil5.25

二、 IAP 介绍

IAP，全称是“In-Application Programming”，中文解释为“在程序中编程”。IAP 是一种对通过微控制器的对外接口（如 USART，IIC，CAN，USB，以太网接口甚至是无线射频通道）对正在运行程序的微控制器进行内部程序的更新的技术（注意这完全有别于 ICP 或者 ISP 技术）。

ICP（In-Circuit Programming）技术即通过在线仿真器对单片机进行程序烧写，而 ISP 技术则是通过单片机内置的 bootloader 程序引导的烧写技术。无论是 ICP 技术还是 ISP 技术，都需要有机械性的操作如连接下载线，设置跳线帽等。若产品的电路板已经层层密封在外壳中，要对其进行程序更新无疑困难重重，若产品安装于狭窄空间等难以触及的地方，更是一场灾难。但若进引入了 IAP 技术，则完全可以避免上述尴尬情况，而且若使用远距离或无线的数据传输方案，甚至可以实现远程编程和无线编程。这绝对是 ICP 或 ISP 技术无法做到的。

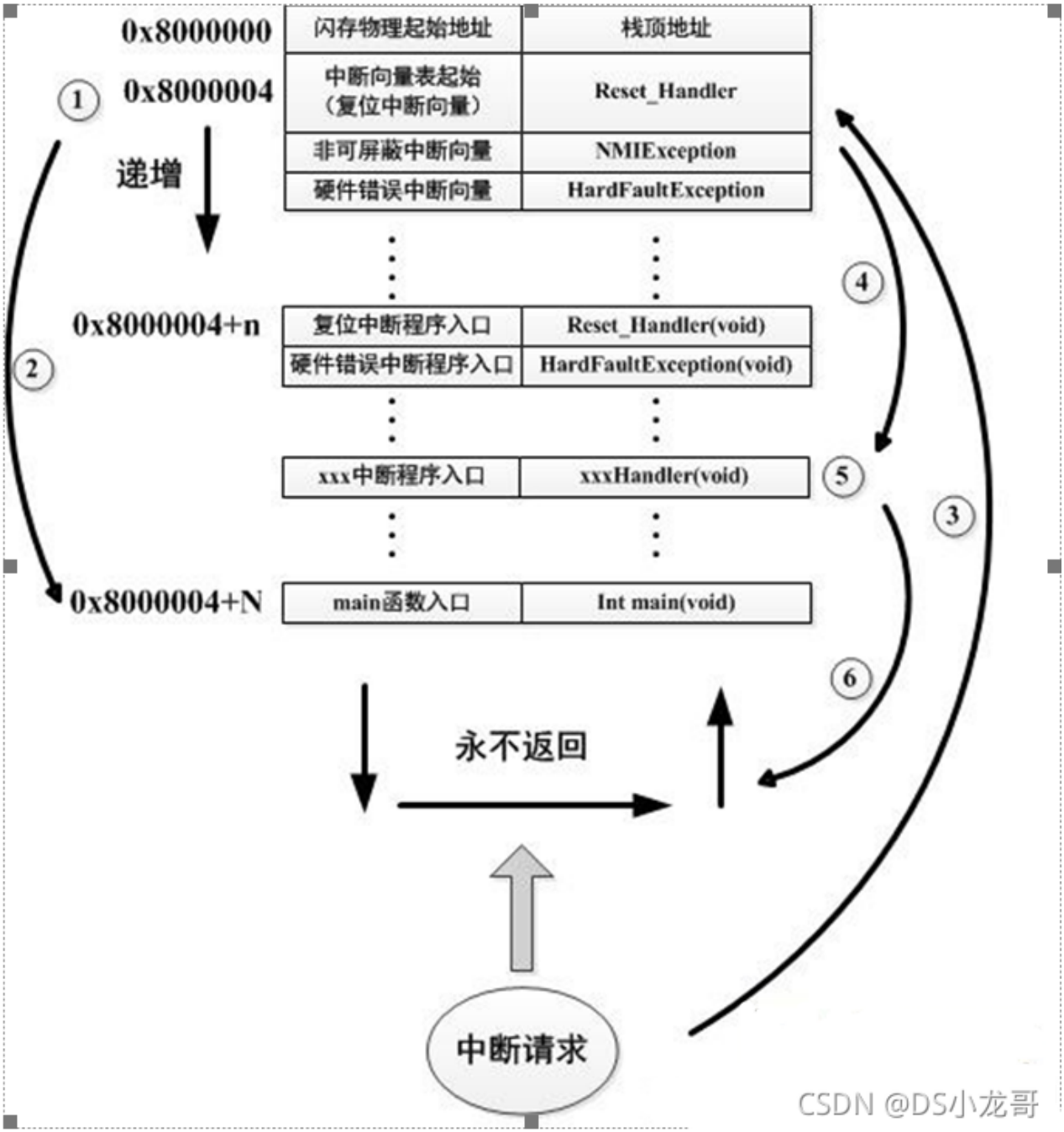
某种微控制器支持 IAP 技术的首要前提是其必须是基于可重复编程闪存的微控制器。STM32 微控制器带有可编程的内置闪存，同时 STM32 拥有在数量上和种类上都非常丰富的外设通信接口，因此在 STM32 上实现 IAP 技术是完全可行的。

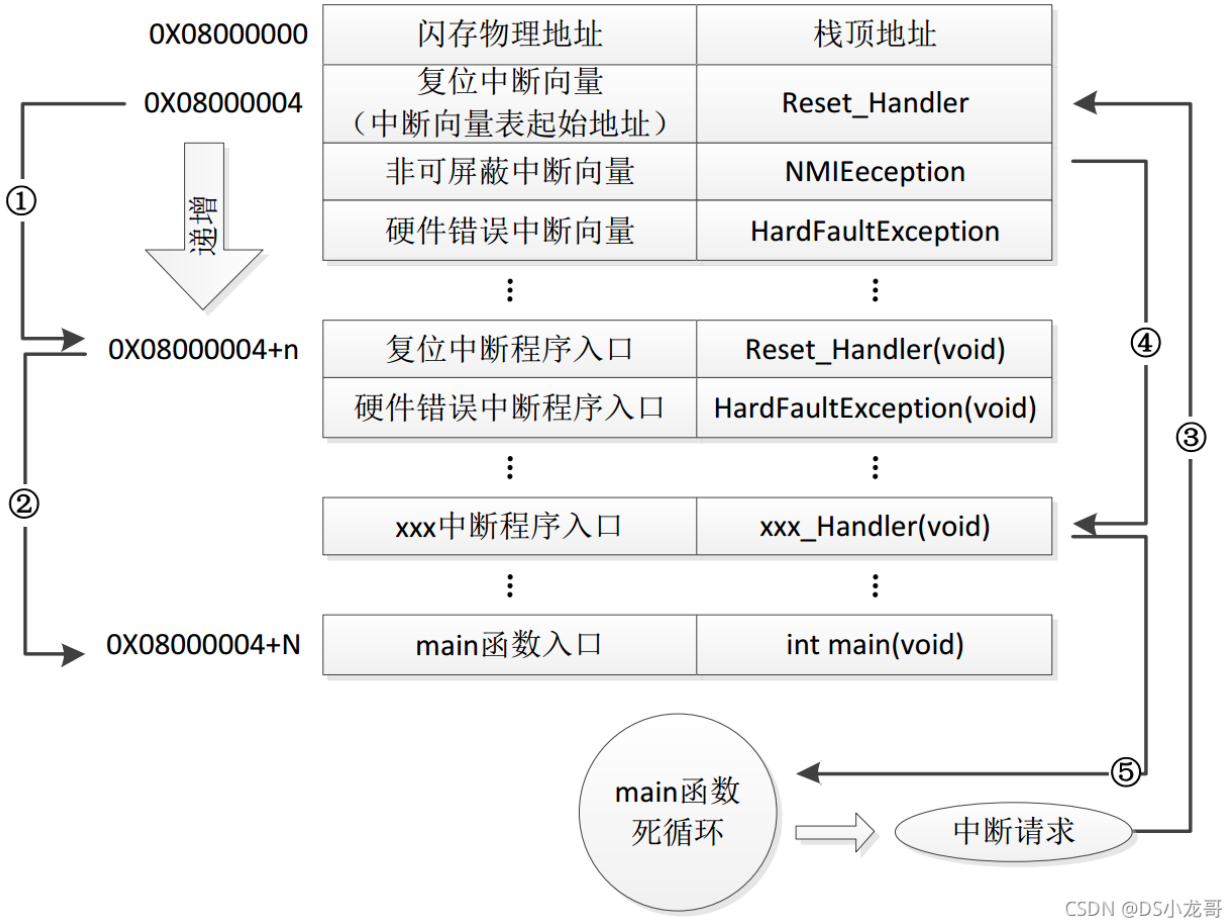
实现 IAP 技术的核心是一段预先烧写在单片机内部的 IAP 程序。这段程序主要负责与外部的上位机软件进行握手同步，然后将通过外设通信接口将来自于上位机软件的程序数据接收后写入单片机内部指定的闪存区域，然后再跳转执行新写入的程序，最终就达到了程序更新的目的。

在 STM32 微控制器上实现 IAP 程序之前首先要回顾一下 STM32 的内部闪存组织架构和其启动过程。STM32 的内部闪存地址起始于 0x8000000，一般情况下，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 0x8000004，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。最后还需要知道关键的一点，通过修改 STM32 工程的链接脚本可以修改程序文件写入闪存的起始地址。
>
> 在 STM32 微控制器上实现 IAP 方案，除了常规的串口接收数据以及闪存数据写入等常规操作外，还需注意 STM32 的启动过程和中断响应方式。

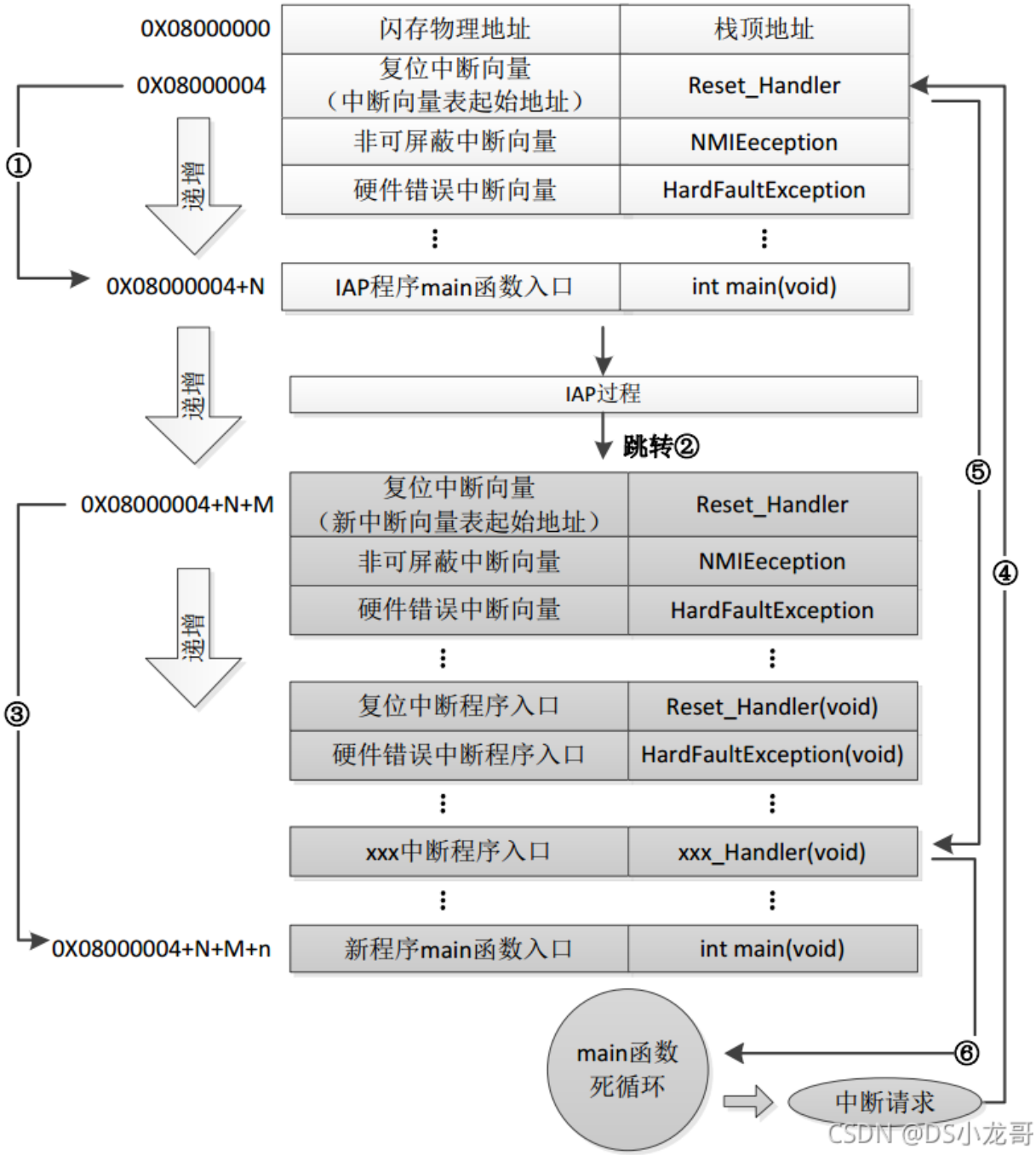
下图显示了 STM32 常规的运行流程：





> **图解读如下：** 1、 STM32 复位后，会从地址为 0x8000004 处取出复位中断向量的地址，并跳转执行复位中断服务程序。 2、 复位中断服务程序执行的最终结果是跳转至 C 程序的 main 函数，而 main 函数应该是一个死循环，是一个永不返回的函数。 3、 在 main 函数执行的过程中，发生了一个中断请求，此时 STM32 的硬件机制会将 PC 指针强制指回中断向量表处。 4、 根据中断源进入相应的中断服务程序。 5、 中断服务程序执行完毕后，程序再度返回至 main 函数中执行。

若在 STM32 中加入了 IAP 程序:



1、 STM32 复位后，从地址为 0x8000004 处取出复位中断向量的地址，并跳转执行复位中断服务程序，随后跳转至 IAP 程序的 main 函数。

2、 执行完 IAP 过程后（STM32 内部多出了新写入的程序，地址始于 0x8000004+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。新程序的 main 函数应该也具有永不返回的特性。同时应该注意在 STM32 的内部存储空间在不同的位置上出现了 2 个中断向量表。

3、 在新程序 main 函数执行的过程中，一个中断请求来临，PC 指针仍会回转至地址为 0x8000004 中断向量表处，而并不是新程序的中断向量表，注意到这是由 STM32 的硬件机制决定的。

4、 根据中断源跳转至对应的中断服务，注意此时是跳转至了新程序的中断服务程序中。

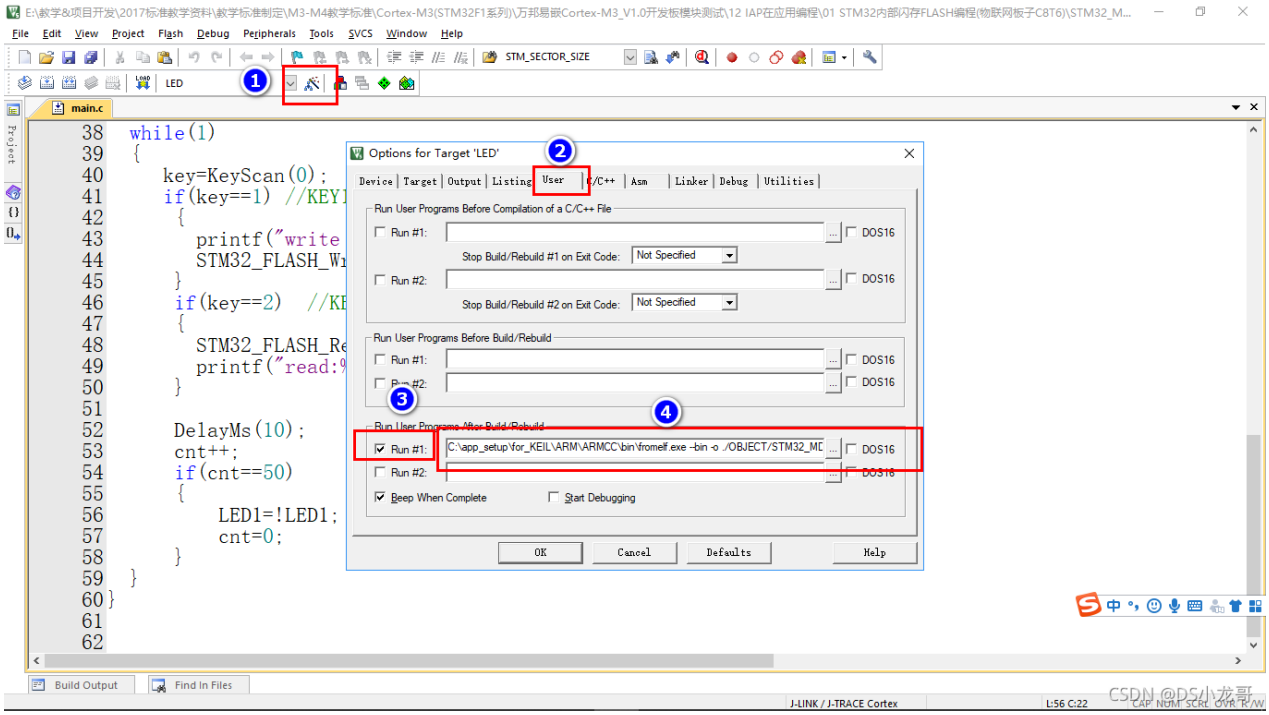
5、 中断服务执行完毕后，返回 main 函数。

二、hex 文件与 bin 文件区别

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。Intel HEX 文件经常被用于将程序或数据传输存储到 ROM、EPROM，大多数编程器和模拟器使用 Intel HEX 文件。 很多编译器的支持生成 HEX 格式的烧录文件，尤其是 Keil c。但是编程器能够下载的往往是 BIN 格式，因此 HEX 转 BIN 是每个编程器都必须支持的功能。HEX 格式文件以行为单位，每行由“：”（0x3a）开始，以回车键结束(0x0d,0x0a)。行内的数据都是由两个字符表示一个 16 进制字节，比如”01”就表示数 0x01；”0a”，就表示 0x0a。对于 16 位的地址，则高位在前低位在后，比如地址 0x010a，在 HEX 格式文件中就表示为字符串”010a”。

hex 和 bin 文件格式 Hex 文件，这里指的是 Intel 标准的十六进制文件，也就是机器代码的十六进制形式，并且是用一定文件格式的 ASCII 码来表示。具体格式介绍如下： Intel hex 文件常用来保存单片机或其他处理器的目标程序代码。它保存物理程序存储区中的目标代码映象。一般的编程器都支持这种格式。 hex 和 bin 文件格式 Hex 文件，这里指的是 Intel 标准的十六进制文件，也就是机器代码的十六进制形式，并且是用一定文件格式的 ASCII 码来表示。具体格式介绍如下： Intel hex 文件常用来保存单片机或其他处理器的目标程序代码。它保存物理程序存储区中的目标代码映象。一般的编程器都支持这种格式。

三、使用 Keil 软件完成 hex 文件转 bin 文件



选项框里的代码:

C:\app_setup\for_KEIL\ARM\ARMCC\bin\fromelf.exe --bin -o ./OBJECT/STM32_MD.bin ./OBJECT/STM32_MD.axf

解析如下:

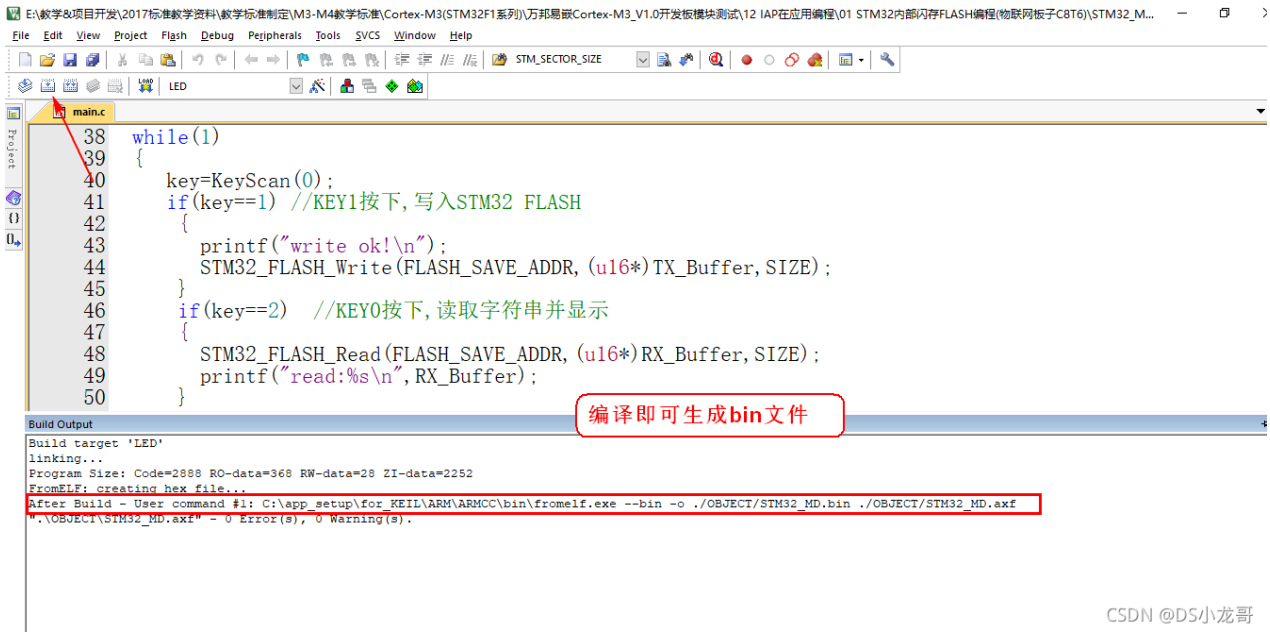
C:\app_setup\for_KEIL\ARM\ARMCC\bin\fromelf.exe:是 keil 软件安装目录下的一个工具,用于生成 bin

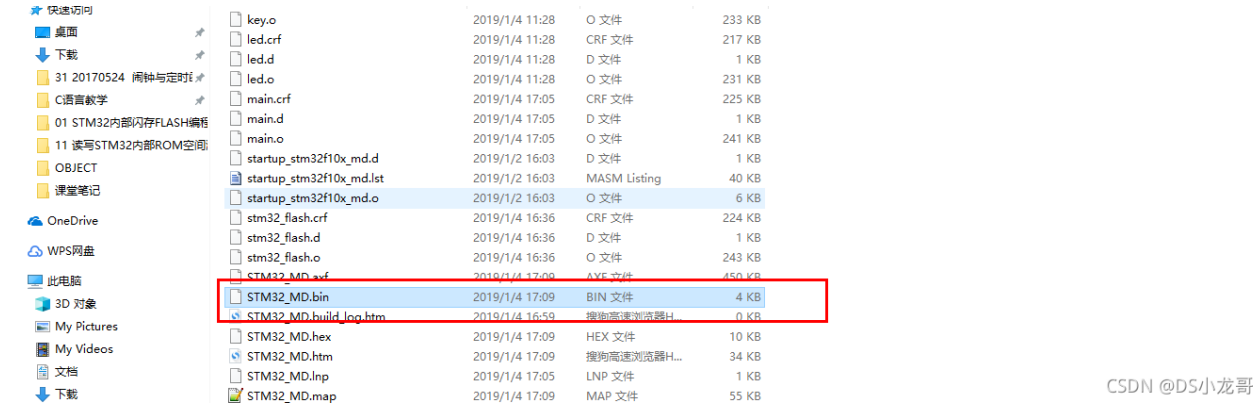
--bin -o ./OBJECT/STM32_MD.bin :指定生成 bin 文件的目录和名称

./OBJECT/STM32_MD.axf :指定输入的文件. 生成 hex 文件需要 axf 文件

新工程的编译指令:

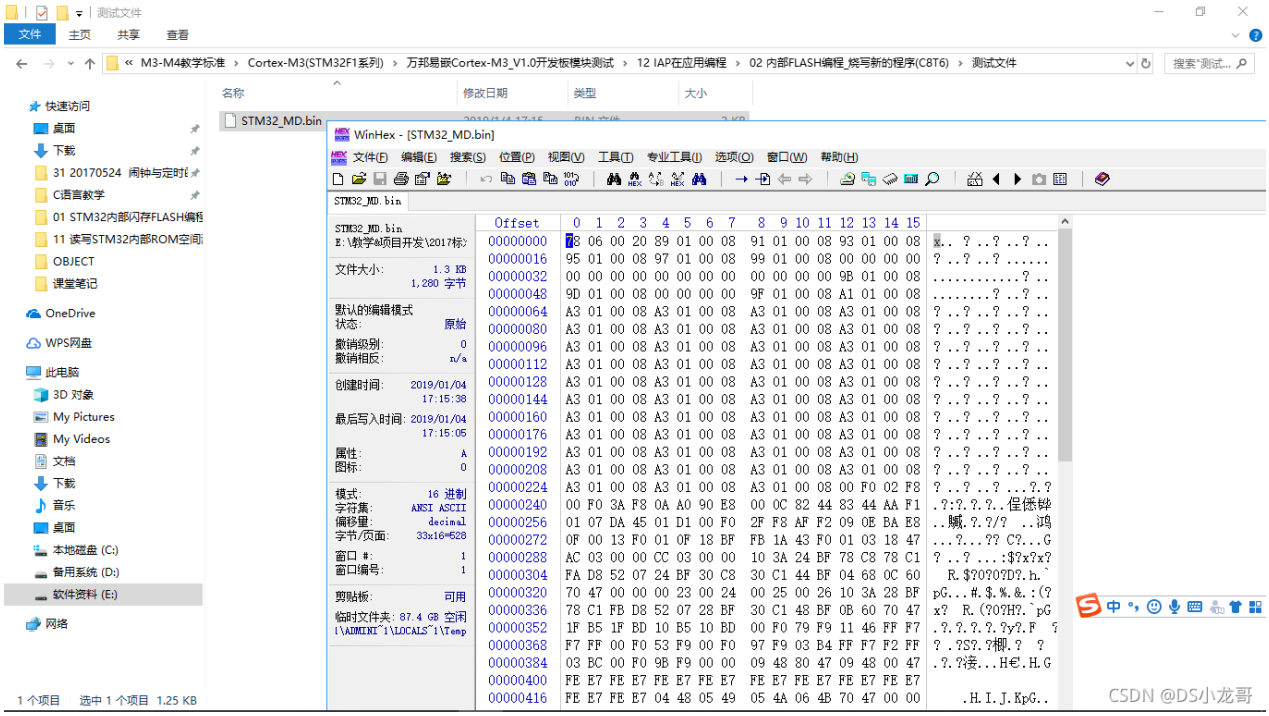
C:\Keil_v5\ARM\ARMCC\bin\fromelf.exe --bin -o ./obj/STM32HD.bin ./obj/STM32HD.axf

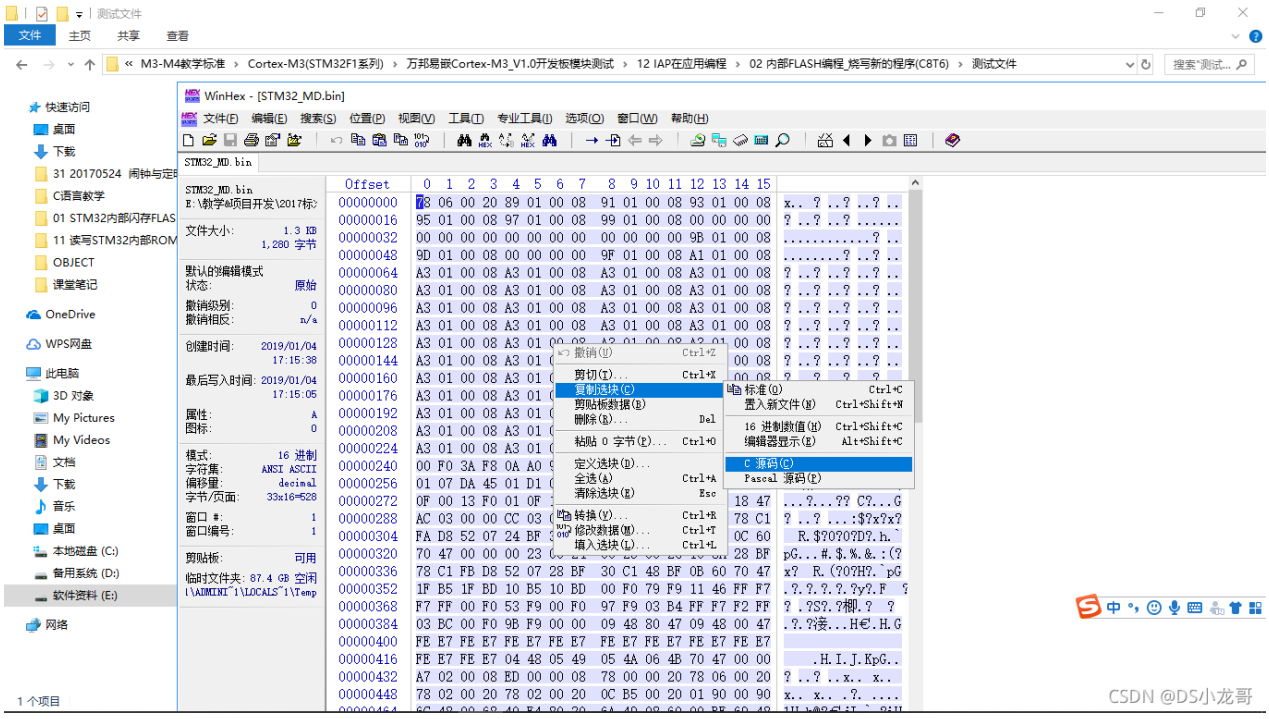




将该文件下载到 STM32 内置 FLASH，复位开发板，即可启动程序。

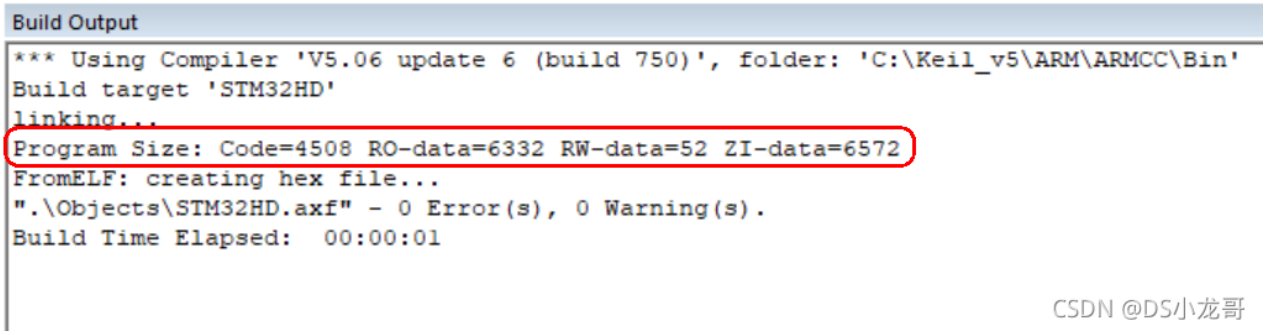
四、 使用 win hex 软件将 bin 文件搞成数组





生成数组之后，可以直接将数组编译到程序里，然后使用 **STM32** *内置 FLASH* 编程代码，将该程序烧写到内置 FLASH 里，再复位开发板即可运行新的程序。

五、 Keil 编译程序大小计算



Program Size: Code=x RO-data=x RW-data=x ZI-data=x 的含义

复制代码

1. Code(代码)：程序所占用的FLASH大小，存储在FLASH.
2. RO-data(只读的数据)：Read-only-data，程序定义的常量，如const型，存储在FLASH中。
3. RW-data(有初始值要求的、可读可写的数据)：
4. Read-write-data,已经被初始化的变量，存储在FLASH中。初始化时RW-data从flash拷贝到SR
5. ZI-data:Zero-Init-data,未被初始化的可读写变量，存储在SRAM中。ZI-data不会被算做代码

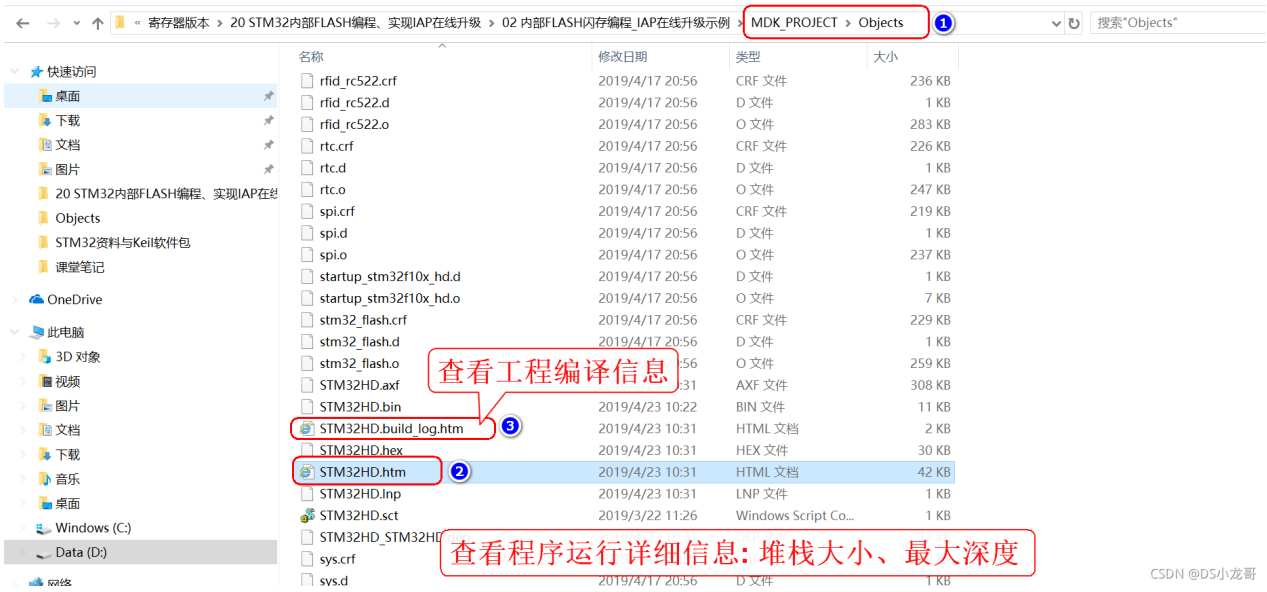
ROM(Flash) size = Code + RO-data + RW-data;

RAM size = RW-data + ZI-data

简单的说就是在烧写的时候是 FLASH 中的被占用的空间为：Code+RO Data+RW Data

程序运行的时候，芯片内部 RAM 使用的空间为： RW Data + ZI Data

六、工程编译信息与堆栈信息查看



对于没有 OS 的程序，堆栈大小是在 startup.s 里设置的： Stack_Size EQU 0x00000800

对于使用用 uCos 的系统，OS 自带任务的堆栈，在 os_cfg.h 里定义：

复制代码

```
/* ————————— TASK STACK SIZE ————————— */
#define OS_TASK_TMR_STK_SIZE     128     /* Timer      task stack size (# of OS_
#define OS_TASK_STAT_STK_SIZE    128     /* Statistics task stack size (# of OS_
#define OS_TASK_IDLE_STK_SIZE    128     /* Idle       task stack size (# of OS_
```

用户程序的任务堆栈，在 app_cfg.h 里定义：

复制代码

```
#define  APP_TASK_MANAGER_STK_SIZE           512
#define  APP_TASK_GSM_STK_SIZE                        512
#define  APP_TASK_OBD_STK_SIZE                         512
#define  OS_PROBE_TASK_STK_SIZE                       128
```

总结:

1, 合理设置堆栈很重要

2, 多种方法结合，相互核对、校验

3, 尽量避免大数组，如果一定要用，尽量定义为 全局变量，使其不占用堆栈空间, 如果函数有重入可能性，则要注意保护。

七、实现 STM32 在线升级程序

7.1 升级的思路与步骤

\1. 首先得完成 STM32 内置 FLASH 编程操作

\2. 将(升级的程序)新的程序编译生成 bin 文件(编译之前需要在 Keil 软件里设置 FLASH 的起始位置)

\3. 创建一个专门用于升级的 boot 程序(**IAP Bootloader)**

\4. 使用网络、串口、SD 卡等方式接收到 bin 文件，再将 bin 文件烧写到 STM32 内置 FLASH 里

\5. 设置主堆栈指针

\6. 将用户代码区第二个字(**第 4*** *个字节**)为程序开始地址(强制转为函数指针)

\7. **执行函数，进行程序跳转**

7.2 待升级的程序 FLASH 起始设置

Bootloader 的程序大小先固定为：20KB，最好是越小越好，可以预留更加多的空间给 APP 程序使用。

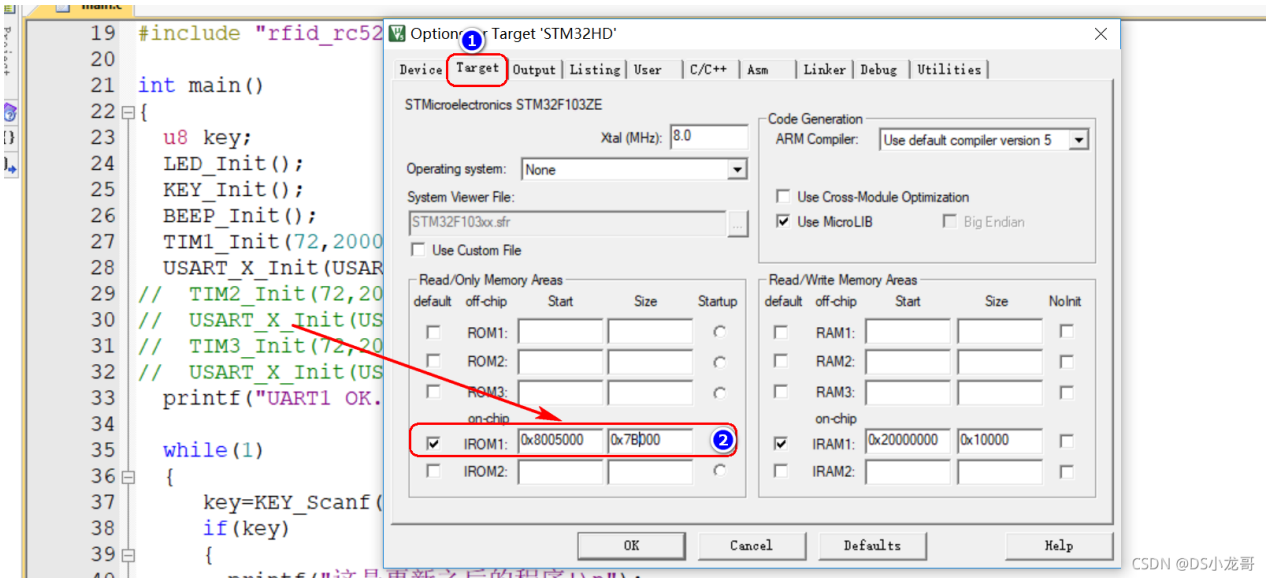
20KB----->20480Byte-----> 0x5000

STM32 内置 FLASH 闪存的起始地址是：0x08000000 ， 大小是 512KB。

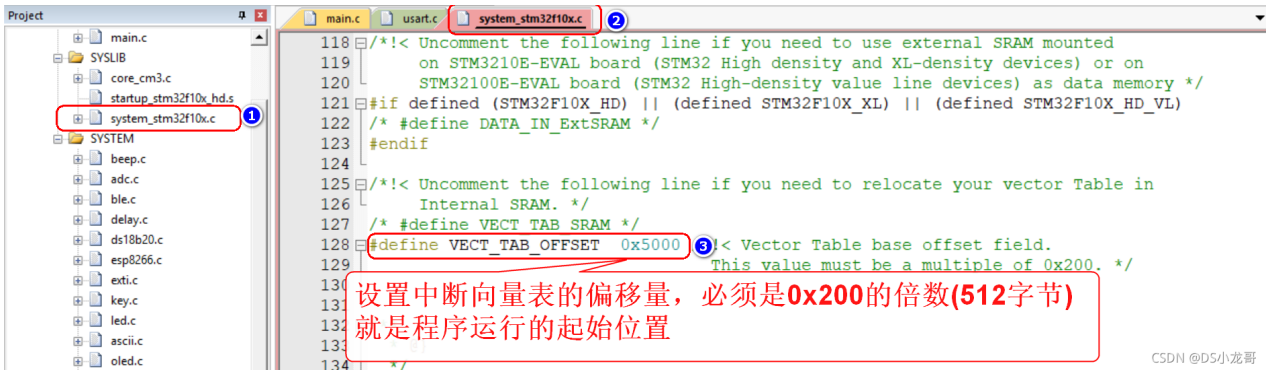
现在将内置 FLASH 闪存前 20KB 的空间留给 **Bootloader** 程序使用，后面剩下的空间就给 APP 程序使用。

APP 程序的起始位置就可以设置为：0x08000000+ 0x5000=0x08005000 剩余的大小就是：512KB-20KB=492KB------>503808Byte-------->0x7B000

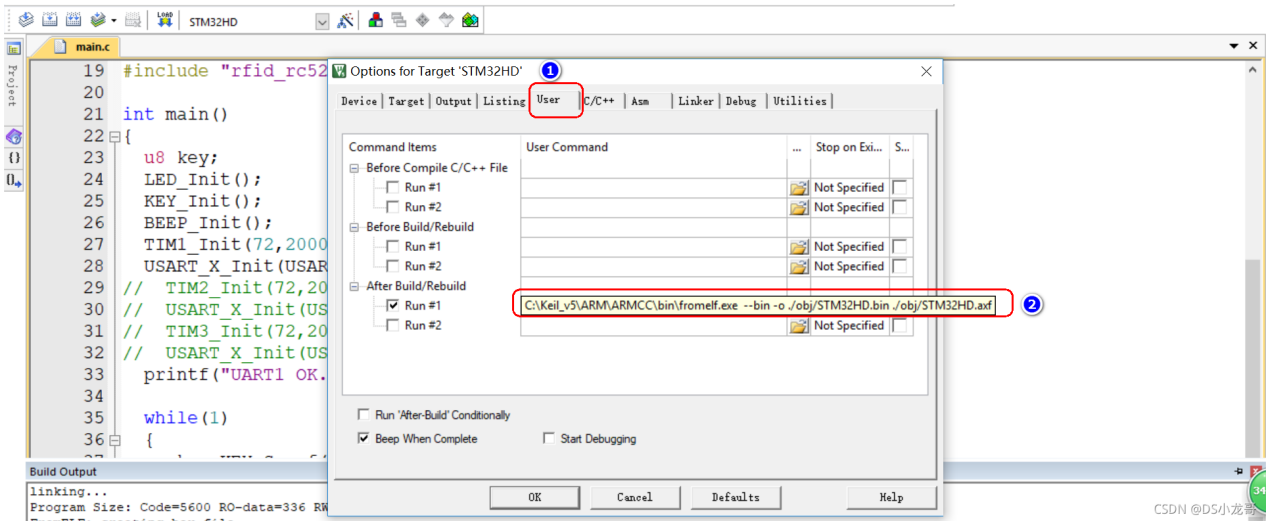
设置 FLASH 的起始位置（APP 主程序）：



中断向量表偏移量设置



设置编译 bin 文件



7.3 Bootloader 的程序设置

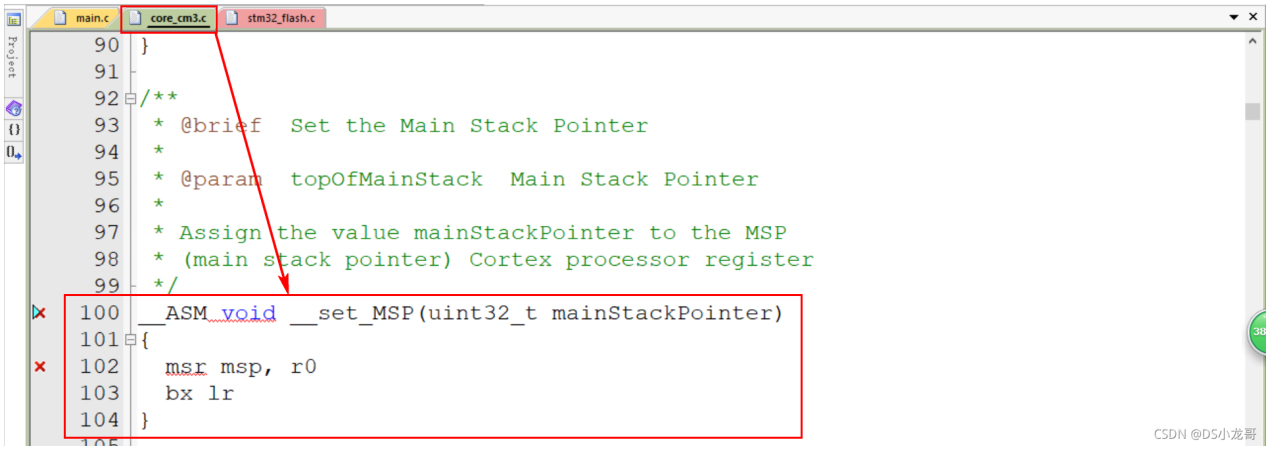
复制代码

```
//设置写入的地址,必须偶数,因为数据读写都是按照2个字节进行
#define FLASH_APP_ADDR      0x08005000      //应用程序存放到FLASH中的起始地址

int main()
{
  printf("UART1 OK.....\n");
printf("进入IAP Bootloader程序!\n");
  while(1)
```

```
    {
        key=KEY_Scanf();
        if(key==1)     //KEY1按下,写入STM32 FLASH
        {
          printf("正在更新IAP程序.............\n");
          iap_write_appbin(FLASH_APP_ADDR,(u8*)app_bin_data,sizeof(app_bin_data));
          printf("程序更新成功....\n");
          iap_load_app(FLASH_APP_ADDR);//执行FLASH APP代码
        }
    }
}

/*
函数功能:跳转到应用程序段
appxaddr:用户代码起始地址.
*/
typedef  void (*iap_function)(void);                    //定义一个函数类型的参数.
void IAP_LoadApp(u32 app_addr)
{
      //给函数指针赋值合法地址
      jump2app=(iap_function)*(vu32*)(app_addr+4);//用户代码区第二个字为程序开始地
      __set_MSP(*(vu32*)app_addr);  //设置主堆栈指针
      jump2app();                    //跳转到APP.
}
```



发布于: 2022-08-09　阅读数: 388