STM32Cube 学习之十五：SDIO+FATFS+IAP

为了简单起见，本篇的实验在上一篇例程基础上进行修改。

本篇例程参考了正点原子的“串口 IAP 实验”，在此声明感谢！

第一部分：IAP 程序

Step1.在用户代码区 0 增加一个函数，设置 PA0 为输入，作为按键输入接口，PF9、PF10 作为输出，控制 LED。并在 main 函数中用户代码区 2 中调用。

```
/* USER CODE BEGIN 0 */
/*设置PA0为输入,PF9,PF10为输出*/
void USER_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOF_CLK_ENABLE();

  /*PA0为按键输入*/
  GPIO_InitStruct.Pin = GPIO_PIN_0;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
  /*PF9、PF10为LED输出*/
  GPIO_InitStruct.Pin = GPIO_PIN_9 | GPIO_PIN_10;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
  GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
  HAL_GPIO_Init(GPIOF, &GPIO_InitStruct);
}
/* USER CODE END 0 */
```

Step2.修改 main 函数。以下是 main 函数的全部代码。

```
int main(void)
{

  /* USER CODE BEGIN 1 */
  uint8_t res;
  uint32_t tickstart = 0;
  uint32_t tmp32 = 0;
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------

  /* Reset of all peripherals, Initializes the Flash interfac
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_SDIO_SD_Init();
  MX_FATFS_Init();

  /* USER CODE BEGIN 2 */
    USER_GPIO_Init();
    f_mount(&SDFatFs, (TCHAR const*)SDPath, 0);
    tickstart = HAL_GetTick();  // 获取当前Tick定时值,单位ms
  /* USER CODE END 2 */
```

```c
  while (1)
  {
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
    if (GPIO_PIN_SET == HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0)) {  // 按键按下
        HAL_Delay(20);
        if (GPIO_PIN_SET == HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0)) {
            res = iap_write_appbin(APP_ADDRESS_IN_FLASH, "APP_code.bin"); // 升级APP
            if (0 == res) {
                // 升级成功
            } else if (2 == res) {
                // 无法找到APP_code.bin文件
            } else {
                // 升级失败
            }
            break;
        }
    }
    tmp32 = HAL_GetTick() - tickstart;
    if ((tmp32 % 1000) < 500) {  // LED闪烁,每秒闪一下
        HAL_GPIO_WritePin(GPIOF, GPIO_PIN_9, GPIO_PIN_SET); // LED0灭
        HAL_GPIO_WritePin(GPIOF, GPIO_PIN_10, GPIO_PIN_SET);// LED1灭
    } else {
        HAL_GPIO_WritePin(GPIOF, GPIO_PIN_9, GPIO_PIN_RESET); // LED0亮
        HAL_GPIO_WritePin(GPIOF, GPIO_PIN_10, GPIO_PIN_RESET);// LED1亮
    }
    if(tmp32 > 10000) {     // 超过10秒,自动跳出循环
        break;
    }
  }
    HAL_GPIO_WritePin(GPIOF, GPIO_PIN_9, GPIO_PIN_SET); // LED0灭
    HAL_GPIO_WritePin(GPIOF, GPIO_PIN_10, GPIO_PIN_SET);// LED1灭
    iap_load_app(APP_ADDRESS_IN_FLASH); // 加载运行APP
    while (1) { // 加载运行APP失败,LED闪烁,每秒闪2.5次
        HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_9);
        HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_10);
        HAL_Delay(200);
    }
  /* USER CODE END 3 */

}
```

main 函数的基本流程是，上电后 LED 闪烁表示 IAP 程序正在运行，10 秒之内如果有按键按下，则从 SD 卡读取根目录下的 APP_code.bin 文件，烧写到 APP_ADDRESS_IN_FLASH 地址中，然后跳转执行 APP。如果没有按键按下，则 10 秒后自动跳转执行 APP。

其中和 IAP 相关操作有两个关键函数，即 iap_write_appbin()和 iap_load_app()，一个用于更新 APP，一个用于加载 APP。这两个函数在 iap.c 和 iap.h 中实现。

Step3.实现 IAP 函数。

iap.h 文件内容：

```c
#ifndef __IAP_H__
#define __IAP_H__

#include "stm32f4xx_hal.h"

typedef  void (*iapfun)(void);                  // 定义一个函数类型的参数.

void iap_load_app(uint32_t appxaddr);           // 跳转到 APP 程序执行
uint8_t iap_write_appbin(const uint32_t appxaddr, const char *fname);// 在指定地址开始,写入 bin

#endif
```

iap.c 文件内容：

```
#include "stm32f4xx_hal.h"
#include "iap.h"
#include "stmflash.h"
#include "ff.h"

// 设置栈顶地址
// addr:栈顶地址
__asm void MSR_MSP(uint32_t addr)
{
    MSR MSP, r0     // set Main Stack value
    BX r14
}

iapfun jump2app;

/*
功能：从 SD 卡读取.bin 文件,写入指定 FLASH 地址中。
输入：appxaddr,应用程序在 FLASH 中的起始地址;fname,应用程序 bin 文件名.
返回：擦除或写入失败,返回 1;打开文件失败,返回 2;所有操作成功,返回 0.
*/
uint8_t iap_write_appbin(const uint32_t appxaddr, const char *fname)
{
  FRESULT res;
  FIL xFile;
  uint32_t real_read_len;
  union {
    uint8_t  dat8[2048];
    uint32_t dat32[512];   // 2K 字节缓存
  }iapbuf;

  uint8_t  res2;
  uint32_t t;
  uint32_t fwaddr;
  uint32_t appsize;

  res = f_open(&xFile, fname, FA_READ);   // 打开 APP bin 文件
  if( FR_OK != res ) {                // 如果失败,返回 2
    return 2;
  }

  appsize = f_size(&xFile);                       // 获取文件大小
  res2 = STMFLASH_Erase(appxaddr, appxaddr + appsize);   // 擦除 FLASH
  if (res2) {                                  // 如果失败,返回 1
    f_close(&xFile);
    return 1;
  }

  fwaddr = appxaddr; //当前写入的地址
     for(t=0; t < appsize; t += 2048)
     {
  res = f_read(&xFile, iapbuf.dat8, 2048, &real_read_len);   // 一次读 2048 字节
  if (FR_OK == res) {
    if (2048 == real_read_len) {
      res2 = STMFLASH_Write(fwaddr, iapbuf.dat32, 512);
    } else {
      res2 = STMFLASH_Write(fwaddr, iapbuf.dat32, (real_read_len >> 2));
```

```
            }
            if (res2) {
               f_close(&xFile);
               return 1;
            }
            fwaddr += 2048; // 偏移 2048
          } else {
            f_close(&xFile);
            return 1;
          }
           }

       f_close(&xFile);
       return 0;
    }

    /*
    功能:跳转到应用程序段
    输入:appxaddr,用户代码起始地址
    */
    void iap_load_app(uint32_t appxaddr)
    {
     if(((*(__IO uint32_t*)appxaddr) & 0x2FFE0000) == 0x20000000) { // 检查栈顶地址是否合法.
       jump2app = (iapfun)*(__IO uint32_t*)(appxaddr+4);        // 用户代码区第二个字为程序开始地址(复位地址)

       MSR_MSP(*(__IO uint32_t*)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.
     }
    }
```

其中，有两个 FLASH 操作的函数，STMFLASH_Erase()和 STMFLASH_Write()，分别是擦除和写入操作。这两个函数在 stmflash.c 和 stmflash.h 中实现。

Step4.实现 FLASH 操作函数。

stmflash.h 文件内容：

```
#ifndef __STMFLASH_H__
#define __STMFLASH_H__

#include "stm32f4xx_hal.h"

//FLASH 扇区的起始地址
#define ADDR_FLASH_SECTOR_0    ((uint32_t)0x08000000)      //扇区 0 起始地址, 16 Kbytes
#define ADDR_FLASH_SECTOR_1    ((uint32_t)0x08004000)      //扇区 1 起始地址, 16 Kbytes
#define ADDR_FLASH_SECTOR_2    ((uint32_t)0x08008000)      //扇区 2 起始地址, 16 Kbytes
#define ADDR_FLASH_SECTOR_3    ((uint32_t)0x0800C000)      //扇区 3 起始地址, 16 Kbytes
#define ADDR_FLASH_SECTOR_4    ((uint32_t)0x08010000)      //扇区 4 起始地址, 64 Kbytes
#define ADDR_FLASH_SECTOR_5    ((uint32_t)0x08020000)      //扇区 5 起始地址, 128 Kbytes
#define ADDR_FLASH_SECTOR_6    ((uint32_t)0x08040000)      //扇区 6 起始地址, 128 Kbytes
#define ADDR_FLASH_SECTOR_7    ((uint32_t)0x08060000)      //扇区 7 起始地址, 128 Kbytes
#define ADDR_FLASH_SECTOR_8    ((uint32_t)0x08080000)      //扇区 8 起始地址, 128 Kbytes
#define ADDR_FLASH_SECTOR_9    ((uint32_t)0x080A0000)      //扇区 9 起始地址, 128 Kbytes
#define ADDR_FLASH_SECTOR_10   ((uint32_t)0x080C0000)      //扇区 10 起始地址,128 Kbytes
#define ADDR_FLASH_SECTOR_11   ((uint32_t)0x080E0000)      //扇区 11 起始地址,128 Kbytes

//FLASH 扇区的大小
```

```
#define FLASH_SECTOR_0_SIZE     ((uint32_t)0x4000)      //扇区 0, 16 Kbytes
#define FLASH_SECTOR_1_SIZE     ((uint32_t)0x4000)      //扇区 1, 16 Kbytes
#define FLASH_SECTOR_2_SIZE     ((uint32_t)0x4000)      //扇区 2, 16 Kbytes
#define FLASH_SECTOR_3_SIZE     ((uint32_t)0x4000)      //扇区 3, 16 Kbytes
#define FLASH_SECTOR_4_SIZE     ((uint32_t)0x10000)         //扇区 4, 64 Kbytes
#define FLASH_SECTOR_5_SIZE     ((uint32_t)0x20000)         //扇区 5, 128 Kbytes
#define FLASH_SECTOR_6_SIZE     ((uint32_t)0x20000)         //扇区 6, 128 Kbytes
#define FLASH_SECTOR_7_SIZE     ((uint32_t)0x20000)         //扇区 7, 128 Kbytes
#define FLASH_SECTOR_8_SIZE     ((uint32_t)0x20000)         //扇区 8, 128 Kbytes
#define FLASH_SECTOR_9_SIZE     ((uint32_t)0x20000)         //扇区 9, 128 Kbytes
#define FLASH_SECTOR_10_SIZE    ((uint32_t)0x20000)         //扇区 10,128 Kbytes
#define FLASH_SECTOR_11_SIZE    ((uint32_t)0x20000)         //扇区 11,128 Kbytes

// 擦除指定地址空间的内容,注意擦除是按扇区操作的
uint8_t STMFLASH_Erase(uint32_t st_addr, uint32_t end_addr);
// 从指定地址开始写入指定长度的数据
uint8_t STMFLASH_Write(uint32_t WriteAddr,uint32_t *pBuffer,uint32_t NumToWrite);

#endif
```

stmflash.c 文件内容：

```
#include "stm32f4xx_hal.h"
#include "stmflash.h"

/*
功能：获取某个地址所在的 flash 扇区号
输入：addr,flash 地址
返回: 0~11,即 addr 所在的扇区号
*/
uint16_t STMFLASH_GetFlashSector(uint32_t addr)
{
        if(addr<ADDR_FLASH_SECTOR_1) return FLASH_SECTOR_0;
    else if(addr<ADDR_FLASH_SECTOR_2) return FLASH_SECTOR_1;
    else if(addr<ADDR_FLASH_SECTOR_3) return FLASH_SECTOR_2;
    else if(addr<ADDR_FLASH_SECTOR_4) return FLASH_SECTOR_3;
    else if(addr<ADDR_FLASH_SECTOR_5) return FLASH_SECTOR_4;
    else if(addr<ADDR_FLASH_SECTOR_6) return FLASH_SECTOR_5;
    else if(addr<ADDR_FLASH_SECTOR_7) return FLASH_SECTOR_6;
    else if(addr<ADDR_FLASH_SECTOR_8) return FLASH_SECTOR_7;
    else if(addr<ADDR_FLASH_SECTOR_9) return FLASH_SECTOR_8;
    else if(addr<ADDR_FLASH_SECTOR_10)return FLASH_SECTOR_9;
    else if(addr<ADDR_FLASH_SECTOR_11)return FLASH_SECTOR_10;
    return FLASH_SECTOR_11;
}

/*
功能：将指定数据写入指定 FLASH 空间
输入：WriteAddr,起始地址(此地址必须为 4 的倍数!);
    pBuffer,数据指针;
    NumToWrite 字(32 位)数(就是要写入的 32 位数据的个数.)
返回：操作成功返回 0,失败返回 1.
【注意】：该函数不包含擦除操作,调用该函数前,需对要写入的 FLASH 空间进行擦除。
*/
uint8_t STMFLASH_Write(uint32_t WriteAddr, uint32_t *pBuffer, uint32_t NumToWrite)
{
  uint32_t i;
```

```
    if((WriteAddr < FLASH_BASE) || (WriteAddr % 4))return 1;       //非法地址

        HAL_FLASH_Unlock();       //解锁
    for (i=0; i < NumToWrite; i++) {
      // 写入单位,WORD,即 4 字节
      if (HAL_OK != HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, WriteAddr, pBuffer[i])) return 1;
      WriteAddr += 4; // 每次写 4 字节,因此地址增加 4
    }
        HAL_FLASH_Lock();  //上锁

    return 0;
}

/*
功能：擦除从 st_addr 到 end_addr 的 FLASH 空间。
输入：st_addr,起始地址,必须为 4 的整数倍;end_addr,结束地址。
返回：正确返回 0;错误返回 1。
*/
uint8_t STMFLASH_Erase(uint32_t st_addr, uint32_t end_addr)
{
        uint32_t PAGEError = 0;
    FLASH_EraseInitTypeDef EraseInitStruct;
        uint8_t st_sector=0, end_sector=0;

    if((st_addr < FLASH_BASE) || (st_addr % 4))return 1;       //非法地址

    st_sector = STMFLASH_GetFlashSector(st_addr);  // 起始扇区
    end_sector = STMFLASH_GetFlashSector(end_addr); // 结束扇区

        EraseInitStruct.TypeErase   = FLASH_TYPEERASE_SECTORS;
        EraseInitStruct.Banks               = FLASH_BANK_1;
        EraseInitStruct.Sector        = st_sector;
        EraseInitStruct.NbSectors   = end_sector - st_sector + 1;
        EraseInitStruct.VoltageRange= FLASH_VOLTAGE_RANGE_3;

        HAL_FLASH_Unlock();       //解锁
        if (HAL_OK != HAL_FLASHEx_Erase(&EraseInitStruct, &PAGEError)) {   // 擦除 FLASH
              return 1;
        }
        HAL_FLASH_Lock();  //上锁

    return 0;
}
```

Step5. 定义 APP 数组。

因为要在 FLASH 中划出空间存放 APP 代码，所以要定义一个 const 数组，并指定其在 FLASH 中的空间。该步骤在 APP_code.c 和 APP_code.h 文件中实现。

APP_code.h 文件内容：

```
#ifndef __APP_CODE_H__
#define __APP_CODE_H__

#include "stm32f4xx_hal.h"
#include "stmflash.h"

#define     APP_ADDRESS_IN_FLASH     (ADDR_FLASH_SECTOR_4)

extern const unsigned char APP_code[] __attribute__((at(APP_ADDRESS_IN_FLASH)));

#endif
```

APP_code.c 文件内容：

```
#include "app_code.h"
#include "stmflash.h"

// APP_ADDRESS_IN_FLASH FLASH的第4,5,6扇区 64k+128k+128k = 320k 存放应用程序APP
const unsigned char
APP_code[FLASH_SECTOR_4_SIZE + FLASH_SECTOR_5_SIZE + FLASH_SECTOR_6_SIZE]
__attribute__((at(APP_ADDRESS_IN_FLASH))) = {
    0
};
```

本例中，如上代码将 APP_code 数组定义到扇区 4 为起始地址，数组大小为 320k，即 APP 最大可用空间为 320k，占用扇区 4、5、6。而扇区 0~3 留给 IAP 程序或者其他用途。

至此，IAP 程序设计完成。目前，APP 代码为空。在完成 APP 并生成相应 bin 文件之后，用 winhex 转换成 C 代码格式，复制到 APP_code[]数组，即可将 APP 代码和 IAP 代码一起烧录。

第二部分：APP 程序

APP 程序只要在普通程序基础上进行几个简单修改即可。下面以 UART 串口为例。

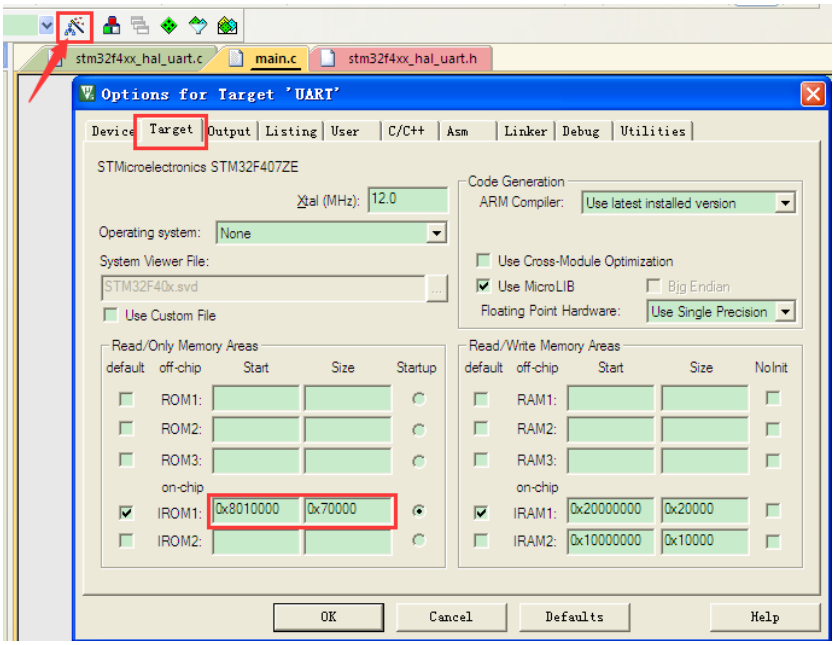
Step1.创建 UART 程序：详细过程请参考本系列笔记第二篇。

Step2.添加串口发送代码。

```
  while (1)
  {
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
      #if 1
          HAL_UART_Transmit(&huart1, "Hello!\r\n", 8, 10);
      #else
          HAL_UART_Transmit(&huart1, "Hello World!\r\n", 14, 10);
      #endif
      HAL_Delay(1000);
  }
  /* USER CODE END 3 */
```

添加如上代码，为后面实验方便，使用预编译处理，通过修改红框中的条件决定串口输出内容。条件为 0 输出 Hello World！,条件不为 0 输出 Hello！。

Step3. 设置 APP 程序起始地址。



打开目标选项设置窗口，在 Target 页面 IROM1 中，将 Start 地址设置成 APP 地址，本例中为 0x80100000，即扇区 4 的地址。Size 地址进行相应的修改，Start 从原来的 0x80000000 改为 0x80100000，增加了 0x100000，Size 要减小 0x100000。本例中改为 0x700000。

Step4.设置向量表位置。

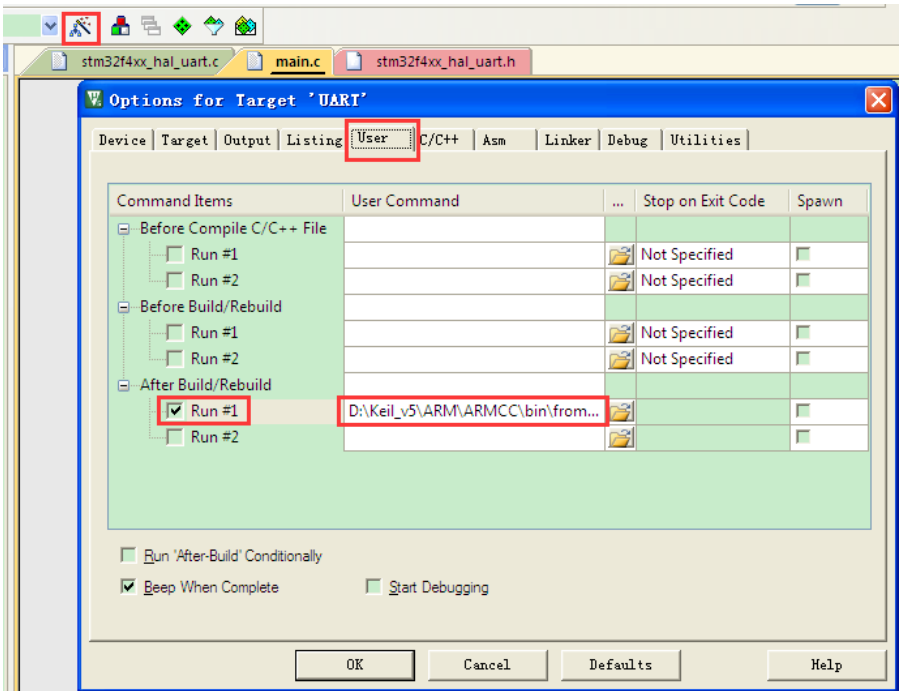
在 main 函数的开始处设置向量表位置，如下图：

```
/* USER CODE BEGIN 0 */
#define ADDR_FLASH_SECTOR_4     ((uint32_t)0x08010000)  //扇区4起始地址, 64 Kbytes
/* USER CODE END 0 */

int main(void)
{

  /* USER CODE BEGIN 1 */
    SCB->VTOR = ADDR_FLASH_SECTOR_4;   // 设置向量表位置
  /* USER CODE END 1 */
```

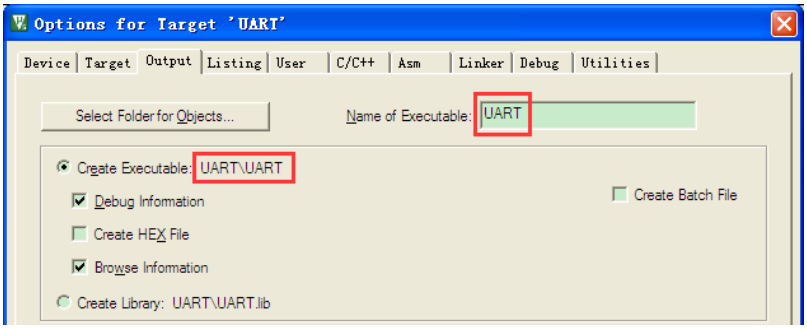
这里设置的向量表位置必须和 IAP 函数给 APP 指定的位置一致。

Step5.设置编译链接转换工具。

打开目标选项设置窗口，在 User 页面中如下图，勾选 Run#1，并在命令栏输入转换工具路径即命令"D:\Keil_v5\ARM\ARMCC\bin\fromelf.exe --bin -o OBJ\APP_code.bin UART\UART.axf"。



其中后面的.axf 文件名称必须和 Output 页面的设置一致，如下图。而.bin 文件是输出文件，路径和名称可以是任意的，本例输出到本工程目录下的 OBJ 文件夹(如果没有就自动创建)，输出文件名称为 APP_code.bin。



Step6.编译生成 bin 文件。

点击编译按钮，即可创建 APP_code.bin 文件。

第三部分：IAP+APP 综合

经过第二部分的操作，得到了 APP 的 bin 文件，这个文件可以通过 SD 卡升级到 MCU 上，也可以和第一部分的 IAP 程序一起烧录到 MCU。

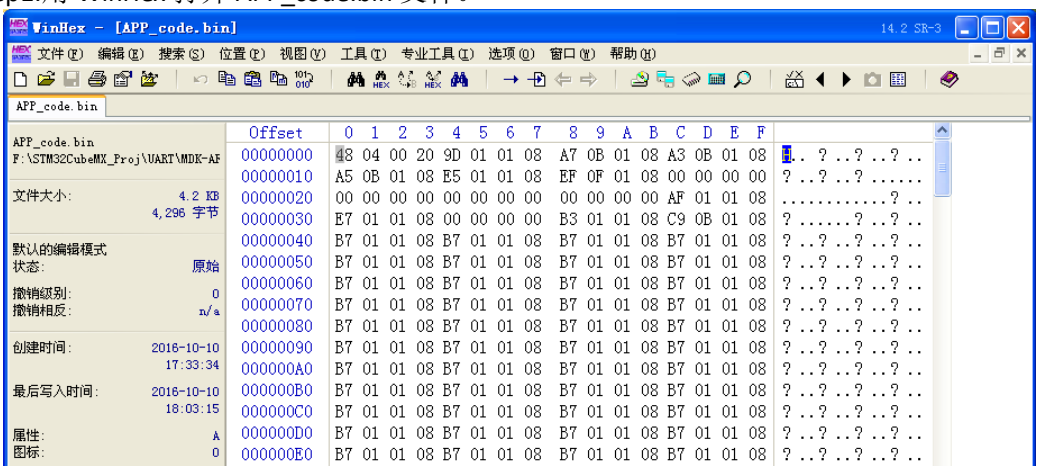
用 SD 卡升级的方法。

Step1.将第一部分的代码通过 J-Link 烧录到 MCU。
Step2.将第二部分生成的 APP_code.bin 文件拷贝到 SD 卡的根目录下。
Step3.将 SD 卡插到开发板的 SD 卡座。
Step4.给开发板上电，LED 会闪烁。
Step5.在上电后 10 秒钟内，按下按键。此时程序会将 APP_code.bin 加载到 MCU 的 FLASH 上。
Step6.将 MCU 的 UART 和 PC 连接，用串口调试助手可以收到 MCU 发送的 Hello!，每秒钟接收到一次。
Step7.如果将开发板重新上电，在无任何操作的情况下，需要等待 10 秒 MCU 才开始发送数据。

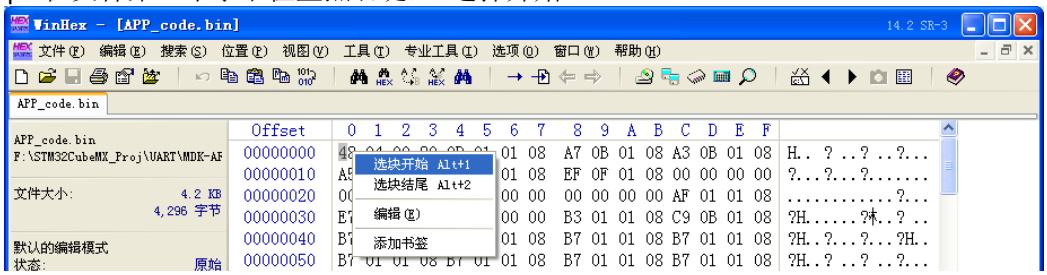
如果将第二部分 Step2 中的预编译条件改为 0，然后重复上述步骤，则串口发送的将是 Hello World！。

IAP 和 APP 一起烧录的方法。

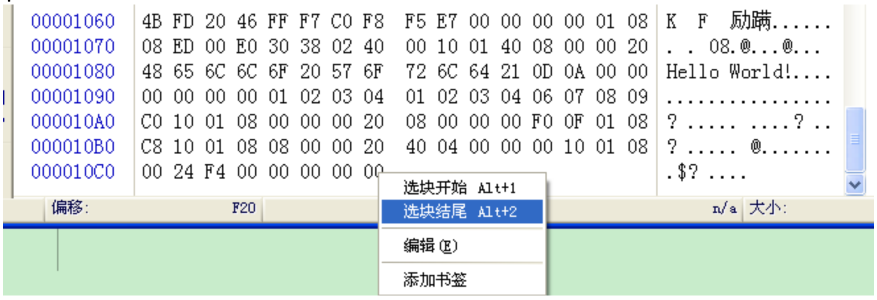
Step1.用 WinHex 打开 APP_code.bin 文件。



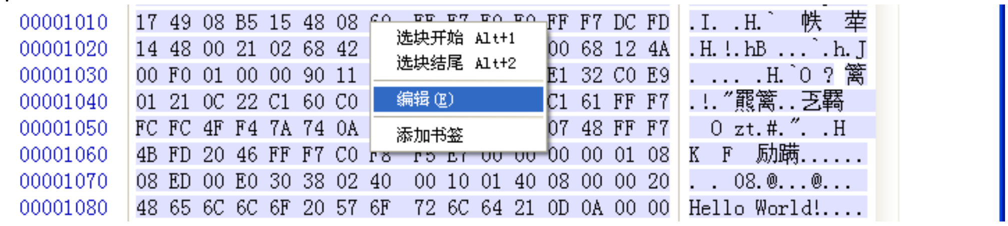
Step2.在文件第一个字节位置点右键，“选择开始”。



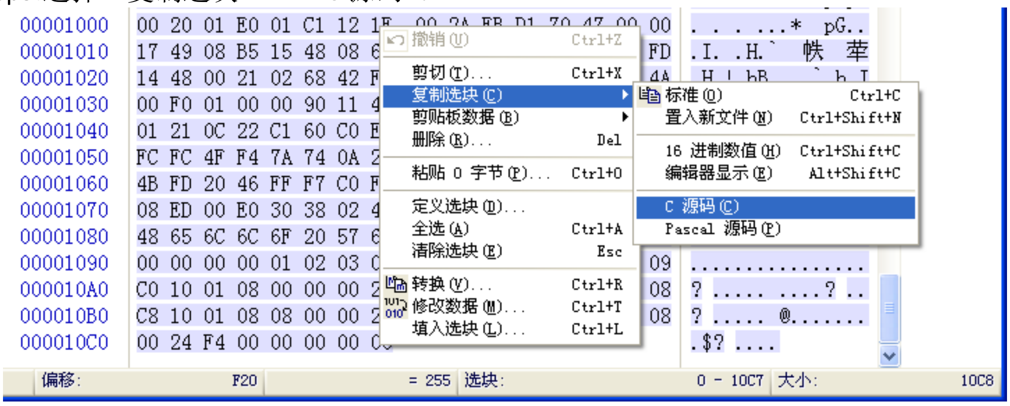
Step3.拖动右边滑块到最底端，在最后一个字节点右键，“选择结尾”。



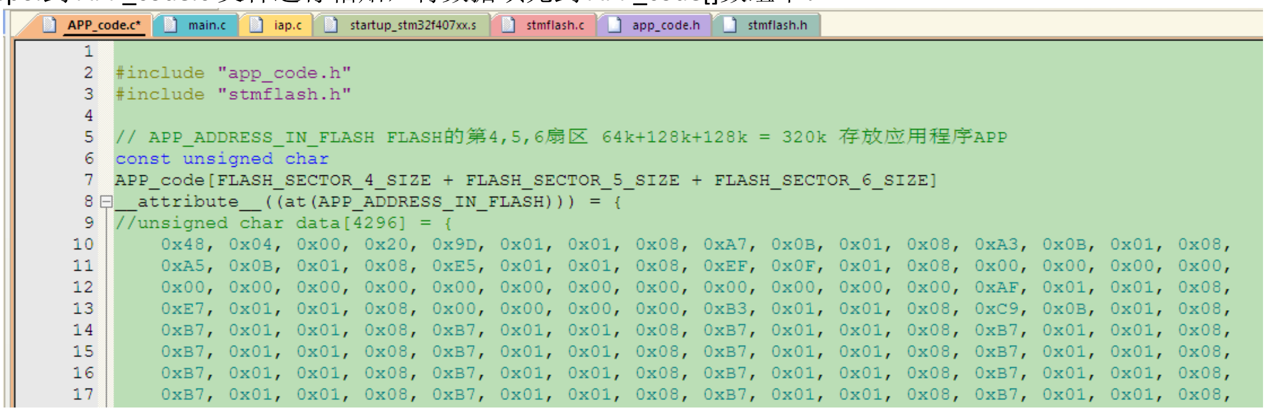
Step4.在被选中数据区域再次点击右键，选择“编辑”。



Step5.选择“复制选块” --> “C 源码”。



Step6.到 APP_code.c 文件进行粘贴，将数据填充到 APP_code[]数组中。



Step7.重新编译 IAP 工程，然后下载。这次下载的 IAP 程序就包含了 APP 的内容。下载完成后，直接运行，10 秒后就可以从串口输出数据了。

S.D.Lu 于深圳
2016 年 10 月