第 6 章　STM32 存储器管理与文件系统

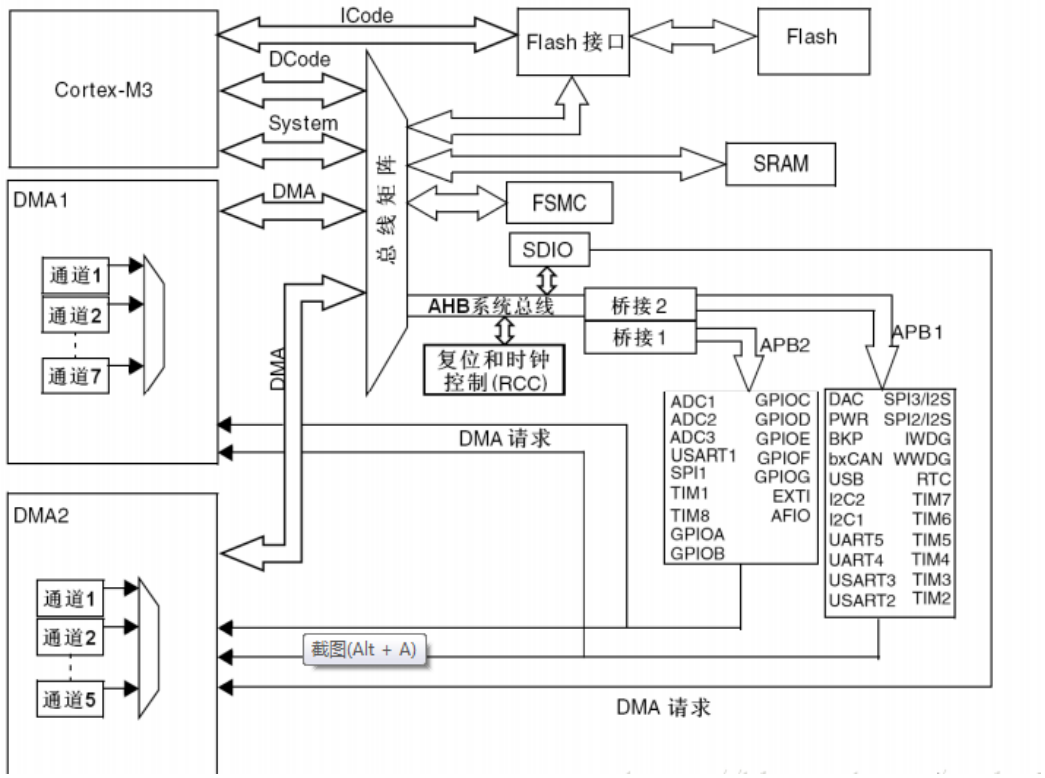
本章简介

本章主要介绍 STM32 存储器的结构、组织和映射；Flash 存储器的结构和读写操作；SRAM 的结构、扩展和备份；SD 卡的结构和驱动；FATFS 文件系统的结构、特点和驱动。

6.1 存储器的组织

6.1.1.存储结构

STM32 有四种存储单元，依次是 SRAM、Flash、FSMC 和 AHB 到 APB 桥（挂载各种外设），其结构图如图 6-1 所示。



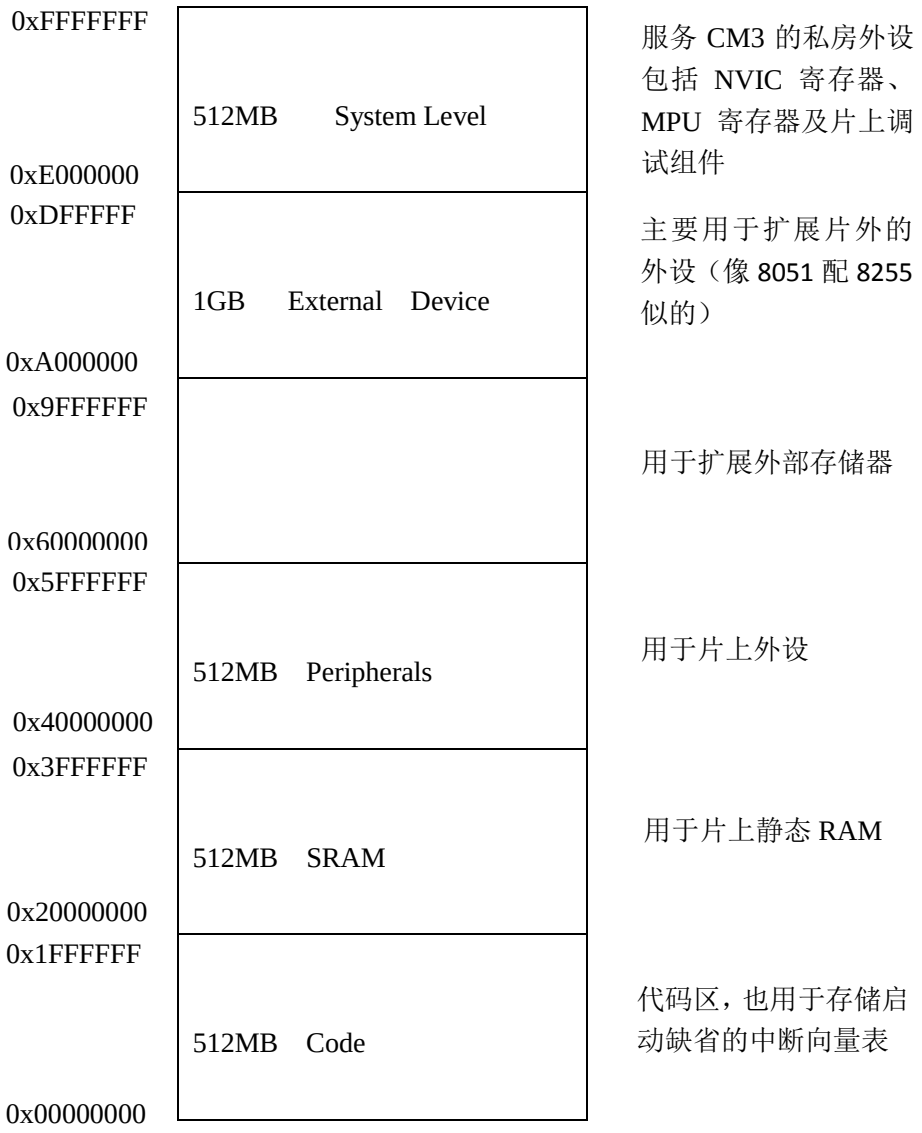
6-1　存储结构图

6.1.2.存储组织

STM 存储组织如图 6-2 所示。程序存储器、数据存储器、寄存器和输入输出端口被组织在同一个 4GB 的线性地址空间内。数据字节以小端格式存放在存储器中。一个字里的最低地址字节被认为是该字的最低有效字节，而最高地址字节是最高有效字节。

可访问的存储器空间被分成 8 个主要块，每个块为 512MB。其他所有没有分配给片上存储器和外设的存储器空间都是保留的地址空间。FLASH 存储下载的程序，

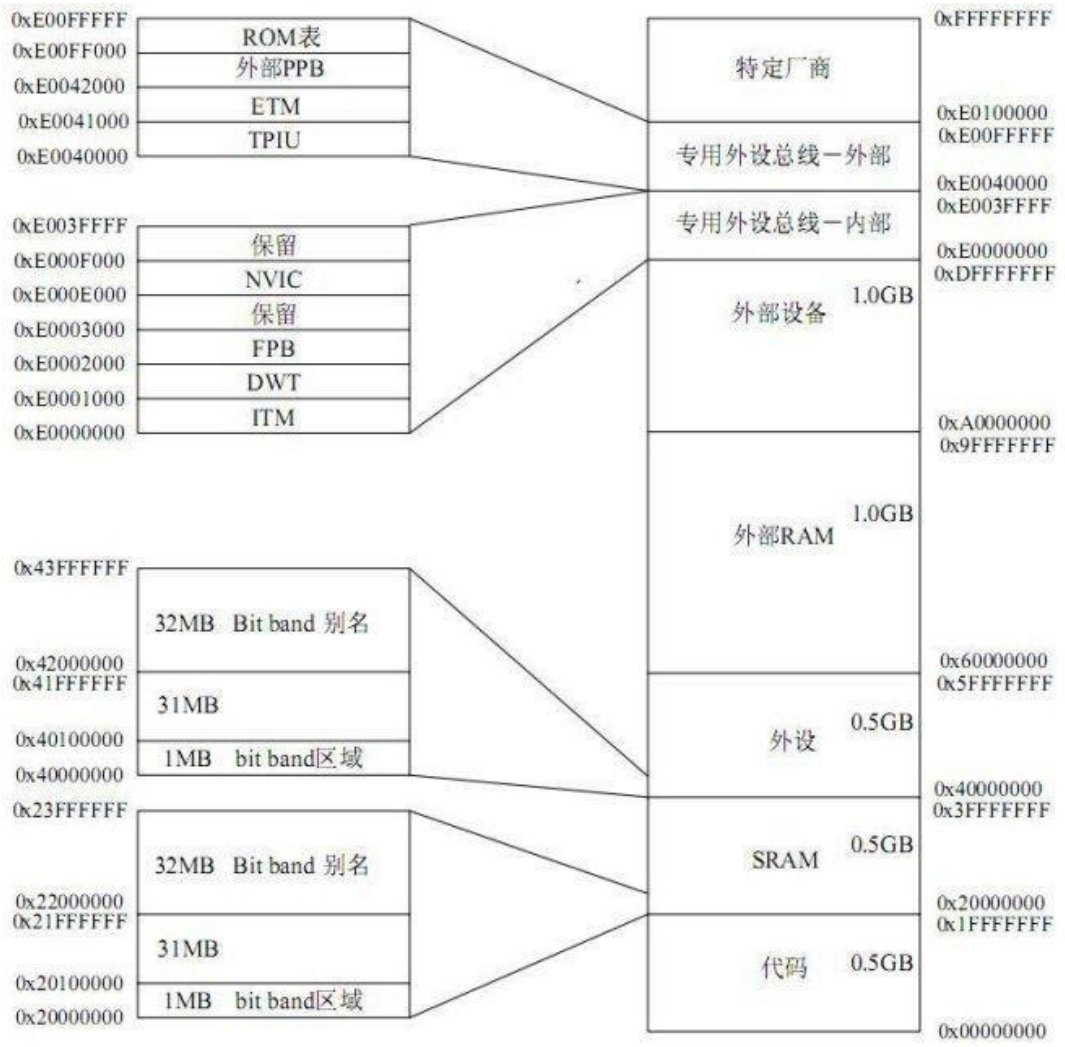
FLASH 是 ROM 的一种。SRAM 是存储运行程序中的数据，SRAM 是 RAM 的一种。所以，只要不外扩存储器，写完的程序中的所有东西也就会出现在这两个存储器中。



6-2 存储组织图

6.1.3.存储器映射

存储器本身不具有地址信息，它的地址是由芯片厂商或用户分配，给存储器分配地址的过程称为存储器映射，如果再分配一个地址就叫重映射（具体地址分配参考芯片数据手册及中文参考手册）。存储器映射图如图 6-3 所示。



6-3　存储器映射图

以上存储器映射的对应地址，用户不可更改。用户的外设、扩展 Flash 和扩展 RAM 只能挂在外部设备区和外部 RAM 区。

6.2 Flash 存储器

从存储器映射图中看出，stm32 的 flash 地址起始于 0x0800 0000，结束地址是 0x0800 0000 加上芯片实际的 flash 大小，不同的芯片 flash 大小不同。Flash 中的内容一般用来存储代码和一些定义为 const 的数据，断电不丢失。

6.2.1.内部 Flash 的构成

STM32F103 的中容量内部 FLASH 包含主存储器、系统存储器、 OTP 区域以及选项字节区域，它们的地址分布及大小如表 6-1 所示。

表 6-1　地址分布

		扇区 0	0x08000000-0x08003FFF	16Kbytes
		扇区 1	0x08004000-0x08007FFF	16Kbytes

主存储器	块 1	扇区 2	0x08008000-0x0800BFFF	16Kbytes
		扇区 3	0x0800C000-0x0800FFFF	16Kbytes
		扇区 4	0x08010000-0x0801FFFF	64Kbytes
		扇区 5	0x08020000-0x0803FFFF	128Kbytes
		扇区 6	0x08040000-0x0805FFFF	128Kbytes
		扇区 7	0x08060000-0x0807FFFF	128Kbytes
		扇区 8	0x08080000-0x0809FFFF	128Kbytes
		扇区 9	0x080A0000-0x080BFFFF	128Kbytes
		扇区 10	0x080C0000-0x080DFFFF	128Kbytes
		扇区 11	0x080E0000-0x080FFFFF	128Kbytes
	块 2	扇区 12	0x08100000-0x08103FFF	16Kbytes
		扇区 13	0x08104000-0x08107FFF	16Kbytes
		扇区 14	0x08108000-0x0810BFFF	16Kbytes
		扇区 15	0x0810C000-0x0810FFFF	16Kbytes
		扇区 16	0x08110000-0x0811FFFF	64Kbytes
		扇区 17	0x08120000-0x0813FFFF	128Kbytes
		扇区 18	0x08140000-0x0815FFFF	128Kbytes
		扇区 19	0x08160000-0x0817FFFF	128Kbytes
		扇区 20	0x08180000-0x0819FFFF	128Kbytes
		扇区 21	0x081A0000-0x081BFFFF	128Kbytes
		扇区 22	0x081C0000-0x081DFFFF	128Kbytes
		扇区 23	0x08E00000-0x081FFFFF	128Kbytes
系统存储区			0x1FFF0000-0x1FFF77FF	30Kbytes
OTP 区域			0x1FFF7800-0x1FFF7A0F	528bytes
选项字节	块 1		0x1FFFC000-0x1FFFC00F	16bytes
	块 2		0x1FFEC000-0x1FFEC00F	16bytes

1.主存储器

一般我们说 STM32 内部 FLASH 的时候，都是指这个主存储器区域它是存储用户应用程序的空间，芯片型号说明中的 1M FLASH、 2M FLASH 都是指这个区域的大小。与其它 FLASH 一样，在写入数据前，要先按扇区擦除。

2.系统存储区

系统存储区是用户不能访问的区域，它在芯片出厂时已经固化了启动代码，它负责实现串口、 USB 以及 CAN 等 ISP 烧录功能。

3.OTP 区域

OTP(One Time Program)，指的是只能写入一次的存储区域，容量为 512 字节，写入后数据就无法再更改， OTP 常用于存储应用程序的加密密钥。

4.选项字节

选项字节用于配置 FLASH 的读写保护、电源管理中的 BOR 级别、软件/硬件看门狗等功能，这部分共 32 字节。可以通过修改 FLASH 的选项控制寄存器修改。

6.2.2.内部 Flash 的写过程

Flash 的写入地址必须是偶数（FLASH 机制决定的 FLASH 写入的时候只能是偶数地址写入，必须写入半字或字，也就是 2 个字节或是 4 字节的内容）。其过程如下。

1.解锁 （固定的 KEY 值）

直接调用#include "stm32f10x_flash.h"中的 void FLASH_Unlock(void)函数，这个函数是官方提供的。

（1）往 Flash 密钥寄存器 FLASH_KEYR 中写入 KEY1 = 0x45670123

（2）再往 Flash 密钥寄存器 FLASH_KEYR 中写入 KEY2 = 0xCDEF89AB

2.数据操作位数

最大操作位数会影响擦除和写入的速度，其中 64 位宽度的操作除了配置寄存器位外，还需要在 Vpp 引脚外加一个 8-9V 的电压源，且其供电间不得超过一小时，否则 FLASH 可能损坏，所以 64 位宽度的操作一般是在量产时对 FLASH 写入应用程序时才使用，大部分应用场合都是用 32 位的宽度。

3.擦除扇区

直接调用固件库官方的函数 FLASH_Status FLASH_ErasePage(uint32_t Page_Address)，在写入新的数据前，需要先擦除存储区域， STM32 提供了扇区擦除指令和整个 FLASH 擦除(批量擦除)的指令，批量擦除指令仅针对主存储区。 扇区擦除的过程如下。

（1）检查 FLASH_SR 寄存器中的“忙碌寄存器位 BSY”，以确认当前未执行任何 Flash 操作；

（2）在 FLASH_CR 寄存器中，将“激活扇区擦除寄存器位 SER ”置 1，并设置“扇 区编号寄存器位 SNB”，选择要擦除的扇区；

（3）将 FLASH_CR 寄存器中的“开始擦除寄存器位 STRT ”置 1，开始擦除；

（4）等待 BSY 位被清零时，表示擦除完成。

4.写入数据

擦除完毕后即可写入数据，写入数据的过程并不是仅仅使用指针向地址赋值，赋值前还还需要配置一系列的寄存器，步骤如下：

（1）检查 FLASH_SR 中的 BSY 位，以确认当前未执行任何其它的内部 Flash 操作；

（2）将 FLASH_CR 寄存器中的 “激活编程寄存器位 PG” 置 1；

（3）针对所需存储器地址（主存储器块或 OTP 区域内）执行数据写入操作；

（4）等待 BSY 位被清零时，表示写入完成。

【实例 6-1】

```
#define FLASH_PAGE_SIZE     ((uint16_t)0x400)       //如果一页为 1K 大小
#define WRITE_START_ADDR    ((uint32_t)0x08008000)       //写入的起始地址
#define WRITE_END_ADDR      ((uint32_t)0x0800C000)     //结束地址
uint32_t EraseCounter = 0x00, Address = 0x00;     //擦除计数，写入地址
uint32_t Data = 0x3210ABCD;       //要写入的数据
uint32_t NbrOfPage = 0x00;        //记录要擦除的页数
volatile FLASH_Status FLASHStatus = FLASH_COMPLETE;     /*FLASH 擦除完成标志*/
void main()
{
 /*解锁 FLASH*/
   FLASH_Unlock();
 /*计算需要擦除 FLASH 页的个数 */
   NbrOfPage = (WRITE_END_ADDR - WRITE_START_ADDR) / FLASH_PAGE_SIZE;
 /* 清除所有挂起标志位 */
              FLASH_ClearFlag(FLASH_FLAG_EOP      |      FLASH_FLAG_PGERR      |
FLASH_FLAG_WRPRTERR);
 /* 擦除 FLASH 页*/
    for(EraseCounter = 0; (EraseCounter < NbrOfPage) && (FLASHStatus ==
FLASH_COMPLETE); EraseCounter++)
      {
     FLASHStatus = FLASH_ErasePage(WRITE_START_ADDR + (FLASH_PAGE_SIZE *
EraseCounter));
      }
 /* 写入 FLASH */
   Address = WRITE_START_ADDR;
```

```
  while((Address < WRITE_END_ADDR) && (FLASHStatus == FLASH_COMPLETE))
      {
        FLASHStatus = FLASH_ProgramWord(Address, Data);
        Address = Address + 4;
      }
/* 锁定 FLASH */
  FLASH_Lock();
}
```

6.2.3.内部 Flash 的擦除过程

1.擦除函数

FLASH_Status FLASH_ErasePage(u32 Page_Address)只要（）里面的数是 flash 第 xx 页中对应的任何一个地址！就是擦除 xx 页全部内容！

2.防止误擦除有用程序代码的方法

方法一：首先要计算程序代码有多少，把 FLASH 存取地址设置在程序代码以外的地方，这样就不会破坏用户程序。原则上从 0x0800 0000 + 0x1000 以后的 FLASH 空间都可以作为存储使用。如果代码量占了 0x3000， 那么存储在 0x0800 0000+ 0x4000 以后的空间就不会破坏程序了。

方法二：先在程序中定义一个 const 类型的常量数组，并指定其存储位置（方便找到写入、读取位置），这样编译器就会分配你指定的空间将常量数组存入 FLASH 中。当你做擦除。读写操作时，只要在这个常量数组所在的地址范围就好。

```
const uint8_t table[10] __at(0x08010000) = {0x55} ;
```

MDK3.03A 开始就支持关键字 __at() 。

方法三：在程序中定义一个 const 类型的常量数组，无需指定其存储位置。只要定义一个 32 位的变量存储这个数组的 FLASH 区地址就行。

```
uint32_t address;//STM32 的地址是 32 位的
const uint8_t imageBuffer[1024] = {0,1,2,3,4,5,6,7};
address = (uint32_t) imageBuffer;/*用强制类型转换的方式，可以把 FLASH 中存储的 imageBuffer[1024]的地址读到 RAM 中的变量 address 里，方便找到写入、读取位置*/
```

3.内部 Flash 的读入过程

```
*(uint32_t *)0x8000000;//读一个字
*(uint8_t *)0x8000000;//读一个字节;
*(uint16_t *)0x8000000;//读半字;
```

【实例 6-2】

```
u16 STMFLASH_ReadHalfWord(u32 faddr)
{
return *(vu16*)faddr;
}
//读取指定地址的半字(16 位数据)
//faddr:读地址(此地址必须为 2 的倍数!!)
//返回值:对应数据.
```

```
u16 STMFLASH_ReadHalfWord(u32 faddr)
{
   return *(vu16*)faddr;
}
#if STM32_FLASH_WREN    //如果使能了写
//不检查的写入
//WriteAddr:起始地址
//pBuffer:数据指针
//NumToWrite:半字(16 位)数
void STMFLASH_Write_NoCheck(u32 WriteAddr,u16 *pBuffer,u16 NumToWrite)
{
   u16 i;
   for(i=0;i<NumToWrite;i++)
   {
      FLASH_ProgramHalfWord(WriteAddr,pBuffer[i]);
    WriteAddr+=2;//地址增加 2.
   }
}
 //从指定地址开始写入指定长度的数据
 //WriteAddr:起始地址(此地址必须为 2 的倍数!!)
 //pBuffer:数据指针
 //NumToWrite:半字(16 位)数(就是要写入的 16 位数据的个数.)
 #if STM32_FLASH_SIZE<256
 #define STM_SECTOR_SIZE 1024 //字节
 #else
 #define STM_SECTOR_SIZE    2048
 #endif
 u16 STMFLASH_BUF[STM_SECTOR_SIZE/2];    //最多是 2K 字节
 void STMFLASH_Write(u32 WriteAddr,u16 *pBuffer,u16 NumToWrite)
{
   u32 secpos;     //扇区地址
   u16 secoff;     //扇区内偏移地址(16 位字计算)
   u16 secremain;    //扇区内剩余地址(16 位字计算)
    u16 i;
   u32 offaddr;    //去掉 0X08000000 后的地址
   if(WriteAddr<STM32_FLASH_BASE||(WriteAddr>=(STM32_FLASH_BASE+1024*STM32_
FLASH_SIZE)))return;    //非法地址
   FLASH_Unlock();                    //解锁
   offaddr=WriteAddr-STM32_FLASH_BASE;       //实际偏移地址.
   secpos=offaddr/STM_SECTOR_SIZE;           //扇区地址  0~127 for STM32F103RBT6
   secoff=(offaddr%STM_SECTOR_SIZE)/2;       //在扇区内的偏移(2 个字节为基本单位.)
   secremain=STM_SECTOR_SIZE/2-secoff;       //扇区剩余空间大小
   if(NumToWrite<=secremain)secremain=NumToWrite ;     //不大于该扇区范围
   while(1)
```

```
  {
    STMFLASH_Read(secpos*STM_SECTOR_SIZE+STM32_FLASH_BASE,STMFLASH_BUF,STM_SECTOR_SIZE/2);    //读出整个扇区的内容
    for(i=0;i<secremain;i++)//校验数据
    {
      if(STMFLASH_BUF[secoff+i]!=0XFFFF)break;   //需要擦除
    }
    if(i<secremain)//需要擦除
    {
      FLASH_ErasePage(secpos*STM_SECTOR_SIZE+STM32_FLASH_BASE); //擦除这个扇区
      for(i=0;i<secremain;i++)//复制
      {
        STMFLASH_BUF[i+secoff]=pBuffer[i];
      }
      STMFLASH_Write_NoCheck(secpos*STM_SECTOR_SIZE+STM32_FLASH_BASE,STMFLASH_BUF,STM_SECTOR_SIZE/2);//写入整个扇区
    }else STMFLASH_Write_NoCheck(WriteAddr,pBuffer,secremain); //写已经擦除了的,直接写入扇区剩余区间.
    if(NumToWrite==secremain)break;   //写入结束了
    else//写入未结束
    {
      secpos++;              //扇区地址增 1
      secoff=0;              //偏移位置为 0
       pBuffer+=secremain;    //指针偏移
      WriteAddr+=secremain;   //写地址偏移
       NumToWrite-=secremain;   //字节(16 位)数递减
      if(NumToWrite>(STM_SECTOR_SIZE/2))secremain=STM_SECTOR_SIZE/2;    // 下一个扇区还是写不完
      else secremain=NumToWrite;    //下一个扇区可以写完了
    }
  };
  FLASH_Lock();   //上锁
}
#endif
//从指定地址开始读出指定长度的数据
//ReadAddr:起始地址
//pBuffer:数据指针
//NumToWrite:半字(16 位)数
void STMFLASH_Read(u32 ReadAddr,u16 *pBuffer,u16 NumToRead)
{
  u16 i;
  for(i=0;i<NumToRead;i++)
  {
```

```
        pBuffer[i]=STMFLASH_ReadHalfWord(ReadAddr);    //读取 2 个字节.
        ReadAddr+=2;//偏移 2 个字节.
    }
}
////////////////////////////////////////////////////////////////////////////////////////////
//WriteAddr:起始地址
//WriteData:要写入的数据
void Test_Write(u32 WriteAddr,u16 WriteData)
{
    STMFLASH_Write(WriteAddr,&WriteData,1);    //写入一个字
```

6.3 SRAM 存储器

STM32 片内自带 SRAM，通常不同型号的 STM32 的 SRAM 大小是不相同的。

6.3.1.SRAM 存放内容

（1）各个文件中声明和定义的全局变量、静态数据和常量

（2）HEAP 区

也称为堆区，一般由程序员使用 malloc 或 new 来进行分配，在适当的时候用 free 或 delete 来进行释放。若程序员不释放，程序结束时可能由操作系统回收。分配方式类似于数据结构中的链表。

（3）STACK 区

也称为栈区，由编译器自动分配和释放，程序员不做干涉。存放函数的参数值、局部变量的值等，其操作方式类似于数据结构中的栈。程序的中断，函数的形式参数传递等都需要 STACK 来实现。

6.3.2.外扩 SRAM 存储器

在使用单片机 STM32 时如果遇到数据内存不够使用，需要进行外扩 SRAM 存储器芯片，可以采用 IS62WV51216（EM681FV16BU-55LF）的 SRAM 存储器芯片作为存储的外扩芯片，首先实现 IS62WV51216（EM681FV16BU-55LF）的访问，需要对 FSMC 进行配置，步骤如下。

（1）使能 FSMC 时钟，并配置 FSMC 相关的 IO 及其时钟使能。

在使用 FSMC 前，先将其时钟开启。然后把 FSMC_D0~15，FSMCA0~18 等相关 IO 接口改为复用输出的配置，最后使能各 IO 组的时钟。

使能 FSMC 时钟的方法：

RCC_AHBPeriphClockCmd(RCC_AHBPeriph_FSMC,ENABLE);

（2）设置 FSMC BANK1 区域 3

FSMC BANK1 区域 3 已经包括设置区域 3 的 SRAM 存储器的位宽、工作模式和读写时序等等。启动使用模式 A、16 位宽，并让读写共同使用一个时序寄存器。

使用的函数是：

void FSMC_NORSRAMInit(FSMC_NORSRAMInitTypeDef* FSMC_NORSRAMInitStruct)

（3）使能 BANK1 区域 3。

使用的函数是：

void FSMC_NORSRAMCmd(uint32_t FSMC_Bank, FunctionalState NewState)；

以上三个步骤，就是完成对 FSMC 的配置设置，设置完毕后就可以访问 IS62WV51216（EM681FV16BU-55LF），需要注意的一点，由于选 BANK1 的 3 使用，因此 HADDR[27:26]=10，而外部内存的首地址设置为 0X68000000。

6.2.3.SRAM 备份

STM32 提供 4KB 的备份 SRAM，在开发程序时可以用于存储掉电不丢失的数据（需要 RTC 纽扣电池支持），特别是一些实时修改的，掉电不能丢失的数据，比如我用于存储雨量累计流量等实时变化的数据，定时存储到 flash，实时存储到备份区（不能频繁的写 flash），当备份区数据丢失了再从 flash 加载，否则每次都从备份区加载。

```
#include "BackupSRAM.h"
#include "system.h"
#include "string.h"
#define BACKUP_SRAM_SIZE          (4*1024)        //备份 SRAM 大小
/*****************************************************************************
*****************************************
* 函数：bool BackupSRAM_Init(void)
* 功能：备份域 SRAM 初始化
bool BackupSRAM_Init(void)
{
  RCC->APB1ENR|=1<<28;                          //使能电源接口时钟
  PWR->CR|=1<<8;                                //后备区域访问使能(RTC+SRAM)
  SYS_DeviceClockEnable(DEV_BKP,TRUE);          //备份区 SRAM 使能
  RCC->AHB1LPENR |= BIT18;                      //睡眠模式期间的备份 SRAM 接口时钟使能
PWR->CSR |= BIT9;    //使能备份调压器,不开启会导致备份 SRAM 掉电丢失-必须先使能时钟
  return TRUE;
}
/*****************************************************************************
*****************************************
* 函数：        u16 BackupSRAM_WriteData(u16 AddrOffset, u8 *pData, u16 DataLen)
* 功能：写入数据到备份 SRAM 中
* 参数：AddrOffset:地址偏移，0-4KB 范围；pData：要写入的数据；DataLen：要写入的数据长度
u16 BackupSRAM_WriteData(u16 AddrOffset, u8 *pData, u16 DataLen)
{
    u32 len;
    if(pData==NULL) return 0;                       //无效的地址
    if(DataLen==0) return 0;                        //无效的数量
    if(AddrOffset >= BACKUP_SRAM_SIZE) return 0;    //起始地址有误
```

```
    len = AddrOffset + DataLen;
    if(len > BACKUP_SRAM_SIZE) len = BACKUP_SRAM_SIZE;     //限制范围，只有 4KB
    len -= AddrOffset;                                     //计算要写入的数据长度
    memcpy((u8 *)BKPSRAM_BASE+AddrOffset, pData, DataLen);
    return len;
}
/*****************************************************************************
*****************************************
* 函数：       u16 BackupSRAM_ReadData(u16 AddrOffset, u8 *pData, u16 DataLen)
* 功能：       从备份 SRAM 中读取数据
* 参数：AddrOffset:地址偏移，0-4KB 范围；pData：要读取的数据缓冲区；DataLen：要读取的数据长度
*u16 BackupSRAM_ReadData(u16 AddrOffset, u8 *pData, u16 DataLen)
{
    u32 len;
    if(pData==NULL) return 0;                              //无效的地址
    if(DataLen==0) return 0;                                   //无效的数量
    if(AddrOffset >= BACKUP_SRAM_SIZE) return 0;               //起始地址有误
    len = AddrOffset + DataLen;
    if(len > BACKUP_SRAM_SIZE) len = BACKUP_SRAM_SIZE;     //限制范围，只有 4KB
    len -= AddrOffset;                                     //计算要写入的数据长度
    memcpy(pData, (u8 *)BKPSRAM_BASE+AddrOffset, DataLen);
    return len;
}
/*****************************************************************************
*****************************
* 文件名     ：BackupSRAM.h
* 功能：       STM32F4  备份域 SRAM 驱动
#ifndef __BACKUP_SRAM_H_
#define __BACKUP_SRAM_H_
#include "system.h"
bool BackupSRAM_Init(void);                    //备份域 SRAM 初始化
u16 BackupSRAM_WriteData(u16 AddrOffset, u8 *pData, u16 DataLen); //写入数据到备份 SRAM 中
u16 BackupSRAM_ReadData(u16 AddrOffset, u8 *pData, u16 DataLen); //从备份 SRAM 中读取数据
#endif //__BACKUP_SRAM_H_
```

6.4 SD 卡

6.4.1 SD 卡的应用

1.SD 卡

SD 卡也称安全数码卡， 它是在 MMC 的基础上发展而来， 是一种基于半导体快闪记忆器的新一代记忆设备。按容量分类，可以将 SD 卡分为 3 类： SD 卡、 SDHC 卡、SDXC 卡。SD 卡(SDSC)：0~2G SDHC 卡：2~32G SDXC 卡：32G~2T。

2.SD 卡操作模式

SD 卡一般有 SD 卡模式和 SPI 模式两种操作模式。SD 卡模式（通过 SDIO 通信）允许 4 线的高速数据传输，只能使用 3.3V 的 IO 电平；SPI 模式同 SD 卡模式相比就是丧失了速度，在 SPI 模式下， CS/MOSI/MISO/CLK 都需要加 10~100K 左右的上拉电阻。

3.SD 卡引脚功能

SD 卡引脚功能如表 6-2 所示。

表 6-2 SD 卡引脚功能表

针脚	1	2	3	4	6	7	8	9
SD 卡模式	CD/DAT3	CMD	VSS	VCC	VSS	DAT0	DAT1	DAT2
SPI 模式	CS	MOSI	VSS	VCC	VSS	MISO	NC	NC

4.SD 卡的寄存器

SD 卡的寄存器如表 6-3 所示。

表 6-3 SD 卡寄存器

名称	宽度	描述
CID	128	卡标识寄存器
RCA	16	相对卡地址寄存器：本地系统中卡的地址，动态变化，在卡的初始化时确定。
CSD	128	卡描述数据寄存器：卡操作条件相关的信息数据
SCR	64	SD 配置寄存器：SD 卡特定信息数据
OCR	32	操作条件寄存器

6.4.2 SD 卡驱动实现

1、SPI 模式下的主要操作命令

（1）卡的识别、初始化等基本命令集
CMD0:复位 SD 卡.
CMD1:读 OCR 寄存器.

CMD9:读 CSD 寄存器.
CMD10:读 CID 寄存器.
CMD12:停止读多块时的数据传输
CMD13:读 Card_Status 寄存器
（2）读卡命令集
CMD16:设置块的长度
CMD17:读单块.
CMD18:读多块,直至主机发送 CMD12 为止 .
（3）写卡命令集)
CMD24:写单块.
CMD25:写多块.
CMD27:写 CSD 寄存器 .
（4）擦除卡命令集
CMD32:设置擦除块的起始地址.
CMD33:设置擦除块的终止地址.
CMD38: 擦除所选择的块.
（5）写保护命令集
CMD28:设置写保护块的地址.
CMD29:擦除写保护块的地址.

2.SD 卡的初始化

（1）初始化硬件配置，SPI 配置，IO 配置等。
（2）上电延时。（>74CLK）
（3）复位卡。（CMD0）
（4）激活卡，内部初始化并获取卡的类型。
（5）查询 OCR，获取供电情况。
（6）是否使用 CRC（CMD59）。
（7）设置读写块数据长度（CMD16）。
（8）读取 CSD，获取存储卡的其他信息（CMD9）
（9）发送 8CLK 后，禁止片选。

3.SPI 模式下的读取数据过程，采用 CMD17 来实现

（1）发送 CMD17。
（2）接收卡响应 R1。
（3）接收数据起始令牌 0XFE。
（4）接收数据。
（5）接收两个字节的 CRC，如果没有开启 CRC。这两个字节在读取后可以丢掉。
（6）8CLK 之后禁止片选。

4.SPI 模式下的写数据过程，采用 CMD24 来实现

（1）发送 CMD24。
（2）接收卡响应 R1。
（3）接收数据起始令牌 0XFE。
（4）接收数据。
（5）发送两个字节的伪 CRC。
（6）8CLK 之后禁止片选。

5.代码实现

```
/*******************************************************************************
*****
* Function Name  : SPI_FLASH_Init
* Description    : Initializes the peripherals used by the SPI FLASH driver.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************
/
void SPI_SD_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStructure;

  /*  使能 SPI 对应引脚的时钟  使能 SPI1 的时钟  */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_SPI1, ENABLE);

  /*配置 SPI 的时钟线 SCK 和 SPI 的 MOSI 线和 SPI 的 MISO 线  */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用功能的推挽输出
  GPIO_Init(GPIOA, &GPIO_InitStructure);
  /*配置 SPI 的片选线：CSN */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_8;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //推挽输出
  GPIO_Init(GPIOA, &GPIO_InitStructure);
  /*  拉高 CSN 引脚，停止使能 SD*/
  GPIO_SetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_8);
  GPIO_SetBits(GPIOA,GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7);
      //  配置 SPI,使它适合 SD 的特性
  SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;  //双线双向全双工
  SPI_InitStructure.SPI_Mode = SPI_Mode_Master;  //主器件
```

```
 SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;//8 位数据长度
 SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;  //时钟悬空时为高
 SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge; //数据捕获于第 2 个时钟沿
 SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;    //NSS 信号由外部管脚管理
 SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_256;//波特率预分频值为 4
 SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB; //数据传输的第一个字节为 MSB
 SPI_InitStructure.SPI_CRCPolynomial = 7;  //CRC 的多项式
 SPI_Init(SPI1, &SPI_InitStructure);
 /*  使能 SPI1  */
 SPI_Cmd(SPI1, ENABLE);
}
     /*******************************************************************************
*****
* Function Name  : SPI_FLASH_SendByte
* Description    :  发送一个数据，同时接收从 FLASH 返回来的数据
* Input          : byte : byte to send.
* Output         : None
* Return         : The value of the received byte.
*******************************************************************************
/
u8 SPIx_ReadWriteByte(u8 byte)
{
 /*  等待数据发送寄存器清空  */
 while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
      /*  通过 SPI 发送出去一个字节数据  */
 SPI_I2S_SendData(SPI1, byte);
      /*  等待接收到一个数据（接收到一个数据就相当于发送一个数据完毕）  */
 while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET);
      /*  返回接收到的数据  */
 return SPI_I2S_ReceiveData(SPI1);
}
     /*******************************************************************************
*****
* Function Name  : SPI_FLASH_SendHalfWord
* Description    :  发送并接受一个半字数据（16 位）
* Input          : Half Word : Half Word to send.
* Output         : None
* Return         : The value of the received Half Word.
*******************************************************************************
/
u16 SPIx_ReadWriteHalfWord(u16 HalfWord)
{
 /*  等待数据发送寄存器清空  */
 while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
```

```
        /* 通过 SPI 发送出去半个字的数据 */
  SPI_I2S_SendData(SPI1, HalfWord);
        /* 等待接收到一个半字数据（接收到一个数据就相当于发送一个数据完毕） */
  while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET);
        /* 返回接收到的数据 */
  return SPI_I2S_ReceiveData(SPI1);
}
//SPI 速度设置函数
//SpeedSet:
//SPI_BaudRatePrescaler_2   2 分频   (SPI 36M@sys 72M)
//SPI_BaudRatePrescaler_8   8 分频   (SPI 9M@sys 72M)
//SPI_BaudRatePrescaler_16  16 分频  (SPI 4.5M@sys 72M)
//SPI_BaudRatePrescaler_256 256 分频  (SPI 281.25K@sys 72M)

void SPIx_SetSpeed(u8 SpeedSet)
{
SPI_InitStructure.SPI_BaudRatePrescaler = SpeedSet ;
   SPI_Init(SPI1, &SPI_InitStructure);
SPI_Cmd(SPI1,ENABLE);
}
      /*******************************END                                                    OF
INIT_SPI*******************************/

/*******************************START OF SD_OPERATION**********************/
/*****************************************************************************
*
* 函数名称        : SD_Select
* 功能描述        : 选择 SD 卡，并等待 SD 卡准备好
* 进入参数        : 无.
* 返回参数        : 0：成功        1：失败
* 备注说明        : SD 卡准备好会返回 0XFF
*****************************************************************************
/
u8 SD_Select(void)
{
uint32_t t=0;
SD_CS(OFF);  //片选 SD，低电平使能
do
{
  if(SD_SPI_ReadWriteByte(0XFF)==0XFF)return 0;//OK
  t++;
}while(t<0XFFFFFF);//等待
SD_DisSelect();  //释放总线
return 1;//等待失败
```

```
}
/*******************************************************************************
*
* 函数名称        : SD_RecvData
* 功能描述        : 从 sd 卡读取一个数据包的内容
* 进入参数        : buf：数据缓存数组      len 要读取的数据的长度
* 返回参数        : 0：成功        其他：失败
* 备注说明        : 读取时需要等待 SD 卡发送数据起始令牌 0XFE
*******************************************************************************
/
u8 SD_RecvData(u8*buf,u16 len)
{
u16 Count=0xF000;//等待次数
while  ((SD_SPI_ReadWriteByte(0XFF)!=0xFE)&&Count)Count--;//等待得到读取数据令牌
0xfe
if     (Count==0)    return MSD_RESPONSE_FAILURE;//获取令牌失败,返回 0XFF
    while(len--)//开始接收数据
    {
       *buf=SPIx_ReadWriteByte(0xFF);
       buf++;
    }
    //下面是 2 个伪 CRC（dummy CRC），假装接收了 2 个 CRC
    SD_SPI_ReadWriteByte(0xFF);
    SD_SPI_ReadWriteByte(0xFF);
    return 0;//读取成功
}
/*******************************************************************************
*
* 函数名称        : SD_SendBlock
* 功能描述        : 向 sd 卡写入一个数据包的内容 512 字节
* 进入参数        : buf:数据缓存区     cmd:数据发送的令牌
* 返回参数        : 0：成功        其他：失败
* 备注说明        : 写数据时需要先发送数据起始令牌 0XFE/0XFC/0XFD
*******************************************************************************
/
u8 SD_SendBlock(u8*buf,u8 cmd)
{
u32 t,Count=0XFFFFFF;
while ((SD_SPI_ReadWriteByte(0XFF)!=0xFF)&&Count)Count--;//等待 SD 卡准备好
if     (Count==0)    return MSD_RESPONSE_FAILURE;//SD 卡未准备好，失败，返回
SD_SPI_ReadWriteByte(cmd); //发送数据起始或停止令牌
if(cmd!=0XFD)//在不是结束令牌的情况下，开始发送数据
{
  for(t=0;t<512;t++)SPIx_ReadWriteByte(buf[t]);//提高速度,减少函数传参时间
```

```
    SD_SPI_ReadWriteByte(0xFF);//发送 2 字节的 CRC
    SD_SPI_ReadWriteByte(0xFF);
 t=SD_SPI_ReadWriteByte(0xFF);//紧跟在 CRC 之后接收数据写的状态
 if((t&0x1F)!=0x05)return MSD_DATA_WRITE_ERROR;//写入错误
}
    return 0;//写入成功
}
/*******************************************************************************
*
* 函数名称        : SD_SendCmd
* 功能描述        : 向 sd 卡写入一个数据包的内容 512 字节
* 进入参数        : cmd：命令 arg：命令参数 crc：crc 校验值及停止位
* 返回参数        : 返回值:SD 卡返回的对应相应命令的响应
* 备注说明        : 响应为 R1-R7，见 SD 协议手册 V2.0 版（2006）
*******************************************************************************
/
u8 SD_SendCmd(u8 cmd, u32 arg, u8 crc)
{
    u8 r1;
u8 Retry=0;
SD_DisSelect();//取消上次片选释放总线
if(SD_Select())return 0XFF;//检查片选信号线是否选择成功
//发送
    SD_SPI_ReadWriteByte(cmd | 0x40);//分别写入命令
    SD_SPI_ReadWriteByte(arg >> 24);
    SD_SPI_ReadWriteByte(arg >> 16);
    SD_SPI_ReadWriteByte(arg >> 8);
    SD_SPI_ReadWriteByte(arg);
    SD_SPI_ReadWriteByte(crc);
if(cmd==CMD12)SD_SPI_ReadWriteByte(0xff);//Skip a stuff byte when stop reading
    //等待响应，或超时退出
Retry=0X1F;
do //发送一定数量的时钟信号，等待 SD 卡回应 0X01（0x01 表示命令发送成功，回复 0XFF
表示失败）
{
 r1=SD_SPI_ReadWriteByte(0xFF);
}while((r1&0X80) && Retry--); //等待返回非 0XFF 的数据
//返回状态值
    return r1;
}
/*******************************************************************************
*
* 函数名称        : SD_GetCID
* 功能描述        : 获取 SD 卡的 CID 信息，包括制造商信息
```

```
* 进入参数       : cid_data(存放 CID 的内存，至少 16Byte
* 返回参数       : 0：成功       其他：失败
* 备注说明       : CID 寄存器内容详见 SD 协议手册 V2.0 版（2006）
*******************************************************************************
/
u8 SD_GetCID(u8 *cid_data)
{
    u8 r1;
    //发 CMD10 命令，读 CID
    r1=SD_SendCmd(CMD10,0,0x01);
    if(r1==0x00)
{
 r1=SD_RecvData(cid_data,16);//接收 16 个字节的数据
    }
SD_DisSelect();//取消片选
if(r1)return 1;
else return 0;
}
/*******************************************************************************
*
* 函数名称       : SD_GetCSD
* 功能描述       : 获取 SD 卡的 CSD 信息，包括容量和速度信息
* 进入参数       : cid_data(存放 CSD 的内存，至少 16Byte
* 返回参数       : 0：成功       其他：失败
* 备注说明       : CSD 寄存器内容详见 SD 协议手册 V2.0 版（2006）
*******************************************************************************
/
u8 SD_GetCSD(u8 *csd_data)
{
    u8 r1;
    r1=SD_SendCmd(CMD9,0,0x01);//发 CMD9 命令，读 CSD
    if(r1==0)
{
    r1=SD_RecvData(csd_data,16);//接收 16 个字节的数据
    }
SD_DisSelect();//取消片选
if(r1)return 1;
else return 0;
}
/*******************************************************************************
*
* 函数名称       : SD_GetSectorCount
* 功能描述       : 获取 SD 卡的总扇区数（扇区数
* 进入参数       : cid_data(存放 CSD 的内存，至少 16Byte
```

```
* 返回参数        : 0：获取容量出错        其他：SD 卡的扇区数量值
* 备注说明        : SD 卡的容量的计算公式 SD 协议手册 V2.0 版（2006）
*******************************************************************************
/
u32 SD_GetSectorCount(void)
{
    u8 csd[16];
    u32 Capacity;
    u8 n;
u16 csize;
//取 CSD 信息，如果期间出错，返回 0
    if(SD_GetCSD(csd)!=0) return 0;  //获取容量失败
    //如果为 SDHC 卡，按照下面方式计算
    if((csd[0]&0xC0)==0x40)  //V2.00 的卡
    {
  csize = csd[9] + ((u16)csd[8] << 8) + 1;
  Capacity = (u32)csize << 10;//得到扇区数
    }
else//V1.XX 的卡
    {
  n = (csd[5] & 15) + ((csd[10] & 128) >> 7) + ((csd[9] & 3) << 1) + 2;
  csize = (csd[8] >> 6) + ((u16)csd[7] << 2) + ((u16)(csd[6] & 3) << 10) + 1;
  Capacity= (u32)csize << (n - 9);//得到扇区数
    }
    return Capacity;
}
/*******************************************************************************
*
* 函数名称        : SD_ReadDisk
* 功能描述        : 读 SD 卡
* 进入参数        : buf:数据缓存区        sector:欲读取的地址      cnt:欲读取的扇区数
* 返回参数        : 0：成功        其他：失败
* 备注说明        : 1.读取的地址必须是一个扇区的起始
    2.必须是 SD2.0 卡，其他的卡不处理
*******************************************************************************
/
u8 SD_ReadDisk(u8*buf,u32 sector,u8 cnt)
{
u8 r1;
if(cnt==1)
{
  r1=SD_SendCmd(CMD17,sector,0X01);//读命令
  if(r1==0)//指令发送成功
  {
```

```
   r1=SD_RecvData(buf,512);//接收 512 个字节
  }
}
else
{
 r1=SD_SendCmd(CMD18,sector,0X01);//连续读命令
 do
 {
   r1=SD_RecvData(buf,512);//接收 512 个字节
   buf+=512;
 }while(--cnt && r1==0);
 SD_SendCmd(CMD12,0,0X01); //发送停止命令
}
SD_DisSelect();//取消片选
return r1;//
}
/*****************************************************************************
*
* 函数名称        : SD_WriteDisk
* 功能描述        : 写 SD 卡
* 进入参数        : buf:数据缓存区        sector:待写的地址       cnt:待写的扇区数
* 返回参数        : 0：成功        其他：失败
* 备注说明        : 1.写的地址必须是一个扇区的起始
    2.必须是 SD2.0 卡，其他的卡不处理
*****************************************************************************
/
u8 SD_WriteDisk(u8*buf,u32 sector,u8 cnt)
{
u8 r1;
if(cnt==1)
{
 r1=SD_SendCmd(CMD24,sector,0X01);//单个扇区写命令
 if(r1==0)//指令发送成功
 {
   r1=SD_SendBlock(buf,0xFE);//写 512 个字节
 }
}
else
{
 if(SD_Type!=SD_TYPE_MMC)
 {
   SD_SendCmd(CMD55,0,0X01);
   SD_SendCmd(CMD23,cnt,0X01);//发送待写入的扇区的数量，此命令用来预擦除所有待写
入的扇区
```

```
  }
   r1=SD_SendCmd(CMD25,sector,0X01);//连续写命令，发送起始地址
 if(r1==0)
 {
  do
  {
   r1=SD_SendBlock(buf,0xFC);//发送 512 个字节
   buf+=512;
  }while(--cnt && r1==0);
  r1=SD_SendBlock(0,0xFD);//发送停止位
 }
}
SD_DisSelect();//取消片选
return r1;//
}
/*******************************************************************************
*
* 函数名称        : SD_Initialize
* 功能描述        : 写 SD 卡
* 进入参数        : 无
* 返回参数        : 0：成功        其他：失败
* 备注说明        : 1.写的地址必须是一个扇区的起始
    2.必须是 SD2.0 卡，其他的卡不处理
*******************************************************************************
/
u8 SD_Initialize(void)
{
    u8 r1;      // 存放 SD 卡的返回值
    u16 retry;  // 用来进行超时计数
    u8 buf[4];
u16 i;
     SPI_SD_Init();  //初始化 IO
 SD_SPI_SpeedLow(); //设置到低速模式
for(i=0;i<10;i++)SD_SPI_ReadWriteByte(0XFF); //发送最少 74 个脉冲,此时保持片选线是高电平
retry=20;
do
{
 r1=SD_SendCmd(CMD0,0,0x95);  //进入复位，同时选中了 SPI 模式（发送 CMD0 时，CSN 为低电平）
}while((r1!=0X01) && retry--);
 SD_Type=0; //默认无卡
if(r1==0X01)
{
```

```
 if(SD_SendCmd(CMD8,0x1AA,0x87)==1) //利用 V2.0 版 SD 卡特有的命令 CMD8 检查是否为 2.0 卡
 {
  for(i=0;i<4;i++)buf[ i]=SD_SPI_ReadWriteByte(0XFF); //Get trailing return value of R7 resp
  if(buf[2]==0X01&&buf[3]==0XAA) //卡是否支持 2.7~3.6V
  {
   retry=0XFFFE;
   do
   {
   SD_SendCmd(CMD55,0,0X01); //发送 CMD55
   r1=SD_SendCmd(CMD41,0x40000000,0X01); //发送 CMD41
   }while(r1&&retry--);
   if(retry&&SD_SendCmd(CMD58,0,0X01)==0) //鉴别 SD2.0 卡版本,读取 OCR 的值
   {
   for(i=0;i<4;i++)buf[ i]=SD_SPI_ReadWriteByte(0XFF); //得到 OCR 值
   if(buf[0]&0x40)SD_Type=SD_TYPE_V2HC;    //检查 CCS (第 30 位)
   else SD_Type=SD_TYPE_V2;
   }
  }
 }
 else   //不是 2.0 卡的情况下，检查是否为 1.0 卡或者 mmc 卡
 {
  SD_SendCmd(CMD55,0,0X01);  //发送 CMD55
  r1=SD_SendCmd(CMD41,0,0X01); //发送 CMD41
  if(r1<=1)//发送 CMD55 和 CMD41 成功，表示这是 1.0 卡
  {
   SD_Type=SD_TYPE_V1;
   retry=0XFFFE;
   do //等待退出 IDLE 模式
   {
   SD_SendCmd(CMD55,0,0X01); //发送 CMD55
   r1=SD_SendCmd(CMD41,0,0X01); //发送 CMD41  进行初始化
   }while(r1&&retry--);
  }
  else //不是 1.0 卡，则考虑是 MMC 卡
  {
   SD_Type=SD_TYPE_MMC; //先假设是 MMC 卡
   retry=0XFFFE;
   do //等待退出 IDLE 模式
   {
   r1=SD_SendCmd(CMD1,0,0X01); //发送 CMD1，利用复位功能判断是否为 MMC 卡
   }while(r1&&retry--); //发送复位命令，超时则复位失败
  }
  if(retry==0||SD_SendCmd(CMD16,512,0X01)!=0)SD_Type=SD_TYPE_ERR;  //MMC 卡复位
```

```
失败
 }
}
SD_DisSelect(); //取消片选
SD_SPI_SpeedHigh(); //高速
if(SD_Type)return 0;   //初始化成功
else if(r1)return r1;   //初始化失败
return 0xaa;//其他错误
```

6.5 FATFS 文件系统

6.5.1 FATFS 文件系统介绍

1.FATFS 文件系统

FATFS 是一个完全免费开源的 FAT 文件系统模块，专门为小型的嵌入式系统而设计。完全用标准 C 语言编写，所以具有良好的硬件平台独立性。可以移植到 8051、PIC、AVR、SH、Z80、H8、ARM 等系列单片机上而只需做简单的修改。它支持 FATl2、FATl6 和 FAT32，支持多个存储媒介；有独立的缓冲区，可以对多个文件进行读 / 写，并特别对 8 位单片机和 16 位单片机做了优化。

2.FATFS 文件系统的特点

（1）Windows 兼容的 FAT 文件系统（支持 FAT12/FAT16/FAT32）
（2）与平台无关，移植简单。全 C 语言编写。
（3）代码量少、效率高。
（4）多种配置选项：支持多卷（物理驱动器或分区，最多 10 个卷）；多个 ANSI/OEM 代码页包括 DBCS；支持长文件名、ANSI/OEM 或 Unicode；支持 RTOS；支持多种扇区大小；只读、最小化的 API 和 I/O 缓冲区等。

3.FATFS 文件的结构

FATFS 文件的结构主要分为底层接口、中间层 FATFS 模块和应用层三部分。
（1）底层接口
包括存储媒介读 / 写接口（disk I/O）和供给文件创建修改时间的实时时钟，需要我们根据平台和存储介质编写移植代码。
（2）中间层 FATFS 模块
实现了 FAT 文件读 / 写协议。FATFS 模块提供的是 ff.c 和 ff.h。除非有必要，使用者一般不用修改，使用时将头文件直接包含进去即可。
（3）应用层
使用者无需理会 FATFS 的内部结构和复杂的 FAT 协议，只需要调用 FATFS 模块提供给用户的一系列应用接口函数，如 f_open，f_read，f_write 和 f_close 等，就可以像在 PC 上读 / 写文件那样简单。

4.FATFS 文件系统包

（1）diskio.c 和 diskio.h 是硬件层，负责与底层硬件接口适配。
（2）ff.c 和 ff.h 是 FatFs 的文件系统层和文件系统的 API 层。
（3）FATFS 模块在移植的时候，我们一般只需要修改 2 个文件，即 ffconf.h 和 diskio.c。
（4）FATFS 模块的所有配置项都是存放在 ffconf.h 里面，通过配置里面的一些选项，来满足用户的需求。

6.5.2 FATFS 文件系统驱动实现

1.实现 disk_initialize()函数

该函数在挂载文件系统的时候会被调用，主要是实现读写 SD 卡前对 SD 卡进行初始化，根据 SD 卡的传输协议，我们按照如下步骤初始化 SD 卡。
（1）判断 SD 卡是否插入，可以通过检查 SD 卡卡座的 CD 脚电平进行判断，一般插入卡后该引脚会变成低电平。
（2）稍微延时一段时间后发送至少 74 个时钟给 SD 卡。
（3）发送 CMD0 命令给 SD 卡，直到 SD 卡返回 0x01 为止，这里可以循环多次发送
程序如下：

```
 /* Start send CMD0 till return 0x01 means in IDLE state */
    for(retry=0; retry<0xFFF; retry++)
     {
        r1 = MSD0_send_command(CMD0, 0, 0x95);
          if(r1 == 0x01)
            {
              retry = 0;
              break;
            }
      }
```

（4）发送 CMD8 获取卡的类型，不同类型的卡其初始化方式有所不同。
（5）根据卡的类型对卡进行初始化。
实现后的程序如下：

```
DSTATUS disk_initialize (BYTE drv/* Physical drive nmuber (0..) */)
{
      int Status;
      switch (drv)
        {
          case 0 :
              Status = MSD0_Init();
              if(Status==0)
{
                 return RES_OK;
                }
else
```

```
{
                  return STA_NOINIT;
                }
            case 1 :
              return RES_OK;
            case 2 :
               return RES_OK;
            case 3 :
              return RES_OK;
            default:
              return STA_NOINIT;
        }
   }
```

2.实现 disk_read()函数

该函数是读取 SD 卡扇区数据的函数，根据 SD 卡数据传输协议可知有读取单扇区和读取多扇区两种操作模式，为提高读文件的速度应该实现读取多扇区函数。

程序如下：

```
DRESULT disk_read ( BYTE drv, /* Physical drive nmuber (0..) */
            BYTE *buff, /* Data buffer to store read data */
              DWORD sector, /* Sector address (LBA) */
            BYTE count      /* Number of sectors to read (1..255) */
            )
  {
    int Status;
    if( !count )
      {
       return RES_PARERR;  /* count 不能等于 0，否则返回参数错误  */
      }
    switch (drv)
     {
       case 0:
         if(count==1)       /* 1 个 sector 的读操作  */
          {
            Status =MSD0_ReadSingleBlock( sector ,buff );
              if(Status == 0)
{
                  return RES_OK;
                }
else
{
                  return RES_ERROR;
                }
```

```
            }
          else          /*  多个 sector 的读操作  */
            {
                    Status = MSD0_ReadMultiBlock( sector , buff ,count);
              if(Status == 0)
{
                return RES_OK;
                 }
else
{
                  return RES_ERROR;
                }
              }
        case 1:
          if(count==1)      /* 1 个 sector 的读操作  */
            {
              return RES_OK;
            }
          else          /*  多个 sector 的读操作  */
            {
              return RES_OK;
            }

      default:
         return RES_ERROR;
      }
 }
```

3.实现 disk_write()函数

该函数主要实现对 SD 卡进行写数据操作，和读数据操作一样也分单块写和多块写，建议实现多块写的方式，这样可以提高写数据速度。

程序如下：

```
DRESULT disk_write (BYTE drv, /* Physical drive nmuber (0..) */
            const BYTE *buff,   /* Data to be written */
            DWORD sector,      /* Sector address (LBA) */
            BYTE count    /* Number of sectors to write (1..255) */
              )
  {
    int Status;
    if( !count )
      {
      return RES_PARERR;  /* count 不能等于 0，否则返回参数错误  */
      }
```

```
        switch (drv)
          {
           case 0:
             if(count==1)        /* 1 个 sector 的写操作  */
               {
                 Status = MSD0_WriteSingleBlock( sector , (uint8_t *)(&buff[0]) );
                if(Status == 0)
{
                   return RES_OK;
                 }
else
{
                   return RES_ERROR;
                 }
                }
         else             /*  多个 sector 的写操作  */
            {
              Status = MSD0_WriteMultiBlock( sector , (uint8_t *)(&buff[0]) , count );
              if(Status == 0)
                                        {
                 return RES_OK;
                  }
else
{
                   return RES_ERROR;
                  }
          }
    case 1:
        if(count==1)        /* 1 个 sector 的写操作  */
        {
           return RES_OK;
         }
       else               /*  多个 sector 的写操作  */
         {
            return RES_OK;
         }
    default:return RES_ERROR;
   }
}
```

4.实现 disk_ioctl()函数

该函数在磁盘格式化、获取文件系统信息等操作时会被调用。
程序如下：

```
DRESULT disk_ioctl ( BYTE drv,       /* Physical drive nmuber (0..) */
                BYTE ctrl,     /* Control code */
               void *buff     /* Buffer to send/receive control data */
                 )
  {
    if (drv==0)
        {
         MSD0_GetCardInfo(&SD0_CardInfo);
         switch (ctrl)
           {
              case CTRL_SYNC :
                   return RES_OK;
              case GET_SECTOR_COUNT :
        *(DWORD*)buff = SD0_CardInfo.Capacity/SD0_CardInfo.BlockSize;
                  return RES_OK;
              case GET_BLOCK_SIZE :
                   *(WORD*)buff = SD0_CardInfo.BlockSize;
                   return RES_OK;
              case CTRL_POWER :
                    break;
              case CTRL_LOCK :
                   break;
              case CTRL_EJECT :
                break;
/* MMC/SDC command */
              case MMC_GET_TYPE :
                   break;
              case MMC_GET_CSD :
                   break;
              case MMC_GET_CID :
                   break;
              case MMC_GET_OCR :
                   break;
              case MMC_GET_SDSTAT :
                 break;
           }
         }
else if(drv==1)
{
              switch (ctrl)
                {
                  case CTRL_SYNC :
                       return RES_OK;
                  case GET_SECTOR_COUNT :
```

```
                    return RES_OK;
                  case GET_SECTOR_SIZE :
                    return RES_OK;
                  case GET_BLOCK_SIZE :
                    return RES_OK;
                    case CTRL_POWER :
                   break;
                  case CTRL_LOCK :
                      break;
                  case CTRL_EJECT :
                   break;
    /* MMC/SDC command */
                  case MMC_GET_TYPE :
                       break;
                  case MMC_GET_CSD :
                    break;
                  case MMC_GET_CID :
                    break;
                  case MMC_GET_OCR :
                    break;
                        case MMC_GET_SDSTAT :
                    break;
                }
              }
          else
    {
              return RES_PARERR;
                }
```

5.文件系统测试

（1）测试写文件

测试代码如下：

```
    printf("write file test.....\n\r");
    res = f_open(&fdst, "0:/test.txt", FA_CREATE_ALWAYS | FA_WRITE);
   if(res != FR_OK)
   {
            printf("open file error : %d\n\r",res);
     }
   else
   {
   res = f_write(&fdst, textFileBuffer, sizeof(textFileBuffer), &bw);        /* Write it to the dst
file */
            if(res == FR_OK)
```

```
                {
                    printf("write data ok! %d\n\r",bw);

                 }
                else
                {
                     printf("write data error : %d\n\r",res);
                }
                   /*close file */
                      f_close(&fdst);
          }
```

（2）测试读文件

```
         printf("read file test.....\n\r");
          res = f_open(&fsrc, "0:/test.txt", FA_OPEN_EXISTING | FA_READ);
            if(res != FR_OK)
            {
                 printf("open file error : %d\n\r",res);
             }
            else
            {
                 res = f_read(&fsrc, buffer, sizeof(textFileBuffer), &br);      /* Read a chunk of src
file */
                 if(res==FR_OK)
                 {
                     printf("read data num : %d\n\r",br);
                     printf("%s\n\r",buffer);
                 }
                 else
                 {

                      printf("read file error : %d\n\r",res);
                 }
          /*close file */
         f_close(&fsrc);
  }
```

本章小结

（1）存储结构：STM32 有四种存储单元，依次是 SRAM、Flash、FSMC 和 AHB 到 APB 桥（挂载各种外设）

（2）存储组织：程序存储器、数据存储器、寄存器和输入输出端口被组织在同一个 4GB 的线性地址空间内。数据字节以小端格式存放在存储器中。一个字里的最低地址字节被认为是该字的最低有效字节，而最高地址字节是最高有效字节。

（3）FLASH 存储器的构成： 包含主存储器、系统存储器、 OTP 区域以及选项字节区域。

（4）Flash 的写过程：解锁、数据操作位数、擦除扇区、写入数据。
（5）SRAM 存放内容：各个文件中声明和定义的全局变量、静态数据和常量；HEAP 区；STACK 区。
（6）外扩 SRAM 存储器：使能 FSMC 时钟，并配置 FSMC 相关的 IO 及其时钟使能；设置 FSMC BANK1 区域 3；使能 BANK1 区域 3。
（7）SD 卡类：SD 卡、SDHC 卡、SDXC 卡。SD 卡(SDSC)：0~2G SDHC 卡：2~32G SDXC 卡：32G~2T。
（8）SD 操作模式：SD 卡模式和 SPI 模式两种操作模式。SD 卡模式（通过 SDIO 通信）允许 4 线的高速数据传输，只能使用 3.3V 的 IO 电平；SPI 模式同 SD 卡模式相比就是丧失了速度，在 SPI 模式下，CS/MOSI/MISO/CLK 都需要加 10~100K 左右的上拉电阻。
（9）SD 卡初始化：硬件配置，SPI 配置，IO 配置等；上电延时。(>74CLK)；复位卡(CMD0)；激活卡，内部初始化并获取卡的类型；查询 OCR，获取供电情况；是否使用 CRC(CMD59)；设置读写块数据长度（CMD16）；读取 CSD，获取存储卡的其他信息（CMD9）；发送 8CLK 后，禁止片选。
（10）SD 卡读数据：发送 CMD17；接收卡响应 R1；接收数据起始令牌 0XFE；接收数据；接收两个字节的 CRC，如果没有开启 CRC。这两个字节在读取后可以丢掉；8CLK 之后禁止片选。
（11）SD 卡写数据：发送 CMD24；接收卡响应 R1；接收数据起始令牌 0XFE；接收数据发送两个字节的伪 CRC；8CLK 之后禁止片选。
（12）FATFS 文件的结构：主要分为底层接口、中间层 FATFS 模块和应用层三部分。
（13）FATFS 文件系统包：diskio.c 和 diskio.h 是硬件层，负责与底层硬件接口适配；ff.c 和 ff.h 是 FatFs 的文件系统层和文件系统的 API 层；FATFS 模块在移植的时候，我们一般只需要修改 2 个文件，即 ffconf.h 和 diskio.c；FATFS 模块的所有配置项都是存放在 ffconf.h 里面，通过配置里面的一些选项，来满足用户的需求。

习 题

6.1 简述 STM32 的存储器组织。
6.2 简述 Flash 存储器的构成及各部分的作用。
6.3 简述 Flash 存储器写数据的过程。
6.4 简述外扩存储器的过程。
6.5 简述 SD 卡初始化的过程。
6.6 简述 FATFS 文件系统包的作用。