```
#include "n32g45x.h"                    // Device header
#include "SPI.h"
#include "USART_SendString.h"
#include "Delay.h"
#include "SD.h"
// 新增:包含memset所属的头文件
#include <string.h>

//定义所有出现的参数
//定义卡类别参数
uint8_t Card = 0;
//定义R1响应
uint8_t Return_R1;
//定义一个数组储存返回的参数值(供除CMD17,CMD18外的指令临时存储数据使用)(可以包含R1响应,CRC16,CRC7校验码)
uint8_t Return_Data[16 + 1 + 2 + 1];
//定义一个数组储存返回的扇区值(供CMD17指令临时存储数值使用)(含R1响应,CRC16,CRC7校验码)
uint8_t Return_Single_Sector[512 + 1 + 2 + 1];
//定义CMD8返回的基于uint32_t类型参数,
uint32_t CMD_Coefficient;
//定义uint16_t类型参数RCA
uint16_t RCA_Coefficient;
//定义基于uint16_t的RCA错误位
uint16_t RCA_FlagStatus;

/*
Definition:   初始化SD V2.0及之后版本SD卡
*/
uint8_t SD_V2_0_Init(void);
/*
Definition:   初始化SD V1.0及之后版本SD卡
*/
uint8_t SD_V1_0_Init(void);
/*
Progress: 修改ACMD41代码发送系数(改为0x40000000,即bit31(HCS位)设置为1)

*/

//优先定义SPI结构体
//允许在后续写入数值时对SPI结构体进行修改
SPI_InitType SPI_InitStructure;

/*
Definition:   初始化SD卡模块,返回基于uint8_t类型的卡种类参数(供后续使用)
Description:  8/21:初始化囊括分析SD卡种类特征(即分析SD卡为1.x, 2.0+)和容积模式(标准Standard,高容积High Capacity)
Notice:       默认电压为3.3V,所有的判断机制基于电压是否在3.3V及其大小10%范围内
*/
uint8_t SD_Init(void){

        SPI_Hardware_Init(SPI_InitStructure);
        //发送74个时钟脉冲
        SPI_Hardware_SendPulse();
        Send_VarString("Clock Pulse Delivered\r\n");

        //发送CMD0
        SPI_Hardware_SendCMD(CMD0, CMD0_Index, CMD0_CSC, Return_Data, NULL);
        if(Return_Data[0] != In_Idle_State)
                return Initialize_Failed;


        //发送CMD8
        SPI_Hardware_SendCMD(CMD8, CMD8_Index, CMD8_CSC, Return_Data, NULL);

        //第一轮判断(R1是否返回0x01)
        //定义卡种类
    uint8_t Card_Type;
        if(Return_Data[0] != In_Idle_State){
                Send_VarString("This might be SD_V1.X card\r\n");
```

```
                Card_Type = SD_V1_0_Init();
        }
        else{
                Send_VarString("This might be SD_V2.0 card\r\n");
                //将接收的数值返回
                CMD_Coefficient = (Return_Data[1] << 24) |
                                  (Return_Data[2] << 16) |
                                  (Return_Data[3] << 8) |
                                  (Return_Data[4]);
                if(CMD_Coefficient != CMD8_Index)
                        return Invalid_Card;

                Card_Type = SD_V2_0_Init();
        }
        //(8.27 修订: 添加CMD16机制)
        //强制复位为512个字节
        uint8_t SD_ReadWriteLength;
        SPI_Hardware_SendCMD(CMD16, (uint32_t)Sector_Length, OTHER_CSC, &SD_ReadWriteLength, NULL);
        if(SD_ReadWriteLength != Card_Ready)
                return Invalid_Card;
        return Card_Type;
}


//SD V2.0初始化代码
uint8_t SD_V2_0_Init(void){
    Card = 1;

        //8.23 修改:循环发送CMD55和ACMD41

        //等待SD卡退出IDLE状态(返回0x00)
        //8/22 修正:需要循环发送ACMD41以获取Return_Data[0](R1)
        //8.23 修正:添加判断机制,尝试让SD卡在不同的HCS位下进行初始化
        //          (SD卡在HCS置高的情况下可能无法初始化成功)
        //8.23 修正:修改为do-while循环
        //发送ACMD41


        //判断是否出现ERR
        uint16_t Time_out = 500;
        uint8_t R1;
        do{
                R1 = SPI_Hardware_SendACMD(Card);
                Delay_ms(10);

        }while((--Time_out) && R1 == 0x01);


        if(R1 != In_Idle_State && R1 != Card_Ready){
                Send_VarString("%d\r\n", R1);
                Send_VarString("Can't Initialize card \r\n");
                return Initialize_Failed;
        }
        else if(R1 != Card_Ready){
                Send_VarString("%d\r\n", R1);
                Send_VarString("Card cannot jump out from the idle state\r\n");
                return Initialize_Failed;
        }

        //再次发送CMD58
        SPI_Hardware_SendCMD(CMD58, CMD58_Index, OTHER_CSC, Return_Data, NULL);
        Send_VarString("%d\r\n", Return_Data[0]);

        if(Return_Data[0] != Card_Ready)
                return Initialize_Failed;
        else{
                Send_VarString("CMD58 delivered successfully \r\n");
                CMD_Coefficient = (Return_Data[1] << 24) |
                                  (Return_Data[2] << 16) |
```

```
                                (Return_Data[3] << 8) |
                                (Return_Data[4]);
                if(!(CMD_Coefficient & SD_V2_0_High_Capacity_CCS))
                        return SD_Ver2_0_Standard_Capacitance;
                else
                        return SD_Ver2_0_High_Capacitance;
        }
}


//SD V1.0初始化代码
uint8_t SD_V1_0_Init(void){

        uint16_t Time_out = 500;
        //等待SD卡退出IDLE状态(返回0x00)
        do{
                SPI_Hardware_SendACMD(Card);
                Delay_ms(10);
        }while((--Time_out) && (Return_Data[0] == 0x01));

        //判断其他位是否被设置
        if((Return_Data[0] & 0xFF) != 0x00)
                return Initialize_Failed;

        return SD_Ver1_x_Standard_Capacitance;
}


/*
Definition:    SD卡获取CID/CSD模块,返回状态(慢速模式)
Description:
*/
uint8_t SD_GetCSD(uint8_t* CSD_Coefficients){
        //发送CMD9,获取CSD参数
        SPI_Hardware_SendCMD(CMD9, 0x00000000, OTHER_CSC, CSD_Coefficients, NULL);
        Send_VarString("Get CSD finished\r\n");
        if(CSD_Coefficients[0] != Card_Ready)
                return Get_CIDCSD_Failed;

        return Get_CIDCSD_Success;
}

uint8_t SD_GetCID(uint8_t* CID_Coefficients){
        //发送CMD10,获取CID参数
        SPI_Hardware_SendCMD(CMD10, 0x00000000, OTHER_CSC, CID_Coefficients, NULL);
        if(CID_Coefficients[0] != Card_Ready)
                return Get_CIDCSD_Failed;

        return Get_CIDCSD_Success;
}



/*
Definition:     从SD卡指定位置读取一个扇区的模块
Description:
*/
uint8_t SD_GetData(uint8_t* Sector_Data, uint32_t Address){
    //清空Return_Single_Sector指针
        memset(Sector_Data, 0x00, Total_Sector_Length);
        //发送CMD17,获取指定内存大小(SC)或指定扇区的512字节(HC,XC)的参数
        SPI_Hardware_SendCMD(CMD17, Address, OTHER_CSC, Sector_Data, NULL);
        if(Sector_Data[0] != Card_Ready)
                return Get_Sector_Failed;

        return Get_Sector_Success;
}
```

```
/*
Definition:    往SD卡指定位置写入512字节的数据
Description:
*/
uint8_t SD_WriteData(uint8_t* Data_In, uint32_t Address){
        //发送CMD24,获取指定内存大小(SC)或指定扇区的512字节(HC,XC)的参数
        SPI_Hardware_SendCMD(CMD24, Address, OTHER_CSC, &Return_R1, Data_In);
        if(Return_R1 != Card_Ready)
                return Write_Sector_Success;
        return Write_Sector_Failed;
}
```