基于 spi-flash 的 fatfs 配置

——王京石

硬件平台：stm32f103VCT6、w25x16
软件平台：fatfs R0.10

由于产品需要存储大量数据，stm32 单片机存储有限需要使用外部 flash 辅助存储。考虑各方面原因最后选用了一款 spi-flash 型号为 w25x16，spi 总线操作，拥有 2M 的存储单元。为了方便，我们想到了使用文件系统 fatfs。此文档记录了配置流程，为以后做参考。

一、 底层移植

Fatfs 的 diskio.c 与 diskio.h 文件用于兼容底层接口，主要配置过程就是重写 disk_initialize、disk_status、disk_read、disk_write、disk_ioctl、get_fattime 六个函数以兼容不同的硬件设备。

1、设备初始化 DSTATUS disk_initialize (BYTE pdrv)

用于初始化硬件设备，在本次项目中主要就是初始化 SPI 总线接口，这个底层函数在执行应用层的 open、write、read 等函数是都会被执行。本项目没有对 flash 进行分区操作，因此设备号应该为 0。

```
DSTATUS disk_initialize (
    BYTE pdrv                       /* Physical drive nmuber (0..) */
)
{
    if(pdrv == 0)       //设备号为 0 则进行初始化操作
    {
        SPI_Flash_Init();
        return 0;           //返回 0 表示成功
    }
    else
    {
        return STA_NODISK;
    }
}
```

2、读取设备状态 DSTATUS disk_status (BYTE pdrv);

用于读取设备状态，判断设备是否处于空闲状态，由于本项目使用的存储单元为 spi-flash 所以始终是可以操作的状态，因此始终返回 OK 就可以。

```
DSTATUS disk_status (
    BYTE pdrv               /* Physical drive nmuber (0..) */
)
{
```

```
    if(pdrv == 0)       return 0;
    else                return STA_NODISK;
}
```

3、读扇区操作

```
DRESULT disk_read (
    BYTE pdrv,          /* 物理设备号 */
    BYTE *buff,         /* 读取数据缓冲 */
    DWORD sector,       /* 扇区号 */
    UINT count          /* 读取的扇区个数（1-128） */
)
```

使用读操作在指定扇区里读取出数据。

```
DRESULT disk_read (
    BYTE pdrv,          /* Physical drive nmuber (0..) */
    BYTE *buff,         /* Data buffer to store read data */
    DWORD sector,       /* Sector address (LBA) */
    UINT count          /* Number of sectors to read (1..128) */
)
{
    if(pdrv != 0)       return RES_WRPRT;

    SPI_Flash_Read(buff, ((uint32_t)sector) << 12, ((uint32_t)count) << 12);

    return RES_OK;
}
```

4、写扇区操作 DRESULT disk_write (BYTE pdrv, const BYTE* buff, DWORD sector, UINT count);

使用写操作在指定扇区里写入相应数据，再写入之前必须要擦除扇区。由于 w25x16 最小的擦除块为 4096 字节，因此将 fatfs 的扇区定义为 4096，而 w25x16 一次性写入 256 字节数据，因此每个扇区需要写入八次数据。

```
DRESULT disk_write (
    BYTE pdrv,              /* Physical drive nmuber (0..) */
    const BYTE *buff,   /* Data to be written */
    DWORD sector,           /* Sector address (LBA) */
    UINT count              /* Number of sectors to write (1..128) */
)
{


    BYTE        *buf        = (uint8_t*)buff;
    uint32_t    secAddr     = ((uint32_t)sector) << 12;
```

```
	uint8_t 	i;
	if(pdrv != 0)		return RES_WRPRT;

	for(i=0; i < count; i++)
	{
		SPI_Flash_Erase_Sector(sector);
		sector ++;

		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;

		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;

		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;

		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;
		SPI_Flash_Write_Page((uint8_t*)buf, secAddr, 256);buf += 256;secAddr += 256;



	}
	return RES_OK;

}
```

5、磁盘控制函数　DRESULT disk_ioctl (BYTE pdrv, BYTE cmd, void* buff);

主要用于应用层程序同步磁盘、获取扇区大小、获取磁盘总扇区数、磁盘擦除块大小与擦除操作等功能。注意通过数组 buff 带入或是带出的参数必须要格式匹配。

```
#if _USE_IOCTL
DRESULT disk_ioctl (
	BYTE pdrv,		/* Physical drive nmuber (0..) */
	BYTE cmd,		/* Control code */
```

```
    void *buff              /* Buffer to send/receive control data */
)
{
    if(pdrv != 0)    return RES_WRPRT;

    switch(cmd)
    {
    case CTRL_SYNC                :    //spi-flash 不需要同步，这一项始终返回 0
        return RES_OK;

    case GET_SECTOR_SIZE      :
        *((WORD *)buff)     = 4096;    //始终通过 buff 返回扇区大小就可以了
        return RES_OK;

    case GET_SECTOR_COUNT     :
        *((DWORD *)buff)    = 512;      //始终通过 buff 返回总扇区数就可以了
        return RES_OK;

    case GET_BLOCK_SIZE       :      //禁止擦除功能，这两项无意义
        return RES_OK;
    case CTRL_ERASE_SECTOR    :
        return RES_OK;      }
```

6. 获取时间　　DWORD　get_fattime (void)

在写操作时需要调用的函数，如果需要真实的时间信息则使用 RTC 始终读出时间信息通过这个函数返回，本项目中不需要时间因此始终返回 0 就可以了。

```
/* RTC function */
#if !_FS_READONLY
DWORD
get_fattime (void)
{
    return 0;
}


#endif
```

二、　　fatfs 配置

在移植完底层之后还需要根据需求配置 fatfs，修改各种配置宏是主要的配置手段。

1、配置数据类型

首先我们应该配置的是数据类型，因为 fatfs 需要运行在各种硬件平台上，而不同位数的机器数据类型的结构也不一样，因此需要统一数据类型，这一配置在 integer 中完成。

```
typedef unsigned char    BYTE;    //BYTE 配置为 8 位无符号类型

typedef short            SHORT; //16 位有符号类型
typedef unsigned short   WORD;//16 位无符号类型
typedef unsigned short   WCHAR;//16 位无符号类型

/* These types MUST be 16 bit or 32 bit */
typedef int                  INT; //32 位有符号类型
typedef unsigned int     UINT;     //32 位无符号类型

/* These types MUST be 32 bit */
typedef long             LONG;    //32 位有符号类型
typedef unsigned long    DWORD;//32 位无符号类型
```

2、配置宏定义

Fatfs 的配置宏主要在文件 ffconf.h 中。接下来让我们一个一个分析这些宏。

```
#define   _FS_TINY       1
```

0 正常模式　1 小型模式

小型模式，主要用于内存资源不丰富的微控制器，在这里我们选 1 配置为小型模式

```
#define _FS_READONLY  0
```

0　可读可写

1　只读

配置文件系统位只读文件系统，这里选 0

```
#define _FS_MINIMIZE   3     /* 0 to 3 */
```

0　使能所有功能函数

1　f_stat(), f_getfree(), f_unlink(), f_mkdir(), f_chmod(), f_utime(), f_truncate() and f_rename()函数被禁止

2　除了 1 之外 f_opendir(), f_readdir() and f_closedir()也被禁止

3　除了 2 之外 f_lseek()也被禁止

用于剪裁文件系统，删除不需要的功能，这里选 3

```
#define   _USE_STRFUNC     0     /* 0:Disable or 1-2:Enable */
```

0　禁止字符串功能

1　使能字符串功能

若使能字符串功能，则可以使用 f_gets ,f_putc,f_puts,f_printf 等函数，这里选禁止

```
#define   _USE_MKFS         1     /* 0:Disable or 1:Enable */
```

0 禁止格式化
1 使能格式化

若选 1 则可以使用函数 f_mkfs 函数格式化磁盘,这里选 1

```
#define   _USE_FASTSEEK     0     /* 0:Disable or 1:Enable */
```

0 禁止快速查找功能
1 使能快速查找功能
使能或禁止快速查找功能，这里选禁止

```
#define _USE_LABEL          0     /* 0:Disable or 1:Enable */
```

0 禁止卷标功能
1 使能卷标功能
使能或禁止卷标功能，这里选禁止

```
#define   _USE_FORWARD      0     /* 0:Disable or 1:Enable */
```

0 禁止 f_forward()函数
1 使能 f_forward()函数

使能或禁止 f_forward()函数，f_forworad()函数用于将文件信息发送到一个数据流设备上去，这里禁止。

```
/*---------------------------------------------------------------------------/
/  语言环境配置
/---------------------------------------------------------------------------*/

#define _CODE_PAGE    932
```

配置语言编码，这里保持默认

```
#define   _USE_LFN      0           /* 0 to 3 */
#define   _MAX_LFN      255         /* Maximum LFN length to handle (12 to 255) */
/* _USE_LFN 用于使能长文件名功能，
/
/    0: 禁止长文件名功能. _MAX_LFN 没有意义；
/    1: 使能长文件名功能，文件名存储在 BSS 段中，不可重入；
/    2: 使能长文件名功能，文件名存储在栈中；
/    3: 使能长文件名功能，文件名存储在堆中；
/
```

#define _LFN_UNICODE 0 /* 0:ANSI/OEM or 1:Unicode */
0 ANSI/OEM 编码
1 Unicode 编码(支持中文)
用于切换文件名编码，这里选 0

#define _STRF_ENCODE 3 /* 0:ANSI/OEM, 1:UTF-16LE, 2:UTF-16BE, 3:UTF-8 */
这个选项用于选择 文件字符操作时的编码，禁止 unicode 时此选项无效

#define _FS_RPATH 0 /* 0 to 2 */
0 禁用相对路径功能，并卸载这个功能
1 使能相对路径功能

禁止或使能相对路径功能，这里禁止
/*---/
/ 设备配置
/---*/

#define _VOLUMES 1
用于定义磁盘中卷的数量

#define _MULTI_PARTITION 0 /* 0:Single partition, 1:Enable multiple partition */
0 单分区模式
1 多分区模式
若使能多分区模式则可以使用 f_fdisk 函数对设备进行分区，这里禁止

#define _MAX_SS 4096 /* 512, 1024, 2048 or 4096 */
定义扇区大小，可选的值位 512 1024 2048 4096

#define _USE_ERASE 0 /* 0:Disable or 1:Enable */
0 禁止扇区擦出功能
1 使能扇区擦除功能

禁止或使能扇区擦除功能，这里选禁止

#define _FS_NOFSINFO 0 /* 0 or 1 */
若选 1 则可以使用 f_getfree()获取卷中剩余的空间大小，这里不适用

/*---/

/系统配置
/--*/

#define _WORD_ACCESS 0 /* 0 or 1 */
如果硬件平台为小端模式则选择 1