Setting up FatFS on Embedded Microcontrollers

(a guide to set up the FAT file system for SD Card storage)

Rohit Dureja
University of Pennsylvania

INTRODUCTION

SD Card storage is an important requirement for embedded systems which require offline processing of data. The sensed/computed data is stored in non-volatile memory, which is Flash in case of a memory card nowadays, for future processing and analysis. For easy readability on a personal computer, these SD cards are formatted to a format which resembles the directory structure understood by the host operating system. Looking at it in another way, a large file can be stored on the card using a computer and then easily read by an embedded microcontroller. The interface diagram of such a system looks like what is show in Figure 1

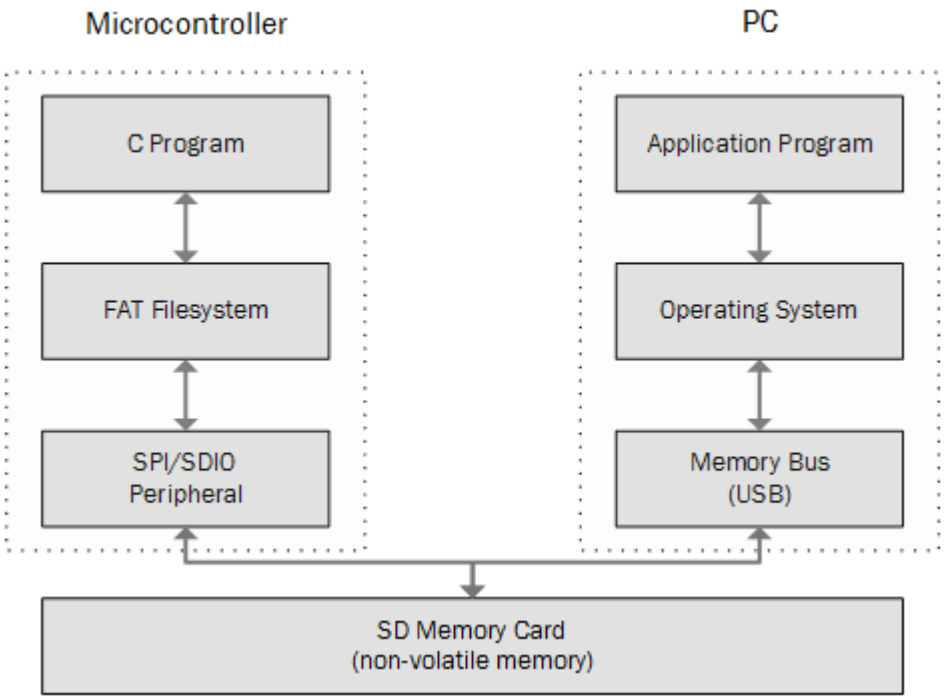

Figure 1: SD Card Interface block diagram

As can be seen in the block diagram, the same memory card can be read/written by the embedded microcontroller as well as the PC. Described below is how the read/write operations are carried out inside the microcontroller. We do not discuss read/write operations on the PC side in this guide.

Writing to the Memory Card

1. The user program, which is usually written in C, collects the data it wants to store and saves it in a buffer. The data in the current format is called raw data.
2. The raw data is sent to the FAT filesystem library implemented on the microcontroller that converts the data to FAT format.
3. The FAT formatted data is now sent to the hardware peripheral (SPI/SDIO) to which the physical SD card is connected. The peripheral uses standard bus protocols to transfer data to the SD Card. A controller on the SD card receives this data and stores it in non-volatile memory and this complete the write process.

Reading from the Memory Card

1. The data to be read is stored on the memory card in FAT format. The controller on the SD card uses standard bus protocols to transfer this data to the hardware peripheral (SPI/SDIO).
2. The hardware peripheral receives this data and sends it to the FAT filesystem library for conversion to raw data.
3. The filesystem library converts the data back to raw and sends it to the user program which can now read it.

All in all, a filesystem describes the way the data is organized in memory. If two devices save data with the same organization, it makes it easy to transfer information between the two. In our case, these two devices are the microcontroller and PC.

This guide aims to make using filesystem libraries on a microcontroller easy and fast. We use a FAT filesystem library, originally written by Elm Chan[1] in this guide. The guide is organized as follows: Section I describes the structure of the library and various components and Section II gives a brief of the hardware connections required for interfacing the memory card. Once the initial setup is done, Section III discusses the portions of the library which need to be modified to work with a generic microcontroller. In our discussion, we use the Texas Instruments Tiva™ ARM® microcontroller, however, the same procedure can be applied to any other microcontroller. In Section IV we start with our first user application for an embedded system which reads from and writes to a memory card. To end this guide, in Section V we document the various pre-existing library ports which are available in the open source domain for various microcontrollers.

SECTION I: THE FATFS FILESYSTEM LIBRARY

The FatFS library is written completely in ANSI C. It is developed by keeping portability in mind. The library can be used on any microcontroller platform assuming the following conditions on portability:

1. It is a middleware library that runs between the application program and hardware. As long as the compiler is in compliance with ANSI C standard, there should be no problems while porting.
2. To make it portable across different 8/16/32 bit architectures, the library assumes sizes of char/short/long are 8/16/32 bit and integer is 16 or 32 bit.

[1] http://elm-chan.org/fsw/ff/00index_e.html

The first step in understanding the library module is to download it. The current version as of December 7, 2014 is R0.10c[2]. For convenience, store the downloaded zip in your workspace which is accessible to your C compiler. You can use any text editor to view/edit file.

Directory Structure

The unzipped library contains several files that are primarily platform independent. These files can then be used with platform specific code files and libraries to develop an embedded system with SD Card storage. Figure 2 shows the file structure of the library and platform dependent files along with their interaction.

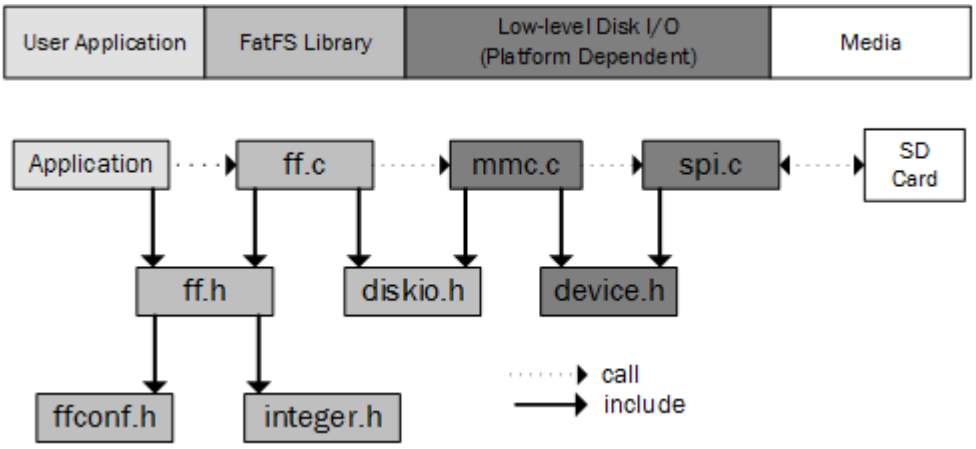

Figure 2: FatFS Library file structure

Let's start with the files associated with the FatFS library. These files should match the ones inside the unzipped folder, which are:

- ff.c – the main FatFS library module.
- ff.h – common include file for FatFS and the application program
- ffconf.h – configuration file for the FatFS library
- diskio.h – common include file for FatFS and the disk I/O module
- integer.h – integer type definitions for FatFS library

Apart from these library files, which are platform independent, we need to create/use a few other files for the specific platform in use. These are:

- mmc.c – platform dependent SPI/SDIO read/write functions.
- spi.c – the platform vendor supplied SPI peripheral driver
- device.h – the platform vendor supplied include file for peripheral/port definitions.

Of the above three, spi.c and device.h are usually provided the platform vendor. The only file which needs to be added/modified is mmc.c. If we successfully create this file, the library can be run on any platform. Before we start porting the library to our own platform, let's see what hardware connections are required between the microcontroller and the memory card.

[2] http://elm-chan.org/fsw/ff/ff10c.zip

SECTION II: HARDWARE CONNECTIONS

SD Card use two methods of interfacing to the microcontroller: SD mode and SPI mode. In this guide, we focus on using a SD/MMC Card connected on the SPI bus to the microcontroller. Using the memory card in SD mode using SDIO peripheral is beyond the scope of this guide. The SD Card has the following 8 pins which need to be connected to the power supply and the communication bus for proper use. These are listed in Table 1.

Pin No.	SPI Mode	SPI Mode Functionality	SD Mode	SD Mode Functionality
1	-	-	DAT2	Data line 2
2	CS	Chip select	DAT3	Data line 3
3	DI	Data input	CMD	Command input
4	VDD	Positive power supply	VDD	Positive power supply
5	SCLK	Clock signal	CLK	Clock signal
6	VSS	Negative power supply	VSS	Negative power supply
7	DO	Data output	DAT0	Data line 0
8	-	-	DAT1	Data line 1

Table 1: Pin layout for SD Card

Of the 8 pins on the SD Card, the number of effective pins used in SPI mode for interfacing are six. Let us now look at how these pins are connected to pins on a microcontroller.

The SPI peripheral on all microcontrollers is a synchronous serial interface. It is a four-wire interface out of which two wires are used for data transmission/reception, one wire is used for synchronizing the MCU with the SPI device and the fourth is used to select the SPI device. These are called:

1. MOSI – Master-Output Slave-Input
2. MISO – Master-Input Slave-Output
3. SCLK – Synchronizing Clock
4. CS – Chip select

On some devices, these four pins might be marked with different names but the functionality remains the same. The master and slave designation is from the point-of-view of the system generating the SCLK signal. In our case, the microcontroller generates the SCLK signal and is thus the Master. The SD card is the Slave. Figure 3 shows the basic connections required between the microcontroller and the memory card.

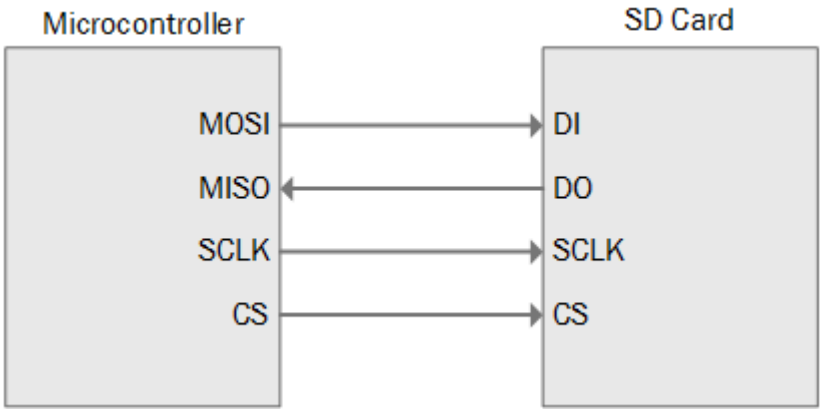

Figure 3: SD Card/Microcontroller connections

In the next section we discuss what platform dependent files need to be added/changed to get the filesystem working on any library. The sections assumes proficiency in using the SPI peripheral, the vendor supplied driver library and general programming concepts.

SECTION III: PORTING THE FILESYSTEM

As described in Section I, the library which we downloaded doesn't contain any platform dependent source and include files. Hence, it is up to the programmer to supply these files when using the library for any application. There are two ways to go about this:

1. We start writing the device dependent files from scratch. This requires a thorough understanding of the SD card controller, the FAT filesystem, the command interface it understands and the type of output it produces.
2. We pick files for another microcontroller platform and modify it to suit our purposes. This is an easy and straightforward approach and requires the programmer to only write the SPI driver specific routines.

We will be going with alternative 2 as it suits our purpose and is easy enough to get things quickly started. In this guide, we will pick up a file[3], which was originally written for Texas Instruments Tiva™ ARM® microcontrollers and modify it to suit our needs.

The first thing that needs to change is the included header file. Replace the code segment in Listing 1 with your own platform dependent header files. The next thing that needs tweaking is the global definitions for the GPIO pins and the SPI peripheral as outlined by the platform library. For the Tiva™ controllers these definitions are outlined in Listing 2.

Now that we have the required header files and global definitions, let us now look at the library functions that need some modification. The modifications they require pertain to the peripheral driver interface of the platform in use. For example, library functions to toggle GPIO pins or send/receive data on the SPI bus.

[3] Download the mmc-dk-tm4c12g.c file from https://github.com/rohitdureja/SD-MMC-FatFS-Library-for-Tiva

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/rom.h"
#include "driverlib/ssi.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "diskio.h"
```

Listing 1: Included header files

```
/* Peripheral definitions board */
// SSI port
#define SDC_SSI_BASE            SSI1_BASE
#define SDC_SSI_SYSCTL_PERIPH   SYSCTL_PERIPH_SSI1

// GPIO for SSI pins
#define SDC_GPIO_PORT_BASE      GPIO_PORTD_BASE
#define SDC_GPIO_SYSCTL_PERIPH  SYSCTL_PERIPH_GPIOD
#define SDC_SSI_CLK             GPIO_PIN_0
#define SDC_SSI_TX              GPIO_PIN_3
#define SDC_SSI_RX              GPIO_PIN_2
#define SDC_SSI_FSS             GPIO_PIN_1
#define SDC_SSI_PINS         (SDC_SSI_TX | SDC_SSI_RX | SDC_SSI_CLK | SDC_SSI_FSS)
```

Listing 2: Global peripheral definitions

The functions that need to be modified are listed in Table 2 along with information on the role they play in the FatFS library. Also refer to the code comments in the file when making changes to these functions.

S.No	Function Name	Role	Changes Required
1	`static void select`	Asserts the chip select on the SPI bus	GPIO toggle for SPI CS pin
2	`static void deselect`	De-asserts the chip select on the SPI bus	GPIO toggle for SPI CS pin
3	`static void xmit_spi`	Transmit a byte via SPI	SPI data write
4	`static byte rcvr_spi`	Receive a byte via SPI	SPI data read
5	`static void send_initial_clock_train`	Send initial clock pulses to the SD Card. Required to get SD Card in SPI mode	GPIO and SPI functions
6	`static void power_on`	Power Control of the SD Card	GPIO, SPI and system power control functions
7	`static void set_max_speed`	Set the SPI speed to the max setting	SPI control functions

Table 2: Functions that need modification

SECTION IV: SAMPLE APPLICATION

Now that we have ported the FatFS library to our platform, let us look into the structure of an application using a SD Card. The library provides a variety of application[4] functions for use in user code but we restrict our discussion to only a few; `f_mount, f_open, f_read, f_write` and `f_close.`

[4] http://elm-chan.org/fsw/ff/00index_e.html in the Application Interface section

Listing 3 summarizes a code segment of an application doing a write operation on a SD Card. We start by including the relevant library file to our code. We have shown the minimum files that need to be included from the library; your application might include other peripheral driver files. The next step is to declare global variables for interacting with the SD card filesystem.

The next step is to mount the SD Card and register a logical drive number to it. This is similar to

```
#include "ff.h"
#include "diskio.h"

/*Variable required for SD Card R/W
FATFS fatfs;
FIL fil;
WORD buffer[4096];
FRESULT rc;
UINT br, bw;

// mount the file system, using logical disk 0.
rc = f_mount(0, &fatfs);
if(rc != FR_OK)
{
        assert('Failure');
        return 0;
}
else
{
        assert('Success');
}

// fill up data buffer
for(i = 0 ; i<4096 ; i++)
        buffer[i]='a';

// open a file
rc = f_open(&fil, "file.txt", FA_CREATE_ALWAYS | FA_WRITE);
if(rc != FR_OK)
{
        assert('Failure');
        return 0;
}
else
{
        assert('Success');
}

// write to file
for(i = 0 ; i < 10000 ; i++)
{
        rc = f_write(&fil, buffer, strlen((const char *)buffer), &bw);
        if(rc != FR_OK)
        {
                assert('Failure');
                return 0;
        }
}

// close file
f_close(&fil);

// unmounts SD Card
f_mount(0, NULL);
```

Listing 3: Sample application

mounting a file system on a personal computer, e.g. C: drive. We use the `f_mount` function to mount

the SD Card. Assuming we don't have an exisiting file with the name "file.txt" on the SD card, we call `f_open` to create a new file. This function returns an error code if file creation fails.

`f_write` is used to write data to SD Card in chunks. We use data chunks of 4096 bytes. The code sample above is self-explanatory on how this function is called and used. Once we are done writing the file, we close it using `f_close`. When we no longer require the SD card, we can disable it by unmounting the file system and deasserting the SPI bus. All this is done in one function call using `f_mount`.

SECTION V: EXISTING PORTS

Before you start writing a port of the FatFS library, we urge you to take a look at pre-existing ports. These can be downloaded[5] from the website and may help fasten the process to create your own library. As of December 2014, the library has been ported to the platforms listed in Table 3.

AVR	Tiva
STM8	STM32
LPC23xx	LPC176x
PIC	RX600

Table 3: Existing ports

[5] http://elm-chan.org/fsw/ff/ffsample.zip