3/15/2009

RICKEY'S WORLD

MMC/SD INTERFACE WITH FAT16 FS

www.8051Projects.net | By David

INTERFACING AN 8052 MICROCONTROLLER TO AN SD MEMORY CARD USING CHAN'S LIBRARY

Table of Contents

Interface to Chan's Library of functions *3*

Target development platform *3*

Setting up the SPI port during startup.A51 .... *5*

Global type definitions and variables *5*

Basic SPI function *8*

TRANSFERRING A SINGLE BYTE OVER THE SPI BUS – RECEIVING A BYTE AT THE SAME TIME 8

SPI CHIP SELECT 8

SETTING THE FREQUENCY FOR THE SPI CLOCK 8

SENDING A COMMAND TO THE SD CARD 9

READING A RESPONSE FROM THE SD CARD 10

PROVIDING A DELAY FUNCTION AND A TIME FUNCTION 10

SD Card Initialization *11*

SETTING UP THE CARD FOR SPI COMMUNICATION 11

Reading and Writing a single sector *12*

One final function… *13*

Working with diskio.c *13*

Pulling it all together *16*

Interface to Chan's Library of functions

First of all, if you haven't downloaded Chan's source code, you should do this first. You can get from him at this link:

http://elm-chan.org/fsw/ff/ff006.zip

When you unzip this file, use the default directories and you will end up with a directory structure like this:

Name	Date modified	Type	Size
avr	1/23/2009 7:59 PM	File Folder	
doc	1/23/2009 8:00 PM	File Folder	
h8	1/23/2009 8:00 PM	File Folder	
lpc2k	1/23/2009 8:01 PM	File Folder	
pic	1/23/2009 8:01 PM	File Folder	
src	1/23/2009 8:02 PM	File Folder	
tlcs	1/23/2009 8:02 PM	File Folder	
v850es	1/23/2009 8:02 PM	File Folder	
00readme	10/19/2008 12:52 ...	Text Document	3 KB

Chan has included examples for a variety of processors. The basic set of routines are found in the **src** *subdirectory. As he notes in his readme file, the files included here are:*

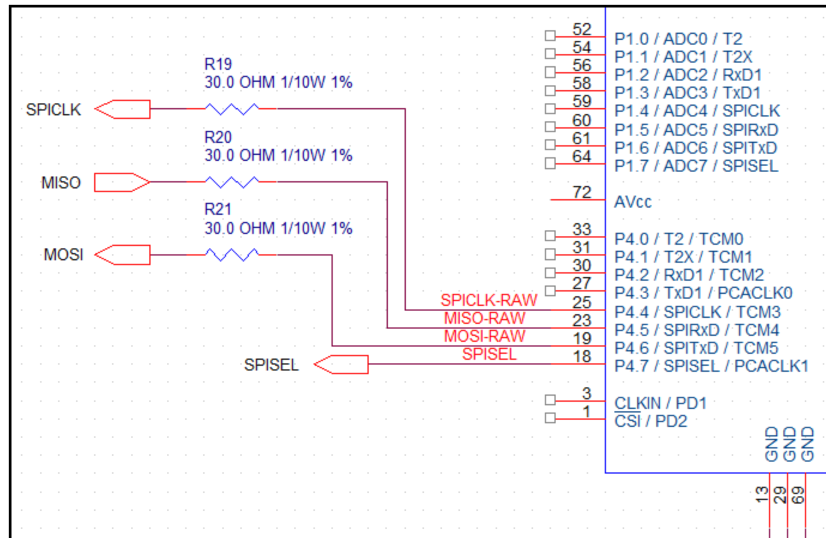
```
ff.h       Common include file for FatFs and application module.
ff.c       FatFs module.
tff.h      Common include file for Tiny-FatFs and application module.
tff.c      Tiny-FatFs module.
diskio.h   Common include file for (Tiny-)FatFs and disk I/O module.
diskio.c   Skeleton of low level disk I/O module.
integer.h  Alternative type definitions for integer variables.
```

Chan also states that ***"the low level disk I/O module is not included in this archive because the FatFs/Tiny-FatFs module is only a generic file system layer and not dependent on any specific storage device. You have to provide a low level disk I/O module that is written to control your storage device."***

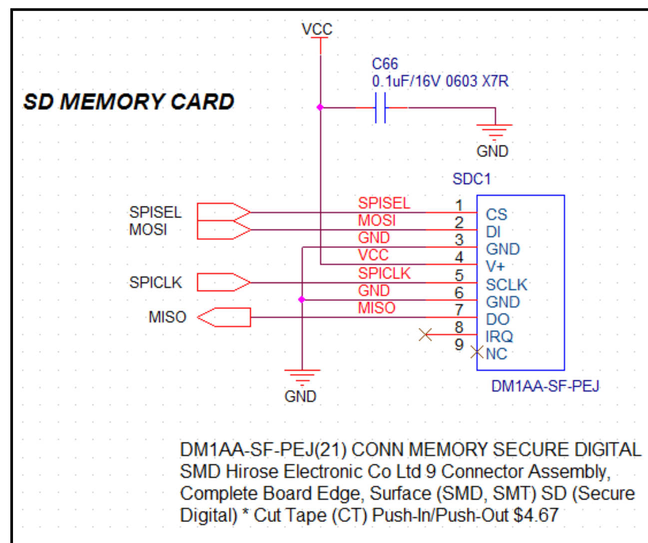
That is the intent behind this document – to show you how to write the low level disk I/O module so the file system layer can work properly.

Target development platform

This tutorial was specifically written around the microcontroller I am using, which is the ST upsd3334D microcontroller. Since it is fundamentally an 8052, you should be able to adapt this tutorial to your specific microcontroller; however, some of the Special Function Registers will probably be different. The upsd3334D is unique in that it is not only an 8052, but also a CPLD(Complex Programmable Logic Device) in the same part. The 8052 portion of the part is 3.3v and the CPLD portion is 5.0v. The hardware interface to the SD card is simple. The signals can connect directly to the 3.3v pins on the 8052, but I added 30 ohm resistors to reduce signal bounce.



In this tutorial, the SD memory card is attached to the 3.3v section of the micro, so no voltage shifting is necessary.



Remember: The SD memory card has a limited voltage range, so ALL signals to and from it are a maximum of 3.3v. If you connect 5.0v signals to the SD memory card, you might damage it.

Additionally, I am using the Keil development PK51 suite, which consists of the IDE (Integrated Development Environment), the C51 compiler, the AX51 assembler, and the LX51 link-loader. The version is 8.17a, but I believe any version should work.

If you would like to read more about CPLD's, you can visit this link: http://en.wikipedia.org/wiki/CPLD

If you would like to read more about the processor I am using, you can visit this link: http://www.keil.com/dd/chip/3639.htm

Setting up the SPI port during startup.A51

*For the upsd3334D, the SFR's that set the functional aspects for the SPI port are P4SFS0 and P4SFS1. Your microcontroller will vary here, but the main idea is to setup the SPI pins so that you have an MOSI, MISO, SPICLK, and CS – These names correspond to "**M**icro **O**UT **S**lave **I**N", "**M**icro **I**N **S**lave **O**UT", "**SPI CL**oc**K**", and "**C**hip **S**elect" respectively.*

For my microcontroller, I set the port for the following settings, from within the STARTUP.A51 file (in assembler):

```
/* Program uPSD PORT-4 registers...                                                        */
/*                                                                                         */
/* P4SFS0 - sets the primary function of the pins - default is '0', which is GPIO pin      */
/*          0 - buzzer output (PWM)                 (set as alternate - 1)                 */
/*          1 - SPIADDRESS SELECT 0 -|                                                     */
/*          2 - SPIADDRESS SELECT 1  | These are used as 3-8 decoder for SPI device select */
/*          3 - SPIADDRESS SELECT 2 -|              (set as GPIO - 0)                      */
/*          4 - spiclock                            (set as alternate - 1)                 */
/*          5 - MISO                                (set as alternate - 1)                 */
/*          6 - MOSI                                (set as alternate - 1)                 */
/*          7 - manual spi select line -            (set as GPIO - 0)                      */
/*                                                                                         */
/*  P4SFS1 - sets the alternate function, if corresponding bit in P4SFS0 is set            */
/*          0 - PCA0 Module 0, TCM0                 (set as 0)                             */
/*          1 - ignored, since P4SFS0 is 0 already  (set as 0)                             */
/*          2 - ignored, since P4SFS0 is 0 already  (set as 0)                             */
/*          3 - ignored, since P4SFS0 is 0 already  (set as 0)                             */
/*          4 - SPI Clock, SPICLK                   (set as 1)                             */
/*          5 - SPI Receive, SPIRXD                 (set as 1)                             */
/*          6 - SPI Transmit, SPITXD                (set as 1)                             */
/*          7 - ignored, since P4SFS0 is 0 already  (set as 0)                             */
/*                                                                                         */
                MOV     P4SFS0, #071H
                MOV     P4SFS1, #070H
```

Again, since your hardware platform will be different, you should setup your SPI port accordingly. The main idea here is to have an automated SPI port, and a manually controlled chip select – Don't let the processor automatically control the CS line, since there are times when you will want it to stay either active or inactive, depending on what you are doing. My microcontroller has a feature that will automatically enable the CS line when the SPI clock runs, then automatically disable the CS line once the transmission is complete – Using this ability was a mistake, and so I disabled it, resorting to a manually controlled CS line.

Global type definitions and variables

Here are some standard definitions I use throughout this tutorial. You can modify them to suite your needs, but these seem to work fine for me. I modified the INTEGER.H file included in Chan's library as follows:

```
/*-------------------------------------------*/
/* Integer type definitions for FatFs module */
/*-------------------------------------------*/

#ifndef _INTEGER

   /* These types must be 16-bit, 32-bit or larger integer */
   typedef int                     INT;
   typedef unsigned int       UINT;

   /* These types must be 8-bit integer */
   typedef signed char        CHAR;
   typedef unsigned char      UCHAR;

   /* These types must be 16-bit integer */
   typedef short                   SHORT;
   typedef unsigned short     USHORT;

   /* These types must be 32-bit integer */
   typedef long                    LONG;
   typedef unsigned long      DWORD;


   #ifndef STANDARD_TYPES
```

```
      #define STANDARD_TYPES
      typedef unsigned char BYTE;
      typedef unsigned int  WORD;
      typedef unsigned long ULONG;
   #endif

   /* Boolean type */
   typedef enum { FALSE = 0, TRUE } BOOL;

   #define _INTEGER
#endif
```

The following variable is global, and necessary for Chan's Library

```
xdata WORD CardType;                   /* MMC = 0, SDCard v1 = 1, SDCard v2 = 2 */
```

These are specific to the SPI interface in my microcontroller. Again, you can modify them to suite your needs, and better match your hardware environment.

```
enum SPI_FREQUENCIES { kHz400, MHz1, MHz5, MHz10 };

   /* ========================================================================== */
   /* Table 59. SPICON0: Control Register 0 (SFR D6h, Reset Value 00h)          */
   /*                                                                            */
   /* BIT   SYMBOL   R/W   DEFINITION                                            */
   /* ---   ------   ---   -------------------------------------------------- */
   /*  7      --      -    Reserved...                                          */
   /*  6      TE     R/W   T)ransmitter E)nable                                 */
   /*                      0 = transmitter disabled, 1 = transmitter enabled    */
   /*  5      RE     R/W   R)eceiver E)nable                                    */
   /*                      0 = receiver disabled, 1 = receiver enabled          */
   /*  4    SPIEN    R/W   SPI) E)nable                                         */
   /*                      0 = entire SPI interface disabled,                   */
   /*                      1 = entire SPI interface enabled                     */
   /*  3     SSEL    R/W   S)lave SEL)ection                                    */
   /*                      0 = SPISEL output always '1',                        */
   /*                      1 = SPISEL output '0' during transfers               */
   /*  2     FLSB    R/W   F)irst LSB)                                          */
   /*                      0 = transfer MSB first, 1 = transfer LSB first       */
   /*  1     SPO     R/W   S)ampling PO)larity                                  */
   /*                      0 = Sample transfer data at falling edge of clock    */
   /*                         (SPICLK is '0' when idle)                         */
   /*                      1 = Sample transfer data at rising edge of clock     */
   /*                         (SPICLK is '1' when idle)                         */
   /*  0      --      -    Reserved...                                          */
   /*                                                                            */
   /* ========================================================================== */
   #define     TE          0x40
   #define     RE          0x20
   #define     SPIEN       0x10
   #define     SSEL        0x08          // This feature is disabled in PDSoft...
   #define     FLSB        0x04
   #define     SPO         0x02
```

The following table is used for setting the frequency of the SPI clock – This is definitely hardware specific. My system's normal crystal frequency is 40.0Mhz, but this example uses a 22.1184Mhz crystal, so that's why I setup this table. Also, the upsd3334D microcontroller is a 4-clocker, not a 12-clocker device, which means it takes only 4 clock cycles to execute a simple instruction, not 12 like a typical 8052. Your hardware will dictate the values you use as timer constants

```
   /* ========================================================================== */
   /* Table 61. SPICLKD: SPI Prescaler (Clock Divider)                          */
   /*           Register (SFR D2h, Reset Value 04h)                             */
   /*           based on frequency, the following are divisors for the          */
   /*           SPI clock register                                              */
   /* ========================================================================== */
   #define     SPI_FREQUENCY_10MHz     4     // Fastest: 10MHz @ 40MHz, and
                                             // 5.529MHz at 22.1184MHz
   #ifdef NORMAL_SPEED
      #define     SPI_FREQUENCY_5MHz   8     // 5MHz
      #define     SPI_FREQUENCY_1MHz   40    // 1MHz
      #define     SPI_FREQUENCY_400KHz 100   // 400kHz
   #else
```

```
   #define     SPI_FREQUENCY_5MHz   4     // 5.529MHz
   #define     SPI_FREQUENCY_1MHz   20    // 1.105MHz
   #define     SPI_FREQUENCY_400KHz 56    // 394.971kHz
#endif

/* ========================================================================================== */
/* Table 62. SPISTAT: SPI Interface Status Register (SFR D3h, Reset Value 02h)               */
/*                                                                                            */
/* BIT   SYMBOL   R/W   DEFINITION                                                           */
/* ---   ------   ---   ------------------------------------------------------------------- */
/*  7      --      -    Reserved...                                                          */
/*  6      --      -    Reserved...                                                          */
/*  5      --      -    Reserved...                                                          */
/*  4     BUSY     R    SPI Busy                                                             */
/*                      0 = Transmit or Receive is completed                                 */
/*                      1 = Transmit or Receive is in process                                */
/*  3     TEISF    R    T)ransmission E)nd I)nterrupt S)ource F)lag                          */
/*                      0 = Automatically resets to '0' when firmware reads this register    */
/*                      1 = Automatically sets to '1' when transmission end occurs           */
/*  2    RORISF    R    R)eceive O)verrun I)nterrupt S)ource F)lag                           */
/*                      0 = Automatically resets to '0' when firmware reads this register    */
/*                      1 = Automatically sets to '1' when receive overrun occurs            */
/*  1     TISF     R    T)ransmission I)nterrupt S)ource F)lag                               */
/*                      0 = Automatically resets to '0' when SPITDR is full                  */
/*                         (just after the SPITDR is written)                                */
/*                      1 = Automatically sets to '1' when SPITDR is empty                   */
/*                         (just after BYTE loads from SPITDR into SPI shift register)       */
/*  0     RISF     R    R)eception I)nterrupt S)ource F)lag                                  */
/*                      0 = Automatically resets to '0' when SPIRDR is empty                 */
/*                         (after the SPIRDR is read)                                        */
/*                      1 = Automatically sets to '1' when SPIRDR is full                    */
/*                                                                                            */
/* ========================================================================================== */
#define     BUSY         0x10
#define     TEISF        0x08
#define     RORISF       0x04
#define     TISF         0x02
#define     RISF         0x01
```

Basic SPI function

TRANSFERRING A SINGLE BYTE OVER THE SPI BUS – RECEIVING A BYTE AT THE SAME TIME

Transferring a single byte over the SPI bus is the most fundamental function. This function is used throughout this tutorial, and is how we send and receive information over the bus. The upsd3334D microcontroller has a full-duplex interface bus, so when you send a byte, you receive a byte as well. This function performs the necessary handshaking to send and receive a single byte.

```
BYTE SPI_Byte( BYTE ThisByte )
   {
   while( !(SPISTAT & TISF) );
   while( (SPISTAT & BUSY) );
   SPITDR = ThisByte;
   while( !(SPISTAT & RISF) );
   while( (SPISTAT & BUSY) );
   return( SPIRDR );
   }
```

SPI CHIP SELECT

Another primitive function is enabling the CS line to the SD card. Since the SD card has a negative CS (the signal is low to select the card), we want to write a 0 when selecting the card, and a '1' when we're not selecting the card. The following two functions perform this process, but again – they are specific to the hardware design, and will be different than this (most probably).

```
void SPI_EnableCS()
   {
   P4 &= 0x7F;       /* enable /CS line to select device  */
   }

void SPI_DisableCS()
   {
   P4 |= 0xF0;       // disable /CS line to select device  */
   }
```

SETTING THE FREQUENCY FOR THE SPI CLOCK

Setting the clock frequency for the SPI bus is accomplished by this function. Note, based on the definitions above, the maximum attainable speed for this microcontroller using the 22Mhz crystal is 5Mhz, so a call of SPI_Init(MHz10); will still only achieve 5Mhz.

```
void SPI_Init( enum SPI_FREQUENCIES ThisFrequency )
   {
   SPI_DisableCS();     /* disable chip select if it's enabled...  */

   /* setup SPI control registers */
   SPICON0 = SPIEN | TE | RE;          /* Enables SPI, Tx and Rx  */
   SPICON1 = 0x00;                     // no interrupts
   switch( ThisFrequency )             // sets frequency...
      {
      case MHz10:
         SPICLKD = SPI_FREQUENCY_10MHz;
         break;
      case MHz5:
         SPICLKD = SPI_FREQUENCY_5MHz;
         break;
      case MHz1:
         SPICLKD = SPI_FREQUENCY_1MHz;
         break;
      case kHz400:
      default:
         SPICLKD = SPI_FREQUENCY_400KHz;
         break;
      }
   }
```

Sending a command to the SD card

Sending a single command, and reading the response from the card is the next logical building block. For a detailed description of the command structure, you must refer to the Product Specification Manual, titled "SanDisk Secure Digital Card, Product Manual, Version 1.9, Document No. 80-13-00169, December 2003". The function I use to do this is as follows:

```
   #define CMD_GO_IDLE_STATE          0
   #define CMD_SEND_OP_COND           1
   #define CMD_SEND_CSD               9
   #define CMD_SEND_CID               10
   #define CMD_STOP_TRANSMISSION      12
   #define CMD_SEND_STATUS            13
   #define CMD_SET_BLOCKLEN           16
   #define CMD_READ_SINGLE_BLOCK      17
   #define CMD_READ_MULTIPLE_BLOCK    18
   #define CMD_WRITE_SINGLE_BLOCK     24
   #define CMD_WRITE_MULTIPLE_BLOCK   25
   #define CMD_PROGRAM_CSD            27
   #define CMD_SET_WRITE_PROT         28
   #define CMD_CLR_WRITE_PROT         29
   #define CMD_SEND_WRITE_PROT        30
   #define CMD_TAG_SECTOR_START       32
   #define CMD_TAG_SECTOR_END         33
   #define CMD_UNTAG_SECTOR           34
   #define CMD_TAG_ERASE_GROUP_START  35
   #define CMD_TAG_ERASE_GROUP_END    36
   #define CMD_UNTAG_ERASE_GROUP      37
   #define CMD_ERASE                  38
   #define CMD_LOCK_UNLOCK            42
   #define CMD_APP_CMD                55
   #define CMD_READ_OCR               58
   #define CMD_CRC_ON_OFF             59
   #define ACMD_SEND_OP_COND          41

   typedef union
   {
      BYTE Index[6];
      struct
         {
         BYTE Command;
         ULONG Argument;
         BYTE Cksum;
         } CA;
      } CommandStructure;

   typedef union
   {
      BYTE b[4];
      ULONG ul;
   } b_ul;

BYTE SD_Command( BYTE ThisCommand, ULONG ThisArgument )
   {
   b_ul Temp;
   BYTE i;

   /* enable the device... */
   SPI_EnableCS();

   /* send buffer clocks to insure no operations are pending... */
   SPI_Byte( 0xFF );

   /* send command */
   SPI_Byte(0x40 | ThisCommand);

   /* send argument */
   Temp.ul = ThisArgument;
   for( i=0; i<4; i++ )
      SPI_Byte( Temp.b[ i ] );

   /* send CRC */
   SPI_Byte((ThisCommand == CMD_GO_IDLE_STATE)? 0x95:0xFF);

   /* send buffer clocks to insure card has finished all operations... */
   SPI_Byte( 0xFF );
   return( 0 );
   }
```

Reading a Response from the SD Card

Every command sent to the SD card invokes a response from the card. The size of the response, along with the content of the response, is dependent on the command sent. Here are two (2) functions that can be used to read response bytes from the SD card.

```
BYTE SD_GetR1()
   {
   BYTE i, j;

   for( i=0; i<8; i++ )
      {                           /* response will be after 1-8 0xffs.. */
      j = SPI_Byte( 0xff );
      if(j != 0xff)               /* if it isn't 0xff, it is a response */
         return(j);
     }
   return(j);
   }


WORD SD_GetR2()
   {
   idata WORD R2;

   R2 = ((SD_GetR1())<< 8) & 0xff00;
   R2 |= SPI_Byte( 0xff );
   return( R2 );
   }
```

Providing a Delay Function and a Time Function

Finally, this tutorial requires a delay function and a time function – The function I use throughout is called Delay(), and the passed value is in milliseconds. You can achieve this in various ways, but the simplest way is to use an interrupt.

My delay function uses an interrupt that updates a system variable called "Ticker" every millisecond. To calculate delays, I wait for the ticker to count the number of milliseconds I want to wait. My Delay function looks like this:

```
void Delay( WORD MilSec )
   {
   ULONG xdata DelayTickValue;

   /* calculate tick value from now until "MilSec" milliseconds later */
   DelayTickValue = Ticker + MilSec *( (float)(TICKS_PER_SECOND) / 1000.0 );

   /* wait until tick value reaches benchmark */
   while( Ticker < DelayTickValue );
   }
```

The time function is a requirement of Chan's library, and is used for time stamping files generated in the FAT. Since my hardware supports a real time clock, my time function looks like this:

```
DWORD get_fattime()
   {
   RTC_CURRENT rtc;


   RTC_read( &rtc );

   return        ((DWORD)((WORD)(rtc.Year) + 20) << 25)
                  | ((DWORD)rtc.Month << 21)
                  | ((DWORD)rtc.Date << 16)
                  | ((DWORD)rtc.Hours << 11)
                  | ((DWORD)rtc.Minutes << 5)
                  | ((DWORD)rtc.Seconds >> 1);
   }
```

SD Card Initialization

SETTING UP THE CARD FOR SPI COMMUNICATION

Initializing the card, and setting it up for communication is the very first step. These are the most fundamental steps necessary before the card can be accessed. The CardType will be set as either a '0', which is a MMC card, and cannot be used by the SPI interface, a '1'which is an SD card, version 1, or a '2', which is an SD card, version 2

```
BYTE SD_Init()
   {
   WORD CardStatus;           // R2 value from status inquiry...
   WORD Count;                // local counter
                              // Global CardType - b0:MMC, b1:SDv1, b2:SDv2

   /* initial speed is slow... */
   SPI_Init( kHz400 );

   /* disable SPI chip select... */
   SPI_DisableCS();

   /* fill send data with all ones - 80 bits long to establish link with SD card  */
   /* this fulfills the 74 clock cycle requirement...                             */
   for(Count=0;Count<10;Count++)
      SPI_Byte( 0xFF );

   /* enable the card with the CS pin... */
   SPI_EnableCS();

   /* ************************************************************************************ */
   /* SET SD CARD TO SPI MODE - IDLE STATE...                                              */
   /* ************************************************************************************ */
   Count = 1000;     // one second...
   CardType = 0;

   /* wait for card to enter IDLE state... */
   do
      {
      Delay(1);
      SD_Command( CMD_GO_IDLE_STATE, 0 );
      } while((SD_GetR1() != IDLE_STATE) && (--Count));

   /* timeout if we never made it to IDLE state... */
   if( !Count )
      return( SD_TIME_OUT );

   /* ************************************************************************************ */
   /* COMPLETE SD CARD INITIALIZATION - FIGURE OUT WHAT TYPE OF CARD IS INSTALLED...       */
   /* ************************************************************************************ */
   Count = 2000;     // two seconds...

   /* Is card SDSC or MMC? */
   SD_Command( CMD_APP_CMD, 0 );
   SD_Command( ACMD_SEND_OP_COND, 0 );
   if( SD_GetR1() <= 1 )
      {
      CardType = 2;
      }
   else
      {
      CardType = 1;
      }

   /* wait for initialization to finish... */
   do
      {
      Delay(1);
      if( CardType == 2 )
         {
         SD_Command( CMD_APP_CMD, 0 );
         SD_Command( ACMD_SEND_OP_COND, 0 );
         }
      else
         {
         SD_Command( CMD_SEND_OP_COND, 0 );
         }
      } while(SD_GetR1() && --Count);
```

```
   if( !Count )
      return( SD_TIME_OUT );

   /* ************************************************************************************ */
   /* QUERY CARD STATUS...                                                                 */
   /* ************************************************************************************ */
   SD_Command( CMD_SEND_STATUS, 0 );
   CardStatus = SD_GetR2();

   if( CardStatus )
      return( SD_ERROR );

   /* ************************************************************************************ */
   /* SET BLOCK SIZE...                                                                    */
   /* ************************************************************************************ */
   SD_Command( CMD_SET_BLOCKLEN, 512 );
   if( SD_GetR1() )
      {
      CardType = 0;
      return( SD_ERROR );
      }

   /* ************************************************************************************ */
   /* SWITCH TO HIGHEST SPI SPEED...                                                       */
   /* ************************************************************************************ */
   SPI_Init( MHz10 );

   /* disable the card with the CS pin... */
   SPI_DisableCS();

   /* return OK... */
   return( 0 );
   }
```

Reading and Writing a single sector

Another fundamental function is reading a single sector of data from the SD card. Since the sector size is fixed in SPI mode, we expect to read 512 bytes at a time. The following function will achieve this.

```
BYTE SD_ReadSector( ULONG SectorNumber, BYTE *Buffer )
   {
   BYTE c, i;
   WORD count;

   /* send block-read command... */
   SD_Command( CMD_READ_SINGLE_BLOCK, SectorNumber << 9 );
   c = SD_GetR1();
   i = SD_GetR1();
   count = 0xFFFF;

   /* wait for data token... */
   while( (i == 0xff) && --count)
      i = SD_GetR1();

   /* handle time out... */
   if(c || i != 0xFE)
      return( 1 );

   /* read the sector... */
   for( count=0; count<SD_DATA_SIZE; count++)
      *Buffer++ = SPI_Byte(0xFF);

   /* ignore the checksum... */
   SPI_Byte(0xFF);
   SPI_Byte(0xFF);

   /* release the CS line... */
   SPI_DisableCS();

   return( 0 );
   }


BYTE SD_WriteSector( ULONG SectorNumber, BYTE *Buffer )
```

```
   {
   BYTE i;
   WORD count;

   /* send block-write command... */
   SD_Command( CMD_WRITE_SINGLE_BLOCK, SectorNumber << 9 );
   i = SD_GetR1();

   /* send start block token... */
   SPI_Byte( 0xFE );

   /* write the sector... */
   for( count= 0; count< 512; count++ )
      {
      SPI_Byte(*Buffer++);
      }
   /* ignore the checksum (dummy write)... */
   SPI_Byte(0xFF);
   SPI_Byte(0xFF);

   /* wait for response token... */
   while( SPI_Byte( 0xFF ) != 0xFF)

   /* these 8 clock cycles are critical for the card to finish up      */
   /* whatever it's working on at the time... (before CS is released!) */
   SPI_Byte( 0xFF );

   /* release the CS line... */
   SPI_DisableCS();
   SPI_Byte( 0xFF );
   return( 0 );
   }
```

One final function…

Finally, you will need one last function to emulate everything Chan's library needs to talk to the SD card. This function is a simple flush function that is used to make sure the card is ready for the next command.

```
BYTE SD_WaitForReady()
   {
   BYTE i;
   WORD j;

   SPI_Byte( 0xFF );

   j = 500;
   do
      {
      i = SPI_Byte( 0xFF );
      Delay( 1 );
      } while ((i != 0xFF) && --j);

   return( i );
   }
```

Working with diskio.c

The only file you need to modify is diskio.c, and the functions within it. This module is the interface for Chan's library to your SD card. If the functions within this file are written as wrappers to your fundamental functions, Chan's library will work without any problems.

The functions in this module are as follows: disk_initialize, disk_status, disk_read, disk_write, and disk_ioctl. Also note that the only feature within disk_ioctl that is used is CTRL_SYNC. These functions, along with all of the previous functions, will do the job.

```
DSTATUS disk_initialize( BYTE drv )
   {
   /* Supports only single drive */
   if( drv != 0)
      return STA_NOINIT;
```

```
   /* if initialization succeeds... */
   if( !SD_Init() )
      {
      /* Clear STA_NOINIT */
      Stat &= ~STA_NOINIT;
      }

   /* return current status */
   return( Stat );
   }



DSTATUS disk_status( BYTE drv )
   {
   /* Supports only single drive */
   if( drv != 0)
      return STA_NOINIT;

   /* return current status */
   return( Stat );
   }



DRESULT disk_read ( BYTE drv, BYTE *buff, DWORD sector, BYTE count )
   {
   /* Supports only single drive and must have a size of 1 sector */
   if( drv || !count || (count>1) )
      return( RES_PARERR );

   /* if we haven't initialized the card yet... */
   if( Stat & STA_NOINIT )
      return( RES_NOTRDY );

   /* Single block read */
   if( SD_ReadSector( sector, buff ) )
      return( RES_ERROR );

   /* return successful result: OK */
   return( RES_OK );
   }



#if _READONLY == 0
   DRESULT disk_write( BYTE drv, const BYTE *buff, DWORD sector, BYTE count )
      {
      /* Supports only single drive and must have a size of 1 sector */
      if( drv || !count || (count>1) )
         return( RES_PARERR );

      /* if we haven't initialized the card yet... */
      if( Stat & STA_NOINIT )
         return( RES_NOTRDY );

      /* Single block write */
      if( SD_WriteSector( sector, buff ) )
         return( RES_ERROR );

      /* return successful result: OK */
      return( RES_OK );
      }
#endif // _READONLY



DRESULT disk_ioctl ( BYTE drv, BYTE ctrl, void *buff )
   {
   DRESULT res;
   BYTE  *ptr = buff;

   /* Supports only single drive */
   if( drv != 0)
      return RES_PARERR;

   /* if we haven't initialized the card yet... */
   if( Stat & STA_NOINIT )
      return RES_NOTRDY;
```

```
res = RES_ERROR;

switch( ctrl )
   {
   /* Flush dirty buffer if present */
   case CTRL_SYNC :
      SPI_EnableCS();
      if( SD_WaitForReady() == 0xFF )
         res = RES_OK;
      break;

   default:
      res = RES_PARERR;
      break;
   }

SPI_DisableCS();
SPI_Byte( 0xFF );
return res;
}
```

Pulling it all together

Using these functions, along with Chan's tff.c module creates all the tools you need to read and write to a FAT16 SD card. An example program to display the root directory of the SD card would look like this:

```
#include "integer.h"
#include "tff.h"
#include "diskio.h"

FATFS SDCard;

void ShowDirectory( char *path );
char *ShowFatTime( WORD ThisTime );
char *ShowFatDate( WORD ThisDate );
DWORD get_fattime();

void main()
   {
   /* 1) mount drive... */
   if( f_mount( 0, &SDCard ) )
      {
      printf("Couldn't mount drive...\r\n");
      while( 1 );
      }

   /* 2) initialize card... */
   if( disk_initialize( 0 ) & STA_NOINIT )
      {
      switch( CardType )
         {
         case 0 :
            printf("Couldn't find SD card\r\n");
            break;
         case 1 :
            printf("Card type is MMC - Can't use this type\r\n");
            break;
         case 2 :
            printf("Couldn't initialize SD Card drive...\r\n");
            break;
         default :
            printf("Unknown Card Type error...\r\n");
            break;
         }
      while( 1 );
      }

   /* 3) show directory... */
   ShowDirectory("");

   while( 1 );
   }

void ShowDirectory( char *path )
   {
   FILINFO finfo;
   DIR dirs;
   FATFS *fs;
   DWORD clust;
   ULONG TotalSpace, FreeSpace;
   FRESULT res;
   char VolumeLabel[12];

   if(disk_read(0, SDCard.win, SDCard.dirbase, 1) != RES_OK)
      {
      printf("\r\nCouldn't read directory sector...\r\n");
      return;
      }

   strncpy( VolumeLabel, &SDCard.win, 11 );
   VolumeLabel[ 11 ] = 0x00;
   if( f_opendir(&dirs, path) == FR_OK )
      {
      if( VolumeLabel[0] == ' ' )
         printf("\r\n Volume in Drive C has no label.\r\n");
      else
         printf("\r\n Volume in Drive C %s\r\n", VolumeLabel );
      printf(" Directory of C:\\%s\r\n\n", path );
```

```
      while( (f_readdir(&dirs, &finfo) == FR_OK) && finfo.fname[0] )
         {
         putchar('[');
         putchar( finfo.fattrib & AM_RDO ) ? 'r' : '.');
         putchar( finfo.fattrib & AM_HID ) ? 'h' : '.');
         putchar( finfo.fattrib & AM_SYS ) ? 's' : '.');
         putchar( finfo.fattrib & AM_VOL ) ? 'v' : '.');
         putchar( finfo.fattrib & AM_LFN ) ? 'l' : '.');
         putchar( finfo.fattrib & AM_DIR ) ? 'd' : '.');
         putchar( finfo.fattrib & AM_ARC ) ? 'a' : '.');
         putchar(']');

         printf(" %s  %s   ",
            ShowFatDate(finfo.fdate), ShowFatTime( finfo.ftime ));
         printf("%s %6ld %s\r\n", (finfo.fattrib & AM_DIR)?"<DIR>":"     ",
            finfo.fsize, finfo.fname );
         }
      }
   else
      {
      printf("The system cannot find the path specified.\r\n");
      return;
      }

   printf("%cCalculating disk space...\r", 0x09 );

   // Get free clusters
   res = f_getfree("", &clust, &fs);
   if( res )
      {
      printf("\nf_getfree() failed...\r\n");
      return;
      }

   TotalSpace = (DWORD)(fs->max_clust - 2) * fs->csize / 2;
   FreeSpace = clust * fs->csize / 2;
   printf("%c%lu KB total disk space.\r\n", 0x09, TotalSpace );
   printf("%c%lu KB available on the disk.\r\n", 0x09, FreeSpace );
   }


char *ShowFatTime( WORD ThisTime )
   {
   char msg[12];
   BYTE AM = 1;

   int Hour, Minute, Second;

   Hour = ThisTime >> 11;        // bits 15 through 11 hold Hour...
   Minute = ThisTime & 0x07E0;   // bits 10 through 5 hold Minute... 0000 0111 1110 0000
   Minute = Minute >> 5;
   Second = ThisTime & 0x001F;   //bits 4 through 0 hold Second...   0000 0000 0001 1111

   if( Hour > 11 )
      {
      AM = 0;
      if( Hour > 12 )
         Hour -= 12;
      }

   sprintf( msg, "%02d:%02d:%02d %s", Hour, Minute, Second*2,
         (AM)?"AM":"PM");
   return( msg );
   }


char *ShowFatDate( WORD ThisDate )
   {
   char msg[10];

   int Year, Month, Day;

   Year = ThisDate >> 9;         // bits 15 through 9 hold year...
   Month = ThisDate & 0x01E0;    // bits 8 through 5 hold month... 0000 0001 1110 0000
   Month = Month >> 5;
   Day = ThisDate & 0x001F;      //bits 4 through 0 hold day...    0000 0000 0001 1111
   sprintf( msg, "%02d/%02d/%02d", Month, Day, Year-20);
   return( msg );
   }
```

```
DWORD get_fattime()
   {
   RTC_CURRENT rtc;


   RTC_read( &rtc );

   return        ((DWORD)((WORD)(rtc.Year) + 20) << 25)
                  | ((DWORD)rtc.Month << 21)
                  | ((DWORD)rtc.Date << 16)
                  | ((DWORD)rtc.Hours << 11)
                  | ((DWORD)rtc.Minutes << 5)
                  | ((DWORD)rtc.Seconds >> 1);
   }
```