2 Brush Court, Canning Vale, WA 6155, Australia • Tel: +61 894676358 • info@brushelectronics.com
ABN 50087977811

Brush Electronics SD/MMC Embedded FAT File System User Manual

Andrew Smallridge

Contents

Introduction

The Brush Electronics' SD/MMC Embedded FAT file system is a software driver for Microchip microcontrollers enabling the microcontrollers to read and/or write standard and high capacity (SDHC) SD/MMC cards using FAT16 or FAT32 formatted media. FAT file system support enables the media to be exchanged between other devices that support the FAT16 and/or FAT32 file system. For example, files created by an embedded system using this driver, can be read from and/or written to on a Windows PC.

Variants of the file system driver are available for the Microchip PIC18F, PIC24, dsPIC33 and PIC32 families of microcontrollers for associated Microchip C18, C30 and C32 compilers. Variants are also available for the PIC18F, PIC24 and dsPIC33 families using the CCS PCH and PCH compilers.

The file system driver supports the standard 8.3 filename convention used by DOS. This code is an implementation of the ELM Generic File System. The driver has been extensively debugged, modified and enhanced with a DOS like CLI.

The file system driver comprises two driver subsystems, the low level SD/MMC media driver (media_io) for performing low level read and write operations on the media, and the file system driver (ff) for file based access to the media utilizing the FAT16 and/or FAT32 file system. The file system driver is provided with a sample application demonstrating the use of the file system driver for typical use case applications.

For the PIC18F series microcontrollers, the driver is supplied as a proof of concept data logger application featuring a DOS like command line interface and incorporating a software real time clock (optionally used by the file system). The data logger application demonstrates the concept of caching the incoming raw data while SD/MMC write operations of the data are in effect thereby decoupling data acquisition and logging. This sample application is based on a production data logger implementation. The marriage of the data logger capability and the DOS command line interface are intended to demonstrate to the developer how to use the various functions.

The file system drivers support standard (high performance) and LITE configuration modes. The standard mode implements a separate 512 byte read/write buffer for the file system and for each file. The LITE mode, enabled via a #define directive in the source code, offers all the features of the standard mode at a lower level of performance. The LITE mode is well suited to applications and microcontrollers that have limited RAM available. The LITE mode shares a single 512 byte read/write buffer between the file system and all open files and is capable of supporting hundreds of files open concurrently. For high performance applications that require two or more files open concurrently, the standard mode is recommended

RAM requirements:	Standard	Lite
File System overhead	560	560
Per File Overhead	540	38

For the PIC24/dsPIC33 series microcontrollers, the driver is supplied with a sample application featuring a DOS like command line interface.

Customizing the driver

The supplied sample application uses a series of #define directives to customise the file system for the target hardware platform and the operational mode of the driver. This configuration is contained in the **platform.h** file used by the sample application. The platform.h file belongs to the sample application and NOT to the file system driver.

Before a developer attempts to integrate the file system driver into their own application, Brush Electronics **STRONGLY RECOMMENDS** the developer first adapt the supplied sample application to run on their target hardware platform. This will provide the developer the necessary insight into the configuration and operation of the driver. Experience has shown that this approach will significantly reduce the time it takes the developer to integrate the driver directly into their user application. Following this approach, it typically takes less than two hours to have our sample application operational on your target platform.

Understanding the platform.h file

The platform.h file contains target specific configuration information such as processor config fuse settings, oscillator frequency, peripheral pin mapping, peripheral usage, control pin assignments and I/O port configuration. The bulk of this information is used to configure the I/O of the target platform in the init_PIC() function of the sample application. This section will examine the platform.h file and its integration into the sample application. The examples used in this manual will be for the CCS PCD compiler however the concept and principles are the same for the CCS PCH compiler and the Microchip C18, C30 and C32 compilers.

The following is an extract from the sample application supplied with the file system driver for the PCD compiler:

```
#CASE                         // ignore case

#define USE_FAT_LITE          // configure the file system for FAT Lite operation

#include <platform.h>         // include the platform specific I/O configuration

#build (stack=512)            // required for the CCS PCD compiler

#include "DataTypes.h"        // typedefs used by the driver
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include "media-io.h"         // low level SD/MMC media driver
#include "ff.h"               // file system driver

#define get_fattime() 0       // null FAT time function
```

The following is an extract of the platform.h file. This extract covers the selection and configuration of the Microchip Explorer16 fitted with a PIC24HJ256GP610 PIM (processor module):

```
// Specify the target hardware platform being used
#define EXPLORER16_REV4
//#define CCS_DSP_ANALOG_BOARD
     . .

// is this the target platform? (it is)
#if defined EXPLORER16 || defined EXPLORER16_REV4

  // The Explorer 16 supports multiple PIC families and types
  // Select the target microcontroller. This is where processor specific
  // configuration is performed. This includes #fuse setting, clock divisors
  // and SPI prescalar definitions

  #if defined __dsPIC30F3014__
    #include <30F3014.h>
     . .

  #elif defined __PIC24HJ256GP610__
     // This is the target processor
     // This section contains the PROCESSOR SPECIFIC configuration
     #include <24HJ256GP610.h>

     // specify the configuration fuse settings.
     // These configuration settings should be copied from the application
     // fuse settings to be used by the final application.
     #fuses HS, PR_PLL, NOOSCIO, NOWDT, DEBUG, NOWRTB, NOPUT, NOWRTSS
     #fuses NOJTAG, NOPROTECT, NORSS, NOWRT

     // External oscillator frequency
     // the clock frequency and PLL configuration are required to derive the
     // baud rate configuration used by the sample application
     // this information is NOT used by the file system driver
     #define XTAL_FREQ 8000000  // OSC freq in Hz
     #define PLLMODE   10       // On-chip PLL setting
     #use delay(clock=80000000) // used by the CCS compiler to generate delays

     // the following #defines are used by the init_PIC() in the sample
     // application to configure the PIC

     // define PLL multiplier
     #define PLLFBD_Def 38     // default value to be assigned to PLLFBR
     #define CLKDIV_Def 0x0000 // default value to be assigned to CLKDIV

     // Setup Port IOs as digital
     #define AD1PCFGL_Def 0xffff; // default value to be assigned to AD1PCFGL
     #define AD1PCFGH_Def 0xffff; // default value to be assigned to AD1PCFGH

     // Setup SPI bus pre and post scalars
     // During the initial communications with the SD/MC card the SPI
     // clock rate should ideally be less than 400KHz. The SPI_LOW
     // prescalars are used by the media_io driver to configure the
     // low SPI speed clock rate
     #define SPI_LOW_PRESCALE_PRI 0  // Primary Scaler = 64:1
     #define SPI_LOW_PRESCALE_SEC 0  // Secondary Scaler = 8:1


     // During the later stages of the media initialization the clock rate is
     // increased. The SPI_HI prescalars are used by the media_io driver
     // to configure the high SPI speed clock rate
     #define SPI_HI_PRESCALE_PRI 1  // Primary Scaler = 16:1
     #define SPI_HI_PRESCALE_SEC 7  // Secondary Scaler = 1:1
   #endif // end of PIC24HJ256GP610 specific configuration
```

```
        // This is the HARDWARE SPECIFIC configuration
        // Baudrate constants for console port
        #define CONSOLE_BAUDRATE      115200

        // Define UART I/O
        #define CONSOLE_UART   2
        #define CONSOLE_TX_TRIS       TRISFbits.TRISF5
        #define CONSOLE_RX_TRIS       TRISFbits.TRISF4

        ///////////////////////////////////////////////////////////////////////////
        //  Default I/O configuration used by the init_PIC() function
        //  Px_DefData (Port x Default Data) specifies the default output level
        //  for port bits configured as outputs. SPI ports are often shared by
        //  multiple peripherals. It is important that all chip selects are
        //  configured for a logic 1 (high) output to avoid SPI bus conflict.
        //
        //   Px_DefTRIS (Port x Default TRIS) specifies the default I/O
        //   configuration of a port (Input versus Output)
        ///////////////////////////////////////////////////////////////////////////
        #define PA_DefData     0b0000000000000000
        #define PA_DefTRIS     0b1111111100000000

        #define PB_DefData     0b0111000000000010
        #define PB_DefTRIS     0b1000111111111101
                //  1          SD_CS top slot of J5

        #define PC_DefData     0b0000000000000000
        #define PC_DefTRIS     0b1111111111111111

        #define PD_DefData     0b1001000000000000
        #define PD_DefTRIS     0b0110111111111111
                // 15          XEE_CS EXPLORER16 non rev 4
                // 12          XEE_CS EXPLORER16_REV4

        #define PE_DefData     0b0000000000000000
        #define PE_DefTRIS     0b1111111111111111

        #define PF_DefData     0b0000000000101000
        #define PF_DefTRIS     0b1111111010010111
                //  8          SPI1_SDO       SD/MMC SPI SDO
                //  7          SPI1_SDI       SD/MMC SPI SDI
                //  6          SPI1_CLK       SD/MMC SPI CLK
                //  5          U2TX
                //  4          U2RX
                //  3          U1TX
                //  2          U1RX
                //  1          SD_WP          SD/MMC Write Protect switch input
                //  0          SD_CD          SD/MMC Card Detect switch input

        #define PG_DefData     0b0000000000000000
        #define PG_DefTRIS     0b1111111010111111
                //  8          SPI1_SDO       SD/MMC SPI SDO
                //  7          SPI1_SDI       SD/MMC SPI SDI
                //  6          SPI1_CLK       SD/MMC SPI CLK

        ///////////////////////////////////////////////////////////////////////////
        //             SD/MMC PICTail Interfce Connections for Top Slot
        ///////////////////////////////////////////////////////////////////////////
        #define SD_SPI_PORT           1
        #define SD_CS                 LATBbits.LATB1        // SD/MMC Chip Select Active low
        #define SD_CD                 PORTFbits.RF0         // card detect pin

        ///////////////////////////////////////////////////////////////////////////
        //             Status LEDs
        ///////////////////////////////////////////////////////////////////////////
        #define LED_Status            LATAbits.LATA0
        #define LED_Status_TRIS       TRISAbits.TRISA0
        #define LED0                  LED_Status
        #define LED0_TRIS             LED_Status_TRIS
        #define LED1                  LATAbits.LATA1
        #define LED1_TRIS             TRISAbits.TRISA1
        // end of the HARDWARE SPECIFIC configuration
#endif
```

File System API, function prototypes

```
FRESULT f_chmod (char *path, BYTE value, BYTE mask);
FRESULT f_close (FIL *fp);
void    f_get_error_mesg(FRESULT Mesg, char *destination);
FRESULT f_getfree (DWORD *nclust);
FRESULT f_lseek (FIL *fp, DWORD ofs);
FRESULT f_mkdir (char *path);
FRESULT f_mountdrv (void);
FRESULT f_open (FIL *fp, char *path, BYTE mode);
FRESULT f_opendir (DIR *scan, char *path);
FRESULT f_read (FIL *fp, void *buff, WORD btr, WORD *br);
FRESULT f_readdir (DIR *scan, FILINFO *finfo);
FRESULT f_rename ( char *path_old, char *path_new);
FRESULT f_stat (char *path, FILINFO *finfo);
FRESULT f_sync (FIL *fp);
FRESULT f_unlink (char *path);
FRESULT f_write (FIL *fp, void *buff, WORD btw, WORD *bw);
```

F_RESULT File system function return codes

FR_OK	File operation successful
FR_NOT_READY	Media not correctly initialized
FR_NO_FILE	File not found
FR_NO_PATH	Path not found
FR_INVALID_NAME	Invalid file or path name
FR_DENIED	Access denied
FR_DISK_FULL	Disk full
FR_RW_ERROR	Read or write error
FR_INCORRECT_DISK_CHANGE	Media changed without reinitializing the file system
FR_WRITE_PROTECTED	Media is write protected
FR_NOT_ENABLED	Driver has not been initialized - see f_mountdrv()
FR_NO_FILESYSTEM	Not FAT file system found on the media

File Object structure

```
typedef struct _FIL
{
  DWORD fptr;           // File R/W pointer
  DWORD fsize;          // File size
  DWORD org_clust;      // File start cluster
  DWORD curr_clust;     // Current cluster
  DWORD curr_sect;      // Current sector

  #ifndef _FS_READONLY
    DWORD  dir_sect;    // Sector containing the directory entry
    BYTE  *dir_ptr;     // Pointer to the directory entry in the window
  #endif

  BYTE  flag;           // File status flags
  BYTE  sect_clust;     // Left sectors in cluster

  #ifndef USE_FAT_LITE
    BYTE  buffer[512]; // File R/W buffer
  #endif
} FIL;
```

Directory Object structure

```
typedef struct _DIROS
{
  DWORD sclust; // Start cluster
  DWORD clust;  // Current cluster
  DWORD sect;   // Current sector
  WORD  index;  // Current index
} DIR;
```

File Status structure

```
typedef struct _FILINFO
{
  DWORD fsize;         // Size
  WORD  fdate;         // Date
  WORD  ftime;         // Time
  BYTE  fattrib;       // Attribute
  char  fname[8+1+3+1];       // Name (8.3 format)
} FILINFO;
```

File System Functions

f_chmod()

Prototype:	`FRESULT f_chmod(char *path, BYTE flags, BYTE mask);`
Syntax	`result = f_chmod(path, flags, mask);`
Parameters	**path** is a pointer to the fully qualified 8.3 formatted null terminated filename path string **flags** attributes to be set **mask** attributes to be cleared
Returns	**FRESULT** return code
Function	Modify the file system attributes of a file

Example	`FRESULT result;`
	`    ..`
	`// set the READ ONLY attribute, clear the HIDDEN and ARCHIVE attributes`
	`result = f_chmod(path, AM_RDO, AM_HID | AM_ARC);`
	`if (result)`
	`   printf("File attribute error\r\n");`

f_chmod() attribute flags

AM_ARC	Archive
AM_DIR	Directory
AM_HID	Hidden
AM_RDO	Read only
AM_SYS	System
AM_VOL	Volume Label

f_close()

Prototype:	FRESULT f_close (FIL *fp);
Syntax	result = f_close(&fdata);
Parameters	**fp** is a pointer to a FIL file control block (handle) of the file to be closed
Returns	**FRESULT** return code
Function	Closes the file and flushes the file control block to the media

Example	FIL fdata;
	FRESULT result;
	   ..
	result = f_close(&fdata);
	if (result)
	   printf("File close error\r\n");

f_get_error_mesg()

Prototype:	void f_get_error_mesg(FRESULT Mesg, char *destination);
Syntax	f_get_error_mesg(FRESULT result, char *destination);
Parameters	**Mesg FRESULT** error code **destination** character array to hold the returned NULL terminated message
Returns	
Function	Convert the FRESULT error code into an ASCII NULL terminated string

Example	FIL fdata;
	char destination[64];
	FRESULT result;
	   ..
	f_get_error_mesg(result, destination);
	printf("FRESULT error %s\r\n",destination);

f_getfree()

Prototype:	FRESULT f_getfree (DWORD *nclust);
Syntax	result = f_getfree(&nclust);
Parameters	**nclust** number of free clusters on the media
Returns	**FRESULT** return code
Function	Returns the number of free clusters

Example	DWORD nclust;
	FRESULT result;
	   ..
	result = f_getfree(&nclust);
	if (result)
	   printf("File system error\r\n");

f_lseek()

Prototype:	FRESULT f_lseek (FIL *fp, DWORD ofs);
Syntax	result = f_lseek(&fdata,ofs);
Parameters	**fp** is a pointer to a FIL file control block (handle) **ods** is the desired offset from the start of the file
Returns	**FRESULT** return code
Function	Moves the file pointer. Typically used to seek to the end of the file in order to append to the end of an existing file.

Example	FIL fdata;
	FRESULT result;
	    ..
	// open the file. If the file does not exist then create it
	result = f_open(&fdata, path, FA_OPEN_ALWAYS | FA_WRITE);
	if (result)
	   printf(“File open error\r\n”);
	
	// seek to the end of the file
	if (fdata.fsize != 0)
	   result = f_lseek(&fdata, fdata.fsize);

f_mkdir()

Prototype:	FRESULT f_mkdir(char *path);
Syntax	result = f_mkdir(fname);
Parameters	**path** is a pointer to the fully qualified 8.3 formatted null terminated filename path string
Returns	**FRESULT** return code
Function	Creates a directory

Example	FRESULT result;
	    ..
	result = f_mkdir(path);
	if (result)
	   printf(“Directory creation error\r\n”);

f_mountdrv()

Prototype:	FRESULT f_mountdrv(void);
Syntax	result = f_mountdrv();
Parameters	
Returns	**FRESULT** return code
Function	Initializes the media and the file system

Example	BYTE x;
	FRESULT result;
	
	// initialize the file system
	x = 10; // maximum number of retries allowed
	do
	{
	   // waiting for the media to initialize
	   result = f_mountdrv();
	   if (result)
	      delay_ms(200); // delay and then retry
	} while (result && x);

f_open()

Prototype:	FRESULT f_open (FIL *fp, char *path, BYTE mode);
Syntax	result = f_open(&fdata,fname,modeflags);
Parameters	**fp** is a pointer **FIL** file control block (handle) **path** is a pointer to the fully qualified 8.3 formatted null terminated filename path string **mode** is the read/write/create file access mode flags
Returns	**FRESULT** return code
Function	Opens or creates a file for read and / or write operations

Example	`FIL fdata;`
	`FRESULT result;`
	`   ..`
	`result = f_open(&fdata, path, FA_OPEN_ALWAYS | FA_WRITE);`
	`if (result)`
	`   printf("File open error\r\n");`

f_open() Mode Flags

FA_CREATE_ALWAYS	Creates a new file. If the file already exists it is truncated and overwritten.
FA_OPEN_ALWAYS	Opens the file if it exists, creates it otherwise. To append data to the file, use f_lseek() function after file open with this mode flag.
FA_OPEN_EXISTING	Opens the file. The function fails if the file does not exist. To append data to the file, use f_lseek() function after file with this mode flag.
FA_READ	Specifies read access to the object. Data can be read from the file. Combine with FA_WRITE for read-write access.
FA_WRITE	Specifies write access to the object. Data can be written to the file. Combine with FA_READ for read-write access

f_opendir()

Prototype:	FRESULT f_opendir (DIR *scan, char *path)
Syntax	result = f_opendir(scan, path);
Parameters	**scan** is a pointer to a **DIR** object data structure **path** is a pointer to the directory string
Returns	**FRESULT** return code
Function	Opens an existing directory and populates the **DIR** structure

Example	`DIR scan;`
	`FRESULT result;`
	`   ..`
	`result = f_opendir(&scan, path);`
	`if (result)`
	`   printf("Directory open error\r\n");`

f_read()

Prototype:	FRESULT f_read (FIL *fp, void *buff, WORD btr, WORD *br);
Syntax	result = f_read(&fdata,buffer,btr,&br);
Parameters	**fp** is a pointer to a FIL file control block (handle) **buff** is a pointer to a buffer for the data being fetched **btr** is the number of Bytes To Read from the file into the buffer **br** is the actual number of bytes read. A **br** value less than **btr** or a value of 0 can be used as an indication the end of file has been reached.
Returns	**FRESULT** return code
Function	Reads the specified amount of data from a file into a buffer

Example	FIL fdata;
	BYTE buff[64];
	BYTE br;
	FRESULT result;
	   ..
	result = f_read(&fdata, buff, 64, &br);
	if (result)
	   printf(“File read error\r\n”);

f_readdir()

Prototype:	FRESULT f_readdir (DIR *scan, FILINFO *finfo);
Syntax	result = f_readdir (&dir, &finfo);
Parameters	**scan** is a pointer to a **DIR** object data structure **finfo** is a pointer to a **FILINFO** file information data structure
Returns	**FRESULT** return code
Function	Reads a directory item. Repeated calls read successive items from the target directory.

Example	DIR dir;
	FILINFO finfo;
	FRESULT result;
	   ..
	result = f_readdir(&dir, &finfo);
	if (result)
	   printf(“Directory read error\r\n”);
	else
	  ..

f_rename()

Prototype:	FRESULT f_rename(char *path_old, char *path_new);
Syntax	result = f_open(&fdata,fname,modeflags);
Parameters	**path_old** is a pointer to the fully qualified 8.3 formatted null terminated filename path string of the existing file **path_new** is a pointer to the fully qualified 8.3 formatted null terminated filename path string of the new file or directory
Returns	**FRESULT** return code
Function	Renames a file or directory

Example	FRESULT result;
	   ..
	result = f_rename(path_old, path_new);
	if (result)
	   printf(“File rename error\r\n”);

f_stat()

Prototype:	`FRESULT f_stat (char *path, FILINFO *finfo);`
Syntax	`result = f_stat(path,&finfo);`
Parameters	**path** is a pointer to the fully qualified 8.3 formatted null terminated filename path string **finfo** is a pointer to a **FILINFO** file information data structure
Returns	**FRESULT** return code
Function	Returns the file status of the target file or directory. The information is populated into the **FILINFO** data structure pointed to be **finfo**

Example	`FIL fdata;`
	`FILINFO finfo;`
	`FRESULT result;`
	`   ..`
	`result = f_open(path, &finfo);`
	`if (result)`
	`   printf("File status error\r\n");`

f_sync()

Prototype:	`FRESULT f_sync(FIL *fp);`
Syntax	`result = f_sync(&fdata);`
Parameters	**fp** is a pointer to a FIL file control block (handle)
Returns	**FRESULT** return code
Function	Flushes cached file data to the media. Used to force the synchronization between the file system and media. Typically used in data logging applications when small data samples are logged infrequently and data loss of the data in the file buffer could occur is the power failed or the media was removed without closing the file. f_close() automatically invokes f_sync()

Example	`FIL fdata;`
	`FRESULT result;`
	`   ..`
	`result = f_sync(&fdata);`
	`if (result)`
	`   printf("File sync error\r\n");`

f_unlink()

Prototype:	`FRESULT f_unlink(char *path);`
Syntax	`result = f_unlink(fname);`
Parameters	**path** is a pointer to the fully qualified 8.3 formatted null terminated filename path string
Returns	**F_RESULT** return code
Function	Delete a file or an empty directory

Example	`FRESULT result;`
	`   ..`
	`result = f_unlink(path);`
	`if (result)`
	`   printf("File delete error\r\n");`

f_write()

Prototype:	FRESULT f_write (FIL *fp, void *buff, WORD btw, WORD *bw);
Syntax	result = f_write(&fdata,buffer,btw,&bw);
Parameters	**fp** is a pointer to a FIL file control block (handle) **buff** is a pointer to a buffer for the data being written **btw** is the number of **b**ytes **t**o **w**rite to the file from the buffer **bw** is the actual number of bytes written. A **bw** value less than **btw** is an indication of an error.
Returns	**FRESULT** return code
Function	Writes the specified amount of data from the buffer to the file

Example	
	```
	FIL fdata;
	BYTE buff[64];
	BYTE bw;
	FRESULT result;
	   ..
	result = f_write(&fdata, buff, 64, &bw);
	if (result)
	   printf(“File write error\r\n”);
	```

Sample Application

```
#CASE

#define USE_FAT_LITE   // configure FAT Lite operation

#include <platform.h>  // platform specific I/O configuration

#build (stack=512)

#include "DataTypes.h"
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include "media-io.h"
#include "ff.h"

#define get_fattime() 0

void file_list(char *ptr)
///////////////////////////////////////////////////////////////////////////
// void file_list(char *ptr)
//
//     Lists the contents of a text file
///////////////////////////////////////////////////////////////////////////
{
   FIL fsrc;
   FRESULT result;      // FatFs function common result code
   char mesg[32];

   result = f_open(&fsrc, ptr, FA_OPEN_EXISTING | FA_READ);

   // display the contents of the file
   if (result == FR_OK)
   {
      WORD i, br;

      do
      {
         result = f_read(&fsrc, mesg, sizeof(mesg), &br);
         for (i = 0; i < br; i++)
            putc(mesg[i]);
      } while ((result == FR_OK) && br);

      if (result != FR_OK)
      {
         printf("TYPE command ERROR\r\n");
         f_get_error_mesg(result,mesg);
         printf("FILE SYSTEM ERROR - %s\r\n",mesg);
      }

      // Close all files
      f_close(&fsrc);
      printf("\r\n");
   }
   else
   {
      f_get_error_mesg(result,mesg);
      printf("FILE SYSTEM ERROR - %s\r\n",mesg);
   }
}
```

```
void init_pic(void)
////////////////////////////////////////////////////////////////////////////
// void init_pic(void)
//
// Initialise the hardware defaults
////////////////////////////////////////////////////////////////////////////
{
   setup_spi(FALSE);
   setup_spi2(FALSE);
   setup_wdt(WDT_OFF);

   #if defined AD1PCFG_Def
      AD1PCFG = AD1PCFG_Def;
   #endif

   #if defined AD1PCFGL_Def
      AD1PCFGL = AD1PCFGL_Def;
   #endif

   #if defined AD1PCFGH_Def
      AD1PCFGH = AD1PCFGH_Def;
   #endif

   #if defined PA_DefData
      // initialise port A
      PORTA = PA_DefData;
      TRISA = PA_DefTRIS;
   #endif

   #if defined PB_DefData
      // initialise port B
      PORTB = PB_DefData;
      TRISB = PB_DefTRIS;
   #endif

   #if defined PC_DefData
      // initialise port C
      PORTC = PC_DefData;
      TRISC = PC_DefTRIS;
   #endif

   #if defined PD_DefData
      // initialise port D
      PORTD = PD_DefData;
      TRISD = PD_DefTRIS;
   #endif

   #ifdef PE_DefData
      PORTE = PE_DefData;
      TRISE = PE_DefTRIS;
   #endif

   #ifdef PF_DefData
      PORTF = PF_DefData;
      TRISF = PF_DefTRIS;
   #endif

   #ifdef PG_DefData
      PORTG = PG_DefData;
      TRISG = PG_DefTRIS;
   #endif

   #ifdef PH_DefData
      PORTH = PH_DefData;
      TRISH = PH_DefTRIS;
   #endif

   #ifdef PJ_DefData
      PORTJ = PJ_DefData;
      TRISJ = PJ_DefTRIS;
   #endif
   #if defined PLLFBD_Def
      PLLFBD = PLLFBD_Def;
   #endif

   #if defined CLKDIV_Def
      CLKDIV = CLKDIV_Def;
   #endif
   #if defined UART1_TX_RPR
      UART1_TX_RPR = 3;       // UART1 TX
   #endif
   #if defined UART1_RX_RP
      UART1_RX_RPR =  UART1_RX_RP;
   #endif

   #if defined UART2_TX_RPR
      UART2_TX_RPR = 5;       // UART2 TX
   #endif

   #if defined UART2_RX_RP
      UART2_RX_RPR =  UART2_RX_RP;
   #endif

   #if defined SPI1_SDI_RP
      SPI1_SDI_RPR = SPI1_SDI_RP;
      SPI1_SDO_RPR = 7;       // SPI1 SDO
      SPI1_CLK_RPR = 8;       // SPI1 CLK
   #endif

   #if defined SPI2_SDI_RP
      SPI2_SDI_RPR = SPI2_SDI_RP;
      SPI2_SDO_RPR = 10;      // SPI2 SDO
      SPI2_CLK_RPR = 11;      // SPI2 CLK
   #endif
}



BYTE write_test (char *ptr)
///////////////////////////////////////////////////////////////////////////
// BYTE write_test (char *ptr)
//
// Opens the file pointed to by ptr and writes to it. If the file exists it
// is appended to. If it does not exist it is created. The file this
// then closed.
///////////////////////////////////////////////////////////////////////////
{
   FIL fsrc;           // file structures
   FRESULT result;      // FatFs function common result code
   WORD btw, bw;        // File R/W count

   char mesg[64];

   // open the file - assumes target is the filename
   result  = f_open(&fsrc, ptr, FA_OPEN_ALWAYS | FA_WRITE);
   if (result)
   {
      printf("write_test FS ERROR on file_open\r\n");
      goto Ex_write_test_no_close;
   }

   // Move to end of the file to append data
   result = f_lseek(&fsrc, fsrc.fsize);

   // write a short string to destination file
   strcpy(mesg, "My String!");
   btw = strlen(mesg);
   result = f_write(&fsrc, mesg, btw, &bw);
   if (result)
   {
      printf("write_test FS ERROR on f_write\r\n");
      goto Ex_write_test;
   }
   // write another short string to destination file
   strcpy(mesg, "\r\nHello \r\n");
   btw = strlen(mesg);
   result = f_write(&fsrc, mesg, btw, &bw);
   if (result)
   {
      printf("write_test FS ERROR on second f_write\r\n");
      goto Ex_write_test;
   }

   // example of syncing the file system
   // not really needed here becasue we are about the
   // close the file anyway
   printf("syncing the file\r\n");
   result = f_sync(&fsrc);
   if (result)
   {
      printf("write_test: Error returned from f_sync\r\n");
      goto Ex_write_test;
   }

   printf("about to close the file\r\n");
   f_close(&fsrc);
   printf("\r\n");
   return(result);


Ex_write_test:
   // Close the file
   f_close(&fsrc);
   return (result);

Ex_write_test_no_close:
   printf("\r\n");
   return (result);
}
```

```
void main ()
{
   FRESULT     FS_Status;
   char target[16];

   disable_interrupts(INTR_GLOBAL);

   init_pic();

   // initialise the interrupts
   // this will intialise both the global and priority interrupts
   clear_interrupt(INT_RDA);
   enable_interrupts(INT_RDA);
   enable_interrupts(INTR_GLOBAL);

   printf("\r\nBasic Application - compiled %s %s\r\n\r\n", __DATE__,__TIME__);

   // initialise the media and filesystem
   // this will loop until the card is found
   do
   {
      FS_Status = f_mountdrv();
      Delay_ms(200);
   }
   while (FS_Status);


   // build the file name
   strcpy(target,"event.log");

   printf("Performing write testing..\r\n");
   write_test(target);

   printf("Displaying the contents of the file\r\n");

   // display the file contents
   file_list(target);

   printf("That's all Folks..\r\n");
   for (;;)
      ;

}// End of main()...
```

SD/MMC Card Integration and Troubleshooting

SD/MMC cards provide a low cost solution for data logging and storage applications for embedded systems. SD/MMC cards can be easily interfaced with a Microcontroller using an SPI interface and between one and three control lines.

The driver includes the low level SD/MMC SPI drivers, the low level disk I/O drivers and file level operations. The sample application, where applicable, demonstrates a DOS like user interface. For information on how to implement the electrical interface between an SD/MMC card and a PIC microcontroller, refer to our hardware reference designs located on our projects page:

http://www.brushelectronics.com/index.php?page=projects

For a 3.3volt powered PIC system, the minimum interface between an SD/MMC card and the PIC is four I/O lines which include the three SPI bus lines (SCK, SDO and SDI) the fourth line is the chip select. Optionally two additional lines connect to the SD/MMC socket for Card Detect (CD) and Write Protect (WP). For 5 volt powered PICs, level translation is required between the 5 volt I/O of the PIC and the 3.3 volt I/O of the SD/MMC Card. Level translation for the PIC SDO, SCK and CS outputs can be implemented with simple resistor voltage dividers. Coming from DO of the SD/MMC Card to the SDI input of the PIC is not so straight forward and requires a TTL buffer. This is because the PIC, in SPI mode, has the SDI input configured as a Schmidt trigger input and the guaranteed logic high out of the SD/MMC card is less than the guaranteed logic high level of the PIC.

When selecting a PIC for applications that write to an SD/MMC card, there are three main criteria; there is sufficient RAM to support the driver mode, there is sufficient program memory to accommodate the file system and the PIC supports an SPI bus. SD/MMC Cards must be written in 512 byte blocks. This is also the sector size of the SD/MMC card. For example, to append a byte to a file, the sector must be read into a sector buffer, the appropriate location modified in the sector buffer, and the sector written back to the media.

If using the Brush Electronics' file system driver in standard mode (high performance), a 512 byte sector buffer is required for file system management and one 512 byte buffer is required for each open file. This means a PIC with 1500 bytes of RAM, such as the PIC18F452, when using the standard file system configuration mode, can realistically have only a single file open at a time. The PIC18F4620, which is pin compatible with the PIC18F452 has 3900 bytes of ram available and therefore can have multiple files open simultaneously.

When the file system driver is configured for LITE mode, a single sector buffer is shared by the file system and all open files. This means a PIC with 1500 bytes of memory, such as the PIC18F452, can have multiple files open simultaneously. The LITE configuration mode is not recommended for high performance applications that require access to multiple open files.

While the electrical interface is relatively straight forward, successfully implementing a solution can be time consuming for the initial implementation. This section looks at some of the common pitfalls encountered.

For developers that are implementing an SD/MMC interface for the first time on their own hardware platform, Brush Electronics **STRONGLY** recommends using our SD/MMC Card driver and test utilities software (http://www.brushelectronics.com/index.php?page=software#SDUTIL) for bringing up new hardware and gain familiarity with the SD/MMC integration before moving on to implementing a File System.

The troubleshooting guidelines are split into three different areas:

- Power Supply
- SPI BUS
- Media

Power Supply

SD/MMC cards provide a relatively low power storage solution for embedded controllers however implementers often do not pay enough attention to the peak power requirements and inrush currents and may encounter situations where the embedded system resets when an SD/MMC card is inserted or the SD/MMC card may stop communicating with the microcontroller.

The power supply circuitry must be able to deal with the momentary inrush current when a card is inserted. This inrush current can be of the order of 200mA. Generally this is handled using large value power supply filter capacitors before and after the voltage regulator. Ignoring the requirements of other components of an embedded system, typical values for the power supply input capacitor are in the range of 220uF to 470uF. Alternatively, lower value capacitors that feature low ESR characteristics (Effective Series Resistance) could be used. A typical filter capacitor after the regulator is a 47uF tantalum capacitor.

In situations where the SD/MMC card socket is not located in close proximity to the regulator filter capacitors, an additional 22uF tantalum capacitor should be placed as close as practical to the SD/MMC card socket between the power supply pins of the socket that correspond to pins 3 and 4 of the media. **CAUTION**: although SD/MMC card socket pin numbers often match the pin-outs of the SD/MMC media, this is not always the case.

A 100nF power supply decoupling capacitor should be placed as close as practical to the SD/MMC card socket between the power supply pins of the socket that correspond to pins 3 and 4 of the media.

SPI Bus

The SPI bus interface between the Microcontroller and the SD/MMC card is straight forward however it is the area that causes the most problems for first time implementers. The interface includes the following mandatory signals.

Microcontroller		**SD/MMC Card**	
SDI	(input)	DO	(output)
SDO	(output)	DI	(input)
SCK	(output)	SCK	(input)

CS	(output)	CS	(input)

The microcontrollers SPI hardware interface drives the SDI, SDO and SCK control lines. The CS line from the PIC is an I/O pin configured as an output that provides the active low chip select input of the SD/MMC card. Two optional control lines from the SD/MMC card to the microcontroller are the CD (Card detect) and WP (Write protect) switch outputs. By convention, these outputs are active low.

In dual voltage systems where the microcontroller runs at a different supply voltage to the SD/MMC card, level conversion logic is required between the microcontroller and the SD/MMC cards. There are multiple ways of implementing level conversion. The following link gives an example using a pair of 74ACT125M transceivers to perform the level conversion: http://www.ljcv.net/picnet1/picnet1-ds0.pdf Here is another example using a single transceiver combined with resistors: http://www.brushelectronics.com/download/BE_Reference_Design_PIC18F4620_ENC28J60.zip

The SPI bus must be correctly terminated. The SDI, CS, CD and WP lines all require pull-up resistors in the range of 10K to 100K. A common omission is the pull-up resistor on the SDI line which can result in intermittent problems with the SD/MMC initialisation sequence. The SPI BUS frequency should in principle be less than or equal to 400KHz for the initialisation phase of the card and then can ramped up to the desired clock frequency, typically 10MHz, for subsequent operation. Operating the SPI bus beyond 20MHz can result in read/write errors.

The SPI bus is often a shared system resource. An SPI bus conflict occurs when two or more chip select lines to peripherals sharing the SPI bus are asserted concurrently. This can result in read / write errors and can result in an SPI device isolating itself from the SPI bus. SPI bus conflicts is a common first time implementation problem which typically occurs when bringing up a new system during development.

Media Format

SD/MMC media are shipped pre formatted from the manufacturer. The Brush Electronics SD/MMC File System Drivers and the Utilities software interpret the data structures that are present on the SD/MMC card.

During the format process, some operating systems examine the data structures that exist on a card and format the card based on the existing structures. In rare situations it is possible that the master boot record containing a partition table structure is corrupt and is not correctly dealt with by the format programs. In this event the Brush Electronics software may not be able to correctly interpret and file system structure and a file system read error may result. If this occurs, format the media with a device that does perform a complete low level format, such as a digit camera.

Licensing Considerations

Aside for hardware and software challenges, the integration of SD and MMC cards in embedded systems is not without it headaches. Licenses for integration of SD Cards or MMC Cards into products may be required from Microsoft, IBM, the SD Card Association (for SD Cards) and the Multimedia Association (for MMC Cards).

Microsoft has been granted patents covering the File Allocation Table (FAT) file system. These patents pertain to the integration of long file names with legacy DOS file names in the same data structure. In order to avoid infringing on Microsoft's patents in this area, our software does not support long file names nor does it support formatting the media. IBM holds patents pertaining to extended file system attributes. Our code does not support extended file attributes.

It is up to the purchaser of this software to ascertain for themselves if licenses are required from any or all of these or other organizations for the integration of the media or our software into their products. In the event licenses are required from any of these or other organizations, it is up to the purchaser of this software to acquire, at their own cost, such licenses directly from the license holders.

Brush Electronics
2 Brush Court
Canning Vale
Western Australia 6155
Australia

Tel: +61 (0) 894676358
Email: info@brushelectronics.com
www: www.brushelectronics.com