AN1045

Implementing File I/O Functions Using Microchip's Memory Disk Drive File System Library

Authors: *Peter Reen and Naveen Mohanswamy*
Microchip Technology Inc.

INTRODUCTION

This application note describes the usage of file I/O functions using Microchip's memory disk drive file system library. Microchip's memory disk drive file system is:

- Based on ISO/IEC 9293 specifications
- Known as the FAT16 file system, used on earlier DOS operating systems by Microsoft® Corporation
- Also supports the FAT32 file system
- Most popular file system with SD (Secure Digital) card, CF (CompactFlash®) card and USB thumb drive

Most SD cards and MMCs (MultiMedia Cards), particularly those sized below 2 gigabytes (GBs), use the FAT16 standard. The FAT32 standard can be used to address memory sized between 2 gigabytes and 2 terabytes. This application note provides a method to read and/or write to these storage devices through a microcontroller. The data of these storage devices can be read by a PC, and the data written by a PC can be read by a microcontroller. Most operating systems (i.e., Windows® XP) support the FAT16 and FAT32 file systems.

SD CARDS AND MMCs

SD cards and MMCs are proprietary and removable Flash technology-based media licensed by the SD Card Association and the MM Card Association (see **"References"**).

Functionally, the two card formats are similar. However, the SD card has optional encryption security features that are not customarily found on the MMC. The specifications of these devices and the terms and conditions for their use vary, and should be verified for further application licensing information.

INTERFACE

The PICtail™ Daughter Board for SD and MMC, Microchip product number AC164122, provides an interface between SD or MMC and a PIC® microcontroller through the Serial Peripheral Interface (SPI) bus. The PICtail Daughter Board is designed to operate with a multitude of demonstration boards, including all those having PICtail or PICtail Plus Daughter Board interfaces.

The SPI protocol uses four basic pins for communication: Serial Data In (SDI), Serial Data Out (SDO), Serial Clock (SCK), and Chip Select ($\overline{CS}$). Additionally, all SD card sockets have two electrically determined signals, card detect and write-protect that allow the user to determine if the card is physically inserted and/or write-protected.

The MMC does not have a physical write-protect signal, but most card connectors will default to a non-write-protected state in this case.

For more information about interfacing PIC microcontrollers to SD cards or MMCs, refer to *AN1003,* "*USB Mass Storage Device Using a PIC® MCU*" (DS01003) available on the Microchip web site (www.microchip.com).

Important:	It is the user's responsibility to obtain a copy of, familiarize themselves fully with, and comply with the requirements and licensing obligations applicable to third party tools, systems and/or specifications including, but not limited to, Flash-based media and FAT file systems available from CompactFlash Association, SD Card Association, MultiMediaCard Association and Microsoft Corporation. Refer to the license agreement for details.

CARD FILE SYSTEM

A FAT16 file system stores data in sectors. A sector size of 512 bytes is common. Since the number of available memory addresses is capped at FFFFh, sectors can be grouped into clusters that share an address to increase the size of the card.

The first sector on a card is the Master Boot Record (MBR). The MBR contains information about different logical subdivisions on a card, known as partitions. Each partition can be formatted with a unique file system. Typically, an SD card or MMC has only one active partition, which comprises the following parts:

- Boot Sector – This is the first sector of the partition and contains basic information about the file system type.
- FAT Regions – This region is the map of the card, which indicates how the clusters are allocated in the data region. Generally, there are two copies of the FAT in this region to provide redundancy in case of data corruption.
- Root Directory Region – In the FAT16 file system, this region follows the FAT region. In the FAT32 file system, the root is an ordinary cluster chain and can be located anywhere on the volume. The root directory is composed of a directory table that contains entries for subdirectories and files. Other directories and files have entries in the directory tables of the directories in the root.

Collectively, the first three sections are the system area. The remaining space is the data region.

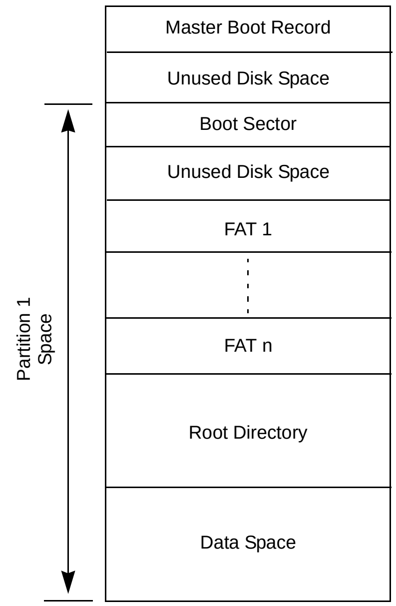
- Data Region – Stores file data or subdirectory directory tables. The data stored in this region remains intact even if it is deleted or until it is overwritten.

The FAT16 system uses 16-bit FAT entries, allowing approximately 65,536 (2^{16}) clusters to be represented; the FAT32 system uses 32-bit FAT entries (effectively only 28 bits) allowing approximately 268,435,456 (2^{28}) clusters to be addressed.

A signed byte in the boot sector defines the number of sectors per cluster for a disk. This byte has a range of -128 to 127. The only usable values in the FAT file system are positive, power-of-two values (1, 2, 4, 8, 16, 32 and 64). This means with the standard 512-byte sector, the FAT16 file system can support a maximum of 2 GB disk space.

The memory structure of an SD card or an MMC is illustrated in Figure 1.

FIGURE 1: DISK STRUCTURE



Legend: n = number of FATs.

Master Boot Record

The MBR contains information that is used to boot the card and information about the partitions on the card. The information in the MBR is programmed at the time of manufacture and any attempt to write to the MBR could render the disk unusable.

Table 1 provides the contents of the MBR.

TABLE 1: CONTENTS OF THE MBR

Offset	Description	Size
000h	Boot Code (machine code and data).	446 bytes
1BEh	Partition Entry 1.	16 bytes
1CEh	Partition Entry 2.	16 bytes
1DEh	Partition Entry 3.	16 bytes
1EEh	Partition Entry 4.	16 bytes
1FEh	Boot Signature Code (55h AAh).	2 bytes

Partition Entry in the MBR

A partition table entry of the master boot record contains the Information about a partition on the disk. A file system descriptor is included in the entry to indicate which type of file system was specified when the partition was formatted. The following file descriptor values indicate the FAT16 formatting:

- 04h (16-bit FAT, < 32M)
- 06h (16-bit FAT, ≥ 32M)
- 0Eh (DOS CHS mapped)

SD cards generally contain a single active partition.

Table 2 provides the contents of a partition table entry.

TABLE 2: PARTITION TABLE ENTRY

Offset	Description	Size
00h	Boot Descriptor (80h if active partition, 00h if inactive).	1 byte
01h	First Partition Sector.	3 bytes
04h	File System Descriptor.	1 byte
05h	Last Partition Sector.	3 bytes
08h	Number of Sectors between the Master Boot Record and the First Sector of the Partition.	4 bytes
0Ch	Number of Sectors in the Partition.	4 bytes

Boot Sector

This is the first sector of a partition. It contains file system information and pointers to important parts of the partition. The first entry in the boot sector is a command to jump past the boot information.

Table 3 provides the entire content of the boot sector.

TABLE 3: BOOT SECTOR ENTRY

Offset	Description	Size
00h	Jump Command.	3 bytes
03h	OEM Name.	8 bytes
0Bh	Bytes per Sector.	2 bytes
0Dh	Sectors per Cluster.	1 byte
0Eh	Total Number of Reserved Sectors.	2 bytes
10h	Number of File Allocation Tables.	1 byte
11h	Number of Root Directory Entries.	2 bytes
13h	Total Number of Sectors (bits 0-15 out of 48).	2 bytes
15h	Media Descriptor.	1 byte
16h	Number of Sectors per FAT.	2 bytes
18h	Sectors per Track.	2 bytes
1Ah	Number of Heads.	2 bytes
1Ch	Number of Hidden Sectors.	4 bytes
20h	Total Number of Sectors (bits 16-47 out of 48).	4 bytes
24h	Physical Drive Number.	1 byte
25h	Current Head.	1 byte
26h	Boot Signature.	1 byte
27h	Volume ID.	4 bytes
2Bh	Volume Label.	11 bytes
36h	File System Type (not for determination).	8 bytes
1FEh	Signature (55h, AAh).	2 bytes

Root Directory

The root directory stores file and directory information in 32-byte entries. Each entry includes the filename, file size, the address of the first cluster of the file and the time the file was created or modified.

In the FAT16 file system, the root directory region is located after the FAT region. In the FAT32 file system, the root is an ordinary cluster chain and can be located anywhere on the volume.

Note:	Generally, a file entry conforms to "eight dot three" short filename format. Only digits, 0 to 9, letters, A to Z, the space character and special characters, '! # $ % & ( ) - @ ^ _ ` { } ~ ',', are used. Although it is customary to consider the period (.) and extension as elements of the filename, in this case, none of the characters after the initial name are used as part of the actual filename. For example, a file named `FILE.txt` would have the filename `FILE_ _ _ _` in the root directory with the final four characters replaced by four instances of the space character, 20h.

Table 4 provides the contents of a root directory entry.

TABLE 4: ROOT DIRECTORY ENTRIES

Offset	Description	Size
00h	Filename[(1)].	8 bytes
08h	File Extension.	3 bytes
0Bh	File Attributes.	1 byte
0Ch	Reserved.	1 byte
0Dh	File Creation Time (ms portion).	1 byte
0Eh	File Creation Time (hours, minutes and seconds).	2 bytes
10h	File Creation Date.	2 bytes
12h	Last Access Date.	2 bytes
14h	Extended Address-Index.	2 bytes
16h	Last Update Time (hours, minutes and seconds).	2 bytes
18h	Last Update Date.	2 bytes
1Ah	First Cluster of the File.	2 bytes
1Ch	File Size.	4 bytes

Note 1: The first character of the filename can take on special values (see Table 5).

TABLE 5: POSSIBLE VALUES FOR THE FIRST CHARACTER IN THE DIRECTORY FILENAME

Value	Description
00h	This entry is available and no subsequent entry is in use.
E5h	The file in this entry was deleted and the entry is available.
05h	The first character in the filename is E5h.
2Eh	This entry points to the current or previous directory.

File Allocation Table

The FAT has space for an entry that corresponds to every cluster in the data cluster section of the partition. This entry would be 2 bytes in case of FAT16 and 4 bytes in the FAT32 file system. For example, the third set of two bytes in the FAT will correspond to the first cluster in the data region.

Figure 2 illustrates an example of this. A value placed in each position can indicate many things.

Table 6 provides a list of FAT values.

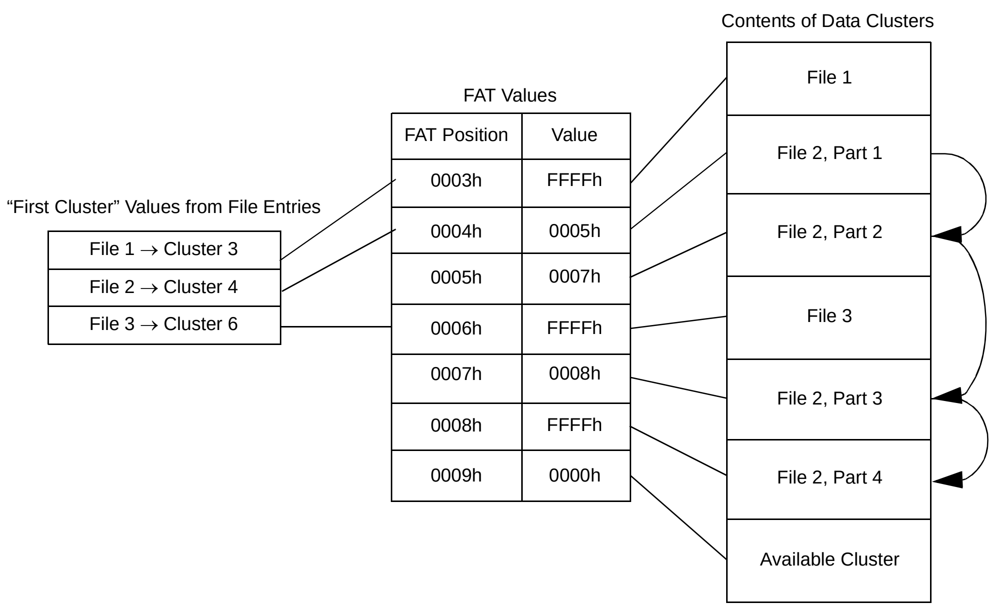
Each file has at least one cluster assigned to it. If that file size is smaller than the size of a cluster, the FAT entry for that cluster will contain the last cluster value indicating that there are no more clusters assigned to that file; else, it will contain the value of the next cluster of the file. By linking clusters in this way, the FAT can create a chain of clusters to contain larger files and can allocate non-sequential clusters to a file. Figure 2 illustrates an example of this.

It is important to note that the values that would point towards Clusters 0 and 1 are reserved to indicate special conditions. Because of this, the first cluster in the data region is labeled as Cluster 2. The FAT entries corresponding to Clusters 0 and 1 contain the media descriptor, followed by bytes containing the value, FFh.

TABLE 6: FAT VALUES

FAT16 Values	FAT32 Values	Description
0000h	0000h	Cluster is available for use.
0001h	0001h	Cluster is reserved.
0002-FFEFh	0000 0002-0FFF FFEFh	Points to next cluster in the file.
FFF0-FFF6h	0FFF FFF0-0FFF FFF6h	Cluster is reserved.
FFF7h	0FFF FFF7h	Cluster is bad.
FFF8h-FFFFh	0FFF FFF8h-0FFF FFFFh	Last cluster of a file.

FIGURE 2: FAT CLUSTER CHAIN



Note 1: Two-byte cluster values in this figure are for the FAT16 file system. FAT32 uses four-byte cluster values, as indicated in Table 6.

The "First Cluster" values in three file entries in the root directory indicate the start of three files. The FAT Values demonstrate the links between the files. File 1 and 3 are smaller than the size of a cluster; hence, only one cluster is assigned to them. The entries in the FAT that correspond to these files contain only the End-Of-File (EOF) value.

File 2 is larger than three clusters, but smaller than four; hence, four clusters are assigned to it. Since three consecutive clusters were not available when File 2 was created, nonconsecutive clusters were assigned to it; this is called "fragmentation". Each value in the FAT for File 2 point to the next cluster in the file. The last entry in the FAT for File 2 contains the End-Of-File value.

Directories

Except for the root directory, the directories in this file system are written in the same way that files are written. Each directory occupies one or more clusters in the data section of the partition, and has its own directory entry and chain of FAT entries. Bit four of the attribute field in the directory entry of a directory is set, indicating that the entry belongs to a directory. Directory names in this library follow the short filename format (8.3 format). Directories differ from files; they do not have an extension.

Each directory contains 32-byte directory entries. Two directory entries, the dot entry and the dot dot entry are present in every directory except the root directory. The dot entry is the first entry in any subdirectory. The name value in this entry is a single dot (2Eh) followed by ten space characters (20h). The pointer of this entry to the first cluster of its “file” will actually point to the cluster that contains the entry itself. The dot dot entry is similar, except the name contains two dots followed by nine spaces, and the pointer to the first cluster in the “file” will point to the directory that contains the entry for the directory that the dot dot entry is in (the previous directory).

When the directories are enabled in this library, all file modification will be done in the Current Working Directory (CWD). When the card is initialized by calling `FSInit`, the CWD is automatically set to the root directory. After this, the CWD can be changed with the `FSchdir` function.

Follow these conventions when specifying path names in the directory manipulation functions:

Directory names in a path string are delimited by the backslash character (\). When denoting a backslash character in a string, an additional backslash must be added as part of an escape sequence, as the backslash is used by C to begin escape sequences.

- If the first character of a path string is a backslash, the path will be assumed to be specified relative to the root directory.
- If a path string begins with a directory name, the path will be assumed to be specified relative to the current working directory.
- If a dot (.) or dot dot (..) is included in the path as a directory name, the code will operate using those directory entries.

  For example, if the user changes the CWD to “`.\TEST\..\TEST\..\.\.`”, the current working directory would not change from where it originally started, assuming that the directory, `TEST`, exists in the original directory.

> **Note:** When hard-coding the string in C, double backslashes are required. Refer to the API descriptions of `FSmkdir`, `FSchdir`, `FSrmdir` and `FSgetcwd`.

Table 7 provides more examples of path strings.

TABLE 7: EXAMPLE DIRECTORY PATH STRINGS

Path	Meaning
“\”	The root directory.
“.”	Current directory.
“..”	Previous directory.
“ONE”	Directory ONE in the current directory.
“.\ONE”	Directory ONE in the current directory.
“\ONE”	Directory ONE in the root directory.
“..\ONE”	Directory ONE in the previous directory.
“ONE\TWO”	Directory TWO in directory ONE in the current directory.
“\ONE\TWO”	Directory TWO in directory ONE in the root directory.
“ONE\..\TWO”	Directories ONE and TWO in the current directory (this path could be used to create non-existent directories in the same place using the `FATmkdir` function).

SOFTWARE LIBRARY

User Functions

To manage file and disk manipulation, call functions are provided in Table 8.

TABLE 8: FILE AND DISK MANIPULATION FUNCTIONS

Function Name	Description
`FSInit`	Initializes the card, loads the master boot record (partition information), loads the boot sector and updates the parameters passed into it with its information.
`FSfclose`	Updates the file information, writes the remaining entry in and frees the RAM from the heap that was used to hold the information about that file. This also updates the time-stamp information for the file.
`FSfeof`	Verifies if the end of the file has been reached.
`FSfopen`	Allocates space in the heap for file information. If the file being opened already exists, `FSfopen` can open it so that the data would be appended at the end of the file, erase it and create a new file with the same name to be written to, or simply open it for reading. If the file does not exist, `FSfopen` can create it. This function then returns a pointer to the structure in the heap that contains information for this file.
`FSfopenpgm`	Opens a file on the SD card and associates an `FSFILE` structure (`stream`) with it using arguments specified in ROM. This function is necessary only on the PIC18 architecture.
`FSfread`	Reads information from an open file to a buffer. The number of bytes written can be specified by its parameters. If `FSfread` is called consecutively on the same open file, the read will continue from the place it stopped after the previous read. This function returns the number of data objects read.
`FSfseek`	Changes the position in a file. When a user calls `FSfseek`, they specify the base address to set, which can either be at the beginning or end of the file, or at the current position in the file. The user also specifies an offset to add to the base (note that if the base address is at the end of the file, the offset will be subtracted). Hence, if `FSfseek` is called with the base set to the beginning of the file and a specified offset of '`0`', the position would be changed to the first byte of the file.
`FSftell`	Returns the current position in the file. The first position in the file is the first byte in the first sector of the first cluster, which has the value '`0`'. Hence, if a file was created and 2000 bytes were written to it, `FSftell` would return the number 1999 if it was called.
`FSfwrite`	Writes information from a buffer to an open file. The algorithm it uses reads a sector from the data region of the disk to SRAM, modifies the relevant bytes and then writes the sector back to the disk. Because each `FSfwrite` call reads the data first, the ability to open multiple files at a time is supported. This also means that writing data in larger blocks takes less time than writing the same data in smaller blocks as fewer sector reads and writes will be needed.
`FSremove`	Searches for a file based on a filename parameter passed into it. If the file is found, its directory entry is marked as deleted and its FAT entry is erased.
`FSremovepgm`	Deletes the file identified by a given filename. If the file is opened with `FSfopen`, it must be closed before calling `FSremovepgm`. The filename must be specified in ROM. This function is necessary only on the PIC18 architecture.
`FSrename`	Changes the name of a file or directory. If the pointer passed into this function is NULL, the name of the current working directory will be changed.
`FSrewind`	Resets the position of the file to the beginning of the file.
`FSmkdir` (directory manipulation)	Creates a new subdirectory in the current working directory.
`FSchdir` (directory manipulation)	Changes the current working directory to the one specified by the user.
`FSrmdir` (directory manipulation)	Deletes the specified directory. The user may also choose to specify whether subdirectories and files contained within the deleted directory are removed. If the user does not permit the function to delete subdirectories, it fails if the user attempts to delete a non-empty directory.

TABLE 8: FILE AND DISK MANIPULATION FUNCTIONS (CONTINUED)

Function Name	Description
`FSgetcwd` (directory manipulation)	Returns the name of the current working directory to the user.
`FindFirst`	Locates files in the current working directory that meet the name and attribute criteria. A `SearchRec` Structure Pointer will be passed into the function. Once a file is located, the file-name, file size, create time and date stamp, and attributes fields in the `SearchRec` structure will be updated with the correct file information.
`FindFirstpgm`	Operates in the same manner as the `FindFirst` function, except the name criteria for the file to be found will be passed into the function in ROM. This function is necessary only on the PIC18 architecture.
`FindNext`	Locates the next file in the current working directory that matches the criteria specified in the last call of `FindFirst` or `FindFirstpgm`. It will then update the `SearchRec` structure provided by the user with the file information.
`FSformat`	Erases the root directory and file allocation table of a card. The user may also call the function in a mode that causes it to create a new boot sector based on the information in the master boot record.
`FSfprintf`	Writes a formatted string to a file. It automatically replaces any format specifiers in the string with dynamic values from variables passed into the function. Integer promotion must be enabled in the build options menu when using this function with the PIC18 architecture.
`SetClockVars`	Used in user-defined Clock mode to manually set the current date and time. This date and time would be applied to files as they are created or modified.

Library Setup

This section provides a list of customizations that can be used with this library. Perform the following steps before compiling a project:

1. Add the appropriate physical layer file to the project. Interfaces for the SD card in SPI mode (`SD-SPI.c`, `SD-SPI.h`) and the CompactFlash card using the PMP module (`CF-PMP.c`, `CF-PMP.h`) or manual bit toggling (`CF-Bit transaction.c`, `CF-Bit transaction.h`) are provided. Set the appropriate physical layer header file by including one of the filenames in `FSconfig.h`.
2. Define the system clock frequency in `FSconfig.h`.
3. Users, who want to configure static memory for file objects should specify the maximum number of files that are going to be open at any one time in `FSconfig.h`.
4. Users, who want to configure SD SPI interface should specify the appropriate register names in `SD-SPI.h`.

   For example, if SPI module 1 is used on PIC24, change the definition of `SPI1CON` to `SPI1CON1`. If module 2 is used, change the definition to `SPI2CON1`.
5. PIC18 users should modify the linker file to include a 512-byte section of RAM that will act as a buffer for file reads/writes. This buffer is defined at the top of the physical interface files. Also create a section in the linker mapped to this RAM called `dataBuffer`. Repeat this process to create a buffer for FAT reads and writes. This buffer requires a section mapped to the RAM you allocate called `FATBuffer`.
6. Users, who want to configure dynamic memory to allocate file objects should set the corresponding preprocessor directive in the `FSconfig.h` file to "`#if 1`". If PIC18 is used, a section called, `_SRAM_ALLOC_HEAP`, must be created in the linker file that contains enough memory to contain all the opened file objects. Each file object is 46 bytes. Due to variation in the memory allocation algorithm, the allocated memory size will be larger. This is also true when using a PIC24. Verify that enough memory was allocated to the heap. Include the `salloc.c` and `salloc.h` files in the project when using PIC18. When using dynamic memory allocation with the PIC24, a heap in the MPLINK30 tab of the Build Options menu should be created.
7. Set the library path and include path (and linker path, if PIC18) in the General tab of the Build Options menu.

8. Set the required input and output pins in your physical layer header file (SD-SPI.h, CF-PMP.h, …).
9. Make sure that all pins used are configured as digital I/Os, including the PORTB pins set in the Configuration registers and any pins that could be analog channels for the A/D converter.
10. Select the appropriate device and language toolset. The compiled code will be appropriate to the processor type (PIC18, PIC24F, PIC24H, dsPIC30 or dsPIC33).
11. There are several definitions in FSconfig.h that can be used to disable option (functionality) to save code space if these functions are not required. To enable the functionality, uncomment the option definition in the code. The available options are shown in Table 9:

TABLE 9: LIBRARY OPTIONS

Option	Description
ALLOW_WRITES	Enables write functions to write data to the card.
ALLOW_DIRS	Enables directory functions such as, creating, changing, and so on. **Note**: Writes must be enabled to use directories.
ALLOW_FORMATS	Enables card formatting function. **Note**: Writes must be enabled to use directories.
ALLOW_FILESEARCH	Enables file and directory search functions, such as FindFirst and FindNext.
ALLOW_PGMFUNCTIONS	Enables the pgm functions, such as FSfopenpgm, FSremovepgm and so on for PIC18. These functions accept parameters passed through ROM (pgm functions) on PIC18. The pgm functions will not work with other architectures. However, arguments in ROM can be passed into standard functions (e.g., FSfopen instead of FSfopenpgm) directly in PIC24, dsPIC30 and dsPIC33 architectures.
ALLOW_FSFPRINTF	Enables FSfprintf function. **Note**: Writes must be enabled to use directories.
SUPPORT_FAT32	Enables FAT32 functionality.

12. Uncomment a define to select a Clock mode for determining file create/modify/access times. The Increment Time-Stamp mode will set the times to a static value and will not provide accurate timing values. This mode is useful when file times are unimportant, as it reduces complexity. The User-Defined Clock mode will allow the user to manually set the timing values using the SetClockVars function. The Use Real-Time Clock mode will set the timing values automatically based on the values in the Real-Time Clock and Calendar (RTCC) module. This mode will require the user to enable and configure the RTCC module, and it is not available in architectures that do not support RTCC.

FAT16/FAT32 Initialization and File Creation

The following C18 code example illustrates a basic sequence of function calls to open a file for reading. This example initializes the card with the `FSInit` function, and then calls `FSfopen` to create a new file. Then, the code calls `FSfopenpgm`, a function which performs the same function as `FSfopen`, but accepts ROM parameters. This call opens an existing file in the Read mode. The code reads one 10-byte object and five 1-byte objects from the existing file. The example also describes how the code writes these objects to the newly created files, and then closes both the files. Finally, the code deletes the old file. It is important to close a currently open file before deleting it.

EXAMPLE 1: INITIALIZATION AND FILE CREATION FOR PIC18

```
#include “FSIO.h”

#define bfrsize 5

    void main(void)
    {
        FSFILE *pOldFile, pNewFile;
        char myData[20];
        char bfr [6];
        int bytesRead, bytesWritten;
        char newFile[] = “newfile.txt”;
        char writeArg = “w”;

        // Must initialize the FAT16/FAT32 library. It also initializes SPI and other related
           pins.
        if( !FSInit() )
            // Failed to initialize FAT16 – do something…
            return 1; // Card not present or wrong format

        // Create a new file

        pNewFile = FSfopen (newFile, writeArg);

        // Open an existing file to read
        pOldFile = FSfopenpgm (“myfile.txt”, “r”);
        if ( pOldFile == NULL )
            // Either file is not present or card is not present
            return 1;

        // Read 10 bytes of data from the file.
        bytesRead = FSfread((void*)myData, 10, 1, pOldFile);
        // read bfrSize (5) items (of size 1 byte). returns items count
        bytesRead = FSfread( (void *)bfr, 1, bfrSize, pOldFile );


        // Write those fifteen bytes to the new file
        bytesWritten = FSfwrite ((void *) myData, 10, 1, pNewFile);
        bytesWritten = FSfwrite ((void *) bfr, 1, bfrSize, pNewFile);

        // After processing, close the file.
        FSfclose( pOldFile );
        FSfclose (pNewFile);

        //Delete the old file
        FSremovepgm (“myfile.txt”);

    }
```


Memory Usage

Table 10 provides the unoptimized memory usage for the file interface library using the SD-SPI physical layer. 512 bytes of data memory are used for the data buffer and an additional 512 bytes are used for the file allocation table buffer. Additional data memory would be required based on the number of files opened at a time. The default values provided are for two files opened in Static Allocation mode. The C18 data memory value includes a 200h byte stack. The first row of the table indicates the smallest amount of memory that the library will use (for Read-Only mode), and each subsequent row indicates the increase in memory caused by enabling other functionality.

TABLE 10: FILE I/O LIBRARY MEMORY USAGE(1)

Functions Included	Program Memory (C30)	Data Memory (C30)	Program Memory (C18)	Data Memory (C18)
All extra functions disabled (Read-Only mode)	11934 bytes	1454 bytes	11099 bytes	2121 bytes
File search enabled	+1854 bytes	+0 bytes	+2098 bytes	+0 bytes
Write enabled	+6810 bytes	+0 bytes	+7488 bytes	+0 bytes
Format enabled (write must be enabled)	+2499 bytes	+0 bytes	+2314 bytes	+0 bytes
Directories enabled (write must be enabled)	+8430 bytes	+78 bytes	+8380 bytes	+90 bytes
Directories and search are both enabled	+51 bytes	+0 bytes	+118 bytes	+0 bytes
`pgm` functions enabled	N/A	N/A	+288 bytes	+0 bytes
`FSfprintf` enabled	+4749 bytes	+0 bytes	+2758 bytes	+0 bytes
FAT32 support enabled	+423 bytes	+4 bytes	+407 bytes	+4 bytes

Note 1: This is a resource requirement for V1.02. Refer to the ReadMe file for version-specific resource requirements. This library was compiled using MPLAB® C18 v8.02, v.3.13 and C30 v.3.01 compilers.

Prerequisites

- During sector reads and writes, the card should not be removed.
- The size of the PIC18 stack might have to be increased. Otherwise, a stack overflow could occur when functions are called and the data is pushed to the stack. If the stack size is increased in this way, the memory model in the **Project > Build Options > C18** tab must be set to “Multi-Bank Model”. To change the size of the stack, the linker script must be modified. An example of this is given in **Appendix A: “The PIC18 Linker Script”**.

Description of Data Types and Structures

- `DISK` – The `DISK` structure contains information about the physical disk. Never directly use the information stored in this structure.
- `FILE` – The `FILE` structure contains information about a file on the disk. Never directly use the information stored in this structure.
- Types defined in `GenericTypedefs.h`:
  - `BYTE` – An unsigned char (8 bits)
  - `WORD` – A short int (16 bits)
  - `SWORD` – An unsigned short long (24 bits)
  - `DWORD` – An unsigned long (32 bits)
- `SearchRec` – The `SearchRec` structure contains the name, create time and date stamps, size and attributes of a file found using the `FindFirst`, `FindFirstpgm` or `FindNext` function.

Table 11 provides the contents of the `SearchRec` structure.

TABLE 11: CONTENTS OF THE `SearchRec` STRUCTURE

<table>
<tr><th>Element</th><th>Function</th></tr>
<tr><td>char filename</td><td>The name of the file (NULL terminated)</td></tr>
<tr><td>unsigned char attributes</td><td>The file attributes</td></tr>
<tr><td>unsigned long file size</td><td>The size of the file in bytes</td></tr>
<tr><td>unsigned long time-stamp</td><td>The create time and date of the file
<table>
<tr><th>Bits</th><th>Value</th></tr>
<tr><td>31:25</td><td>Year (0 = 1980, 1 = 1981, …)</td></tr>
<tr><td>24:21</td><td>Month (1 = Jan, 12 = Dec)</td></tr>
<tr><td>20:16</td><td>Day (1-31)</td></tr>
<tr><td>15:11</td><td>Hours (0-23)</td></tr>
<tr><td>10:5</td><td>Minutes (0-59)</td></tr>
<tr><td>4:0</td><td>Seconds/2 (0-29)</td></tr>
</table></td></tr>
<tr><td>unsigned int entry</td><td>The file entry in the current working directory (for internal use only)</td></tr>
<tr><td>char search name</td><td>The string that the user searched for (for internal use only)</td></tr>
<tr><td>unsigned char search attr</td><td>The file attributes that the user searched for (for internal use only)</td></tr>
<tr><td>unsigned int cwd clus</td><td>The cluster number of the directory that the search was performed in (for internal use only)</td></tr>
<tr><td>unsigned char initialized</td><td>Indicates that the <code>SearchRec</code> object has been initialized with search information by a call from <code>FindFirst</code> (for internal use only)</td></tr>
</table>

UNSUPPORTED FEATURES

Long filenames are not supported.

REFERENCES

- SD Card Association – http://www.sdcard.org
- CompactFlash® Association – http://www.compactflash.org
- The following documents are referenced by this application note.
  - SD Memory Card Specifications, Part 1 "*Physical Layer Specification*", Version 1.01, September 2000
  - SD Memory Card Specifications, Part 2 "*File System Specification*", Version 1.0, February 2000
- MultiMediaCard Association – http://www.mmca.org
- PCGuide: FAT File System Disk Volume Structures – http://www.pcguide.com/ref/hdd/file/fat.htm
- ISO/IEC 9293 – http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER = 21273
- FAT32 File System Specification – http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx
- From Wikipedia – http://en.wikipedia.org/wiki/Fat16

CONCLUSION

File creation and storage are undoubtedly useful for applications that need to store large or small amounts of data over a long period. By using this application note and the C18/C30 code provided with it, project development time can be minimized.

APPENDIX A: THE PIC18 LINKER SCRIPT

This sample linker script reserves three blocks of memory:

- Specified by section, `_SRAM_ALLOC_HEAP`
- Specified by section, `dataBuffer`
- Specified by section, `FATBuffer`

The heap section need not be reserved if dynamic memory is not being used to store file objects.

This script contains a 0x200 byte stack. If a stack spans multiple memory banks, like the Example A-1 script does, the “Multi-Bank” model should be selected from the Project Build Options menu.

EXAMPLE A-1: PIC18 LINKER SCRIPT

```
// $Id: 18f8722i.lkr,v 1.4 2005/12/19 16:40:18 nairnj Exp $
// File: 18f8722i.lkr
// Sample ICD2 linker script for the PIC18F8722 processor

LIBPATH .

FILES c018i.o
FILES clib.lib
FILES p18f8722.lib

CODEPAGE   NAME=vectors    START=0x0           END=0x29          PROTECTED
CODEPAGE   NAME=page       START=0x2A          END=0x1FD7F
CODEPAGE   NAME=debug      START=0x1FD80       END=0x1FFFF       PROTECTED
CODEPAGE   NAME=idlocs     START=0x200000      END=0x200007      PROTECTED
CODEPAGE   NAME=config     START=0x300000      END=0x30000D      PROTECTED
CODEPAGE   NAME=devid      START=0x3FFFFE      END=0x3FFFFF      PROTECTED
CODEPAGE   NAME=eedata     START=0xF00000      END=0xF003FF      PROTECTED

ACCESSBANK NAME=accessram  START=0x0           END=0x5F
DATABANK   NAME=gpr1       START=0x60          END=0xFF
DATABANK   NAME=gpr2       START=0x100         END=0x1FF
DATABANK   NAME=gpr3       START=0x200         END=0x2FF
DATABANK   NAME=gpr4       START=0x300         END=0x3FF
DATABANK   NAME=gpr5       START=0x400         END=0x4FF
DATABANK   NAME=gpr6       START=0x500         END=0x5FF
DATABANK   NAME=gpr7       START=0x600         END=0x6FF
// Allocate 0x200 bytes for the data buffer
DATABANK   NAME=buffer1    START=0x700         END=0x8FF         PROTECTED
// Allocate 0x200 bytes for the FAT buffer
DATABANK   NAME=buffer2    START=0x900         END=0xAFF         PROTECTED
// Allocate 0x200 bytes for the heap
DATABANK   NAME=gpr8       START=0xB00         END=0xBFF
DATABANK   NAME=gpr9       START=0xC00         END=0xDFF
DATABANK   NAME=gpr10      START=0xE00         END=0xEF3
DATABANK   NAME=dbgspr     START=0xEF4         END=0xEFF         PROTECTED
DATABANK   NAME=gpr11      START=0xF00         END=0xF5F
ACCESSBANK NAME=accesssfr  START=0xF60         END=0xFFF         PROTECTED


SECTION    NAME=CONFIG            ROM=config
// Create a heap section
SECTION    NAME=_SRAM_ALLOC_HEAP  RAM=gpr8
// Create the data buffer section
SECTION    NAME=dataBuffer        RAM=buffer1
// Create the FAT buffer section
SECTION    NAME=FATBuffer         RAM=buffer2

STACK SIZE=0x200 RAM=gpr9
```

APPENDIX B: API DETAILS

FSInit

This API initializes the hardware and mounts the card in the library. If the card is not detected, it returns FALSE. This must be called before calling any other API function. If the card is removed and inserted, the application must call FSInit to remount the card. To verify if the card is present, call the MediaIsPresent() low-level function.

Syntax

```
int FSInit(void)
```

Parameters

None

Return Values

TRUE if card is present and the format is FAT12, FAT16 or FAT32; FALSE otherwise.

Precondition

None

Side Effects

None

EXAMPLE B-1: FSInit CODE

```
// Initialize library and detect card
if ( FSInit() != TRUE )
   // Failed to initialize FAT16
```

`FSfclose`

This API closes an opened file.

Syntax

`int FSfclose( FSFILE *stream )`

Parameters

`stream` – A pointer to a `FILE` structure obtained from a previous call of `FSfopen`.

Return Values

Returns 0 on success.

Returns EOF (-1) on failure.

Precondition

`FSfopen` was called and the `stream` contains the pointer returned by `FSfopen`.

Side Effects

None

EXAMPLE B-2: `FSfclose` CODE

```
if( FSfclose( stream ) == EOF )
{
  // Failed to close the file
  ...
}
...
```

`FSfeof`

This API detects if End-Of-File (EOF) position is reached.

Syntax

`int FSfeof( FSFILE *stream )`

Parameters

`stream` – Pointer to opened file.

Return Values

Returns non-zero if the End-Of-File (EOF) indicator is reached.

Returns 0 otherwise.

Precondition

File is opened successfully.

Side Effects

None

EXAMPLE B-3: `FSfeof` CODE

```
if (FSfeof (pFile) == 0)
{
  // Error
  ...
}
```

FSfopen

This API opens a file on the card and associates a FILE structure (stream) with it.

Syntax

```
FSFILE * FSfopen ( const char * fileName, const char *mode )
```

Parameters

filename – A NULL terminated char string specifying the filename. This string must be stored in RAM. The filename must be less than 8 characters, followed by a radix (.), followed by an extension containing three or lesser characters. The filename cannot contain any directory or drive letter information.

mode – A NULL terminated string specifying the file operation. This string must also be specified in RAM for PIC18.

The valid strings are:

r	Read-Only	
w	Write	If a file with the same name exists, it will be overwritten. No reads are allowed.
a	Append	If the file exists, the current location will be set to the end of the file; otherwise, the file will be created. No reads are allowed.

Return Values

A pointer to an FSFILE structure to identify the file in subsequent operations; NULL if the specified file could not be opened.

Precondition

FSInit is called.

Side Effects

None

EXAMPLE B-4: FSfopen CODE

```
// Create argument strings in RAM and use them to call the function
   FSFILE * fPtr;
   char [11] name = “myFile.txt”;
   char [2] modeArg = “w”;
   fPtr = FSfopen( name, modeArg );
```

FSfopenpgm

This API opens a file on the SD card and associates a FSFILE structure (stream) with it using arguments specified in ROM.

Syntax

```
FSFILE * FSfopenpgm (const rom char * fileName, const rom char *mode)
```

Parameters

filename – A NULL terminated char string specifying the filename. This string must be stored in ROM. The filename must be less than 8 characters, followed by a radix (.), followed by an extension containing three or less characters. The filename cannot contain any directory or drive letter information.

mode – A NULL terminated string specifying the file operation. This string must also be specified in ROM.

The valid strings are:

r	Read-Only	
w	Write	If a file with the same name exists, it will be overwritten. No reads are allowed.
a	Append	The file must exist for this operation. No reads are allowed.

Return Values

A pointer to the FILE structure to identify the file in subsequent operations, NULL if the specified file could not be opened.

Precondition

FSInit is called.

Side Effects

None

EXAMPLE B-5: FSfopenpgm CODE

```
// Create a file called MYFILE.TXT
FSFILE * fPtr;
fPtr = FSfopen( “myfile.txt”, “w”);
```

FSfread

This API reads data from the previously opened file. FSfread reads n items of data, each of length size bytes from the given file stream. The data is copied to the buffer pointed by ptr. The total number of bytes transferred is n * size.

Syntax

```
size_t FSfread( void *ptr, size_t size, size_t n, FSFILE *stream )
```

Parameters

ptr	–	Pointer to buffer to hold the data read.
size	–	Length of item in bytes.
n	–	Number of items to read.
stream	–	stream pointer to file opened with read (r) mode.

Return Values

On success, FSfread returns the number of items (not bytes) actually read.

On End-Of-File or error it returns ‘0’.

Precondition

File is opened in Read mode.

Side Effects

None

EXAMPLE B-6: FSfread CODE

```
...
//Read 100 packets of size 10 bytes each
nItems = FSfread( bfr, 10, 100, pFile );

if( nItems == 0 )
{
  // No packet was read
  ...
}
else if( nItems < 100 )
{
  // did not read all 100 packets. Possible EOF
  ....
}
else
{
  //read all 100 packets
  ...
}
```

FSfseek

This API moves the File Pointer position associated with the `stream`. The new position is `offset` bytes from the file location given by whence.

Syntax

`int FSfseek( FSFILE *stream, long offset, int whence )`

Parameters

`whence` – File location defining the starting point for offset. Must be 0, 1 or 2 as follows:

SEEK_SET	0	File beginning
SEEK_CUR	1	Current File Pointer position
SEEK_END	2	End-Of-File

`offset` – Number of bytes away from the starting point defined by whence.

`stream` – Pointer to opened file.

Return Values

Return 0 if success.

Returns -1 on error.

Precondition

File is opened successfully.

Side Effects

None

EXAMPLE B-7: FSfseek CODE

```
// move 100 bytes forward from the current
position


if( FSfseek( pFile, 100, SEEK_CUR ) !=  0 )
{
 ...  // handle error condition
}
```

FSftell

This API returns the current position of the File Pointer.

Syntax

`long FSftell( FSFILE *stream )`

Parameters

`stream` – Pointer to opened file.

Return Values

Returns the current File Pointer position on success.

Returns -1 on error.

Precondition

File is opened successfully.

Side Effects

None

EXAMPLE B-8: FSftell CODE

```
// get current file position

    long pos = FSftell( pFile );
    if (pos == -1)
{
  ...  //handle error condition
}
```

FSfwrite

This API writes data to the previously opened file, FSfwrite, writes n items of data, each of length size bytes to the given file stream. The data is copied from the buffer pointed to by ptr. The total number of bytes transferred is n* size.

Syntax

```
size_t FSfwrite( const void *ptr, size_t size, size_t n, FSFILE *stream )
```

Parameters

ptr	–	Pointer to buffer holding data to write.
size	–	Length of item in bytes.
n	–	Number of items to write.
stream	–	stream pointer to file opened with write (w) or append (a) mode.

Return Values

On successful completion, FSfwrite returns the number of items (not bytes) actually written; on error it returns a short count or 0.

Precondition

File is opened in Write (w) or Append (a) mode.

Side Effects

None

EXAMPLE B-9: FSfwrite CODE

```
if( FSfwrite( ptr, 100, 20, pFile ) != 20 )
{
  // not all items were written
  ...  //handle error condition
}
```

FSremove

This API deletes the file identified by `filename`. If the file is opened with `FSfopen`, it must be closed before calling `FSremove`. The filename must be specified in the RAM.

Syntax

```
int FSremove (const char * filename)
```

Parameters

`filename` – A pointer to a NULL terminated string in RAM.

Return Values

Returns 0 on success.

Returns EOF (-1) on failure.

Precondition

`FSInit` is called successfully.

Side Effects

None

EXAMPLE B-10: FSremove CODE

```
// Create a string for the file name in RAM and then deletes the file with that name

    char name[] = “myfile.txt”;
  if( FSremove(name) == EOF )
  {
    // error handling
    ...
  }
  ...
```


FSremovepgm

This API deletes the file identified by `filename`. If the file has been opened with `FSfopen`, it must be closed before calling `FSremovepgm`. The filename must be specified in ROM.

Syntax

`int FSremove (const rom char * filename)`

Parameters

`filename` – A pointer to a NULL terminated string in ROM.

Return Values

Returns 0 on success.

Returns EOF (-1) on failure.

Precondition

`FSInit` is called successfully.

Side Effects

None

EXAMPLE B-11: FSremovepgm CODE

```
// Deletes MYFILE.TXT

if( FSremovepgm ("myfile.txt") == EOF )
{
  // error handling
  ...
}
...
```


FSrewind

This API resets the file position to the beginning of the file.

Syntax

```
void FSrewind (FSFILE *stream)
```

Parameters

`stream` – A pointer to `FILE` structure obtained from a previous call of `FSfopen`.

Return Values

None

Precondition

File should already be opened by a previous call of `FSfopen`.

Side Effects

None

`SetClockVars`

This API sets the timing variables used to set file create/modify/access times. This function is only used when the user-defined Clock mode is selected.

Syntax

```
int SetClockVars (unsigned int year, unsigned char month, unsigned char day,
unsigned char hour, unsigned char minute, unsigned char second);
```

Parameters

`year` – The year, from 1980 to 2107.

`month` – The month, from 1-12.

`day` – The day, from 1-31.

`hour` – The hour of the day, from 0 (midnight) to 23.

`minute` – The current minute, from 0 to 59.

`second` – The current second, from 0 to 59.

Return Values

Returns 0 on success.

Returns -1 if an invalid parameter is passed in.

Precondition

`USERDEFINEDCLOCK` is defined in `FSconfig.h`.

Side Effects

Modified global timing variables.

EXAMPLE B-12: `SetClockVars` CODE

```
// Set the date and time to
// 2:35:20 PM, January 12, 2007

   if (SetClockVars (2007, 1, 12, 14, 35, 20))
   {
     // Invalid values passed in
     ...
   }
```

`FSformat`

This API erases the root directory and file allocation table of a card. It can also create a new boot sector, based on the mode the user calls the function in. FAT32 formatting is not supported.

Syntax

```
int FSformat (char mode, long int serialNumber, char * volumeID);
```

Parameters

`Mode`	–	0	Just erase FAT and root.
		1	Create a new boot sector. This will fail if the MBR is not present.
`serialNumber`	–		The serial number to program into the new boot sector.
`volumeID`	–		The name of the card; must be 8 or fewer characters.

Return Values

Returns 0 on success.

Returns -1 otherwise.

Preconditions

None

Side Effects

None

EXAMPLE B-13: `FSformat` CODE

```
char volID[] = “MyCard”;
// Erase FAT and root, create new boot sector
// Set Card serial number to 0x12345678,
// Set Card name to “MyCard”

   if (FSformat (1, 0x12345678, volID))
   {
     // Format failed
     …
   }
```

FSmkdir

This API creates a directory based on the path string passed by the user. Every non-existent directory in the path string will be created. Directory names in the path string must be no more than 8 ASCII characters. Directory names are delimited by the backslash (\) character. A dot (.) as a directory name will access the current directory. Two dots (..) will access the previous directory. Beginning the path string with a backslash will create the directories specified in the root directory. Beginning the path string with a directory name will create the directories specified in the current working directory.

Syntax

```
int FSmkdir (char * path);
```

Parameters

`path` – The path of directories to create.

Return Values

Returns 0 on success.

Returns -1 otherwise.

Precondition

`FSInit` is called successfully.

Side Effects

None

EXAMPLE B-14: `FSmkdir` CODE

```
char path[] = “\\ONE\\TWO\\THREE\\FOUR”;
// The path starts with a ‘\’ so dir ONE will be created in the root directory if it does
   not exist
// Dir TWO will be created in dir ONE if it does not exist. THREE will be created in TWO
   FOUR will be created in THREE
if (FSmkdir (path))
{
  // Error
  ...
}
```

FSchdir

This API changes the current working directory based on the path string passed by the user. Directory names are delimited by the backslash (\) character. A dot (.) as a directory name will access the current directory. Two dots (..) will access the previous directory. Beginning the path string with a backslash will change to the directory specified starting from the root directory. Beginning the path string with a directory name will change to the directory specified starting from the current working directory.

Syntax

```
int FSchdir (char * path);
```

Parameters

path – The path of directory to change to.

Return Values

Returns 0 on success.

Returns -1 otherwise.

Precondition

`FSInit` is called successfully.

Side Effects

The current working directory will be changed.

EXAMPLE B-15: FSchdir CODE

```
char path[] = “\\ONE\\TWO\\THREE”;
char path2[] = “..\\..\\..”;

// Change to directory THREE

if (FSchdir (path))
{
  // Error
  ...
}

// Change back to the root
// The first .. will change from THREE to TWO
// The second .. will change from TWO to ONE
// The third .. will change from ONE to the root
// Calling this function with a path of “\\” would also change to the root
if (FSchdir (path2))
{
  // Error
  ...
}
```

FSrmdir

This API deletes a directory based on the path string passed by the user. Directory names in the path string must be no more than 8 ASCII characters. Directory names are delimited by the backslash (\) character. A dot (.) as a directory name will access the current directory. Two dots (..) will access the previous directory. Specify if the subdirectories and files in the directory should be deleted.

Syntax

```
int FSrmdir (char * path, unsigned char rmsubdirs);
```

Parameters

path	–	The path of the directory to delete.	
rmsubdirs	–	TRUE	All subdirectories and files will be deleted.
		FALSE	The directory will only be deleted if it is empty.

Return Values

Returns 0 on success.

Returns -1 otherwise.

Precondition

FSInit is called successfully.

Side Effects

None

EXAMPLE B-16: FSrmdir CODE

```
char path[] = “\\ONE\\TWO\\THREE\\FOUR”;

// Delete directory FOUR if it exists
if (FSrmdir (path, FALSE))
{
  // Error
  // Maybe there’s something in FOUR
  // Try to delete all contents

  if (FSrmdir (path, TRUE))
  {
    // Error
    // Maybe FOUR just does not exist
    ...
  }
  ...
}
```

FSgetcwd

This API returns the path of the current working directory copied into a character array passed by the user. If the user passes a NULL Array Pointer, a default array of 10 bytes will be used. If the current working directory name is too large for the array, the number of characters that fit in the array will be copied into it, starting from the beginning of the path.

Syntax

`char * FSgetcwd (char * path, int numchars);`

Parameters

`path`	–	The path to copy the current working directory name to.
`numchars`	–	The number of characters that can be copied into the path.

Return Values

Returns a pointer to the current working directory name string.

Precondition

`FSInit` is called successfully.

Side Effects

The default name string will be overwritten if the function is called with a NULL Path Pointer.

EXAMPLE B-17: FSgetcwd CODE

```
char dir[] = “\\ONE\\TWO\\THREE\\FOUR”;
char buffer[40];
char * pointer;
char * pointer2;

FSmkdir (dir);
FSchdir (dir);

// Our current working directory is now \ONE\TWO\THREE\FOUR
// Copy the first 40 characters of the path name into buffer

pointer = FSgetcwd (path, 40);

// Get a pointer to a string with the first 10 chars of the path name

pointer2 = FSgetcwd (NULL, NULL);
```


FindFirst

This API locates the first file in the current working directory that matches the naming and attribute criteria passed by the user and copies its parameters into a structure passed by the user.

Syntax

```
int FindFirst (const char * fileName, unsigned int attr, SearchRec * rec);
```

Parameters

`fileName` – The name the file must correspond to (refer to Table B-1 for filename formats).

`attr` – The attributes that the file may have (refer to Table B-2 for attribute values).

`rec` – Pointer to the structure that will contain file information if a file is found.

TABLE B-1: FILENAME FORMATS

Format	Function
`*.*`	Find any file or directory
`FILENAME.ext`	Find a file named `FILENAME.ext`
`FILENAME.*`	Find a file with name `FILENAME` and any extension
`*.ext`	File a file with any name and the extension, `ext`
`*`	Find any directory
`ADIRNAME`	Find a directory named `ADIRNAME`
`FI*.E*`	Find any file with name starting with `FI-` and extension starting with `E-`

TABLE B-2: ATTRIBUTE VALUES

Attribute	Value	Function
`ATTR_READ_ONLY`	01h	File may have read-only attribute
`ATTR_HIDDEN`	02h	File may have hidden attribute
`ATTR_SYSTEM`	04h	File may be a system file
`ATTR_VOLUME`	08h	File may be a volume label
`ATTR_DIRECTORY`	10h	File may be a directory
`ATTR_ARCHIVE`	20h	File may have archive attribute
`ATTR_MASK`	3Fh	File may have any attributes

Return Values

Returns 0 on success.

Returns -1 otherwise.

Precondition

`FSInit` is called successfully.

Side Effects

The search criteria in the `SearchRec` structure from the last call of `FindFirst` or `FindFirstpgm` will be lost.

EXAMPLE B-18: FindFirst CODE

```
SearchRec file;
unsigned char attributes = ATTR_HIDDEN | ATTR_SYSTEM | ATTR_READ_ONLY | ATTR_VOLUME |
ATTR_ARCHIVE;

char name[] = “FILE*.*”;

// Find any non-directory file that has a name starting
// with the letters FILE-

if (FindFirst (name, attributes, &file))
{
  // Error
  ...
}

// Delete the file we found if its empty

if( file.size == 0)
FSremove (file.filename);
```


FindFirstpgm

This API performs the same function as the `FindFirst` function, but accepts a filename string passed into the function in ROM. This function will be required only on the PIC18 architecture.

Syntax

```
int FindFirstpgm (const rom char * fileName, unsigned int attr, SearchRec * rec);
```

Parameters

`fileName`	–	The name the file must correspond to.
`attr`	–	The attributes that the file may have.
`rec`	–	Pointer to the structure that will contain file information if a file is found.

Return Values

Returns 0 on success.

Returns -1 otherwise.

Precondition

`FSInit` is called successfully.

Side Effects

The search criteria from the last call of `FindFirst` or `FindFirstpgm` will be lost.

EXAMPLE B-19: `FindFirstpgm` CODE

```
SearchRec file;
unsigned char attributes = ATTR_MASK;

// Find any file that has a name starting with the letters FILE-

if (FindFirstpgm (“FILE*.*”, attributes, &file))
{
  // Error
  ...
}

// Delete the file we found if its empty

if( file.size == 0)
FSremove (file.filename);
```

FindNext

This API locates the next file in the current working directory that matches the naming and attribute criteria specified by the last call of `FindFirst` or `FindFirstpgm` on the `SearchRec` object that is passed into the function.

Syntax

`int FindNext (SearchRec * rec);`

Parameters

`rec` – Pointer to the structure that will contain file information if a file is found.

Return Values

Returns 0 on success.

Returns -1 otherwise.

Precondition

`FindFirst` or `FindFirstpgm` is called successfully.

Side Effects

None

EXAMPLE B-20: FindNext CODE

```
SearchRec file;
unsigned char attributes = ATTR_MASK;
char name[] = “*.*”;

// Find any file or directory

if (FindFirst (name, attributes, &file))
{
  // Error
  ...
}

// Find the next file or directory

if( FindNext (&file))
{
  // Error
  ...
}
```

FSrename

This API changes the name of a file or directory. If the pointer passed into this function is NULL, the name of the current working directory will be changed.

Syntax

`int FSrename (const char *fileName, FSFILE * fo)`

Parameters

`fileName` – The new name of the file.

`fo` – The file to rename.

Return Values

Returns 0 on success.

Returns -1 otherwise.

Precondition

None

Side Effects

None

EXAMPLE B-21: FSrename CODE

```
FSFILE *fs;
//  Here,  Assign  “fs”pointer  to  a  file
structure whose name to be renamed


if (!Fsrename(“NEWNAME.TXT”, fs))
      // Success
else
     // Handle error
```

FSfprintf

The `FSfprintf` function will write a formatted string to a file.

Syntax

`int FSfprintf (FSFILE *fptr, const char * fmt, ...)`

Parameters

`fptr`	–	Pointer to a file to write to.
`fmt`	–	The string to write (specified in ROM).
`...`	–	Format specifiers.

Return Values

Returns the count of characters written on success.

Returns -1 otherwise.

Precondition

The file to be written to has been opened successfully.

Side Effects

None

Remarks

The `FSfprintf` function formats output, passing the characters to the specified stream. The format string is processed one character at a time and the characters are output as they appear in the format string, except for format specifiers. A format specifier is indicated in the format string by a percent sign, %; following that, a well-formed format specifier has the following components. Except for the conversion specifier, all format specifiers are optional.

1. Flag Characters:
   - '-' – The result of the format conversion will be left justified.
   - '+' – By default, a sign is only prefixed to a signed conversion if the result is negative. If this flag is included, a '+' sign will be prefixed if the result of a signed conversion is positive.
   - '0' – This flag will prefix leading zeros to the result of a conversion until the result fills the field width. If the '-' flag is specified, the '0' flag will be ignored. If a precision is specified, the '0' flag will be ignored.
   - ' ' – The space flag will prefix a space to the result of a signed conversion if the result is positive. If the space flag and the '+' flag are both specified, the space flag will be ignored.
   - '#' – This flag indicates the "alternate form" of a conversion. For the '0' conversion, the result will be increased in precision, such that the first digit of the result will be '0'. For the 'x' conversion, a '0x' will be prefixed to the result. For the 'X' conversion, a '0X' will be prefixed to the result. For the 'b' conversion, a '0b' will be prefixed to the result. For the 'B' conversion, a '0B' will be prefixed to the result.
2. Field Width:

   The field width specifier follows the flag specifiers. It determines the minimum number of characters that result from a conversion. If the result is shorter than the field width, the result is padded with leading spaces until it has the same size as the field width. If the '0' flag specifier is used, the result will be padded with leading zeros. If the '-' flag specifier is used, the result will be left justified and will be followed by trailing spaces.

   The field width may be specified as an asterisk character (*). In this case, a 16-bit argument will be read from the list of format specifiers to specify the field width. If the value is negative, it is as if the '-' flag is specified, followed by a positive field width.

3. Field Precision:

   The field precision specifies the minimum number of digits present in the converted value for integer conversions, or the maximum number of characters in the converted value for a string conversion. It is indicated by a period (.), followed by an integer value or by an asterisk (*). If the field precision is not specified, the default precision of 1 will be used.

   If the field precision is specified by an asterisk character, a 16-bit argument will be read from the list of format specifiers to specify the field precision.

4. Size Specification:

   The size specification applies to any integer conversion specifier or pointer conversion specifier. The integer conversion specifiers are as follows: the size specifIer will determine what type of argument is read from the format specifier list. For the n conversion, the size specifier for each pointer type corresponds to the specifier for that data type. So, to convert something to a Long Long Pointer, you would use the specifier for a long long data type with the n conversion.

TABLE B-3: SIZE SPECIFIERS

Argument Type	C18	C30
signed char, unsigned char	hh	hh
short int, unsigned short int	h	h
short long, unsigned short long	H	—
intmax_t, uintmax_t	j (32-bit)	j (64-bit)
long, unsigned long	1	1
long long, unsigned long long	—	q
size_t	z	z
sizerom_t	Z	—
ptrdiff_t	t	t
ptrdiffrom_t	T	—

5. Conversion Specifiers:
   - c – The int argument will be converted to an unsigned char value and the character represented by that value will be written.
   - d, i – The int argument is formatted as a signed decimal.
   - o – The unsigned int argument will be converted to an unsigned octal.
   - u – The unsigned int argument will be converted to an unsigned decimal.
   - b, B – The unsigned int argument will be converted to an unsigned binary.
   - x – The unsigned int argument will be converted to an unsigned hexadecimal. The characters, a, b, c, d, e and f, will be used to represent the decimal numbers, 10-15.
   - X – The unsigned int argument will be converted to an unsigned hexadecimal. The characters, A, B, C, D, E and F, will be used to represent the decimal numbers, 10-15.
   - s – Characters from the data memory array of char argument are written until either a terminating '\0' character is seen ('\0' is not written) or the number of chars written is equal to the precision.
   - S – Characters from the program memory array of char arguments are written until either a terminating '\0' character is seen ('\0' is not written) or the number of chars written is equal to the precision. In C18, when outputting a far rom char *, make sure to use the H size specifier (%HS).
   - p – The pointer to void the (data or program memory) argument is converted to an equivalent size unsigned integer type and that value is processed as if the x conversion operator had been specified. In C18, if the H size specifier is present, the pointer is a 24-bit pointer; otherwise, it is a 16-bit pointer.
   - P – The pointer to void the (data or program memory) argument is converted to an equivalent size unsigned integer type and that value is processed as if the X conversion operator had been specified. In C18, if the H size specifier is present, the pointer is a 24-bit pointer; otherwise, it is a 16-bit pointer.
   - n – The number of characters written so far shall be stored in the location referenced by the argument, which is a pointer to an integer type in data memory. The size of the integer type is determined by the size specifier present for the conversion, or a 16-bit integer if no specifier is present.
   - % – A literal percent sign will be written.

   If the conversion specifier is invalid, the behavior is undefined.

EXAMPLE B-22: `FSfprintf` CODE

```
unsigned long long hex = 0x123456789ABCDEF0;
FSfprintf (fileptr, “This is a hex number:%#20X%c%c”, 0x12ef, 0x0D, 0x0A);
FSfprintf (fileptr, “This is a bin number:%#20b%c%c”, 0x12ef, 0x0D, 0x0A);
FSfprintf (fileptr, “%#26.22qx”, hex);

// Output:
// This is a hex number:  0x12EF
// This is a bin number:  0b0001001011101111
// 0x0000123456789ABCDEF0
```


APPENDIX C: LIBRARY DIRECTORY

TABLE C-1: LIBRARY DIRECTORY ORGANIZATION(1)

Directory	Content
MDD File System-PIC18-CF-DynMem-UserDefClock	Sample project for PIC18 using the CompactFlash® interface, user-defined clock values and dynamic file object allocation.
MDD File System-PIC24-SD-StatMem-RTCC	Sample project for PIC24F using the SD card interface, the Real-Time Clock and Calendar (RTCC) module and static file object allocation.
Microchip\MDD File System	C files for MDD file system.
Microchip\PIC18 salloc	C file for PIC18 dynamic memory allocation.
Microchip\Include	Contains miscellaneous include files, including a standard data type definition file.
Microchip\Include\MDD File System	Include files for MDD file system.
Microchip\Include\PIC18 salloc	Include file for C18 dynamic memory allocation.

Note 1: These directories are relative to the installation directory.

NOTES:

MICROCHIP

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

01/02/08