PARALLAX SEMICONDUCTOR
www.parallaxsemiconductor.com
sales@parallaxsemiconductor.com
support@parallaxsemiconductor.com
phone: 916-632-4664 • fax:916-624-8003

Application Note AN006

FAT16/FAT32 Full File System Driver

Secure Digital (SD) cards have become the choice medium for mass storage in embedded systems. The goal of this application note is to provide the reader with knowledge on how to use SD cards, important features of the FAT file system, and how to use the FAT16/32 Full File System Driver software library for the Propeller chip (P8X32A).

Introduction

The Full File System Driver gives the Propeller chip the ability to read or write to files on an SD card and perform directory operations. Using the driver successfully requires a good understanding of SD cards and the FAT16/32 file system.

The SD Card

Three SD card formats are available today: standard, high capacity, and extended capacity.

Standard SD cards come in three physical packages: full sized, mini, and micro. Similarly, high capacity SD cards come in the same three formats. Extended capacity SD cards come only in full-sized and micro formats.

Table 1: SD Card Formats http://www.sdcard.org

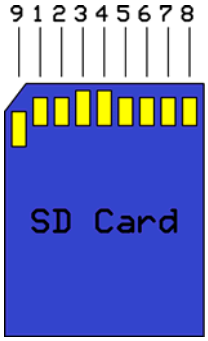
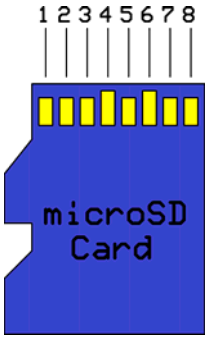
Format	Capacity	Packages Available
SD – Standard	Up to 2 GB	SD, Mini SD, Micro SD
SDHC – High Capacity	4 GB to 32 GB	SD, Mini SD, Micro SD
SDXC – Extended Capacity	Over 32 GB up to 2 TB	SD, Micro SD

The SD Association[1] makes available complete information on SD card specifications for both consumers and developers.

SD Card Pin Layouts

Table 2 shows the pin layouts for full sized SD cards and micro SD cards. Full sized SD card slots supply the card detect (CD) and write protect (WP) pins. However, Micro SD card slots only have the card detect pin.

Table 2: Pin Layouts for SD and Micro SD Packages

SD Card		Micro SD Card	
	Pin 1 – Chip Select (CS)		Pin 1 – Unused
	Pin 2 – Data In (DI)		Pin 2 – Chip Select (CS)
	Pin 3 – Ground		Pin 3 – Data In (DI)
	Pin 4 – 3.3V		Pin 4 – 3.3V
	Pin 5 – Clock (CLK)		Pin 5 – Clock (CLK)
	Pin 6 – Ground		Pin 6 – Ground
	Pin 7 – Data Out (DO)		Pin 7 – Data Out (DO)
	Pin 8 – Unused		Pin 8 – Unused
	Pin 9 – Unused		

Simple Electrical Interface Schematics

Use the electrical schematics in Figure 1 and Figure 2 for interfacing with SD cards when not using card detect and/or write protect pins. Pull-up resistors on the DI and CLK lines are optional, but prevent bus floating. Pull-up resistors on the DO and unused lines are also optional; however, leaving them out will increase power consumption. The pull-up resistor on the CS line is also optional, but recommended for MMC compatibility. Additionally, the lines shown below should be as short as possible. Additionally, connect a power supply decoupling capacitor (above 100 µF) and/or filter (not shown) near the SD or micro SD card slot to prevent brownouts from occurring.

Figure 1: SD Card Slot Simple Interface

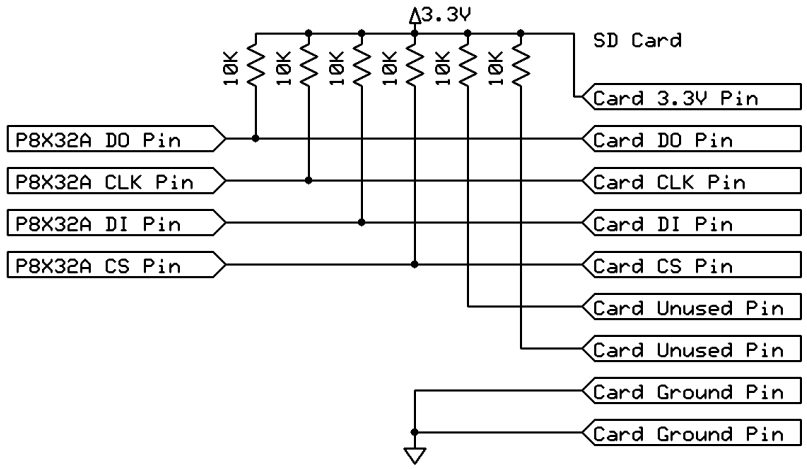
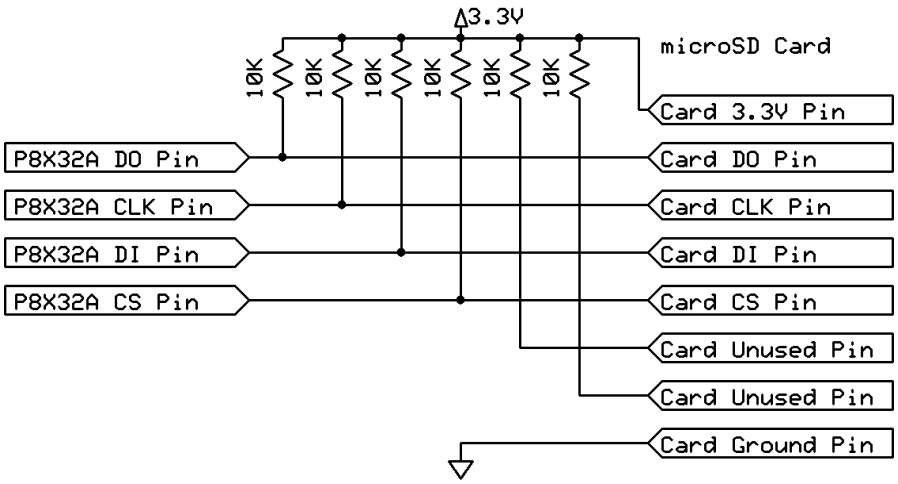


Figure 2: Micro SD Card Slot Simple Interface



Advanced Electrical Interface Schematics

Use the electrical schematics in Figure 3 and Figure 4 for interfacing with SD cards using card detect and/or write protect pins. Pull-up resistors on the DO and unused lines are optional; however, leaving them out will increase power consumption. The pull-up resistor on the CS line is optional too, but recommended for MMC compatibility. The pull-up resistor on the CLK line for the micro SD card is also optional, but prevents bus floating. Additionally, the lines shown below should be as short as possible.

Figure 3: SD Card Slot Advanced Interface

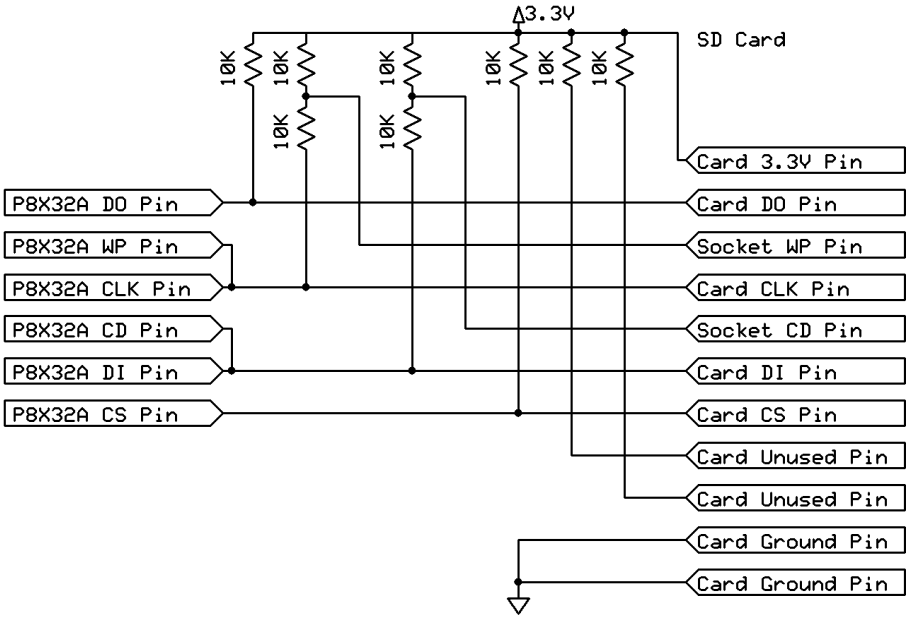
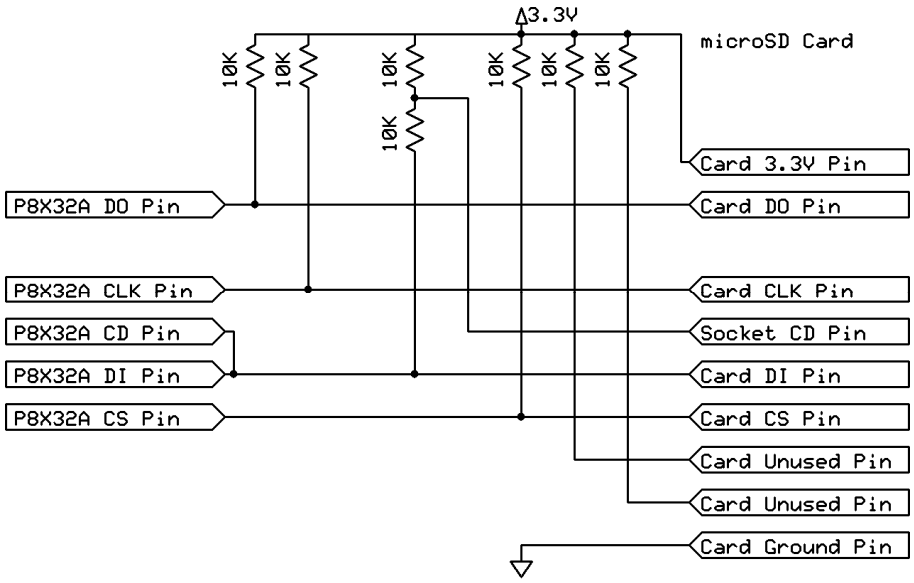


Figure 4: Micro SD Card Slot Advanced Interface

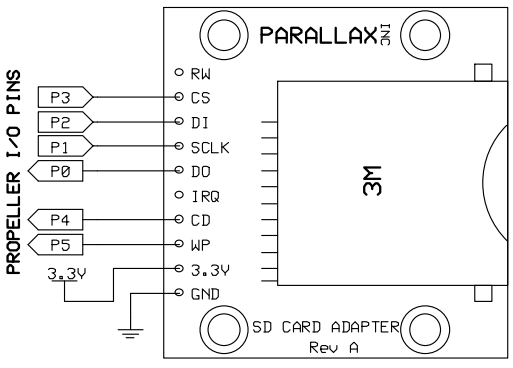
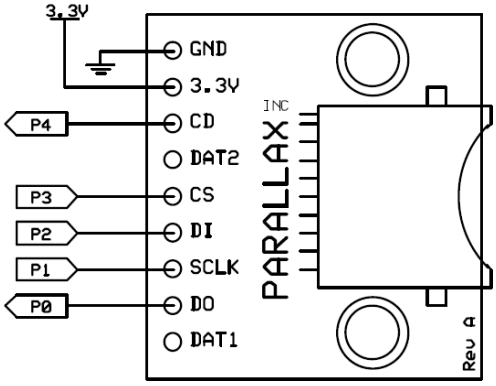


Additionally, connect a power supply decoupling capacitor (above 100 µF) and/or filter (not shown above) near the SD or micro SD card slot to prevent brownouts from occurring.

Parallax Electrical Interface Schematics

Use the electrical schematics in Table 3 for interfacing with SD cards using Parallax Inc.'s full-sized SD Card Adapter Kit[2] or micro-SD Card Adapter[3]. Again, the lines shown below should be as short as possible.

Table 3: Parallax Interface

SD Card	Micro SD Card
	

P0 connects to the driver's DO pin, P1 connects to the driver's CLK pin. P2 connects to the driver's DI pin, P3 connects to the driver's CS pin, P4 connects to the driver's CD pin, and P5 (if present) connects to the driver's WP pin. Other pins on the Propeller chip than the ones listed above will work with the Full File System Driver.

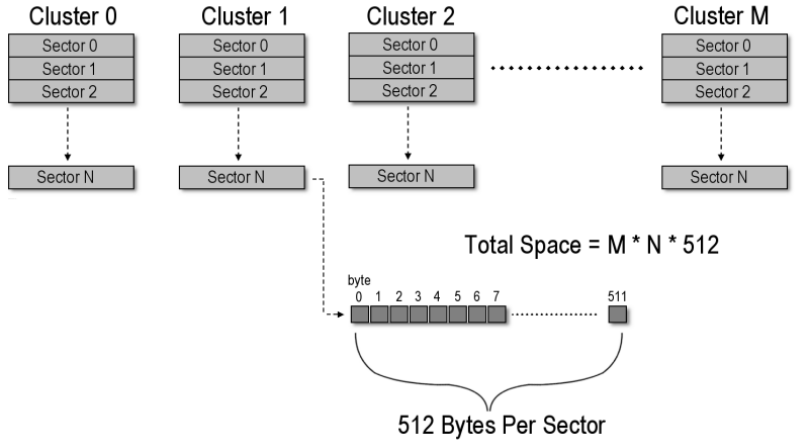
Disk Partitioning and Formatting

SD cards with a master boot record (MBR) support up to four FAT16 or FAT32 partitions while SD cards without a MBR support only one FAT16 or FAT32 partition.

Sectors and Clusters

Bytes on SD cards are accessible in 0.5 KB chunks called sectors. Sectors are accessible in 0.5 KB, 1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, and 64 KB chunks called clusters.

Figure 5: Clusters and Sectors[4]



The default allocation unit for the FAT16/32 file system is the cluster, and files or folders span one or more clusters. This means a one-byte file takes up an entire cluster. Thus, smaller cluster sizes save more space and larger cluster sizes waste more space.

Clusters are accessed one sector at a time, linearly, by the Full File System Driver. The driver can access a list of linear sectors more quickly than a list of non-linear sectors. However, files and folders are not guaranteed to be made out of a linear list of clusters (a file with a linear list of clusters has a linear list of sectors). Thus, a file system with a small cluster size will be slower than a file system with a large cluster size.

FAT Type Differences

There are three main FAT file system formats: FAT12, FAT16, and FAT32. The Full File System driver does not support FAT12—FAT12 exists solely for floppy disks. Use FAT16 for small SD cards fewer than 4 GB in size. Inversely, use FAT32 for large SD cards more than 4 GB in size. This is because FAT file systems under approximately 64 K clusters are FAT16 while FAT file systems over approximately 64 K clusters are FAT32. Keep in mind that a FAT32 partition on a small SD card will have a large number of small-sized clusters while a FAT16 partition on a large SD card will have a small number of large-sized clusters. Neither of the above cases is optimal for performance or space savings.

Warning: The default factory settings for the file system type and cluster size are the optimal settings for any SD card. Changing them may reduce performance.

The FAT16/32 Full File System Driver

Start the secure digital card block driver first before using the Full File System Driver. The following piece of code illustrates how to set up the secure digital card block driver.

```
CON

  _clkmode = xtal1 + pll16x ' The clkfreq is 80MHz.
  _xinfreq = 5_000_000 ' Demo board compatible.

  _dopin = 0
  _clkpin = 1
  _dipin = 2
  _cspin = 3
  _cdpin = 4 ' -1 if unused.
  _wppin = 5 ' -1 if unused.

  _rtcclkpin = -1 ' -1 if unused.
  _rtcdatpin = -1 ' -1 if unused.
  _rtcbuslck = -1 ' -1 if unused.

OBJ fat: "SD-MMC_FATEngine.spin"

PUB main

  fat.fatEngineStart( _dopin, _clkpin, _dipin, _cspin, _wppin, _cdpin, {
                    } _rtcdatpin, _rtcclkpin, _rtcbuslck)
```

The **fatEngineStart** method initializes the SD card block driver that handles communication with the SD card. The method returns true if it succeeds and false if it fails. The above code does not check if **fatEngineStart** failed because the method will always succeed in the above situation. However, when dynamically starting and stopping the SD card block driver, check what **fatEngineStart** returns. There is also a **fatEngineStop** method, which shuts down the SD card block driver.

Once the block driver is running, call the **partitionCardNotDetected** method to check if no SD card is detected. In addition, to check if the SD card is write protected, call **partitionWriteProtected**. Card detection is disabled if the CD pin is set to -1. Write protection is also disabled if the WP pin is set to -1. When card detection is disabled, the driver assumes an SD card is always detected. When write protection is disabled, the driver assumes the detected SD card is not write protected.

Note: MMC cards do not feature the mechanical write protection slider found on SD cards.

Real Time Clock Support

The Full File System Driver also supports real time clock (RTC) modules. The above code shows how the driver interfaces with an I^2C RTC module. The driver supports several different RTC modules. If a RTC module is to be used with the Full File System Driver, select the desired version of the Full File System driver that supports your RTC module. The driver can also be modified to support any other real time clock module.

Mounting and Un-mounting

After initializing the SD card block driver, a FAT16 or a FAT32 partition on the SD card can be mounted for use. A FAT16 or FAT32 partition must be mounted before performing file system operations. The code below shows how to do so.

```
CON _errorpin = 23 ' Error LED.

PUB main | errorNumber, errorString

  fat.fatEngineStart( _dopin, _clkpin, _dipin, _cspin, _wppin, _cdpin, {
                    } _rtcdatpin, _rtcclkpin, _rtcbuslck)

  errorString := \code ' Returns the address of the error string or null.
  errorNumber := fat.partitionError ' Returns the error number or zero.

  if(errorNumber) ' Light a LED if an error occurs.
    outa := constant(|<_errorpin)
    dira := constant(|<_errorpin)

  repeat ' Wait until reset or power down.

PRI code ' Put the file system calls in a separate method to trap aborts.

  fat.mountPartition(0) ' Mount the default partition 0.  Can be 0 - 3.
```

Most of the methods inside of the Full File System Driver abort when an error occurs. Because of this, set up an error handler to trap abort errors. Whenever a file system function aborts, it returns a pointer to an error string describing the error. Additionally, it is possible to get the error number associated with the error string by calling the **partitionError** method. Listed in the code below are the error numbers.

Note: **partitionError** returns the error number only once and then clears the error. If another error occurs before calling **partitionError**, the old error will be overwritten.

```
CON ' Use obj#errorName to get the error number constant. E.g. fat#disk_io_error.

  #1, Disk_IO_Error, Clock_IO_Error, {
    } File_System_Corrupted, File_System_Unsupported, {
    } Card_Not_Detected, Card_Write_Protected, {
    } Disk_May_Be_Full, Directory_Full, {
    } Expected_An_Entry, Expected_A_Directory, {
    } Entry_Not_Accessible, Entry_Not_Modifiable, {
    } Entry_Not_Found, Entry_Already_Exist, {
    } Directory_Link_Missing, Directory_Not_Empty, {
    } Not_A_Directory, Not_A_File
```

Most of the errors above are recoverable. However, **Disk_IO_Error**, **Clock_IO_Error**, **File_System_Corrupted**, **File_System_Unsupported**, and **Card_Not_Detected** are unrecoverable errors. Unrecoverable errors un-mount the file system; a recoverable error does not. If an unrecoverable error occurs, call the **mountPartition** method again to remount the file system. Error handling can become arbitrarily complicated; the above error handler is only a simple example.

Call the **partitionMounted** method (not shown in code above) to get the state of the Full File System Driver at any time. When the partition is not mounted, all of the methods in the driver will do nothing when called and return null—they will not abort. After you mount the partition, however, methods may abort. In addition, call the methods shown in Table 4 to obtain more information about the file system once mounted.

Table 4: File System Information Methods

Method Name	Action
partitionDiskSignature	Returns the disk signature number
partitionVolumeIdentification	Returns the volume identification number
partitionVolumeLabel	Returns a pointer to the volume label string
partitionFileSystemType	Returns a pointer to the file system type string
partitionBytesPerSector	Returns the count of bytes per sector for the current partition
partitionSectorsPerCluster	Returns the count of sectors per cluster for the current partition
partitionDataSectors	Returns the count of sectors for the current partition
partitionCountOfClusters	Returns the count of clusters for the current partition
partitionUsedSectorCount	Returns the current used sector count on this partition
partitionFreeSectorCount	Returns the current free sector count on this partition

Note: **partitionUsedSectorCount** and **partitionFreeSectorCount** scan the entire file allocation table (FAT) when called and may take a long time to return.

To un-mount the partition, call the **unmountPartition** method. Always un-mount any mounted partition. Once called on a writable SD card partition, **mountPartition** flags the partition it mounts as dirty; when **unmountPartition** is called, it flags the writable SD card partition as clean. The **mountPartition** method returns true if it mounts a dirty partition, and false if it mounts a clean partition. A dirty partition may be corrupted. Check dirty partitions for consistency before using them.

Note: If you try to use a dirty partition with Microsoft Windows it will run Check Disk on the dirty partition to check for errors and to try to fix them (results may vary).

Formatting

To format a partition on the SD card, call the **formatPartition** method. This method deletes all files and folders on a partition. Do not call the **formatPartition** method if you do not want to delete everything on a partition.

Note: The **formatPartition** method only zeros the partition FAT and root directory. It does not partition or format an unformatted SD card nor does it zero every sector. Some disk recovery utilities may be able to recover files and folders on a formatted partition.

Files and Folders

Files and folders are called directory entries in a FAT16/32 file system. Directory entries have an identifying "8.3" file name, a set of attributes, access time stamps, and a number of bytes of information stored on the partition. The Full File System Driver accesses directory entries on the file system partition through file system paths and target names.

An 8.3 file name (short file name) is a file or folder name with 8 characters for the name plus a 3-character file extension. Short file names can use the following characters:

- Uppercase A – Z. Lowercase a – z convert to uppercase A – Z
- Numbers 0 – 9
- Space
- Symbols: ! # $ % & ‘ ( ) - @ ^ _ ` { } ~
- Values 128 – 255

This excludes the following characters, however:

- Symbols: " * + , . / : ; < = > ? [\ ] |
- Control Characters 1 – 31
- Value 127

Table 5 shows a few examples of short file names. The Full File System Driver enforces the above short file name rules and translates offending short file names passed to it.

Table 5: Short File Name Examples

Passed Short File Name	Translated Short File Name
HelloWorld.html	HELLOWOR.HTM
Hello.World.html	HELLO.WOR
Hello+World.html	HELLO_WOR.HTM
Hello World . html	HELLO WO.HTM
.txt	_.txt
	_
..	..
.	.

Note: The bottom two short file names are special short file names handled by the driver.

The Full File System Driver automatically replaces lower case characters with upper case characters, replaces excluded characters with the underscore character, skips leading space in front of the file name and file name extension, and translates empty short file names to the underscore character.

Translation is handled by reading the first 8 leading valid characters after leading spaces for the file name and the first 3 leading valid characters after leading space characters after a period is encountered for the file name extension.

By default, Microsoft Windows names files and folders using “long file names” (LFNs) which the Full File System Driver does not support. LFNs can have a file name and extension up to 255 characters long, and support upper case and lower case characters. Additionally, LFNs support UTF-16 characters while short file names only support ASCII characters. Excluded characters in LFNs are the \ / : * ? “ < > | characters.

Whenever Microsoft Windows creates a file or folder using an LFN it also creates a directory entry with a short file name. However, the short file name created for use with the LFN is only remotely similar to the LFN. E.g. “Parallax.txt” translates to “PARALL~1.TXT” while “Parallax Propeller.txt” translates to “PARALL~2.TXT” for another file or folder in the same directory. Nevertheless, when Microsoft Windows creates an LFN following the short file name rules, the short file name matches the LFN exactly. E.g., “PROPEL.TXT” translates to “PROPEL.TXT”. Avoid LFN to short file name confusion by naming files and folders following the short file name rules.

In addition to having a short file name, files and folders on the file system partition can have the following attributes:

- **Read Only** – The file or folder cannot be deleted or moved. If the directory entry is a file, it also cannot be opened for writing.
- **Hidden** – The Full File System Driver does nothing with this attribute. However, Microsoft Windows will not list hidden files or folders by default.
- **System** – The Full File System Driver also does nothing with this attribute. However, Microsoft Windows will not list system files or folders by default and will not allow modification to system files or folders without administrator rights.
- **Directory** – Directory entries with this attribute are folders, directory entries without this attribute are files.
- **Archive** – Files with this attribute have been modified (written or moved). Folders cannot have this attribute. The archive attribute is set by the Full File System Driver and cleared by backup utilities performing file system maintenance.

New files and folders have their creation time stored in their directory entry. Files opened for reading or writing also have their last access time stored in their directory entry. Additionally, files opened for writing have their modification time stored in their directory entry. Folders, however, do not have their last access time or last modification time recorded and stored in their directory entry—only their creation time.

Files and folders are stored the same way in the FAT16/32 file system. Folders are simply containers for more files and folders. Both files and folders have cluster chains and use up space on the partition. However, only the files have their size recorded in their directory entry—folders do not have their size recorded; it is always zero.

Folders have two special directory entries inside of them called the “.” (dot) and “..” (dotdot) entries. The dot entry points to the current folder the file system is in, and the dotdot entry points to the previous folder the file system was in (up one level). The dot and dotdot entries support navigation through the file system and pathing. Trying to modify the dot and dotdot entries will cause the Full File System Driver to error.

File System Paths and Target Names

The Full File System Driver accesses files and folders on the FAT16/32 file system partition using file system paths and target names. A file system path is simply a string of directory (folder) short names separated by the "/" or "\" characters. A target name is the short name of the file or folder to perform a file system operation on. There are two types of file system path strings: relative path strings and absolute path strings. A relative path string starts from the current directory the file system is in. An absolute path string starts from the root directory of the file system.

A relative path string: "folderName/folderName/…/folderName/targetName"
It may also just be "targetName"
An absolute path string: "/folderName/folderName/…/folderName/targetName"
It may also just be "/targetName"

Note: "/" and "\" characters are interchangeable.

After the Full File System Driver mounts a partition, the current directory is the root directory. The root directory is the default directory on a FAT16/32 file system partition. To change directories after mounting a partition, call the **changeDirectory** method to change the current directory. The **changeDirectory** method takes a file system path string and changes the current directory to be the target folder's directory. When passed just the "/" or "\" character (in a string) the **changeDirectory** method will change the current directory back to being the root directory.

File System Manipulation Methods

The Full File System Driver supports the following file system manipulation methods:

To create new files, call the **newFile** method. The **newFile** method takes the file system path string of a new file and creates it. New files take up zero clusters on the partition until written to.

To create new folders, call the **newDirectory** method. The **newDirectory** method takes the file system path string of a new folder and creates it. New folders take up one cluster initially on the partition.

To delete files or folders, call the **deleteEntry** method. The **deleteEntry** method takes the file system path string of a file or folder to delete and deletes it. Some disk recovery utilities may be able to recover files and folders deleted by the **deleteEntry** method.

To move files or folders, call the **moveEntry** method. The **moveEntry** method takes the file system path string of a file or folder to move and moves it to another file system path given by another file system path string. Use the **moveEntry** method to rename files or folders.

To change file or folder attributes, call the **changeAttributes** method. This method takes the file system path string of a file or folder to change the attributes of and a string containing the new attributes for that file or folder. Use the **changeAttributes** method to make files or folders read-only and to clear or set the archive flag for files.

Listing Files and Folders

To collect information about files and folders on the FAT16/32 file system partition, use the listEntry method. The listEntry method takes the file system path string of a file or folder to collect information about and loads the listing methods with information about that file or folder. Described in Table 6 are the listing methods.

Table 6: Listing Methods

Method Name	Action
listName	Returns the listed entry's name string
listSize	Returns the listed entry's size in bytes
listCreationDay	Returns the listed entry's creation day
listCreationMonth	Returns the listed entry's creation month
listCreationYear	Returns the listed entry's creation year
listCreationSeconds	Returns the listed entry's creation seconds
listCreationMinutes	Returns the listed entry's creation minutes
listCreationHours	Returns the listed entry's creation hours
listAccessDay	Returns the listed entry's last access day
listAccessMonth	Returns the listed entry's last access month
listAccessYear	Returns the listed entry's last access year
listModificationDay	Returns the listed entry's modification day
listModificationMonth	Returns the listed entry's modification month
listModificationYear	Returns the listed entry's modification year
listModificationSeconds	Returns the listed entry's modification seconds
listModificationMinutes	Returns the listed entry's modification minutes
listModificationHours	Returns the listed entry's modification hours
listIsReadOnly	Returns if the listed entry is read-only
listIsHidden	Returns if the listed entry is hidden
listIsSystem	Returns if the listed entry is system
listIsDirectory	Returns if the listed entry is directory
listIsArchive	Returns if the listed entry is archive

The listing methods simply access information about the currently listed file or folder by the listEntry method. Call listEntry before calling any listing methods to load information about a target file or folder.

In addition to the listEntry method is the listEntries method. The listEntries method, when called in a loop, loads information about all files and folders in the current directory one at a time. Use the listEntries method to collect information about all files and folders in the current directory. The code below shows how to use listEntry, listEntries, and the listing methods.

```
PRI howToUseListEntry(fileSystemPathString)

  fat.listEntry(fileSystemPathString) ' Find an entry.

  ' Save the information returned by the below methods.
  ' Or print the information returned by the below methods.

  fat.listName
  fat.listSize
  fat.listCreationDay
  fat.listCreationMonth
  fat.listCreationYear
  fat.listCreationSeconds
  fat.listCreationMinutes
  fat.listCreationHours
  fat.listAccessDay
  fat.listAccessMonth
  fat.listAccessYear
  fat.listModificationDay
  fat.listModificationMonth
  fat.listModificationYear
  fat.listModificationSeconds
  fat.listModificationMinutes
  fat.listModificationHours
  fat.listIsReadOnly
  fat.listIsHidden
  fat.listIsSystem
  fat.listIsDirectory
  fat.listIsArchive

PRI howToUseListEntries | entryName

  fat.listEntries("W") ' Wrap around.
  repeat while(entryName := fat.listEntries("N"))
    ' "entryName" points to the string name of the next entry.

    ' Save the information returned by the below methods.
    ' Or print the information returned by the belowmethods.

    fat.listName
    fat.listSize
    fat.listCreationDay
    fat.listCreationMonth
    fat.listCreationYear
    fat.listCreationSeconds
    fat.listCreationMinutes
    fat.listCreationHours
    fat.listAccessDay
    fat.listAccessMonth
    fat.listAccessYear
    fat.listModificationDay
    fat.listModificationMonth
    fat.listModificationYear
    fat.listModificationSeconds
    fat.listModificationMinutes
    fat.listModificationHours
    fat.listIsReadOnly
    fat.listIsHidden
    fat.listIsSystem
    fat.listIsDirectory
    fat.listIsArchive
```

The **listEntries** method does not reset itself and wrap around to the top of the current directory when other file system methods are called (besides **listEntry**). This allows the **listEntries** method to collect the short names of files and folders in the current directory linearly while other file system methods are called in between calls to **listEntries**. E.g., **listEntries** can be used to open every file in the current directory.

```
PRI openAllFiles | entryName ' In the current directory.

  fat.listEntries("W") ' Wrap around.
  repeat while(entryName := fat.listEntries("N"))

    ' Make sure the entry is not a directory.
    ifnot(fat.listIsDirectory)
      fat.openFile(entryName, "R")
      ' Preform operations.
      fat.closeFile

PRI deleteAllFiles | entryName ' In the current directory.

  fat.listEntries("W") ' Wrap around.
  repeat while(entryName := fat.listEntries("N"))

    ' Make sure the entry is not a directory.
    ifnot(fat.listIsDirectory)
      fat.deleteEntry(entryName)
      ' Perform other operations.
```

Soft Load RAM from File

The Full File System Driver supports the ability to reboot the Propeller chip to run code from any valid Spin EEPROM or BIN file. To do this, call the **bootPartition** method, and pass to it the file system path string of the file to reboot the Propeller chip from. If the file is a valid Spin EEPROM or BIN file, the Propeller chip will immediately reload and begin executing the new code. Otherwise, the Propeller chip will shut down and wait for reset.

File Access

Perform file operations only after the file system is mounted. File operations consist of reading or writing bytes, words, and/or longs from or to a file. FAT16 and FAT32 support files up to about 4 GB in size; however, the Full File System Driver will only allow access to the first 2 GB of a file and will never report file sizes larger than 2 GB. The driver will not write past the 2-GB boundary. This limitation is due to the nature of the Spin language supporting only signed arithmetic operations.

A file must be open first for the file system to read or write data to it. Files can be opened for reading, writing, or appending. When a file is opened for reading, it cannot be written to. A file opened for writing, however, can be read and written. Opening a file for appending is similar to opening a file for writing except that data is written after the end of the file instead of to the beginning of the file.

Note: Files opened for writing are not truncated in length because they are opened for reading and writing at the same time. Delete and remake a file to truncate it in length.

The code below shows how to create a new file and then open it for writing. Remember that **newFile** returns a pointer to a string containing the name of the file it created.

```
PRI code ' Put the file system calls in a separate method to trap aborts.

  fat.mountPartition(0) ' Mount the default partition 0.  Can be 0 - 3.

  fat.openFile(fat.newFile(string("text.txt")), "W") ' Create and open text.txt.
```

The code above creates a new file called "text.txt" and opens it for writing. Follow the code below to open a previously created file called "text.txt" for reading.

```
PRI code ' Put the file system calls in a separate method to trap aborts.

  fat.mountPartition(0) ' Mount the default partition 0.  Can be 0 - 3.

  fat.openFile(string("text.txt"), "R") ' Open text.txt.
```

In addition, the code below shows how to open a file called "text.txt" for appending.

```
PRI code ' Put the file system calls in a separate method to trap aborts.

  fat.mountPartition(0) ' Mount the default partition 0.  Can be 0 - 3.

  fat.openFile(string("text.txt"), "A") ' Open text.txt for appending.
```

Table 7 describes the file access methods in the Full File System Driver.

Table 7: File Access Methods

Method name	Action
readByte	Returns a byte from the file and advances the file position by one.
readShort	Returns a short from the file and advances the file position by two
readLong	Returns a long from the file and advances the file position by four
readString	Returns a string from the file and advances the file position by the string size
writeByte	Puts a byte in the file and advances the file position by one
writeShort	Puts a short in the file and advances the file position by two
writeLong	Puts a long in the file and advances the file position by four
writeString	Puts a string in the file and advances the file position by the string size
readData	Reads an arbitrary amount of data from a file and advances the file position by that amount of data
writeData	Writes an arbitrary amount of data from a file and advances the file position by that amount of data
fileSize	Returns the file size in bytes of the opened file
fileTell	Returns the file position in bytes of the opened file

Access files opened for reading by using the **readByte**, **readShort**, **readLong**, **readString**, and **readData** methods. Any calls to the write methods while the file is open for reading will do nothing. Access a file open for writing, however, by calling the **writeByte**, **writeShort**, **writeLong**, **writeString**, or **writeData** methods in addition to calling the reading methods.

Note that the **readString** method intelligently reads bytes in from the file until it encounters 0 (ACSII Null), 10 (ASCII Line Feed), 13 (ASCII Carriage Return), runs out of space to store the string it is reading in from the file, or encounters the end of the file.

Also note that the **writeString** method quickly writes an arbitrary-sized string to the file.

To move back and forth inside of a file use the **fileSeek** method. The **fileSeek** method allows changing of the file position inside of the file when reading or writing.

When the **fileTell** and **fileSize** methods return the same value, the end of file has been reached. After the end of file has been reached, reads have no effect.

When writing data to a file opened for writing or appending, the Full File System Driver buffers the data temporarily before writing it to disk. The driver does this to improve read and write performance. Whenever enough data is written to fill up the buffer, or the file is closed, the driver flushes the buffer to the disk. The driver has a **flushData** method that gives the ability to flush written data to the disk at will. The **flushData** method comes in handy when writing to a file over a long period of time (think days) and it is necessary to make sure the driver writes all the data to the disk. The **flushData** method is not needed for normal operation, as the driver will handle flushing regularly.

File reading and writing performance improves as larger chunk sizes are used. The **readByte**, **writeByte**, **readShort**, **writeShort**, **readLong**, and **writeLong** methods are all bottlenecked by the Full File System Driver. For quick file access, use the **readData** and **writeData** methods with external data buffers. Refer to Table 8 for the average speeds of the reading and writing methods.

Table 8: File Access Methods Throughput

Method	Access throughput average
readByte	3 KB
readShort	6 KB
readLong	13 KB
readString	3 KB
writeByte	3 KB
writeShort	6 KB
writeLong	12 KB
writeString	110 KB
readData	241 KB
writeData	110 KB

Some outliers may exist from the table above. Below are a series of file access examples.

```
PRI code | counter, buffer[128] ' Put calls in a separate method to trap aborts.

  fat.mountPartition(0) ' Mount the default partition 0.  Can be 0 - 3.

  fat.openFile(fat.newFile(string("text.txt")), "W") ' Create and open text.txt.

  ' // Examples below. Note: If "text.txt" already exist abort...

  fat.writeString(string("Hello World!")) ' Write "Hello World!" to the file.
  fat.fileSeek(0) ' Go to the beginning of the file - was at position 12.
  ' Position 0 in the file is "H" ... position 11 in the file is "!".

  ' Note: After using fileSeek it is not possible to go back to position 12.
  '       This is because it doesn't exist yet.  Nothing has been written there.
  '       Position 11 is the last position which has "!" in it.
  '       Position 11 must be rewritten to get back to position 12.
  '       Moral of the story?  Try to only use file seek in a fixed length file.

  fat.readString(@buffer, 512) ' Read the string just written back from the file.
  ifnot(strcomp(@buffer, string("Hello World!"))) ' Should read back what was written.
    abort string("Um...") ' This should not happen.

  repeat counter from 0 to 9 ' Write "0123456789"
    fat.writeByte("0" + counter) ' Same as fat.writeString(string("0123456789")).

  fat.fileSeek(fat.fileTell - 4) ' Go back 4 steps.  Should be at "6".
  '                    "9""8""7""6"
  if(fat.readLong <> $39_38_37_36) ' Remember!  Little endian!
    abort string("Eh...") ' No good.

  fat.fileSeek(0) ' Back to the beginning again.

  ' Note: Seeking back and forth inside of a sector (512 Bytes) is quick!
  '       The farther fileSeek has to travel, the longer it takes!

  repeat until(fat.fileTell == fat.fileSize) ' Same as "repeat fat.fileSize"
    fat.readByte ' This is just an example of how to read every byte in a file.

  fat.writeData(@buffer, 512) ' Write 0.5KB quick.  Should mostly be zeros.

  fat.openFile(string("text.txt"), "A") ' Close and open the file again.
  ' Opening another file closes the previous one.

  fat.writeLong(1_234_567_890) ' Write some data at the end of the file.

  fat.flushData ' Make sure the data is written to the file.

  ' Okay, clean up time!

  fat.closefile ' Make sure that data is written to the file.
  ' fat.flushData is not needed when opening and closing a file quickly.
  ' fat.flushData is for cases in which the file is open for a long time (think days).
  ' Or, alternatively for cases in which data is written very slowly.

  ' The only way to shrink a file is to delete it and start over.
  fat.deleteEntry(string("text.txt")) ' So, delete it.
  fat.openFile(fat.newFile(string("text.txt")), "W") ' And remake it.

  fat.writeString(string("All Done!")) ' Done.
  fat.unmountPartition ' This function also closes any open files.
  ' Not unmounting the partition may cause bad things to happen.
```

Once finished reading or writing data from and to a file, close the file by calling the **closeFile** method. The **closeFile** method writes any buffered data to disk and then updates the file access and modification times and the archive flag. Always close a file opened for writing or it will become corrupted. For files opened for writing for a long period of time (think days), periodically call the **flushData** method to prevent possible corruption due to power failure, brown outs, etc. Files opened for reading do not have to be closed; they will not become corrupted if not closed, as files opened for writing will be. However, try to always close all opened files!

Multiple File Access

Each included instance of the Full File System Driver object allows one file open at a time. For more than one file open at a time, simply include more than one copy of the driver object. Each driver object shares the SD card and works independently of the other driver objects. The driver objects only share access to the file system and the secure digital card block driver. Call the **FATEngineStart** method only once for all the included copies of the driver object. Because each of the driver objects is independent from the other driver objects, each one of them needs to call the **mountPartition** method to mount a partition on the SD card, etc. Below is an example of using two instances of the driver object to have more than one file open at a time.

```
OBJ

  fat0: "SD-MMC_FATEngine.spin"
  fat1: "SD-MMC_FATEngine.spin"

PUB main | errorNumber0, errorNumber1, errorString

  ' "fatEngineStart" is only called once. Either driver object can call it.
  fat0.fatEngineStart( _dopin, _clkpin, _dipin, _cspin, _wppin, _cdpin, {
                     } _rtcdatpin, _rtcclkpin, _rtcbuslck)

  errorString := \code ' Returns the address of the error string or null.
  errorNumber0 := fat0.partitionError ' Returns the error number or zero.
  errorNumber1 := fat1.partitionError ' Returns the error number or zero.

  if(errorNumber0 or errorNumber1) ' Light a LED if an error occurs.
    outa := constant(|<_errorpin)
    dira := constant(|<_errorpin)

  repeat ' Wait until reset or power down.

PRI code ' Put the file system calls in a separate method to trap aborts.

  fat0.mountPartition(0) ' Mount the default partition 0.  Can be 0 - 3.
  fat1.mountPartition(0) ' Mount the default partition 0.  Can be 0 - 3.

  fat0.openFile(fat0.newFile(string("text.txt")), "W") ' Create and open text.txt.
  fat1.openFile(fat1.newFile(string("test.txt")), "W") ' Create and open test.txt.

  ' ... Each object interacts with the partition file system separately.

  fat0.unmountPartition ' Closes the open file and unmounts the partition.
  fat1.unmountPartition ' Closes the open file and unmounts the partition.
```

Multiple Cog File System Access and Locking

The Full File System Driver supports multiple cogs running the Spin interpreter accessing it at the same time. The driver accomplishes this by letting only one cog running the Spin interpreter access it at a time by using locks. This feature allows multiple cogs to access a single file using one instance of the driver or multiple cogs accessing multiple files through multiple instances of the driver. However, because the driver supports locks, deadlocking can occur. Deadlocking occurs when one cog locks the driver and never unlocks it, so other cogs cannot access it. Deadlocking will occur when a cog is "**cogstop**'ed" while it is still using the driver. Always wait until a cog has finished using the driver before "**cogstop**'ing" it.

Note that some of the file access methods can take a long time to complete and unlock the file system. This may cause a multi-cog system to become unresponsive and lock up.

Simultaneous File and Folder Access

Each instance of the Full File System Driver supports only one file open at a time. Additionally, each instance of the driver will close any file it has open after calling a non-file access method. More specifically, after calling **changeDirectory**, **deleteEntry**, **changeAttributes**, **moveEntry**, **newFile**, **newDirectory**, **listEntries**, **listEntry**, **bootPartition**, **formatPartition**, **mountPartition**, or **unmountPartition** in one instance of the driver, that instance of the driver will close any file it has open. Each instance of the driver will not allow an open file and use of the above methods at the same time—the drivers do this to prevent file access conflicts. However, each instance of the driver does not share with the other instances of the driver which file it has open. Nevertheless, all instances of the driver still share the underlying file system partitions. Because of this, there are some illegal operations that will cause file and folder corruption.

- Deleting a file with one instance of the Full File System Driver while it is open in another instance of the driver.
- Moving a file with one instance of the Full File System Driver while it is open in another instance of the driver.
- Formatting a partition with once instance of the Full File System Driver while another instance of the driver is still using that partition.

Also, more complicated file access operations can be preformed because of this "feature."

- Open a file for reading multiple times with multiple instances. This may be useful in certain situations. This is not dangerous.
- Open a file for writing multiple times with multiple instances. This may be useful in certain situations. This is dangerous, however, and will likely result in corruption unless file system buffering is very well understood.

Stack Space

Within the driver's source code, the "Stack Longs" value under each method header is an estimate of how large a Spin interpreter's stack size can increase when it calls the method.

The "Stack Longs" value only accounts for the stack size increase due to bookkeeping information, passed parameters, and local variables. It does not account for stack size increase due to expression evaluation or calls to other Spin objects. The estimated stack size for a Spin interpreter should be set to be at least 20% larger than the expected estimate stack usage from the information given above to account for expression evaluation and additional startup Spin interpreter booking information. It may be the case that the stack size never grows as large in practice as the maximum stack size estimate value—but this is okay.

Resources

Each archive zip file includes the Full File System Driver, the real time clock driver, the real time clock datasheet, and a simple SD card tester/profiler example/demo code file.

Full File System Driver with DS1302 RTC.zip
Full File System Driver with DS1307 RTC.zip
Full File System Driver with S35390A RTC.zip
Full File System Driver without RTC.zip

References

1. Information from SD Association; http://www.sdcard.org
2. SD Card Adapter Kit, Parallax Inc., #32313; www.parallax.com
3. micro-SD Card Adapter, Parallax Inc, #32212; www.parallax.com
4. Image courtesy of André LaMothe; www.xgamestation.com

Revision History

Version 1.0: original document.