MMC / SD MEMORY CARD
FAT16 / FAT32 DRIVER

TECHNICAL MANUAL

V1.06

embedded-code.com

INDEX

Driver Overview

MMC (MultiMediaCard) and SD (Secure Digital) memory cards provide embedded devices with a very inexpensive and convenient way of storing anything from very small to very large amounts of data. Using a MMC or SD card in your embedded device with the FAT filing system allows you to very easily read and write multiple files and exchange this data with other embedded devices and PC's. Apart from the convenience of such a powerful and flexible filing system, being able to read and write PC compatible files can add huge benefits to your product. However writing a MMC/SD FAT filing system driver is a complex and daunting task. This driver removes that complexity for you and allows you to read and write files with ease using either card type and the various mini versions of the MMC or SD card.

This driver has been specifically designed from the ground up for embedded applications using 8, 16 or 32 bit processors or microcontrollers. Whilst the code has been kept as small as possible, it hasn't been reduced to such a point that the driver becomes difficult to use. Instead great importance has been put on being able to use as many of the standard ANSI-C file system functions as possible and with as many of each of their features as possible.

The MMC / SD card FAT16 / FAT32 driver code has been designed and tested using ANSI compliant C compliers. Using the driver with other ANSI compliant C compliers and with other processors / microcontrollers should not present significant problems, but you should ensure that you have sufficient programming expertise to carry out any modifications that may be required to the source code. Embedded-code.com source code is written to be very easy to understand by programmers of all levels. The code is very highly commented with no lazy programming techniques. All function, variable and constant names are fully descriptive to help you modify and expand the code with ease.

The MMC / SD card FAT16 / FAT32 driver and associated files are provided under a licence agreement. Please see www.embedded-code.com/licence.php for full details.

The remainder of this manual provides a wealth of technical information about the driver as well as useful guides to get you going. We welcome any feedback on this manual and the driver.

As with any development project you should ensure that backup copies are made of any files stored on a MMC or SD card that is used with the driver until you have completed your development and thoroughly tested the operation of the driver in your application.

Features

Designed for both FAT16 and FAT32 formatted SD, SDHC (high capacity), MMC and MMCplus (high capacity) cards with a 4 pin serial interface to a microcontroller or processor.

Optimised for embedded designs. Only a single 512 data buffer is required for all operations. (It is not possible to write to MMC or SD cards without a 512 byte buffer as sectors have to be read to local memory, modified and written back as a whole).

Intelligent use of the local ram sector buffer. Read and writes of sector data only occur when necessary, avoiding unnecessary and slow repeated read or write operations to the MMC or SD card.

Optimised file delete function for fast deleting of large files. Instead of altering each FAT table entry one at a time, a complete sector of FAT table entries are altered in one operation before writing back to the card, resulting in a large speed improvement.

Provides the following standard ANSI-C functions:

fopen, fseek, ftell, fgetpos, fsetpos, ffs_rewind, fputc, putc, fgetc, getc, fputs, fgets, fwrite, fread, fflush, fclose, remove, rename, clearer, feof and ferror

Standard DOS '*' and '?' wildcard characters may be used in file operations.

Multiple files may be opened at the same time.

Optional real time clock support for applications that include time keeping. File creation, last modified and last accessed time and date values are automatically stored.

DRIVER TECHNICAL NOTES

The data area of MMC and SD memory cards is accessed through the use of a 512 byte sector buffer. All data read and write operations work through the reading and writing a 512 byte block of sector data. Therefore to modify a single byte, a complete sector of data must be read to local ram, modified and then the complete sector written back to the card.

Other flash memory devices, such as flash memory IC's also typically use the same system whereby a complete block of data must be erased to reset all of the bytes in that block back to 0xFF ready for writing again, as many flash memory technologies work on the principal of turning individual bits from high bits to low, not low to high.

This 512 byte buffer is an issue when it comes to designing a driver to provide fast read and write access. The reason is that as a programmer you want to be able to access individual bytes of a file without worrying about sectors, but you don't want the driver continuously reading and writing 512 bytes of data every time you modify a byte, resulting in painfully slow access. This driver overcomes these problems by only reading and writing when an operation needs to access a byte that is contained in a different sector on the card. Whilst this requires some instances of quite complex driver code, this complexity is worthwhile due to the massive speed improvements this approach provides.

If you want to gain an understanding of exactly how the driver works then this manual contains a thorough description of the layout of FAT based MMC / SD cards. Once you understand this each of the driver functions are relatively easy to understand. However you don't need to do this and if you just want to read and write FAT16 or FAT32 MMC or SD cards then you can skip these in-depth parts of the manual.

Finally you should also note that different MMC / SD memory cards can take different amounts of time to complete internal operations, such as preparing to read or writing a new sector of data. If your application is very time sensitive you may need to consider using some processor RAM memory to act as some sort of FIFO buffer for read and write operations. For example say you are designing a MP3 player that needs to send MP3 file data to a MP3 decoder IC within a certain response time when it requests it. You may find that a slow MMC or SD card might not be able to provide the next byte of data fast enough when it moves from one sector to the next, resulting in your MP3 decoder IC temporarily running out of data. By using some form of circular FIFO RAM buffer in your application you could read data from the MMC or SD card as one process, always trying to fill the data buffer so its full, and read data from the buffer to send to the MP3 decoder IC when it requests it as a separate interrupt based process.

ADDING THE DRIVER TO YOUR PROJECT

NOTES ABOUT OUR SOURCE CODE FILES

How We Organise Our Project Files

There are many different ways to organise your source code and many different opinions on the best method! We have chosen the following as a very good approach that is widely used, well suited to both small and large projects and simple to follow.

Each .c source code file has a matching .h header file. All function and memory definitions are made in the header file. The .c source code file only contains functions. The header file is separated into distinct sections to make it easy to find things you are looking for. The function and data memory definition sections are split up to allow the defining of local (this source code file only) and global (all source code files that include this header file) functions and variables. To use a function or variable from another .c source code file simply include the .h header file.

Variable types BYTE, WORD, SIGNED_WORD, DWORD, SIGNED_DWORD are used to allow easy compatibility with other compilers. A WORD is 16 bits and a DWORD is 32 bits. Our projects include a 'main.h' global header file which is included in every .c source code file. This file contains the typedef statements mapping these variable types to the compiler specific types. You may prefer to use an alternative method in which case you should modify as required. Our main.h header file also includes project wide global defines.

This is much easier to see in use than to try and explain and a quick look through one of the included sample projects will show you by example.

Please also refer to the resources section of the embedded-code.com web site for additional documentation which may be useful to you.

Modifying Our Project Files

We may issue new versions of our source code files from time to time due to improved functionality, bug fixes, additional device / compiler support, etc. Where possible you should try not to modify our source codes files and instead call the driver functions from other files in your application. Where you need to alter the source code it is a good idea to consider marking areas you have changed with some form of comment marker so that if you need to use an upgraded driver file its as easy as possible to upgrade and still include all of the additions and changes that you have made.

STEP BY STEP INSTRUCTIONS

Move The Main Driver Files To Your Project Directory

The following files are the main driver files which you need to copy to your main project directory:

`mem-ffs.c`	The FAT16/32 file system driver functions
`mem-ffs.h`	
`mem-mmcsd.c`	The lower level MMC / SD card driver functions
`mem-mmcsd.h`	

Move The Generic Global Defines File To You Project Directory

The generic global file is located in each driver sample project directory. Select the most suitable sample project based on the compiler used and copy the following file to your main project directory:

`main.h` The embedded-code.com generic global file:

Check Driver Definitions

Check the definitions in each of the following files to see if any need to be adjusted for the microcontroller / processor you are using, and your hardware connections.:-

```
mem-ffs.h
mem-mmcsd.h
```

Check the definitions in the following file and adjust if necessary for your compiler-

```
main.h
```

Timers

You will need to provide some form of timer for the driver. Typically this can be done in your applications general heartbeat timer if you have one. Do the following every 10mS:-

```
//----- FAT FILING SYSTEM DRIVER TIMER -----
if (ffs_10ms_timer)
	ffs_10ms_timer--;
```

If you do not have a matching timer then using a time base that is slightly greater than 10mS is fine. Note that the timer must be interrupt based as it is used to provide timeout protection in some of the driver functions.

SPI Port Setup

The SPI interface needs to function in the following way:-

Clock is low in idle bus state
Data is valid on the rising edge of the clock. Data is outputted on the falling edge of the clock.

The speed of the SPI bus is set using 3 separate defines in mem-mmcsd.h. It needs to be between 100KHz and 400KHz when initialising a new card, and up to 20MHz or 25MHz for MMC or SD cards once initialised.

> If your device does not have an SPI port, or if you suspect you may be experiencing issues with your devices SPI peripheral (e.g. due to a silicon bug), a bit based SPI interface is available using the included files mem-spi.c and mem-spi.h in your project. See the mem-spi.h header file for details.

Application Requirements

In each .c file of your application that will use the driver functions include the 'mem-ffs.h' file.

You will need to periodically call the drivers background processing function. Typically this can be done as part of your applications main loop. This function looks to see if a MMC or SD card has been inserted or removed and updates the driver appropriately. Add the following call:-

```
//----- PROCESS FAT FILING SYSTEM -----
ffs_process();
```

IMPORTANT HARDWARE DESIGN NOTES

Please see the:

'Signal Noise Issues With MMC & SD Memory Cards (& Clocked Devices In General)

page in the resources area of our web site for details of a common PCB level problem experienced when using MMC and SD memory cards.

USING THE SAMPLE PROJECTS

Sample projects are included with the driver for specific devices and compilers. The example schematics at the end of this manual detail the circuit each sample project is designed to work with. You may use the sample projects with the circuit shown or if desired use them as a starting block for your own project with a different device of compiler. To use them copy all of the files in the chosen sample project directory into the same directory as the driver files and then open and run using the development environment / compiler the project was designed with.

SAMPLE PROJECTS INCLUDED

Rowley CrossWorks Compiler for ARM

Compiler: Rowley Associates CrossWorks 2 C Compiler for ARM
Device: NXP LPC2365

Microchip C18 Compiler

Compiler: Microchip C18 MPLAB C Compiler for PIC18 family of 8 bit microcontrollers
Device: PIC18F4620
Notes: The C18 project uses a modified version off the Microchip standard linker script for the PIC18F4620. This is required as the C18 compiler does not support data buffers over 256 bytes without a modification to the linker script to define a larger bank of microcontroller ram. A 512 byte buffer is required by the driver. You will see in the sample linker script that 2 consecutive gpr banks have been removed and instead replaced with:-

DATABANK NAME=ffs_512_byte_ram_section START=0x#00 END=0x#FF

where '0x#00' is the start address of the first removed bank and '0x#FF ' is the end address of the second removed bank. If modifying other device linker scripts ensure that you also check the bank used by the stack and change it to another bank if it conflicts.

Microchip C30 Compiler

Compiler: Microchip C30 MPLAB C Compiler for PIC24 family of 16 bit microcontrollers and dsPIC digital signal controllers
Device: PIC24HJ64GP206

SAMPLE PROJECT FUNCTIONS

When run the 2 LED's operate as follows:-

Red LED indicates that PCB is powered up but no card is detected

When a card is inserted and has been detected the red LED goes off and the green LED lights.

When the switch is pressed the following occurs:
- All files in the root directory are deleted
- A new text file called test.txt is created containing example test data.
- A new Excel compatible spreadsheet file called test.csv is created containing test data from the test.txt file.
- When the file operations have been completed the green LED goes off.

If there is a file operation error both LED's will light.

USING THE DRIVER IN YOUR PROJECT

Checking If A MMC or SD Card Is Available

The following example checks to see if a MMC or SD card is available to use:-

```
//IS A FAT FORMATTED MMC/SD CARD INSERTED AND READY TO USE?
if (ffs_card_ok)
{

}
```

MMC / SD Card Operations

Below is a list of the available functions and a detailed description of each is provided later in this manual. The included sample projects contain examples of using many of the driver functions.

`ffs_fopen`	Opens a file for read and or write access.
`ffs_fseek`	Change the byte location in the file which the next read or write access will address.
`ffs_fsetpos`	An alternative to `ffs_seek`. The value used is intended to be file system specific and obtained using the `ffs_getpos` function. However as the type is recommended to be a long and this doesn't provide enough space to store everything needed for the low level file position this function calls the `ffs_fseek` function.
`ffs_ftell`	Returns the current position within the file (the next byte that will be read or written).
`ffs_fgetpos`	An alternative to `ffs_tell`. The value returned is intended to be file system specific and only to be used with fsetpos. However as the position type is recommended to be a long and this doesn't provide enough space to store everything needed for the low level file position this function calls the `ffs_tell` function.
`ffs_rewind`	The file byte pointer is set to the first byte of the file and the file access error flag is cleared if it has been set.
`ffs_fputc` *or* `ffs_putc`	Write byte to file
`ffs_fgetc` *or* `ffs_getc`	Read Byte From File
`ffs_fputs` *or* `ffs_fputs_char`	Writes a string to the file until a null termination is reached.
`ffs_fgets`	Reads characters from file and stores them into the specified buffer until a newline (\n) or EOF (end of file) character is read or (length - 1) characters have been read.
`ffs_fwrite`	Writes `count` number of items, each one with a size of `size` bytes, from the specified `buffer`.

`ffs_fread` — Reads `count` number of items each one with a size of `size` bytes from the file to the specified `buffer`.

`ffs_fflush` — Write any data that is currently held in microcontroller / processor ram that is waiting to be written to the card. Update the file filesize value if it has changed. This function does not need to be called by your application, but may be called if your application opens a file for a long period of time to avoid data loss if your device suddenly looses power.

`ffs_fclose` — Closes an open file, saving any unsaved data to the card and updating the file filesize value if it has changed.

`ffs_remove` — Delete file

`ffs_rename` — Rename file

`ffs_clearerr` — Clear Error & End Of File Flags

`ffs_feof` — Has End Of File Been Reached

`ffs_ferror` — Has An Error Occurred During File Access

`ffs_is_card_available` — Is A Card Inserted and Available

Characters That May Be Used In DOS Compatible File Names

Upper case letters A-Z (lowercase will be modified to uppercase).
Numbers 0-9
Space (though trailing spaces are considered to be padding and not a part of the file name)
! # $ % & ( ) - @ ^ _ ` { } ~ '
Values 128-255

Partitions

This driver does not support multiple partitions. It will access the first partition of a MMC or SD card. Other partitions will not be damaged, but they cannot be accessed.

Working With Multiple Files

You are able to open multiple files at the same time and perform any operation on any of these files at any time. However all read and write operations involve reading a complete 512 byte block of data from the MMC or SD card and storing the complete block back to the card if any of the data has been modified before moving onto another block of data. The driver deals with this block requirement in an intelligent way, only reading and writing a block when it has to. If working on more than one file best speed will be achieved by working on one file as much as possible before working on another file. This is because each time you swap to a different file the driver has to save or dump the block of data currently being written or read and then load the data block being written or read for the other file. Therefore if doing an operation such as copying data from one file to another try and copy as much data as possible to processor ram before starting writing it to the other file. You don't have to, but doing this will significantly increase the speed of your application.

Ensure Data Is Saved For Write Operations

Files may be opened and kept open indefinitely. However you should try and carry out file write operations in one process and close the file again when it is not required in case your product should loose power. If power is lost while a file is open any data that has been written since the last close of the file may be lost, as the current file size value may not have been written back to directory entry for the file. Whilst the data may have already been stored to the MMC or SD card, without the file size value the next time the file is accessed by the driver or another device the data will effectively not exist and the sectors that contain it will be lost on the card (until it is formatted or a disk repair utility is run). In theory the file size value could be updated every time a new block of data is written to the card, however the driver does not do this as it would significantly slow down bulk write operations. If you need to keep a file open for a long period of time then you should periodically call the ffs_fflush function to ensure that the most recent data is saved.

Reading & Writing A Text File

.txt files are as simple as it gets. They are simply comprised of ASCII bytes with a CR (carriage return) & LF (line feed) character at the end of each line of text.

In addition to being a great way of storing and retrieving configuration and operating data for your project, writing text files can be a really useful way of debugging complex problems with an application, by being able to write large quantities of text and then analysing this with any standard text application on a PC. In addition, if your designing a product that may experience problems in certain installations it is typically quite a simple matter to write some code to provide logging of the products operation, such as communications sent and received, to a .txt file on a MMC or SD card which a user can then email you for remote analysis.

Reading & Writing A Spreadsheet File

.csv files are a great way of reading and writing spreadsheet data. They are exactly the same as a text file, except that the comma ',' character is used to mark moving on to the next column. Every time the CF and LF characters are used the next row is started.

.csv files may be directly read and written by Microsoft Excel™.

Fast Reading Of Bulk File Data

The ANSI-C fread function is provided to allow blocks of data to be read but this can be too slow for some applications. This is because of the overhead the C library functions require which is fine and very useful on systems with enough processor power so it doesn't notice, but can waste huge amounts of clock cycles in speed sensitive embedded applications. The following is a simple method that will allow complete sectors (512 bytes) to be read as a data block, used by your application as required and then the next sector read.

Open a file for reading using fopen as normal and then use the fgetc function to read the first byte. In reading the first byte the driver will actually read the first sector of file data into the drivers sector buffer FFS_DRIVER_GEN_512_BYTE_BUFFER. Subsequent calls to the fgetc or other read functions will simply read data from this buffer without accessing the card, but with all of the background checks the driver has to do for each byte read. Instead you can simply access the buffer directly in your application. When you are ready to read the next sector do the following:-

```
your_file_name->current_byte_within_file += 511;
your_file_name->current_byte += 511;
```

That's it. In modifying the 2 above values you reposition the drivers internal processes into thinking that it last accessed the last byte in the current sector. To load the next sector call the fgetc function again and repeat the process. When using this method just bear in mind that you will need to detect the end of file yourself as the last sector read for a file will contain unused data bytes unless the file size is an exact multiple of 512 bytes.

An example:

```
our_file_1 = ffs_fopen(filename_test_txt, read_access_mode);
while( )  //Repeat this as many times as you wish
{
      i_temp = ffs_fgetc(our_file_1);
      //The FFS_DRIVER_GEN_512_BYTE_BUFFER has been loaded with
      //the next 512 bytes which you can now read directly from
      // the buffer without calling any ffs functions.

      //Then do this:
      our_file_1->current_byte_within_file += 511;
      our_file_1->current_byte += 511;
}
//This example doesn't check for file end – remember to check for this if you
need to
```

Fast Writing Of Bulk File Data

This can be achieved in the same was as fast reading of bulk data above. Use the fputc function to write the first byte of a new sector. Then write the rest of the data directly to the buffer. When you are ready to write the next sector do the following:-

```
your_file_name->current_byte_within_file += 511;
your_file_name->current_byte += 511;
your_file_name->file_size += 511;
```

In modifying the 3 above values you reposition the drivers internal processes into thinking that it last wrote to the last byte in the current sector. To write the next sector call the fputc function again and repeat the process.

An example:

```
our_file_1 = ffs_fopen(filename_test_txt, write_access_mode);
while( )  //Repeat this as many times as you wish
{
      ffs_fputc((int)b_temp, our_file_1);
      //The FFS_DRIVER_GEN_512_BYTE_BUFFER has been prepared for
      //a write of 511 further bytes which you can now write
      //directly to the buffer without calling any ffs functions.

      //Then do this:
      our_file_1->current_byte_within_file += 511;
      our_file_1->current_byte += 511;
      our_file_1->file_size += 511;
}
//This example doesn't check for a file write error – remember to do this if
you wish to check for errors
```

N.B. For even faster writing of large quantities of data it may be helpful to combine this technique with the use of the ffs_change_file_size function (see the ffs_change_file_size section of this manual for details).

Using MMC or SD Cards For Firmware Updates

A MMC or SD card may be used to allow new firmware files to be read off a card and programmed into your devices memory. You could use a standard raw .hex format or your own encrypted format. Remember that if reading the file directly off the card and into program memory you will need to allow sufficient boot loader program memory space for the MMC / SD card driver. If space is at a premium the driver could be 'hacked' down to the bare bones of just reading files with no writing or file re-positioning capabilities to reduce its size.

Deleting Files

Deleting a single file

```
const char filename_1[] = {"test.txt"};

ffs_remove(filename_1);
```

Deleting all files in the root directory:-

```
const char filename_all[] = {"*.*"};

while (ffs_remove(filename_all) == 0)
      ;
```

Searching In The Directory

There is no function that directly provides this, as its not provided by the standard ANSI-C functions. However, a relatively simple way of achieving this is to add a global variable to the driver that is usually zero, or add an additional variable to the `ffs_find_file function` declaration. In the `ffs_find_file` function use this variable so that if it is greater than zero the function does not return when it finds a matching file, but instead decrements the value and looks for the next match. When used with wildcard characters in the file name this allows you to find each matching file in turn, by setting the variable to zero and then every time the function returns with a cluster number for a match you set it to the last value +1, continuing until the functions returns with the not found value.

DISK VIEWING & EDITING UTILITIES

If you want to be able to view the contents of a MMC or SD card on your PC, which can be very useful when debugging or just learning about how disks are structured, then the WinHex application is very good. This is available from http://www.x-ways.net.

INFORMATION

MMC / SD MEMORY CARDS & FAT FILING SYSTEM

Mechanically MMC and SD cards are very small, with smaller compatible variants also available. They are low power and may be used with +3V3 systems. They use a serial interface based on the SPI specifications with fast transfer speeds possible (0-20MHz max clock rate for MMC, 0-25MHz max clock rate for SD) using only 4 pins. Data reliability is also provided by built-in defect management and error correction technologies. Whilst MMC and SD cards may also be communicated with using a 4 bit data interface this protocol is protected and not available without significant licence payments. The MMC card SPI interface protocol is available without any licence fee payable and is therefore more widely used than the 4 bit significantly more complex (and expensive!) protocol. SD cards are backwards compatible with the MMC card SPI interface and therefore this is typically the interface of choice for SD cards also. Note that the 'Secure' of Secure Digital, whilst available to licensed developers, is not widely used and you can just think of SD cards as a standard memory card in the same way as MMC cards (you do not need to implement security functionality to use them).

At the simplest level a MMC or SD card is just a large memory array which may be used in a similar way to a standard flash memory IC. Very simple applications may just use a MMC or SD card like any other memory device, storing data on it as required by the application. However this has the obvious limitation that the contents of the card is only readable and writable by the device that is using it. To allow other devices to easily read and write data to the card requires the use of a standardised file system. If a filing system is chosen that is also used by computers then sharing data with computer applications is made very simple.

There are 3 flavours of FAT (File Allocation Table):- FAT12, FAT16 and FAT32. FAT12 has now effectively become obsolete as the very small memory sizes of card this was useful for (<=16MB) are no longer generally available. This leaves FAT16 and FAT32. The 16 and 32 simply refer to the size of the cluster value in bits, although FAT32 is actually only 28 bits as 4 bits are reserved (see below for an explanation of clusters etc). This simply means that a FAT32 table takes up more space on a disk (or memory card), as each entry uses more bytes, but it allows addressing of larger memory sizes with smaller cluster sizes, resulting in less wastage of disk space. This use of smaller cluster sizes can quickly pay off in terms of efficiency as less space wastage at the end of each file frees up more space than the larger FAT32 table uses up.

Limits of FAT16

- Maximum volume size is 2GB
- Maximum file size is 2GB
- Maximum number of files is 65,517
- Maximum of 512 files or folders per folder

Limits of FAT32

- Maximum volume size is 2TB
- Maximum file size is 4GB
- Maximum number of files is 268,435,437
- Maximum of 65,534 files or folders per folder

You may think that you don't need anything more than FAT16 for your application if you don't plan to store more than 2GB of data on a MMC or SD card. After all, many embedded applications only need to store relatively small amounts of data. However MMC and SD cards with capacities greater than 256MB are typically supplied pre-formatted with FAT32. This is because FAT32 uses larger volumes more efficiently than FAT16 and is also less susceptible to a single point of failure due to the use of a backup copy of critical data structures in the boot record. Therefore if you use a driver that only supports FAT16 for your application your users will need to find a PC with a MMC or SD card adaptor to re-format larger capacity cards to be FAT16 before they can be used with your device. You also run the risk of increased technical support demands from users who haven't read your instructions or don't understand how to format a card as FAT16 instead of the default FAT32 and can't work out why their new MMC or SD card won't work in your device. Using a driver that supports FAT16 and FAT32 doesn't result in a large amount of additional code space by today's standards, as the two systems are very similar, and it makes life a lot easier for you and your users.

See the 'Layout of a MMC or SD Card With FAT' section later in this manual for detailed information of the FAT16 and FAT32 filing system.

MMC, SD AND FAT LICENSING

The implementation and use of the FAT file system, the MMC and the SD specifications may require a license from various entities, including, but not limited to Microsoft® Corporation, IBM, SD Card Association and the MultiMediaCard Association. It is your responsibility to obtain information regarding any applicable licensing requirements.

Microsoft offers licensing for the use of its FAT filing system on a per unit sold basis. However it is generally viewed that this only applies to applications that implement the patented long file name system (LFN). It is our understanding that if long filenames are not used then no licence fee is due, however you should ascertain if you agree with this view yourself (to our knowledge Microsoft have not stated this but others have determined this based on original releases of the FAT standard by Microsoft).

IBM patents may also apply to technology supporting extended attributes within the file system.

Our understanding of the MMC and SD card licensing requirements are that no licence fee is payable if using the SPI bus mode as the required per card licence fee is paid by card manufacturers. However if you require legal clarification of this you should contact the relevant organisation yourself.

SPECIFICATIONS

Card Capacities

This driver uses a buffer / block size of 512 bytes which is the standard block size supported by all MMC & SD cards. Some 2GB and 4GB SD cards provide a 1024 byte or 2048 byte block size as this was required prior to the release of V2.00 of the SD Physical Layer Specification. There is some confusion regarding this in relation to 2GB and 4GB SD cards. V1.01 of the SD specification allowed the original (V1.00) maximum block size of 512 bytes to be changed to 1024 or 2048 bytes, to deal with memory capacities of 2GB and 4GB. This lead to compatibility problems as host devices adhering to the V1.00 specification either did not recognise 2GB or 4GB cards, or would incorrectly interpret the card as 1GB and only access the first 1GB.

The issue is not to do with problems of being able to access data beyond 1GB using the actual read and write commands (which use a 32 bit address so have no problems), but is to do with the card identification data that a host uses to determine the capacity of a card. Due to the specification limiting the maximum sectors per cluster to 4096 and the number of blocks per cluster to 512, a buffer size of 512 bytes meant a limit of 1GB (4096 clusters x 512 blocks per cluster x 512 bytes per block). By changing the block size to 1024 or 2048 bytes card sizes of 2GB and 4GB can be specified in this identification data. However, although a card may specify that it has a maximum buffer size of 1024 or 2048 bytes there is no requirement to use it. This driver will correctly access 2GB and 4GB SD cards because it does not utilise the card identification data (it doesn't need to as it doesn't provide formatting) and because it specifies a block size of 512 bytes when initialising a card.

V2.00 of the SD specification addresses this problem and allows for higher card capacities. New SD cards of capacities greater that 2GB now use the SDHC standard, which allows for capacities of up to 2TB (although not all of this capacity is currently allowed under the official specification). It is also now specified that the block size must always be a maximum of 512 bytes to provide a common memory requirement and backwards compatibility. 2GB and 4GB SD cards may continue to specify to a host that they have a maximum block size of 1024 bytes or 2048 bytes, but to adhere to V2.00 they must not allow a block size of greater than 512 bytes to actually be used with the read and write commands.

Note that SDHC cards use an alternative addressing method that is not backwards compatible with SD cards, so although physically compatible a host needs to implement the new addressing in software to allow access to a SDHC card.

This driver supports the following cards (operating at the standard +3.3V):-

- All standard SD cards (up to 4GB which is the maximum possible)
- All standard SDHC cards
- All standard MMC cards
- All standard MMC Plus cards

Card Voltages

This driver is designed for standard +3.3V powered MMC and SD cards. Use of cards at other voltages may require additions to the driver to provide voltage compatibility checking.

Reduced Size Cards

The reduced size versions of the SD and MMC cards are electrically and software compatible. Only the physical size is different.

Formatting

This driver does not provide a format function. The reason for this is that formatting is complex and therefore code space heavy. All MMC and SD cards are supplied pre formatted so the inclusion of a format feature is not generally required.

Sub Directories

To avoid a significantly large code space requirement this driver supports reading and writing of files in a MMC or SD cards root directory only.

Long Filenames

This driver does not support long file names. Adding long filename support would use additional code space which is not desirable in many embedded applications, and is also subject to patent / licence restrictions / costs as Microsoft holds patents for the long filename specification. Files stored on a card using a long file name may still be accessed using their DOS equivalent short file name.

Using The Driver With a RTOS or Kernel

The stack / driver is implemented as a single thread so you just need to make sure it is always called from a single thread (it is not designed to be thread safe).

Code and Data Memory Requirements

C18 Compiler Code & Data Size

The following are based on compiling the complete PIC18 demo project (including the driver) using the Microchip C18 compiler with all optimisations turned on.

Approximately 11522 program memory words (16 bit)

Approximately 799 bytes of RAM. This includes a continuous 512 byte buffer that is required by the driver (it is possible to share this buffer with other parts of an application – see the 512 Byte Buffer Define section of this manual).

An additional 22 bytes of static RAM are required for each file that may be opened simultaneously (set by the FFS_FOPEN_MAX define).

The driver requires a moderate amount of variable storage space from the stack for its functions.

C30 Compiler Code & Data Size

The following are based on compiling the complete PIC24 demo project (including the driver) using the Microchip C30 compiler with all optimisations set to smallest code size.

Approximately 5217 program memory words (24 bit)

Approximately 600 bytes of RAM. This includes a continuous 512 byte buffer that is required by the driver (it is possible to share this buffer with other parts of an application – see the 512 Byte Buffer Define section of this manual).

The driver requires a moderate amount of variable storage space from the stack for its functions.

MMC / SD Card Mode

The driver accesses a MMC or SD card using the licence free SPI mode.

MMC & SD Memory Card Specifications

The MMC and SD card SPI bus specifications are available from the following web sites:-
http://www.sdcard.org
http://www.mmca.org

If you need to read these specifications take care to ensure that you are reading the correct section of the specifications when dealing with SPI bus commands. The commands and responses used with the 4 bit parallel interface (not supported by this driver) are not exactly the same as the SPI based commands.

How The Driver Works

Note – this section of the manual is for information only. You do not need to read and understand this large and in depth section to use the driver! However you may want to if you wish to gain an understanding of how each of the driver components works.

The Driver Functions & Defines

Pin Defines

`FFS_CE`	MMC / SD card Chip select pin (output)
`FFC_DI`	DO pin of MMC / SD card, DI pin of processor (used by the driver to check if pin is being pulled low by the card) (input)

The MMC or SD card detect pin is assigned using several defines to make it easy to use a direct microcontroller / processor pin or an external input buffer IC:-

`FFS_CD_PIN_REGISTER`	The register that should be read when reading the card detect pin state (e.g. the port register, or a ram register that gets read from a buffer IC).
`FFS_CD_PIN_BIT`	The bit of the register that is card detect pin (must be one of 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02 or 0x01).
`FFS_CD_PIN_FUNCTION`	Optional function to call to read the `FFS_CD_PIN_REGISTER`. Just comment this out if its not required (i.e. if your not using an external buffer IC).
`FFS_CD_PIN_NC`	Optional define which should be included if the card socket card detect pin is normally closed (breaks when a card inserted), or should be commented out if pin is normally open. A 0V common pin is assumed for this with the card detect pin pulled up by a resistor. If using a +v common with a pull down resistor then reverse the logic of this define.

SPI Bus Defines

`FFS_SPI_BUF_FULL`	A bit definition that is >0 when the SPI receive buffer contains a received byte, also signifying that transmit is complete.
`FFS_SPI_TX_BYTE(data)`	A macro to write a byte and start transmission over the SPI bus.
`FFS_SPI_RX_BYTE_BUFFER`	Register that the last received SPI bus byte may be read from.

512 Byte Buffer Define

`FFS_DRIVER_GEN_512_BYTE_BUFFER`	The microcontroller / processor ram buffer that is used to buffer a complete sector of MMC or SD card data. A define is used as some compilers may have special requirements to create a large data buffer. The driver only accesses the buffer using pointers, in case your compiler requires this. This buffer may also be shared with other functions in your application if you call the `ffs_fflush()` function for each open file and set `ffs_buffer_contains_lba = 0xFFFFFFFF` first.

Watchdog Timer Define

CLEAR_WATCHDOG_TIMER	Use this if you have a watchdog timer that needs to be reset for operations that can take a long time. Just comment this out if its not required.

User Options

`FFS_FOPEN_MAX`	The maximum number of files that may be opened simultaneously (1 - 254). 22 bytes of memory are required per file.

Standard Type And Function Names

For ease of interoperability this driver uses modified version of the standard ANSI-C function names and FILE data types. To avoid conflicting with your compilers stdio.h definitions you can comment out this section and use the modified ffs_ (flash filing system) names in your code. If you want to use the ANSI-C standard names then un-comment this section:-

```
#define     fopen           ffs_fopen
#define     fseek           ffs_fseek
#define     ftell           ffs_ftell
#define     fgetpos         ffs_fgetpos
#define     fsetpos         ffs_fsetpos
#define     rewind          ffs_rewind
#define     fputc           ffs_fputc
#define     fgetc           ffs_fgetc
#define     fputs           ffs_fputs
#define     fgets           ffs_fgets
#define     fwrite          ffs_fwrite
#define     fread           ffs_fread
#define     fflush          ffs_fflush
#define     fclose          ffs_fclose
#define     remove          ffs_remove
#define     rename          ffs_rename
#define     clearerr        ffs_clearerr
#define     feof            ffs_feof
#define     ferror          ffs_ferror
#define     putc            ffs_putc
#define     getc            ffs_getc

#define     EOF             FFS_EOF
#define     SEEK_SET        FFS_SEEK_SET
#define     SEEK_CUR        FFS_SEEK_CUR
#define     SEEK_END        FFS_SEEK_END
```

Open File

```
FFS_FILE* ffs_fopen (const char *filename, const char *access_mode)
```

This function opens a file for read and or write access.

For ease of use this driver does not differentiate between text and binary mode. You may open a file in either mode (or neither) and all file operations will be exactly the same (basically is if the file was opened in binary mode. LF characters will not be converted to a pair CRLF characters and vice versa. This makes using functions like fseek much simpler and avoids operating system difference issues. (If you are not aware there is no difference between a binary file and a text file – the difference is in how the operating system chooses to handle text files)

`filename`	Only 8 character DOS compatible root directory filenames are allowed. Format is F.E where F may be between 1 and 8 characters and E may be between 1 and 3 characters, null terminated, non-case sensitive. The '*' and '?' wildcard characters may be used.	
`access_mode`	"r"	Open a file for reading. The file must exist.
	"r+"	Open a file for reading and writing. The file must exist.
	"w"	Create an empty file for writing. If a file with the same name already exists its content is erased.
	"w+"	Create an empty file for writing and reading. If a file with the same name already exists its content is erased before it is opened.

"a"	Append to a file. Write operations append data at the end of the file. The file is created if it doesn't exist.
"a+"	Open a file for reading and appending. All writing operations are done at the end of the file protecting the previous content from being overwritten. You can reposition (fseek) the pointer to anywhere in the file for reading, but writing operations will move back to the end of file. The file is created if it doesn't exist.

Return value. If the file has been successfully opened the function will return a pointer to the file. Otherwise a null pointer is returned (0x00).

Move File Byte Pointer

`int ffs_fseek (FFS_FILE *file_pointer, long offset, int origin)`

This function allows you to change the byte location in the file which the next read or write access will address. The function is quite complex as it looks to see if the new location is in the same cluster as the current location to avoid having to read all of the FAT table entries for the file from the file start where possible, which results in a large speed improvement.

`file_pointer` Pointer to the open file to use.

`origin` The initial position from where the offset is applied
FFS_SEEK_SET (0) Beginning of file
FFS_SEEK_CUR (1) Current position of the file pointer
FFS_SEEK_END (2) End of file

`offset` Signed offset from the position set by origin

returns 0 if successful, 1 otherwise

`int ffs_fsetpos (FFS_FILE *file_pointer, long *position)`

This function is an alternative to `ffs_seek`. The value used is intended to be file system specific and obtained using the `ffs_getpos` function. However as the type is recommended to be a long and this doesn't provide enough space to store everything needed for the low level file position this function calls the `ffs_fseek` function.

Get The Current Position In The File

`long ffs_ftell (FFS_FILE *file_pointer)`

This function returns the current position within the file (the next byte that will be read or written).

`int ffs_fgetpos (FFS_FILE *file_pointer, long *position)`

This function is an alternative to `ffs_tell`. The value returned is intended to be file system specific and only to be used with fsetpos. However as the position type is recommended to be a long and this doesn't provide enough space to store everything needed for the low level file position this function calls the `ffs_tell` function.

Returns 0 if successful, 1 otherwise

Set File Byte Pointer To Start Of File

`void ffs_rewind (FFS_FILE *file_pointer)`

The file byte pointer is set to the first byte of the file and the file access error flag is cleared if it has been set.

`file_pointer` Pointer to the open file to use.

Write Byte To File

`int ffs_fputc (int data, FFS_FILE *file_pointer)`
or
`ffs_putc(int data, FFS_FILE *file_pointer)`

`file_pointer`	Pointer to the open file to use.
`data`	The data byte to write which is converted to a byte before writing (the int type is specified by ANSI-C)
Returns	If there are no errors the written character is returned. If an error occurs FFS_EOF is returned.

Read Byte From File

```
int ffs_fgetc (FFS_FILE *file_pointer)
```
or
```
int ffs_getc (FFS_FILE *file_pointer)
```

`file_pointer`	Pointer to the open file to use.
Returns	The byte read is returned as an int value (int type is specified by ANSI-C). If the End Of File has been reached or there has been an error reading FFS_EOF is returned.

Write String To File

```
int ffs_fputs (const char *string, FFS_FILE *file_pointer)
```
or
```
int ffs_fputs_char (char *string, FFS_FILE *file_pointer)
```

This function writes a string to the file until a null termination is reached. The null termination is not written to the file. If a new line character (\n) is required it should be included at the end of the string

The alternative `ffs_fputs_char` function is not part of the ANSI-C standard but may be needed writing a string from ram with compilers that won't deal with converting the ram string to a constant string.

Returns	Non-negative value if successful. If an error occurs FFS_EOF is returned.

Read String From File

```
char* ffs_fgets (char *string, int length, FFS_FILE *file_pointer)
```

This function reads characters from file and stores them into the specified buffer until a newline (\n) or EOF character is read or (length - 1) characters have been read. A newline character (\n) is not discarded. A null termination is added to the string

Returns	Pointer to the buffer if successful. A null pointer (0x00) if there is an error of the end-of-file is reached (use ffs_ferror or ffs_feof to check what happened).

Write Data Block To File

```
int ffs_fwrite (const void *buffer, int size, int count, FFS_FILE *file_pointer)
```

Writes `count` number of items, each one with a size of `size` bytes, from the specified `buffer`. No translation occurs for files opened in text mode. The total number of bytes to be written is (`size` x `count`).

Returns	The number of full items (not bytes) successfully written. This may be less than the requested number if an error occurred.

Read Data Block From File

```
int ffs_fread (void *buffer, int size, int count, FFS_FILE *file_pointer)
```

Reads `count` number of items each one with a size of `size` bytes from the file to the specified `buffer`. Total amount of bytes read is (`size` x `count`).

Returns	The number of items (not bytes) read is returned. If this number differs from the requested amount (`count`) an error has occurred or the End Of

File has been reached (use ffs_ferror or ffs_feof to check what happened).

(For a very fast method of reading complete sectors at a time see the 'Using The Driver In A Project' section later in this manual).

Store Any Unwritten Data To The Card

```
int ffs_fflush (FFS_FILE *file_pointer)
```

Write any data that is currently held in microcontroller / processor ram that is waiting to be written to the card. Update the file filesize value if it has changed.

This function does not need to be called by your application, but may be called if your application opens a file for a long period of time to avoid data loss if your device suddenly looses power.

Returns	0 if successful, 1 otherwise

Close File

```
int ffs_fclose (FFS_FILE *file_pointer)
```

Closes an open file, saving any unsaved data to the card and updating the file filesize value if it has changed.

Returns	0 if successful, 1 otherwise

Delete File

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

This function is optimised to avoid unnecessary read and writes of the FAT table to greatly improve its speed.

Returns	0 if the file is successfully deleted, 1 if there was an error (the file doesn't exist or can't be deleted as its currently open.

Change File Size

```
int ffs_change_file_size (const char *filename, DWORD new_file_size)
```

This function allows you to increase or decrease a files size and is included to allow faster writing in certain situations. When writing a new file every time a sector is completed the driver must read the FAT table to find the next available sector, write to both FAT tables to mark the next sector as now used and then continue with writing the file. When needing to write a large amount of live data quickly this repeated process has a significant effect on write speeds and data buffering requirements. By using this function an application has the possibility to create an oversized file prior to the write starting and then overwriting the file with the data to be stored. As the file is already big enough all the driver has to do as each sector is completed is read the FAT table to find the location of the next sector, removing the need to scan and write to both FAT tables. Once the writing of the file is complete, if the total size of the data is smaller than the file size this function can be used again to reduce the file size.

Returns	0 if the file size was successfully changed, 1 if there was an error (the file doesn't exist or can't be changed as its currently open.

Rename File

```
int ffs_rename (const char *old_filename, const char *new_filename)
```

Return value	0 if the file is successfully renamed, 1 if there was and error (the file doesn't exist or can't be renamed as its currently open)

Clear Error & End Of File Flags

```
void ffs_clearerr (FFS_FILE *file_pointer)
```

Has End Of File Been Reached

```
int ffs_feof (FFS_FILE *file_pointer)
```

Has An Error Occurred During File Access

```
int ffs_ferror (FFS_FILE *file_pointer)
```

Is A Card Inserted And Available

```
BYTE ffs_is_card_available (void)
```

Do Background Tasks

```
void ffs_process (void)
```

This function needs to be called regularly from your applications main loop to detect a new card being inserted so that it can be initialised ready for access.

THE DRIVER SUB FUNCTIONS

These functions are used by the driver but should not be used by your application.

Find File

```
DWORD ffs_find_file (const char *filename, DWORD *file_size, BYTE *attribute_byte,
                  DWORD *directory_entry_sector,
                  BYTE *directory_entry_within_sector,
                  BYTE *read_file_name, BYTE *read_file_extension)
```

This function searches for a specified filename. If wildcard characters are used then the first file that matches with the standard and wildcard characters will be found.

`filename`	Only 8 character DOS compatible root directory filenames are allowed. Format is F.E where F may be between 1 and 8 characters and E may be between 1 and 3 characters, null terminated. The '*' and '?' wildcard characters are allowed.
`*file_size`	Pointer where the file size (bytes) will be written to.
`*attribute_byte`	Pointer where the attribute byte will be written to.
`*directory_entry_sector`	Pointer where the sector number that contains the files directory entry will be written to.
`*directory_entry_within_sector`	Pointer where the file directory entry number within the sector that contains the file will be written to.
`*read_file_name`	Pointer to a 8 character buffer where the filename read from the directory entry will be written to (this may be needed if using this function with wildcard characters)
`*read_file_extension`	Pointer to a 3 character buffer where the filename extension read from the directory entry will be written to (this may be needed if using this function with wildcard characters)
Returns	The file start cluster number (0xFFFFFFFF = file not found)

Convert File Name To Dos Filename

```
BYTE ffs_convert_filename_to_dos (const char *source_filename, BYTE *dos_filename,
                                    BYTE *dos_extension)
```

Used by functions to convert the application supplied filename to a driver specific DOS type filename. The `source_filename` is a case insensitive string with between 1 and 8 filename characters, a period (full stop) character, between 1 and 3 extension characters and a terminating null.

Returns	1 if the filename contained any wildcard characters, 0 if not (this allow calling functions to detect invalid names if they are creating a new file)

Read Next Directory Entry

```
BYTE ffs_read_next_directory_entry (BYTE *file_name, BYTE *file_extension,
                                    BYTE *attribute_byte, DWORD *file_size,
                                    DWORD *cluster_number,
                                    BYTE start_from_beginning,
                                    DWORD *directory_entry_sector,
                                    BYTE *directory_entry_within_sector)
```

`*file_name`	Pointer where the 8 character array filename will be written to.
`*file_extension`	Pointer where the 3 character array filename extension will be written to.
`*attribute_byte`	Pointer where the file attribute byte will be written to.
`*file_size`	Pointer where the file size will be written to.
`*cluster_number`	Pointer where the start cluster for the file will be written to.
`start_from_beginning`	Set to cause routine to start from 1st directory entry (this must be set if the drivers data buffer has been modified since the last call)
`*directory_entry_sector`	Pointer where the sector number that contains the files directory entry will be written to.
`*directory_entry_within_sector`	Pointer where the file directory entry number within the sector that contains the file will be written to.
Returns	1 if a file entry was found, 0 if not (marks the end of the directory

Overwrite The Last Directory File Name

```
void ffs_overwrite_last_directory_entry (BYTE *file_name, BYTE *file_extension,
                                          BYTE *attribute_byte, DWORD *file_size
                                          DWORD *cluster_number)
```

`*file_name`	Pointer to an 8 character filename (must be DOS compatible - uppercase and any trailing unused characters set to 0x20)
`*file_extension`	Pointer to 3 character filename extension (must be DOS compatible - uppercase and any trailing unused characters set to 0x20)
`*attribute_byte`	Pointer to the file attribute byte
`*file_size`	Pointer to the file size
`*cluster_number`	Pointer to the start cluster number for the file

Get The Start Cluster Number For A File

```
DWORD get_file_start_cluster(FFS_FILE *file_pointer)
```

Returns the cluster number of the start of the file. Further cluster numbers are read from the FAT table.

Create A New File

```
BYTE ffs_create_new_file (const char *file_name, DWORD *write_file_start_cluster,
                          DWORD *directory_entry_sector,
                          BYTE *directory_entry_within_sector)
```

*file_name	Pointer to an 8 character filename
*write_file_start_cluster	The cluster number that contains the start of the file.
`*directory_entry_sector`	Pointer where the sector number that contains the files directory entry will be written to.
`*directory_entry_within_sector`	Pointer where the file directory entry number within the sector that contains the file will be written to.
Return value	1 if successful, 0 if not

Find Next Free Cluster In FAT Table

```
DWORD ffs_get_next_free_cluster (void)
```

Find the next available free cluster from the FAT table. The last found free cluster number is stored to help speed up successive calls to this function.

Returns	The cluster number, or 0xFFFFFFFF if no free cluster found (card is full)

Get Next Cluster Value From FAT Table

```
DWORD ffs_get_next_cluster_no (DWORD current_cluster)
```

This function looks up the `current_cluster` number in the FAT table and returns the FAT table entry which will be the next cluster number or the end of file marker.

Modify Cluster Value In FAT Table

```
void ffs_modify_cluster_entry_in_fat (DWORD cluster_to_modify,
                                      DWORD cluster_entry_new_value)
```

The `cluster_to_modify` FAT table entry is overwritten with `cluster_entry_new_value`.

Read Sector To Buffer

```
void ffs_read_sector_to_buffer (DWORD sector_lba)
```

Reads a sector of data (usually 512 bytes) to the microcontroller / processor ram buffer.

`sector_lba`	The ‘Logical Block Address’ / sector number to read.

Write Sector From Buffer

```
void ffs_write_sector_from_buffer (DWORD sector_lba)
```

Write a sector of data (usually 512 bytes) from the microcontroller / processor ram buffer.

`sector_lba`	The ‘Logical Block Address’ / sector number to read.

Is Card Present

```
BYTE ffs_is_card_present (void)
```

Returns	1 if present, 0 if not

Write Byte To Card

```
BYTE ffs_write_byte (BYTE data)
```

Read Word From Card

```
WORD ffs_read_word (void)
```

Read Byte From Card

```
BYTE ffs_read_byte (void)
```

LAYOUT OF A MMC OR SD CARD WITH FAT

Note – this section of the manual is for information only. You do not need to read and understand this large and in depth section to use the driver! However you may want to if you wish to gain an understanding of disk access, the FAT filing system and how this driver works.

Terms used for hard disks and therefore MMC / SD memory cards

Remember when understanding these terms that hard disks uses multiple disks of magnetic material with a read/write head for each side of each disk. Bytes are read from and written to a disks surface in circular paths.

Track

The circular track on one surface of a disk (numbered 0 - #). This is not usually referred to.

Cylinder

All of the tracks in the same position on all of the surfaces (numbered 0 - #). This is not usually referred to other than when determining the parameters of a disk during initialisation.

Head

Each side of a disk has a read / write head (numbered 0 - #). This is not usually referred to other than when determining the parameters of a disk during initialisation.

Sector

This is the fundamental unit of disk mapping - all reading and writing to disks is carried out in sectors. A sector is usually 512 bytes in size, but can be 128 – 1024 bytes.
(Numbered as 1 - # (0 is reserved for identification purposes)).

Cluster

A cluster is a specified group of sectors. It is clusters that are the addressing unit when reading and writing files using the FAT system (i.e. a directory will point to a particular file using the cluster number that contains the start of the file). A cluster may only be used by one file, and large files will use multiple clusters to hold their data. A disk with a large cluster size (lots of sectors per cluster) will mean that disk space is wasted as any unused bytes after the end of a file in its final cluster will not be available for anything else. A disk with a small cluster size means less wastage. However, a small cluster size means a larger FAT table as a FAT table contains an entry for every cluster on a disk (or in the partition if the disk is partitioned), hence the need to FAT32 instead of FAT16 for larger volumes.

The valid range is 1 – 64 sectors per cluster. The first cluster that may be used is number 2 (clusters 0 & 1 are reserved).

The FAT filing system was developed for DOS and DOS thinks of a disk as a linear object, not as it is actually constructed. This means that DOS treats the sectors of a disk as a sequential list of sectors, from the first on the disk to the last. Whilst this made things more complex when writing drivers for hard disks, it makes things easier when dealing with modern flash memory cards as these are linear memory objects.

Byte Ordering

The FAT file system uses ‘little endian’. That is that the first byte read is the least significant byte of a large value, the next byte read is more significant than the last and so on. For example this is how a 32bit value would be stored (with the bit numbers shown):-

```
        byte[3] 3 3 2 2 2 2 2 2                //This is the last byte read from the disk
                1 0 9 8 7 6 5 4

        byte[2] 2 2 2 2 1 1 1 1
                3 2 1 0 9 8 7 6

        byte[1] 1 1 1 1 1 1 0 0
                5 4 3 2 1 0 9 8

        byte[0] 0 0 0 0 0 0 0 0                //This is the first byte read from the disk
                7 6 5 4 3 2 1 0
```

The following sections show how the different sections of a disk are organised for FAT16 and FAT32, looking at the disk as a linear memory object (which is how it is addressed). See the following sections for an in depth description of each block.

The Layout of a FAT16 Volume

Start Address	Size	Contents	
0x00000000	512 bytes	Master Boot Record (Amongst other things this specifies the address of each of the main partitions).	
Partition Start Address + 0	512 bytes	Partition 1	The Boot Record. Located in the first sector of a partition.
Partition Start Address + 512	As specified in the Boot Record		FAT table 1
Partition Start Address + 512 + (Size of FAT Table x (FAT table # - 1))	As specified in the Boot Record		FAT table # (specified by 'Number of Copies of FAT' in master boot record. A value of 2 is normal)
Partition Start Address + 512 + (Size of FAT Table x Number of Copies of FAT)	As specified in the Boot Record		Root directory
Partition Start Address + 512 + (Size of FAT Table x Number of Copies of FAT) + Size of Root Directory	Calculated from the Master Boot Record Total Partition Size		Data area for files and other directories. (This area occupies the remainder of the disk, or the space to the start of the next partition).

Then follows further partitions if present:-

Start Address	Size	Contents	
Partition Start Address + 0	512 bytes	Partition 2	The Boot Record. Located in the first sector of a partition.
Partition Start Address + 512	As specified in the Boot Record		FAT table 1
Partition Start Address + 512 + (Size of FAT Table x (FAT table # - 1))	As specified in the Boot Record		FAT table # (specified by 'Number of Copies of FAT' in master boot record. A value of 2 is normal)
Partition Start Address + 512 + (Size of FAT Table x Number of Copies of FAT)	As specified in the Boot Record		Root directory
Partition Start Address + 512 + (Size of FAT Table x Number of Copies of FAT) + Size of Root Directory	Calculated from the Master Boot Record Total Partition Size		Data area for files and other directories. (This area occupies the remainder of the disk, or the space to the start of the next partition).

Repeated for each partition

Note - Shaded cells may repeat or not be present at all.

The Layout of a FAT32 Volume

This is basically the same as for a FAT16 volume, but without the root directory included (and with each block using a different amount of space).

Start Address	Size	Contents	
0x00000000	512 bytes	Master Boot Record (Amongst other things this specifies the address of each of the main partitions).	
Partition Start Address + 0	512 bytes	Partition 1	The Boot Record. Located in the first sector of a partition.
Partition Start Address + 512	As specified in the Boot Record		FAT table 1
Partition Start Address + 512 + (Size of FAT Table x (FAT table # - 1))	As specified in the Boot Record		FAT table # (specified by 'Number of Copies of FAT' in master boot record. A value of 2 is normal)
Partition Start Address + 512 + (Size of FAT Table x Number of Copies of FAT)	Calculated from the Master Boot Record Total Partition Size		Data area for files and other directories. (This area occupies the remainder of the disk, or the space to the start of the next partition).

Then if there is more than 1 partition, the additional partitions follow:-

Start Address	Size	Contents	
Partition Start Address + 0	512 bytes	Partition 2	The Boot Record. Located in the first sector of a partition.
Partition Start Address + 512	As specified in the Boot Record		FAT table 1
Partition Start Address + 512 + (Size of FAT Table x (FAT table # - 1))	As specified in the Boot Record		FAT table # (specified by 'Number of Copies of FAT' in master boot record. A value of 2 is normal)
Partition Start Address + 512 + (Size of FAT Table x Number of Copies of FAT)	Calculated from the Master Boot Record Total Partition Size		Data area for files and other directories. (This area occupies the remainder of the disk, or the space to the start of the next partition).

Repeated for each partition

Note - Shaded cells may repeat or not be present at all.

THE MASTER BOOT RECORD

The first sector of a hard disk is set aside for the Master Boot Record. This is operating system independent. It is located on the first Sector of the disk, at Cylinder 0, Head 0, Sector 1. It contains the partition table, which defines the different sections of your hard drive and if this section of a disk is corrupted it can mean that the disk is dead!

Note – if trying to view the master boot record using PC disk viewing software ensure that you have selected the correct section of the disk. Some software will show you the contents of the first partition by default, not the first sector containing the master boot record.

Byte (0x00000000 + #)		**Value**		
0	0x0000	446 bytes of boot up executable code and data.		
‖	‖			
445	0x01BD			
446	0x01BE	Partition 1	Offset 0x00	Current State of Partition (00h=Inactive, 80h=Active)
447	0x01BF		Offset 0x01	Beginning of Partition – Head
448	0x01C0		Offset 0x02	Beginning of Partition – Cylinder/Sector
449	0x01C1		Offset 0x03	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Cylinder Bits 7 to 0 (bits 15–8); Cylinder Bits 9+8 (bits 7–6); Sector Bits 5 to 0 (bits 5–0)
450	0x01C2		Offset 0x04	Type of Partition: 0x00 Unknown or Nothing 0x01 12-bit FAT 0x04 16-bit FAT (Partition Smaller than 32MB) 0x05 Extended MS-DOS Partition 0x06 16-bit FAT (Partition Larger than 32MB) 0x0B 32-bit FAT (Partition Up to 2048GB) 0x0C Same as 0x0B, but uses LBA 0x13 extensions 0x0E Same as 0x06, but uses LBA 0x13 extensions 0x0F Same as 0x05, but uses LBA 0x13 extensions The above values relate to Microsoft operating systems – there are others. LBA = Logical Block Addressing which uses the Int 0x13 extensions built into newer BIOS's to access data above the 8GB barrier, or to access strictly in LBA mode, instead of CHS (Cylinder, Head, Sector).
451	0x01C3		Offset 0x05	End of Partition – Head
452	0x01C4		Offset 0x06	End of Partition – Cylinder/Sector
453	0x01C5		Offset 0x07	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Cylinder Bits 7 to 0 (bits 15–8); Cylinder Bits 9+8 (bits 7–6); Sector Bits 5 to 0 (bits 5–0)
454	0x01C6		Offset 0x08	Number of sectors between the master boot record and the first sector in the partition.
455	0x01C7		Offset 0x09	
456	0x01C8		Offset 0x0A	
457	0x01C9		Offset 0x0B	
458	0x01CA		Offset 0x0C	Number of sectors in the partition
459	0x01CB		Offset 0x0D	
460	0x01CC		Offset 0x0E	
461	0x01CD		Offset 0x0F	

462	0x01CE	Partition 2	Offset 0x00	Current State of Partition (00h=Inactive, 80h=Active)
463	0x01CF		Offset 0x01	Beginning of Partition - Head
464	0x01D0		Offset 0x02	Beginning of Partition - Cylinder/Sector (Format as per partition 1)
465	0x01D1		Offset 0x03	
466	0x01D2		Offset 0x04	Type of Partition (Format as per partition 1)
467	0x01D3		Offset 0x05	End of Partition - Head
468	0x01D4		Offset 0x06	End of Partition - Cylinder/Sector (Format as per partition 1)
469	0x01D5		Offset 0x07	
470	0x01D6		Offset 0x08	Number of sectors between the master boot record and the first sector in the partition.
471	0x01D7		Offset 0x09	
472	0x01D8		Offset 0x0A	
473	0x01D9		Offset 0x0B	
474	0x01DA		Offset 0x0C	Number of sectors in the partition
475	0x01DB		Offset 0x0D	
476	0x01DC		Offset 0x0E	
477	0x01DD		Offset 0x0F	
478	0x01DE	Partition 3	Offset 0x00	Current State of Partition (00h=Inactive, 80h=Active)
479	0x01DF		Offset 0x01	Beginning of Partition - Head
480	0x01E0		Offset 0x02	Beginning of Partition - Cylinder/Sector (Format as per partition 1)
481	0x01E1		Offset 0x03	
482	0x01E2		Offset 0x04	Type of Partition (Format as per partition 1)
483	0x01E3		Offset 0x05	End of Partition - Head
484	0x01E4		Offset 0x06	End of Partition - Cylinder/Sector (Format as per partition 1)
485	0x01E5		Offset 0x07	
486	0x01E6		Offset 0x08	Number of sectors between the master boot record and the first sector in the partition.
487	0x01E7		Offset 0x09	
488	0x01E8		Offset 0x0A	
489	0x01E9		Offset 0x0B	
490	0x01EA		Offset 0x0C	Number of sectors in the partition
491	0x01EB		Offset 0x0D	
492	0x01EC		Offset 0x0E	
493	0x01ED		Offset 0x0F	
494	0x01EE	Partition 4	Offset 0x00	Current State of Partition (00h=Inactive, 80h=Active)
495	0x01EF		Offset 0x01	Beginning of Partition - Head
496	0x01F0		Offset 0x02	Beginning of Partition - Cylinder/Sector (Format as per partition 1)
497	0x01F1		Offset 0x03	
498	0x01F2		Offset 0x04	Type of Partition (Format as per partition 1)
499	0x01F3		Offset 0x05	End of Partition - Head
500	0x01F4		Offset 0x06	End of Partition - Cylinder/Sector (Format as per partition 1)
501	0x01F5		Offset 0x07	
502	0x01F6		Offset 0x08	Number of sectors between the master boot record and the first sector in the partition.
503	0x01F7		Offset 0x09	
504	0x01F8		Offset 0x0A	
505	0x01F9		Offset 0x0B	
506	0x01FA		Offset 0x0C	Number of sectors in the partition
507	0x01FB		Offset 0x0D	
508	0x01FC		Offset 0x0E	
509	0x01FD		Offset 0x0F	
510	0x01FE	Boot signature (= 0xAA55)		
511	0x01FF			

THE BOOT RECORD

The first sector of a partition contains a boot record. There are differences between the FAT16 and FAT32 boot records.

(Greyed out FAT32 entries indicate that the contents is the same as for FAT16)

FAT16

Offset	Description
0x0000	Jump Code + NOP
0x0001	
0x0002	
0x0003	8 byte OEM Name
||	
0x000A	
0x000B	Bytes Per Sector
0x000C	
0x000D	Sectors Per Cluster (Restricted to powers of 2 (1, 2, 4, 8, 16, 32…))
0x000E	Reserved Sectors
0x000F	
0x0010	Number of Copies of FAT. (A value of 2 is recommended – values other than 2 are possible by are not recommended by Microsoft)
0x0011	Maximum Root Directory Entries
0x0012	
0x0013	Number of Sectors in Partition Smaller than 32MB
0x0014	
0x0015	Media Descriptor (F8h for Hard Disks)
0x0016	Sectors Per FAT
0x0017	
0x0018	Sectors Per Track
0x0019	
0x001A	Number of Heads
0x001B	
0x001C	Number of Hidden Sectors in Partition
0x001D	
0x001E	
0x001F	
0x0020	Number of Sectors in Partition
0x0021	
0x0022	
0x0023	

FAT32

Offset	Description
0x0000	Jump Code + NOP
0x0001	
0x0002	
0x0003	8 byte OEM Name
||	
0x000A	
0x000B	Bytes Per Sector
0x000C	
0x000D	Sectors Per Cluster (Restricted to powers of 2 (1, 2, 4, 8, 16, 32…))
0x000E	Reserved Sectors
0x000F	
0x0010	Number of Copies of FAT. (A value of 2 is recommended – values other than 2 are possible by are not recommended by Microsoft)
0x0011	Maximum Root Directory Entries (not applicable for FAT32)
0x0012	
0x0013	Number of Sectors in Partition Smaller than 32MB (not applicable for FAT32)
0x0014	
0x0015	Media Descriptor (F8h for Hard Disks)
0x0016	Sectors Per FAT (not applicable for FAT32 – bigger field below)
0x0017	
0x0018	Sectors Per Track
0x0019	
0x001A	Number of Heads
0x001B	
0x001C	Number of Hidden Sectors in Partition
0x001D	
0x001E	
0x001F	
0x0020	Number of Sectors in Partition
0x0021	
0x0022	
0x0023	

From this point the boot records are not the same – continued on next page...

0x0024	Logical Drive Number of Partition
0x0025	
0x0026	Extended Signature (29h)
0x0027	Serial Number of Partition
0x0028	
0x0029	
0x002A	
0x002B	11 bytes of volume name of the partition
||	
0x0035	
0x0036	FAT Name (FAT16)
0x003E	448 bytes of executable code and data
||	
0x01FD	
0x01FE	Boot signature (= 0xAA55)
0x01FF	

0x0024	Number of Sectors Per FAT
0x0025	
0x0026	
0x0027	
0x0028	Flags:
0x0029	15:8 Reserved Bit 7 1 = FAT Mirroring is Disabled, only 1 FAT is active as specified in bits 3:0 0 = FAT Mirroring is Enabled into all FATs 6:4 Reserved Bits 3:0 Number of active FAT (0-#). Only valid if mirroring disabled.
0x002A	Version of FAT32 Drive (high byte = major version, low byte = minor version)
0x002B	
0x002C	Cluster Number of the Start of the Root Directory (Usually 2, but not required to be)
0x002D	
0x002E	
0x002F	
0x0030	Sector Number of the File System Information Sector (Referenced from the start of the partition)
0x0031	
0x0032	Sector Number of the Backup Boot Sector (Referenced from the start of the partition)
0x0033	
0x0034	Reserved (12 bytes)
||	
0x003F	
0x0040	Logical Drive Number of Partition
0x0041	Unused
0x0042	Extended Signature (29h)
0x0043	Serial Number of Partition
0x0044	
0x0045	
0x0046	
0x0047	11 byte volume name of the partition
||	
0x0051	
0x0052	8 byte FAT Name (FAT32)
||	
0x0059	
0x005A	420 bytes of executable code and data
||	
0x01FD	
0x01FE	Boot Signature (= 0xAA55)
0x01FF	

THE FAT TABLES

The FAT table (whether FAT16 or FAT32) contains an entry for every cluster on the disk (or partition if the disk is partitioned). Each entry is either 16 bits in size for FAT16, or 32bits in size for FAT32. The contents of an entry may be as follows:-

FAT16 Table Entry Values:-

0x0000	The cluster is free.
0x0001	Reserved
0x0002 – 0xFFF0	This cluster is used. The value indicates the next cluster number for the file.
0xFFF7	Cluster is bad
0xFFF8 – 0xFFFF	EOC (End Of Clusterchain) (typically you should use 0xFFFF)

FAT32 Table Entry Values:-

0x#0000000	The cluster is free.
0x0001	Reserved
0x0002 – 0xFFF0	This cluster is used. The value indicates the next cluster number for the file.
0x#FFFFFF7	Cluster is bad
0x#FFFFFF8 – 0x#FFFFFFF	EOC (End Of Clusterchain) (typically you should use 0x#FFFFFFF

(The top 4 bits are reserved and will not necessarily be zero. They must be ignored when reading a cluster number but maintained when writing a new value to an entry)

When a file is stored the first available free cluster is found from the FAT table and stored in the files directory entry (see later in this manual). The file is written to the cluster. If it doesn't fit within the cluster then the next free cluster is found and the new cluster number is written in the previous clusters FAT table entry. This continues until the last cluster that is required for the file (which may be the first cluster if the file will fit within one cluster). The EOC marker is written to the FAT tables for the last cluster to indicate that no further clusters are used.

Therefore when reading a file the start cluster number is determined from the files entry in the directory the file is located in. Then the FAT table is used to find the next cluster that holds the next block of the files data, then the next etc. Whilst the EOC marker indicates that a cluster is the last cluster used to store a file, the exact file size is stored in the files directory entry so that the last used byte number of the file can be determined.

FAT16 FAT Table

Byte (Partition Start Address + 512 + #)		**FAT Entry**	**Value**
0	0x0000	1	Reserved. Contains the media type value in the low 8 bits and all other bits are set to 1
1	0x0001		
2	0x0002	2	Reserved – set on format to the EOC marker. The top 2 bits may be used as 'dirty volume' flags: Bit 15 1 = volume is 'clean'. 0 = volume is 'dirty' (the file system driver did not complete its last task properly and it would be good idea to run a disk checking program. Bit 14 1 =no disk read/write errors were encountered. 0 = the file system driver encountered a disk I/O error on the volume the last time it was used, which indicates that some sectors may have gone bad on the volume. It would be a good idea to run a disk checking program.
3	0x0003		
4	0x0004	3	The FAT entry for the 1st cluster in the data area of the disk / partition
5	0x0005		
6	0x0006	4	The FAT entry for the 2nd cluster in the data area of the disk / partition
7	0x0007		
||	||		

#	0x####	#	The FAT entry for the last cluster in the data area of the disk / partition
#	0x####		

FAT32 FAT Table

Byte (Partition Start Address + 512 + #)		FAT Entry	Value
0	0x0000	1	Reserved. Contains the media type value in the low 8 bits and all other bits are set to 1
1	0x0001		
2	0x0002		
3	0x0003		
4	0x0004	2	Reserved – set on format to the EOC marker. The top 2 bits may be used as 'dirty volume' flags: Bit 27 1 = volume is 'clean'. 0 = volume is 'dirty' (the file system driver did not complete its last task properly and it would be good idea to run a disk checking program. Bit 26 1 =no disk read/write errors were encountered. 0 = the file system driver encountered a disk I/O error on the volume the last time it was used, which indicates that some sectors may have gone bad on the volume. It would be a good idea to run a disk checking program.
5	0x0005		
6	0x0006		
7	0x0007		
8	0x0008	3	The FAT entry for the 1st cluster in the data area of the disk / partition
9	0x0009		
10	0x000A		
11	0x000B		
12	0x000C	4	The FAT entry for the 2nd cluster in the data area of the disk / partition
13	0x000D		
14	0x000E		
15	0x000F		
||	||		
#	0x####	#	The FAT entry for the last cluster in the data area of the disk / partition
#	0x####		
#	0x####		
#	0x####		

FAT16 uses 2 FAT tables, one after the other, and FAT32 uses up to 4 FAT tables. This provides a backup in case of corruption of one of the tables. If you change the contents of the FAT table, ensure that all copies are updated (checking for FAT32 to see which tables should be updated).

Location & Size

The first FAT table starts straight after the Boot Record. Therefore the start address of the first FAT table:
= Start address of partition + No of reserved sectors

Each additional FAT table follows straight on after the last. The number of FAT tables is recommended to be 2 due to old systems that assume a value of 2. However the number of FAT tables does not have to be 2 and for flash drives where a backup of the FAT table is redundant only a single table may be used. It is also possible to have more than 2 FAT tables.

Root Directory & Other Directories

A FAT directory is simply a 'file' containing a linear list of 32 byte entries. The only special directory, which must always be present, is the root directory. For FAT16 volumes the root directory is located in a fixed location on the disk immediately following the last FAT and is a fixed size in sectors as specified in the Boot Record.

For FAT16 the first sector of the root directory is sector number relative to the first sector of the FAT volume:

For FAT32 the root directory can be of variable size and is a cluster chain just like any other directory. The first cluster of the root directory is specified in the Boot Record.

Each directory entry is 32 bytes and formatted as follows:

Byte		Value
0	0x00	Name
1	0x01	(The 8 character filename)
2	0x02	
3	0x03	
4	0x04	
5	0x05	
6	0x06	
7	0x07	
8	0x08	Extension
9	0x09	(The 3 character filename extension)
10	0x0A	
11	0x0B	Attributes Bit: 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 Value: 0 | 0 | Archive | Directory | Volume Label | System | Hidden | Read Only
12	0x0C	NT (Reserved for WindowsNT always 0)
13	0x0D	Created time – mS (0 if not used)
14	0x0E	Created time - hour and minute (0 if not used)
15	0x0F	
16	0x10	Created date (0 if not used)
17	0x11	
18	0x12	Last accessed date (0 if not used)
19	0x13	
20	0x14	Extended Attribute
21	0x15	(reserved for OS/2, always 0) High word of cluster for FAT32 volumes
22	0x16	Time of last write to file
23	0x17	
24	0x18	Date of last write to file
25	0x19	
26	0x1A	Start cluster (referenced from the start of the data area of the volume)
27	0x1B	
28	0x1C	File size
29	0x1D	
30	0x1E	
31	0x1F	

(Shaded bytes we're unused in the original DOS specification and may still be left unused if desired)

Special Markers

If the first byte of a directory entry is 0xE5 then the entry has been erased. If the first byte is 0x00 then the entry has never been used (this can be used to detect the end of the table as all following entries will also be 0x00).

Location & Size

For FAT16 the root directory is located directly after the 2nd FAT table:

= Start address of partition + No of reserved sectors + (Number of FAT tables x FAT table size)

Its size is specified by the boot record:

= maximum number of root directory entries x 32 bytes per entry

The data area starts straight after the root directory. The only difference between the root folder and any other folders is that the root folder is at a specified location and has a fixed number of entries.

For FAT32 the root directory can be of variable size and is a cluster chain, just like any other directory is. The first cluster of the root directory on a FAT32 volume is stored in the sector specified in the boot record.

For both FAT16 and FAT23, unlike other directories, the root directory itself does not have any date or time stamps, does not have a file name (other than the implied file name "\"), and does not contain "." and ".." files as the first two directory entries in the directory. The only other special aspect of the root directory is that it is the only directory on the FAT volume for which it is valid to have a file that has only the 'Volume ID' attribute bit set.

Date and Time Formats

If date and time are not supported then they should be written as zero. Bytes 22 – 25, time of last write and date of last write, must be supported according to the FAT specification but if a device has no real time clock then this isn't possible.

Date field

A 16-bit field that is a date relative to 01/01/1980:-

Bits 15:9	Count of years from 1980, valid range 0 – 127 (=1980–2107).
Bits 8:5	Month of year, valid range 1–12 (1 = January)
Bits 4:0	Day of month, valid range 1-31

Time Format.

A 16-bit field with a valid range from Midnight 00:00:00 to 23:59:58:-

Bits 15:11	Hours, valid range 0 – 23
Bits 10:5	Minutes, valid range 0 – 59
Bits 4:0	2-second count, valid range 0–29 (= 0 – 58 seconds)

DATA AREA

The remainder of the volume is the data area, which may contain files and directories. It is this area that the FAT tables relate to.

Start Address

For FAT16 the start address of the data area is:-

Start address of partition + Number of reserved sectors + (Number of FAT tables x FAT table size) + Number of root directory sectors

For FAT32 the start address of the data area is:-

Start address of partition + Number of reserved sectors + (Number of FAT tables x FAT table size)

For a given cluster number in the FAT table, the start address of that sector is:-

data area start address + ((FAT table cluster number – 2) x sectors per cluster)

Because sectors per cluster is restricted to powers of 2 (1, 2, 4, 8, 16, 32…), division and multiplication by sectors per cluster can actually be performed via shift operations which is often faster than multiply or divide instructions

FAT32 FILE SYSTEM INFORMATION SECTOR

(Not applicable to FAT16)

The partition boot record specifies the sector that contains this information block, which can be utilised by a FAT driver to speed up write operations.

Byte (Sector Start + #)		Value
0	0x0000	Signature = 0x41615252. This validates that this is a File System Information Sector.
1	0x0001	
2	0x0002	
3	0x0003	
4	0x0004	480 reserved bytes
||	||	
483	0x01E3	
484	0x01E4	Signature = 0x61417272. Another signature that is more localized in the sector to the location of the fields that are used.
485	0x01E5	
486	0x01E6	
487	0x01E7	
488	0x01E8	Number of Free Clusters on the volume. Set to 0xFFFFFFFF if unknown and needs computing. This should be range checked at least to make sure it is <= volume cluster count.
489	0x01E9	
490	0x01EA	
491	0x01EB	
492	0x01EC	It indicates the cluster number at which the driver should start looking for free clusters – it is a hint for the FAT driver. Because a FAT32 FAT is large, it can be rather time consuming if there are a lot of allocated clusters at the start of the FAT and the driver starts looking for a free cluster starting at cluster 2. Typically this value is set to the last cluster number that the driver allocated. If the value is 0xFFFFFFFF, then there is no hint and the driver should start looking at cluster 2. Any other value can be used, but should be checked first to make sure it is a valid cluster number for the volume.
493	0x01ED	
494	0x01EE	
495	0x01EF	
496	0x01F0	12 reserved bytes
||	||	
507	0x01FB	
508	0x01FC	Trailing signature = 0x000055AA Used to validate that this is a File System Information Sector.
509	0x01FD	
510	0x01FE	
511	0x01FF	

TROUBLESHOOTING

If you are experiencing problems using the driver in your project the following tips may help:-

Double check IO pin definitions in the driver header file.

Verify with a scope that all of the control and data pins to the MMC or SD card are working correctly.

Check that no other device on the SPI bus is outputting while the driver is trying to communicate with the MMC or SD card.

Single step through the initialise new card part of the ffs_process function. There are several points at which the driver verifies the correct value is returned by the MMC or SD card and if the correct value is not being returned this may point to the cause of a problem.

Try using a different MMC or SD card made by a different manufacturer. We have occasionally come across faulty cards or cards that do not properly conform to the MMC or SD standard, even from reputable manufacturers.

Check that your microcontroller is not resetting due to a watchdog timer timeout. Read and write operations to MMC or SD cards can sometimes take time to complete that may exceed your watchdog timer setting?

If you are using a write protect input (FFS_WP_PIN_REGISTER is defined in mem-mmcsd.h) check that it is not configured incorrectly and blocking write operations.

See the:
'Signal Noise Issues With MMC & SD Memory Cards (& Clocked Devices In General)
page in the resources area of our web site for details of a common signal noise problem experienced when using MMC and SD memory cards.

Check that you have enough stack space allocated. This driver uses a moderate amount of ram from the stack and if your application is already using large amounts of the stack before calling driver functions this may be causing a stack overrun?

If you are using a 32bit device ensure that for the driver files WORD = 16 bits and DWORD = 32 bits.

SUPPORT

Please visit the support section of the embedded-code.com web site if you have any queries regarding this driver. Please note that our support covers the use of this driver with the reference designs in this manual. Where possible we will try to help solve any problems if the code is used with other devices or compilers, but given the huge number of devices and compilers available we are unable to guarantee 'out of the box' compatibility. If you plan to use the source code with a different processor, microcontroller and/or compiler you should ensure that you have sufficient programming expertise to carry out any modifications that may be required to the source code.

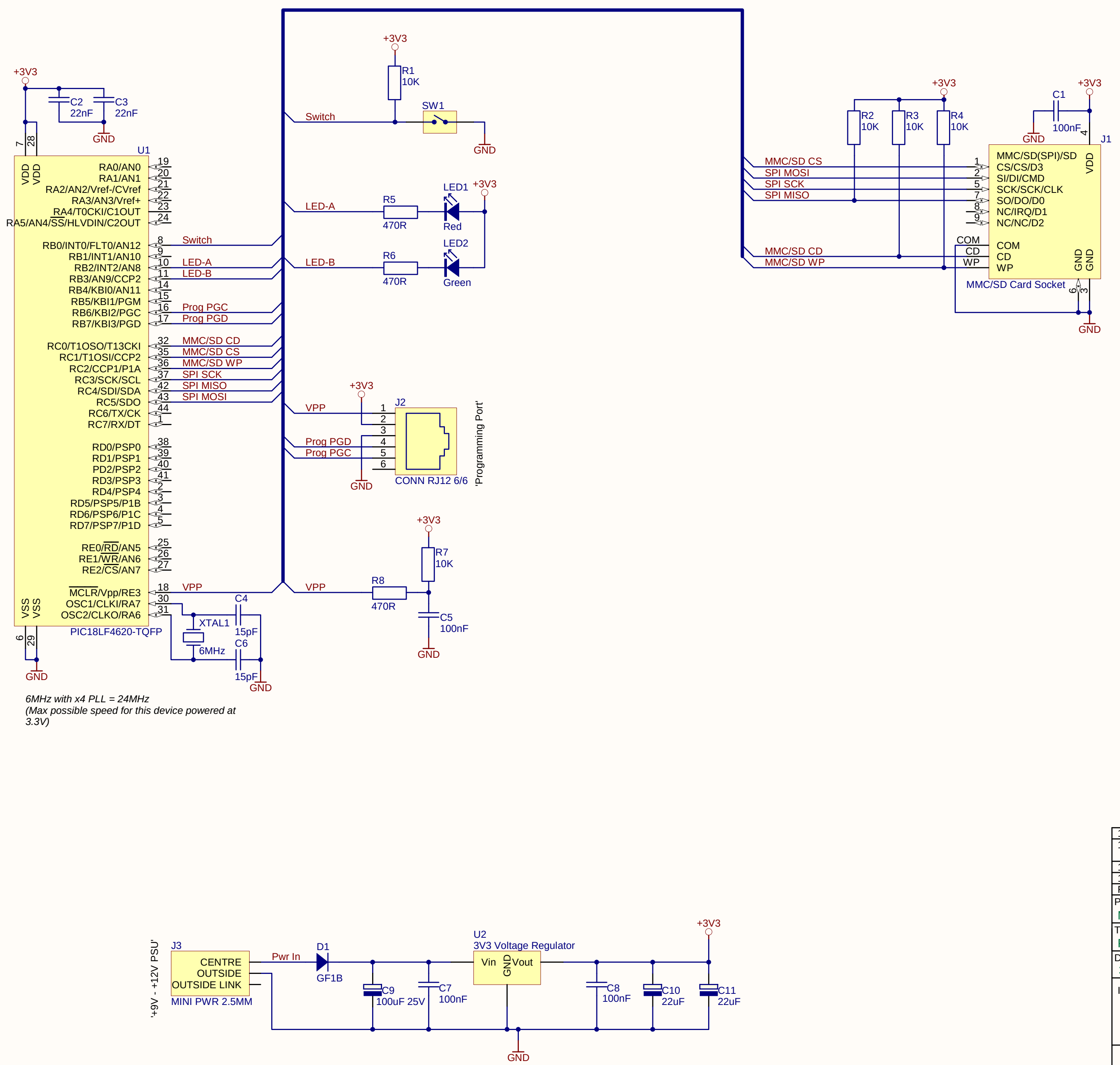
If you do encounter issues using the driver with other compilers or devices and are able to give us details of the issue you encountered we will try and include changes or notes across our range of drivers to help other programmers avoid similar issues in the future. Please use the contact us page of our web site to report any such issues discovered.

Revision History

See driver revision history file

embedded-code.com

Web: www.embedded-code.com



Rev.	Notes
1.03	Added LPC2365 circuit.
1.02	Changed PIC 18 to PIC18LF4620 and added PIC24HJ64GP206 circuit.
1.01	Moved 'MMC/SD CD' from PIC pin RC0 to RC2 to match code.
1.00	Original release

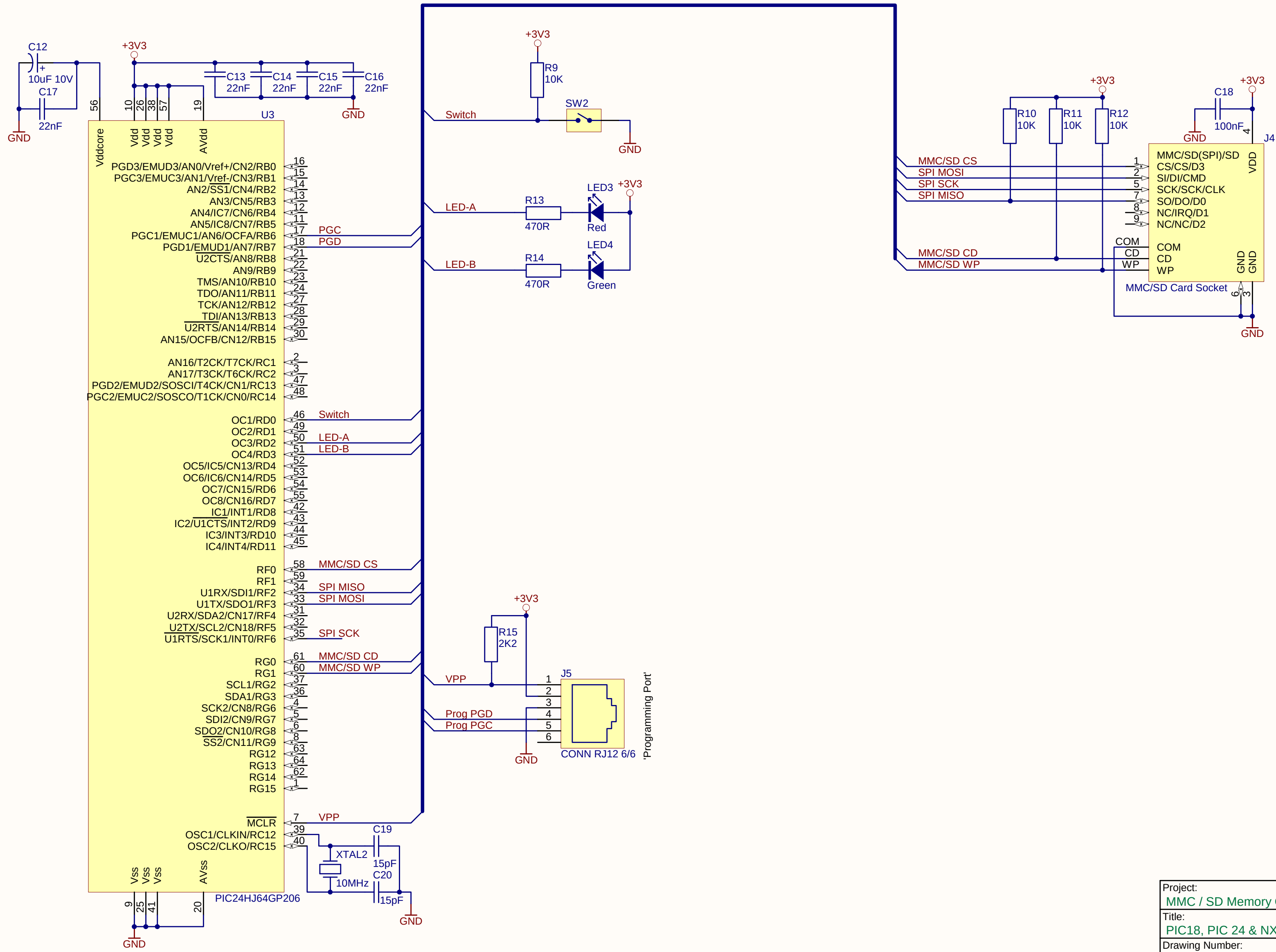
Project:
MMC / SD Memory Card Driver

Title:
PIC18, PIC 24 & NXP LPC23xx Sample Project Circuits

Drawing Number: 3563-005
Rev: 1.03
Sheet: 1 of 3

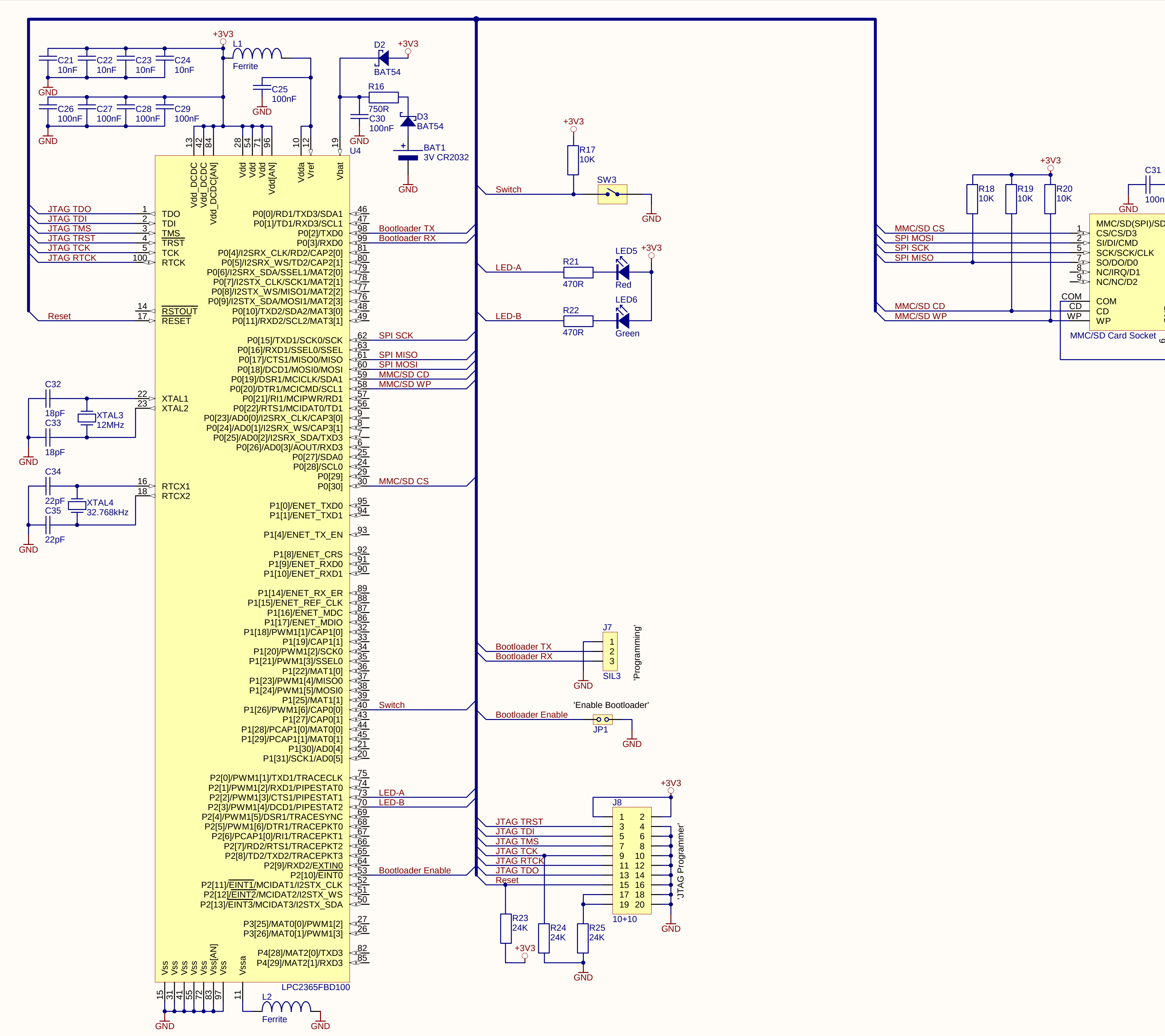
embedded-code.com

1	2	3	4	5	6	7	8



Project:		
MMC / SD Memory Card Driver		
Title:		
PIC18, PIC 24 & NXP LPC23xx Sample Project Circuits		
Drawing Number: 3563-005	Rev: 1.03	Sheet: 2 of 3

embedded-code.com



Project: MMC / SD Memory Card Driver		
Title: PIC18, PIC 24 & NXP LPC23xx Sample Project Circuits		
Drawing Number: 3563-005	Rev: 1.03	Sheet: 3 of 3

embedded-code.com