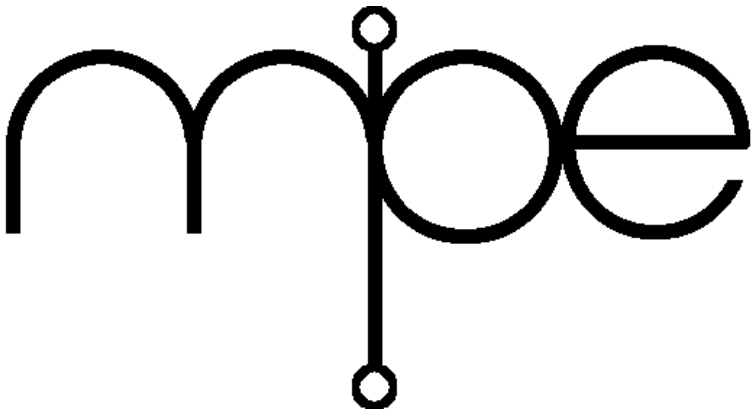
FAT filing system

v2.3

Brad Eckert, Stephen Pelc



Published by Microprocessor Engineering

FAT filing system
User manual
Manual revision 2.3
16 March 2013

Software
Software version 2.3

For technical support
Please contact your supplier

For further information
MicroProcessor Engineering Limited
133 Hill Lane
Southampton SO15 5AF
UK

Tel: +44 (0)23 8063 1441
Fax: +44 (0)23 8033 9691
e-mail: mpe@mpeforth.com
tech-support@mpeforth.com
web: www.mpeforth.com

Table of Contents

1 FAT file system introduction

The file *FatCore.fth* contains the core FAT filing system. before you compile this file you must provide some configuration information, a prortability harness for your Forth kernel, and a sector read/write interface for your disk system. Some example interfaces are provided in the issue code.

To build the code, there are example build files, including amongst others:

`FatVfxW.bld`
: Builds a demonstration system on VFX Forth for Windows. This can be used with any drive formatted with a FAT file system.

`SSP2148.bld`
: Builds code for an Olimex LPC-P2148 board using the SSP (SPI1) driver from the ARM cross compiler.

`mci23xx.bld`
: Builds code for an NXP device using the MCI controller (4 bit SD card) on LPC23xx/24xx and some LPC17xx devices.

`qspi52259.bld`
: Builds code for the QSPI device on Coldfire MCF522xx devices.

`STM32F4eval.bld`
: Builds code for an STM32F4xx on the ST STM3240G-EVAL board.

`STM32F4disco.bld`
: Builds code for an STM32F4xx on the STM32F4 Discovery board.

The *Docs* folder includes useful documentation about FAT file systems, including *fatgen103.pdf* from Microsoft.

Some useful links:
`http://www.seas.ucla.edu/classes/mkampe/cs111.sq05/docs/dos.html`

1.1 Design objectives

This is a filing system for embedded systems, in particular for use with SD/SDHC/MMC and other Flash cards. You must provide the sector read and write routines.

Because the filing system is designed for use with removable media, it is assumed that they will be formatted on a PC.

RAM usage is more important than performance, although the use of the LRU sector cache considerably improves performance with both hard and floppy drives, as well as with slow Flash drives.

1.2 Capabilities

- FAT112, FAT16 and FAT32 are all supported.
- Directories and subdirectories are supported.
- The ANS Forth File wordset is supported.
- Can be used with any media type having a sector read and write interface.

1.3 Restrictions

- A 32 bit host Forth is assumed.
- Sectors must be 512 bytes in length. As a result, 2Gb standard capacity SD cards may not work, but SDHC cards will work.
- Sector numbers are 32 bit unsigned numbers. This restricts drive size to 2048Gb (2Tb) before considering restrictions in the FAT32 data.
- Only one drive is supported, although future expansion is provided for.
- File size is internally held as a 32 bit number, and is thus restricted to 4Gb per file.
- The file system is not thread safe. Enable the interlock (e.g. a semaphore) to the ANS file layer if you need thread safe operation.
- Directory deletion must be performed file by file, unless you use the extension in *DelDirs.fth*.
- File names must be in the old DOS 8.3 format. Long file names are not supported, but drives with long file names can be handled using the short alias.
- Drive formatting is not supported.
- The file system must be explicitly initialised at each power up and insertion of a card/drive.

1.4 Build files

Each supported configuration has a build file with a *.bld* extension. The build file contains equates describing the maximum number of open files, number of sector buffers and some CPU characteristics. The build file than loads a the required files, e.g. for an STM32F4 board:

```
include %FFDir%\XC6harn                \ harness
include %FFDir%\Drivers\sdioSTM32F4xx \ STM32F4 SDIO (1-bit DMA)
include %FFDir%\FatCore                \ core file system code
include %FFDir%\PlainDir               \ simple directory listing
include %FFDir%\DelDirs                \ directory and disk deletion
```

Note the use of text macros to define the directory containing the file system code.

The file *sdioSTM32F4xx.fth* is the CPU specific driver. Where there are hardware differences, e.g. for the card detect pin, write protectpin, power control or pin assignments, there may also be a file describing the hardware. This is often called *\i{hwxxxxxx.fth) and may be compiled before the build file.

1.5 Configuration options

The following equates control compilation of some features. The defaults here are used if the equates have not previously been defined, e.g.in your control file.

`1 equu FATopen?  \ -- flag`
Set non-zero to compile the FAT code with all heads present.

`1 equ FATtext?  \ -- flag`
Set non-zero to compile the FAT code for `READ-LINE` and `WRITE-LINE`.

`1 equ FATio?    \ -- flag`
Set non-zero to compile the FAT code with a file generic I/O device.

`1 equ FATspace? \ -- flag`

Set non-zero to compile the FAT code for used space.

```
0 equ FATtest?  \ -- flag
```

Set non-zero to compile the FAT code with test code.

2 FAT file system core

The file *FatCore.fth* contains the bulk of the file system code.

2.1 Buffers, globals and tools

File reads and writes use sector cache buffers for file I/O. One buffer gives the lowest performance, since accessing more than one sector causes disk thrashing. `WRITE-FILE` accesses the FAT table and the data sectors of the disk so there should be at least two sector buffers for each file being actively written. `READ-LINE` tends to like more than two buffers - six are recommended, especially with slow drives.

```
#64 equ MAX_PATH        \ -- len
```
The maximum size of a pathname including the file name.

```
#buffers bytes/sec * buffer: bufs       \ -- addr
```
Sector cache.

```
#buffers cells buffer: csects   \ -- addr
```
Holds sector numbers of the currently buffered sectors.

```
#buffers cells buffer: ctr#s    \ -- addr ; SFP007
```
Holds transaction numbers of the currently buffered sectors. The lowest number identifies the oldest sector.

```
#buffers buffer: wrpendings     \ -- addr
```
Holds the write-pending flags for the currently buffered sectors.

```
#buffers buffer: pdrives        \ -- addr
```
Holds the physical devices for the currently buffered sectors.

```
variable buff#  \ -- addr
```
Holds the next available sector buffer number (0, 1, ..).

```
variable tr#    \ -- addr ; SFP007
```
Holds the current transaction number.

```
cvariable cdrive        \ -- addr
```
Holds the current drive number, which must be 0 for the moment.
The following variables hold the disk characteristics. These will be changed at a later date to support multiple drives by providing a structure per drive.

```
variable SectorsPerTrack        \ -- addr
```
Holds the number of sectors per track for the raw disk.

```
variable NumHeads       \ -- addr
```
Holds the number of heads for the raw disk.

```
variable rootentries    \ -- addr
```
Number of 32-byte directory entries in the root directory.

```
variable rootdsecs      \ -- addr
```
Number of sectors in the root directory.

```
variable rootsector     \ -- addr
```
Sector of root directory.

```
variable sec/cluster    \ -- addr
```

Sectors per cluster.

`variable clusshift \ -- addr`
Equivalent shift count for sectors per cluster.

`variable reservedsec \ -- addr`
Reserved sectors: 1 for FAT12/16, usually 32 for FAT32.

`variable totalsec \ -- addr`
Total number of sectors.

`variable FATsize \ -- addr`
FAT size in sectors.

`variable basesector \ -- addr`
Start sector of the current partition, currently 0

`variable dcurs \ -- addr`
Holds the display cursor position. This is an aid for later directory listing tools.

`2variable savedDirEnt \ -- addr`
Directory sectory and entry saved when a file/directory is found.
The following words manipulate the data above.

`: initSecCache \ --`
Clear the sector cache buffers.

`: buf \ -- addr`
return the address of the current/next sector cache buffer.

`: csect \ -- addr`
Return the address holding the current sector number.

`: ctr# \ -- addr`
Return the address holding the current transaction number.

`: writePend \ -- addr`
Return the address of the current sector write-pending flags (bytes).

`: pdrive \ -- addr`
Return the address of the device number (byte) in the cache line.

`: bumpBuff# \ --`
Step the current sector to the next sector cache buffer.

`: wmark \ --`
Mark the current sector cache buffer as modified, so that it will be written back.

`: RootDirSectors \ -- n`
Number of sectors for the root directory.

`: StartSector \ -- n`
Return the sector number of the first data sector.

`: c>s \ nclus -- nsec`
Convert a number of clusters to the corresponding number of sectors. The `EQU FastShift?` determines whether this word is implemented as a multiply by `sec/cluster` or a shift by `clusshift`.

`: s>c \ nsec -- nclus`

Convert a number of sectors to the corresponding number of clusters. The `EQU FastShift?` determines whether this word is implemented as a divide by `sec/cluster` or a shift by `clusshift`.

`: cluster>sec  \ cluster -- sector`
Converts a cluster number to a physical sector.

`: sec>cluster  \ sector -- cluster`
Converts a physical sector to a cluster number.

`: b>c  \ bytes -- clusters`
Returns the number of clusters needed to hold a number of bytes.

`: b>cn  \ bytes -- nclus noff`
Convert a number of sectors to a number of complete clusters and the sector offset (0..noff) within the following partially filled cluster.

`: c>b  \ clusters -- bytes`
Converts clusters to a byte count.

`: CountofClusters  \ -- n`
Number of clusters available for data. According to Microsoft, the FAT type (12/16/32) depends solely on this number of clusters, and not on the ASCII string in sector 0.

`: nodir  \ --`
Reset the directory display cursor.

2.2 Sector operations

`: wflush  \ --`
If the current sector cache buffer is marked as modified, write it back.

`: flushall  \ --`
Write all marked sector cache buffers back.

`: seeksector  \ sector -- found? ; look for a current sector`
Return true if the given sector is already cached.

`: setCtr#  \ --`
Update the global transaction number and set the current sector's transaction number from it.

`: oldest  \ --`
Set `buff#` to the oldest cached sector.

`: readsector  \ sector --`
Ensure sector is in the cache buffers.

2.3 FAT type specific routines

Several routines are vectored according to the FAT implementation on the drive. These routines use a table of xts to hold the action required for FAT12, FAT16 and FAT32. The variable `FATtype` holds the offset in bytes used to index the table.

Note that this code assumes a 32 bit Forth implementation.

`variable FATtype  \ -- addr`
Holds the offset (0/4/8) in the FAT function tables.

`: FATexec  \ ?? table -- ??`
From the function table, executes the function indexed from `FATtype`.

```
: *3/2          ( n -- n' )     dup 2/ +  ;
```
Signed multiply by 3/2. For FAT12 operations.

```
: *2/3          ( n -- n' )     $AAAAAAAA um* nip ;
```
Unsigned multiply by 2/3. For FAT12 operations.

```
create /fats    \ -- addr
```
Vector table for `/FAT` below.

```
: /fat          \ #bytes -- #entries
```
Given a memory size in bytes, returns the number of complete FAT entries in it.

```
create FAToffsets       \ -- addr
```
Vector table for `FAToffset` below.

```
: FAToffset     \ cluster -- offset sector
```
Get sector number and offset for the given FAT entry.

```
: FAT@32        \ cluster -- n
```
Return the 28 bit FAT32 entry for the given cluster.

```
: FAT@16        \ cluster -- n
```
Return the 16 bit FAT16 entry for the given cluster.

```
: fat@12bndry   \ offset sector -- w
```
Read a 16 bit FAT12 entry that crosses a sector boundary. *Offset* is `Bytes/Sector-1` and *sector* is the sector number of the first sector.

```
: FAT@12        \ cluster -- n
```
Return the 12 bit FAT12 entry for the given cluster.

```
create FAT@s    \ -- addr
```
Vector table for `FAT@`.

```
: FAT@          \ cluster -- n
```
Return the FAT entry for the given cluster.

```
: FAT!32        \ n cluster --
```
Set the FAT32 entry for the given cluster.

```
: FAT!16        \ n cluster --
```
Set the FAT16 entry for the given cluster.

```
: fat!12bndry   \ w offset sector --
```
Write a 16 bit FAT12 entry that crosses a sector boundary. *Offset* is `Bytes/Sector-1` and *sector* is the sector number of the first sector.

```
: wmerge12      \ n cluster nr -- nw
```
Merge two FAT12 entries before writing. *N* is the new entry, *nr* is the entry from the disc, and *cluster* is the entry number to which the value *nw* will be written.

```
: FAT!12        \ n cluster --
```
Set the FAT12 entry for the given cluster.

```
create FAT!s    \ -- addr
```
Vector table for `FAT@`.

```
: FAT!          \ n cluster --
```
Set the FAT entry for the given cluster.

```
create |cls     \ -- addr
```

Table holding last-cluster markers for each FAT type.

`: |cl \ -- n`
Last-cluster marker.

2.4 FAT File system initialisation

`variable bptr \ -- addr`
Stream pointer into the current sector cache buffer.

`variable bperror \ -- addr`
If set non-zero, an error exists in the BPB.

`: newb \ --`
Reset the input stream.

`: skipb \ --`
Step to the next stream byte.

`: getb \ -- byte`
Get stream byte and step to next.

`: getw \ -- word`
Get 16 bit stream item and step past it. The item is read in little-endian format.

`: getl \ -- long`
Get 32 bit stream item and step past it. The item is read in little-endian format.

`: wantb \ byte --`
Read the next stream byte and if it is not the given byte, set `bperror` to 1.

`: wantw \ word --`
Read the next stream word and if it is not the given word, set `bperror` to 1.

`: lengthen \ w -- u`
Read the next 32 bit item. If w is non-zero, discard the new item and return w, otherwise return the new item.

`: bp_init \ --`
Initialise the system, read the drive characteristics and set up for file operations.

2.5 Directories

Each sector of a directory contains 16 entries indexed by `DIRENTRY`. The sector number `DIRSECTOR` is a 32-bit value addressing up to 2Tbytes. The root directory is a contiguous run of `ROOTDSECS` sectors starting at the sector in `ROOTSECTOR`. Subdirectories use a chain in the FAT table just like files. So, the root directory has a fixed number of possible directory entries. `DIRCLUSTER` is 0 when the root directory is selected.

For most drives, cluster 0 and cluster 1 are occupied by two copies of the FAT. Cluster 2 is **usually** the start of the root directory. For FAT12 and FAT16, the root directory is of fixed size.

`: dt \ x acc n bits -- acc' x`
Shift accumulator left by *bits*, add in n, and bring the next item x to the top. Ugh! Convert the current date and time into the packed 32 bit format for directory entries:

```
yyyyyyymmmmddddd:hhhhhmmmmmmssssss
```

`variable dircluster \ -- addr`

Holds first cluster of the current directory, or 0 for the root directory.

`variable dirsector \ -- addr`
Holds current sector number of directory.

`variable direntry \ -- addr`
Holds current entry number.

`: dir[] \ offset -- addr`
Index into the current directory entry. Ensures that the entry is in the sector cache.

`: secmask \ -- sector mask`
Return the sector and the mask for testing the current directory sector.

`: firstsec? \ -- flag`
Return true if we are at the first sector of a cluster.

`: lastsec? \ -- flag`
Return true if we are at the last sector of a cluster.

`: attrib? \ mask -- flag`
Test file attribute bits, returning non-zero if any bits in the mask are set. The attributes are a bit mask

- bit 0, $01 - read only
- bit 1, $02 - hidden
- bit 2, $04 - system
- bit 3, $08 - volume ID
- bit 4, $10 - directory
- bit 5, $20 - archive
- $0F - long file name

`: dir? \ -- b`
Return the first byte of the directory entry.

`: dirsmudged? \ -- flag`
Return true if the current directory entry is "smudged".

`: dirsmudge \ --`
Mark the current directory entry as "smudged".

`: get-es \ -- entry sector`
Return the current directory entry and sector.

`: set-es \ entry sector --`
Set the current directory entry and sector.

`: dircluster@ \ -- clus`
Return the current directory cluster.

`: dircluster! \ clus --`
Sect the current directory cluster, and hence start sector.

`: DirValid? \ -- flag`
Are we at a valid directory entry for display?

`: sbackward \ --`
Go to the previous directory cluster, use 1- for root.

```
: sforward      \ --
```
Go to the next directory cluster, use FAT for subfolders, 1+ for root

```
: sector0?      \ -- flag
```
Are we at the first sector of a directory?

```
: dsecup        \ --
```
Step to the next directory sector.

```
: dsecdn        \ --
```
Step to the previous directory sector.

```
: entbump       \ offset -- n
```
Go to next/prev entry, where offset = +/-1

```
: dir++         \ --
```
Go to the next directory sector/entry.

```
: dir--         \ --
```
Go to the previous directory sector/entry.

```
: noend?        \ clus -- notlast?
```
Return true if the cluster number is not an end-chain-marker.

```
: OKprev        \ -- flag
```
True if the directory pointer is not at the very beginning of the directory.

```
: OKnext        \ -- flag ; okay to go forward?
```
True if the directory pointer is not at the very end of the directory.

```
: dirnext       \ -- flag
```
Jump to the next valid directory entry. Return non-zero if the next entry is valid.

```
: todir         \ --
```
Skip to first valid entry.

```
: dirprev       \ -- flag
```
Jump to previous valid directory entry.

```
#13 buffer: npad        \ -- addr
```
Temporary buffer for building 8.3 filenames.

```
: padc          \ char --
```
Add one character to `npad`.

```
: pad$          \ caddr len --
```
Add string to `npad`.

```
: dfname        \ -- caddr len
```
Return the main part of the file name with trailing spaces removed.

```
: dfext         \ -- caddr len
```
Return the extension part of the file name with trailing spaces removed.

```
: dfullname     \ -- caddr len
```
Return filename with extension.

```
: parent?       \ -- flag
```
Return true if the current directory entry is a parent directory, i.e. `".."`.

```
: cluster0      \ -- n
```

Get the first cluster number of the data for this entry.

```
: setClus0      \ u dirent --
```
Set the cluster number into the given directory entry.

```
: cluster0!     \ n --
```
Set the first cluster number of the data for this entry.

```
: folder?       \ -- flag
```
Is the current entry a directory/folder?

```
: dsize         \ -- n
```
The file size in bytes of the file for the current directory entry.

```
: dirsect0      \ -- sect
```
Return the starting sector of the current directory.

```
: dirhome       \ --
```
Select the first entry in the current directory.

2.6 FAT chains

```
: freechain     \ clus0 --
```
Writes zeros to a chain in the FAT table.

```
: _freesecs     \ --
```
Frees the chain of the current file.

```
: _delete       \ --
```
Deletes the current file.

```
: getcluster    \ clus -- clus'
```
Gets the next free cluster number after this one and marks it as the last.

```
: chain+        \ cl -- cl'
```
Appends a cluster to the end of the chain.

```
: shrink        \ clus0 --
```
Shrinks a FAT chain and marks the last cluster.

```
: expand        \ clus0 n --
```
Expands a FAT chain and marks the last cluster.

2.7 File/Path name handling

File names are checked against allowable characters. Dot (.) is allowable once to separate filename and ext fields. The widths of these fields may not exceed 8 and 3 characters respectively.

```
create goodchars        \ -- addr
```
An array of packed bit flags: 1=allowed: 0..7, 8..9, ...127

```
: goodch        \ char -- flag
```
Check if the character is allowable in a file/path name.

```
: goodstr       \ caddr len limit -- flag
```
Check that string has only allowable chars and is not too long.

```
: name.ext      \ a u -- a1 u1 a2 u2
```
Separate filename string into name and extension strings.

```
: goodname      \ caddr len -- flag
```

Are filename and extension (could be in a path) valid?
A filename that contains one or more slashes (either '/' or' \') is assumed to be a file path ending in a filename. When one of these is encountered, the path string is stripped off and used it to set the file path. The current directory is not changed. If all of your files are in the same directory, you can use `CWD` and friends to set the current directory. Begin with a slash to start at the root directory.

```
: /parse        \ addr len -- addr' len' a1 u1
```
Pick the file name off left end of path. The string *a1/u1* contains the next directory element

2.8 Current Working Directory

The current file path is tracked by a path string. The file tree is navigated by selecting the cluster for the subdirectory and adding its name to the path. A `".."` folder name trims the path name to go back a level. This path string is for display use.

Because this is not a fully thread-safe file system and only has a single "working directory" for all tasks/threads, you will need to be careful with external access systems such as HTTP, FTP and USB.

```
pathmax buffer: pathbuf         \ -- addr
```
Path name buffer.

```
: nopath        \ --
```
Empty the path name buffer.

```
: path          \ -- caddr len
```
Return the current path string.

```
: bpath         \ b --
```
Bump the pathname string length.

```
: pc+           \ char --
```
Add *char* to the path string.

```
: /?            \ -- flag
```
Return true if the last+1 character is '/'.

```
: trimp         \ --
```
Trim path to just before the last '/' character.

```
: addp          \ caddr len --
```
Add string to path, separated by a '/' character.

2.9 Directory management

```
: dirstart      \ cluster --
```
Starts referencing a new directory at the given cluster.

```
: rootdir       \ --
```
Start at the root directory.

```
: dirnest       \ --
```
Start at the folder pointed to by the current directory entry.

```
: sel-folder    \ --
```
Select the current directory entry as a folder.

`: _test-file  \ caddr len attr -- found?`
Search in the current directory for an existing filename matching none of the given* attribute bits. Return non-zero if found. If found, the directory sector and entry are saved in `savedDirEnt` and the directory pointers address the found filename entry. The attribute bits settings are:

- bit 0, $01 - read only
- bit 1, $02 - hidden
- bit 2, $04 - system
- bit 3, $08 - volume ID
- bit 4, $10 - directory
- bit 5, $20 - archive
- $0F - long file name

`: test-file  \ caddr len -- found?`
Search for a file that is neither a directory nor a volume ID. Upon exit when found, the directory pointers address the found file.

`: cd+  \ caddr len -- ior`
Open new directory. `CD+` changes the directory up or down a level.

`: $cwd  \ addr len -- ior`
Open a subdirectory given a file path. *Ior*=0 if okay.

`: setdir  \ addr len -- addr' len' ior`
Select a subdirectory given a file path, stripping off the filename.

`: wipeclus  \ clus --`
Fill a cluster with zeros, `THROW` on error.

`: getdir  \ -- ior`
Finds the first directory entry that may be overwritten.

2.10 Handles and File support

File support is ANS standard, after the stream model used by mainstream computing. File status is held in array of records:

```
8-bit filemode           3=R/W, 1=R/O, 2=W/O
                         bit 2 = 1 = updated
                         bit 1 = 1 = writable
                         bit 0 = 1 = readable
8-bit device             reserved for file device/drive
8-bit directory entry    directory index (0..15) for this file
8-bit                    reserved
32-bit directory sector  directory index for this file
32-bit position          file position for up to 4GB files
32-bit start cluster     FAT location
32-bit current sector    active sector
32-bit file size         length in bytes
```

The following data types are used:

`fam` "File Access Method", describes read/write permission etc.

`ior` "IO Result", A return result from most IO calls, this value is 0 for success or non-zero as an error-code.

`fileid` "File Identifier", a handle for a file.

`1 constant R/O  \ -- fam`
Get ReadOnly fam. This **must** be bit 0 to match DOS.

`2 constant W/O  \ -- fam`
Get WriteOnly fam.

`3 constant R/W  \ -- fam`
Get ReadWrite fam.

`: bin           \ fam -- fam'`
Modify a file-access method to include BINARY.
The corresponding FAT attribute bits at byte 11 in each directory entry are:

```
#define ATTR_READ_ONLY 0x01
#define ATTR_HIDDEN    0x02
#define ATTR_SYSTEM    0x04
#define ATTR_VOLUME_ID 0x08
#define ATTR_DIRECTORY 0x10
#define ATTR_ARCHIVE   0x20
  top two bits are always zero
#define ATTR_LONG_NAME 0x0F   \ continuation entry of long file name
```

`6 cells constant bytes/handle   \ -- len`
Number of bytes needed for each handle. Must be an integer number of cells for portability.

`#files bytes/handle * buffer: filehandles       \ -- addr`
Array of file handle records.

`cell +user handle       \ -- addr`
Holds file handle during some operations.

`: +handle[]     \ n -- addr`
Given a handle, return the address in the handle record array. Why not just use the address as the handle?

```
\ H.FAM must be first.
: h.fam   ( -- addr )  0 +handle[]  ;                  \ r/o, w/o, r/w, etc.
: h.dev   ( -- addr )  1 +handle[] ;                   \ device of this file, 0..15
: h.dire  ( -- addr )  2 +handle[]  ;                  \ directory entry 0..15
: h.dirs  ( -- addr )  cell +handle[]  ;               \ directory sector
: h.pos   ( -- addr )  [ 2 cells ] literal +handle[] ;  \ file position
: h.clus0 ( -- addr )  [ 3 cells ] literal +handle[] ;  \ starting cluster
: h.sect  ( -- addr )  [ 4 cells ] literal +handle[] ;  \ current sector
: h.len   ( -- addr )  [ 5 cells ] literal +handle[] ;  \ file length
```

`: closeall      \ --`
Wipes out the file structure to ensure files are closed.

`: h.clus!       \ cluster --`
Set the current file sector.

`: h:dir \ --`
Points the directory pointers at the file addressed by current handle.

`: after/before \ -- after before`
Get the number of bytes before and after the current sector position.

`: _resize \ len --`
Given a file size in bytes, allocate clusters in FAT for the file.

`: nexthan \ -- fileid ior`
Find available file handle, and make it the current handle.

`: (repo-file) \ u -- ior`
Repositions the current file. Must not be past the end. Unprotected.

`: repo-file \ u -- ior`
Repositions the current file. Must not be past the end. protected.

`: [dir \ -- ; R: -- i*x`
Saves current directory information on the return stack for later restoration by `DIR]`. This word depends on the return address being a single cell on the Forth return stack.

`: dir] \ -- ; R: i*x --`
Restores directory information previously saved on the return stack by `[DIR`. This word depends on the return address being a single cell on the Forth return stack.

2.11 File Read

`: samesect? \ -- flag`
Checks if the next read or write is in a new sector.

`: sameclus? \ -- cluster same?`
Checks if the next sector is in the same cluster.

`: bumppos \ n -- overflow`
Bump file position, check for wrap to next sector.

`: bumpsec \ len --`
Bump file position and maybe sector/cluster.

`: read-1sec \ addr len -- addr' len' n`
Read from current sector.

`: _read-file \ baddr blen -- len ior`
File read primitive.

2.12 File write

`: timestamp \ --`
Adds the time and date to the current directory entry.

`: copytime \ --`
Copy the creation date.

`: archive+ \ --`
Sets the current directory entry's archive bit.

`: filestamp \ --`
Stamps the directory entry addressed by the current handle.

`: stamp \ --`

Mark the file as stamp-pending.

`: newchain \ --`
Begin a new chain.

`: nextwsec \ --`
Find next writable sector in file.

`: write-1sec \ a u -- a' u'`
Write 512 bytes or less to file.

`create crlf \ -- addr`
Holds a CR/LF pair for `WRITE-LINE` below.

`: name>dir \ caddr len dirent --`
Copy the given file name into the given directory entry.

`: prepdir \ caddr len clus0 attribs --`
Set up a new directory entry using the given name, cluster and attributes. Note that *attribs* is in DOS form, not in the form used by *h.fam*.

`: _newfold \ addr len cluster --`
add a new folder entry to the current folder, and step to the next directory entry.

`: es^ \ --`
Set the current handle's entry and sector fields.

`: ?open_err \ a b c flag err# -- a b c | -- -1 err# [exits caller]`
If flag is false/0, just the first three parameters are returned. If flag is true, the first three parameters are discarded and -1 and the error number are returned and exit is from the caller. Provided to reduce code space and dependent on the return address being on the Forth return address. The caller **must not** use local variables.

`: _open-file \ caddr len fam -- fileid ior`
File open primitive. Performs no directory restoration. This word will **not** open a directory.

`: _create-file \ caddr len fam -- fileid ior`
Create a file on disk, returning a 0 ior for success and a file id. If the file already exists, it is truncated to zero length. Performs no directory restoration.

`: _mkdir \ c-addr u -- ior`
C-addr/u is an 8.3 directory name with no directory separators. `MKDIR` creates a new directory of that name in the current directory. Directory removal is not properly supported without the code in *DelDirs*.fth. You can use `DELETE-FILE` to remove an empty directory; but if the directory is not emptied the disk will be corrupted. Performs no directory restoration.

`: _mkDirEx \ c-addr u -- ior`
As `-mkdir`, but supports full pathnames.

`: (read-file) \ caddr len fileid -- #read ior`
Read data from a file. The number of characters actually read is returned as *#read*, and *ior* is returned 0 for a successful read. Unprotected.

`: (write-file) \ caddr len fileid -- ior`
Write a block of memory to a file. Unprotected.

`: (close-file) \ fileid -- ior`
Close an open file. Unprotected.

`: (read-line) \ caddr u1 fileid -- u2 flag ior`

Read an line of text from a file into a buffer, without EOL. Unprotected.

`: (delete-file) \ caddr len -- ior`
Delete a named file from disk, and return ior=0 on success. Unprotected.

`: (rename-file) \ caddr1 len1 caddr2 len2 -- ior`
Rename a named file on the disk, and return ior=0 on success. Unprotected.

`: (initFATfs)  \ -- ior`
Initialize the FAT file system. Unprotected.

`: (termFATfs)  \ --`
Shut down the FAT file system. Unprotected.

2.13 Protection tools

`Semaphore FileSem  \ --`
Interlocks the file system for multitasking.

`: +FileLock  \ --`
Wait until the file system is available and lock access

`: -FileLock  \ --`
Unlock access to the file system.

2.14 User API, mostly ANS Forth

The API layer is made safe for multitasking systems by a semaphore. Because the underlying hardware driver may `THROW`, e.g. for a fatal read/write error, these words include a `CATCH`.

`: open-file  \ caddr len fam -- fileid ior`
Open an existing file on disk. This word will **not** open a directory.

`: create-file  \ caddr len fam -- fileid ior`
Create a file on disk, returning a 0 ior for success and a file id. If the file already exists, it is truncated to zero length.

`: read-file  \ caddr len fileid -- #read ior`
Read data from a file. The number of characters actually read is returned as *#read*, and *ior* is returned 0 for a successful read.

`: write-file  \ caddr len fileid -- ior`
Write a block of memory to a file.

`: close-file  \ fileid -- ior`
Close an open file.

`: write-line  \ caddr len fileid -- ior`
Write data followed by CR/LF pair. IOR=0 for success.

`: read-line  \ caddr u1 fileid -- u2 flag ior`
Read an line of text from a file into a buffer, without EOL. The EOL marker may be either CR/LF (DOS) or LF (Unix). Read the next line from the file specified by fileid into memory at the address *caddr*. At most *u1* characters are read. Up to two line-terminating characters may be read into memory at the end of the line, but are not included in the count *u2*. The line buffer provided by *caddr* should be at least *u1*+*2* characters long.
If the operation succeeds, *flag* is true and *ior* is zero. If a line terminator was received before *u1* characters were read, then *u2* is the number of characters, not including the line terminator, actually read (0 <= u2 <= u1). When u1 = u2, the line terminator has yet to be reached.

If the operation is initiated when the value returned by `FILE-POSITION` is equal to the value returned by `FILE-SIZE` for the file identified by *fileid*, *flag* is false, *ior* is zero, and *u2* is zero. If *ior* is non-zero, an exception occurred during the operation and *ior* is the I/O result code.

An ambiguous condition exists if the operation is initiated when the value returned by `FILE-POSITION` is greater than the value returned by `FILE-SIZE` for the file identified by fileid, or if the requested operation attempts to read portions of the file not written.

At the conclusion of the operation, `FILE-POSITION` returns the next file position after the last character read.

`: resize-file  \ len-ud fileid -- ior`
Set the size of the file to *ud*, an unsigned double number. After using `RESIZE-FILE`, the result returned by `FILE-POSITION` may be invalid.

`: reposition-file  \ len-ud fileid -- ior`
Set file position, and return ior=0 on success. See the ANS Forth document.

`: file-position \ fileid -- len-ud ior`
Return file position, and return ior=0 on success.

`: file-size  \ fileid -- len-ud ior`
Get size in bytes of an open file as a double number, and return ior=0 on success.

`: delete-file  \ caddr len -- ior`
Delete a named file from disk, and return ior=0 on success.

`: FileExist?  \ c-addr u -- flag`
Look to see if a specified file exists, returning TRUE if the file exists.

`: file-status  \ caddr len -- x ior`
Return the status of the file identified by the character string *c-addr/len*. If the file exists, *ior* is zero; otherwise *ior* is the implementation-defined I/O result code. *X* contains implementation-defined information about the file (always zero for FATfiler).

`: rename-file  \ caddr1 len1 caddr2 len2 -- ior`
Rename the file named by the character string *c1addr/len1* to the name in the character string *caddr2/len2*. *Ior* is zero on success.

`: mkdir  \ c-addr u -- ior`
C-addr/u is an 8.3 directory name with no directory separators. `MKDIR` creates a new directory of that name in the current directory. Directory removal is not supported unless the code in *DelDirs.fth* is compiled. You can use `DELETE-FILE` to remove a directory but this is not recommended unless you know that the directory is empty. Otherwise, you may end up with lost chains.

`: mkDirEx  \ c-addr u -- ior`
As `mkdir` but accepts directory separators. Directories can be created outside the current directory. Directory removal is not supported unless the code in *DelDirs.fth* is compiled. You can use `DELETE-FILE` to remove a directory but this is not recommended unless you know that the directory is empty. Otherwise, you may end up with lost chains.

`: cwd  \ "<dir>" -- ; CWD <dirname>`
An equivalent to the DOS CD or Unix cwd command.

`: pwd  \ -- ; PWD`

Displays the current working directory name. Because this is not a fully thread-safe file system and only has a single "working directory" for all tasks/threads, you will need to be careful with external access systems such as HTTP, FTP and USB.

`: SyncDrive \ --`
Force all updated sectors to the disk and empty the sector cache. This operation is necessary when directories are modified by a separate task, e.g. when USB makes direct sector read/write accesses.

`: initFATfs \ -- ior`
Initialize the FAT file system.

`: termFATfs \ --`
Shut down the FAT file system.

2.15 Generic I/O file device

This device allows files to be used with `KEY`, `EMIT` and friends. To avoid having to create a device for each file, one device is built that takes the file id/handle from the `USER` variable `MyFID`.

The application programmer is responsible for opening the file, setting the user variable and closing the file.

`cell +user MyFID \ -- addr`
User variable holding the file handle (0..n-1) in the low 16 bits. The four bytes are used as follows:

`0/1`	file handle
`2`	character buffer for file read/write
`3`	flags
`3.0`	reserved
`3.1`	set if EOF
`3.2`	set after file error

`KEY?` always returns true. After an EOF or file error, `KEY` always returns an LF character. This prevents words such as `ACCEPT` blocking on error.

`: FileKey? \ -- flag`
Returns true. Does not call `PAUSE`.

`: FileKey \ -- char`
The file version of `KEY`. Executes `PAUSE` if LF is returned.

`: FileType \ caddr len --`
The file equivalent of `TYPE`. After any file error, the string is discarded.

`: FileEmit \ char --`
The file equivalent of `EMIT`. Uses `FileType`.

`: FileCr \ char --`
The file equivalent of `CR`. Uses `FileType` and calls `PAUSE`.

`create FileCon \ -- addr ; OUT managed by upper driver`
Generic I/O device for files.

`: FileAccept \ c-addr +n1 -- +n2 ; read up to LEN chars into ADDR`
As `ACCEPT` with no echoing or flow control.

`: FileEOF? \ -- flag`
Return true if the file is at EOF.

`: FileErr? \ -- flag`
Return true if the file has returned an error.

`: ?SetFID \ fileid ior -- ior`
Set `MyFID` according to the results of a file open or create.

`: OpenFileCon \ caddr len -- ior`
Open the given file path in read-only mode as the current task's file input device. The path is given by *caddr/len* and the mode by *fam*. The I/O pointers `IPVEC` and `OPVEC` are not set.

`: CreateFileCon \ caddr len -- ior`
Create the given file path in read/write mode as the current task's file output device. The path is given by *caddr/len* and the mode by *fam*. The I/O pointers `IPVEC` and `OPVEC` are not set.

`: CloseFileCon \ --`
Close the file handle in `MyFID`.

2.16 Drive details

`FATspace? [if]`
The code below will be compiled if the equate is non-zero.

`: .FATtype \ --`
Display FAT type.

`: .Drive \ --`
Display drive details. `InitFATfs` must have been successfully run.

`: TotalClusters \ -- n`
Returns the

2.17 Test code

`FATtest? [if]`
The code below will be compiled if the equate is non-zero.

`#512 buffer: SBscr \ -- addr`
Scratch sector buffer.

`-1 value hFile \ --`
File handle for tests.

`: go \ --`
Start up the file system.

`: testline \ n -- caddr len`
A line of text to output.

`: testw \ #lines "<filename>" --`
Test file writing, use in the form:

```
#lines testw file.ext
```

`: testr \ "<filename>" --`
Test file reading with `read-line`. Use in the form:

```
  testr file.ext
```

```
: ?eol          \ caddr len -- caddr' len'
```
Strip the lst character from the line if it is CR or LF.

```
: rdLnCc        \ caddr len -- len' ; 0=eof
```
Read a line from the file on a character by character basis. The line terminator is a line feed character.

```
: testcr        \ "<filename>" --
```
Test file reading character by character. Use in the form:

```
  testcr file.ext
```

```
0 value <s>     \ -- u
```
Last sector dumped.

```
: secdump       \ u --
```
Display the contents of the given sector.

```
: ns            \ --
```
Dump next sector.

```
: ps            \ --
```
Dump previous sector.

```
: ss            \ --
```
Dump same sector again.

3 Simple Directory listing

3.1 Directory listing

The directory listing is designed to fit on a small screen so it usually lists just filenames and not very many at that. The cursor position is marked using a method different from that used to mark highlighting. For example, a different background shade for highlights and foreground shade for the cursor.

3.2 File size display

Display size in bytes, Kbytes or Mbytes. Sized to fit in a 6-char field. K and M units are 2^10 and 2^20 scale.

```
: #n            \ n u --
```
Display n as u digits.

```
: ##            \ n --
```
Display n as two or more digits.

```
: 10^n          \ n -- 10^n
```
Generate 10^n.

```
: _size         \ u scale c -- int 0
```
Show size split in int and frac parts.

```
: (size)        \ u -- addr len
```
Convert file size using 6 chars or less

```
: .size         ( u -- ) (size) type ;
```
Display file size.

3.3 File date and time

```
: unp           \ u bits -- field u'
```
Unpack the low *bits* bits of *u* as *field* with the shifted accumulator *u'*.

```
: .filedate     \ u --
```
Prints a 16-bit date in yyyy.mm.dd format.

```
: .filetime     \ u --
```
Prints a 16-bit time in hh:mm:ss format.

```
: d-date        ( -- )  24 dir[] uaLEw@ .filedate  ;
```
Display file date.

```
: d-time        ( -- )  22 dir[] uaLEw@ .filetime  ;
```
Display file time.

3.4 Display operations

```
: >pos          \ +n --
```
Place cursor on current line to column n if possible.

```
: d-name        ( -- )  dfullname type  ;
```
Display filename.

```
: d-size        \ --
```

Display file size or "folder".

```
: zcurs         \ --
```
Zero cursor position.

```
variable dirmode  \ -- addr
```
Set non-zero to get detailed directory listing.

```
variable +Dirs  \ -- addr
```
Set non-zero to display directories in listings.

```
: _dirline      \ row -- row
```
Display filename on one line.

```
: _dir          \ live? -- #lines
```
Display short list of file names.

```
: dirfine       \ --
```
Display detailed listing for one entry.

```
: delete        \ --
```
Deletes the highlighted file and repaints the directory.

```
: dir           \ --
```
Standard directory listing.

4 deleting directories

The code in *DelDirs.fth* allows you to delete directories recursively. By deleting from the root directory, the whole disk can be erased.

```
: entValid?     \ -- flag
```
Are we at a valid directory entry? In terms of DOS attributes, we only reject smudged entries and volume labels.

```
: entNext       \ -- flag
```
Step to the next valid directory entry. Return zero at the end of the directory, otherwise non-zero.

```
: toEnt         \ --
```
Skip to first valid entry.

```
: thisDir?      \ -- flag
```
Return true if name is '.'

```
: delFileEnt    \ --
```
Delete the current directory entry, which must be for a file.

```
: delDirEnt     \ --
```
Delete the current directory entry, which must be for a directory.

```
: delEntry      \ --
```
Delete the current directory entry.

```
: delCurrDir    \ --
```
Delete all the files and directories in the current directory.

```
: delDisk       \ --
```
Delete all the files on the disk

```
: _$rmd         \ caddr len -- ior
```
Delete a directory and its contents. Unprotected.

```
: $rmd          \ caddr len -- ior
```
Delete a directory and its contents. Protected.

```
: cd+           \ caddr len -- ior
```
Open new directory. `CD+` changes the directory up or down a level.

5 VFX Forth for Windows build file

5.1 Configuration

```
16 constant dirlines    \ -- n
```
Number of lines on a DIR display screen.

```
80 constant pathmax     \ -- n
```
Length of path string.

```
10 constant #files      \ -- n
```
Maximum number of open files.

```
6 constant #buffers     \ -- n
```
Number of sector buffers.

```
512 constant bytes/sec  \ -- n
```
Number of bytes in a sector - nearly always 512, but Microsoft documents say that it can be different. Some operations will fail if this is not a power of two.

```
bytes/sec 1- constant secsizemask       \ -- mask
```
Bit mask used for some positioning operations. Relies on `bytes/sec` being a power of two.

```
9 constant secshift     \ -- u
```
Number of bits to shift to scale by the sector size. Nearly always 9 as 2^9=512.

```
1 constant FastShift?   \ -- n
```
If non-zero, the conversion between clusters and sectors is performed by a shift rather than a multiply or divide. Set this according to the capabilities of your CPU.

5.2 Files

```
include VFXharn                 \ VFX Forth for Windows harness
include Drivers\WinDriver       \ Windows raw disk read/write
include FatCore                 \ core file system code
include PlainDir                \ simple directory listing
```

6 VFX Forth harness

The harness files permit you to optimise the FAT file system for speed or code size.

6.1 Sector handling

How you code these will depend on the host CPU and Forth implementation.

`: secs>bytes \ u -- #byes`
Multiply a sector number by the number of bytes per sector. Optimise this for performance.

`: bytes>secs \ #bytes -- #secs`
Divide a number of bytes by the number of bytes per sector, returning the number of complete sectors. Optimise this for performance.

`: bytes>offsecs \ #bytes -- offset #secs`
Divide a number of bytes by the number of bytes per sector, returning the offset in a sector and the number of complete sectors. Optimise this for performance.

6.2 Tools

These tools should be compiled as required.

`: buffer: \ u -- ; -- addr`
Reserve *u* bytes of uninitialised read/write data.

`: scan \ addr len char -- addr' len'`
Find char in string.

`: upc \ char -- char'`
Convert *char* to upper case.

`: FOR \ -- ; n -- ; R: -- n'`
Starts a loop that is executed one or more times. At runtime, the loop is entered with the loop count on the stack. In the body of the loop the loop index is on the return stack and counts down from n-1 to zero. Use in the form:

```
( -- n ) FOR ... NEXT ( -- )
```

`: NEXT \ -- ; R: n -- ; -- [n]`
Terminates a `[m ... m]` structure, discarding the loop index.

`: ucompare \ a1 u1 a2 u2 -- notsame?`
Case insensitive string compare. Returns zero for a match.

`: ucmove \ src dest len --`
As `CMOVE` but characters are converted to upper case.

`: N++ \ n1 x -- n1+1 x`
Increment *n1*.

`: byte-split \ w -- lo hi`
Split a 16 bit item into its low and high bytes.

`: byte-join \ lo hi -- w`
Convert two bytes into a 16 bit item.

`: word-split \ x -- wlo whi`

Split a 32 bit item into its low and high 16 bit items.

```
: word-join     \ wlo whi -- x
```
Convert two 16 bit items into a 32 bit item.

```
: qlog2         \ n -- log2[n]
```
Given n, a power of two, return its power.

```
: @C @ ;
```
Fetch cell from code space (e.g. for tables).

6.3 Alignment checking

Some CPUs require aligned memory accesses. The code here checks alignment at run-time.

6.4 Embedded systems compatibility

Embedded systems may have words `TARGET` and `HOST` that may be ignored.

```
[undefined] target [if]
: target ;
: host ;
[then]


: cvariable  variable  ;        \ -- ; -- addr
```
In embedded systems, `CVARIABLE` can be used to reduce RAM use.

6.5 Unaligned memory access

Systems that require aligned accesses for 16 and 32 bit values must provide unaligned access words. Similarly, big-endian CPUs must provide little-endian access as all the FAT data structure items are little-endian items.

7 Disk read/write for VFX Forth for Windows

The file *WinDriver.fth* is the VFX Forth for Windows driver layer for testing the FAT file system with floppy drive A or a Flash drive.

For demonstration on a host PC, this simple harness connects to a floppy or USB drive on any Windows 2000/XP PC. You can set the drive letter using `SetDrive` below. Your C: drive won't work if it is formatted for NTFS. Also, drives with multiple partitions are currently not supported.

The file system needs these words to access the mass storage:

`SecRead` (addr sector dev –) Reads a sector from the specified device. `THROW`s on error.

`SecWrite` (addr sector dev –) Writes a sector to the specified device. `THROW`s on error.

`MassInit` (–) Initializes mass storage access.

`MassTerm` (–) Terminates mass storage access.

To simplify the code, the raw read/write interface treats all read/write errors as fatal, and `THROW`s on error. Retries should be accommodated within the sector read/write code.

7.1 Windows details

From Microsoft's documentation on how to read raw data from a disk: To open a logical drive, direct access is of the form

```
\\.\X:
```

where X: is a hard-drive partition letter, floppy disk drive, or CD-ROM drive.

You can open a physical or logical drive using the CreateFile() application programming interface (API) with these device names provided that you have the appropriate access rights to the drive (that is, you must be an administrator). You must use both the CreateFile() FILE_SHARE_READ and FILE_SHARE_WRITE flags to gain access to the drive.

Once the logical or physical drive has been opened, you can then perform direct I/O to the data on the entire drive. When performing direct disk I/O, you must seek, read, and write in multiples of sector sizes of the device and on sector boundaries. Call DeviceIoControl() using IOCTL_DISK_GET_DRIVE_GEOMETRY to get the bytes per sector, number of sectors, sectors per track, and so forth, so that you can compute the size of the buffer that you will need.

For this implementation, sectors are 512-byte, sector numbers starting from 0. Addresses are RAM source/dest, and ior is nonzero if an error occurs.

7.2 Sector read/write interface

This section depends on the target on which the FAT file system is being hosted. Recode this for your hardware or host operating system.

```
: CHS           \ LBA -- Sector Head Cylinder
```

Calculates the address on the drive of the given sector. By default, this word is commented out as it is usually only required by old IDE drives.

```
Sector   = (LBA mod SectorsPerTrack)+1
Cylinder = (LBA/SectorsPerTrack)/NumHeads
Head     = (LBA/SectorsPerTrack) mod NumHeads
```

`1 constant devices`
Number of physical mass storage devices supported.

`create Drive$   \ -- addr`
Holds a counted string holding the drive name string. The drive letter is the fifth character.

`: SetDrive      \ char --`
Select the required drive (0..n). Must be done before `OpenDrive` below is run. The default is 'A'.

`-1 value hDrive \ -- handle`
Handle returned by Windows for the current raw drive. Set to -1 when closed.

`: openDrive     \ --`
Open the raw drive.

`: closeDrive    \ --`
Close the raw drive.

`: selDrive      \ dev --`
Devices are numbered 0..n. In this case an error occurs if the device is non-zero.

`: brw           \ sector addr -- addr len handle`
Given a sector number and a buffer address, return the address sector length and the raw disc handle.

`: SecRead       \ addr sector dev --`
Read a sector from the specified device. An `err_RawRead THROW` occurs on error.

`: SecWrite      \ addr sector dev --`
Write a sector to the specified device. A `err_RawWrite THROW` occurs on error.

`: MassInit      \ --`
Initialize the mass storage manager. This is done at startup.

`: MassTerm      \ --`
Shut down the mass storage manager. This is done at program exit.

8 Forth 7 cross-compiler harness

The harness files permit you to optimise the FAT file system for speed or code size.

8.1 Sector handling

How you code these will depend on the host CPU and Forth implementation.

`: secs>bytes  \ u -- #byes`
Multiply a sector number by the number of bytes per sector. Optimise this for performance.

`: bytes>secs  \ #bytes -- #secs`
Divide a number of bytes by the number of bytes per sector, returning the number of complete sectors. Optimise this for performance.

`: bytes>offsecs \ #bytes -- offset #secs`
Divide a number of bytes by the number of bytes per sector, returning the offset in a sector and the number of complete sectors. Optimise this for performance.

8.2 Tools

These tools should be compiled as required.

`: FOR  \ -- ; n -- ; R: -- n'`
Starts a loop that is executed one or more times. At runtime, the loop is entered with the loop count on the stack. In the body of the loop the loop index is on the return stack and counts down from n-1 to zero. Use in the form:

```
( -- n ) FOR ... NEXT ( -- )
```

`: NEXT  \ -- ; R: n -- ; -- [n]`
Terminates a `[m ... m]` structure, discarding the loop index.

`: ucompare  \ a1 u1 a2 u2 -- notsame?`
Case insensitive string compare. Returns zero for a match.

`: ucmove  \ src dest len --`
As `CMOVE` but characters are converted to upper case.

`: N++  \ n1 x -- n1+1 x`
Increment *n1*.

`: byte-split  \ w -- lo hi`
Split a 16 bit item into its low and high bytes.

`: byte-join  \ lo hi -- w`
Convert two bytes into a 16 bit item.

`: word-split  \ x -- wlo whi`
Split a 32 bit item into its low and high 16 bit items.

`: word-join  \ wlo whi -- x`
Convert two 16 bit items into a 32 bit item.

`: qlog2  \ n -- log2[n]`
Given *n*, a power of two, return its power.
Error codes. Note that the Forth200x specification allows iors to be thrown, so these numbers should be unique.

```
#100 dup equ err_FATfiler       \ base of all FAT file system error codes
  dup equ err_FATread  3 +      \ base read error - 3 available, H/W specific
  dup equ err_FATwrite  3 +     \ write error - 3 available, H/W specific
  dup equ err_NoHandles  1+     \ not enough free file handles
  dup equ err_RootFull  1+      \ no room left in the root directory
  dup equ err_FATparams  1+     \ error in FAT parameters
  dup equ err_FileSize  1+      \ file size longer than 32-bit
  dup equ err_FileNotFound  1+  \ file was not found
  dup equ err_BadDir  1+        \ not a valid directory
  dup equ err_DirNotFound  1+   \ directory was not found
  dup equ err_w/o  1+           \ 14 = attempt to read W/O file
  dup equ err_r/o  1+           \ attempt to write R/O file or device
  dup equ err_BadPathName  1+   \ invalid created pathname
  dup equ err_BadFileName  1+   \ invalid created filename
  dup equ err_DirFull  1+       \ full directory
  dup equ err_NotDir  1+        \ invalid directory name
  dup equ err_BadPos  1+        \ Reposition beyond end of file
  dup equ err_NoDrive           \ Drive not present, e.g. card removed
drop
```

```
: time&date     0 0 0 1 1 1980  ;
```

If `TIME&DATE` is not available, a dummy value is returned.

8.3 Alignment checking

Some CPUs require aligned memory accesses. The code here checks alignment at run-time.

```
0 [if]  \ Do or do not test for unaligned memory access:
: @
  dup 3 and if
    .rs  1 abort" Unaligned 32-bit read"
  endif
  @ ;
: !   dup 3 and if  .rs 1 abort" Unaligned 32-bit write"  endif ! ;
: W@  dup 1 and if  .rs 1 abort" Unaligned 16-bit read"  endif w@ ;
: W!  dup 1 and if  .rs 1 abort" Unaligned 16-bit write" endif w! ;
[then]
```

8.4 Unaligned memory access

Systems that require aligned accesses for 16 and 32 bit values must provide unaligned access words. Similarly, big-endian CPUs must provide little-endian access as all the FAT data structure items are little-endian items.

```
: uaLEw@        \ caddr -- w
```

Unaligned 16 bit little-endian fetch.

```
: uaLE@         \ caddr -- x
```

Unaligned 32 bit little-endian fetch.

```
: uaLEw!        \ w caddr --
```

Unaligned 16 bit little-endian store.

```
: uaLE!          \ w caddr --
```
Unaligned 32 bit little-endian store.

8.5 Portability words

These words provide some portability for different CPU architectures.

```
: @P    @  ;    \ addr -- x
```
32 bit fetch from code space.

```
: W@P   w@  ;   \ addr -- w
```
16 bit fetch from code space.

```
: c@p   c@  ;   \ addr -- b
```
8 bit fetch from code space.

```
: @C    @  ;    \ addr -- x
```
Fetch 32 bits from code space (e.g. for tables).

```
: W@C   w@  ;   \ addr -- w
```
Fetch 16 bits from code space (e.g. for tables).

```
: c@C   c@  ;   \ addr -- b
```
Fetch 8 bits from code space (e.g. for tables).

9 Generic SPI driver for SD cards

The file *SDspi.fth* contains a generic driver for SD and MMC cards in SPI mode. The low level SPI interface should provide the following words:

```
MMCon     ( -- )       Drives CS low.
MMCoff    ( -- )       Drives CS high.
spiFlush  ( -- )       Flush SPI receive queue (if any).
spi!      ( c -- )     Write SPI, discard received byte.
spi@      ( -- c )     Write $FF byte, return received byte.
spiwr     ( a -- a' )  Reads next byte and writes it to SPI
spird     ( a -- a' )  Reads SPI and writes to next byte.
initSDspi ( -- )       Initialise SPI for SD/MMC use.
spiSlow   ( -- )       Set SPI clock to low speed
spiFast   ( -- )       Set SPI clock to high speed
```

`SPIRD` and `SPIWR` are expected to be fast. Their semantics were chosen to minimize threading overhead for target Forths that use threaded code.

Please note the following restrictions.

- Sector size must be 512 bytes. This will not be changed by MPE, so 2Gb standard capacity cards may not be supported, but SDHC cards (usually 4Gb and up) are supported.
- If the card contains a partition table, only the first partition is accessible.

For further details of SD and MMC cards see

`http://www.sdcard.org/about/memory_card/pls/`

`http://wolverine.caltech.edu/referenc/SDMMCv52.pdf`

The standard SD/MMC card pinout is as follows.

```
        SD mode                                  SPI mode
Pin     Name    Type    Description              Name    Type    Description
1       CD/DAT3 I/O/PP  Card detect / data 3     CS      I       Chip select (low)
2       CMD     PP      Command/Response line    DI      I       Data input
3       Vss1    S       Supply voltage (gnd)     VSS     S       Supply voltage
4       Vdd     S       Power supply             VDD     S       Power supply
5       CLK     I       Clock                    SCLK    I       Clock
6       Vss2    S       Supply voltage           VSS2    S       Supply voltage
7       DAT0    I/O/PP  data line 0              O       O/PP    Data output
8       DAT1    I/O/PP  data line 1              RSV
9       DAT2    I/O/PP  data line 2              RSV
```

where the pins are numbered on the connector as below:

```
9 1 2 3 4 5 6 7 8
  * * * * * * * *
*
```

9.1 Configuration

`1 equ MciSem?  \ -- x`
Set this non-zero to use a an exclusive access semaphore in the sector read/write routines `SecRead` and `SecWrite`. `MciSem?` can be defined before this file and will override this copy. This semaphore will be needed if the disc can be accessed by other sources such as USB.

9.2 SD/MMC functions

The SD error/throw codes are defined in terms of the `err_FATread` and `err_FATwrite` codes from *XC6harn.fth*.

```
err_FATread equ sd_read_fail            \ SD card failure codes
err_FATread 1+ equ sd_cmd_fail          \ command hung
err_FATwrite equ sd_write_fail          \ write error
err_FATwrite 1+ equ sd_reset_fail       \ failure to reset
```

`1000 equ sd_timeout  \ -- ms`
Timeout in milliseconds. Slower targets may want a smaller `SD_TIMEOUT` value, especially those whose SPI interface takes more than a millisecond per transfer. This usually only comes into play when using a bit-banged interface.

`: SD_idle  \ cnt --`
Send idle (0xFF) a specified number of times.

`: SD_sendArg  \ arg32 --`
Send *arg32* in big-endian format.

`: SD_prep  \ --`
Prepare to send an SD command.

`: SD_NB  \ -- res8|-1`
Wait until the card is not busy or times out, returning -1 for no response or the 8-bit response. In SPI mode, all commands generate at least one byte of response.

`: SD_send  \ arg32 cmd6 -- result8 ; -1=timeout, other = result`
Sends a command and tests for time-out, returning -1 for no response or the 8-bit response. In SPI mode, all commands generate at least one byte of response. The line is left with CS active.

`: +spi@  \ x -- x<<8+b`
Shift x 8 bits left, read the next SPI byte in a response and merge it.

`: SD_ReadResp32 \ -- x32`
Read four bytes in big-endian format.

`: SD_sync  \ --`
Get the card into a known state.

`0 value SD2card?  \ -- flag`
Returns true when the card follows SD v2 specification.

`0 value HCcard?`
Returns true when the card is a High Capacity (SDHC) card.

`: tryCMD0  \ -- ior`

Try 10 times at 100 ms intervals to reset the card, returning 0 for success.

`: CMD8 \ -- b`
Send command 8 for range 2.7-3.6 volts, and return 0 for a good response, a non-zero response byte for a bad command, or -1 for bad response data.

`: CheckInit \ sd2? -- ior`
Initialise the card. The parameter *sd2* is -1 for an SD2 card or 0 for a SD1 card. *Ior* is returned 0 for success. **N.B.** This operation may take several seconds.

`: SD_reset \ --`
Reset the card, and perform the initialisation sequence. On failure, an `sd_reset_fail throw` occurs.

`: SD_cmd \ arg32 cmd --`
A more robust version of `SD_send`.

`: SD_readstatus \ -- status`
Read the SD card status register.

`: SD_wait \ xt -- timeout?`
Wait for condition specified by *xt*, which has the stack effect `( -- flag)` where flag is returned non-zero to finish the operation before timeout.

`; equ SD_writedone? \ -- xt`
Gives the xt of a `:NONAME` word used with `SD_wait` while writing a data block.

`; equ SD_readready? \ -- xt`
Gives the xt of a `:NONAME` word used with `SD_wait` before reading a data block.

`: SD_waitread \ --`
Wait until ready to read.

`: SD_readcsd \ addr --`
Read the CSD - 16 bytes.

9.3 Sector read and write

The basic sector read and write routines use the simple API above. If your SPI unit supports DMA operation, you can use it for the 512 byte block transfers by providing the words

```
sdWriteBlk  \ asrc --
sdReadBlk   \ adest --
```

These routines write and read the 512 bytes of data to and from the SPI bus.

`: SD_addr \ blockno -- sdaddr`
Convert a sector number to an SD card address.

`: SD_writeblock \ asrc blockno --`
Write a 512-byte block from address *asrc* to block *blockno*.

`: SD_writeblock \ asrc blockno --`
Write a 512-byte block from address *asrc* to block *blockno*.

`: SD_readblock \ adest blockno --`
Read a 512-byte block to address *adest* from block *blockno*.

`: SD_readblock \ adest blockno --`
Read a 512-byte block to address *adest* from block *blockno*.

9.4 CSD handling

The card CSD is read during initialisation into `CSDbuff`. You can assume that the card supports a minimum block size of 512 bytes. The size information is contained in the following byte offsets in `CSDbuff`.

```
05  ----rrrr  Minimum read block size = 2^r bytes
06  ------ss
07  ssssssss  s = 12-bit size
08  ss------
09  ------mm  Size multiplier = 2^(m+2)
10  m-------
```

The number of 512-byte blocks in the SD card is s * 2^(m+2+r-9). Formatting involves writing a new boot record and clearing the FAT table. Searching for bad sectors is recommended but a brute force search will take a long time. Other data in the CSD includes write-protect status and access times. Card formatting is not yet supported as it is assumed that cards will be formatted on a PC.

`: GetBit \ caddr u -- 0/1`
Get bit number *u* from a bit array. Bit 0 is the top bit of the first byte. Bit 15 is the bottom bit of the second byte.

`: GetBits \ caddr start len -- x`
Get *len* bits starting at bit *start* from the bitmap starting at *caddr*.

`0 value #Sectors`
Number of sectors on the card.

`#16 buffer: CSDbuff \ -- addr`
Buffer to read CSD.

`: CheckCSD \ --`
Extract the CSD information, particularly the card capacity in sectors.

9.5 FAT file system API

The file system needs these words to access the mass storage:

`SecRead` (addr sector dev –) Reads a sector from the specified device. `THROW`s on error.

`SecWrite` (addr sector dev –) Writes a sector to the specified device. `THROW`s on error.

`MassInit` (–) Initializes mass storage access.

`MassTerm` (–) Terminates mass storage access.

To simplify the code, the raw read/write interface treats all read/write errors as fatal, and `THROW`s on error. Retries should be accommodated within the sector read/write code.

`Semaphore SecSem \ -- addr`
Exclusive access semaphore for sector read/write routines.

`: SecRead \ addr sector dev --`
Reads a sector from the specified device. `THROW`s on error.

`: SecWrite \ addr sector dev --`

Writes a sector to the specified device. `THROW`s on error.

`: MassInit \ --`
Initialize the mass storage manager. This is done when a new card is detected.

`: MassTerm \ --`
Shut down the mass storage manager. This is done when a card is removed.

9.6 Test code

`: .sector \ u --`
Read and dump a sector.

`0 [if]`
To compile the test code, change the conditional compilation in the file from `0 [if]` to `1 [if]`.

`: ParseCSD \ *csd --`
Parse the CSD structure and display the contents.

`: .CSD \ --`
Read and display the CSD contents.

Index

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

Z