PowerFile

Embedded File System

Microprocessor Engineering Limited

Published by Microprocessor Engineering

MPE PowerFile
User manual
Manual revision 2.00
8 March 2007

Software
Software version 2.00

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@mpeltd.demon.co.uk
tech-support@mpeltd.demon.co.uk
web: www.mpeltd.demon.co.uk

Table of Contents

1 PowerFile overview

1.1 Introduction

The *PowerFile* code provides a filing system for embedded applications. It can be used in conjunction with the *PowerNet* TCP/IP stack and web server, and with the PowerView embedded GUI. *PowerFile* was originally developed by CEM Systems Ltd. for use in networked access control systems. This release of *PowerFile* is distributed under license by MicroProcessor Engineering.

PowerFile is written for the MPE Forth cross compilers, and takes full advantage of the VFX code generator. The use of a 32 bit Forth is assumed. *PowerFile* provides:

• Simple configuration. Different applications can use different configurations held in a separate configuration file.
• Files wih the standard ANS Forth interface for code compatibility with other applications.
• Directories are nestable to a configured level.
• Command line tools
• Diagnostics
• Easy interface to a range of mass storage media, including USB memory sticks, CompactFlash, IDE/ATAPI drives and serial EEPROM or DataFlash.

1.2 Disk structure

The *PowerFile* data structures are loosely based on the pSOS pHILE system. All data items except for strings are 32 bit items known as elements.

The media surface is divided into blocks, which usually correspond to disk sectors. The size of these blocks vary depending on the media. Normally each block is 512 bytes long.

Disk blocks are allocated from a volume bitmap, which indicates whether or not the block is available for use. The bitmap is contained in a special file called *bitmap.sys*.

Each file on the disc is controlled by a File List Descriptor (FLDs). When these entries are read into RAM they are referred to as File Control Blocks (FCBs). All the FLDs are contained in a special file called *flist.sys*. Each FLD contains data that describes the file's location on the disk

Directories are files which contain records consisting of the file name and a record number in *flist.sys*. The current directory is called '.' and the next directory level down (closer to the root) is called '..'. The directory separator character is '/' by default. File and directory names have a maximum length of 12 characters including a count byte. A subdirectory is just a special file in a directory.

When the disc is initialised (formatted), the first few blocks are set up and a root directory and the special files are created. The special files are of fixed length.

```
Number     Item            Comment
0               Boot load 1     Not used
1          Boot load 2     Not used
2          Root block      Defines volume
3          Root Directory
4          Usage bit map   Defines used blocks on media
4 + X      File list descriptors  File layouts
4 + X + Y  Data area
```

```
Element Item               Comment
0       Bit map address    Location block of usage bit map
1       File list address  Location block of file lists
2       Data address       Location block of data.
3       Creation time      Date and time when created
4       Volume name
7       Volume size        Number of blocks
8       File list size     Number of file list descriptors
9       Validation key
```

1.2.1 Root Block

1.2.2 Directories

Each entry has a size of 16 bytes, so each block of 512 bytes will hold 32 file names.

```
Item         Size (bytes)       Comment
File number  4             Index to record in FLD blocks
Name         12            Count byte + 11 chars max.
```

1.2.3 Volume bit map

This table is a binary representation of the blocks on the media surface. When a block is defined as used, its state will be changed from a 0 to a 1. This has the file name of *bitmap.sys* and its size is fixed when the media is initialised, based on the size of the media device in use.

1.2.4 File list descriptor

Every file has a file list descriptor record associated with it. They are contained in the special file *flist.sys* whose size is fixed when the media is initialised. This size is determined by data in the configuration file *FSconfig.fth* from the maximum number of files and directories that can be held on the media. Each entry is 128 bytes long, and is copied into RAM when the file is opened.

```
Element  Item                  Description
0        Next FCB              RAM FCB only
1        File size (bytes)
2        File size (blocks)
3        File type             System, data, directory file
4        Modification time
5        Expansion size        Minimum size increment
6        Open occurrences      RAM FCB only
7        File list descriptor  Index in FLIST.SYS
8        Extent start 0        Block number of first data block
9        Extent size 0         Number of contiguous blocks used
10..27   Extents 1..9          For a total of 10 extents
28       Indirect block
29       Not used
30       Not used
31       Not used
```

1.3 Software data structures

Only the two most important structures are described here. All the others can be found in the source code file *STRUCTS.FTH*.

1.3.1 Open file table

Every task has a series of Open File Tables, one for every open file it is using. There is a limit to the number of files that a task may have open at the same time.

```
Element  Item
0        Read/write pointer
1        File control block pointer
2        File FLD number
```

1.3.2 File control block

Every open file has a File Control Block (FCB) associated with it. More than one task can have access to the same FCB. The system has a fixed number of FCBs, created at startup of the system. These contain a copy of the FLD for the particular open file to permit access to the data.

2 ANS File word set

This group of words provides the ANS file word set using the native file system facilities.

2.1 File access methods

These are ignored at present, but are provided for compatibility.

```
: bin           \ fam -- 'fam
```
Modify a file-access method to include BINARY.

```
1 constant r/o  \ -- fam
```
Specifiy a read-only file.

```
2 constant w/o  \ -- fam
```
Get Writeonly fam.

```
3 constant r/w  \ -- fam
```
Get ReadWrite fam.

2.2 Basics

```
: FSerr:        \ err# -- ; n -- 0|err#
```
The underlying file system primitives return zero on error. The children of this word return 0 on success or a given error code which can in turn be used as a throw code.

```
#-258 FSerr: ?err_open          \ n -- n'
#-412 FSerr: ?err_create        \ n -- n'
#-414 FSerr: ?err_read          \ n -- n'
#-415 FSerr: ?err_write         \ n -- n'
#-416 FSerr: ?err_close         \ n -- n'
#-417 FSerr: ?err_seek          \ n -- n'
#-418 FSerr: ?err_delete        \ n -- n'
#-419 equ    err_handle         \ -- n
#-420 equ    err_resize         \ -- n
```

```
: Open-File     \ c-addr u fam -- fileid ior
```
Create a file on disk, returning a 0 ior for success and a file id.

```
: Create-File   \ c-addr u fam -- fileid ior
```
Create a file on disk, returning a 0 ior for success and a file id.

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

```
: Read-File     \ caddr u fileid -- u2 ior
```
Read data from a file. The number of bytes actually read is returned as u2, and ior is returned 0 for a successful read. Note that all reads with zero length succeed.

```
: Write-File    \ caddr u fileid -- ior
```
Write a block of memory to a file. Note that all writes with zero length succeed.

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

```
: file-position \ fileid -- ud ior
```

Return file position, and return ior=0 on success.

```
: Reposition-File       \ ud fileid -- ior
```
Set file position, and return ior=0 on success.

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

```
: Resize-File   \ 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.

3 Command line tools

3.1 File and Directory display

`: tail_f { l_*name l_len | l_handle l_*buff l_key-wait -- }`
List/Type the given file in ASCII format. Suitable for ASCII files.

`: tail \ -- ; TAIL <filename>`
List/Type a file in ASCII format. Suitable for ASCII files.

`: pwd \ --`
Display the current working directory.

`: cd \ -- ; CD <name>`
Change directory to that given by the following text (if any). Then display the working directory.

`: 5.r \ n --`
Display n as 5 digits plus a space.

`: .ExtData \ addr --`
Display the extent data starting at **addr**.

`: show-file-details \ fcb --`
Displays extended file information from an FCB/FLD structure.

`: ls { | l_buff l_fld-buff -- }`
List the files in the working directory with a lot of diagnostic information.

3.2 Host to target file transfer

These tools are compiled if the Xmodem file transfer tools have been compiled. They allow files to be copied to and from the host over the `CONSOLE` (serial) link. The ANS file interface and the Xmodem code in *COMMON\XmodenTxRx.fth* are required. Host file selection must be performed by the host's communication tools. If you are using AIDE's PowerTerm tool, this is built in.

`0 value fh \ -- fileid`
temorary file handle

`: wf128 \ --`
Write 128 bytes from the host to the target file given by `FH`.

`: Host>File \ caddr len --`
Copy a host file to the named target file, which will be created using the name given by *caddr/len*.

`: rf128 \ --`
Read 128 bytes from the target file given by `FH`.

`: File>Host \ caddr len --`
Copy the named target file to the host.

4 Configuring the file system

The file <*FileSys*>*FSconfig.fth* contains the default configuration for the file system. To set your own configuration, copy <*FileSys*>*FSconfig.fth* and rename it. You can put it where you like, so placing it in your main application source folder is sensible. Then compile your file **before** *FileSys.bld*, which will ignore the default file if it finds a previously loaded configuration.

4.1 Diagnostics

```
1 equ FSdiags?  \ -- n
```
Set non-zero to compile diagnostic and test code.

4.2 Sector/Block size

The file system operates on fixed-size blocks on the disk. The size of each block is normally 512 bytes. It can be changed as required. If the block size is changed, you will need to redefine the compiler macros in this section. These macros use shift and logical operations on the assumption that these are faster than division operations. You can change these macros to suit your CPU architecture, or replace them by words if the block size is not a power of two.

```
#512 equ BLOCK_SIZE     \ -- u
```
Block size in bytes on disk.

```
: BSdiv         \ size -- #blocks
```
Given a length, return the number of complete blocks required. Equivalent to:

```
  BLOCK_SIZE /
```

```
: BSmod         \ size -- rem
```
Given a length, return the number of bytes in the last block. Equivalent to:

```
  BLOCK_SIZE mod
```

```
: BS/mod        \ size -- rem #blocks
```
Given a length, return the number of bytes in the last block and the number of complete blocks. Equivalent to:

```
  BLOCK_SIZE /mod
```

```
: BSblks        \ size -- #blocks
```
Given a size in bytes, return the number of blocks required to contain them. Unlike `BSdiv`, this allows for a partly used last block.

4.3 Disk information

These equates affect the proportion of the disk used for management and how much RAM is required.

4.4 Filing system

```
BLOCK_SIZE equ MIN_FILE_SIZE    \ -- u
```
Defines how much space is initially allocated when a file is created.

```
#32 equ MAX_FILES       \ -- u
```
Maximum number of files on the drive.

```
3 equ PATH-LIMIT        \ -- u
```

Maximum nesting of directory levels.

```
char / equ DIR-DELIMIT  \ -- char
```

Defines the character used as a directory name delimiter.

```
char . equ THIS-DIR     \ -- char
```

Defines the character used to mean "this directory".

```
#11 equ PATH-LENGTH     \ -- u
```

Maximum length of a file name.

4.5 Disk Cache

The disk can operate with a choice of caching modes, which provide trade-offs between disk performance, RAM usage, determinism and interrupt latency.

```
0 equ NoDiskCache       \ -- 0
```

No Disk Cache. All disk activity is to the drive. Lowest RAM usage and lowest performance. This may be the best solution for systems with limited RAM and drives with their own cache.

```
1 equ CloseDiskFlush    \ -- 1
```

The disk cache is only written back on demand by the word `SYNC-VOL`, or when a file is closed, or when a directory is created.

```
2 equ TaskDiskFlush     \ -- 3
```

A separate task `DiskTask` executes `SYNC-VOL` when required. When data is written out depends on `SyncMs` and `SyncOnIdle` below.

```
CloseDiskFlush equ DiskCacheMode        \ -- n
```

Define the required disk cache mode.

```
#4 equ #CACHE_BLOCKS    \ -- n
```

Number of disk blocks that are cached in RAM.

```
#250 equ SyncMs \ -- ms
```

Time in milliseconds after a write after which the disk cache will be flushed.

```
0 equ SyncOnIdle        \ -- n
```

If non-zero `SYNC-VOL` is executed `SyncMs` milliseconds after the last disk write, otherwise it is executed `SyncMs` milliseconds after the first disk write.

```
#15000 equ VOL_TIMEOUT  \ -- ms
```

Number of milliseconds after a write before the cache is flushed.

4.6 Disk information block

The file system was originally written to use CompactFlash cards or ATAPI drives. These return a disk identification block in little endian format. See *Specs\cfspec1_4.pdf* Table 40 for the gory details. This format is retained for all hardware, regardless of host CPU and attached hardware. At present, the only **required** data is the number of sectors on the disk.

```
#57 2* equ ID.#SECTORS  \ -- offset
```

Offset of the 32 bit number of sectors on the disk.

4.7 End of configuration

This must be the last section of the configuration file.

```
1 equ FSconfigured?     \ -- n
```

Must be defined non-zero after all configuration information has been defined.

5 Portability harness

The file *%FileSysDir%\Harness.fth* provides portability and architecture specific definitions for the file system.

5.1 Miscellaneous

```
: y/n?          \ -- flag
```
Wait for a key, echo it and return true if it was a 'Y'.

```
Synonym .hex .dword     \ n --
```
Display n as an 8 digit hex number.

```
: .decimal      ( n -- )        base @ swap  decimal  0 .r  base !  ;
```
Display in decimal.

```
2 field-type short      \ n -- n+2 ; addr -- addr+n
```
Defines a 2-byte field in a structure definition.

5.2 Little Endian memory operations

These words are usually implmented as compiler macros, and are provided for interchange of storage media between big and little endian media. They are also necessary when using CompactFlash, ATA or ATAPI drives to avoid endian and data alignment problems.

```
: c!(le)        \ b caddr --
```
Little endian byte store.

```
: w!(le)        \ w caddr --
```
Little endian 16 bit unaligned word store.

```
: l!(le)        \ x caddr --
```
Little endian 32 bit unaligned store.

```
: c@(le)        \ caddr -- b
```
Little endian byte fetch.

```
: w@(le)        \ caddr -- w
```
Little endian unaligned 16 bit word fetch.

```
: l@(le)        \ caddr -- x
```
Little endian unaligned 32 bit word fetch.


```
: w!(le)        \ w caddr --
```
Little endian 16 bit unaligned word store.

```
: l!(le)        \ x caddr --
```
Little endian 32 bit unaligned store.

```
: c@(le)        \ caddr -- b
```
Little endian byte fetch.

```
: w@(le)        \ caddr -- w
```
Little endian unaligned 16 bit word fetch.

```
: l@(le)        \ caddr -- x
```
Little endian unaligned 32 bit word fetch.

5.3 File system data

`variable lseconds \ -- addr`
Holds UNIX LSECONDS since ...

`cell +user DIR-BLOCK \ -- addr`
Pointer to a directory block.

`cell +user *CWD \ -- addr`
Pointer to current work directory name structure.

`semaphore drive-access \ -- addr`
Interlock for access to the low level disk driver.

`semaphore cache-sema \ -- addr`
Interlock for access to the disk cache software.

`semaphore drive-control \ -- addr`
Interlock for access to the drive management software.

`semaphore fld-update`
Interlock for access to the file management software.

`: init-sema \ sema --`
Initialise a counted semaphore.

5.4 Heap

`: allocate \ u -- addr ior`
A redefinition of `ALLOCATE` that issues a console warning message if it fails.

`: free \ addr -- ior`
A redefinition of `ALLOCATE` that issues a console warning message if it fails.

`: ?free \ addr --`
A convenvenient version of `FREE` that does nothing if addr is zero and does not return an error code.

5.5 Strings

`: extract-string \ caddr len char -- caddr' len'`
Given a string return the substring between the delimiters, ignoring leading delimiters.

`: jump-string \ caddr len char -- caddr' len'`
Ignore leading delimiters, and return the string to the right of the delimiter character including the delimiter.

6 File system structures

Structs.fth defines the data structures that are used in the file system.

```
  equ PATH-STRING       \ -- len
```

Maximum size of a full path name.

```
PATH-LENGTH 1+ field-type lstring       \ len n -- len+n ; addr -- addr+len
```

Filed type for counted strings.

```
struct dir-struct       \ -- len
```

Directory entry in a directory block.

```
  int      ->file-number        \ the FLD record number
  lstring  ->file-name          \ file name as counted string
end-struct
```

```
BLOCK_SIZE DIR-STRUCT / equ DIR/BLOCK   \ -- n
```

Number of directory entries per block.

```
struct file-extents     \ -- len
```

Each extent has the same format.

```
  int  ->extent-start           \ location of data
  int  ->extent-size            \ size of data
end-struct
```

```
NO-OF-EXTENTS file-extents * equ EXTENT_LIST    \ -- len
```

Size of the extent area in an FCB/FLD.

```
struct FLD-rec  \ -- len
```

FILE LIST DESCRIPTOR and FILE CONTROL BLOCK

```
  int  ->next-fcb               \ FCB, points to next FCB
  int  ->file-size              \ number of bytes used in file
  int  ->file-blocks            \ number of blocks used by file
  short  ->prev-dir
  short  ->file-type            \ data, directory or system
  int  ->file-last-mod          \ data last written
  int  ->file-expansion         \ size of increase
  int  ->open-tables            \ FCB, current number of open tables
  int  ->fld-no                 \ FCB, ptr to FLD
  int  ->extent-start-0         \ location of data
  int  ->extent-size-0          \ size of data
  EXTENT_LIST file-extents - +  \ remaining extents
  int  ->indirect-block         \ points to indirect block.
  3 cells +                     \ currently unused
end-struct
```

```
BLOCK_SIZE FLD-REC / equ FLD/BLOCK      \ -- len
```

Number of FLDs in a disk block.

```
struct indir-extent     \ -- len
```

Indirect extents are not the same as direct extents.

```
    int  ->ind-extent-start     \ start sector
    int  ->ind-logical-end      \ last+1
end-struct
```

```
struct open-table-struct        \ -- len
```
Defines details of a file that a task has open.
```
    int  ->next-otb             \ chain of OTBs
    int  ->r/w-data-ptr         \ read/write pointer
    int  ->valid-otb            \ marker for validity check
    int  ->fcb-number           \ FCB address
end-struct
```

```
$5a1234a5 equ CEM_FORMAT        \ -- x
```
A code to show its DOS type.
```
struct root-block       \ -- len
```
Layout of the start of the root block.
```
    int ->CEM_FORMAT            \ file type indicator
    int ->USAGE_BLOCK0          \ start of bitmap
    int ->FLD0_BLOCK            \ start of FLDs
    int ->DATA_BLOCK0           \ first data block
    int ->init-time             \ disk format time
    lstring ->vol-name          \ volume name
    int ->vol-size              \ max blocks
    int ->fd-size               \ unused
end-struct
```

```
struct boot-block       \ -- len
```
Layout of start of boot block.
```
    int ->primary-boot          \ location of primary application
    int ->primary-size          \ size
    int ->backup-boot           \ location of secondary application
    int ->backup-size           \ size
    int ->CEM_FORMAT-B          \ marker
end-struct
```

```
struct block-alloc      \ -- len
```
This structure is used when grabbing or freeing disk sectors.
```
    int ->last-amount
    int ->#blocks
    int ->start-block
    int ->start-address
    int ->first-amount
end-struct
```

```
struct file-extension   \ -- len
```
This structure is used when extending a file.
```
    int ->ext-fid
    int ->ext-size
```

```
    int ->ext-start
    int ->ext-no
end-struct

struct CACHE-INFO       \ -- len
```
Data structure controlling each cached disk sector.
```
  int ->cache-link      \ link to next structure in chain; MUST be first
  int ->mem-block       \ pointer to data
  int ->disk-block      \ disk block number
  int ->cache-valid     \ nz = cache valid
  int ->cache-time      \ last read/write time
  int ->cache-dirty     \ nz = modified
end-struct
```

7 24C512 Serial EEPROM Disk

This file requires the I2C driver:

```
Examples\I2C\Devices\AT24C512drv.fth
```

Assuming that a single device is used gives a disk size of 64 kbytes.

Disk sectors are referenced by a block number, 0..DISK_SIZE-1.

Other example disk drivers may be found in the folder *%FileSysDir%\Drivers*.

```
#64 kb BLOCK_SIZE / equ #SECTORS/DISK   \ -- n
```
Number of sectors per disk.

```
: show-drive-details    \ addr --
```
Display drive details from given data buffer

```
: (drive-id)    \ -- buff
```
Allocates a 512 byte structure that includes the disk size in sectors. For IDE and CompactFlash drives, this word should use the IDE-ID ($EC) command.

```
: write-block   \ blk# caddr -- flag ; 0=bad
```
Write the buffer at **caddr** to the disk sector **blk#**.

```
: read-f-block  \ blk# caddr -- t/f ; 0=bad
```
Read the disk sector **blk#** into the buffer at **caddr**.

```
: erase-block   \ blk# -- t/f
```
Erase (set to zero) the given disk block.

```
: InitDisk      \ --
```
Initialise disk access hardware

8 Disk Cache

The disk cache reduces the number of accesses made to the physical drive.

```
struct CACHE-INFO       \ -- len
```
Data structure controlling each cached disk sector.

```
  int ->cache-link      \ link to next structure in chain; MUST be first
  int ->mem-block       \ pointer to data
  int ->disk-block      \ disk block number
  int ->cache-valid     \ nz = cache valid
  int ->cache-time      \ last read/write time
  int ->cache-dirty     \ nz = modified
end-struct
```

```
CACHE_SIZE buffer: cache-memory \ -- addr
```
The cache data blocks.

```
CACHE_HDR_SIZE buffer: cache-hdr-table  \ -- addr
```
The cache lookup table.

```
variable active-c-block \ -- addr
```
Anchors active block chain.

```
variable free-c-blocks  \ -- addr
```
Anchors free block chain.

```
: init-cache    \ --
```
Initialise the disk cache.

```
: release-cache \ struct --
```
Put the given cache structure back onto the free chain. Exit quietly if the structure cannot be found.

```
: release-all-cache     \ --
```
Put all cache structures back on the free chain.

```
: (get-cache)   \ -- struct|0
```
Get a new cache structure, or return 0 if one cannot be found.

```
: get-oldest-cache      \ struct --
```
If the cache item is dirty (needs to be written back), write it back and mark it as unused.

```
: get-cache     \ ?struct -- struct
```
If there is a free cache block, discard the given one and use a free one, otherwise reuse it.

```
: blk-in-cache? \ blk# -- struct t/f
```
If the given block is in the disk cache, return its cache structure and true, otherwise return the oldest cache structure and false.

```
: read-blk>c    \ blk struct -- cache
```
Read the given block into cache, returning the actual cache structure used, or zero if no cache is available.

```
: (read-blk)    \ blk -- struct
```
If the block is already in cache, return its cache structure, otherwise read it into cache, returning the cache structure used or zero for failure.

`: read-p-blk  \ lba buff start size -- t/f`
Read a partial block/sector into the given buffer. Return true for success.

`: read-f-blk  \ lba buff -- t/f`
Read a full block/sector into the given buffer. Return true for success.

`: write-p-blk  \ blk buffer start end -- t/f`
Write a partial block/sector from the given buffer. Return true for success.

`: write-f-blk  \ lba buff -- t/f`
Does a full block write to disk, replaces `WRITE-BLOCK`. Return true for success.

`: sync_vol  \ --`
Synchronise the cache and the disk by writing out any modified blocks.

`: time_sync_vol \ --`
Write out any modified cache blocks that have been in cache loger than `VOL_TIMEOUT` milliseconds.

8.1 Cache diagnostics

This code is only compiled if the equate `FSdiags?` is set non-zero.

`: (show-cache)  \ anchor --`
Display the cache blocks in a particular chain.

`: show-cache  \ --`
Display the cache blocks in the active chain.

`: show-f-cache  \ --`
Display the cache blocks in the free chain.

9 Intermediate file I/O functions

`: read-d-block  \ block buffer b-start size -- t/f`
Read a full or partial disk block from the drive or the cache. A return value of zero 0 indicates an error.

`: write-d-block \ block buffer b-start b-end -- t/f }`
Write a full or partial disk block to the drive or the cache. A return value of zero 0 indicates an error.

`: read-hd-block          \ block buffer -- t/f`
Read a full disk block from the drive or the cache. A return value of zero 0 indicates an error.

`: write-hd-block         \ block buffer -- t/f`
Write a full disk block to the drive or the cache. A return value of zero 0 indicates an error.

10 Sector Bitmap

These functions are used for indicating disk usage. Space is indicated by a bitmap where one bit represents one block (512 bytes) on the media. When a block is in use, the appropiate bit in the BITMAP table will be set to one. For groups of 32 or more blocks, these groups will always start on a multiple of 32 blocks.

The bitmap is set up so that block 0 is indicated by the top bit of the first byte.

```
variable p-blocks       \ -- addr
```
Holds groups of less than 32 blocks required.

```
variable no-32          \ -- addr
```
Holds number of 32 block groups required.

```
variable b-ptr          \ -- addr
```
Holds disk buffer search pointer.

```
variable grab/free      \ -- addr
```
Holds true for grab, false for release

```
BLOCK_SIZE 8 * equ BLOCKS/BIT_TABLE     \ -- n
```
Number of bits (blocks) per block.

```
BLOCKS/BIT_TABLE equ BLOCKS/GROUP       \ -- n
```
Synonym.

```
$FFFFFFFF equ 32BLOCKS  \ -- mask
```
Indicator for 32 blocks.

```
: bitmask       \ n -- mask
```
Creates an n-bit mask in the low bits.

```
: topBitmask    \ n -- pattern
```
Creates an n-bit mask starting at the highest bit.
- 1 bit is $80000000
- 4 bits is $f0000000
- 16 bits is $ffff0000

```
: (find-partial-block) { l_data l_size | l_block-no -- start-block-no }
```
Finds partial block of SIZE anywhere in 32 blocks. Returns -1 if it won't fit.

```
: find-<32blocks        \ -- start|-1
```
Search the current bitmap block for the number of blocks held in variable `P-BLOCK`, returning -1 if NOT FOUND, otherwise returning the start block.
- no find of continue => -1 resets "start block" pointer, continue search
- no find of small block => -1 continue search
- find of continue => 0 all found and done
- find of small block => ptr all found done, set "start block" pointer

```
: find-free-blocks      { l_req-blocks | l_start-block l_32-found -- start-block }
```
Find the required number of blocks, returning the start block, or -1 if space cannot be found.

```
: block-order   { l_start l_amount | l_buff -- buffer }
```

Set up the data structure required to grab or release a number of blocks starting at a given block.

`: adjust-blk-addr \ addr1 -- addr2`
COMPILER: Convert a block number to address a cell containing its indicator bit. Equivalent to:

```
#32 / cells
```

`: 32mod \ n -- mod(n,32)`
COMPILER: Given a block number, return the bit number of its indicator bit in a cell. Equivalent to:

```
#32 mod
```

`: take/free \ data mask -- data'`
Apply the bit mask to grab or free the required blocks in **data**.

`: take/free32 \ -- mask`
Return the data value to grab/free 32 blocks.

`: grab-first-block \ start-block start-address first-amount --`
Using the starting bitmap block **start-block**, grab/free **first-amount** blocks from **start-address** onwards.

`: grab-32complete-blocks \ block --`
Grab/free the complete block of the usage bit map.

`: grab-last-block \ last-amount last-block --`
Grab/free the given amount in the last required block of the usage bitmap.

`: allocate-blocks { l_buff -- }`
Using the data structure created by `BLOCK-ORDER`, grab/free the blocks and `FREE` the buffer.

`: need-blocks \ amount -- start`
This attempts to find a contigous area of required space and allocates it. **Amount** is the mumber of blocks required. **Start** is the starting block number on the media, or -1 if space cannot be found.

`: free-blocks \ start amount --`
This will release the required number of blocks from the start point.

10.1 Diagnostics

`: d-fbm { | l_*buff -- }`
Display the disk bitmap.

11 File Control Blocks

11.1 Variables and buffers

```
SYS_NFCB FLD-REC * equ FCB-SIZE \ -- len
```
Size of FCB table.

```
FCB-SIZE buffer: FCB-TABLE      \ -- addr
```
RAM table containing FCBs loaded from disk FLDs.

```
TASK_NCFILE open-table-struct * equ OPENT-SIZE  \ -- len
```
Size of open file table.

```
OPENT-SIZE buffer: OPEN-TABLE   \ -- addr
```
RAM table of open file structures.

```
FLD-REC cell - equ FCB_DATA_REC \ -- len
```
Size of data in an FCB/FLD. The first element in the structure must be preserved as it is the link field.

```
FLD-REC buffer: cdir-buffer     \ -- addr
```
Scratch FCB/FLD buffer.

```
variable active-fcbs    \ -- addr
```
Anchors chain of FCBs in use.

```
variable free-fcbs      \ -- addr
```
Anchors chain of unused FCBs.

```
variable active-otbs    \ -- addr
```
Anchors chain of open file structures in use.

```
variable free-otbs      \ -- addr
```
Anchors chain of unused open file structures.

11.2 Read/Write FLDs

```
: fld-d-block   \ fld -- blk
```
Returns disk block of required FLD.

```
: fld-m-block   \ fld -- offset
```
Returns block offest of required FLD

```
: rewrite-fld   { l_fld l_buff -- t/f }
```
Update specific FLD to disk via cache

```
: read-in-fld { l_fld l_buff | -- t/f }
```
Gead in specific FLD from disk via cache

11.3 FCB primitives

```
: existing-fcb? \ fld# -- struct|0
```
If the FLD number is already in RAM, return its address, otherwise return zero.

```
: get-fcb       \ -- struct|0
```
Get a new FCB structure and return its address or 0 if no free FCBs are available.

```
: create-fcb    { l_fld | l_fcb -- FCB|0 }
```
Return file control block address if successful, otherwise return 0.

`: release-fcb  \ struct --`
Put an FCB structure back on the free chain; exit quietly if the FCB is not on the active chain.

`: dec-fcb-open  \ fcb -- t/f`
Decrement the number of open instances of this FCB. Returns true if all access closed, i.e. no instance is still open.

`: init-fcbs  \ --`
Initialise the table of FCBs.

11.4 Open Table Blocks (OTBs)

Because a file may have been opened more than once, some data must be held for each open instance. An Open Table Block holds the instance data and the address of the FCB for the file. The OTB address is used as handle (FID) of the file.

`: init-otbs  \ --`
Initialise the OTB structures.

`: create-otb  \ -- fid|0`
Get a free OTB and return its address or 0 if one cannot be found.

`: release-otb  \ fid --`
Put the OTB back on the free chain if it is active, otherwise do nothing.

`: fcb-addr  \ fid -- fcb`
COMPILER: Returns the FCB address from the FID.

`: fid-file-size \ fid -- size`
COMPILER: Returns the file size given the FID.

`: set-f-r/w-ptr \ value fid --`
COMPILER: Set the file read/write pointer.

`: get-f-r/w-ptr \ fid -- value`
Get the file read/write pointer.

`: get-r/w-b-ptr \ fid -- block`
Get the n-th block in the file which includes the data at the read/write pointer.

`: get-r/w-pb-ptr  \ fid -- offset`
Get the offset in the block in the file which includes the data at the read/write pointer.

`: inc-f-r/w-ptr \ step fid --`
Add **step** to the file's read/write pointer.

`: r/w-ptr-inrange?  \ fid -- t/f`
Return true if the file read/write pointer is within the number of allocated blocks.

`: r-ptr-inrange?  \ fid -- t/f`
Return true if the file read/write pointer is within the current file size.

12 Filename parsing

`: get-path \ *name len -- *name2 len2`
Return the string up to (to the left of) the next directory separator.

`: get-next-path \ *name len -- next-addr next-len`
Return the string to the right of the first path, including any director separator.

`: from-root? \ *name len -- t/f`
Returns true if the first character of the string is a directory separator.

`: get-fid { l_*name l_q-len l_dir-buff | l_f-num l_wild -- file-num|0 }`
Look for file name in a directory block buffer. If the file is found return its index, otherwise return 0.

`: find-file? { l_dir-blk l_*name l_len | l_*buff -- fn|0 }`
Look for file name in a directory block. If the file is found return its index, otherwise return 0.

`: no-of-paths \ caddr len -- n`
Count the number of directory levels in file name.

`: set-default-dir \ --`
Set the working directory to the root directory.

`: step-next-dir \ *name len buff -- *name1 len1 dir-blk`
Step to the next directory in the pathname (using the given directory block buffer), make that the current directory, and return the remaining path name and directory block number.

`: get_fn { l_*name l_len | l_dir l_fld l_*fld -- fld-num|0 }`
Given a path/file name, return the file index number if found or zero if not found. Directory path numbers are negated.

`: find-next-fld { | l_fld l_*buff -- fld-no }`
Find the next free FLD number, returning -1 if there is no more space, or 0 if buffer memory is exhausted.

`: inc-str-len \ n --`
COMPILER: increase size of working directory string.

`: add-delimit-str \ --`
Add a directory separator to the working directory string.

`: clr-dir-str \ --`
Clear the working directory string.

`: append-dir-str \ caddr len --`
Append the string to the working directory string.

`: move-back-dir \ --`
Move back one level in the working directory string.

`: back-dir? \ *name -- t/f`
Return true if the start of the name is '..'.

`: remove-dir-hdr \ *name len -- '*name 'len`
Remove a directory header from the name.

13 Native API

The file *%FileSysDir%\file-main.fth* builds the native API of the file system in preparation for the ANS layer.

File identifiers (shows as FID) are the OTB address when files are open.

`: open_f#  \ fn -- fid|0`
Given a file number, open it and return the FID address or 0 if the operation fails.

`: open_f  \ caddr len -- fid|0`
Open the named file and return its FID or zero if the operation fails.

`: close_f  \ fid -- code`
Close the file, returning: 0 on error; -1 if fully closed; or -2 if there is still an open instance.

`: file-not-exist?  \ caddr len -- t/f`
COMPILER: Return true if the file does not exist.

`: splLt-path&file  { l_*name l_len | l_*tname l_tlen -- name len }`
Split Dir path and filename, returning the file name part.

`: (adjust-blocks)  \ len -- blocks`
COMPILER: Convert a length to the minimum number of blocks required to hold that much data.

`: adjust-blocks  \ len -- blocks`
Convert a length to the minimum number of blocks required to hold that much data. Always returns at least one.

`: required-blocks  \ len fid -- blocks`
Returns the number of blocks required to write len bytes from the current file position.

`: need-dir-$buffer  \ --`
`ALLOCATE` a working directory buffer if one has not already been allocated.

`: (ch-dir)  \ caddr len -- t/f`
Switch to the given working directory and return true. If the change cannot be made, set the default directory and return 0.

`: ch-dir  \ caddr len -- t/f`
Switch to the given working directory and return true. If the change cannot be made, set the default directory and return 0.

`: add-to-dir  { l_fld l_*name l_len l_block | l_success -- flag }`
Add the given file number and name to the given directory block, returning true for success.

`: remove-file-dir  { l_fld l_block | l_success -- flag }`
Remove the given file number from the given directory block, returning true for success.

`: (create_f)  { l_*name l_len l_size l_extent | l_start l_*buff -- t/f }`
In the current directory create a file with the given name, initial size and extension. Return true for success. The file is not opened.

`: create_f  { l_*name l_len l_size_b l_extent_b | l_*tname l_tlen -- t/f }`
Create a file with the given name, initial size and extension. The file name may include existing directories. The file is not opened. Return true for success.

`: (create_d)  { l_*name l_len | l_start l_fld l_prev-fld l_*buff -- t/f }`
In the current directory create a subdirecory with the given name. Return true for success.

`: create_d  \ caddr len -- t/f`
Create a directory with the given name. The name may include existing directories. Return true for success.

`: rename_f  { *old-name old-len *new-name new-len | l_fld -- t/f }`
Rename a file from the old name to the new name. Return true for success.

`: (delete_f)  { l_fn | l_*buff l_size -- t/f }`
Delete the given file number. Return true for success.

`: delete_f  \ name len -- t/f`
Delete the given file by name. Return true for success.

`: delete-files  { l_*name l_len -- t/f }`
Delete a group of files if the first character of the name is '*', otherwise delete a single file. Return true for success.

`: lseek_f  \ ptr fid -- t/f`
Seek to the given location in the file. Return true for success.

`PTR=0` set to start

`PTR=-1` set to end of written data

`PTR=+ve` from start of file

`PTR=-ve` from end of data

`: file-read-area  { l_fid l_len | l_*c-pos -- b-start size }`
Caculate how much can be read from the current position with a single block read.

`: r/w-block  { l_fid | l_hd-block l_block l_fcb -- phys-blk }`
Return the block number containing the current file read/write pointer location. Return -1 on error or -2 if the pointer is beyond the end of file.

`: read_f  { l_buf l_len l_fid | l_read-size l_start-buff -- len }`
Read len bytes into buff from the given file, returning the number of bytes actually written.

`: update-size&time  { l_fid | l_size l_fcb -- t/f }`
Update the file size and modification time.

`: write_f  { l_buf l_len l_fid | l_size l_start l_req-blk -- len }`
Write len bytes from buff from the given file, returning the number of bytes actually written.

`: zero-file  \ l_fid -- t/f`
Reduce the file to zero size without changing the disk allocation. Return true for success.

`: InitFileTask  \ --`
Before using the file system, each task must run this word.

`: doDiskTask  \ --`
The action of the disk service task.

`Task DiskTask  \ -- addr`
The disk service task.

`: InitFileSys  \ --`
Before using the file system, this word must be run at power up.

14 File system diagnostic tools

The diagnostic code requires the file FSTOOLS.FTH, which is compiled if the equate `FSdiags?` is set non-zero.

`: df-v \ --`
Display disk information. N.B. The free space information is incorrect for small drives.

`: list-fcb \ --`
Display all open files.

`: list-otb \ --`
Display all open file handle details.

`: fdump { l_*name | l_handle l_*buff -- }`
Dump (memory display) the file whose name (counted string) is given.

`: list { | l_buff l_fld-buff -- }`
Display a directory listing in UNIX format.

Index

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Y

Z