



DataFlash® File System Reference Manual

Version 1.1

Copyright © 1999 – 2003 Atmel Corporation All Rights Reserved
2125 Orchard Parkway
San Jose, CA 95131
Phone (408) 441-0311

Table of Contents

CHAPTER 1 INTRODUCTION	4
OVERVIEW OF FAT FILE SYSTEM	4
<i>Boot Sector and BPB</i>	4
<i>The FAT</i>	5
<i>Root Directory</i>	5
<i>Data</i>	5
NON-SUPPORTED FEATURES	5
<i>Partitioning</i>	6
<i>FAT32</i>	6
<i>Long Filenames</i>	6
FLASH RELIABILITY ENHANCEMENTS	6
<i>Run-Time Errors</i>	6
<i>Data Caching</i>	7
<i>Wear Leveling</i>	7
<i>Error Correction</i>	8
CHAPTER 2 PRODUCT PERSPECTIVE	9
OVERVIEW OF THE MODEL	9
DEVICE MODEL	10
<i>Driver Object</i>	10
<i>Device Object</i>	11
DISPATCH INTERFACE	12
<i>IO_DISPATCH Method</i>	12
Parameters:	12
Return status:	12
Prototype:	12
Example:	13
<i>I/O Packet</i>	13
DEVICE DRIVER TABLE	14
Device Driver Identification Section	14
Driver Table	14
DISCLAIMER	15
CHAPTER 3 SYSTEM REQUIREMENTS.....	16
PLATFORM COMPATIBILITY	16
<i>Memory Management</i>	16
<i>Character Functions</i>	16
<i>String Management</i>	17
<i>Time Management</i>	17
<i>Synchronization Objects</i>	18
C LANGUAGE CROSS REFERENCE	19
<i>Memory Management</i>	19
<i>Character Functions</i>	19
<i>String Management</i>	20
<i>Time Management</i>	20
<i>Synchronization</i>	21
MEMORY REQUIREMENTS	21
CHAPTER 4 FILE SYSTEM API	22

FILE SYSTEM METHODS	22
<i>Volume Specific Operations</i>	22
<i>Directory Specific Operations</i>	23
<i>File Specific Operations</i>	23
<i>Miscellaneous (Support) Operations</i>	23
STRUCTURES	24
DIRECT	24
STAT	24
TYPES	25
SYSTEM PROGRAMMER'S INTERFACE	26
sys_close()	26
sys_closedir()	27
sys_creat()	28
sys_devstats()	29
sys_eof()	30
sys_format()	31
sys_lseek()	32
sys_mkdir()	33
sys_mount()	34
sys_open()	35
sys_opendir()	37
sys_read()	38
sys_readdir()	39
sys_remove()	40
sys_rename()	41
sys_rewinddir()	42
sys_rmdir()	43
sys_stat()	44
sys_tell()	45
sys_umount()	46
sys_unlink()	47
sys_write()	48
CHAPTER 5 I/O MANAGER	49
I/O TABLE	49
IOEntry	49
INITIALIZATION	50
FILE CONTROL BLOCK	51
Implementation	51
CHAPTER 6 FILE SYSTEM DRIVER	53
FSD OBJECTS	53
FAT Device Extension	53
File Object	54
FAT File Extension	54
DISPATCH OPERATIONS	55
DEVICE_IO_OPEN	55
DEVICE_IO_CLOSE	55
DEVICE_IO_READ	56
DEVICE_IO_WRITE	56
DEVICE_IO_CONTROL	56
mount	56
unmount	57
format	57
lseek	57
rename	57
mkdir	57
rmdir	58
readdir	58
filestat	58
CHAPTER 7 INTEGRATION AND USAGE TIPS	59
ROOT DIRECTORY MANAGEMENT	59
FAT DETERMINATION	59
Initialization	60
APPENDIX A MISCELLANEOUS	61
COMPATIBLE <i>ERRNO</i> CODES	61

DEVSTATS.....	63
<i>BLOCK_DEVSTATS</i>	63
<i>CHAR_DEVSTATS</i>	64
EXCEPTION HANDLING.....	64
<i>FUNCTIONALITY</i>	64
<i>DEFINED EXCEPTIONS</i>	65
<i>Example</i>	65
APPENDIX B MEMORY REQUIREMENTS	66
APPENDIX C EXAMPLES.....	68
DOING A UNIX “LS” (DIRECTORY LISTING).....	68
LOCALTIME	72
INDEX	76

Chapter 1

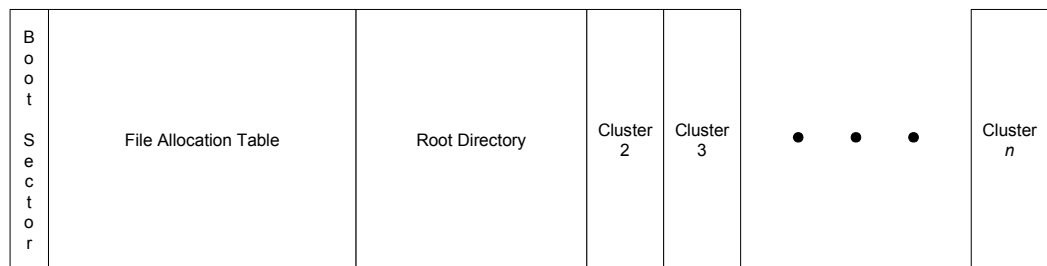
Introduction

The Atmel DataFlash File System is a FAT12/FAT16 Design and targets Atmel Flash EEPROM devices. It delivers a layered driver architecture using a simple binding model allowing for OS-agnostic portability.

Overview of FAT File System

Developed by Microsoft in the early 1980s, FAT, File Allocation Table, is a file system used to keep track of file fragments on a storage media. These fragments are called file clusters and a file is composed of one or more of these clusters up to a given maximum (typically 2 Gigabytes). Each cluster is composed of one or more smaller segments called sectors, which generally adhere to the default write block size of the part. The Atmel driver defaults to 512 bytes since most applications worldwide hardwire this value internally.

The Volume is divided up into 4 regions: Boot, FAT, Root, and Data.



Boot Sector and BPB

Located at the first sector of the file volume. It does not necessarily mean that it exists at the very front of the device, since this portion of the device may be reserved for other use. The Boot Sector contains Metadata that is important in determining the size and type of the volume, i.e. FAT12 or FAT16 (FAT32 is not supported, as will be explained shortly).

If at anytime the Boot Sector becomes corrupted, then the file system is rendered useless and all data shall be lost.

The FAT

The next data structure, immediately following the Boot Sector, is the File Allocation Table or FAT as it is better known. The FAT defines all of the clusters within the Data Region of the file system. Each entry in the table is an index to a corresponding cluster and the contents of that entry specify whether the cluster is free, or is an end of file, or a link to the next cluster of the file or directory (which is a special file with the DIR attribute set). When a cluster cannot be written to or read from, the cluster is marked as BAD and becomes unusable for the lifetime of the volume.

The size of each entry is dependent upon the format. FAT12 have 12 bit entries, FAT16, 16 bit entries, and so on.

Corruption of the FAT can result in lost cluster chains, resulting in corrupted files and/or directories. This eventually could render the file volume useless.

Root Directory

On FAT12 and FAT16, the Root Directory immediately follows the FAT. For FAT32, the Root resides in the Data Region. Because of this, clusters are not defined for the Root Directory in FAT12 and FAT16 and the size is fixed, typically no larger than 512 entries, with the exception of very large FAT16 volumes (256MB and 512MB). In reality, since an end marker is present, there is always one less entry, since the last entry is always the end of directory.

Data

This is the region that hosts all of the cluster chains associated with files and directories. As mentioned, directories are a special file with the DIR attribute set.

Non-Supported Features

Due to the tight coupling nature of the File System portion with the Data Flash, certain options are not supported.

Partitioning

Partitioning is more suitable for movable media such as a Floppy or Hard Disk. The device driver for the Flash Device determines what part of the Flash part is accessible by the File System Driver.

FAT32

Volumes that use FAT32 as the primary format, must be greater than 512Mbytes in size in order to be compliant with Microsoft's standards. FAT32 is not supported simply because there are currently no parts capable of this yet, i.e. 4 Megabit. However, FAT16 is only capable of handling volumes less than 4 Gigabytes in size and that comes with a few adjustments to the size of the ROOT directory.

Long Filenames

The current version of the FFS only supports the traditional 8.3 naming convention (alternate naming convention when dealing with volumes that support long filenames). Because of the overhead involved and resource limitations of embedded environments and especially with FAT12 volumes, long filenames are not included with this release of the FFS.

Flash Reliability Enhancements

Unlike a hard-drive or ram storage media, a flash device may become worn and induce errors in stored data over time. To effectively manage this and to extend the reliability of the end application, the Fusion File System has incorporated several features that greatly extend end user product reliability. These enhancements include strategic caching of key data, wear leveling of erasable flash blocks, dynamic tracking of bad blocks and user callable error correction functions.

Run-Time Errors

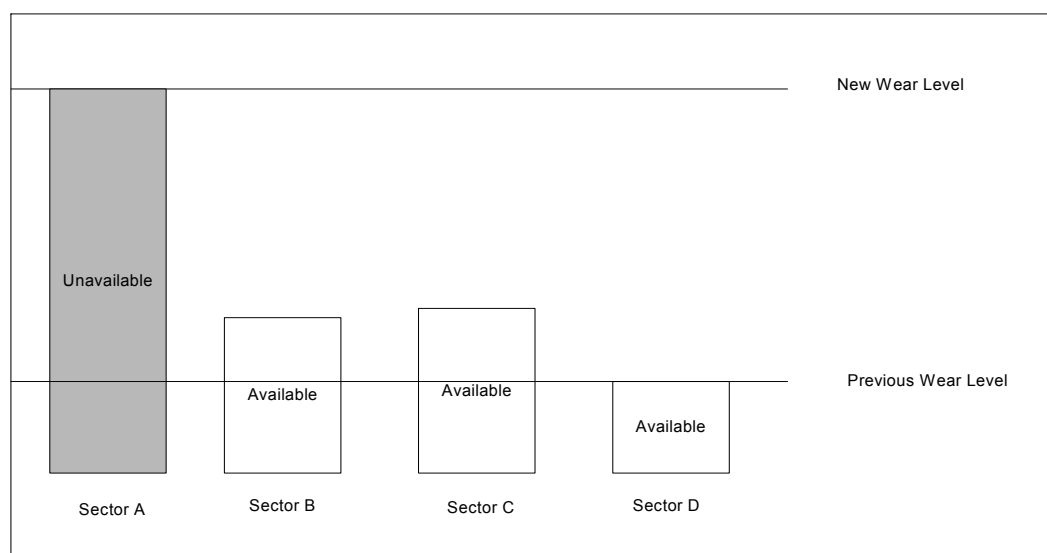
The Fusion Flash File system can dynamically handle cluster failures. This means when a section of the flash is bad or goes bad during the product life cycle the file system will dynamically mark the clusters as defective and not use those again.

Data Caching

The Fusion File System has implemented limited caching to greatly reduce wear of flash storage media. The Fusion Flash File system will cache one data sector (usually 512 bytes) for each open file. This reduces wear for files that are frequently written to. An example may be an update of the time once a minute. The data will continue to be written to the cached data and not the flash each minute. The data will not be written until either the file is closed or new data spans across another sector. This is a cache miss. When this happens the data is stored to the flash and a new sector for the file is cached. The Fusion File system also buffers at least one data sector of the File Allocation Table (FAT). For embedded applications that have additional memory resources available, the FAT cache is user configurable. The user may specify the number of FAT data sectors to buffer. This greatly reduces the flash wear where the FAT table is located in flash.

Wear Leveling

The Fusion Flash File System incorporates a wear-leveling algorithm to evenly distribute the wear over the entire flash device. The user sets the wear level threshold step for each level of wear. When an erasable section reaches a specified wear level the file system will mark that block as unavailable. The file system will then find a new cluster to use. When there are no longer any available sectors, the file system will reset the flash driver for a new wear level and mark all of the unavailable FAT clusters as available and then start a new wear level.



Error Correction

The Fusion Flash File System provides user callable error correction functions. This enables the user to error encode their data and then correct it should an error occur during storage. The error correction algorithm is a Reed-Solomon Error Corrector. Reed-Solomon codes are block-based error correcting codes with a wide range of applications in digital communications and storage. Reed-Solomon codes are used to correct errors in many systems including:

- *Storage devices (including tape, Compact Disk, DVD, barcodes, file systems etc)*
- *Wireless or mobile communications (including cellular telephones, microwave links, etc)*
- *Satellite communications*

The Reed-Solomon encoder takes a block of digital data and adds extra "redundant" bits. Errors occur during transmission or storage for a number of reasons (for example noise or interference, scratches on a CD, etc). The Reed-Solomon decoder processes each block and attempts to correct errors and recover the original data. The number and type of errors that can be corrected depends on the characteristics of the Reed-Solomon code. For this application, two bytes out of each 32 bytes can be corrected.

Reed Solomon codes are a subset of BCH codes and are linear block codes. A Reed-Solomon code is specified as $RS(n,k)$ with s -bit symbols.

This means that the encoder takes k data symbols of s bits each and adds parity symbols to make an n symbol codeword. There are $n-k$ parity symbols of s bits each. A Reed-Solomon decoder can correct up to t symbols that contain errors in a codeword, where $2t = n-k$.

The following diagram shows a typical Reed-Solomon codeword (this is known as a Systematic code because the data is left unchanged and the parity symbols are appended):

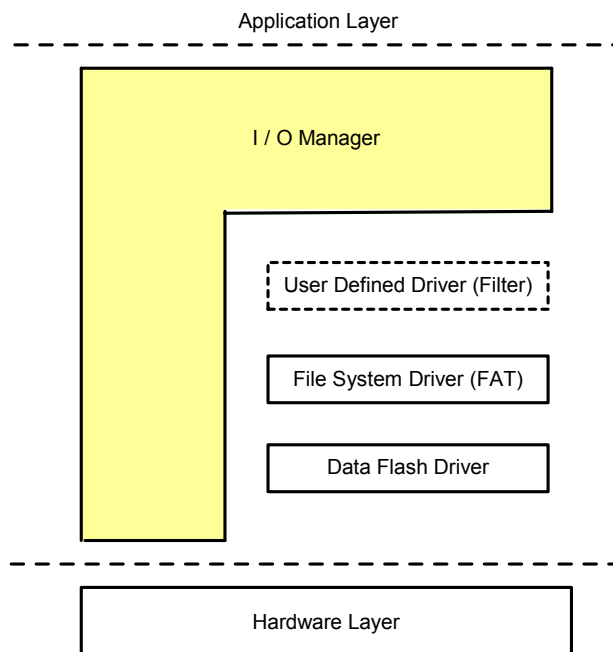
The version available with the Fusion File System is a $RS(32,28)$ with 8 bit symbols. This means each 28-byte block of data is encoded into 32-byte correctable blocks. Each 32-byte block can have up to two bad bytes and still be successfully corrected.

Chapter 2

Product Perspective

Overview of the Model

The product consists of two drivers that are tightly coupled into a single entity: File System Driver (FAT FSD) and the Data Flash Driver (specific to Atmel Data Flashes). While the lower Data Flash Driver can stand alone or be called directly, the File System Driver (FSD or FAT Driver) cannot. It is dependent upon the Flash Driver or a Driver of similarity in order to access the device the file system resides. Therefore the Flash Driver gives users a physical view of the hardware, the FSD gives them more of a logical one.

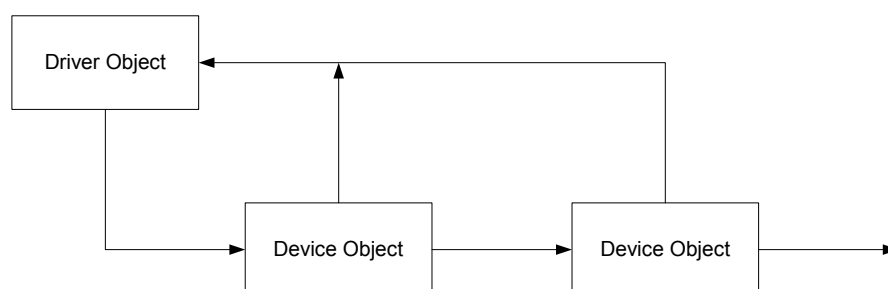


The coupling of the FSD to the FAT is done via a driver model, solely developed for this product. It's technology is actually a potpourri of existing driver architectures integrated into a single solution. It is stripped down for speed and portability, since this is very important for embedded software engineering. The model also allows for developers to add drivers above or below the file system.

Device Model

The File System uses a simple model that can be ported into any environment with minimal performance degradation and memory requirements. It allows the file system to be extensible and portable to these environments with improved integration time.

There are two parts to the model: Driver Object and Device Object.



Driver Object

The Driver Object implements the interface and operations for all devices that are similar. For example an FSD Driver for FAT. It goes to reason that the operations for read and write a FAT volume does not change from device to device, logically speaking, therefore a single Driver Object for all FAT file systems, would bind to one or more FAT devices.

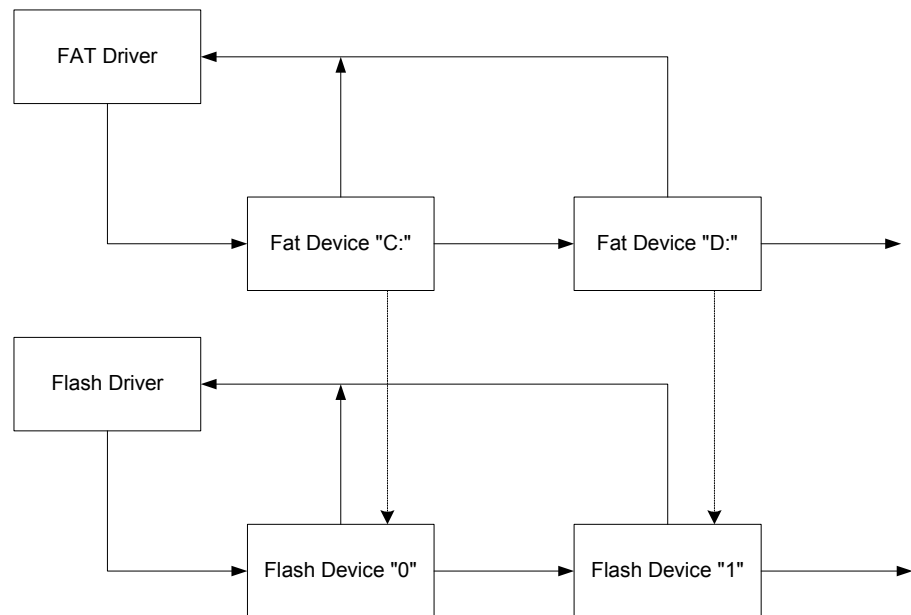
nFlags	Flags specific to the driver (currently not used)
nDriverNo	Identifies the Driver Class and Major Identification or Subclass (used as the prefix in all devices belonging to this driver. This part of the ID, is unique and cannot be used by any other driver in the system.
pDeviceList	The list of devices belonging to this driver.
pIoDispatch	I/O dispatcher for the driver.
pDriverInit	Load or initialization function for the driver.
pDriverUnload	Unload function for the driver.

Device Object

The Device Object is a data structure describing the device to which it binds. Device Objects may bind to other device objects in a chain and in the implementation of the file system, that object would be the Flash Device, which in turns binds directly to the hardware interface. This is why the Flash Driver is the physical view of the file system and the file system driver (FSD) is the logical view.

nDeviceNo	The Major and Minor number that identifies the Device. Since the Major Number is equivalent to the Minor number of the Driver to which this device belongs, this number is unique throughout the system.
nFlags	Current operational and status flags for the device.
nRefCount	Number of open descriptors on this device (this is only used if the device is to be unloaded, with open descriptors still on it).
pDrvParent	The Driver this device belongs to.
pDevNext	The next device in the list.
pDevBinded	The device to which this device is binded to. If NULL, then typically this is a Physical Device Object that talks directly to the hardware or hardware API.
pfnDeviceInit	Initialization function for this device.
pDevice_Ext	The device extension area. (explained in more detail later).

A single device object can only link to a single device object below it, but can accept multiple links from above it. For the purpose of the file system, it is best to use a one to one mapping between the FSD devices and the Flash devices.



Dispatch Interface

Access to a device is done through a dispatch interface. In reality, it is not a true dispatch as the term implies, but can be extended to one, since the data is passed via an I/O packet or object. The dispatching is simply taking the opcode of the packet and routing it to the proper handler of the driver.

IO_DISPATCH Method

Parameters:

piop	Pointer to I/O Packet
------	-----------------------

Return status:

iostatus	The result of the operation.
----------	------------------------------

Prototype:

```
DEV_IOSTATUS (*pIoDispatch)(PDEV_IOPACKET piop);
```

It is advised to use the macro, `IO_DISPATCH(piop)`, which takes a pointer to the I/O packet as an argument.

Example:

```
DEV_IOSTATUS iostat;
DEV_IOPACKET iop;

iop.nIoOperation = DEVICE_IO_CONTROL;
iop.pDevice = GET_DEVICE_OBJECT(devno);
iop.nFunction =
    _IOCTL_MAKECODE(
        MAJOR_DEVNO( Device_GetId(_iop.pDevice) ),
        _FSD_IOCTL_MOUNT, 0
    );

iostat = IO_DISPATCH( &iop );
```

One drawback as seen here is that the Device Object of the target device is required to properly route the packet to its target device. The developer must either supply the device number directly or map the ID with a device mnemonic, like many other operating systems do. This mapping is optional by the user, but an example of a very simple I/O manager is included to show how devices can use these mnemonics, like “C:”, to identify them.

Also included, is an example of a File I/O API that maps system level functions that are very familiar to “C” programmers.

I/O Packet

pDevice	The Target Device Object the packet is destined for
nIoOperation	One of five I/O operations: Open, Close, Read, Write, IoControl
nFunction	Function of the I/O operations. This is necessary when doing IoControl. This can also be used as an attribute field for other operations, especially “Open”.
pDescriptor	The Descriptor for the device, which contains at the very least address information for the target.

pOutBuffer	The data to output to the device
nOutBufferLen	The length of the data to output
pInBuffer	The data to input from the device
pInBufferLen	The length of the data inputted

More descriptive uses for the I/O Packet will be explained later, when the functionality of the FSD is explained.

Device Driver Table

The Device Driver Table, defined within the header file *devtbl.h*, is the only part of the Device Object model itself that will vary from platform to platform. This Table defines the Driver Objects for the system, in this case, the FAT FSD and the Flash Port Driver. An example is provided that can be reused with little modification required to port the file system to a new platform.

The table is order dependent and therefore it is important as to the order in which the drivers are placed in the table, since lower level drivers must initialize before higher level ones. The drivers initialized first, appear first in the table.

The contents of the Device Driver Table, in *devtbl.h*, are organized with the Driver Identification Section(s) first for all the device drivers in the system, followed by the table itself:

Device Driver Identification Section

- DEVICE DRIVER ID (Major Number): A list of Ids is found in *devobj.h*. Only
- DRIVER EXPORTS: The three master functions for initialization, shutdown, and dispatching.
- DEVICE LIST: The list of the Device Objects Ids (can be multiple) that are aggregated by the driver. This is the Minor Number of the Device identification.

For each type of device driver in the system a group of the above will be defined.

Driver Table

As previously mentioned, the devices go in the table FIFO, meaning they initialize from the top down.

Each entry in the Table is a Driver Object entry, defined statically. It is easiest to use the Macro, `DEFINE_DEVICE_DRIVER`, to define a device driver for each grouping of device driver in the system.

The file *devtbl.h* is very well documented and porting to a specific implementation should be painless. For the most part, very little if anything, should be done with this table, unless a new class of device drivers is added.

DISCLAIMER

The object model used by the Flash File system is a public domain model and is therefore not covered directly by any of the copyrights pertaining to this software. The model is actually a simplistic derivation from other platform models, to provide for maximum portability and was only used to tightly integrate the *File System Driver* to the *Flash Port Driver*. It will also help for extending the file system, if other FSDs, supporting other file system types is desired, i.e. NTFS.

Chapter 3

System Requirements

The header file *platform.h* contains definitions that are imperative to properly port the FFS from one environment to another. The inability to handle these or define these for the specific environment will result in major modifications to the drivers to compensate and may require a larger data footprint in certain cases.

Platform Compatibility

A table at the end of this chapter maps the functions to their “C” equivalent. For more details on these, a “C” reference guide will be required to demonstrate their usage, since that is not the premise of this document.

Memory Management

The file system requires several memory management functions to be provided by the target environment

- DEVOBJ_MALLOC – Allocates memory block.
- DEVOBJ_FREE – Frees (releases) a memory block.
- DEVOBJ_MEMSET - Sets buffers to a specified character.
- DEVOBJ_MEMCPY - Copies characters between buffers.
- DEVOBJ_MEMCMP - Compare characters in two buffers.

Character Functions

The file system requires several character management functions to be provided. These are commonly found in the C-header file, *ctype.h*.

- DEVOBJ_ISLOWER - Returns true if the character is a particular representation of a lowercase character.

- DEVOBJ_ISUPPER - Returns true if the character is a particular representation of an uppercase character.
- DEVOBJ_TOLOWER – Converts a character to a lowercase representation.
- DEVOBJ_Toupper – Converts a character to an uppercase representation.

String Management

The file system requires several string management functions to be provided. This is very critical since pathnames are frequently used throughout many of the calls.

- DEVOBJ_STRCPY - Copy a string.
- DEVOBJ_STRCAT – Append a string.
- DEVOBJ_STRCMP – Compare strings.
- DEVOBJ_STRNCPY - Copy *n* characters of one string to another.
- DEVOBJ_STRNCAT - Append *n* characters of a string.
- DEVOBJ_STRNCMP – Compare the first *n* characters of two strings.
- DEVOBJ_STRCHR - Find a character in a string.
- DEVOBJ_STRSTR – Find a substring in a string.

Time Management

The file system requires time management functions to be provided. A system timer should be implemented that keeps some sort of base time. Because Unix time is based upon the number of seconds since 1970, some provision to handle this must be made.

- DEV_TIME – Integral type that stores the number of seconds since Jan 1st, 1970. Typically this is an unsigned long or time_t as it is known in “C”.
- DEV_LOCALTIME – A structure that stores time information in a more specific manner. The structure is typically known as a tm structure in “C” and has the following layout:

tm_sec	Seconds after the minute - [0,59]
--------	-----------------------------------

tm_min	Minutes after the hour - [0,59]
tm_hour	Hours since midnight - [0,23]
tm_mday	Day of the month - [1,31]
tm_mon	Months since January - [0,11]
tm_year	Years since 1900
tm_wday	Days since Sunday – [0,6]
tm_yday	Days since January 1 – [0,365]
tm_isdst	Daylight savings time flag

All values are integers (*int* or DEVOBJ_INT)

- DEVOBJ_TIME – Retrieves the current system time in number of seconds since Jan 1st, 1970.
- DEVOBJ_LOCALTIME – Converts a time value (DEVOBJ_TIME) into the LOCALTIME structure above.

Synchronization Objects

The file system requires several synchronization object capabilities to be provided. The critical section allows only one task or thread to access it at any one time. Critical sections are typically implemented with Mutual Exclusion objects or a Semaphore with a count of “1”. The implementation of the actual object is on a per environment basis (i.e. it’s dependent of the operating system used and what type of synchronization is provided by that OS).

- CRITICAL_SECTION – Data type for a critical section. User implementation can be as simple as a semaphore with a max count of 1 (implying mutual exclusion).
- DEVOBJ_DECLARE_CRITICAL_SECTION – Declares the CRITICAL_SECTION object, but does not initialize it.
- DEVOBJ_INIT_CRITICAL_SECTION – Initializes or defines a CRITICAL_SECTION object.

- **DEVOBJ_DESTROY_CRITICAL_SECTION** – Destroys an initialized **CRITICAL_SECTION** object.
- **DEVOBJ_ENTER_CRITICAL_SECTION** – Request ownership the **CRITICAL_SECTION** object. The task or thread is blocked if another already has ownership, until it is released (order is FCFS).
- **DEVOBJ_LEAVE_CRITICAL_SECTION** – Release ownership of the **CRITICAL_SECTION** object.

C Language Cross Reference

Memory Management

Platform	C Language Equivalent
DEVOBJ_FREE(<i>memblock</i>)	void free(void *memblock)
DEVOBJ_MALLOC(<i>size</i>)	void *malloc(size_t size)
DEVOBJ_MEMCMP(<i>buf1, buf2, count</i>)	int memcmp(const void *buf1, const void *buf2, size_t count)
DEVOBJ_MEMCPY(<i>dest, src, count</i>)	void *memcpy(void *dest, const void *src, size_t count)
DEVOBJ_MEMSET(<i>dest, src, count</i>)	void *memset(void *dest, int c, size_t count)

Character Functions

Platform	C Language Equivalent
DEVOBJ_ISLOWER(<i>c</i>)	int islower(int c)
DEVOBJ_ISUPPER(<i>c</i>)	int isupper(int c)
DEVOBJ_TOLOWER(<i>c</i>)	int tolower(int c)

DEVOBJ_Toupper(<i>c</i>)	int toupper(int <i>c</i>)
----------------------------	------------------------------------

String Management

Platform	C Language Equivalent
DEVOBJ_STRCAT(<i>dest, src</i>)	char *strcat(char *<i>dest</i>, const char *<i>src</i>);
DEVOBJ_STRCHR(<i>string, c</i>)	char *strchr(const char *<i>string</i>, int <i>c</i>);
DEVOBJ_STRCMP(<i>string1, string2</i>)	int strcmp(const char *<i>string1</i>, const char *<i>string2</i>);
DEVOBJ_STRCPY(<i>dest, src</i>)	char *strcpy(char *<i>dest</i>, const char *<i>src</i>);
DEVOBJ_STRLEN(<i>string</i>)	size_t strlen(const char *<i>string</i>);
DEVOBJ_STRNCAT(<i>dest, src, count</i>)	char *strncat(char *<i>dest</i>, const char *<i>src</i>, size_t <i>count</i>);
DEVOBJ_STRNCMP(<i>str1, str2, count</i>)	int strncmp(const char *<i>str1</i>, const char *<i>str2</i>, size_t <i>count</i>);
DEVOBJ_STRNCPY(<i>dest, src, count</i>)	char *strncpy(char *<i>dest</i>, const char *<i>src</i>, size_t <i>count</i>);
DEVOBJ_STRSTR(<i>string, strset</i>)	char *strstr(const char *<i>string</i>, const char *<i>strset</i>);

Time Management

Platform	C Language Equivalent
DEVOBJ_TIME(<i>timer</i>)	time_t time(time_t *<i>timer</i>)
DEVOBJ_LOCALTIME(<i>timer, tms</i>)	struct tm *localtime(const time_t *<i>timer</i>)

Note: for Localtime, the C-equivalent returns a pointer to the structure. This could pose problems at the device driver level, since blocking can occur which would invalidate the pointer if another task made the same call. Therefore, an example has been included with the code and in the appendix to remedy this, where localtime takes two arguments, one for the timer and one for the tm struct.

Synchronization

There is no cross reference for Synchronization objects, since there is no provision in the language to support them. The target environment, should at worst case support Mutual Exclusion synchronization, since Critical Sections are regions of software that only allow one execution thread or task access at any one time.

The CRITICAL_SECTION object could be nothing more than a simple *semaphore* implementation with a count of 1.

Memory Requirements

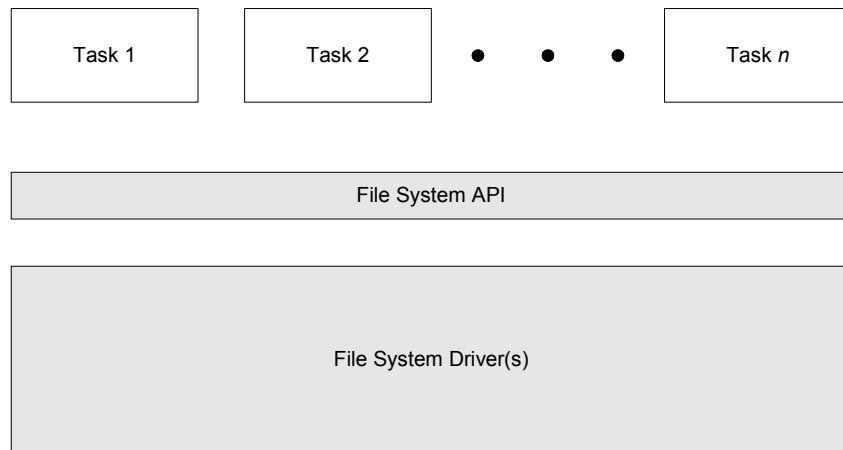
Memory requirements for the FFS will vary from platform to platform, depending on the word size and the byte order (Little Endian vs. Big Endian). The feature of the Flash File system has been optimized both in data and code space, to provide maximum efficiency without sacrificing functionality.

The data resources required are described in the appendix for the major objects that are present in the FFS.

Chapter 4

File System API

The System API is the entry point into the file I/O sub-system. Although at a system level, the calls resemble those that are familiar to the ANSI-C calls commonly found in Unix operating systems. Only a subset of calls are supported, since many are redundant and for embedded products are really not required.



File System Methods

During integration the developer can include all or a portion of these calls to meet their specific needs for their product. At a minimum, the volume operations and file operations should be included, if no directory hierarchy is required.

Volume Specific Operations

These are methods that are device or volume-wide:

- `sys_format` – Formats a volume file system
- `sys_mount` – Mounts a file system (to a device name)
- `sys_unmount` – Unmounts a file system

Directory Specific Operations

These are operations specific to special files known as directories:

- `sys_closedir` – Closes a directory
- `sys_mkdir` – Creates a new subdirectory
- `sys_opendir` - Opens a directory
- `sys_readdir` - Reads an entry from a directory
- `sys_rmdir` – Removes a directory

File Specific Operations

These are operations specific to regular files:

- `sys_close` – Closes a file
- `sys_creat` – Creates a new file or truncates an existing one
- `sys_open` – Opens a file
- `sys_read` – Reads data from a file
- `sys_write` – Writes data to a file

Miscellaneous (Support) Operations

These are methods that are miscellaneous, but generally are used for informational or special handling of files.

- `sys_eof` – Test if current file pointer is at the end of the file
- `sys_lseek` – Moves the file pointer in an opened file
- `sys_remove` – Deletes a file
- `sys_rename` – Renames a file
- `sys_stat` – Retrieves information from a file

- `sys_tell` – Gets the current file position in a file

Structures

There are a few important data structures that allow the user to gather information about the file system and even perform an `ls` method (as shown in the appendix) so that given a pathname a listing of what is contained within that path is returned.

DIRECT

This is a universal directory structure that is file system independent. It is filled during a call to `readdir` with the filename, not the full path, and an i-number, which in Unix specifies the index of a file's inode.

<code>d_ino</code>	This is the i-node number of the file, which is not supported in FAT file systems, but a value of '0' regardless will indicate a deleted file or free entry.
<code>d_iname</code>	This is the string name of the file (path not inclusive). The size of this field is dependent upon the value for <code>DIRSIZE</code> , defined in <i>fsd.h</i> .

STAT

This structure contains information about the file and is accessed via the `stat` function.

<code>st_dev</code>	The major and minor device numbers of the device which the file is stored on.
<code>st_ino</code>	The i-node number (i-number) of the file
<code>st_mode</code>	A set of bits encoding the type of file and the access permissions it has.
<code>st_nlink</code>	The number of hard links to the file, including the file itself. This value is always 1, since hard links are not implemented in FAT file systems.

st_uid	The user id of the owner of the file. In FAT this value is always 0.
st_gid	The group id of the owner of the file. In FAT this value is always 0.
st_rdev	The type of device if the file resides on a special device. This is always set to st_dev for the file system.
st_size	The size of the file in bytes.
st_atime	The last time the file was accessed (this is disabled for the file system and is only modified if the file was open for write access – prevents wear leveling problems).
st_mtime	The last time the file was modified
st_ctime	The time the file was created.

Types

- FILEDESC – An index number, file descriptor, that denotes an opened file. A value of –1 or UNDEFINED_FILEDESC indicates an invalid file descriptor.
- FILEPOS – An integer value representing the current offset within the file.
- DIRDESC – Synonymous with FILEDESC, except for directories.

System Programmer's Interface

These are the system level APIs whose purpose is to give a Linux-C compatible look and feel to standard File I/O. The biggest difference, due to linkage issues, is the return value. The return values are zero or positive on success (depending on the function) and negative for error.

Use the Macro **_ISERROR(*e*)** to determine if the return value is an error and **_ERRNO(*e*)** to get the “errno” equivalent as found in the appendices.

All of the methods are found in *file_api.h* and corresponding structures in *fsd.h*.

sys_close()

Closes a file previously opened with *sys_open*.

PROTOTYPE:

```
int sys_close( FILEDESC fd )
```

INPUT PARAMETERS:

fd: Valid file descriptor (previously assigned by *sys_open* or *sys_creat*).

RETURN VALUES:

int; ESUCCESS = Success, <0 = Failure.

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
FILEDESC _fd;

_fd = sys_open( filename, _OPEN_RDWR, 0 );
if ( _ISERROR(_fd) )
{
    /*
     * Handle Errors here
     */

    ....
}
else
{
    ....
    sys_close( _fd );
}
```

sys_closedir()

Closes a directory previously opened with sys_opendir.

PROTOTYPE:

```
int sys_closedir( DIRDESC fd )
```

INPUT PARAMETERS:

fd: Valid file descriptor for the directory (previously assigned by *sys_opendir*).

RETURN VALUES:

int; ESUCCESS = Success, <0 = Failure.

EXAMPLE:

```
char _dirname[] = "C:\\Temp";
DIRDESC _fd;

_fd = sys_opendir( _dirname );
if ( _ISERROR(_fd) )
{
    /*
     * Handle Errors here
     */
    ....
}
else
{
    ....
    sys_closedir( _fd );
}
```

sys_creat()

Creates a new file or opens and truncates an existing one.

PROTOTYPE:

```
FILEDESC sys_creat( char* name, int perm )
```

INPUT PARAMETERS:

name: Name of the new file.

perm: Permissions set to a newly created file.

_PMODE_READ	File has read access (for FAT this is always TRUE)
_PMODE_WRITE	File has write access
_PMODE_RDWR	_PMODE_READ _PMODE_WRITE
_PMODE_EXECUTE	Has no meaning for FAT file system

RETURN VALUES:

FILEDESC; 0 or greater = Valid File Descriptor (Success), <0 = Failure

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
FILEDESC _fd;

_fd = sys_creat( _filename, _PMODE_RDWR );
if ( _ISERROR(_fd) )
{
    ....
}
else
{
    ....
    sys_close( _fd );
}
```

sys_devstats()

Gets the statistics of the device, such as number of blocks and block size (used for testing drivespace). This is a proprietary function only and it's use is to avoid the complexity of obtaining volume information by other means.

PROTOTYPE:

```
int sys_devstats(char* devname, PDEVSTATS stats )
```

INPUT PARAMETERS:

devname: string name for the device (i.e. C:).

stat: Pointer to a DEVSTATS structure.

RETURN VALUES:

int, ESUCCESS = Success, <0 = Failure.

EXAMPLE:

```
char _devname[] = "C:";
DEVSTATS _devstats;
unsigned long _nfree;

sys_devstats( devname, &_devstats );

/*
 * The number of free blocks
 */

_nfree = _devstats.blkStats.nFreeBlocks * _devstats.blkStats.nBlockSize;
```

sys_eof()

Tests for end-of-file on a valid file descriptor.

PROTOTYPE:

```
int sys_eof( FILEDESC fd )
```

INPUT PARAMETERS:

fd: Valid file descriptor (previously assigned by *sys_open* or *sys_creat*).

RETURN VALUES:

int; 1 = end of file is true, 0 = end of file is not reached, <0 = Failure.

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
int _fd;

_fd = sys_open( _filename, _OPEN_RDONLY, 0 );
if ( _ISERROR(_fd) )
{
    ....
}
else
{
    ....
    while ( !sys_eof(_fd) )
    {
        sys_read(...);
        ....
    }
    sys_close( _fd );
}
```


sys_format()

Formats a file system on a mounted device. Care must be taken when doing a format, for if the device has already been formatted, then all data currently on that volume will be erased.

PROTOTYPE:

```
int sys_format( char* devname, char* volname )
```

INPUT PARAMETERS:

devname; string name for the device (i.e. C:).

volname: string name that is used as the volume label.

RETURN VALUES:

int; ESUCCESS = Success, <0 = Failure.

EXAMPLE:

See example for *sys_mount*.

sys_lseek()

Moves the file pointer associated with *fd* to a new location that is *offset* bytes from *origin*. The next operation on the file occurs at the new location.

PROTOTYPE:

```
FILEPOS sys_lseek( FILEDESC fd, FILEPOS offset, int origin )
```

INPUT PARAMETERS:

fd: Valid file descriptor (previously assigned by *sys_open* or *sys_creat*).

offset: Number of bytes from *origin* forward or backwards.

origin: Initial position (one of three values):

 SEEK_BEG – Seek from beginning of file

 SEEK_CUR – Seek from the current position

 SEEK_END – Seek from end of file

RETURN VALUES:

FILEPOS; the offset, in bytes, of the new position from the beginning of the file,
<0 = Failure.

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
int _fd;

_fd = sys_open( _filename, _OPEN_RDWR, 0 );
if ( _ISERROR(_fd) )
{
    ....
}
else
{
    int _err = sys_lseek(_fd, 100, _SEEK_BEG );

    if ( _ISERROR(_err) )
    {
        ....
    }
    ....
    sys_close( _fd );
}
```

sys_mkdir()

Creates a new subdirectory. The path up to the new directory must exist otherwise an error will occur.

PROTOTYPE:

```
int sys_mkdir( char* dirname )
```

INPUT PARAMETERS:

dirname: Fully qualified path of the directory name to create.

RETURN VALUES:

int, ESUCCESS = Success, <0 = Failure.

EXAMPLE:

```
char _dirname[] = "C:\\Temp\\Newdir";
int _errno;

/*
 * The subdirectory Temp must exist to create Newdir below it.
 */

_errno = sys_mkdir(_dirname);

if ( _ISERROR(_errno) )
{
    ....
}

/*
 * Success
 */

else
{
    ....
}
```

sys_mount()

Mounts a filesystem so that file I/O operations can take place. If using the I/O Manager and have it configured correctly, then the user can set auto-mounting, as in Unix, of file systems at initialization time.

NOTE: The `sys_mount` and `sys_unmount` functions are the only two functions that return an actual error code, i.e. *errno*.

PROTOTYPE:

```
int sys_mount( char* devname )
```

INPUT PARAMETERS:

devname: String name of the device.

RETURN VALUES:

int; ESUCCESS = Success, <0 = Failure.

EXAMPLE:

```
char _devname[] = "C:";
int _errno;

_errno = sys_mount( _devname );

/*
 * Format error?
 */

if ( _ISERROR(_errno) )
{
    /*
     * Failure is sometimes due to the file system not being
     * formatted
     */

    if ( _errno == ENOEXEC )
    {
        _errno = sys_format( _devname )

        /*
         * Critical FILE SYSTEM error?
         */

        if ( _ISERROR(_errno) )
        {
            return _ERRNO(_errno);
        }

        ....
    }
    else
    {
        return _ERRNO(_errno);
    }
}
```

sys_open()

Opens the file specified by the fully qualified pathname, *name*. If the filename is a directory (see `sys_opendir`), then the directory must exist and if it does, only opens it for read-only access.

PROTOTYPE:

```
FILEDESC sys_close( char* name, int flags, int perm)
```

INPUT PARAMETERS:

name: Fully qualified pathname of the file.

flags: Type of operations allowed on file.

_OPEN_RDONLY	Open the file for reading only (default)
_OPEN_WRONLY	Open the file for writing only
_OPEN_RDWR	Open the file for reading and writing
_OPEN_APPEND	Append to the file when writing rather than at the beginning
_OPEN_CREAT	Create the file if it does not exist (perm should be used when this mode is specified)
_OPEN_TRUNC	Truncates the file to zero length if opened for writing
_OPEN_EXCL	Return an error if the file is to be created and already exists
_OPEN_DELETE	Deletes a file on close (use sys_remove instead)

perm: Permissions set to a newly created file (ignored if file already exists), see `sys_creat`.

RETURN VALUES:

FILEDESC; 0 or greater = Valid File Descriptor (Success), <0 = Failure.

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
int _fd;

_fd = sys_open( _filename, _OPEN_CREATE | _OPEN_EXCL, _PMODE_RDWR );
if ( _ISERROR(_fd) )
{
    /*
     * Handle Errors here
     */
    ....
}
```

```
}  
else  
{  
    ....  
    sys_close( _fd );  
}
```

sys_opendir()

Opens the file specified by the fully qualified pathname, *dirname*.

NOTE: Directories can only be open as read-only.

PROTOTYPE:

`DIRDESC sys_opendir(char* dirname)`

INPUT PARAMETERS:

*char**: Fully qualified pathname of the directory.

RETURN VALUES:

DIRDESC; 0 or greater = Valid File Descriptor (Success), <0 = Failure.

EXAMPLE:

```
char _dirname[] = "C:\\Temp";
int _fd;

/*
 * Open the Temp Directory
 */

_fd = sys_opendir( _dirname );
if ( _ISERROR(_fd) )
{
    ....
}
else
{
    ....
    sys_closedir( _fd );
}
```

sys_read()

Closes a file previously opened with `sys_open`.

PROTOTYPE:

```
int sys_read( FILEDESC fd, void* buf, unsigned int count )
```

INPUT PARAMETERS:

fd: Valid file descriptor (previously assigned by `sys_open` or `sys_creat`).

buf: Buffer to read into.

count: Maximum number of bytes to read.

RETURN VALUES:

int: The number of bytes read or `EFAIL` if an error occurred. If the number of bytes read is less than *count*, then an EOF condition was probably encounter (see `sys_eof`).

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
int _fd;
char buf[128];

_fd = sys_open( _filename, _OPEN_RDWR, 0 );
if ( _ISERROR(_fd) )
{
    ....
}
else
{
    int _nread;

    while ( (_nread = sys_read( _fd, buf, 128 )) > 0 )
    {
        ....
    }

    if ( _ISERROR(_nread) )
    {
        /*
         * Handle Errors here
         */
    }
    ....
    sys_close( _fd );
}
```


sys_readdir()

Reads a single entry from a directory, previously opened with **sys_opendir**.

PROTOTYPE:

```
int sys_readdir( DIRDESC fd, PDIRECT dir )
```

INPUT PARAMETERS:

fd: Valid file descriptor (previously assigned by *sys_opendir*).

dir: Pointer to a DIRECT structure, which is defined as follows:

d_ino	The i-node number (i-number) of the file. If “0”, then this entry is an unused or free entry.
d_name[DIRSIZE]	Name of the file up to DIRSIZE characters

RETURN VALUES:

int, ESUCCESS = Success, <0 = Failure.

EXAMPLE:

```
char _dirname[] = "C:\\Temp";
int _fd;

_fd = sys_opendir( _dirname );
if ( _ISERROR(_fd) )
{
    ....
}
else
{
    DIRECT _dir;
    int _errno;

    while( (_errno = sys_readdir( _fd, &_dir )) == ESUCCESS )
    {
        /*
         * Free or Unused entry?
         */

        if ( _dir.d_ino == 0 )
        {
            continue;
        }
        ....
    }
    ....
    sys_closedir( _fd );
}
```

sys_remove()

Deletes a file from the file volume. Directories can also be removed using this function as long as they are empty.

sys_unlink performs the same operation.

PROTOTYPE:

```
int sys_remove( char* name )
```

INPUT PARAMETERS:

name: Fully qualified pathname of the file.

RETURN VALUES:

int, ESUCCESS = Success, EFAIL = Unsuccessful.

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
int _errno;

_errno = sys_remove( _filename );
if ( _ISERROR(_errno) )
{
    /*
     * Errors are handled here
     */
    ....
}
```

sys_rename()

Closes a file previously opened with sys_open.

PROTOTYPE:

```
int sys_rename( char* oldname, char* newname )
```

INPUT PARAMETERS:

oldname: Fully qualified pathname of the file.

newname: Name for the new file (not fully qualified).

RETURN VALUES:

int, ESUCCESS = Success, <0 = Unsuccessful.

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
char _newname[] = "test.old"
int _errno;

_errno = sys_rename( _filename, _newname );
if ( _ISERROR(_errno) )
{
    /*
     * Errors are handled here
     */

    ....
}
else
{
    int _fd;

    DEVOBJ_STRCPY( _filename, "C:\\Temp\\" );
    DEVOBJ_STRCAT( _filename, _newname );

    _fd = sys_open( _filename, _OPEN_RDWR, 0 );
    if ( _ISERROR(_fd) )
    {
        ....
    }
    else
    {
        ....
        sys_close( _fd );
    }
}
```

sys_rewinddir()

Rewinds the directory to the first entry. Since directories can only be opened as read-only and their size, indeterminate on some file systems, seeking on them is only allowed in this manner.

PROTOTYPE:

```
int sys_rewinddir( FILEDESC fd )
```

INPUT PARAMETERS:

fd: Valid file descriptor.

RETURN VALUES:

int, ESUCCESS = Success, <0 = Unsuccessful.

EXAMPLE:

```
char _dirname[] = "C:\\Temp";
int _fd;

/*
 * Open the TEMP Directory
 */

_fd = sys_opendir( _dirname );
if ( !_ISERROR(_fd) )
{
    ....
}
else
{
    DIRECT _dir;

    while ( !_ISERROR( sys_readdir( _fd, &_dir ) ) )
    {
        ....
    }

    /*
     * Rewind to first entry in the directory
     */

    rewinddir( _fd );
    ....
    sys_closedir( _fd );
}
```

sys_rmdir()

Deletes a directory.

PROTOTYPE:

```
int sys_rmdir( char* dirname )
```

INPUT PARAMETERS:

dirname: Path of directory to be removed.

RETURN VALUES:

int; ESUCCESS = Success, EFAIL = Unsuccessful.

EXAMPLE:

```
char _dirname[] = "C:\\Temp";
int _errno;

/*
 * The subdirectory Temp must exist to create Newdir below it.
 */

_errno = sys_rmdir(_dirname);

if ( _ISERROR(_errno) )
{
    ....
}
```

sys_stat()

Get status information on a file or directory.

PROTOTYPE:

```
int sys_stat( char* name, PSTAT pstat )
```

INPUT PARAMETERS:

name: Pathname of the file.

pstat: Pointer to structure of type STAT

RETURN VALUES:

int, ESUCCESS = Success, EFAIL = Unsuccessful.

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
STAT _stat;
int _errno;

_errno = sys_stat( filename, &_stat );
if ( !_ISERROR(_errno) )
{
    /*
     * Is FILE Writable?
     */

    if ( _stat.st_mode & _SMODE_IWRITE )
    {
        ....
    }
    ....
}
```

sys_tell()

Get the current file position.

PROTOTYPE:

```
FILEPOS sys_tell( FILEDESC fd )
```

INPUT PARAMETERS:

fd: Valid file descriptor (previously assigned by *sys_open* or *sys_creat*).

RETURN VALUES:

FILEPOS; Current position of the file descriptor or EFAIL if the file descriptor is invalid.

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
int _fd;
char buf[128];

_fd = sys_open( _filename, _OPEN_RDWR, 0 );
if ( _ISERROR(_fd) )
{
    ....
}
else
{
    int _nread;

    while ( (_nread = sys_read( _fd, buf, 128 )) > 0 )
    {
        ....
    }

    if ( _ISERROR(_nread) )
    {
        /*
         * Handle Errors here
        */
    }

    /*
     * Determine size of file based on file position
    */

    else
    {
        FILEPOS _fp = sys_tell( _fd );

        ....
    }
    ....
    sys_close( _fd );
}
```

sys_umount()

Unmounts a filesystem, rendering it unusable until it is mounted again. The filesystem must have been previously mounted with the *sys_mount* function or an error will be returned.

PROTOTYPE:

```
int sys_umount( char* devname )
```

INPUT PARAMETERS:

devname: String name of the device.

RETURN VALUES:

int, ESUCCESS = Success, *errno* = Unsuccessful.

EXAMPLE:

```
char _devname[] = "C:";
int _errno;

_errno = sys_mount( _devname );

/*
 * Format error?
 */

if ( _ISERROR(_errno) )
{
    /*
     * Test for ERROR Type (see sys_mount)
     */

    ....
}

/*
 * If execution gets here, then the system is mounted and formatted
 */

....

sys_umount( _devname );
```


sys_unlink()

Unlinks or removes a file from the file system. This function is the same as the *sys_remove* function described earlier.

PROTOTYPE:

```
int sys_unlink( char* name )
```

INPUT PARAMETERS:

name: Fully qualified pathname of the file.

RETURN VALUES:

int, ESUCCESS = Success, EFAIL = Unsuccessful.

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
int _errno;

_errno = sys_unlink( _filename );
if ( _ISERROR(_errno) )
{
    /*
     * Errors are handled here
     */
}
```

sys_write()

Closes a file previously opened with `sys_open`.

PROTOTYPE:

```
int sys_write( FILEDESC fd, void* buf, unsigned int count )
```

INPUT PARAMETERS:

fd: Valid file descriptor (previously assigned by `sys_open` or `sys_creat`).

buf: Buffer to write from.

count: Maximum number of bytes to write.

RETURN VALUES:

int, The number of bytes written or `EFAIL` if an error occurred.

EXAMPLE:

```
char _filename[] = "C:\\Temp\\test.txt";
int _fd;

_fd = sys_open( _filename, _OPEN_RDWR | _OPEN_TRUNC, 0 );
if ( !_ISERROR(_fd) )
{
    ....
}
else
{
    int _nwritten;
    char buf[128];

    do
    {
        /*
         * Initialize the buffer with some data
         */

        ....

        _nwritten = sys_write( _fd, buf, 128 )
    }
    while ( !_ISERROR(_nwritten) );

    if ( !_ISERROR(_nwritten) )
    {
        /*
         * Handle Errors here
         */
    }
    ....
    sys_close( _fd );
}
```

Chapter 5

I/O Manager

The I/O manager included with the Flash File System manages the mapping between application and driver layers. The I/O manager released here is also responsible for keeping synchronization between open files on a device and the tasks that access them.

The version included enforces mutual exclusion only as a more complex scheme would require much more software and more interaction between the I/O Manager and the File System Driver.

I/O Table

The I/O Table is an extension of the Driver table but it specifically targets the device objects and not the drivers and only those devices that have character names assigned to them. Basically it's intention is for all devices that are FSDs.

- Map the logical device names to their Device Ids (Major and Minor Number) so that calls can be dispatched to the appropriate device.
- Manage the File Control Block Table for each device

IOEntry

Each member of the I/O is defined as follows:

szDevname	The name of the device up to DEV_STRLEN characters.
nDevId	The device number (Major and Minor number)
bAuto	Indicates whether the device is automatically mounted
tblFCB	The table of FCBs. The length of the table is FCB_TBLLEN and since a Hash Table is used, it's best that this be equivalent to a prime number to reduce the number of collisions.

The device name allows higher level and even system level functions to access the device as a string name. For example, file systems can be named C, D, E, AUX, etc. By default the device delimiter is the old MS-DOS “:”, but this can be altered by developers by redefining DEVNAME_DELIMITER, found in *iomgr.h*.

With this table, the I/O Manager can do a sequential search, if more than one file system exists, and map the device mnemonic (name) to it’s ID which is used to aid in dispatching calls to the device.

NOTE: The table is not dynamic and is therefore NOT updated due to mounting/unmounting of file systems.

Initialization

There are two calls, not mentioned in the file system API that are essential in startup and shutdown of the FFS.

- `sys_startup`
- `sys_shutdown`

It is VERY important that prior to using the FFS, that `sys_startup` be invoked, otherwise failures, perhaps catastrophic may occur. The I/O manager is responsible for mounting the file systems to their devices and setting up all the linkage between itself and the devices. Semaphores are also initialized that help to protect accessing of files and descriptors during this time.

Basically the file system should be initialized sometime during or even after the target System initializes itself (OS Initialization) and should be shutdown prior to OS Shutdown.

```
Sys_main()
{
    /* OS STARTUP */

    ....

    sys_startup();

    /* RUN THE OS */

    ....

    sys_shutdown();

    ....

    /* OS SHUTDOWN */
}
```

```
}
```

`sys_shutdown` should be invoked, whenever the system powers down, or even goes into standby mode, as this will commit all open file descriptors, closing them, preventing data loss. Descriptors left open and not handle properly could result in a fragmented file system that contains lost clusters that can never be allocated.

File Control Block

The last entry in the IOEntry block is a table that contains an array of pointers to File Control Blocks. This table maintains the number of open descriptors from the file system's point of view (to be explained shortly). The File Control Block, FCB, is simply a structure that although transparent to the client, is very important to the driver, since the File Object is part of this block.

There is a single FCB table or list per device, therefore for each file system there is an FCB table it. When a request to allocate a FCB, via a File Descriptor (returned to the user as an integer), the I/O Manager verifies the pathname of the request by checking the table for all existing file descriptors allocated to that file.

Implementation

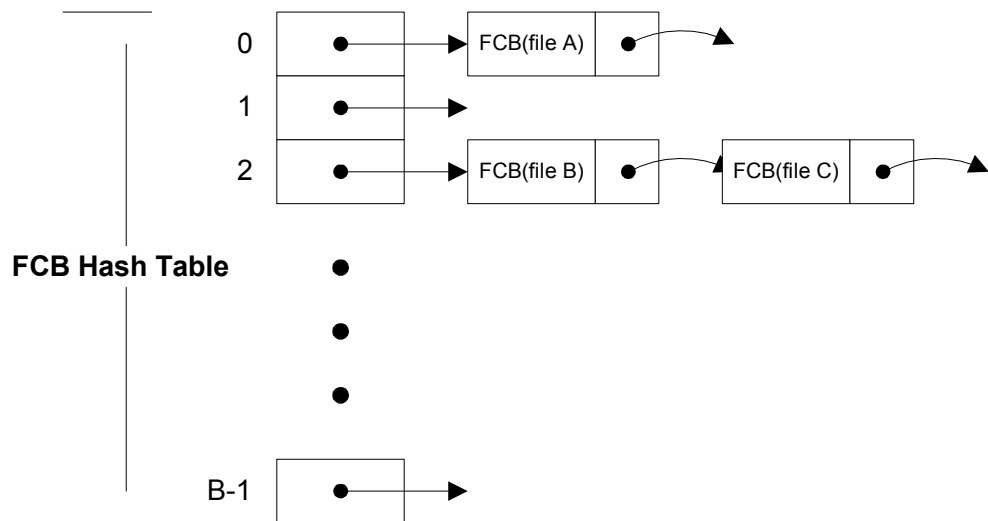
For this version, FCBs are mutually exclusive and therefore, if an entry in the table matches the pathname of the user request, then an error is returned along with an invalid file descriptor. Therefore the operation, an *open* request, fails. The problem is, however, that there are many ways that a pathname can be derived that points to the same file and by hashing this string, multiple FCB entries for the same file could exist in the table.

For example:

`C:\temp\..\test\..files\..files\thisfile.dat = C:\test\files\thisfile.dat`

Now typically no one in their right mind would pass such a pathname, as shown on the left, to the file system. However, in situations where things are running in an automated fashion and builds a filename based on criteria without verification (since it may not know the valid character set of a pathname), such a condition could occur. The file system must be able to handle such conditions, regardless. Therefore, the first thing the I/O manager does, after verifying the device ("C:"), is to *normalize* the pathname part of the string before running it through the hash function, $h(x)$. By doing this, both versions of the previous file name will evaluate to the same index or bucket in the table.

An “open” hash table is used by the I/O Manager to store the FCB entries. Hashing the FCBs allow for quicker access when using the full pathname as the key into the table. An open table is used, since collisions can occur due to the possibly infinite number of potential members. An open hash table uses a linked list for each bucket in the table. Therefore if a collision occurs, then the FCB is appended as the last element in the list.



The previous diagram shows the memory layout of the FCB Table with B buckets (suggested prime number).

Chapter 6

File System Driver

The File System Driver, FSD, is any driver that manages a file volume on a device. There are numerous FSDs available such as VFAT, FAT, NTFS, etc. The FAT, specifically FAT12 and FAT16, will be discussed in detail here, although the operations typically port from one file system to the next with no changes to the arguments.

FSD Objects

FAT Device Extension

Each Device Object has an extension area that only the Driver for those devices knows the layout of. For the FAT driver, the device extension is defined as follows:

FAT_DataClusters	The count of total data clusters
FAT_FreeClusters	The count of free data clusters
FAT_FirstFreeCluster	The index of the first free cluster
FAT_RootDirSectors	The number of root directory sectors
FAT_FirstRootDirSector	The index of the start of the root directory
FAT_Format	The format type of the volume, FAT12 or FAT16
FAT_Eof	The End-of-File marker for the volume, since this varies between format types.
FAT_CacheSector	The current sector of the File Allocation Table currently cached.
FAT_Cache	The cache buffer size
FAT_CriticalSection	The critical section object guarding the FAT

FAT_BootSector	The boot sector information for the volume (this follows byte for byte the boot record stored. See appendix).
----------------	---

Some of the information within the extension is included to reduce the amount of time to calculate it based upon other fields. Since these calculations would be performed quite often, it speeds access time with minimal effect to memory resources.

File Object

The file object is a descriptor used by the FSD to describe information about the access of the file. Not all devices ever used may require a file object, but for the FSD it is mandatory. There is a single file object for a single open instance of a file. Typically file objects exist one for one with a descriptor table defined higher up in the I/O system architecture. These descriptors may actually index another structure that contains both an instance of the file object and another point to an FCB, which allows synchronization between files and directories in the system as a higher level.

pDeviceObj	Pointer to the device object the file object references
nFlags	Status flags of the file object
nPos	Absolute position within the file
pFileExt	File extension area
nBufferLen	Length of the file cache
pBuffer	File buffer or cache.

FAT File Extension

The File extension for FAT Volumes is as follows:

theEntry	Directory entry for the file (directories are special entries)
----------	--

theParentEntry	The parent entry for the file (directory in which it resides)
nDirSector	Sector the directory entry for the file exists
nDirOffset	Offset within the sector the directory entry for the file exists
nCluster	Current cluster index (corresponds to the file position)
nOffset	Current offset within the cluster (the sector can be computed given cluster and offset).

Dispatch Operations

The I/O Dispatch functionality was discussed previously and within the packet was a field that contained the I/O operation and function to be performed. Typically the function code is used for I/O Control operations only.

The following is the “5” I/O Operations, which are placed in the nOperation field of the I/O Packet for the FSD and how they are setup prior to dispatching and what the return code and information is. Fields not listed below are not used by the Operation.

NOTE: All packets must fill the pointer to the Device Object as this is imperative to routing the request.

DEVICE_IO_OPEN

Opens a file for reading and/or writing.

pDescriptor	Pointer to a pre-allocated (uninitialized) File Object
pOutBuffer	Name of the FILE to open, path inclusive
nFunction	Open flags and permissions packed into a single word

DEVICE_IO_CLOSE

Closes and opened file.

pDescriptor	Pointer to the open File Object
-------------	---------------------------------

DEVICE_IO_READ

Inputs data from a file.

pDescriptor	Pointer to the open File Object
pInBuffer	Pointer to the buffer to read into
pInBufferLen	Size of the buffer, indicating the number of characters to read

DEVICE_IO_WRITE

Outputs data to a file

pDescriptor	Pointer to the open File Object
pOutBuffer	Pointer to the buffer to write from
nOutBufferLen	Size of the buffer, indicating the number of characters to write

DEVICE_IO_CONTROL

Performs a specific I/O operation on the device, based on the function code passed. The following are the function codes, nFunction, supported by the File System:

mount

Mounts a volume, allowing access to the file system. A mount must be the first I/O Control operation performed on a file system before access to files and directories are permitted.

No parameters required

unmount

Unmounts a mounted file volume. The unmount must be the last I/O Control operation performed

No parameters required	No parameters required
------------------------	------------------------

format

Formats a file system onto a mounted volume.

pOutBuffer	Pointer to volume label
nOutBufferLen	Length of the volume label

lseek

Seek to a new position in an open file.

pDescriptor	Pointer to the open File Object
pOutBuffer	Pointer to the offset to seek
nOutBufferLen	Set to indicate the Origin to seek from
pInBuffer	Pointer to place position after seek

rename

Renames a file and/or directory.

pDescriptor	Pointer to a File Object
pOutBuffer	Pointer to the new name
nOutBufferLen	String length of the new name
pInBuffer	Pointer to the old name
pInBufferLen	String length of the old name

mkdir

Create a subdirectory

pDescriptor	Pointer to a File Object
pOutBuffer	Pathname of the directory to create
pOutBufferLen	Length of the pathname

rmdir

Remove a subdirectory

pDescriptor	Pointer to a File Object
pOutBuffer	Pathname of the directory to remove
pOutBufferLen	Length of the pathname

readdir

Reads a directory entry.

pDescriptor	Pointer to the open File Object
pInBuffer	Pointer to a DIRECT structure
nInBufferLen	sizeof(DIRECT)

filestat

Gets information on a file

pDescriptor	Pointer to the open File Object
pOutBuffer	Pointer to the pathname of file
nOutBufferLen	String length of the pathname
pInBuffer	Pointer to a STAT function
pInBufferLen	sizeof(STAT)

Chapter 7

Integration and Usage Tips

ROOT Directory Management

One tip that will help wear-leveling of the FLASH part near the front, is to reduce amount of write cycles to the ROOT directory. It is always best to create a subdirectory farm in the root, then build the tree off of that. The reason for this is simply that the ROOT directory in FAT12 and FAT16 is at a fixed location. It cannot be moved and once it goes bad, the file system becomes unstable and possibly unusable. Subdirectories on the other hand can be shifted between different locations as wear leveling conditions are detected to load level the part and extend the life of it as long as possible.

FAT Determination

The source code is currently equipped to autodetect if the target part should be formatted for FAT12 and FAT16. There are rules to this if the system is to be compatible with other FAT File Systems.

FAT Type	Size _{min}	Size _{max}	Clusters _{min}	Clusters _{max}
FAT12	0	< 4.1MB	0	4084
FAT16	4.1 MB	2.0 GB	4085	65524
FAT32	32.5 MB	> 32 GB	65525	

NOTE: FAT32 is not supported with this version as it typically not the format of choice unless the part is at least 512 MB in size (this is 4 Gigabit Flash).

Atmel Part	Density	FAT _{type}	Clusters _{size}	Sectors _{cluster}
AT45DB011B	1 MBit	FAT12	512 bytes	1
AT45DB021B	2 MBit	FAT12	512 bytes	1

AT45DB041B	4 MBit	FAT12	1024 bytes	2
AT45DB081B	8 MBit	FAT12	1024 bytes	2
AT45DB16B	16 Mbit	FAT12	1024 bytes	2
AT45DB321B	32 Mbit	FAT12	2048 bytes	4
AT45DB642	64 Mbit	FAT16	1024 bytes	2
AT45DB1282	128 MBit	FAT16	1024 bytes	2
AT45DB2562	256 MBit	FAT16	2048 bytes	4

Initialization

When the file system is mounted to the device, it queries the target device as to information about the minimum block size it supports on read/write and the number of these blocks available. The total part size is determined on that and is used in one of the two lookup tables including in the software. As seen in the first table preceeding this section, devices up to and including 4.0 MBytes in size are always FAT12 and anything over that and up to 512 Mbytes is FAT16.

The tables, located at the top of the file `fat_vol.c`, are based upon 512 sector sizes, and by Microsoft's definition cannot be used for any sectors larger, but arithmetic shifting based on sector sizes of 1024, 2048 will work just fine. Typically for sector sizes greater than 512, a device should be at the very least 4.1 Mbytes in size, but this is not a mandatory requirement.

Appendix A

Miscellaneous

Compatible *errno* codes

These codes are compatible with those defined by Unix, although not all these are returned by the Flash File System.

Error Code	Definition
EPERM	Operation not permitted
ENOENT	No such file or directory
ESRCH	No such process
EINTR	Interrupted system call
EIO	I/O error (read or write)
ENXIO	No such device or address
E2BIG	Argument list is too long
ENOEXEC	Exec format error
EBADF	Bad file number
ECHILD	No child processes
EAGAIN	Try again
ENOMEM	Out of memory
EACCES	Permission denied
EFAULT	Bad address
ENOTBLK	Block device required

EBUSY	Device or resource busy
EEXIST	File exists
EXDEV	Cross-device link
ENODEV	No such device
ENOTDIR	Not a directory
EISDIR	Is a directory
EINVAL	Invalid argument
ENFILE	File table overflow
EMFILE	Too many open files
ENOTTY	Not a typewriter (obsolete)
ETXTBSY	Text file busy
EFBIG	File is too large
ENOSPC	Device is out of space
ESPIPE	Illegal seek
EROFS	Read-only file system
EMLINK	Too many links
EPIPE	Broken pipe
EDOM	Math argument out of domain of function
ERANGE	Math result not representable
EDEADLK	Resource deadlock would occur
ENAMETOOLONG	File name too long
ENOLCK	No record locks available
ENOSYS	Function not implemented

ENOTEMPTY	Directory not empty
ENOSTR	Device not a stream
ENODATA	No data available
ETIME	Timer expired
ENOMEDIUM	No medium found
EMEDIUMTYPE	Wrong medium type

DEVSTATS

This structure is actually a union, since a device can either be a block or character type device. Because it is a union the nType field overlaps with the nType field of both the structure types for Block or Character.

For the File System, both the Flash Driver and FAT File System Driver are Block Devices.

nType	Identifies the device type, either block or character.
blkStats	Block device structure
chrStats	Character device structure

BLOCK_DEVSTATS

nType	Identifies the device type, either block or character.
nClass	Device classification
nBlockSize	Size of blocks (in bytes) for the device
nTotBlocks	Total number of blocks in device
nFreeBlocks	Total number of available blocks

CHAR_DEVSTATS

nType	Identifies the device type (.
nClass	Device classification
nTotalChars	Total number or maximum number of characters
nAvailChars	Number of current available characters

Exception Handling

There is builtin exception handling to the drivers and the functionality on the working of these can be found in the file, *platform.h*.

There are two implementation methods used based upon what resources the development platform contains:

- METHOD 1: SETJMP / LONGJMP Pair

Traditionally, the ANSI-C method of doing exception handling by method of non-local gotos. If there is no support non-local gotos in the development platform, then the 2nd method must be used.

- METHOD 2: GOTO

This violates methods of traditional structured programming practices, but since speed is relevant in this case, sometimes loop nesting prevent a more optimal solution.

FUNCTIONALITY

_TRY	
_CATCH(e)	
_AND_CATCH(e)	
_END_CATCH	
_THROW(e)	
_THROW_NULL()	

DEFINED EXCEPTIONS

Exception_Unhandled	
Exception_None	
Exception_Default	
Exception_Memory	
Exception_Dispatch	
Exception_File	
Exception_Device	

Example

The following example is just a basic usage of the Exception Handling macros implemented throughout the File System Driver.

```
int main()
{
    _TRY
    {
        ....

        _THROW(Exception_Memory);

        ....
    }
    _CATCH(Exception_File)
    {

        /* Handle File specific errors here */

    }
    _AND_CATCH(Exception_Memory)
    {

        /* Memory specific exceptions are handled here */

    }
    _END_CATCH

    return 0;
}
```

Appendix B

Memory Requirements

The following tables are the memory requirements necessary for each object allocated in the system and the maximum number of each object.

Object	Maximum Allocated	Persistence	Memory Required (in BYTES)
DRIVER_OBJ	2 total (one for Atmel Flash devices and one for FAT volume devices)	Until driver unload	48 (24 for each one)
ATMEL_FLASH_DEVOBJ	1 per flash device	Until device unload	28
FAT_DEVOBJ	1 per FAT volume	Until device unload	188 + (2 * Sector Size)
DEV_IOPACKET	1 per request	Until I/O completion	32
IOENTRY	1 per FAT volume	Until device unload	40
File Descriptors	FILEDESC_MAX (defined in file_api.c)	Continuous	4
FCB	Maximum number of descriptors allowed	Until descriptor releases	64
FAT_FILEOBJ	Maximum number of descriptors allowed	Until file closes or descriptor releases	84 + Sector Size

NOTE: Pointers and WORD size are assumed 32 bits.

The table does not account for any buffers locally used to read and write data, only internal caching.

So assume a file system that allows a maximum of 16 files open on it at any one time and an I/O request is being performed on each open file. Also assume that the sector size is 512 (which by default for this file system is the case), then the total number of bytes required is as follows:

Object	Allocated	Memory Required
DRIVER_OBJ	2	48
ATMEL_FLASH_DEVOBJ	1	28
FAT_DEVOBJ	1	1212
DEV_IOPACKET	16	512
IOENTRY	1	40
File Descriptors	16	64
FCB	16	1024
FAT_FILEOBJ	16	9536
TOTAL		12464 or 12.17 KB

NOTE: This does not take into consideration of internal I/O requests nor the buffers allocated for each I/O packet.

Appendix C

Examples

Doing a UNIX “ls” (directory listing)

This function was written and tested on a Windows platform emulating the Flash File System environment. There is a non-printable version that puts the entire directory into a string provided in *support.c*.

```
#define LIST_LINELEN      80
#define LIST_CRLF        "\x0d\x0a"

DEV_LONG ListDir(
    DEV_STRING strPath,
    DEV_STRING* pstrListing
)
{
    DIRDESC _hDir;
    DIRECT _direct;
    DEV_STRING _strFile;
    DEV_STRING _strCur;
    DEV_LONG _nEntries = 0;

    /*
     * Debug
     */

    DEVOBJ_ASSERT( strPath != 0 );
    DEVOBJ_ASSERT( pstrListing != 0 );

    /*
     * Open the directory
     */
    do
    {
        _hDir = sys_opendir( strPath );
        if ( _ISERROR(_hDir) )
        {
            break;
        }

        /*
         * Count the number of Entries
         */

        while ( sys_readdir( _hDir, &_direct ) == ESUCCESS )
        {
            if ( _direct.d_ino != 0 )
            {
```

```

        _nEntries++;
    }
}

if ( _nEntries == 0 )
{
    *pstrListing = 0;
    break;
}
sys_rewinddir( _hDir );

/*
 * Allocate the buffer
 */

*pstrListing =
    (DEV_STRING)DEVOBJ_MALLOC( LIST_LINELEN * _nEntries );
if ( *pstrListing == 0 )
{
    break;
}
*pstrListing[0] = '\0';
_strCur = *pstrListing;

/*
 * Now for each entry
 */

_strFile = DEVOBJ_MALLOC( DEVOBJ_STRLEN(strPath) + DIRSIZE );
if ( _strFile == 0 )
{
    break;
}
while ( sys_readdir( _hDir, &_direct ) == ESUCCESS )
{
    STAT _stat;

    /*
     * Valid Entry?
     */

    if ( _direct.d_ino != 0 )
    {
        DEVOBJ_STRCPY( _strFile, strPath );
        DEVOBJ_STRCAT( _strFile, "\\\" );
        DEVOBJ_STRCAT( _strFile, _direct.d_name );

        /*
         * Format the DATE and TIME
         */

        if ( !_ISERROR( sys_stat( _strFile, &_stat ) ) )
        {
            register DEV_LONG _nLen;
            DEV_LOCALTIME* _pLocalTime;

            DEVOBJ_LOCALTIME( &_stat.st_mtime, _pLocalTime );

```

```

/*
 * Concatenate Date
 */

DEVOBJ_STRCPY( _strCur, "00" );
DEVOBJ_LTOA( _pLocalTime->tm_mon + 1, _strFile, 10 );
_nLen = DEVOBJ_STRLEN(_strFile);
if ( _nLen < 2 )
{
    *( ++_strCur ) = *_strFile;
}
else
{
    DEVOBJ_STRCPY( _strCur, _strFile );
}
_strCur += _nLen;
*_strCur++ = '/';

DEVOBJ_STRCPY( _strCur, "00" );
DEVOBJ_LTOA( _pLocalTime->tm_mday, _strFile, 10 );
_nLen = DEVOBJ_STRLEN(_strFile);
if ( _nLen < 2 )
{
    *( ++_strCur ) = *_strFile;
}
else
{
    DEVOBJ_STRCPY( _strCur, _strFile );
}
_strCur += _nLen;
*_strCur++ = '/';

DEVOBJ_LTOA(
    _pLocalTime->tm_year + 1900,
    _strFile, 10
);
DEVOBJ_STRCPY( _strCur, _strFile );
_strCur += DEVOBJ_STRLEN(_strFile );
DEVOBJ_STRCPY( _strCur, " " );
_strCur += 2;

/*
 * Concatenate Time
 */

DEVOBJ_STRCPY( _strCur, "00" );
DEVOBJ_LTOA( _pLocalTime->tm_hour + 1, _strFile, 10 );
_nLen = DEVOBJ_STRLEN(_strFile);
if ( _nLen < 2 )
{
    *( ++_strCur ) = *_strFile;
}
else
{
    DEVOBJ_STRCPY( _strCur, _strFile );
}
_strCur += _nLen;
*_strCur++ = ':';

```



```

DEVOBJ_STRCPY( _strCur, "00" );
DEVOBJ_LTOA( _pLocalTime->tm_min + 1, _strFile, 10 );
_nLen = DEVOBJ_STRLEN(_strFile);
if ( _nLen < 2 )
{
    *( ++_strCur ) = *_strFile;
}
else
{
    DEVOBJ_STRCPY( _strCur, _strFile );
}
_strCur += _nLen;
*_strCur++ = ':';

DEVOBJ_STRCPY( _strCur, "00" );
DEVOBJ_LTOA( _pLocalTime->tm_min + 1, _strFile, 10 );
_nLen = DEVOBJ_STRLEN(_strFile);
if ( _nLen < 2 )
{
    *( ++_strCur ) = *_strFile;
}
else
{
    DEVOBJ_STRCPY( _strCur, _strFile );
}
_strCur += _nLen;
*_strCur++ = ':';

DEVOBJ_STRCPY( _strCur, "00" );
DEVOBJ_LTOA( _pLocalTime->tm_sec + 1, _strFile, 10 );
_nLen = DEVOBJ_STRLEN(_strFile);
if ( _nLen < 2 )
{
    *( ++_strCur ) = *_strFile;
}
else
{
    DEVOBJ_STRCPY( _strCur, _strFile );
}
_strCur += _nLen;

/*
 * Type of FILE and Size
 */

if ( (_stat.st_mode & _SMODE_IFMT) == _SMODE_IFDIR )
{
    DEVOBJ_STRCPY( _strCur, "    <DIR>    " );
    _strCur += 20;
}

else
{
    DEVOBJ_LTOA( _stat.st_size, _strFile, 10 );
    DEVOBJ_STRCPY( _strCur, "    " );

    _nLen = DEVOBJ_STRLEN(_strFile);
    _strCur += ( 19 - _nLen );
}

```

```

        DEVOBJ_STRNCOPY( _strCur, _strFile, _nLen++ );
        _strCur += _nLen;
    }
}

/*
 * Finally the FILE NAME
 */

DEVOBJ_STRCPY( _strCur, _direct.d_name );
_strCur += DEVOBJ_STRLEN( _direct.d_name );
DEVOBJ_STRCPY( _strCur, LIST_CRLF );
_strCur += DEVOBJ_STRLEN( LIST_CRLF );
}
}

DEVOBJ_FREE( _strFile );
}
while(0);

/*
 * And CLOSE OUT
 */

sys_close( _hDir );

/*
 * Return the total number of valid entries
 */

return _nEntries;
}

```

localtime

One of the problems with the C-library version of `localtime` is that, like `strtok`, it uses a static variable that returns a pointer to the caller:

```
struct tm* localtime( time_t* timer )
```

Synchronization is a problem in this regard, unless a guard in the form of a mutex is wrapped around the call, allowing only one task access at anyone time. The problem there is, the client or caller would have to provide the mechanism by acquiring the mutex, then releasing it when finished, otherwise a second thread could inadvertently call the function, corrupting the previous callers information, although minimally.

Included in the file `support.c` (for show only) is an example of implementing the `localtime` function as the following more thread-safe version:

```
DEV_LOCALTIME* LocalTime(
    DEV_TIME nTime,
```

```

        DEV_LOCALTIME ptmTime
    )

```

```

/*
 * Constants
 */

#define _SECONDSPERMINUTE      60L
#define _MINUTESPERHOUR       60L
#define _HOURSPERDAY          24L

#define _SECONDSPERHOUR       ( _MINUTESPERHOUR * _SECONDSPERMINUTE )
#define _SECONDSPERDAY        ( _SECONDSPERHOUR * _HOURSPERDAY )

#define _DAYSPERWEEK           7L
#define _DAYSPERYEAR           365L
#define _MONTHSPERYEAR        12L

enum _MONTHSOFYEAR {

    eJanuary = 0,
    eFebruary,
    eMarch,
    eApril,
    eMay,
    eJune,
    eJuly,
    eAugust,
    eSeptember,
    eOctober,
    eNovember,
    eDecember,

    eTotalMonths
};

enum _DAYSOFWEEK {

    eSunday = 0,
    eMonday,
    eTuesday,
    eWednesday,
    eThursday,
    eFriday,
    eSaturday,
    eTotalDays
};

DEV_LOCALTIME* LocalTime(
    DEV_TIME nTimer,
    DEV_LOCALTIME* ptmTime
)
{
    static DEV_UINT s_nMonthDay[] = {
        31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };
    register DEV_INT _nIdx;

```

```

/*
 * Debug
 */

DEVOBJ_ASSERT( ptmTime != 0 );

/*
 * The time is based on the UNIX time, which is seconds since
 * January 1, 1970
 *
 * Get seconds offset within minute (convert to minutes)
 * Get minutes offset within hour (convert to hours)
 * Get hours offset within day (convert to days)
 */

ptmTime->tm_sec = DEVOBJ_REM(nTimer, _SECONDSPERMINUTE);
nTimer /= _SECONDSPERMINUTE;

ptmTime->tm_min = DEVOBJ_REM(nTimer, _MINUTESPERHOUR);
nTimer /= _MINUTESPERHOUR;

ptmTime->tm_hour = DEVOBJ_REM(nTimer, _HOURSPERDAY);
nTimer /= _HOURSPERDAY;

/*
 * nTimer is now set to the number of days (calculate the day of the
 * week, since Jan 1, 1970 was on a "Thursday (day 4)"
 */

ptmTime->tm_wday = DEVOBJ_REM(( nTimer + eThursday ), _DAYSPERWEEK);

/*
 * Now for the DATE Part, adjust to days and remember to calculate
 * leap years.
 */

ptmTime->tm_year = nTimer / _DAYSPERYEAR;
nTimer = DEVOBJ_REM(nTimer, _DAYSPERYEAR);

/*
 * NOW adjust the time based on Leap Year information
 */

nTimer -= (( ptmTime->tm_year + 2 ) / 4 );

/*
 * Is current year a Leap Year?
 *
 * Adjust year since it is year since 1900 (i.e. add 70)
 * and set the YEAR Day
 */

ptmTime->tm_year += 70;
ptmTime->tm_yday = nTimer;
if ( DEVOBJ_REM(ptmTime->tm_year, 4) == 0 )
{
    s_nMonthDay[ eFebruary ] = 29;
}

```

```

else
{
    s_nMonthDay[ eFebruary ] = 28;
}

/*
 * Calculate Month/Day
 */

_nIdx = 0;
while (1)
{
    if ( nTimer < (DEV_TIME)s_nMonthDay[_nIdx] )
    {
        ptmTime->tm_mday = nTimer + 1;
        break;
    }

    nTimer -= s_nMonthDay[_nIdx];
    ptmTime->tm_mon;

    /*
     * Next Month
     */

    _nIdx++;
}

/*
 * The Month is wherever the Index currently is (Will never reach
 * _MONTHSPERYEAR or shouldn't)
 */

ptmTime->tm_mon = _nIdx;
ptmTime->tm_isdst = -1;

return ptmTime;
}

```

Index

A	
Appendix A Miscellaneous	61
Appendix B Memory Requirements	66
Appendix C Examples	68
B	
BLOCK_DEVSTATS	63
Boot Sector	4
C	
C Language Cross Reference	19
CHAR_DEVSTATS	64
Character Functions	16, 19
Compatible errno codes	61
D	
Data Caching	7
Device Driver	14
Device Driver Table	14
Device Model	10
Device Object	11
DEVICE_IO_CLOSE	55
DEVICE_IO_CONTROL	56
DEVICE_IO_OPEN	55
DEVICE_IO_READ	56
DEVICE_IO_WRITE	56
DEVSTATS	63
Direct	24
Directory Operations	23
Disclaimer	15
Dispatch Operations	55
Driver Object	10
Driver Table	14
E	
Error Correction	8
Example localtime	72
Example UNIX is	68
Exception Defines	65
Exception Example	65
Exception FUNCTIONALITY	64
Exception Handling	64
F	
FAT	5
Fat Control Block	51
FAT Determination	59
Fat Device Extension	53
Fat File Extension	54
FAT Overview	4
FAT32	6
File Object	54
File Operations	23
File System API	22
File System Driver	53
filestat	58
File-System Methods	22
Flash Reliability Enhancements	6

format	57
FSD Objects	53
I	
I/O Dispatch Interface	12
I/O Manager	49
I/O model Overview	9
I/O packet	13
I/O Table	49
Implementation	51
Index	76
Initialization	50, 60
Integration and Usage Tips	59
Introduction	4
IO_DISPATCH Method	12
IOEntry	49
L	
lseek	57
M	
Memory Management	16, 19
Memory Requirements	21
Miscellaneous Operations	23
mkdir	57
mount	56
P	
Partitioning	6
Platform Compatibility	16
Product Perspective	9
R	
readdir	58
rename	57
rmdir	58
Root	5
ROOT	59
Run-Time Errors	6
S	
Stat	24
String Management	17, 20
Structures	24
Synchronization	21
Synchronization Objects	18
sys_close	26
sys_closedir	27
sys_creat	28
sys_devstats	29
sys_eof	30
sys_format	31
sys_lseek	32
sys_mkdir	33
sys_mount	34
sys_open	35
sys_opendir	37
sys_read	38
sys_readdir	39

sys_remove	40
sys_rename	41
sys_rewinddir	42
sys_rmdir	43
sys_stat	44
sys_tell	45
sys_unlink	47
sys_unmount	46
sys_write	48
System Programmers Interface	26
System Requirements	16

T

Time Management	17, 20
-----------------------	--------

Types	25
-------------	----

U

unmount	57
---------------	----

V

Volume Operations	22
-------------------------	----

W

Wear Leveling	7
---------------------	---