AP-620

APPLICATION NOTE

LFS File Manager Software: LFM

SAMUEL DUFOUR
MEMORY COMPONENTS DIVISION

DEBORAH SEE
SENIOR SOFTWARE ENGINEER

October 1995

Order Number: 292175-001

Information in this document is provided solely to enable use of Intel products. Intel assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of Intel products except as provided in Intel's Terms and Conditions of Sale for such products.

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local Intel sales office or your distributor to obtain the latest specifications before placing your product order.

MDS is an ordering code only and is not used as a product name or trademark of Intel Corporation.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

*Other brands and names are the property of their respective owners.

Additional copies of this document or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

or call 1-800-879-4683

© INTEL CORPORATION 1995

CG-041493

1.0 INTRODUCTION

A simple method for storing variable sized files on an Intel Flash PC CARD or RFA (Resident Flash Array) is to use a Linear File Store format recently standardized by PCMCIA–the Linear File Store (LFS) File Manager (LFM). LFM is well-suited for embedded applications that would like to have a simple, operating system independent, file-object header structure and contiguous (non-fragmented) file objects without the overhead associated with other linked list filing systems. LFS was originally developed to accommodate storage of objects such as: eXecute In Place (XIP) code, photographic images, audio messages, and small-object-store (used for WinPad*, Magic Cap*, etc.). Intel's Memory Components Division has developed reference software to assist OEMs in customizing an LFM solution suited for nonvolatile storage of data.

The distinction to be made between LFS and LFM is that LFS is simply a well-defined file object storage partition (file header) contained within LFM; these storage partitions are termed "devices." LFM, on the other hand, is Intel's implementation of the PCMCIA-defined LFS specification. The LFM acts as a mini-file system providing basic file system functionality for reading and writing different-sized file objects. LFS stores file objects contiguously in a device and arranged in a one-way linked list. The first 32 bits of each file object make up the LFS header. The header consists of basic file information in addition to a link to the following file object (see Figure 1). The LFM reference code consists of several pieces—an LFS File System Driver (LFS – FSD), Flash Media Manager (FFM), and a low-level driver that interfaces to flash (see Figure 2).

NOTE

Throughout the context of this document and the code definitions, the term "device" is used to mean a LFS data partition rather than a single flash component. A LFS device can be comprised of a portion of a single component, or extend over several flash components. The size of each device (or partition) is defined by the user.

List of Abbreviations Used in This Document:

API - Application Programming Interface

FM- Flash Manager

FMM - Flash Media Manager

FSD - File System Driver

LFM - LFS File Manager

LFS - Linear File Store

OS - Operating System

PIA - Primary Initialization Area

POST - Power Off Self Test

RFA - Resident Flash Array

SIA - Secondary Initialization Area

XIP - eXecute In Place

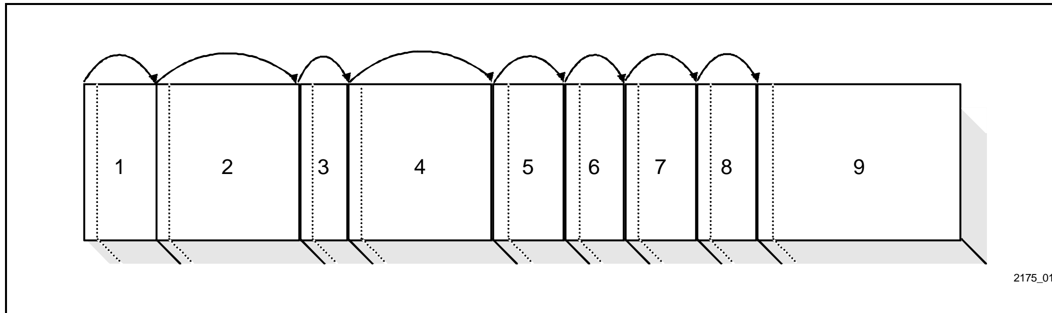

Figure 1. Linear File Store

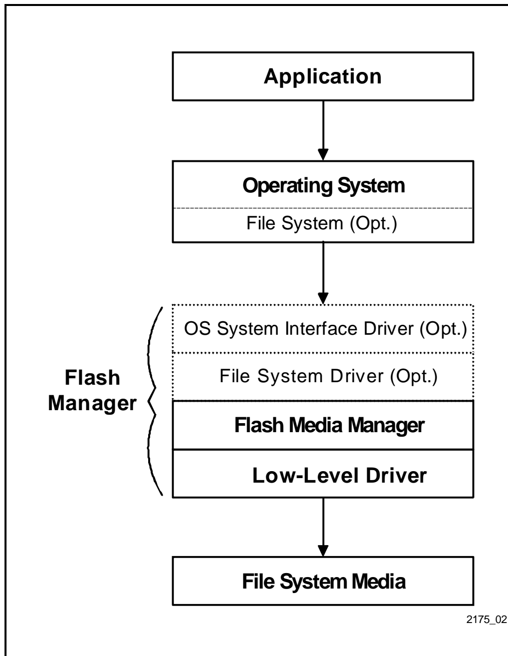

Figure 2. Application's Request Reaching the Device

The following sections will offer an in-depth look at the workings of LFM and explain concepts behind customizing the LFM code, as well as how an actual design was accomplished using the Intel provided LFM reference code.

2.0 LFM IN A "REAL LIFE" APPLICATION

Intel recognizes that flash technology, while providing many benefits in terms of cost-effective, reliable, high-performance nonvolatile storage presents some software challenges; thus Intel's Memory Components Division (MCD) has focused on developing software tools and reference code to help customers apply their technology. Thus, when confronted by a third-party vendor looking to design and implement a file I/O system for a network router (currently in mass production), Intel offered its help. The router's designer chose Intel Flash memory over other storage solutions for its device organization, density and availability. This flash memory would allow for storage of the router's system executables, configuration data files, run-time data files and run-time control files; however, the RFA (Resident Flash Array) targeted for use in this system needed suitable software to be able to manage these files. MCD set out to find a solution for this customer that would meet their non-volatile storage needs, minimize the time to market, and remain low in cost.

The OEM initially considered, in the tradition of older embedded systems, not to provide a file I/O system to manage the flash memory. However, the router's development team was charged with ensuring compatibility to other product lines resulting in the requirement of maintaining file system compatibility. Furthermore, the lack of a file system would reduce the potential feature set of the system. The customer realized a file system was necessary for maintaining the organization of data, allowing for the deletion and creation of files and to provide an Application Programming Interface (API) for both the system's boot device and existing application software; specifically, a Flash Manager (FM) is necessary for providing standard file capabilities, i.e., reading, writing, deleting, etc., while handling the peculiarities of the flash memory devices as opposed to rotating magnetic media.

Unfortunately, most of the commercially available FM's assume a particular operating environment. These FM's assume that MS-DOS is the resident OS, and as such, the FM must utilize the standard BIOS interface; however, this assumption did not apply to this situation. The customer was uncertain which platforms and operating systems would be used by the router.

Fortunately Intel, having recognized the lack of file system software available to pure embedded systems developers, developed the Linear File Store (LFS) File Manager module. When presented with the solution, the vendor concluded that LFM would greatly exceed expectations. The customer acquired the reference LFM code and with only slight alterations and minimal time, tailored the code to work with the router's RFA.

> ***With only slight alterations and minimal time the customer's LFM code worked with the router's flash RFA.***

The following sections describe the process by which Intel's LFM reference code can be modified to fit designs using Intel Flash to store files with low update frequency or needing XIP capabilities. Accompanying sections will illustrate parallels between the general procedure Intel prescribes for modification and use of the LFM code, and how the developer of the router followed these guidelines in customizing the code to meet the router's file system requirements.

2.1 Requirements of Example Target Application Covered by LFM

The router's designer considered the following issues important to the router's design, and LFM met each of these concerns as you will see in the following sections.

- Quick time-to-market for OEMs
- Robust power-off recovery
- Backward-compatible with targeted device's existing application software
- File name and file date retention
- Portable to any OS and/or hardware platform
- Supports multiple flash chips

2.2 Implementation Specific Details of the OEM's Router

In designing the router, two Intel components were key to providing a robust file I/O system. These two components were the Intel 28F008SA FlashFile™ memory (several components arranged as an Resident Flash Array or RFA) and an Intel 28F001BX boot block device. The router comes equipped with either 2 MB, 3 MB, or 4 MB of flash in the RFA and a single boot block of 128 KB. This unique design makes use of the boot block device to access the RFA for the purposes of initializing the FMM, loading operational software and writing copies of the operational software to DRAM for execution, and for system installations or upgrades.

The architectural design of the boot block allowed the developers of the router to create a multi-function device (see Figure 3 for specifics). The first 8-KB boot partition of the device houses the Primary Initialization Area or PIA. The following two 4-KB parameter blocks allow for data storage. The first parameter block holds records that are updated as frequently as every 10 minutes. The second parameter block is set aside to be used for future updates to the Secondary Initialization Area or SIA. The last 112-KB region is used for the SIA which is responsible for updating the system's operational software. For the purposes of reducing code size and complexity, the upper level interface (the Flash Manager Application Program Interface or FM API) was not included in the SIA. The router's own Operational Software contains the FM API.

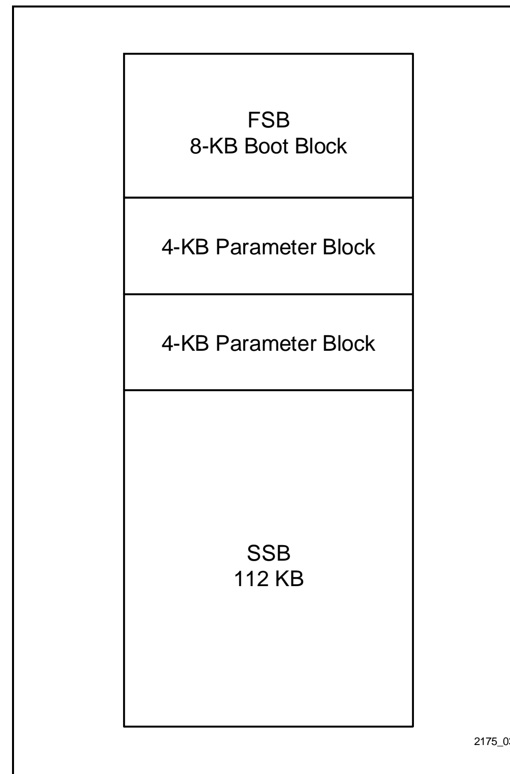

Figure 3. Boot Block Diagram

Upon power-up of the router, the PIA (located in the first 8 KB of the boot block) initializes the CPU and the rest of the system. Once the PIA finishes, it gives control to SIA. The SIA then attempts to initialize the LFS. If the initialization works, the system proceeds in attempting to boot the system, (i.e., load the OS via calls to cfs_open, cfs_read, etc.). If the initialization of the LFS fails, then the system proceeds to automatically format (erase) the RFA. As a result, the system knows that no OS files are present in the RFA, so the system then drops into an X-Modem session, requesting OS files to be presented. Since media updates rarely occur in RFA systems this automatic format happens quite infrequently (when updating due to bug fixes and upon the very first power-up of the system). After formatting, the particular device gets mounted.

Power On Self Tests (POSTs) determine the size of the RFA, as well as other memory resources. In sizing the media to be mounted, the system sends intelligent identifier commands to the flash components. Using the device type to determine the array size, devices are set up by using a table based on array size. Running the POSTs is one of the SIA's main responsibilities. This operation also stores the size of the installed RFA for future references. The low-level device driver makes use of this sizing data and stores it in a device size list. The router's version of the low-level device driver subsequently uses this information to properly initialize itself and the LFS. Finally, if an OS is present in the RFA, the system makes a call to CFS_OPEN to open the file, CFS_READ to read the file, and CFS_CLOSE closes the file then attempts to boot.

Once up and operative, any file calls made to the RFA are passed through the FSD API and handled by the LFM and its low-level driver.

The system must bridge and route data quickly; thus the developer minimized processor usage for background tasks such as reclaiming deleted file space and the overhead associated with the filing system software by utilizing multithreaded processes (multitasking events).

Furthermore, a scrambled file system, due to an unexpected power-down, would be totally unacceptable. The router would not be able to power-up correctly without an intact file system. Thus the customer made use of LFM's robust power-down recovery capabilities. Upon initialization LFM closes and marks deleted any files which were left open for write at power-off. It also tracks the state of any reclaims to allow the reclaim to complete on the next initialization if power is lost.

The customer made use of LFM's robust power-down recovery capabilities.

3.0 ARCHITECTURAL VIEW OF HOW LFM SOFTWARE MANAGES FLASH MEDIA

The basic ingredient needed to create an operational FM under LFS consists of the FSD which has been designed with the intent of re-usability; this means that the core FSD remains the same, regardless of the OS, processor hardware, or flash storage media configuration (card versus RFA). To use the FSD, however, a low-level hardware driver that responds to the FSD's low-level interface must be linked to the FSD. In using the FSD as a file system, as was the case in the implementation of the router, an operating system interface must be linked as well. Intel Memory Components Division provides a generic LFS File System Driver (FSD) as a direct implementation of the LFS concept (see Figure 2). This driver is not a file system, but serves as a device driver (written in ANSI "C") which may be linked to other pieces of code to provide a File System Interface to the OS.

The following sections will follow a hierarchical path from the file system interface driver down through the file system driver to the lower level driver interface and finish with the low-level driver (refer to Figure 2).

3.1 General Operating System Interface

To serve as a File System, the FSD needs an Operating System Interface to the OS. This Operating System Interface is different for each OS; programmers familiar with DOS would recognize this as the network redirector interface. However, there may or may not be a need for a file system interface driver. Basically, all that this driver is used for is a layer of translation between the OS or application and the FSD. If this "API" is not used then the FSD can be thought of as a library of functions.

In creating an API, it must be understood that the OS or application will make calls to the FSD that reference this API. In many cases this is just an extra step which may not be necessary. If the OS/App has already been written, then an API may be necessary. As was previously mentioned, the API would be written to translate the commands of the top layer OS/App into the functional file manipulation commands contained within the FSD. This API can provide FSD/hardware interchangeability.

3.1.1 ROUTER FILE SYSTEM INTERFACE

Since the customer's system was written in "C" and the test platform (a DOS platform) used the "C" standard I/O library (e.g., fopen(), fclose(), ...) the customer chose to provide a platform-dependent implementation of the functions contained within the FSD as an API. The API is a set of functions which comprise a subset of the "C" standard file I/O module. The actual naming convention used differed slightly to isolate their body of source code from the standard "C" functions.

By implementing an API between the OS and the FSD, the customer readily ensures backward compatibility to other existing product lines.

> ***API between the OS and the FSD readily ensures backward-compatibility.***

3.2 General Flash Manager Organization

The generic flash manager contains all of the file system interface functions necessary for basic file manipulation within flash media. The FSD portion of the LFM code should require no modification by the OEM, and it is required for all implementations.

The FSD has the capability to operate on multiple files; multiple files may be open for reading on a per device basis, and one file open for writing per device (partition). The maximum number of files open for reading per device must be predefined and the number of devices must be predefined.

Once a method has been established to identify files within multiple storage devices (our assumption stated above), the file system must be able to provide basic file operations: read, write, delete, etc. In an attempt to create a generic FSD, only the most basic functions exist. For added flexibility, a special pass-through function exists which allows custom functionality to be built into the FSD based on unique OEM needs.

The Flash Media Manger is a separate layer beneath the FSD. The primary purpose of the FFM is reclamation of flash media.

To better understand how the FSD and the FMM operates, observe Figure 2. Although we haven't discussed the low-level driver yet, the four components connected in the same box represent the complete FM. In this diagram, let's assume that the application is requesting a file operation from the OS. The OS routes the file operation request to the FSD. When the FSD receives the request, it needs to extract the device and the identifier for the file being accessed as well as the function to perform.

Once the OS's request has been translated into FSD function calls, the FSD utilizes the low-level driver to access the flash device. Refer to Sections 3.3 and 3.4 for greater detailed descriptions of the low-level driver and its interface to the FSD.

3.2.1 GENERAL FSD DATA STRUCTURES

Each file entry in an LFS device begins with the 32-byte header in Listing 1. This header contains status information about the file, the link to the next file and the unique IDs associated with the file.

```
typedef struct lfs_header {
DWORD   link;         /* Link to next header */
DWORD   size;         /* Size of LFS HEADER */
DWORD   type;         /* PCMCIA assigned type */
DWORD   offset;       /* Offset to file object */
DWORD   flags;        /* Flags (deleted) */
DWORD   stroff;       /* Offset to string  or extended header */
DWORD   id;           /* Unique ID */
DWORD   reserved;  /* Future */
} LFS_HEADER;
```

Listing 1. PCMCIA Defined LFS Header

Field	Description
link	This field contains the offset from the start of this header to the next LFS header in the device (partition). If each bit in this field is equal to bit D0 of the flags field, this is the last entry in the device.
size	This is the actual size of the LFS header.
type	PCMCIA requires that LFS headers contain a stamp indicating the type of the header. For our implementation, this field has been assigned ZERO (0), indicating the 32-byte header above.
offset	This field indicates how far from the start of this header into the entry the file data begins. This allows LFS implementations that use extended headers to be compatible with drivers that can't read the extended header.
flags	This is a bit-mapped flags field. Bit D0 indicates the nature of the flash (1 erase or 0 erase). Bit D1 indicates whether or not this file entry is valid or deleted: if D1 matches D0 the file is valid, if D1 differs, than that file is deleted.
stroff	This field points to the extended header associated with this file. The actual location of the filename string is determined by adding the value in this field to the address of the LFS_HEADER. If this field is zero, there is no filename.
id	This value is unique to each file object.

Notice there is no filename associated with the file. This is because not all operating systems will use a filename to locate records. However, under DOS, when an application requests a file access, its most basic form consists of: 1) a device indicator, and 2) a unique identifier. These two elements would look like "a:\readme.txt," where "a:" is the device indicator for that storage unit, and "\readme.txt" corresponds to the unique identifier. This pair could also look like "1301011989" where "13" is the device indicator and "01011989" is the identifier. In this case, the Unique ID may be the date that the file was recorded. In LFS, a file contains two identification fields: "type" and "ID." The type field is assigned to this header to distinguish it from other headers that may exist in the device. This prevents the FSD from reading a header it cannot interpret. The Unique ID field is to be used as a file identifier. If a true filename is desired, it may be stored in an extended LFS header referenced by "stroff." See Appendix A for a detailed schematic representation of each field.

Each FSD library function requires that a pointer to the following packet be passed to it. This "command structure" contains the following fields:

```
typedef struct cfs_ctrl {
DWORD   device;     /* Logical Device */
DWORD   status;     /* Return Status */
DWORD   buffer;     /* Buffer Address */
DWORD   count;      /* Transfer Count */
DWORD   actual;     /* Transfer Actual */
DWORD   scmd;       /* Sub-command */
DWORD   type;       /* PCMCIA EntryType */
DWORD   id;         /* PCMCIA Unique ID */
DWORD   aux;        /* Aux Data */
} CFS_CTRL;
```

Listing 2. Command Structure

Field	Description
device	This field indicates the device (partition) number to be acted upon. Device can also be thought of as a partition. This field is zero based.
status	This field provides a mechanism to pass a detailed failure back to the application. Typical functions will place a detailed error in this field and return SUCCESS or FAILURE back to the calling function.
buffer	This field provides a pointer to a memory buffer to translate data either to or from a function.
count	This field indicates the number of bytes transferred. This field is usually used on write operations.
actual	This field indicates whether or not a read or write is possible to a position by comparing the requested source file size with the actual size of the target position.
scmd	This field provides a subcommand for possible use in the future.
type	PCMCIA requires that LFS headers contain a stamp indicating the type of the header. For our implementation, this field has been assigned ZERO (0), indicating the 32 byte header above.
id	This value is unique to each file object.
aux	This field does not get initialized in the FlashDevMount function. It provides a mechanism to pass extra information to/from the low-level function set.

The status field returns the extended error code if the calling function returns a FAILURE or one (1). The sub-command field can be used to call a function for a specific purpose that it may support, such as the "CFS_special" command. The PCMCIA Entry Type has been given the value of zero (0) for this LFS implementation. This stamp indicates that the structure is the 32-byte PCMCIA defined LFS header. The Unique ID is supplied by the user.

Once a file has been opened or created, a File Info structure maintains the file pointer and other statistics of the file, much like the FILE structure in ANSI "C." An array of File Info structures will exist per device for files open for read and a single File Info structure will exist per device for files open for write.

The calls that the FSD makes to the low-level functions receive the Device Info structure for the device being accessed. All devices should be mounted at initialization. Mounting the target device fills the following structure with the appropriate data. The fields in this structure are filled in during the low-level "mount" command. Special functions will exist to modify these structures to allow removable media to indicate insertion or removal of devices.

```
typedef struct device_info {
DWORD   device;       /* Device # */
DWORD   status;       /* Device status */
DWORD   blocksize;  /* Size of blocks */
DWORD   numberblocks;  /* Number of blocks */
DWORD   lfs_offset;  /* Beginning of lfs device */
DWORD   lfs_size;     /* Size of lfs part (not inc spare blk) */
DWORD   lfs_end;      /* End of lfs part (not inc spare blk) */
DWORD   aux;          /* Aux data */
} DEVICE_INFO[MAX_DEVICES];
```

Listing 3. Device Info Structure

Field	Description
device	This field indicates the device (partition) number to be acted upon. Device can also be thought of as a partition. This field is zero based.
status	This field provides a mechanism to pass a detailed failure back to the application. Typical functions will place a detailed error in this field and return SUCCESS or FAILURE back to the calling function.
blocksize	This field should be initialized by the FlashDevMount function and provides a mechanism to allow multiple devices (partitions) with different Intel components (which may have different block sizes). This field should reflect the size of media that will be erased when one block is erased.
numberof blocks	This field indicates the number of blocks in the entire media. If their are two flash cards, each would be considered its own media. If the media is an RFA, each RFA is considered its own media.
lfs_offset	This field indicates the beginning address of the device (partition). If this is the first device on the media, the address follows the Media Status Table. If this is the second device on the media, the address follows the spare block of the first device.
lfs_size	This value should contain the usable portion of the device (partition). The boundaries of size not include the spare block.
lfs_end	This field indicates the address of the end of the current device(partition). This value should not include the spare block which is always assumed to be the block following the device.
aux	This field does not get initialized in the FlashDevMount function. It provides a mechanism to pass extra information to/from the low-level function set.

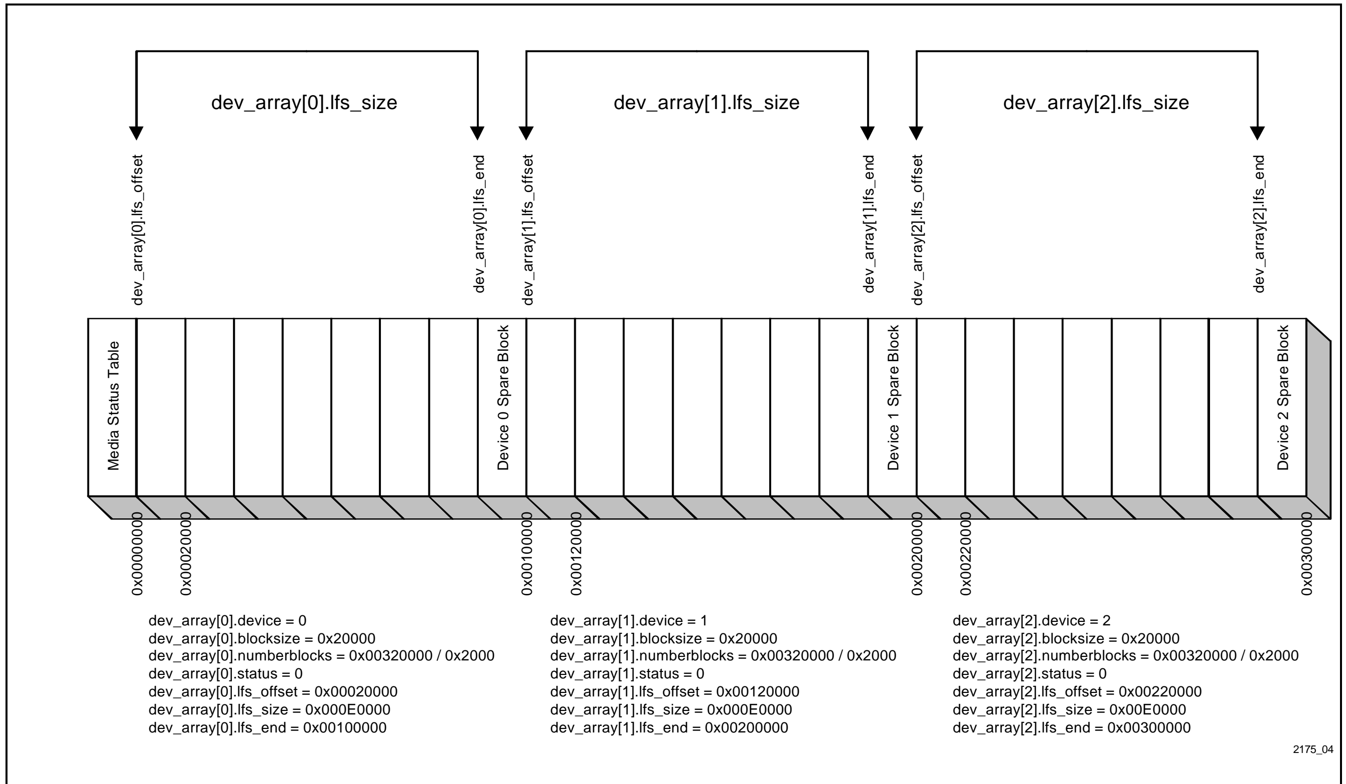

Figure 4. A Mounted DEVICE_INFO Array

3.2.2 EXAMPLE INITIALIZED DEVICE_INFO STRUCTURE

Figure 4 provides an example of a "mounted" DEVICE_INFO array. This information will need to be adapted to apply to your setup. If only one device (partition) is desired, change the MAX_DEVICES define as described in Section 5.1. This example is a Resident Flash Array (an array of flash components or RFA) in which only a portion of the array is used for LFM and three devices (or partitions) are defined.

3.2.3 ROUTER FSD ORGANIZATION

The developer of the router did not need to alter the File System Drivers in any way. However, there was a need to retain file names and file dates so the customer chose to implement an extended header (See Section 3.1.3.1 for header information). Adding an extended header took little more than adding an extra library to the FSD. The extended header was defined to be a structure containing two fields: 1) a character array containing the filename and, 2) a TIME structure which is the creation time and date of the file (the extended header can be customized to retain any data). This customized extended header allows for file names to be searched within specified flash devices. When a requested file is found, the corresponding FILE ID (a 32-bit integer), from the LFS header, will be returned (if the file exists) to allow access to the file.

The (Optional) Extended Header was customized to hold the file name and file time.

Another benefit derived from the FSD was that since all of the FSD (as well as the FSD API and low-level driver) was written in "C," the customer possessed a file system that was operating system and hardware independent. Hence, all doors were left open to future platform portability.

All doors were left open to future platform portability.

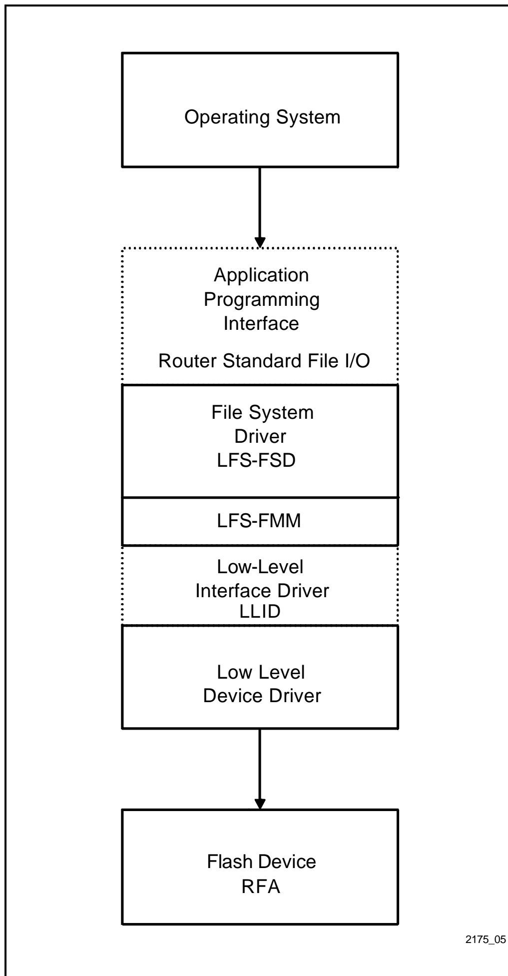

Figure 5. Router File System Structure

The general structure of the router's file system is depicted in Figure 5.

3.3 General Low-Level Interface Driver

All media level I/O occurs through the Low-Level Interface Driver (LLID). When the FSD needs to access the flash media, it makes a call to one of the low-level function calls which are supplied by the LLID. The basic functions associated with the low-level driver are: read, write, and special. An erase procedure exists within the special functions. Since the FSD was designed to work with flash media, an erase function for reclaiming deallocated space is essential.

3.3.1 ROUTER LOW-LEVEL INTERFACE DRIVER

All low-level function names are consistent with those required by the FSD. The customer simply followed Intel's naming conventions while coding the router's low-level drivers.

3.4 General Low-Level Driver

The low-level driver is responsible for accessing the device (RFA or PC CARD) in response to requests from the Operating System/Application. The low-level driver provides the low-level media operations such as read/write/erase as well as filling in the structures that the FSD needs to operate properly. The low-level driver's biggest responsibility is managing multiple devices. In the case of a fixed RFA, the low-level driver merely has to translate logical to physical addresses for the FSD. If removable media is present, the low-level driver must be able to identify the devices that exist on the media and handle all the hardware necessary to access the media. Figure 6 depicts the responsibilities the low-level driver must address for the two different media types (RFA or memory card).

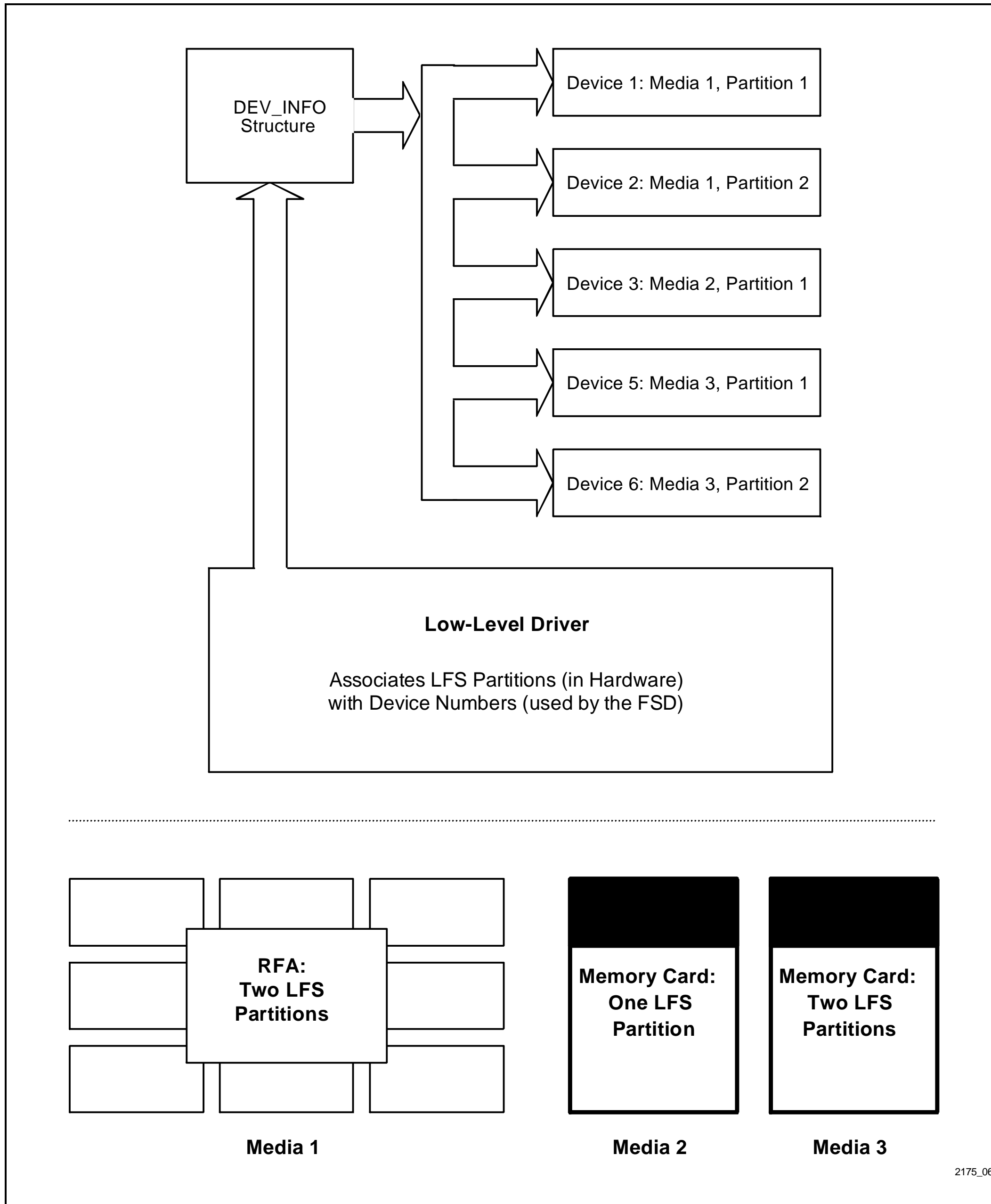

Figure 6. Low-Level Driver

The following sections represent important requirements of the low-level driver. In coding the low-level driver, designers customizing the LFM code must be certain to incorporate the following structures.

3.4.1 DEVICE ACCESS

Internally, the FSD may operate on multiple devices (partitions) at any time. This requires that the number of devices in the system be predetermined. This also requires that removable media must predefine the number of devices per slot.

The low-level driver contains whatever technology driver algorithms are needed to read, write and erase the flash media as well as fill in structures for the file system driver and any other special functions required.

3.4.2 MEDIA STATUS TABLE

There will be one Media Status Table defined per media. The Media Status Table will reserve one block and will be used during initialization and reclaim. The LFS Media Status Table is intended to assist in making the LFS File Manager more robust. It provides memory to store status updates when reclaim is occurring, so that the system may recover from an unexpected power-off. For a discussion of the Media Status Table and how it will assist in recovery from an unexpected power-off see Appendix E.

3.4.3 ROUTER LOW-LEVEL DRIVER

In creating a custom low-level driver, the OEM wrote two copies. One was used to verify the Intel LFS. The verification of the LFS was run on a PC and a binary disk file was used to emulate flash memory. The other copy was the router's low-level driver which was written from scratch; however, sample code was supplied by Intel which helped to reduce coding time. The developer chose to write the driver from scratch because there was a need for the source to meet internal coding standards and to make it easier to maintain.

In addition to the usual low-level device operations (e.g., read, write, erase, etc.), multi-tasking events were of great concern to this customer because utilization of the CPU's time for tasks other than bridging or routing data needed to be kept to a minimum. Therefore, in order to guarantee such CPU restrictions, the low-level device driver was designed to put a task that attempted to do a block erase or a large block write to sleep (since block writes can be relatively slow). Moreover, due to the router's system architecture, the type of "task sleep" varies depending on whether the flash reclaim is instigated automatically or via user input at the console. These multi-tasking modifications were fairly simple to make. Multiple partitioning of the router RFA allows the file system to act upon each device simultaneously; thus multiple devices help to decrease search time and allow more than one file to be open at a time.

> ***Multiple devices help to decrease search time and allow more than one file to be open at a time.***

4.0 REQUIRED FILES

The following table gives an overview of which files in the COMMON directory are necessary for your implementation. Each file is described more completely below. Files existing in the CARD directory are the low-level files necessary for an example 82365SL PCIC implementation on a DOS PC platform. The files existing in the RFA directory are the low-level files necessary for an example RFA using 28F008 components on a 486SL evaluation platform. Your implementation will need to replace the functionality in the low-level as indicated in Appendix C of this document.

Table 1. Files Included within LFM Reference Code

File Name	Necessary?
MULTIMAIN.C	NO
LFSSUB.C	NO
FSD.C	YES
RECLAIM.C	YES
CFSTOP.H	YES
CFSEXT.H	YES
LFSEXT.H	NO
LFSMGR.H	NO

4.1 MULTIMAIN.C

This file contains an application interface for testing all aspects of the filing system. This file is not required for your implementation, but should be replaced by either your own application, or an interface that translates your operating systems file commands into LFM file commands.

4.2 LFSSUB.C

This file contains subroutines for the example application. It is not required for your implementation.

4.3 FSD.C

This file contains the file system interface functions. It is required for all implementations. This file should require no modifications for your implementation.

4.4 RECLAIM.C

This file contains the subroutines for reclaim functionality. It is required for all implementations. This file should require no modifications for your implementation.

4.5 CFSTOP.H

This file contains error code defines, subcommand definitions, structure definitions, and any other information required by the user. Modifications necessary to this file are described below in Section 5.

4.6 CFSEXT.H

This file contains prototype information for all functions in the FSD.C and RECLAIM.C files. This file is required for all implementations. It should require no modifications for your implementation.

4.7 LFSEXT.H

This file contains prototype information for subroutines in LFSSUB.C and external declarations. It is not required for your implementation. The register structure definitions are Borland Library definitions, and do not need to be replicated for your implementation.

4.8 LFSMGR.H

This file contains a few defines that are application specific. It is not required for your implementation. If you do choose to use this file, please see the modifications necessary as described in Section 5.

4.9 Files Used by the Router

Of the files previously listed for use, the only files used to implement LFM for the router's RFA were: FSD.C and RECLAIM.C. However, some of the other ".h" files where renamed and included in the FSD.C file; these files contained definitions and type-definitions from the two required ".h" files CFSTOP.H and CFSEXT.H. All the definitions and type-definitions were still utilized from the files, but the router's developer chose to break them up into several files instead of the two they were contained within. A few headers were also added to "RECLAIM.C" for multi-tasking.

5.0 MODIFYING HEADER FILES

The following header files require some modification. Each definition requiring modification is described to assist modification to adapt to your implementation.

5.1 CFSTOP.H

#define BYTE unsigned char

This define is platform specific. The unsigned char should be replaced by any keywords that create an 8-bit value. This may be compiler specific.

#define WORD unsigned int

This define is platform specific. The unsigned char should be replaced by any keywords that create a 16-bit value. This may be compiler specific.

#define DWORD unsigned long

This define is platform specific. The unsigned char should be replaced by any keywords that create a 32-bit value. This may be compiler specific.

#define TUPLE_XXXX X

This group of defines can be removed for non-removable media. Tuples are part of the format of a media, which is not required for non-removable media.

#define MAX_DEVICES 3

This define determines the total number of devices (partitions) acceptable in the entire system. You should modify this variable to suit your needs. Multiple devices are useful if separating file types is necessary (it can be treated as a rudimentary sub-directory). If only one single device is required, set this value to one.

<u>**#define MAX_OPEN_READ 5**</u>

This define determines the maximum number of files that can be open for reading at any given time. You should choose a large enough number for your needs, however, make sure the number is not too large, or it may hide application errors (such as forgetting to close files).

<u>**#define MAX_PART_MEDIA 3**</u>

This define determines the maximum number of devices (partitions) that can be created on any one media. If you are using non-removable media and there is only one media (i.e., one RFA), then the define should equal MAX_DEVICES. If you are using multiple non-removable media's or removable media, it should be an equally divisible portion of MAX_DEVICES.

An example of how this define is used would be for a system that contains two PC CARD slots and would like to create one device on each card. For this example, MAX_DEVICES would equal two and MAX_PART_MEDIA would equal one. If this same system wanted to allow two devices per card, the MAX_DEVICES define would equal four, and the MAX_PART_MEDIA define would equal two.

<u>**#define TUPLES_BUFFER_LEN 300**</u>

Removable media requires the format information to be kept on the media. The software expects the first block on the media to contain the format (also known as tuples in PCMCIA). The Media Status Table (MST) that tracks the progress of reclaim is also kept in this first block. This define is used to indicate when the Media Status Table is growing too large and should be reclaimed. The MST is kept at the end of the first block and grows backwards toward the beginning. If non-removable media is being used, the buffer does not need to remain this large, however, it should not decrease below 150 bytes. This allows a buffer for cleanup of the MST. This is necessary because the MST cannot be checked for cleanup every time it is decremented due to the status of the spare block.

<u>**#define EXTERNAL_HEADER 1**</u>

The external header is a conditional option. If this define is kept, external headers will exist as you define them. If there is no desire for external headers, this define may be commented out and extended headers will not be used. If this definition remains, a pointer to an external header must be passed in on the OPEN_CREATE sub-command of the CFS_open function. This header is readable at a later time through the CFS_find command as well as the OPEN_OPEN sub-command of the CFS_open function.

The following is the currently defined external header:

```
typedef struct ext_header {
        BYTE    file_name[8];
        BYTE    file_ext[3];
}EXT_HEADER
```

This header may be modified to contain any elements you choose, such as time and date of file creation, compression ratio information, or any other information you may wish to link to the file.

5.2 LFSMGR.H

It is recommended that this file not be used if possible because these code segments exist within CFSTOP.H. However, if it is used, the defines that are replicated in CFSTOP.H must match those chosen in CFSTOP.H.

<u>**#define BYTE unsigned char**</u>

This define is platform specific. The unsigned char should be replaced by any keywords that create an 8-bit value. This may be compiler specific.

<u>**#define WORD unsigned int**</u>

This define is platform specific. The unsigned char should be replaced by any keywords that create a 16-bit value. This may be compiler specific.

<u>**#define DWORD unsigned long**</u>

This define is platform specific. The unsigned char should be replaced by any keywords that create a 32-bit value. This may be compiler specific.

6.0 SUMMARY

This application note has discussed the hierarchical design underlining Intel's LFM reference code as well as necessary modifications needed for porting LFM to designs requiring Flash Media Management. This application note has also given detailed examples of how an Intel customer used LFM and Intel Flash memory to create a robust file I/O system; LFM also met each of the designer's concerns.

7.0 ADDITIONAL INFORMATION

7.1 References

Order Number	Document
290429	28F008SA 8-Mbit Flash Memory Datasheet
290406	28F001 BX-T/28F001 BX-B 1-Mbit CMOS Flash Memory Datasheet

7.2 Revision History

Number	Description
-001	Original Version

7.3 Intel BBS*

Name	Description	BBS Location*	BBS filename & *Type*
LFM	**LFS** Flash Media **F**ile **M**anager. Uses PCMCIA defined LFS (Linear File Store) Spec for a linked-list type method of storing "Objects." For cards or a RFA/RFD. Includes iCARDRV1 source for reference.	Flash/FlashFile (Area 4-5) Flash/Flash Card (Area 5-5)	!LFM.EXE *"C" source* (same file as) !LFMCRD.EXE *"C" source*
iLFMDOS Redirector	A DOS installable file system for **L**inear **F**ile **S**tore partitions based on Intel's LFS Flash Media File Manager (LFM).	Flash/Flash Card (Area 5-5)	tbd
1Mb Boot Block Drivers	Reference code for read, write, and erase functions on Intel 1-Mb Boot Block Flash Components	Flash/Boot Block (Area 3-5)	BOOTDRV.ZIP *"C" & x86 ASM86 Source*
2 and 4-Mb Boot Block Drivers	Reference code for read, write, and erase functions on Intel 2 and 4-Mb boot block flash Components	Flash/Boot Block (Area 3-5)	24BOOTDR.EXE *"C" & x86 ASM86 Source*
8-Mb FlashFile™ Drivers	Reference code for read, write, and erase functions on Intel 8-Mb FlashFile components	Flash/FlashFile (Area 4-5)	8MBITDRV.EXE *"C" & x86 ASM86 Source*
16-Mb FlashFile™ Drivers	Reference code for read, write, and erase functions on Intel 16-Mb FlashFile Components	Flash/FlashFile (Area 4-5)	16MBDRV1.EXE *"C" & x86 ASM86 Source*

NOTE:
The Intel Applications Support BBS can be reached at: US/Canada/ Japan/ APAC...916-356-3600, Europe +44(0)793-49-6340

APPENDIX A
LFS HEADER FIELDS

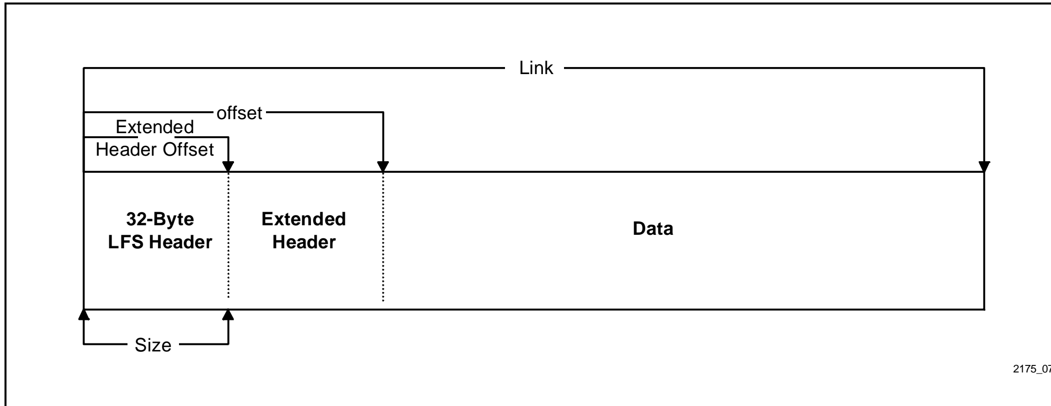

Figure 7. LFS Header Offsets

```
typedef struct lfs_header {

        DWORD           link;
        DWORD           size;
        DWORD           type;
        DWORD           offset;
        DWORD           flags;
        DWORD           stroff;
        DWORD           id;
        DWORD           reserved;

} LFS_HEADER;
```

LFS Header Offsets Structure

This schematic represents an instance of a 32-byte LFS header, its extended header, and the associated data region. This structure is described in Section 3.1.1.

APPENDIX B
FSD LIBRARY FUNCTIONS DEFINED: FILE SYSTEM INTERFACE DRIVER

The following library functions comprise the complete set of functions available within the FSD library. The details of each function are presented here to help familiarize the reader with the purpose and correct usage of each of these functions.

Each function call at the top level of the FSD takes a CFS_CTRL structure pointer as an argument. If the function fails, it returns a one (1) and updates the status field of the structure. The fields in the CFS_CTRL structure may be input or output, depending on the function call. The following table describes the usage of the fields for each function call (I=input, O=output, NA = not applicable).

	Open	Close	Read	Write	Find	Delete	Special**
Device	I	I	I	I	I	I	I
Status	O	O	O	O	O	O	O
Buffer	I	NA	I	I	I	NA	I/O
Count	NA	NA	I	I	NA	NA	I/O
Actual	NA	NA	O	O	NA	NA	I/O
SCMD	I	NA	NA	NA	I	NA	I/O
Type	I	I	I	I	O	I	I/O
ID	I	I	I	I	O	I	I/O
Aux	I	NA	NA	NA	NA	NA	NA

** = One or more of the sub-commands or user special functions use these fields.

int CFS_open(CFS_CTRL *)

Functionality

CFS_open opens a file for reading, or creates a file for writing. The user specifies a device, PC CARD type and a Unique ID, plus a sub-command. The sub-command (SCMD) indicates open-for-read or open-for-create. If open-for-read, the FSD tries to locate the file in the device and will return an error if it's not found. If open-for-create, the FSD will return error if the file exists.

Open can also return physical information about the file. If the "buffer" field contains a pointer to a data buffer, Open will fill the buffer with the FULL header information (FULL_HEADER). The FULL_HEADER consists of the LFS_HEADER and the EXTENDED_HEADER if one exists.

The user may add an extended header to give a file a filename or any other additional header information by passing a pointer to a pre-defined header structure through the aux field. The user may add a special user function to access this information and do any other special handling.

The status field will contain a detailed error code if the return indicates the routine failed.

Sub-command

OPEN_OPEN	Open for read (R).
OPEN_CREATE	Open for create (C).

Possible Return Error Codes

ERR_DEVICE	R/C
ERR_OPEN	R/C
ERR_EXISTS	C
ERR_NOTEXISTS	R
ERR_CREATE	C
ERR_PARAM	R/C
ERR_SPACE	R/C
ERR_READ	R/C
ERR_WRITE	C
ERR_NONE	R/C

int CFS_close(CFS_CTRL *)

Functionality.

If the file indicated by device, type, and ID was opened for write, the FSD writes the file's LFS header. Note that the LFS header for a newly created file is not written until the file is closed. This is due to the nature of flash memory; the "link" field in the header must be the total length of the file object including the header, which isn't known until all writing has been completed. The file information in the devices write file structure will be erased. If the file indicated by device, type, and ID was opened for read, the file information in the devices read file array will be erased. If a failure occurs, the status field will contain a descriptive error code.

Possible Return Error Codes

ERR_DEVICE
ERR_WRITE
ERR_PARAM
ERR_SPACE
ERR_READ
ERR_NOTEXISTS
ERR_NONE

int CFS_read(CFS_CTRL *)

Functionality

This function reads "count" number of bytes from the file specified by device, type, and ID and places them into "buffer." The file must be opened with the OPEN_READ option. The file read begins at the location of the last read or whatever file offset the special seek option set. If during the process there is a failure, the "actual" field indicates how many bytes were transferred and the status will contain a descriptive error code.

Possible Return Error Codes

ERR_OPEN
ERR_DEVICE
ERR_READ
ERR_NONE

int CFS_write(CFS_CTRL *)

Functionality

This function writes "count" number of bytes from "buffer" to the file specified by device, type, and ID. The file must be opened with the OPEN_CREATE option. If during the process there is a failure, the "actual" field indicates how many bytes were transferred and the status will contain a descriptive error code. The file write begins at the location of the last write.

Possible Return Error Codes

ERR_NOTOPEN
ERR_WRITE
ERR_DEVICE
ERR_SPACE
ERR_NONE

int CFS_delete(CFS_CTRL *)

Functionality

Calling this function with a device, PC CARD type and a Unique ID of a closed file will update its "flags" field to indicate it has been deleted. The space cannot be reused until a reclaim has been performed. Bit 0 is verified for erase state then bit 1 is set to the NOT of bit 0.

Possible Return Error Codes

ERR_NOTEXIST
ERR_NOTCLOSED
ERR_DEVICE
ERR_WRITE
ERR_READ
ERR_SPACE
ERR_NONE

int CFS_find(CFS_CTRL *)

Functionality

Find will locate the FIRST or NEXT file in the device. As with CFS_open, if "buffer" is non-zero, it will be filled with information from the FULL header (FULL_HEADER). The FULL_HEADER consists of the LFS_HEADER and the EXTENDED_HEADER if one exists. The type and ID will be filled in for the user to use in future calls with this file.

Subcommands

FIND_FIRST

FIND_NEXT

Possible Return Error Codes

ERR_NOTEXISTS
ERR_DEVICE
ERR_SPACE
ERR_READ
ERR_NONE

int CFS_special(CFS_CTRL *)

"Special" provides a mechanism for hardware direct and media specific operations. All file system specific operations will call routines within the file system level. All hardware specific operations will call the low-level special function. Some operations may do both. In addition, the Special function provides a way for the host OS to pass or receive user-specific information through the FSD to the low-level functions. The sub-command selects which special function will be called. The first 20 functions have been reserved for internal development. OEMs may use any special sub-command greater than 20. This procedure will call FlashDevSpecial() with the respective sub-command. The predefined functions are described in the low-level function section.

NOTE:

Before any other LFM function is called, a call to CFS_special with sub-command INIT must be made. Among other things, this call initializes the DEVICE_INFO structure which is necessary for all operations

SP_INIT

Functionality

The SP_INIT sub-command of the special function will initialize all DEVICE_INFO structures necessary for all operations. It will also initialize all file information structures, as well as closing and deleting any files which were left open at power-off.

Possible Return Error Codes

ERR_DEVICE
ERR_READ
ERR_NONE

SP_GET_INFO

Functionality

The SP_GET_INFO sub-command will allow the user to retrieve device information. The information is defined in the DEVICE_INFO structure and will be placed in memory pointed to by "buffer." Any failures will be described by status.

Possible Return Error Codes

ERR_DEVICE
ERR_PARAM
ERR_NONE

SP_FORMAT

Functionality

The SP_FORMAT sub-command will allow the user of removable media to format the media. This may be in the form of PCMCIA extended CIS structures or any other user specified method of device definition. Users of non-removable media will fill in the device info structures during initialization and will not need to place the information on the media. The device defines which device to format, and the count defines how large to make the device. This function should place the information on the removable media so it may be read at a later time.

Possible Return Error Codes

ERR_PARAM
ERR_READ
ERR_DEVICE
ERR_WRITE
ERR_NONE

SP_ERASE

Functionality

The SP_ERASE sub-command will erase a block of any device. The device should be specified and the pointer should point to an address in the block to be erased.

Possible Return Error Codes

ERR_DEVICE
ERR_ERASE
ERR_NONE

SP_SPACE

Functionality

The SP_SPACE sub-command will return the free space left on the device. If a file is currently open for write on the device, this file is not taken into account. If a file is marked as deleted, this space will not be available until a reclaim has been performed.

Possible Return Error Codes

ERR_DEVICE
ERR_SPACE
ERR_READ
ERR_NONE

SP_RECLAIM

Functionality

The SP_RECLAIM sub-command will reclaim all unused space on the device specified. Reclaim removes file objects that have been marked deleted and relocates valid file objects to create the maximum contiguous free space.

Possible Return Error Codes

ERR_DEVICE
ERR_READ
ERR_WRITE
ERR_ERASE
ERR_NONE

SP_ERASE_CARD

Functionality

The SP_ERASE_CARD sub-command applies to removable media and will erase all blocks on the card.

Possible Return Error Codes

ERR_DEVICE
ERR_ERASE
ERR_READ
ERR_NONE

SP_SEEK

Functionality

The SP_SEEK sub-command uses the device, type, and ID to determine if a file is open for read. If the file is open for read, the buffer will be used as a new offset into the file.

Possible Return Error Codes

ERR_DEVICE
ERR_PARAM
ERR_NOTOPEN
ERR_NONE

SP_TELL

Functionality

The SP_TELL sub-command uses the device, type, and ID to determine if a file is open for read. If the file is open for read, the current offset into the file will be placed into the buffer field and the file size will be placed into the aux field.

Possible Return Error Codes

ERR_DEVICE
ERR_NOTOPEN
ERR_NONE

SP_CARD_DETECT

Functionality

The SP_CARD_DETECT sub-command re-initializes all device info structures. This function applies only to removable media. This allows devices on the card which have been removed to have their device info structure set to values to indicate the device does not currently exist.

Possible Return Error Codes

ERR_DEVICE
ERR_READ
ERR_NONE

DEFAULT

Functionality

Calls the low-level special routine. This will give the user direct access to hardware level functions. The CFS_CTRL aux field will be placed into the DEVICE_INFO aux field to allow the user to pass information to the low-level special call. The DEVICE_INFO aux field will be placed into the CFS_CTRL aux field on the return to allow the user to return information.

Possible Return Error Codes

ERR_PARAM
ERR_NONE

APPENDIX C
LOW-LEVEL DRIVER INTERFACE FUNCTIONS

The following functions are supplied by the low-level driver (LOWLVL.C) and must adhere to FSD data structure error handling. Due to the unique characteristics of the hardware interface to flash created by individual OEM's, the following functions must be supplied by the OEM's low-level driver. If an error occurs, each function returns one (1) and updates the status field in the DEVICE_INFO field. If the implementation is for removable media, the CARD directory provides example low-level functionality. This example uses a DOS interrupt to utilize I/O control functions that exist in the ICARDRV1 driver included with the package. If the targeted implementation is for non-removable media, the RFA directory provides an example. Access to the RFA will probably be much simpler than the example provided.

int FlashDevRead (DEVICE_INFO * dev_ptr, DWORD offset, DWORD length, BYTE *buffer)

Read fills the specified buffer with the number of bytes defined by length from the device's absolute physical address. This function uses dev_ptr->device to determine device. The return code should indicate ERROR (1) or OK(0). If the return code indicates ERROR, the dev_ptr->status field should indicate the return error code.

This function must access the flash through the hardware interface on your system. If your system uses memory windows (i.e., only a portion of the flash is mapped into the main memory map at any given time), it must be accounted for in this algorithm.

Possible Return Error Codes

ERR_READ
ERR_NONE

int FlashDevWrite (DEVICE_INFO *dev_ptr, DWORD offset, DWORD length, BYTE * buffer)

This function writes length bytes of data from a specified buffer to the destination address within the device. This function uses dev_ptr->device to determine device. The return code should indicate ERROR (1) or OK(0). If the return code indicates ERROR, the dev_ptr->status field should indicate the return error code.

This function must access the flash through the hardware interface on your system. If your system uses memory windows (i.e., only a portion of the flash is mapped into the main memory map at any given time), it must be accounted for in this algorithm.

Possible Return Error Codes

ERR_WRITE
ERR_NONE

Int FlashDevErase (DEVICE_INFO * dev_ptr)

In order to reuse the flash media, an erase command must be provided for the FSD. This command erases a single flash erase-block beginning at the address specified by the aux field in the DEVICE_INFO structure. The blocksize field of the DEVICE_INFO structure is used to force the aux ptr to a block boundary. Removable media would use the device field to determine on which media the block should be erased. The device field of the DEVICE_INFO structure is used to determine which device the block resides in.

This function must access the flash through the hardware interface on your system. If your system uses memory windows (i.e., only a portion of the flash is mapped into the main memory map at any given time), it must be accounted for in this algorithm.

Possible Return Error Codes

ERR_ERASE
ERR_NONE

int FlashDevSpecial (DEVICE_INFO * dev_ptr, DWORD scmd)

All sub-commands perform the functions defined below. Other functions which are not specific to the hardware will be performed at the FSD level.

NOTE:

> Before any other LFM function is called, a call to CFS_special with sub-command INIT must be made. Among other things, this call initializes the DEVICE_INFO structure which is necessary for all operations.

SP_INIT

Calls the support functions FlashCompatCheck() to verify compatibility and FlashDevMount() to initialize all devices. All other initialization will occur at the FSD level. Returns error if all components are not Intel devices or if an error occurs. An error could occur in the FlashDevMount algorithm if a removable media is not present. This will be detailed in the FlashDevMount() function description in Appendix D. This function should remain the same as the RFA\CFSLOW.C version for all applications.

Possible Return Error Codes

ERR_JEDEC
ERR_DEVICE
ERR_READ
ERR_NONE

SP_FORMAT

This function is responsible for creating the LFS device for removable media only. Non-removable media should return either OK or ERROR, depending on the results expected by the high level calling application. Functions called for removable media include CreatePartition() and FlashDevMount(). This function should remain the same as defined in the CARD\CFSLOW.C file for removable media.

Possible Return Error Codes

ERR_PARAM
ERR_DEVICE
ERR_WRITE
ERR_READ
ERR_NONE

SP_ERASE

This function provides the erase block capability. Functions called include FlashDevErase(). This function should remain identical to either the CARD\CFSLOW.C or the RFA\CFSLOW.C versions.

Possible Return Error Codes

ERR_ERASE
ERR_NONE

SP_ERASE_CARD

This function allows the entire media to be erased. The size and location of the media is defined in the low-level by the user. This will allow the user to erase all partitioning on a media to start from an erased media. Functions called include Erase_Card() and FlashDevMount(). This function should remain as it is in both CFSLOW.C files.

Possible Return Error Codes

ERR_READ
ERR_ERASE
ERR_DEVICE
ERR_NONE

SP_CARD_DETECT

For non-removable media, this function may remain as seen in RFA\CFSLOW.C, or it may just return ERR_NONE. For removable media, this function will provide a way to indicate when a device has been removed or inserted. An interrupt handler must be created that will call CFS_special with the SP_CARD_DETECT option which will handle high level functionality and call this low-level subcommand. The filing system will then clean up all internal structures which are applicable (open file structures), and the low-level function is responsible for re-initializing the DEVICE_INFO array. This should be done by calling FlashDevMount(). This function should remain the same as CARD\CFSLOW.C.

Possible Return Error Codes

ERR_READ
ERR_NONE

DEFAULT

This function will call the User_Special() function which will allow the user to add additional functionality to the software. This option should remain as is for removable or non-removable media.

Possible Return Error Codes

ERR_PARAM
ERR_NONE

APPENDIX D
LOW-LEVEL SUPPORT FUNCTIONS
(LOWLVL.C)

int FlashDevMount( )

To determine the presence of a device, and to initialize the internal FSD structures to use a device, the low-level driver must provide a Device Mount function. This function must call the FlashCompatCheck()(removable media only) function as well as filling in the DEVICE_INFO structures. If the FlashCompatCheck function fails, an ERR_JEDEC must be returned to the calling function. If a failure occurs detecting a device, the function returns one (1). Fields to be initialized in the DEVICE_INFO global array include the device, blocksize, numberblocks, status, lfs_size, lfs_offset, and lfs_end.

When using non-removable media, the format does not need to be kept on the flash media. This allows this function to use pre-determined defines to set up the DEVICE_INFO array structure. If a modifiable non-removable media is required, a User_Special function can be created which will allow the information to be passed in from the application level.

For removable media, this function should call the FindPartition() function which analyzes the PCMCIA structures on removable media to assist in initializing this information.

Possible Return Error Codes

ERR_READ
ERR_JEDEC
ERR_NONE

int Erase_Card (DEVICE_INFO * dev_ptr)

This function applies to removable media and will erase all blocks in the media on which the device resides. This function uses the device, blocksize, and numberblocks fields of the DEVICE_INFO structure passed in to determine what it should erase. This function would typically be used before creating the initial format of the media during manufacturing. If using removable media, this function will erase the tuples in the format to allow a new format to be created. If using removable media, use the CARD\CFSLOW.C version of this function. If using non-removable media, use the RFA\CFSLOW.C version of this function.

Possible Return Error Codes

ERR_ERASE
ERR_DEVICE
ERR_NONE

int EraseDevice (DEVICE_INFO *dev_ptr)

This function needs to erase the contents of the device (partition) specified by dev_ptr->device. This will allow the current files to be erased on this device, yet not affect information stored on any other device. If using removable media, use the CARD\CFSLOW.C version of this function. If using non-removable media, use the RFA\CFSLOW.C version of this function.

Possible Return Error Codes

ERR_ERASE
ERR_DEVICE
ERR_NONE

int CreatePartition (DEVICE_INFO *dev_ptr)

This command applies to removable media only. It erases the area which will be used to define the device (partition). Then it formats the first block in the media with the necessary PCMCIA CIS structures. It uses the device, blocksize, numberblocks, and aux (size of device) fields to create device information. If using removable media, use the CARD\CFSLOW.C version of this function.

Possible Return Error Codes

ERR_DEVICE
ERR_PARAM
ERR_READ
ERR_NONE

int FindPartition (BYTE array_ctr)

This function applies to removable media only; however, when using removable media, use the CARD\CFSLOW.C version of this function. This function assists in reading the PCMCIA CIS structures on removable media to determine what devices exist on the media. It fills in the lfs_size, lfs_offset, and lfs_end fields of the device array beginning at the array index passed in for the maximum number of devices per media.

Possible Return Error Codes

ERR_READ
ERR_NONE

int FlashCompatCheck (void)

This function needs to be created by each user to interface to specific hardware. This function should send the intelligent identifier command to each component of the entire flash media. It should then read the manufacturer's identifier for each component and verify that the value indicates the component is an Intel component. If any component in the media is not an Intel component, this function should return an ERR_JEDEC failure.

Possible Return Error Codes

ERR_JEDEC
ERR_NONE

int User_Special (DEVICE_INFO *dev_ptr, DWORD scmd)

This function is completely customizable by the user. Any low-level functions not supported by the existing file system should be created as an additional sub-command in the FlashDevSpecial() of the low-level function. When an unrecognized value is found in FlashDevSpecial(), this function will be called.

APPENDIX E
USE OF MEDIA STATUS TABLE FOR UNEXPECTED POWER-OFF

The signature exists at the end of the status block and will assist the filing system when it searches for the Media Status Table. This will be beneficial if an unexpected power-off occurs during cleanup of the Status Table.

```
struct {
        DWORD   FilePointer;        /* Points to file to be deleted */
        DWORD   Residue;            /* Length of residue */
        DWORD   FreeSpace;      /* Offset in device to copy files back to */
        DWORD   SpareIndex;         /* Offset to next fetched file in reclaimed block */
        BYTE    ResidueFlag;        /*  If block has residue from last block */
        BYTE    data_x_check;       /*  If data in Reclaim_status_info has been updated */
        BYTE    status;             /*  Status of device */
} MEDIA_STATUS_TABLE;
```

Listing 4. Media Status Table Structure

Field	Description
FilePointer	This pointer stores which file is currently being evaluated for deletion status.
Residue	This variable stores the length of residue which exists from a previous file in the spare block.
FreeSpace	This variable stores the offset in the device where the valid files can be copied back to.
SpareIndex	This variable stores the offset from the beginning of the device to the next fetched file in the reclaimed block.
ResidueFlag	This field indicates to the system whether the spare block contains residue from a file which existed in the last block. The actual amount of residue is stored in another parameter.
data_x_check	has not been updated. If the field is 0, all data has been updated and this is a valid RECLAIM_STATUS_INFO.
status	The status values and their meanings follow.

Condition Status Value	Definition	Reclaim Status
1	Moving Block Data to Spare Block	Reclaim will either start from the beginning of this block to move the data to the spare block or will use the valid available pointers to begin where it left off.
2	Erasing Original Block	The block which was being erased should be checked to determine if it is erased. If not, the erase sequence should occur and reclaim should continue from this point.
3	Copy File Residue	The spare block should be evaluated to copy the residue from a previous file to the free space.
4	Copying Valid information to Free Space	The last valid data to be copied should be verified. If this process did not complete, it should occur again. Otherwise, the next step in the reclaim may occur.
5	Erasing Spare Block	The block which was being erased should be checked to determine if it is erased. If not, the erase sequence should occur and reclaim should continue from this point.
6	Evaluate Next	This status indicates that the next file should be evaluated to determine if reclaim is necessary.
7	Reclaim Complete	No cleanup required.

Initialization of MST

The initialization procedure evaluates the media to determine if a Media Status Table exists. If the Media Status Table does not exist in the first block, the system searches the media for the Signature field of the Media Status Table. If this does not exist in the media, the initialization procedure erases the media and creates a Media Status Table in the first block. If the Media Status Table exists inside the media, it is assumed that it was undergoing cleanup and the initialization process should complete this cleanup before continuing. If the Media Status Table already exists in the first block, the initialization process evaluates the reclaim status information structures to determine if reclaim was interrupted at any time. If reclaim was interrupted, initialization completes reclaim. In addition, the initialization evaluates the last file in each device to make sure that the file was not open for write. The initialization process truncates and deletes any files which were left open for write at power-off.

Media Status Table Cleanup

After the reclaim process occurs several times, the Media Status Table will reach the end of the first block. At this time the Media Status Table will need to be reclaimed or cleaned up. The reclaim process must use the spare block of the device currently being reclaimed. The entire Media Status Table will be copied to the spare block, and all current information will be copied back to the first block of the media after the block is erased.

Reclaim Cleanup

The reclaim cleanup must first evaluate which device (if any) was being reclaimed. It must then continue the reclaim where it left off. The table above illustrates the action of the reclaim process when each status is encountered.

Filename:	292175_1.DOC
Directory:	C:\TESTDOCS\DOCS
Template:	C:\WINDOWS\WINWORD6\TEMPLATE\ZAN____1.DOT
Title: E	
Subject:	
Author:	Mary Ann Hooker
Keywords:	
Comments:	
Creation Date:	09/09/95 12:22 PM
Revision Number:	65
Last Saved On:	12/06/95 11:26 AM
Last Saved By:	Ward McQueen
Total Editing Time:	531 Minutes
Last Printed On:	12/06/95 11:30 AM
As of Last Complete Printing	
Number of Pages:	30
Number of Words:	9,904 (approx.)
Number of Characters:	56,454 (approx.)