Intel® Virtual Small Block File Manager (VFM) User's Guide

Version 2.3

November 2000

Order Number: 298132-002

Contents

Figures

Tables

Chapter 1
Introduction

Introduction 1

Welcome to the Intel® **V**irtual Small Block **F**ile **M**anager User's Guide, the Intel Corporation-developed flash media manager and filing system software solution! It will be referred to as **VFM** throughout the rest of this manual. VFM is comprised of two main functions:

- A flash media manager, the Intel® Virtual Small Block (VSB), handles all aspects of managing the flash memory. It provides the logic to create the smaller "sector-sized" virtual small blocks from the larger physical flash erase blocks. It handles the writing and management of the VSB units. It does the component level "clean-up" and a host of other lower-level, sector-oriented management. VSB can be used "stand-alone" to provide component-oriented "sector to flash" capability.
- A flash file manager, the Intel® VSB File Manager (VFM), is based on VSB but adds FILE capability, multiple component/partition handling, and a host of other higher-level file system capabilities.

VFM tracks all aspects of files, which are broken down into sectors or VSBs. The file manager provides the user robust file system functionality with the capability to have multiple files open for creation, read, write, and edit. More details on this are provided throughout the rest of this manual. However, before you continue, please refer to Appendix C, "Glossary," which lists common and Intel-specific terms/definitions.

1.1 Implementation Considerations

There are several features and requirements relevant to this implementation of VFM. They are as follows:

1.1.1 Features

— User-selectable sector (VSB) size.

— Allows multiple partitions.

— Allows multiple files open for read, write or edit per partition.

— Allows append, insert, replace and delete editing capability on a per byte basis.

— Uses Type and ID fields to identify and/or categorize files (allows for pseudo-subdirectory capability).

— Extensible through the Extended Header which can be used for ASCII file names or other file information.

— Sector allocation algorithm designed to handle component spanning which encourages wear-leveling.

— Robust power-loss recovery assures that any saved data on the flash is safe from corruption during power-loss, or system hangs.

— Robust cleanup algorithms for "dangling reference" and "garbage" data resulting from power-loss events.

- Common Flash Interface to increase device compatibility and reduce software development time. Supports all current Intel flash including Intel® StrataFlash™ components.
- Supports the Intel® Scalable Command Set (SCS) (for current technology components), as well as Intel® Basic Command Set (BCS) for legacy Intel component support.
- Exclusion area allows use of code and data in one device.
- Virtual Components allow user to treat multiple devices as one.Alternatively, a user may divide one device into several virtual components. Only one erase block must be reserved per virtual component.
- Automatic identification of Intel StrataFlash memory during initialization.

1.1.2 Requirements

- Requires one spare block per flash component or virtual component.
- Must be able to use a flat directory structure (non-hierarchical; no true sub-directory).
- No simultaneous code execution and data storage in the same chip (without code modifications).
- No multi-tasking or multi-threaded access (without code modifications)

1.2 File System Driver Architecture

The File System Driver (FSD) design is intended to provide code that is modular and can be easily integrated with your application and platform. The core FSD and Flash Media Manager remain the same, regardless of the operating system, processor hardware or flash storage media configuration (card versus RFA). To use the FSD, a low-level hardware driver that responds to the low-level interface definitions needs to be linked to the FSD and an Operating System Interface Driver. This document specifies the File System Driver (see Figure 1-1, “Application’s Request Reaching the Partition” on page 1-4).

The FSD is geared towards small and large embedded designs that use flash media as their primary storage. Because of its compact size, certain file access limitations are necessary. For example, VFM does not support directory tree functionality.

It is also important to note that the FSD is **not** re-entrant. Certain data and control structures are shared internally by multiple FSD functions. The host O/S must ensure that only one task or process calls the FSD functions at any given time.

1.2.1 Operating System Interface

To serve as a file system, the FSD needs an Operating System Interface. This Operating System Interface is different for each O/S. For example, programmers familiar with MS-DOS* will undoubtedly recognize this concept as the network redirector interface. The FSD architecture allows it to be interfaced with any O/S through a set of library functions. Each FSD library function performs some basic operation on the partition. By combining these basic functions, the complicated requests that an operating system requires of a file system can be assembled. This architecture will be described in more detail in Chapter 3.

One assumption has been made regarding a request of a file system from an application (O/S). When an application requests a file access, its most basic form consists of a partition indicator and a unique file identifier. In MS-DOS, these two elements would look like “a:\readme.txt,” where “a:\” is the partition indicator for that storage unit, and “readme.txt” corresponds to the unique file identifier. This pair could also look like “1301011989,” where “13” is the partition indicator and 01011989 is the identifier. In this case, it may be the date that the file was recorded which is used as the unique ID. The FSD has the capability to operate on multiple files. The FSD will allow multiple files open for reading and writing. The maximum number of files open per partition must be predefined and the number of partitions must be predefined.

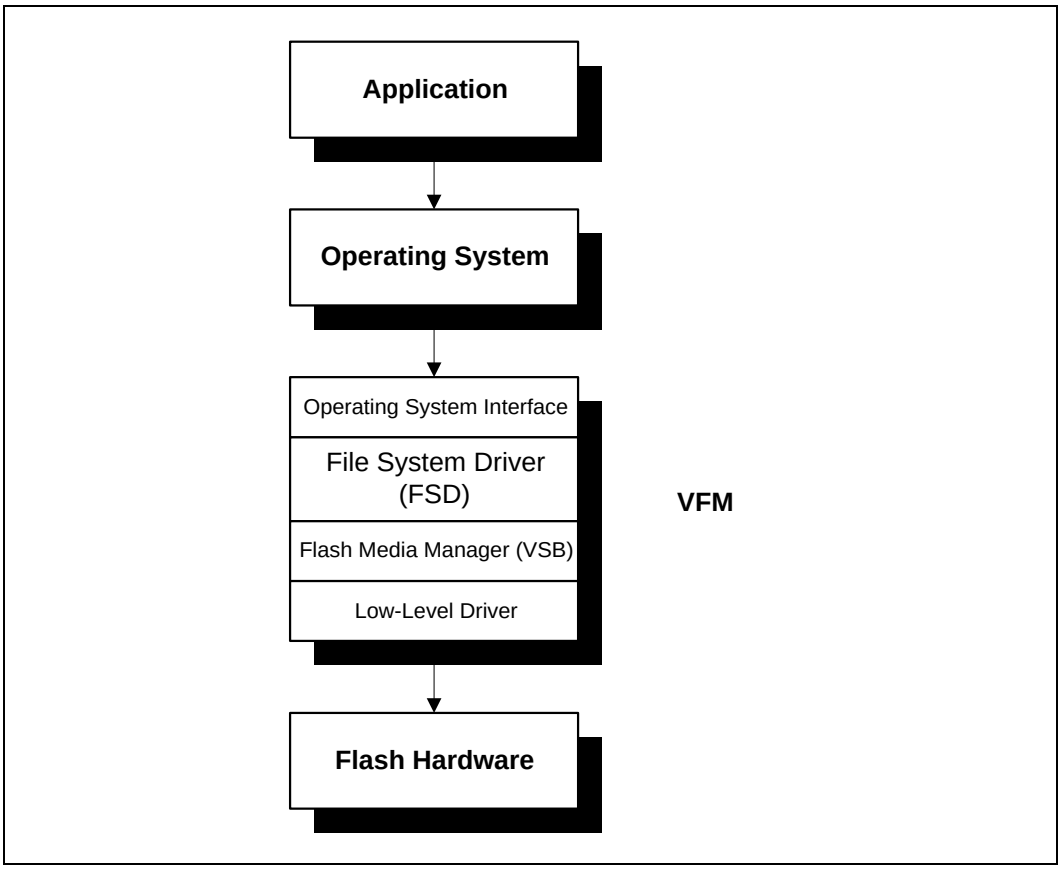
Once a method has been established to identify files within multiple storage partitions (our assumption stated above), the file system must be able to provide basic file operations: read, write, edit, delete, etc. In an attempt to create a generic FSD, only the most basic functions exist. For added flexibility,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

Although neither the Flash Media Manager nor the Low Level Driver have been discussed yet, the four components connected in the same box represent the complete file system. In this diagram, an application requests a file operation from the O/S. The O/S routes the file operation request to the file system. When the file system receives the request, it needs to locate the partition and the identifier for the file being accessed, as well as the function to perform.

Once the O/S request has been translated into FSD function calls by the operating system interface, the FSD utilizes the Flash Media Manager that uses the Low Level Driver to access the partition. The Low Level Driver provides the low-level media operations such as read, write and erase. The current version does not support removable media. However, this feature may be added by simply incorporating card detection and device mounting algorithms.

To better understand how the FSD operates, see Figure 1-1, “Application’s Request Reaching the Partition” on page 1-4.

Figure 1-1. Application's Request Reaching the Partition



1.2.2 File System Driver (FSD)

The File System Driver tracks all aspects of file information. This specific FSD tracks the different sectors of each file and tracks how the files fit together. The File System Driver provides functionality to open, close, read, write, edit, manage file access and perform cleanup.

1.2.3 Flash Media Manager

The Flash Media Manager tracks all aspects of flash blocks that are broken down into sectors or Virtual Small Blocks (VSBs). The Flash Media Manager handles the translation of the logical virtual small blocks to physical addresses and manages the flash media. This Flash Media Manager allows the user the capability to use flash as a disk drive. The VSB methodology divides each flash component into a number of "sectors" or virtual small blocks. Each erase block in a flash component contains a VSB Allocation Table (VAT) at the beginning of the block. The VAT is used to determine the **logical to physical** translation for each VSB within a block and the state of each VSB within that block.

1.2.4 Low Level Interface Driver

The Low Level Interface Driver does all media level I/O. When the FSD needs to access the media, it makes a call to one of the low-level function calls that are supplied by the Low Level Interface Driver.

1.2.4.1 Basic Functions

The basic functions associated with the Low Level Driver are: read, write, and erase. The FSD was designed to work with flash media; therefore, it needs an erase function to reclaim deallocated space.

1.2.4.2 Common Flash Interface

The Common Flash Memory Interface, or CFI, is a specification developed jointly by Intel, AMD, Sharp, and Fujitsu. This interface is part of an industry-wide effort to increase the interchangeability of current and future flash memory devices offered by different vendors. CFI allows one set of software drivers to identify and use a variety of different flash products, because all identifying information for the device, such as memory size, byte/word configuration, block configuration, necessary voltages, and timing information, is stored directly on the chip.

For components that are not CFI-enabled (legacy components), information about the flash device must be stored in tables within the system software. When a new non-CFI device is released, software generally must be modified to include a new table of information describing the device. With CFI, the system developer creates code that will run on today's flash and be ready to use in the next generation's cost-reduced versions. This provides the ability to use lower cost flash memory devices as they become available, without rewriting code.

The Intel® Scalable Command Set (SCS) was developed to provide CFI low-level functionality. In order to provide a seamless interface for the customer, the Low Level Drivers use Intel SCS commands to query the flash and assign the appropriate values to VFM global variables. Devices that do not support CFI simply ignore the query and do not provide the requested information. In this situation, the Low Level Drivers will look up device information in the JEDEC tables and use it to assign the global values, using the legacy method. With this procedure, CFI-compliant and non-compliant Intel® FlashFile™ component devices are supported by the VFM code. Furthermore, VFM has been optimized to use the advanced features, such as page buffers, provided on many Intel CFI-compliant devices. Based on the CFI query results, VFM will choose the appropriate low-level routine for that device.

1.3 Getting Started with VFM

The Intel Virtual Small Block File Manager (VFM) was designed to be as easy and straight forward to adapt to application as possible. It was also designed to be flexible and adaptable, which requires a considerable number of code options that need to be considered and set appropriately for successful implementation. This section attempts to provide a "roadmap" to this resource (and any others) to help make the adaptation as smooth as possible.

It is assumed, of course, that anyone attempting to adapt VFM is familiar with programming and programming techniques, is familiar with his/her own particular development environment (and flavor of 'C' compiler), and is familiar with the hardware platform into which VFM is being adapted. A thorough familiarity with the application's flash interface and memory map is required for a successful VFM adaptation.

Familiarity with this guide is also vital. Reading and understanding this manual from cover to cover is the best way to ensure a successful adaptation, but since this is the real world, the following section provides a brief overview.

1.3.1 Using This Manual

The purpose of this manual is to provide information on VFM that is as complete and comprehensive as possible, to assist you in the use of VFM at various stages–evaluation, integration/adaptation, test, and use. Not every portion of the guide is necessary at every stage of your development project. The following is the organization and usage of the various sections of this document:

Section	Title	Description	Stage Used
Chapter 1	Introduction	VFM Overview and Getting Started	Evaluation, Integration, Test, and Use
Chapter 2	VFM Media Control	How VFM does what it does	Evaluation, Test, and Use
Chapter 3	Filing System Interface Functions	APIs and features	Integration, Test, and Use
Chapter 4	Power-Loss Recovery Process in VFM	How VFM does power-loss recovery	Evaluation, Test, and Use
Chapter 5	Porting VFM to Your System	How to adapt VFM to your system, including which files do what and compile time options	Evaluation, Integration, and Use
Appendix A	FAQ	Frequently asked questions	Evaluation, Integration, Test, and Use
Appendix B	Documentation & Technical Support	Other useful related documentation & Contacting Technical Support	Evaluation, Integration, and Use
Appendix C	Definitions And Conventions	Glossary of terms and definitions	Evaluation, Integration, and Use

1.3.2 Tips for Successfully Porting VFM

The following tips, if followed, will help make your VFM experience a more pleasant one:

Know your hardware

A thorough understanding of the layout and operation of your system is crucial to a successful VFM adaptation. You must be very familiar with the operation of your board, including the operation of the flash interface: setting interface speeds, linear or windowed, data-path widths x8, x16, x32, single or interleaved components (single x8 or x16 access, paired x8 for x16 access, paired x16 for x32 access, etc.), voltage control, write protect, etc. You must know the particulars of your processor (big endian vs. little endian, any other special operation).

Know your development environment

A thorough understanding of your development environment is required to make the changes necessary to successfully compile VFM in your particular ‘C’ compiler, and to build within the development environment. VFM is generic ANSI ‘C,’ and is usually usable in just about any environment, but sometimes changes are required.

Read this User’s Guide

Familiarity with all of the aspects of VFM will make adapting it much easier. This guide contains a wealth of information on using and adapting VFM. We all hate to “RTFM” (Read the Full Manual), but if you do, adapting and using VFM will be much quicker and easier.

Understand and properly set the compile time options

VFM is very flexible in a wide variety of situations, however, this flexibility can sometimes hinder an adaptation because you forget, or don’t know, to set important options. Knowing what options exist, understanding how they affect how VFM works in your system, and setting them intelligently is vital. For example:

EVERY adaptation of VFM needs to set what appears to be the “memory card” settings in the DATATYPE.H file-

MC_ADDR = flash base address

MC_WIND_MASK = size of window to flash

Virtual components are complicated to understand initially, but give the software great flexibility. Set this option carefully.

Understand the layout and organization of the VFM code

The layout of the functions of VFM is somewhat complex due to the layered architecture of the software. This layering, and the ability to split off and use the VSB media manager separate from the file manager, causes a degree of added complexity to your adaptation. Functions are found scattered across different organizations within the code. Some defines and/or compile-time options have related and coordinated settings and values that must be set in different modules, in different source directories of the VFM source. It is easy to get settings out of sync, thus rendering the software inoperative in that particular incorrect configuration. If you understand the divergent nature of the source, you are less likely to make mistakes that will cause problems for your adaptation.

Follow these tips and your VFM experience will be a better one. Although the VFM code may be a little intimidating at first, with a little effort you will find it to be a powerful and adaptable Intel flash solution.

With the next chapters of this manual, you can start porting and using Intel® VFM!

Chapter 2
VFM Media Control

VFM Media Control 2

Of the four software layers that make up the VFM filing system, it is necessary to understand the in-depth structural details of the file system driver and the flash media manager. The following sections illustrate the structures required and their functionality.

2.1 File System Driver Media Control

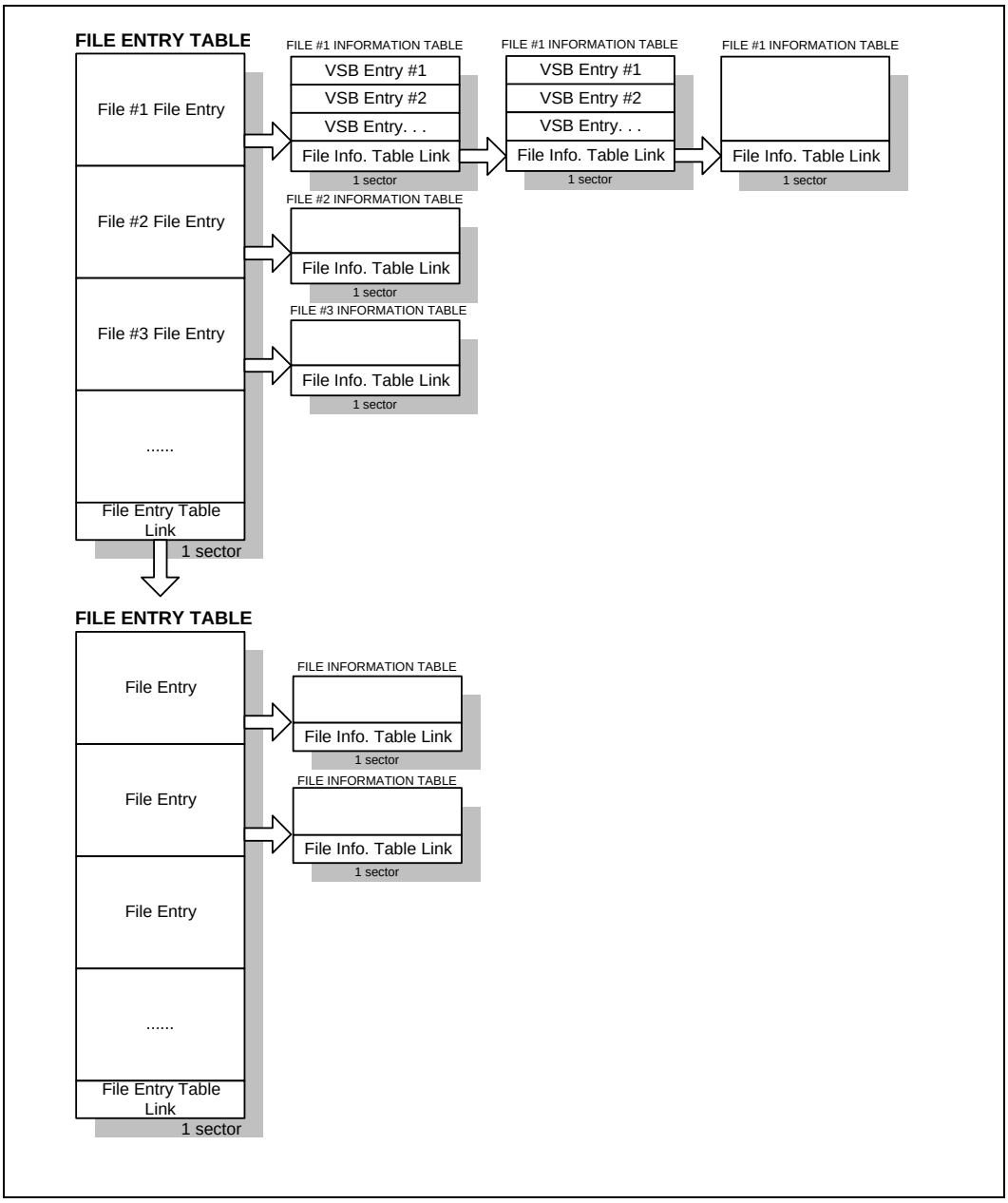
The File System Driver uses control structures for its data access operations, shown below:

- **File Entry** structure—contains file header information.
- **File Entry Table**—a table of file entry structures.
- **File Information Table** structure—contains sector information.
- **File Information Table**—a table of sector entries, which details how file pieces are put together.
- **File Link** structure—a File Entry Table Link contains information to link File Entry Table structures together; the File Information Table Link chains multiple File Information Tables for individual files.

The system uses the control structures as follows:

1. When accessing a file, the File Entry structure assists in determining the location of each file. The File Entry structure provides the location of the first File Information Table for each file. The total entries are calculated by subtracting the File Entry Link size from the sector size (VSB size). Divide the result by the File Entry structure size, to obtain the total file entries per sector. The File Entry Link chains multiple File Entry Tables together to comprise a root directory chain.
2. The File Information Table contains entries corresponding to VSB locations containing data, along with the location of the next chained File Information Table. The total number of VSB locations per File Information Table is calculated by subtracting the File Information Table Link size from the sector size (VSB size). Divide the result by the File Information Table entry size to obtain the total entries per sector. If additional file space is needed, the next piece of the file's information table can be found by following the File Information Table Link. Each sector representing a File Information Table contains a File Information Table Link, thereby comprising a single file chain.
3. The File Link will assist in parsing the File Entry Table and/or each file's File Information Table. Multiple File Entry Tables are chained together by a File Entry Table Link. Multiple File Information Tables are chained together by a File Information Table Link.

Figure 2-1. File Media Structure Interaction



2.1.1 File Entry Structure

There is a minimum of one VSB allocated for a File Entry Table. If the number of files extends beyond one VSB, the File Entry Link structure will point to the VSB containing its continuation. The first File Entry Table can be found in the first component of the partition and will be in logical VSB #1. Exclusion space does not require a VSB or any other file tracking structures. For more information about exclusion space, see the VFM Porting Guide.

Listing 2-1. File Entry Structure

```
struct file_entry
{
        DWORD           type;
        DWORD           id;
        WORD            attributes;
        DWORD           extended_header_size;
        BYTE            FIT_component;
        WORD            FIT_VSB;
}FILE_ENTRY;
```

2.1.1.1 File Entry Structure Fields

type

This field is a “stamp” indicating the type of the header. For our implementation, this field has been assigned zero (0). This field could be used in any way the application chooses. Since directories do not exist, it could be used to differentiate between file types.

id

This value is unique to each file object, within a given type. It is used along with the file type as file identification.

attributes

A bit mapped status field which indicates the file attributes.

Condition Status Value	Definition
11111111b	Erased. File has not been created yet
00000011b	Active File
00000001b	File Deletion Begun
00000000b	File Deleted

extended_header_size

The extended header offset field indicates if an extended header exists (field is zero when it doesn’t exist) and where it is located. The extended header structure must be predefined in a header file. The extended header should be a constant size.

FIT_component

Provides the device location of first File Information Table for the file.

FIT_VSB

Provides the VSB location within the device, which contains the first File Information Table for the file.

2.1.2 File Information Table Structure

The File Information Table structure will help determine how the pieces of a file fit together. This table is an array of the following structures. Each file will have a minimum of one VSB allocated for its File Information Table Structure. When a table terminator (0xFFFF) is reached, this indicates the end of the table for the file in question. A valid link field in the File Link Structure indicates that there's more file data in the file chain.

Listing 2-2. File Information Table Structure

```
struct file_info_table
{
        BYTE            component_number;
        WORD            VSB_number;
        WORD            num_valid_bytes;
}FILE_INFO_TABLE;
```

2.1.2.1 File Information Table Structure Fields

component_number

This field of the table indicates which component this sector of the file is in.

VSB_number

This field indicates in which VSB of the related component this sector of the file resides.

num_valid_bytes

This field indicates how many bytes in the VSB are part of the file.

2.1.3 File Link Structure

Listing 2-3. File Link Structure

```
struct file_link
{
        BYTE            next_component;
        WORD            next_VSB;
        BYTE            previous_component;
        WORD            previous_VSB;
}FILE_LINK;
```

2.1.3.1 File Link Structure Fields

next_component

This field contains the component number for locating the next piece of the File Information Table or the next piece of the File Entry Table.

next_VSB

This field contains the VSB number for locating the next piece of the File Information Table or the next piece of the File Entry Table.

previous_component

This field contains the component number for locating the previous piece of the File Information Table or the previous piece of the File Entry Table chain.

previous_VSB

This field contains the VSB number for locating the previous piece of the File Information Table or the previous piece of the File Entry Table chain.

2.1.4 Command Control Structure

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

Listing 2-4. Command Control Structure

```
typedef struct cmd_ctrl
{
    DWORD       partition;      /* Logical Partition */
    DWORD       buffer;         /* Buffer Address*/
    DWORD       count;          /* Transfer Count */
    DWORD       actual;         /* Transfer Actual */
    DWORD       scmd;           /* Subcommand */
    DWORD       type;           /* EntryType */
    DWORD       id;             /* UniqueID */
    DWORD       aux;            /* Aux Data*/
} CMD_CTRL;
```

2.1.4.1 Command Control Structure Fields

partition

Indicates logical partition with which the call should be associated. This field should be filled in on all function calls.

buffer

Pointer to buffer to read or write data. Used also in some special functions. Details follow in the function interfaces listed in following sections.

count

Indicates the number of bytes to read or write.

actual

Indicates number of bytes read or written.

scmd

The subcommand field can be used to call a function for a specific purpose that it may support, such as the “FSD_Special” command.

type

This field is a “stamp” indicating the type of the header. For our implementation, this field has been assigned zero (0). This field could be used in any way the application chooses. Since directories do not exist, it could be used to differentiate between file types.

id

The user-supplied unique id which identifies a unique file.

aux

This field is provided to allow a path to pass information from the file system to the low level software.

2.1.5 Partition Info Structure

The FSD utilizes the Partition Info structure, stored as a global RAM array, to determine information about the partition being accessed. Initializing the target partition fills the following structure with the appropriate data. An array of partition info structures exist in a header file and must be modified to fit the user’s setup.

Listing 2-5. Partition Info Structure

```
typedef struct partition_info
{
    DWORD       block_size;         /* Size of blocks */
    WORD        partition_offset;   /* Beginning of partition */
    WORD        partition_end;      /* End of partition */
} PARTITION_INFO[MAX_PARTITIONS];
```

2.1.5.1 Partition Info Structure Fields

block_size

This field indicates the block size in bytes, for this partition.

partition_offset

This field indicates the component number at which this partition begins (inclusive).

partition_end

This field indicates the end component of this partition (inclusive).

2.1.6 Open File Info Structure

Upon opening or creating a file, an Open File Information structure maintains the file pointer and other statistics concerning the file. This operates similar to the FILE structure in ANSI 'C.' The open files get tracked via the global RAM array of these structures. An array of FILE_INFO structures will exist per **partition** for open files.

Files may be opened in three different modes; "open mode" for read only, "create mode" for writing new files and "edit mode" for editing existing files. You may read file data in any of the three modes. Both the read or create modes support appending data. You may perform edit operations only while in edit mode.

Listing 2-6. Open File Info Structure

```
typedef struct file_info
{
    BYTE        FET_component;
    WORD        FET_VSB;
    WORD        FET_index;
    BYTE        FIT_component;
    WORD        FIT_VSB;
    WORD        file_mode;
    DWORD       file_offset;
    DWORD       size;
    DWORD       file_ptr;
    DWORD       type;
    DWORD       id;
} FILE_INFO[MAX_FILES_OPEN]
```

2.1.6.1 Open File Info Structure Fields

FET_component

The component containing a VSB location for a File Entry Table.

FET_VSB

The VSB containing the File Entry Table with an entry for the current file.

FET_index

This file's index position within the File Entry Table.

FIT_component

Component containing initial File Information Table structure for this file.

FIT_VSB

VSB in component above containing initial File Information Table structure for this file.

file_mode

Subcommand passed in on FSD_Open. This will allow tracking of what functions are legal to perform on the file.

file_offset

Indicates offset to file data (beyond extended header)

size

Indicates the current size of the file

file_ptr

Pointer into the file. This is useful for “seek and tell” when a file is open for reads, or when a file is open for editing.

type

Indicates file type for file which is open.

id

Indicates file id for file which is open.

2.1.7 Global MediaStatistics Structure

The media statistics is stored as a global RAM array to determine information about the media statistics. During Initialization, all the VAT Entries are being read and the following structure is filled with the appropriate data. An array of global mediastatistics structures exist which will be maintained at all times and can be read by the user via SP_SPACE function.

Listing 2-7. Global MediaStatistics Structure

```
typedef struct memory_status
{
  WORD          clean_sectors;      /* total available VSBs in comp */
  WORD          dirty_sectors;      /* total discarded VSBs in comp */
  WORD          valid_sectors;      /* total allocated VSBs in comp */
}MEMORY_STATUS;
```

2.2 Flash Media Manager Control

The VSB software layer of VFM organizes and divides each flash component into a number of “sectors” or virtual small blocks. It manages the logical to physical translation for these pieces, within their respective flash blocks, and manages the process which relocates these sectors during the cleanup process.

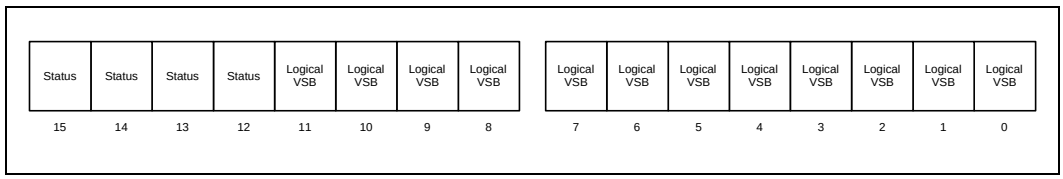
2.2.1 Virtual Small Block Allocation Table

Each block in a flash component, except one, contains a VSB Allocation Table (VAT) at the beginning of the block. A single block per component gets set aside for reclaiming dirty sectors. More discussion regarding reclamation will follow. The VAT determines the **logical to physical** translation for each VSB within a block and the state of each VSB within that block. There's a direct correlation between each VAT entry offset and the physical location of a corresponding VSB. Refer to the physical block allocation shown in Figure 2-3, "28F008 Mode with VSB" on page 2-11. The VAT entry states include: "free," "allocating," "allocated," "valid," and "discarded." The upper four bits in each VAT entry contain the status for a VSB. The remaining twelve bits represent the **logical** VSB number assigned to the corresponding VAT entry's VSB.

The logical VSB number must never exceed the 12-bit predefined length. The diagram below shows an example VAT entry; see Figure 2-2. You can determine the maximum number of VSBs per component by obtaining the size of the flash component and the VSB size. As an example, a 28F400 device with a 512-byte VSB size will have a maximum of 1024 VSBs. Subtract 256 VSBs corresponding to the spare block, to obtain 768. As for VAT overhead, in this example each block contains (256 VSBs * 2 VAT) entry bytes per VSB = 512 bytes which fits into one 512 byte VSB. Total number of VAT entries is 256 -1 = 255, as we don't need a VAT entry for the VAT itself. Keep in mind that a VAT can span multiple VSBs depending on VSB size and block size. Complete the calculation by subtracting one VSB for each VAT of the three remaining usable blocks, for a total of 765 maximum VSBs. The formula may be expressed as: 1024 -256 -3 = 765. Each component is treated separately and must have its own spare block. For this example, the largest possible VSB number does not exceed the 12-bit VAT entry length. Chapter 5, *Porting VFM to Your System*, discusses this topic in detail.

The following figure shows how each bit in a VAT entry is used to track logical VSB numbers and the corresponding status.

Figure 2-2. Bit Usage for VAT Entry



2.2.2 VAT State Transitions

The VSB attributes are used to determine the status of a physical VSB. A VSB will always be in one of five states—"free," "allocating," "allocated," "valid," or "discarded." Several constants are defined to enable checking the VSB status, these are VSB_VAT_FLAGS, VSB_FREE, VSB_ALLOCATING, VSB_ALLOCATED, VSB_VALID and VSB_DISCARDED. The four most significant bits of a VAT entry are used as flags to indicate intermediate states. The intermediate states are used to lower the probability of power interruptions corrupting the VAT data. VSB_VAT_FLAGS is used as a mask to remove the flags from the true VSB attributes. When a VSB is prepared for use, the status is set to VSB_ALLOCATING. Once the VSB data is written, the VSB number is written to the VAT and the status changes to VSB_ALLOCATED. At this point, a check is made to determine if a VSB exists with the same logical number (indicating the logical VSB is being overwritten). If a VSB exists with the same logical number, it is discarded and the status of the new VSB is changed to VSB_VALID. When a VSB is discarded the VAT status for the VSB changes to VSB_DISCARDED. This single bit status manipulation ensures a known state upon reset, and a robust power off recovery path. Table 2-1, "Bit Definition for VAT Entry Status" on page 2-10 summarizes the VSB status.

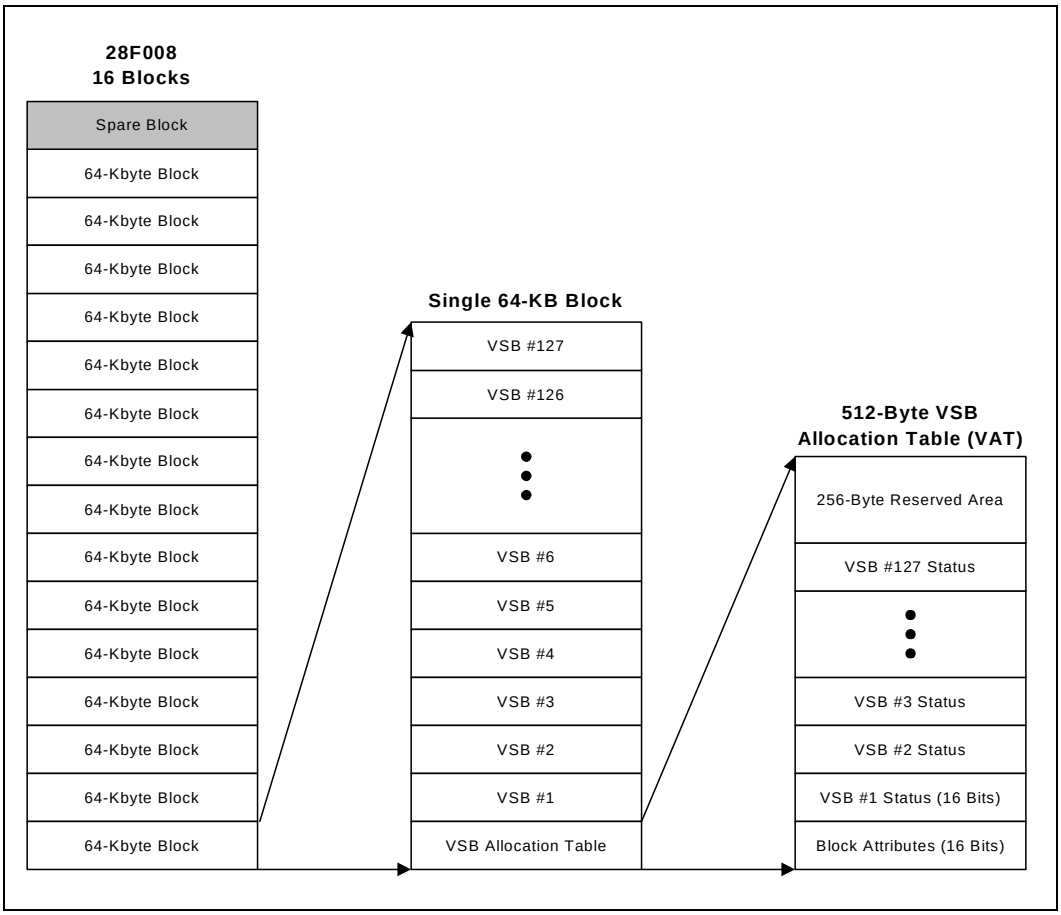
Table 2-1. Bit Definition for VAT Entry Status

Status in VAT entry (upper four bits)		Definition
1111	Binary	VSB_FREE—indicates the VSB has not been written to and all bits are in the erased state.
1011	Binary	VSB_ALLOCATING—indicates that the process of placing data into the corresponding physical VSB has begun.
1001	Binary	VSB_ALLOCATED—All data has been written to the VSB and a logical VSB number has been assigned in the VAT entry.
1000	Binary	VSB_VALID—Any previous VSB assigned the same logical number has been discarded and this is the currently valid data for the corresponding logical VSB number.
0000	Binary	VSB_DISCARDED—Data has been deleted or replaced.

2.2.3 Flash Memory Block Organization

An example of the physical organization of a single flash block within a flash component is shown in Figure 2-3. In this example, each VSB is 512 bytes long. The first 512 bytes of each block are reserved for the VAT and depending on the VSB size and block size, there may be some reserved space following the VAT. Each VSB attribute in the VAT is 16 bits long. The first 16 bits in any block are reserved for the block attribute. The block attributes are discussed in Section 2.2.2, “VAT State Transitions”. The first actual VSB will start at block address 512 and each subsequent VSB will start 512 bytes after the previous one.

Figure 2-3. 28F008 Mode with VSB



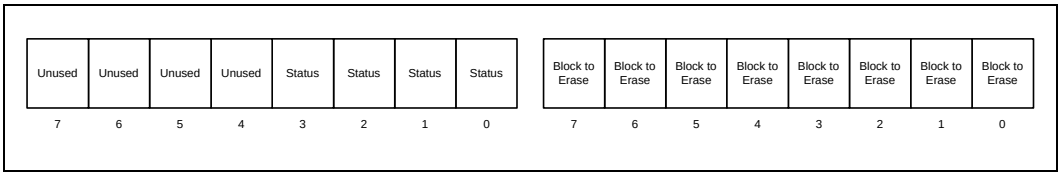
There are five states for a flash block. These states are: “erased,” “writing,” “full,” “recovering,” and “erasing.” Initially all blocks but one will be in the “erased” state. This one block will be in the “writing” state, which indicates that VSBs are being written to the block. When all VSBs within the block are “allocated” or “discarded,” the block will enter the “full” state and the next erased block will be updated to the “writing” state. This procedure will continue until one erased block (spare block) remains.

When reclamation takes place, the spare block will be placed in the “recovering” state while allocated VSBs from a “dirty” block are copied into the spare block. When all allocated VSBs are copied, the “dirty” block will be placed in the “erasing” state, which indicates that the copied block is being erased. When the erase completes, this new clean block becomes the new spare block. The previous spare block, which now contains valid data, transitions from the “erasing” state to the “writing” state, thus creating a new write block. The spare block and reserved sectors exist to allow reclamation of discarded VSBs. A VSB free count and threshold value are used to determine the frequency of automatic reclamation. The user definable threshold parameter gets set during initialization. The reclamation process will recover all discarded VSBs in a single flash block per “pass.” Automatic reclaim will select the “dirtiest” block and proceed to reclaim only that single block.

2.2.4 Flash Memory Block Selection

In order to write VSBs to flash memory, a block must be erased and must indicate it is ready to accept new VSBs. This selection is accomplished by using a block attribute to determine the block's state. The lower four bits of the first byte of each block of a flash component specify the block's state. The second byte is the logical block ID associated with that block. These bits are used as follows:

Figure 2-4. Bit Usage for Block Status and Block ID



Several constants are defined to indicate these states: VSB_FB_ERASED, VSB_FB_WRITE, VSB_FB_FULL, VSB_FB_ERASING, and VSB_FB_RECOVER. VSB_FB_ERASED indicates that the block is erased and available as a spare or free block. VSB_FB_WRITE indicates that the block is currently selected for writing VSBs. VSB_FB_FULL indicates that all VSBs in the block are either allocated or discarded.

Initially all blocks in a flash component are erased, hence all blocks will be in the VSB_FB_ERASED state. The initialization function will change the state of the second block to VSB_FB_WRITE. Subsequent VSB writes will be performed on this block until all VSBs in the block are exhausted. The first VSB write setup after that point will change the state of the block from VSB_FB_WRITE to VSB_FB_FULL, then change the state of the next available block to VSB_FB_WRITE. This process will continue until all free VSBs are consumed, at which point no more VSB writes may take place. Table 2-2 summarizes the block status.

Table 2-2. Bit Definitions for Block Status

Status in VAT entry (lower four bits of the first byte)	Definition
1111 1111 Binary	VSB_FB_ERASED—indicates the block has not been written to and all bits are in the erased state.
1111 1110 Binary	VSB_FB_RECOVER—indicates that the process of placing data into this block from a block being reclaimed has begun.
1111 1100 Binary	VSB_FB_ERASING—All data has been transferred from a block being reclaimed to this block and the block indicated is undergoing erase.
1111 1000 Binary	VSB_FB_WRITE—This block contains VSBs in the VSB_FREE state.
1111 0000 Binary	VSB_FB_FULL—This block contains no VSBs in the VSB_FREE state.

2.2.5 Global MediaStatistics

MediaStatistics are maintained in three global variables: freeVSBs, dirtyVSBs and validVSBs, which are available for quick polling via SP_SPACE. VFM memory is scanned by reading all the VAT entries only once in initialization during power on, and from then on is kept up-to-date at all times.

2.2.6 Reclaim Procedure

The reclamation process can be handled manually, automatically or both. The media control routine, called *ReclaimVSB*, recovers discarded sectors in a block. It identifies the “dirtiest” of all the blocks and proceeds to recover the discarded sectors contained in that single “dirtiest” block. The “special” interface called SP_RECLAIM initiates a manual reclaim. The special function repeatedly calls the media control routine for each block in all of the components, thus cleaning all dirty sectors in every block, for multiple components.

To establish automatic reclamation, the user must first set a value used to enable automatic reclamation. Three variables are used to determine when automatic reclamation should begin, *FreeVSB, ReclaimEnable* and *ReclaimThreshold*. The *ReclaimThreshold* represents this “watermark” value used to measure the point at which a proactive reclaim will occur. This is defined to five free VSBs as the default. The sole purpose of the *ReclaimThreshold* is to provide a mechanism to trigger automatic reclaim, thus providing a way to gauge an appropriate time to recover discarded blocks. One or more blocks may contain discarded sectors. However, automatic reclaim selects and cleans only the dirtiest block. Your application requirements will enable automatic reclamation, or perhaps you may want to control reclamation manually upon demand.

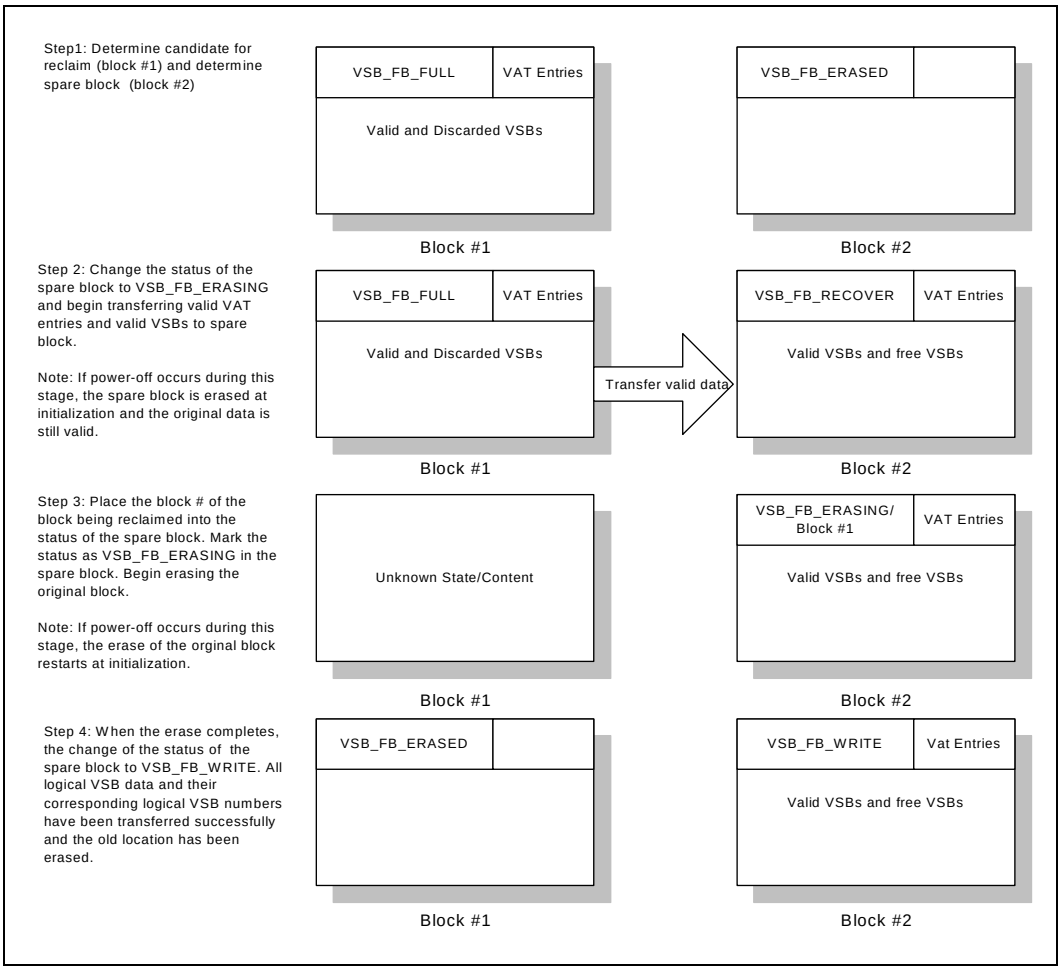
The user enables or disables the automatic reclamation at format and initialization time using the special routines, SP_FORMAT and SP_INIT. This value gets passed as input via the *count* field in the CMD_CTRL interface. The *InitializeVSB* function sets *FreeVSB* to equal the number of free VSBs, and sets *ReclaimEnable* to the user specified value. The *ReclaimThreshold* is defined to a value 5 to ensure proper free space management. The reserved sectors below the threshold will eventually be used. Upon reaching conditions where all other sectors are allocated and none are dirty, the pool of reserved VSBs will be allocated, falling below the *ReclaimThreshold*.

Setting the *count* field in the CMD_CTRL structure to zero will disable automatic reclamation, and setting it to a non-zero value will enable automatic reclamation.

Each time *WriteSector* is called, *FreeVSB* is decremented and is compared to *ReclaimThreshold*. If *ReclaimThreshold* and *FreeVSB* are equal, the reclamation function will be performed.

The *ReclaimVSB* function may be called at any time. If there are no discarded VSBs, no action will take place. Only one block will be reclaimed for each call to *ReclaimVSB*. Figure 2-5, “Step by Step Reclaim” on page 2-14 describes the block state transitions which occur during the *Reclaim* process.

Figure 2-5. Step by Step Reclaim



Chapter 3
Filing System Interface Functions

Filing System Interface Functions 3

3.1 Operating System Interface

The VFM software is created as generic, modular software. The test procedures utilize direct application access to the FSD layer. Customer applications can directly access the File System software functions or access them indirectly through the operating system.

3.1.1 Direct Application Access

The VFM FSD can be treated as a library of file system functions. Many embedded systems do not have an operating system, or the operating system is not flexible and does not allow additional “partitions” or “devices” to be installed. These systems should directly call the FSD functions. Using the direct call approach allows your software to have a simple interface to the library. The operating system interface layer of the VFM stack does not need to exist with direct access calls to the FSD.

3.1.2 Creation on an O/S Interface

Many operating systems allow multiple device drivers to be installed and provide a mechanism to accomplish this. These operating systems provide a way to perform file reads and writes through the system I/O. In this case, you must identify what functions must be supported by the Operating System Interface, create those functions, and use the functions as translators between the operating system and the VFM FSD interface. For example, at initialization time a partition or device gets installed into the system I/O tables. When reading from this partition, the system I/O transfers any necessary information to the driver’s read interface function. This function should translate the information passed to it so that VFM FSD can understand the information, then proceed to call the read function provided in the FSD. Any post processing would be done after this call completes and control returns to the system I/O interface, which transfers control back to the calling application.

3.1.3 Error Codes

The following list of error codes may possibly be returned from any interface function of the VSB/ VFM software layers.

Table 3-1. Error Codes (Sheet 1 of 2)

Return Error	Meaning	VFM Error Type
ERR_NONE	Indicates command was successful.	None
ERR_FLASH_INVALID, ERROR_JEDEC	Indicates that there is a problem with the flash or the flash hardware interface.	H/W
ERR_SYSTEM	Indicates a hardware problem was detected. There is a problem with the flash or flash hardware interface.	H/W
ERR_READ	Indicates an error reading the flash component.	H/W
ERR_WRITE	Indicates an error in the status register when writing to the flash component. Potentially due to a locked block if the flash component has this capability.	H/W
ERR_ERASE	Indicates an error erasing the flash block. Potentially due to a locked block if using flash with this capability.	H/W
ERR_SPACE	Indicates that no clean/free space is available for the function. If automatic reclaim is disabled, manual reclaim needs to occur before re-attempting the command. This error cannot occur if automatic reclaim is enabled.	Media dirty– Need reclaim
ERR_FORMAT_INVALID ,ERR_FORMAT	Indicates that initialization found that the flash media is corrupted–need to reformat.	Data corrupt– Need format
ERR_VSB_INVALID	Indicates that any flash access found that the requested VSB ID is invalid-media is corrupted– need to reformat.	Data corrupt– Need format
ERR_NOT_FOUND	Indicates that a VSB flash access found the flash media corrupted–need to reformat.	Data corrupt– Need format
ERR_NOTEXISTS	Indicates an error when attempting to open a file that does not exist.	User created error
ERR_VSB_OVERFLOW	Indicates that the current flash write request exceeds the VSB size. VFM users will not get this error unless something really bad has happened.	Data corrupt– Need format
ERR_CREATE	Returned by VSB to indicate an empty FET was discovered. Data is corrupted–need to reformat.	Data corrupt– Need format
ERR_MEMORY_FULL, ERR_DRV_FULL	Indicates that there's not enough flash memory media to perform the function.	User created error
ERR_PARTITION	Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.	User created error
ERR_PARAM	Indicates an incorrect parameter to a function.	User created error
ERR_SETUP	Indicates maximum VSB size exceeds the number of bits in the VAT Entry.	User created error
ERR_OPEN	Indicates **maximum** files open or an operation is attempted on an open file when the file should be closed.	User created error
ERR_EXISTS	Indicates the attempt to create a file or directory that already exists.	User created error
ERR_NOTCLOSED	Indicates an error in performing an operation on an open file that must be closed.	User created error

Table 3-1. Error Codes (Sheet 2 of 2)

Return Error	Meaning	VFM Error Type
ERR_NOTOPEN	Indicates an error in performing an operation on a file that is not open but must be to complete the operation.	User created error
ERR_FILEMODE	Indicates an error in performing an operation on a file that is open but in a different mode.	User created error
ERR_DETECT	Indicates an error in detecting presence of the flash device.	H/W
ERR_CONNECT	Indicates a connectivity error.	H/W
ERR_MEDIA_TYPE	Indicates an SRAM device is being used without the appropriate compile-time option SRAM_MEDIA.	User created error
ERR_MAX_OPEN	Indicates the number of files open exceeds the MAX_FILES define.	User created error
ERR_WRITE_ UNSUPPORTED	Indicates a ROM device is in use.	User created error

The Test-Script Interpreter is a PC-based software environment that simulates application interaction with VFM. The TSI code is available for use during the development process. See the *TSI User's Guide* for more information on this testing environment.

3.2 File System Driver (FSD) Interface

The following functions are available with VFM. Table 3-2, "Filing System Interface Function" on page 3-4 outlines which fields in the command control structure are valid for each call. The right-hand section of the table shows the valid modes for the files on which you are operating.

Table 3-2. Filing System Interface Function

	Parti-tion	Buffer	Count	Actual	Scmd	Type	ID	Aux	OPEN Mode	CREATE Mode	EDIT Mode	Closed
Open: any mode	I	i	—	—	I	I	I	—	—	—	—	√
Close	I	—	—	—	—	I	I	—	√	√	√	—
Read	I	I	I	O	—	I	I	—	√	√	√	—
Write	I	I	I	O	—	I	I	—	—	√	√	—
Edit												
–Insert	I	I	I	O	I	I	I	—	—	—	√	—
–Replace	I	I	I	O	I	I	I	—	—	—	√	—
–Delete	I	—	I	O	I	I	I	—	—	—	√	—
–Append	I	I	I	O	I	I	I	—	—	—	√	—
Find												
–First	I	i	—	—	—	O	O	I	√	√	√	√
–Next	I	i	—	—	—	O	O	I	√	√	√	√
–Matched	I	i	—	—	—	I	I	I	√	√	√	√
Delete	I	—	—	—	—	I	I	—	—	—	—	√
Special												
–Init	I	—	I	—	I	—	—	—	—	—	—	√
–Format	I	—	I	—	I	—	—	—	—	—	—	√
–Reclaim	I	—	I	—	I	—	—	—	√	√	√	√
–Seek	I	I	—	—	I	I	I	—	√	√	√	—
–Tell	I	O	—	—	I	I	I	O	√	√	√	—
–Space	I	—	—	—	I	—	—	i	√	√	√	√
–FETCleanup	I	—	—	—	I	—	—	—	—	—	—	√
–FITCleanup	I	—	—	—	I	—	—	—	—	—	—	√

NOTES:
I = Input
i= Input Optional
O = Output Optional

3.2.1 File Open

FSD_Open opens a file for reading, editing, or creates a file for writing. The user specifies a *partition*, *type*, unique *id* and a subcommand via *scmd*. The subcommand indicates open-for-read, open-for-edit, or open-for-create. If open-for-read (OPEN_OPEN) or open-for-edit (OPEN_EDIT), the FSD tries to locate the file in the partition and will return an error if it's not found. If open-for-create (OPEN_CREATE), the FSD will return an error if the file exists.

The *type* and *id* fields in the CMD_CTRL structure are the primary FSD interface used to identify and retrieve an existing file or to create a new file. The *buffer* field in the CMD_CTRL structure may be used as an optional secondary interface. It may be used as a pointer to retrieve file header information for existing files and/or it may be used as a pointer to pass additional file extension information, during file creation. Setting the *buffer* field to zero will disable this secondary CMD_CTRL interface.

If the *buffer* field on the input is non-zero and the file is being opened for read or edit, the buffer pointer returns the FULL_HEADER. The FULL_HEADER contains the FILE_ENTRY structure and may contain the optionally compiled EXTENDED_HEADER structure. If you want to retrieve the FULL_HEADER, then you need to set up the *buffer* field to a valid, initialized pointer. Otherwise, simply set *buffer* to zero and the search routine will skip retrieval of the FULL_HEADER.

When opening a file for create, the *buffer* interface may be used to pass file extension data to store with the file. The EXTENDED_HEADER compile-time option must be selected and the application must initialize the *buffer* interface to point to the extended header data. When removing the EXTENDED_HEADER option, the corresponding *buffer* code gets removed from the file creation routine, eliminating the usage of the *buffer* field.

Call Format:

The 'C' call to the open procedure is as follows:

WORD FSD_Open(CMD_CTRL *ptr)

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

Subcommand:

OPEN_OPEN	Open for read only
OPEN_CREATE	Open for create/append or read
OPEN_EDIT	Open for read/edit

Possible Return Error Codes:

ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.
ERR_OPEN—Indicates MAXIMUM files open or operation on an open file when it should be closed.
ERR_EXISTS—Indicates the attempt to create a file or directory that already exists.
ERR_NOT_FOUND—Specified VSB was not found in component.
ERR_PARAM—Indicates an incorrect parameter to a function.
ERR_SPACE—Reclamation needs to be done.

ERR_READ—Indicates an error reading the flash component.
ERR_WRITE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_NONE—Function completed successfully.
ERR_VSB_INVALID—An invalid VSB number was specified on function entry.
ERR_MEMORY_FULL—Free VSB was not found in component.
ERR_VSB_OVERFLOW—Indicates that the current flash write request exceeds the VSB size.

3.2.2 File Close

If the file indicated by *partition*, *type*, and *id* was open, the FSD performs any cleanup necessary to close the file. This includes any necessary changes to the media structures. The file information in the partition's open file array structure will be set to a predetermined value. If a failure occurs, the *status* field returned by the function will contain a descriptive error code.

Call Format:

The 'C' call to the close procedure is as follows:

WORD FSD_Close(CMD_CTRL * ptr)

Possible Return Error Codes:

ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.
ERR_PARAM—Indicates an incorrect parameter to a function.
ERR_NONE—Function completed successfully.

3.2.3 File Read

This function reads *count* number of bytes from the file specified by *partition*, *type*, and *id* and places them into *buffer.* The file can be read in any of the open file modes. To open a file in read-only mode, the *aux* field must be set to OPEN_OPEN. To read or append data to a new file, the *aux* field must be set to OPEN_CREATE. To read, append or edit data in an existing file, the *aux* field must be set to OPEN_EDIT. The file read begins at the location of the last read or whatever file offset the special Seek option has set. If during the process there is a failure, the *actual* field indicates how many bytes were transferred and the status field returned by the function will contain a descriptive error code.

Call Format:

The 'C' call to the read procedure is as follows:

WORD FSD_Read(CMD_CTRL *)

Possible Return Error Codes:

ERR_OPEN—Indicates MAXIMUM files open or operation on an open file when it should be closed.
ERR_NOT_FOUND—Specified VSB was not found in component.
ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.

ERR_READ—Indicates an error reading the flash component.
ERR_NONE—Function completed successfully.
ERR_PARAM—Indicates an incorrect parameter to a function.
ERR_VSB_INVALID—An invalid VSB number was specified on function entry.

3.2.4 File Write

This function writes *count* number of bytes from *buffer* to the file specified by *partition*, *type*, and *id*. The file must have previously been opened with the OPEN_CREATE or OPEN_EDIT option to append. If during the process there is a failure, the *actual* field indicates how many bytes were transferred and the status field returned by the function will contain a descriptive error code. The file write begins at the location of the last write, which always represents the end of the file. When appending data, the *file_ptr* field of the global Open File Info structure automatically gets set to the end of the file. Attempts to reset the file pointer to the end of the file via Seek will result in an error. **Do not** perform Seek prior to Append.

Worst Case Power Loss:

Append: Appended sectors are written to the media, up to the point of the power failure. Depending upon the failure point, some of the written sectors, those written last, may be treated as garbage. The media manager controls the recovery and state transitioning for the current sector being written; refer to Section 2.2.2, “VAT State Transitions” on page 2-9, and Section 3.3.1, “Initialize VSB” on page 3-17. The file may report only a portion of the appended data, representative of only those sectors which could be written and whose File Information Tables were updated properly, prior to the power loss.

Continue writing where the append left off, prior to the power failure.

Call Format:

The ‘C’ call to the write procedure is as follows:

WORD FSD_Write(CMD_CTRL * ptr)

Possible Return Error Codes:

ERR_NOTOPEN—Indicates an operation on a file that is not open but must be to complete the operation.
ERR_WRITE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.
ERR_SPACE—Reclamation needs to be done.
ERR_FILEMODE—Indicates an error from operating on a file that is open but in a different mode.
ERR_NONE—Function completed successfully.
ERR_VSB_OVERFLOW—Indicates that the current flash write request exceeds the VSB size.
ERR_VSB_INVALID—An invalid VSB number was specified on function entry.
ERR_MEMORY_FULL—Free VSB was not found in component.
ERR_NOT_FOUND—Specified VSB was not found in component.
ERR_READ—Indicates an error reading the flash component.
ERR_PARAM—Indicates an incorrect parameter to a function.

3.2.5 File Edit

This function writes *count* number of bytes from *buffer* to the file specified by *partition*, *type*, and *id*. The manner in which the information is written depends on the *scmd* passed in. The file must have previously been opened with the OPEN_EDIT option if a previously existing file is to be modified using append, insert, replace, or delete bytes. The offset address to begin performing the edit operation is determined by the *file_ptr* field of the global Open File Info structure. When appending data, the *file_ptr* field automatically gets set to the end of the file. Attempts to reset the file pointer to the end of the file via Seek will result in an error. **Do not** perform Seek prior to Append. However, in the case of Insert, Replace or Delete, the *file_ptr* value MUST be set up prior to performing the edit operation. First, perform a Seek operation to set up the offset within the file to begin the edit operation. If during the Edit process there is a failure, the *actual* field indicates how many bytes were transferred and the status field returned by the function will contain a descriptive error code.

Worst Case Power Loss:

Append: Appended sectors are written to the media, up to the point of the power failure. Depending upon the failure point, some of the written sectors, those written last, may be treated as garbage. Garbage may be created due to the File Information Table not properly being updated, prior to the power loss. The media manager controls the recovery and state transitioning for the current sector being written; refer to Section 2.2.2, “VAT State Transitions” on page 2-9, and Section 3.3.1, “Initialize VSB” on page 3-17. The file may report only a portion of the appended data, representative of only those sectors which could be written and whose File Information Tables were updated properly, prior to the power loss.

Continue writing where the append left off, prior to the power failure.

Insert: Insert builds the inserted data chain, then as a final step, inserts the new chain into the existing file chain. If a power loss should occur during this process, the new data chain does not impact the original data chain. The original chain remains intact and the new data chain becomes garbage data.

At power on, initialization will recover those garbage sectors that are not hooked into the file chain and the original chain remains as before. Rerun the Insert a second time.

Replace: Replace operates on each sector sequentially. Once each sector has been replaced, it is marked complete before operation on the next sector begins. In a power recovery situation, therefore, replace can positively determine whether whole sectors have been replaced, but not the sector in process at the time of power loss. Refer to the media manager control for individual sector recovery. The file reflects only the new data for those sectors completely rewritten prior to the power loss.

Rerun Replace for a complete or partial update.

Byte Delete: Delete may discard only a portion of the request, up to the point of the power failure. The file still contains the portion of the data which was not deleted. The file may potentially be corrupted if power fails prior to updating modified file information entries. This routine may be easily enhanced to provide graceful recovery for this case. However, this type of change will increase overhead, due to rewriting the File Information Table upon modifying each entry/sector in the file.

Determine the appropriate offset and complete the Delete for those remaining bytes.

Call Format:

The ‘C’ call to the edit procedure is as follows:

WORD FSD_Edit(CMD_CTRL * ptr)

Subcommands:

FILE_APPEND
FILE_INSERT
FILE_REPLACE
FILE_DELETE

Possible Return Error Codes:

ERR_NOTOPEN—Indicates an operation on a file that is not open but must be to complete the operation.
ERR_WRITE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.
ERR_SPACE—Reclamation needs to be done.
ERR_FILEMODE—Indicates an error from operating on a file that is open but in a different mode.
ERR_NONE—Function completed successfully.
ERR_PARAM—Indicates an incorrect parameter to a function.
ERR_NOT_FOUND—Specified VSB was not found in component.
ERR_READ—Indicates an error reading the flash component.
ERR_VSB_INVALID—An invalid VSB number was specified on function entry.
ERR_MEMORY_FULL—Free VSB was not found in component.
ERR_VSB_OVERFLOW—Indicates that the current flash write request exceeds the VSB size.

3.2.6 File Find

Find will locate the first, next or a matching file in the partition, specified via the *aux* field. To locate a file using the FIND_MATCHED search, the *type* and *id* fields are provided as input values in order to recover the target file. Performing FIND_FIRST or FIND_NEXT searches will locate the corresponding file, retrieve the file’s *type* and *id* and return them to the caller. The CMD_CTRL *type* and *id* are the primary interface used to recover a file. As with FSD_Open, if *buffer* is non-zero, this secondary interface will be filled with information from the FULL_HEADER. The FULL_HEADER consists of any file system specific headers and the EXTENDED_HEADER, if one exists.

Call Format:

The ‘C’ call to the find procedure is as follows:

WORD FSD_Find(CMD_CTRL *)

Auxiliary:

FIND_FIRST
FIND_NEXT
FIND_MATCHED

Possible Return Error Codes:

ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.
ERR_READ—Indicates an error reading the flash component.
ERR_NONE—Function completed successfully.
ERR_NOT_FOUND—Specified file was not found in component.

3.2.7 File Delete

Calling this function with a *partition*, *type* and a unique *id* of a closed file will update the *attribute* field in the FILE_ENTRY structure, within the File Entry Table. The *attribute* field gets updated to indicate the file was deleted. The space cannot be reused until performing a reclaim operation.

Worst Case Power Loss:

File Delete: This request begins by marking the file entry as FILE_DISCARDING and scans to the end of the File Information Table chain, for the target file. It discards each sector corresponding to a FIT entry, then discards the related FIT. Working backwards in the file chain, it deletes each data sector and control structure for the file, then marks the file entry as FILE_DISCARDED. If there's a power loss, a dangling reference may be created within the file chain, which may be removed by cleanup.

The FETCleanup and FITCleanup remove all files which have a FILE_DISCARDING attribute in their file entry.

Call Format:

The 'C' call to the delete procedure is as follows:

WORD FSD_Delete(CMD_CTRL *)

Note: If extended headers are enabled, and you set or leave the "buffer Ptr" entry passed in the cmd_ctrl structure to a non-null value, delete will return the extended header data of the deleted file in the buffer pointed to by the "buffer ptr." Set "buffer ptr" to null if you do not wish the extended header data returned to you.

Possible Return Error Codes:

ERR_NOT_FOUND—Specified file was not found.
ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.
ERR_WRITE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_READ—Indicates an error reading the flash component.
ERR_NOTCLOSED—Indicates an error attempting an operation on an open file that must be closed.

ERR_NONE—Function completed successfully.
ERR_SPACE—Reclamation needs to be done.
ERR_PARAM—Indicates an incorrect parameter to a function.
ERR_FILEMODE—Indicates an error from operating on a file that is open but in a different mode.
ERR_VSB_INVALID—An invalid VSB number was specified on function entry.
ERR_MEMORY_FULL—Free VSB was not found in component.

3.2.8 Special File Calls

"Special" provides a mechanism for File Media Manager (FMM)-specific operations. All file system-specific operations will call routines within the file system level. All FMM specific operations will call the Special function. Some operations may do both. In addition, the Special function provides a way for the host O/S to pass or receive user specific information through the FSD to the FMM functions. The *scmd* selects which Special function will be called. The first 20 functions have been reserved for internal development. OEMs may use any special subcommand greater than 20. This procedure will call FSD_Special with the respective subcommand. The predefined functions are described in the low-level function section.

Note: Before any other FSD function is called, a call to FSD_Special with subcommand SP_INIT must be made. This call initializes the PARTITION_INFO structure which is necessary for all operations, as well as performing power off recovery for reclaim procedures, files left open for write, and miscellaneous other error situations.

Call Format:

The 'C' call to the special procedure is as follows:

WORD FSD_Special(CMD_CTRL *)

3.2.8.1 Initialization

The SP_INIT subcommand performs power off recovery and initializes all the filing system control structures. The SP_INIT subcommand calls the media manager initialization routine *InitializeVSB*, for each component of the media. The user submits the *count* field as input to the initialization routine, which sets up the automatic reclaim indicator, known as the *ReclaimEnable*. Refer to Section 2.2.6, "Reclaim Procedure" on page 2-13 for details about the *ReclaimEnable*. The *count* indicates if the automatic reclaim is enabled or disabled. Using a value of zero in this field will disable automatic reclaim and a non-zero value would enable it.

Then SP_INIT does file table cleanup, which checks for "dangling references." A "bit table" is built, which gets stored as a global in RAM. This is accomplished by scanning the File Entry Table chain and each file's corresponding File Information Table chain, marking corresponding bit of the array element in the table as "1" for every valid sector. This bit table identifies all the valid VSBs used by the existing FSD and those which are available. This is copied to a local bit table array. Next, system cleanup executes which validates each VSB against the File Entry Table chain to identify garbage sectors by zeroing out the "bit table" entries. Any sector known by the media manager but not valid in the chain gets discarded as garbage data. Garbage may be caused by a power failure while performing an insert or a FET Cleanup. Then it checks if all the entries in the "bit table" are zeroed out. This is a way to test proper power off recovery. The originally built "bit table" entries are again copied back from a local "bit table" array. Finally, all the global RAM structures used by the filing system are initialized. Care must be taken to avoid repeated calls to SP_INIT from the application or O/S interface. Multiple calls to SP_INIT can create undesired

results, such as trashing existing globals. When implementing multiple tasks via mutually exclusive semaphores, consider tasks which may inadvertently recall SP_INIT. Evaluate the task synchronization, how and what functions are being blocked and code isolation. A single call to SP_INIT should occur, whether using hardware semaphores, block/wakeup or counting semaphores.

SP_INIT provides power off recovery for an existing system with VFM data. Each call to *InitializeVSB* verifies the state of each block in the component. It verifies the validity of the media manager control structures by identifying a "Write Block," a block of flash defined by BLOCK_SIZE, and checks if there are any unknown block states. SP_INIT makes two assumptions. First, that the media was either received by the factory in an erased condition or that it has been previously formatted to a "known" valid state via the SP_FORMAT function. Second, that there may have been existing VFM data and filing system control structures on the media, prior to the power-failure. The SP_INIT routine does not replace the SP_FORMAT routine. The purpose of SP_FORMAT is to erase the media, then initialize the media manager control structures, thus establishing a clean VFM formatted media. The distinction between these two important operations is that SP_FORMAT "establishes" a new VFM media and SP_INIT "verifies/recovers" an existing VFM media.

Possible Return Error Codes:

ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.
ERR_PARAM—Indicates an incorrect parameter to a function.
ERR_NOT_FOUND—Specified VSB was not found in component.
ERR_READ—Indicates an error reading the flash component.
ERR_WRITE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_NONE—Function completed successfully.

3.2.8.2 Format

The SP_FORMAT subcommand will allow the user to format the media and establish a VFM format. The format size gets bounded by the compiler defines, which establish the partition dimensions. This function first erases the partitions, then it initializes the media manager control structures. Next, it initializes the filing system global RAM structures. Like the SP_INIT routine, a *count* parameter gets passed into the format routine. The user-defined *count* field enables or disables the automatic reclamation. Refer to prior Section 2.2.6, "Reclaim Procedure" on page 2-13, for details about the *ReclaimEnable*.

Possible Return Error Codes:

ERR_PARAM—Indicates an incorrect parameter to a function.
ERR_READ—Indicates an error reading the flash component.
ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.
ERR_ERASE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_WRITE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_NONE—Function completed successfully.

3.2.8.3 Reclaim

The SP_RECLAIM subcommand performs a manual reclaim. It will reclaim a specified number of blocks per component of a partition. This user-defined value gets passed via the *count* field. The component reclaim will continue until *count* number of blocks are reclaimed or until the entire component has no blocks left to reclaim, whichever comes first. Reclaim recovers discarded sectors for some number of dirty blocks. To guarantee recovery of all discarded sectors for all dirty blocks in a component, set the *count* to the total blocks in a component. Then the reclaim algorithm will execute for every dirty block in each component until there are no dirty blocks left.

Possible Return Error Codes:

ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.
ERR_READ—Indicates an error reading the flash component.
ERR_WRITE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_ERASE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_NONE—Function completed successfully.

3.2.8.4 Seek

The SP_SEEK subcommand uses the *partition*, *type*, and *id* to determine if a file is open. If the file is open, Seek verifies that the file offset request does not exceed the file size. Then the user-defined input field called *buffer* sets the new offset into the file. It accomplishes this by updating the *file_ptr* field of the global Open File Info structure. The purpose of the Seek operation is to set the file pointer to a valid offset within an existing file. The Seek operation must be called to establish the file pointer, prior to performing any Edit operations which insert, replace or delete bytes in a file. File writes which append data to the bottom of the file DO NOT require seeking to the end of the file. The write algorithm automatically sets the file pointer to the end of the file. The file write routine obtains the file size from the *file_size* field in the global Open File Info structure.

Note: The *file_ptr* is a zero-based file pointer. The *file_size* field is absolute. When the file pointer gets loaded with the file size, it points at the next position to begin writing the append. Likewise, each append leaves the zero-based file pointer at the next position, ready to begin another write. The VFM architecture does not carry any additional end-of-file markers in the file. Instead, it calculates the file size using the FIT's valid bytes. It maintains the *file_size* value in a global while the file remains open.

Possible Return Error Codes:

ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.
ERR_PARAM—Indicates an incorrect parameter to a function.
ERR_NOTOPEN—Indicates an operation on a file that is not open but must be to complete the operation.
ERR_NONE—Function completed successfully.

3.2.8.5 Tell

The SP_TELL subcommand uses the *partition*, *type*, and *id* to determine if a file is open. If the file is open, the current offset into the file will be placed into the *buffer* field. The file size will be placed into the *aux* field. The *buffer* field gets set to the zero-based *file_ptr* field of the global Open File Info structure.

Possible Return Error Codes:

ERR_PARTITION—Indicates a fatal error has occurred to the partition, or that the media may be unformatted, or it may be hardware write protected.
ERR_NOTOPEN—Indicates an operation on a file that is not open but must be to complete the operation.
ERR_NONE—Function completed successfully.

3.2.8.6 Get Memory_Status

The SP_SPACE subcommand will store the status of the VSBs in the partition in the variables whose address is sent by the caller. If memory is marked as dirty, this memory will not be available until a reclaim has been performed. The user must set up and pass a pointer to a MEMORY_STATUS structure, via the *aux* parameter. This structure is filled in by the routine.

This operation provides a way to monitor the VSB usage of the entire media. These values represent VSB usage for filing system control structures and application data. The total clean sectors remaining do not include those sectors held in reserve (ReclaimThreshold). When enabling the automatic reclamation feature, eventually all of the dirty sectors will be recovered for use, then the reserved sectors will be used.

Listing 3-1. Get Memory Status Structure

```
typedef struct
{   WORD        clean_sectors;
    WORD        dirty_sectors;
    WORD        valid_sectors;
}MEMORY_STATUS;
```

Possible Return Error Codes:

None.

3.2.8.7 FETCleanup

The SP_FETCleanup subcommand processes a collapse of the File Entry Table, removing any file entries whose attributes show the file is deleted FILE_DISCARDED. This routine has a built-in algorithm which decides whether to cleanup or cleanup currently not needed. It also identifies file entries marked as FILE_DISCARDING. This flag indicates that a “delete-in-process” never completed, possibly due to a power failure. Upon encountering this flag, it proceeds to scan to the

end of the File Information Table chain, for this file. Upon reaching the expected table terminator value, or recognizing a dangling reference pointer in the link structure, it deletes the remainder of the file working backwards in the chain.

It performs a File Entry Table scan and reports the total number of file entries and deleted files. The algorithm determines whether there are enough deleted files to justify a collapse of the entire chain. It also determines the total number of sectors required to rebuild the chain against the total number available. Removing deleted entries compresses one or more File Entry Tables in the chain. Although there may be deleted entries, there may not be enough fragmentation which would trigger a chain collapse. A collapse reduces the overall total number of File Entry Tables in the chain.

The top of the chain gets written out last, to provide recovery from power-off during this critical chain rebuild. The worst case scenario of power-off during a chain build would result in garbage data generated from the partially built new chain. However, the original File Entry Table chain remains intact, with the existing fragmentation as before. The VSBs from the inaccessible new chain will be recovered by System Cleanup in Initialization. Recalling the FETCleanup routine will collapse the FET chain, removing fragmentation caused by deleted file entries.

The automatic FETCleanup can be triggered after deletion of each file via setting a user-defined compile-time option AUTO_FETCLEANUP to TRUE.

Possible Return Error Codes:

ERR_READ—Indicates an error reading the flash component.
ERR_PARAM—Indicates an incorrect parameter to a function.
ERR_NOT_FOUND—Specified VSB was not found in component.
ERR_MEMORY_FULL—Free VSB was not found in component.
ERR_VSB_INVALID—An invalid VSB number was specified on function entry.
ERR_NONE—Function completed successfully.
ERR_SPACE—Reclamation needs to be done.
ERR_WRITE—Either flash component is bad or V_{pp} is out of tolerance.

3.2.8.8 FITCleanup

The SP_FITCleanup subcommand processes a collapse of the File Information Table on the specified file by removing any entries whose num_valid_bytes are zero. The type and id fields must be provided in the CMD_CTRL structure as input in order to recover the target file. FIT_Cleanup will locate the file by using a FIND_MATCHED search. The routine has a built-in algorithm that decides whether to cleanup or if a FIT cleanup is currently not needed.

It performs a File Information Table scan and reports the total number of FIT entries and deleted entries. The algorithm determines whether there are enough to deleted FIT entries to justify a collapse of the entire chain. It also determines the total number of sectors required to rebuild the chain against the total number available. Removing deleted entries compresses one or more File Information Tables in the chain. Although there may be deleted entries, there may not be enough fragmentation which would trigger a chain collapse. A collapse reduces the overall total number of File Information Tables in the chain.

The top of the chain gets written out last, to provide recovery from power-off during this critical chain rebuild. The worst case scenario of power-off during chain build would result in garbage data generated from the partially built new chain. However, the original File Information Table chain remains intact, with the existing fragmentation as before. The VSBs from the inaccessible new chain will be recovered by System Cleanup in Initialization. Recalling the FIT_Cleanup routine will collapse the FIT chain, removing fragmentation caused by deleted FIT entries.

FIT_Cleanup is enabled in the code with the DO_FITCLEANUP compile time option (in vsbext.h).

Possible Return Error Codes

ERR_READ—Indicates an error reading the flash component.
ERR_PARAM—Indicates an incorrect parameter to a function.
ERR_NOT_FOUND—Specified VSB was not found in component.
ERR_MEMORY_FULL—Free VSB was not found in component.
ERR_VSB_INVALID—An invalid VSB number was specified on function entry.
ERR_NONE—Function completed successfully.
ERR_SPACE—Reclamation needs to be done.
ERR_WRITE—Either flash component is bad or V_{PP} is out of tolerance.

3.2.8.9 Default

Returns an ERR_PARAM message.

This may be modified to give the user direct access to hardware level functions.

3.3 Virtual Small Block (VSB) Flash Media Manager Functions

The following functions are supplied by the VSB Flash Media Manager. These functions support the VSB File Manager (VFM) by providing a method to manage sectors or “virtual small blocks” on the media. This software performs all logical to physical translations necessary to allow data to be relocated successfully.

The virtual small block interface will translate logical sectors (or virtual small blocks) to physical addresses. This may allow an existing disk drive to be replaced by flash media.

Table 3-3. VSB Function Summary

Function	Description
InitializeVSB	Initializes entire flash media
FormatVSB	Erases all blocks in a specified component
FindVSB	Searches VATs from beginning of component for specified VSB
WriteSector	Updates necessary globals and writes one VSB from a specified buffer
ReadVSB	Reads one VSB into a buffer
ReadBytes	Reads a specified number of bytes
DiscardVSB	Marks allocated data as discarded for cleanup at a later time
ReclaimVSB	Performs cleanup on one block
GetMemoryStatus	Determines number of free VSBs, as well as the number of deallocated VSBs to return to user

3.3.1 Initialize VSB

The **InitializeVSB** function must be called before any other VSB support functions. This function will scan all available blocks in a flash component to find any blocks in an invalid state. Any block in the VSB_FB_RECOVER state will be erased. Any block in the VSB_FB_ERASING state points to the block being erased. This operation will be completed and the block will be put into the VSB_FB_WRITE state. After all blocks are determined to be in a valid state, each VAT will be scanned for any VSBs in an invalid state. Any VSB in the VSB_ALLOCATING state will be discarded. Any VSB in the VSB_ALLOCATED state will force the system to search for a pre-existing logical VSB, discard the old VSB, then change the new VSB to the VSB_VALID state. Once the state of all blocks and VSBs are determined to be valid, global variables will be initialized and the function will return. A return value of zero indicates success. Non-zero values require *FormatVSB***;** the value specifies the flash component that needs formatting. The prototype for *InitializeVSB* follows.

Call Format:

The ‘C’ call to the *InitializeVSB* function is as follows:

WORD InitializeVSB(BYTE Component, BYTE Threshold);

Entry Parameters:

Component—Flash component to initialize.
Threshold—Specifies the minimum number of free VSBs before triggering automatic reclamation.

Global Variables Affected:

status_ptr->clean_sectors—Set to number of free VSBs in component.
status_ptr->dirty_sectors—Set to number of discarded VSBs in component.
status_ptr->valid_sectors—Set to number of valid VSBs in component.
ReclaimEnable—Set to value which determines if auto Reclamation is allowed or not.
WriteBlock—Set to last block found with a block attribute of VSB_FB_WRITE.
SpareBlock—Set to first block found with a block attribute of VSB_FB_ERASED.
CurrentBlock—Set to same value as *WriteBlock*.
CurrentComponent—Set to Component specified on function entry.

Return Values:

ERR_NONE—Function completed successfully.
ERR_FLASH_INVALID—Either flash is inaccessible (unknown state), or does not match the Intelligent Identifier values.
ERR_FORMAT_INVALID—An unrecoverable VAT entry or block attribute was found.
ERR_ERASE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_WRITE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_SETUP—Maximum logical VSB size exceeds the number of bits available in the VAT entry.

3.3.2 Format VSB

The *FormatVSB* function simply erases all blocks in the specified flash component. *InitializeVSB* must be called after *FormatVSB* completes. A return value of zero indicates success, non-zero indicates an erase failure occurred. The prototype for *FormatVSB* follows.

Call Format:

The 'C' call to the *Format_VSB* procedure is as follows:

WORD FormatVSB(BYTE Component);

Entry Parameters:

Component—Specifies the flash component number to erase.

Global Variables Affected:

None.

Return Values:

ERR_NONE—Function completed successfully.
ERR_ERASE—Either flash component is bad or V_{pp} is out of tolerance.

3.3.3 Find VSB

The *FindVSB* function searches for the specified VSB and sets up necessary global variables. Any subsequent read or discard operations will use the global variables. The search will always start in the current block. A return value of zero indicates the VSB was found, non-zero indicates the VSB was not found. The prototype for *FindVSB* follows.

Call Format:

The 'C' call to the *FindVSB* procedure is as follows:

WORD FindVSB(BYTE Component, WORD VSBToFind);

Entry Parameters:

Component—Specifies the flash component to search.
VSBToFind—Specifies the VSB number to search for.

Global Variables Affected:

CurrentVSB—Set to physical offset within block.
CurrentBlock—Set to block VSB was found in.
CurrentComponent—Set to Component specified on function entry.

CurrentIndex—Set to VAT entry index within block.
CurrentByte—Set to 0.
VSBFound—Set to TRUE.

Return Values:

ERR_NONE—Function completed successfully.
ERR_VSB_INVALID—An invalid VSB number was specified on function entry.
ERR_NOT_FOUND—Specified VSB was not found in component.

3.3.4 Write Sector

The *WriteSector* finds the next free VSB, then updates the physical VSB's VAT entry to indicate that the VSB is in use (VSB_ALLOCATING). If no free VSBs are found, the reclamation process will be run if enabled. The number of bytes to write may be from zero up to VSB_SIZE. *CurrentByte* will be maintained to count the number of bytes written to the selected VSB. If the number of bytes exceeds VSB_SIZE, an error will be returned. After the data is successfully written, this routine updates the physical VSB's VAT entry to indicate that the data write is complete, but the corresponding VAT entry has to be marked valid (VSB_ALLOCATED). If *RewriteFlag* is set TRUE, any VSB with the same ID number will be discarded. And last, the VSB just written will be marked as valid by writing VAT entry VSB_VALID.

Call Format:

The 'C' call to the *WriteSector* procedure is as follows:

WORD WriteSector(BYTE Component, WORD VSBToWrite, BYTE *DataToWrite, WORD RewriteFlag, WORD ByteCount);

Entry Parameters:

Component—Specifies the flash component to setup write in.
VSBToWrite—Specifies the VSB number to setup write for.
*DataToWrite—Pointer to buffer containing the data to write.
RewriteFlag—Set to TRUE or FALSE to decide if/if not we need to invalidate the existing same logical VSB ID.
ByteCount—Number of bytes to write, if this parameter equals 0 the write is terminated.

Global Variables Affected:

VSBToWrite—logical VSB number that will be written.
CurrentByte—Set to 0.
VSBFound—Set to FALSE.

Return Values:

ERR_NONE—Function completed successfully.
ERR_WRITE—Either flash component is bad or V_{PP} is out of tolerance.
ERR_NOT_FOUND—Specified VSB was not found in component.

ERR_VSB_INVALID—An invalid VSB number was specified on function entry.
ERR_MEMORY_FULL—Free VSB was not found in component.
ERR_SPACE—Reclamation needs to be done.

3.3.5 Read VSB

The *ReadVSB* function reads the VSB_SIZE byte from the previously found VSB (using *FindVSB*). A return value of zero indicates that *ReadVSB* was successful. Non-zero indicates a failure occurred; the value will indicate the error that occurred. The prototype for *ReadVSB* follows.

Call Format:

The 'C' call to the *ReadVSB* procedure is as follows:

WORD ReadVSB(BYTE *DataToRead);

Entry Parameters:

*DataToRead—Pointer to destination buffer.

Global Variables Affected:

None.

Return Values:

ERR_NONE—Function completed successfully.
ERR_READ—Indicates an error reading the flash component.
ERR_NOT_FOUND—Specified VSB was not found in component.

3.3.6 Read Bytes

The *ReadBytes* function reads the specified number of bytes from the previously found VSB (using *FindVSB*). The number of bytes to read may be from zero up to VSB_SIZE. *CurrentByte* will be maintained to count the number of bytes read from the selected VSB. If the number of bytes exceeds VSB_SIZE, an error will be returned. A return value of zero indicates that *ReadBytes* was successful. Non-zero indicates a failure occurred; the value will indicate the error that occurred. The prototype for *ReadBytes* follows.

Call Format:

The 'C' call to the *ReadBytes* procedure is as follows:

WORD ReadBytes(BYTE *DataToRead, WORD ByteCount);

Entry Parameters:

*DataToRead—Pointer to destination buffer.
ByteCount—Number of bytes to read.

Global Variables Affected:

CurrentByte—Updated upon each call to *ReadBytes.*

Return Values:

ERR_NONE—Function completed successfully.
ERR_READ—Indicates an error reading the flash component.
ERR_NOT_FOUND—Specified VSB was not found in component.
ERR_VSB_OVERFLOW—Indicates that the current flash write request exceeds the VSB size.

3.3.7 Discard VSB

The *DiscardVSB* function marks the previously found VSB (using *FindVSB*) as discarded. A return value of zero indicates that *DiscardVSB* was successful. Non-zero indicates a failure occurred; the value will indicate the error that occurred. The prototype for *DiscardVSB* follows.

Call Format:

The ‘C’ call to the *DiscardVSB* procedure is as follows:

WORD DiscardVSB(void);

Entry Parameters:

None.

Global Variables Affected:

VSBFound—Set to FALSE on completion.

Return Values:

ERR_NONE—Function completed successfully.
ERR_NOT_FOUND—*FindVSB* was not called first.
ERR_WRITE—Either flash component is bad or V_{PP} is out of tolerance.

3.3.8 Reclaim VSB

The *ReclaimVSB* function will reclaim discarded VSBs from a single block in a single flash component. The block will be selected by *ReclaimVSB*, the selection will be made based on the largest number of discarded VSBs. To recover all discarded VSBs in a component, *ReclaimVSB* must be called until ERR_NOT_FOUND is returned. A return value of zero indicates that *ReclaimVSB* was successful. Non-zero indicates a failure occurred; the value will indicate the error that occurred. The prototype for *ReclaimVSB* follows.

Call Format:

The ‘C’ call to the Reclaim procedure is as follows:

WORD ReclaimVSB(BYTE Component);

Entry Parameters:

Component—Flash component to perform reclamation on.

Global Variables Affected:

status_ptr->clean_sectors—Set to number of free VSBs in component.
status_ptr->dirty_sectors—Set to number of discarded VSBs in component.
status_ptr->valid_sectors—Set to number of valid VSBs in component.
SpareBlock—Will be updated to reflect any changes to the current spare block.
WriteBlock—Will be updated to reflect any changed to the current write block.

Return Values:

ERR_NONE—Function completed successfully.
ERR_NOT_FOUND—No discarded VSBs were found.
ERR_WRITE—Either flash component is bad or V_{pp} is out of tolerance.
ERR_WRITE_IN_PROGRESS—A write is in progress.
ERR_READ—Indicates an error reading the flash component.
ERR_SPACE—Reclamation needs to be done.
ERR_MEMORY_FULL—Free VSB was not found in component.
ERR_VSB_OVERFLOW—Indicates that the current flash write request exceeds the VSB size.
ERR_VSB_INVALID—An invalid VSB number was specified on function entry.
ERR_ERASE—Either flash component is bad or V_{pp} is out of tolerance.

3.3.9 GetMemoryStatus

The *GetMemoryStatus* function returns information to indicate the number of free VSBs remaining, the number of valid VSBs, and the number of discarded VSBs that exist. The syntax for *GetMemoryStatus* follows.

Call Format:

The ‘C’ call to *GetMemoryStatus* is as follows:

void GetMemoryStatus(WORD *freeSectors, WORD *deallocatedSectors, WORD *validSectors);

Entry Parameters:

freeSectors—pointer to variable which will contain the number of free sectors available upon exit.

deallocatedSectors—pointer to variable which will contain the number of dirty sectors available to reclaim.

validSectors—pointer to variable which will contain the number of valid sectors.

Possible Errors:

None.

3.4 Low-Level Driver Interface Functions

The following functions are supplied by the low-level driver. These functions are completely implementation dependent and must be supplied by the OEM for their particular implementation.

3.4.1 Low-Level Compatibility Check

This function will verify that the flash is compatible with the higher level flash algorithms used. Two methods are used: CFI query and intelligent identifier command. If the CFI query indicates that CFI-compliant Intel Flash devices are not being used, an error is returned. However, if the query fails to return any information, the legacy intelligent identifier method is used as described below.

This function sends the intelligent identifier command to each flash chip in the media, reads the manufacturers identifier, and verifies that the value read equals the Intel® Manufacturer Identifier. If any intelligent identifiers in the media do not equal the Intel Manufacturer ID, an error is returned.

Call Format:

The 'C' call to this procedure is as follows:

BYTE FlashDevCompatCheck(DWORD address);

Possible Return Error Codes:

ERR_JEDEC—Must use this if all manufacturers IDs do not match the Intel Manufacturer ID
ERR_NONE—Function completed successfully.

3.4.2 Low-Level Read

This function will read a predetermined number of bytes (length parameter) from an absolute physical address (media_address parameter) and will place the data into a buffer passed in (buffer parameter). The return code should indicate success or failure as defined below.

Call Format:

The 'C' call to this procedure is as follows:

BYTE FlashDevRead(DWORD address, DWORD length, BYTE*buffer)

Possible Return Error Codes:

ERR_READ—Indicates an error reading the flash component.
ERR_NONE—Function completed successfully.

3.4.3 Low-Level Write

This function writes the data from a specified buffer to the destination address. The return code should indicate success or failure as defined below. The offset determines what address in the device to begin the write.

Call Format:

The 'C' call to this procedure is as follows:

BYTE FlashDevWrite(DWORD offset, DWORD length, BYTE * buffer)

Possible Return Error Codes:

ERR_WRITE—Either flash component is bad or VPP is out of tolerance.
ERR_NONE—Function completed successfully.

3.4.4 Low-Level Erase

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.

Call Format:

The 'C' call to this procedure is as follows:

BYTE FlashDevEraseBlock(DWORD erase_block_address)

Possible Return Error Codes:

ERR_ERASE—Either flash component is bad or VPP is out of tolerance
ERR_NONE—Function completed successfully.

Chapter 4
Power Off Recovery Process in VFM

Power Off Recovery Process in VFM 4

4.1 Preparation for Power Loss Recovery in VFM

VFM implements extremely robust power loss recovery mechanisms throughout the code. This safeguards your valuable data stored in the flash. By utilizing the unique characteristics of flash, we can assure the integrity of the existing data should power fail (or system lock-up, or other similar issue) while writing to flash. The following are the “worst case” power loss situations for each FSD operation:

4.1.1 During FSD_WRITE

Append

Issue– Appended sectors are written to the media, up to the point of the power failure. Depending upon the failure point, the sector that is written last may be treated as garbage if write was not completed, or the write might have been completed and *VATEntry* alone might not have been updated. The media manager controls the recovery and state transitioning for the current sector being written; refer to Section 2.2.2, “VAT State Transitions” on page 2-9, and Section 3.3.1, “Initialize VSB” on page 3-17, in this document. The file may report only a portion of the appended data, representative of only those sectors which could be written and whose File Information Tables were updated properly, prior to the power loss.

Recovery– If there is a garbage sector, it will be discarded and *VATEntries* will be appropriately updated during initialization process; refer to Section 4.2.2, “Recovery Process During Initialization (SP_INIT) in VSB” on page 4-3.

4.1.2 During FSD_EDIT

Append

Issue– Appended sectors are written to the media, up to the point of the power failure. Depending upon the failure point, the sector that is written last, may be treated as garbage if write was not completed or the write might have been completed and *VATEntry* alone might not get updated, prior to the power loss. The media manager controls the recovery and state transitioning for the current sector being written; refer to Section 2.2.2, “VAT State Transitions” on page 2-9, and Section 3.3.1, “Initialize VSB” on page 3-17, in this document. The file may report only a portion of the appended data, representative of only those sectors which could be written and whose File Information Tables were updated properly, prior to the power loss.

Recovery– If there is a garbage sector, it will be discarded and *VATEntries* will be appropriately updated during initialization process; refer to refer to Section 4.2.2, “Recovery Process During Initialization (SP_INIT) in VSB” on page 4-3.

Insert

Issue– Insert builds the inserted data chain, then as a final step, inserts the new chain into the existing file chain. If a power loss should occur during this process, the new data

chain does not impact the original data chain. The original chain remains intact and the new data chain becomes garbage data.

Recovery– At power-on, the Initialization process will recover those garbage sectors that are not hooked into the file chain, and the original chain remains as before. If insert is still necessary, application will have to rerun Insert.

Replace

Issue– Replace can guarantee a graceful recovery up to a sector boundary. Those sectors which were replaced byte-for-byte get written up to the point of the power loss. The current sector being replaced may or may not be written to the media. Refer to the media manager control for individual sector recovery. The file reflects only the new data for only those sectors rewritten, prior to the power loss.

Recovery– Application will have to rerun Replace again for a complete or partial update if necessary.

Byte Delete

Issue– Delete may discard only a few sectors deleted resulting in partial delete of the request, up to the point of the power failure. The file still contains the portion of the data which was not deleted.

Recovery– Application has to delete the remaining bytes by doing another Byte Delete if necessary.

4.1.3 During FSD_DELETE

Append

Issue– This request begins by marking the file entry as FILE_DISCARDING and scans to the end of the File Information Table chain for the target file. It discards each sector corresponding to a FIT entry, then discards the related FIT. Working backwards in the file chain, it deletes each data sector and control structure for the file, then marks the file entry as FILE_DISCARDED. If there's a power loss, a dangling reference may be created within the file chain, which may be removed during Initialization.

Recovery– At power on, initialization will remove all the files which have a FILE_DISCARDING attribute in their file entry. This is done automatically during initialization and is not a user-defined option.

4.1.4 During FETCleanup

Append

Issue– The *FETCleanup* routine will collapse the FET chain, removing fragmentation caused by deleted file entries. The top of the chain gets written out last, to provide recovery from power-loss during this critical chain rebuild. The worst case scenario of power-loss during a chain build would result in garbage data generated from the partially-built new chain. However, the original File Entry Table chain remains intact, with the existing fragmentation as before.

Recovery– The VSBs from the inaccessible new chain will be recovered during initialization by doing a System Cleanup. This System Cleanup is done automatically during initialization and is not a user-defined option.

4.2 Power Loss Recovery Process

4.2.1 Status Bits

Upon initialization, the states of all status bits are checked. If a status bit is in an undefined state, the function will return ERR_FORMAT_INVALID, which indicates that the media is corrupted.

4.2.2 Recovery Process During Initialization (SP_INIT) in VSB

During Initialization, the following steps take place for proper Power Loss Recovery.

Step 1:

All blocks are validated.

Erase any block in the VSB_FB_RECOVER state (in the midst of Reclaim).

Completely erase any block in the VSB_FB_ERASING state (in the midst of erasing). This block is put into the VSB_FB_WRITE state after erase completion. If more than one block is found to be in the VSB_FB_ERASING state (will happen only on an unformatted flash component), it is assumed that the entire flash component is in an invalid state and it must be reformatted. Hence returns ERR_FORMAT_INVALID.

Step 2:

Validates all the VAT entries.

Any VSB in the VSB_ALLOCATING state (in the midst of write or edit) is discarded.

Any VSB in the VSB_ALLOCATED state (write has been completed but VAT Entry did not get updated) will force the system to search for a pre-existing logical VSB, discard the old VSB, then change the new VSB to the VSB_VALID state.

Step 3:

Identifies partially deleted files and completes their deletion.

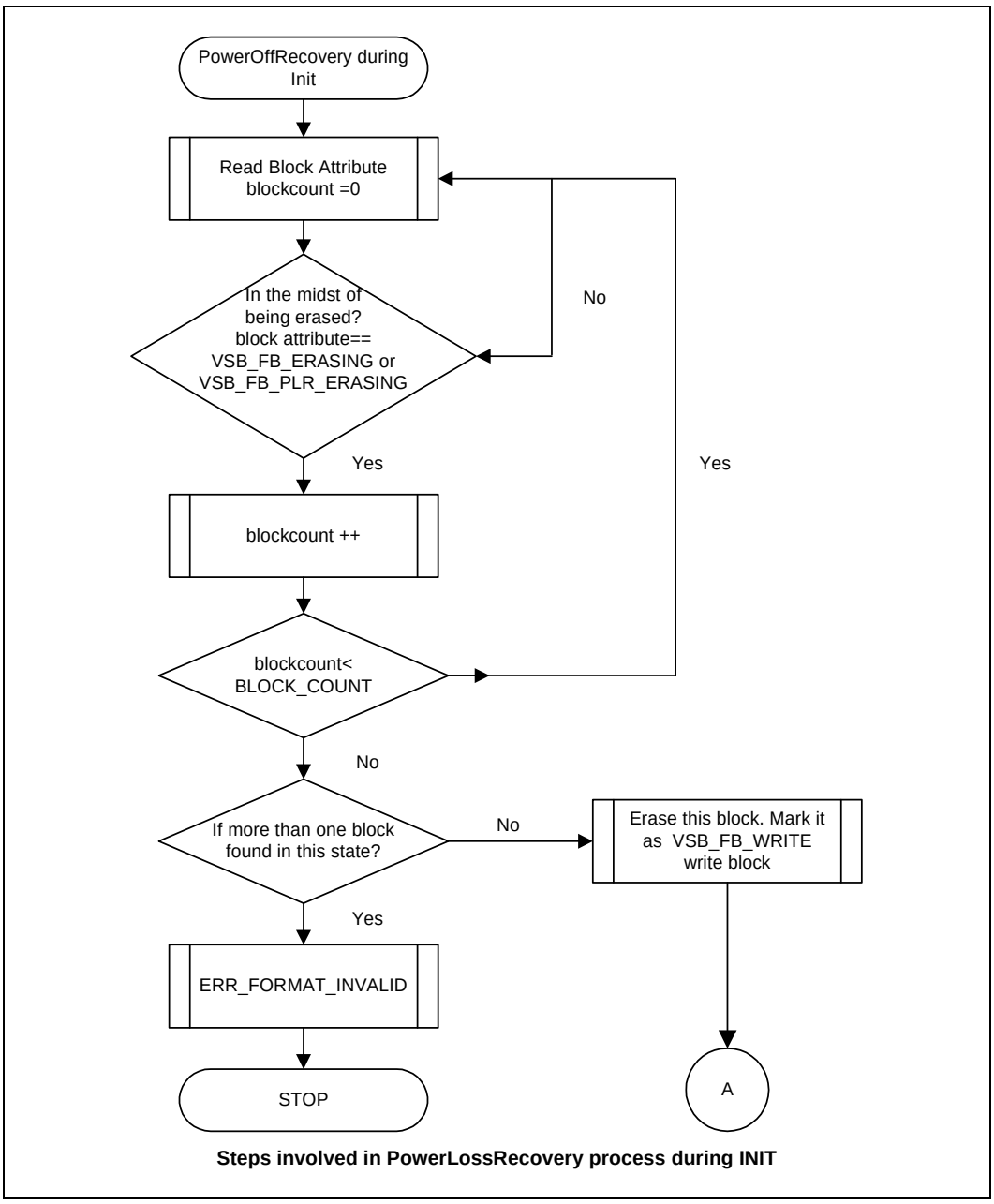
Scans the FET chain and identifies the file entries marked as FILE_DISCARDING. This flag indicates that a “delete-in-process” never completed, possibly due to a power failure. Upon encountering this flag, it proceeds to scan to the end of the File Information Table chain for this file. Upon reaching the expected TABLE TERMINATOR value, or recognizing a dangling reference pointer in the link structure, it deletes the remainder of the file working backwards in the chain.

Step 4:

Unreferenced sectors are discarded.

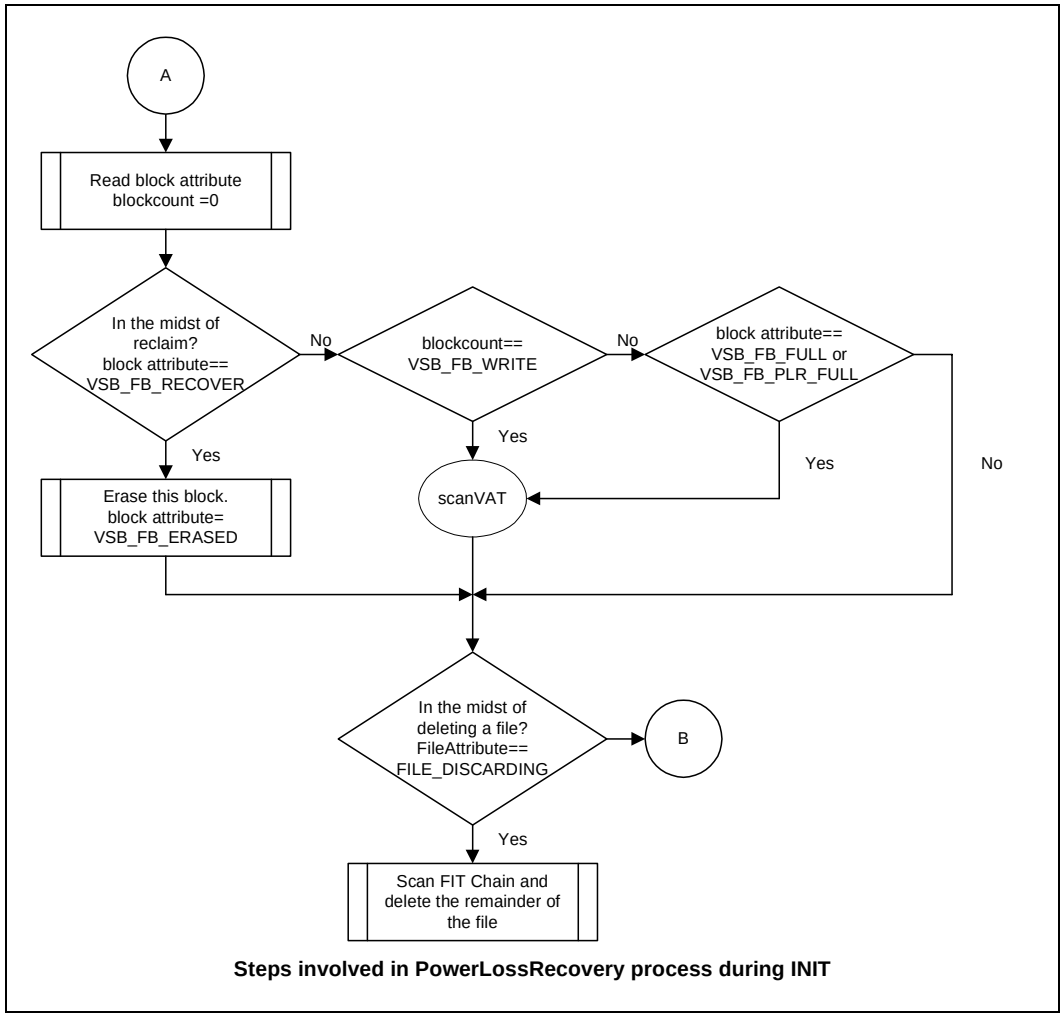
Any used VSBs which are not recognized as part of the File Entry Table chain, File Information Table and corresponding data sectors are considered “garbage” and discarded automatically by doing a System Cleanup process. After reclaim recovers those dirty sectors, they can be used for further data writes.

Figure 4-1. Power Loss Recovery Flow



Steps involved in PowerLossRecovery process during INIT

Figure 4-2. Power Loss Recovery Flow (Continued)



Steps involved in PowerLossRecovery process during INIT

Figure 4-3. Power Loss Recovery Flow (Continued)

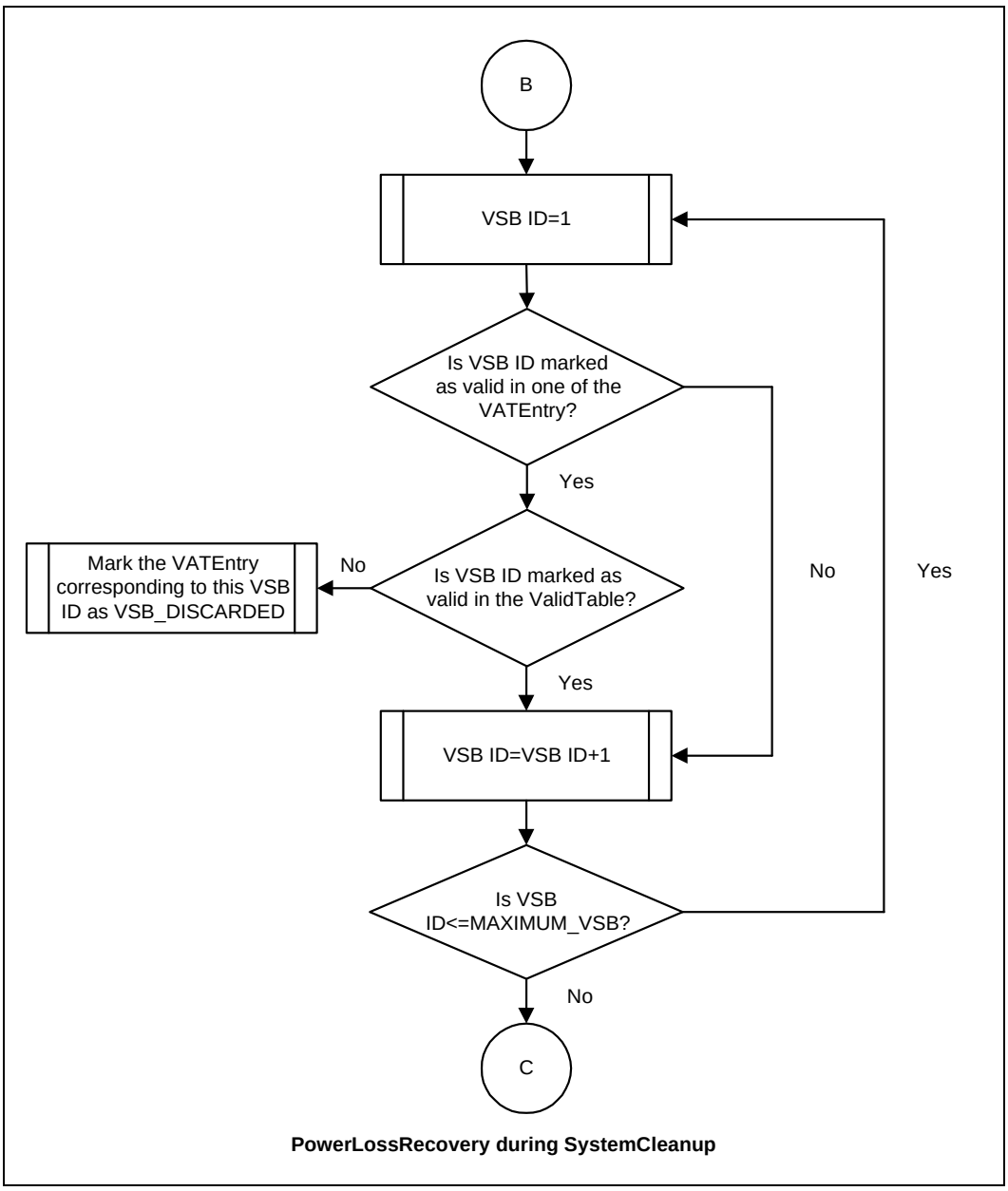


Figure 4-4. Power Loss Recovery Flow (Continued)

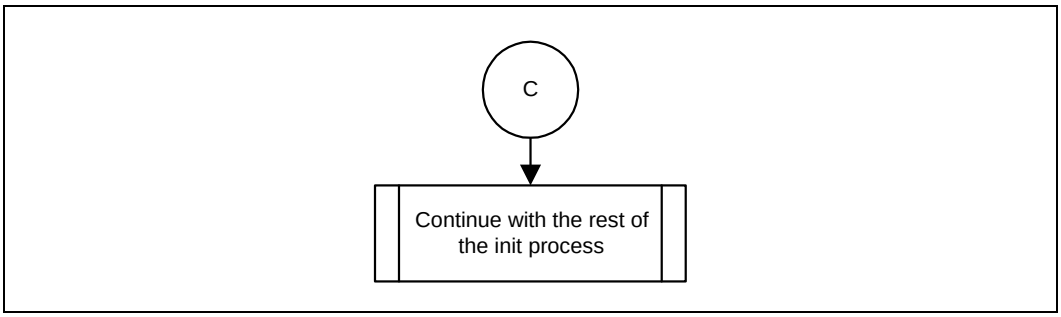
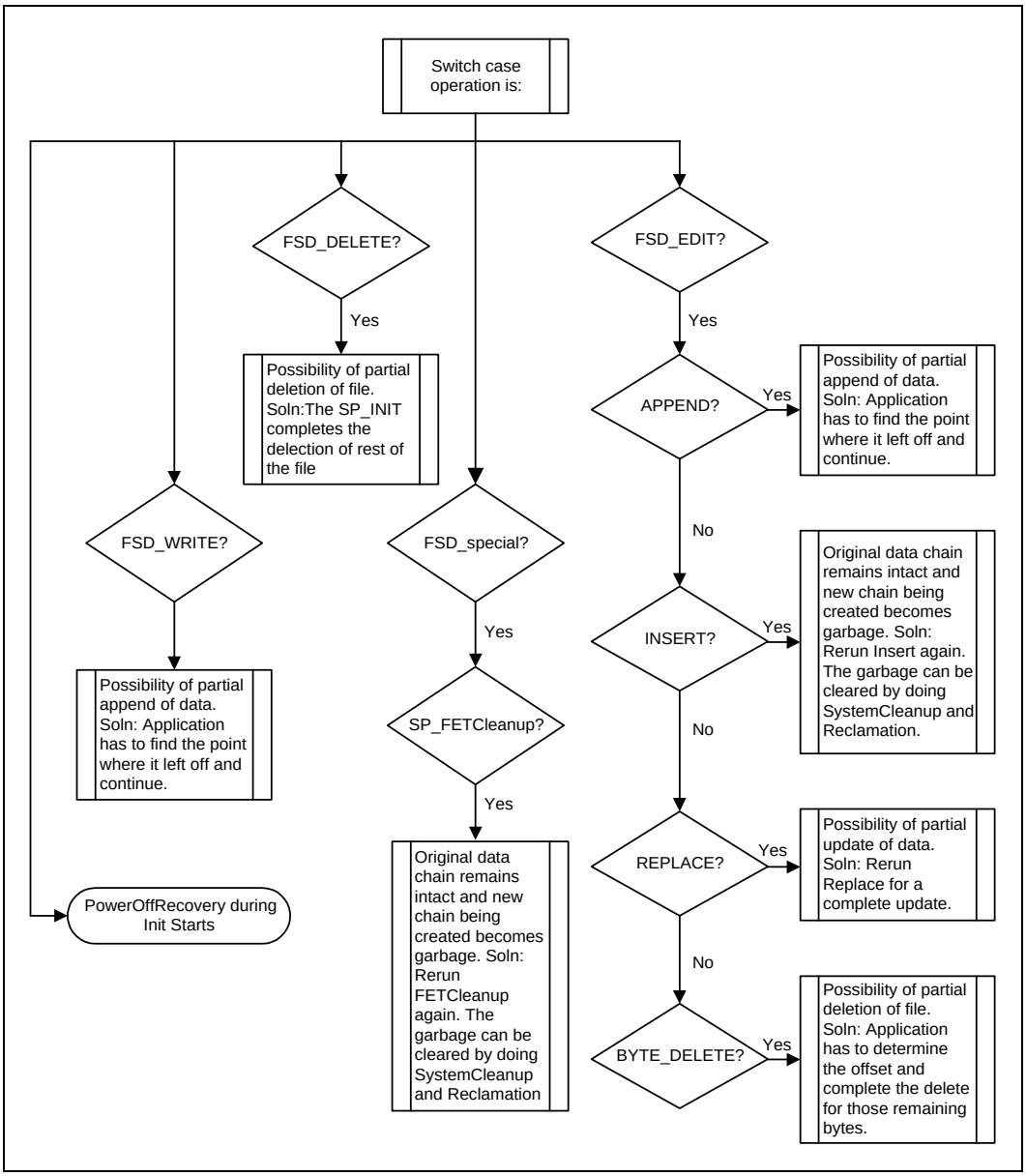


Figure 4-5. Power Loss Recovery Flow (Continued)



Chapter 5
Porting VFM to Your System

Porting VFM to Your System 5

5.1 Introduction

Flash technology brings unique attributes to system memory. Similar to RAM, flash memory changes data by electrical, in-system updates. The nonvolatile nature of flash, like ROM, provides data retention after removing power. Flash memory reads and writes on a byte-by-byte basis. Flash adds a new bit requirement; once written, the bit's state remains the same until it is erased. Bytes may be rewritten, provided each bit changes from an erase state to a non-erase state. Software manages this unique characteristic of flash media.

The **V**irtual Small Block **F**ile **M**anager, **VFM**, tracks files broken down into smaller blocks or sectors. Two major "functions" comprise the VFM solution; the VFM **F**ile **S**ystem **D**river (**FSD**) and the Flash Media Manager, known as **V**irtual **S**mall **B**lock (**VSB**). FSD represents the core filing system and acts as the File System (OS) Interface to the Operating System. VSB handles the translation of logical small blocks to physical flash erase blocks, as well as the flash media management. Additionally, a low level driver supplies the necessary interface to control the flash command set and flash interface control.

FSD, VSB, and the low level drivers are written in ANSI 'C.' See Figure 1-1, "Application's Request Reaching the Partition" on page 1-4 for a review of the layers. This chapter discusses the software modules that are part of the VFM package and how to port this software to your system and hardware setup. Familiarity with programming and the general contents of the reference software is assumed.

Note: Throughout the code and this chapter, the term "partition" represents a VFM data partition. A VFM partition may represent a single component, or up to several components. The user defines the size of each partition, described in Section 5.3, "Media Partitioning" on page 5-2. If using virtual components, the term "component" refers to the "virtual component." See Section 5.3.2, "Example Media Partitioning" on page 5-4, for more details on virtual components.

5.2 Getting Started

The first step toward porting VFM to your system is to determine what type of flash media will be used. Current available options include PCMCIA flash cards, Resident Flash Arrays (RFA), or any Intel Flash-based proprietary card you may be working with. The majority of this chapter will discuss flash in terms of removable or non-removable media. Flash cards are considered removable, unless your system always assumes the flash is present. RFAs, and any cards which are never removed, are considered non-removable.

This adaptation of VFM only supports non-removable media, in card or RFA configurations. Systems using removable media must be concerned with placing a format onto the media and card detection. Additional support routines are required to adapt VFM to support this functionality. This chapter focuses primarily on non-removable media and briefly covers requirements for removable media.

5.2.1 Modification Partitioning

There are three main steps involved to porting VFM to your system. These include:

- Adaptation to your flash setup.

  If the flash characteristics (e.g., density, blocking, etc.) in your system have changed from a previous design, be sure to synchronize all affected files to reflect the new information. These include files *flashdef.h* and *datatype.h*. For more information on flash-specific header files, see Section 5.5, “Modifying Source and Header Files (Compile-Time Options)” on page 5-23.

- Replacing Low Level Driver flash interface functions.

  For more information, see Section 5.11, “Replacing Low-level Driver Interface Functions LOWLVL, PCIC” on page 5-42.

- Creation of an O/S interface or modification of the direct application interface

  For more information, see Section 5.12, “Creation of an O/S Interface or Direct Application Access” on page 5-44.

5.3 Media Partitioning

The FSD provides a mechanism to identify partition dimensions and how the entire media gets divided. The FSD calls low level functions which receive the partition information structure, identifying the partition affected by the operation.

Note: Throughout the examples of media characterization, we assume a media erase block size of 128 KB (0x20000H). This is a typical size, but may not match your particular hardware configuration.

5.3.1 Partition Info Structure

Partition boundaries are described by the PARTITION_INFO structure and are contained in the Partition Array. The user establishes the maximum number of partitions used by VFM. The value for the maximum number of partitions gets set through the MAX_PARTITIONS define. As shown in Figure 5-1, “Partition Info Control Structure” on page 5-3, each partition is assigned the number for the physical component(s) corresponding to that partition. Dimensions are implementation specific and must be defined via the preprocessor definitions described below. Refer to the (*vsbext.h*) file, described in Section 5.5, “Modifying Source and Header Files (Compile-Time Options)” on page 5-23.

Note: The sample solution assumes non-removable media and assigns static values. When using multiple types of removable media, the code may be adapted to include a low level “mount.” A mount routine provides a mechanism which fills the Partition Array structure to correspond to different media types, thus allowing a way to handle insertion and removal of different types of flash cards. Low level routines may be incorporated into VFM to handle card insertion and removal.

Listing 5-1. Partition Preprocessor Definitions

```
#define     MAX_PARTITIONS      3           /* user defined */
#define     BLOCK_SIZE          0x20000     /* media specific */

#define     PARTITION_BEGIN_1   0           /* user defined */
#define     PARTITION_END_1     0           /* Partition 0 */

#define     PARTITION_BEGIN_2   1           /* user defined */
#define     PARTITION_END_2     1           /* Partition 1 */

#define     PARTITION_BEGIN_3   2           /* user defined */
#define     PARTITION_END_3     3           /* Partition 2 */
```

Figure 5-1. Partition Info Control Structure

```
typedef struct partition_info
{
    DWORD     block_size;                   /*Component block size*/
    WORD      partition_offset;             /*Partition start component (0 based)*/
    WORD      partition_end;                /*Partition end component (0 based)*/
}
PARTITION_INFO;

PARTITION_INFO partition_array[] =
{
    { BLOCK_SIZE_PARTITION_BEGIN_1, PARTITION_END_1} ,    /*Partition 0*/
    { BLOCK_SIZE_PARTITION_BEGIN_2, PARTITION_END_2} ,    /*Partition 1*/
    { BLOCK_SIZE_PARTITION_BEGIN_3, PARTITION_END_3}      /*Partition 2*/
{ ;
```

block_size

This field represents the block size of all components in the partition. This field gets initialized via the preprocessor definition, known as BLOCK_SIZE. The value should correspond to the size of a component's erase block. When considering card or RFA dimensions, be sure to identify paired devices when calculating this value. For example, set this value to 0x00020000H when using Value Series 100 cards which contain paired 28F008 components.

partition_offset

This field represents the zero-based, inclusive component number where the partition starts. This field gets initialized by the preprocessor definition which corresponds to the selected partition. Each partition begins on a component boundary. A component number within the media gets assigned as the start of this partition. Components are zero-based and sequentially ordered. When using cards with paired devices, consider the pair as a single entity.

partition_end

This field represents the zero-based, inclusive component number where the partition ends. This field gets initialized by the preprocessor definition which corresponds to the selected partition. Each partition ends on a component boundary. A component number within the media gets assigned as the end of the partition. Components are zero-based and sequentially ordered. When using cards or an RFA with paired devices, consider the pair as a single entity.

Note: The boundaries provide a mechanism to calculate the total media size. Care should be taken to avoid duplicate entries and/or overlap of components between partitions. The total amount of space available for data storage and total sectors may be calculated. Sample calculations will follow in Section 5.6, “Media Calculations” on page 5-30.

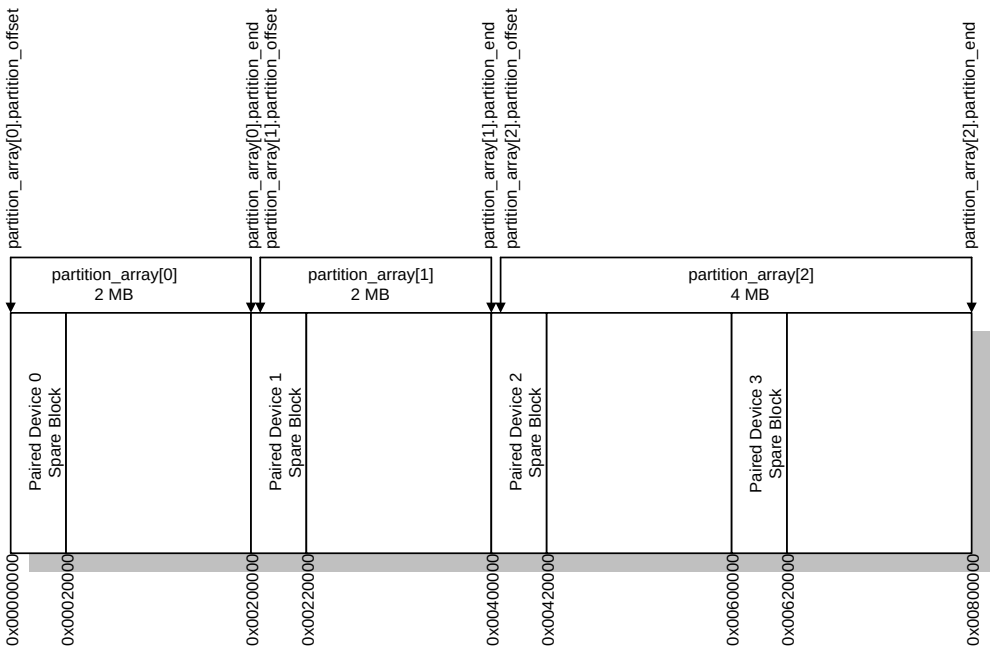
5.3.2 Example Media Partitioning

The following diagram provides an example of the PARTITION_INFO array. You will need to adapt this information to apply to your setup. The example below displays three partitions. If configuring a single partition, change the define labeled MAX_PARTITIONS accordingly; refer to a detailed description in Section 5.5, “Modifying Source and Header Files (Compile-Time Options)” on page 5-23.

This example displays four device-pairs, a total of eight 28F008 devices. This representation for an 8-MB Value Series 100 card may be adapted to an RFA implementation of multiple devices. There are three partitions shown in Figure 5-2 on page 5-5. The first two megabytes are allocated to the first partition, two megabytes to the next and the remaining four megabytes to the last partition.

The partition array fields do not contain absolute address offsets; instead, they provide logical component assignments. The flash media manager and low level flash control determine logical to physical translations. Together with the logical component number and default media parameters, the lower layers can calculate address offsets. As shown in Figure 5-2, the actual flash devices are represented in pairs, giving a total of four logical components. Logical component zero gets allocated to the first partition, component one to the second partition and the remaining components to the last partition. The flash media manager calculates a physical address in a partition using the logical component number, block number, block size, logical VSB number and sector (VSB) size. At initialization, the spare block gets assigned to the starting block. However, during the reclamation process, the spare gets relocated to a clean block in the component.

Figure 5-2. Media Partition Diagram for 28F008 Devices



partition_array[0].block_size = 0x20000 partition_array[2].block_size = 0x20000

partition_array[0].partition_offset = 0partition_array[2].partition_offset = 2

partition_array[0].partition_end = 0 partition_array[2].partition_end = 3

partition_array[1].block_size = 0x20000

partition_array[1].partition_offset = 1

partition_array[1].partition_end = 1

NOTE: The spare block is not fixed in one location. Rather, it may move as a result of the reclaim process. For more information on this topic, see Section 5.9, “Sector Allocation and Wear Leveling” on page 5-38.

5.3.3 VFM Exclusion Space

VFM has the ability to exclude a section of the flash from the VFM managed space on an erase block boundary. This feature is useful in a variety of applications such as those using the flash to handle code plus data. In this situation, the location of code may have certain restrictions beyond those for data, and must be treated separately. For example, the code may be located in the top of memory, with no option for moving it. Through use of exclusion area, VFM can handle these special requirements easily and conveniently.

In this scenario, the code would be stored in the “exclusion area,” whose size is defined at compile-time. For standard configurations, exclusion space definition is done by updating the *flashdef.h* file. This area is not touched by VFM. The remainder of the flash memory is managed by VFM and

is segmented into one or more “virtual components.” Virtual components are discussed in Section 5.3.5, “Virtual Components” on page 5-6. The actual size of the exclusion space is flexible, and may be any whole number of erase blocks.

5.3.4 Execute in Place (XIP) Considerations

VFM does not include the ability to execute code directly from the same component storing VFM managed data. Other techniques and modifications to the low level, beyond the scope of this chapter, are required for that functionality. Key modifications include copying low level routines to RAM where they can manage writes to the flash device. Other application and system dependent changes would also be necessary.

XIP modifications could be considered in environments where there are random or periodic events needing access to the code while erase, write, or reclaim operations could be in progress. For example, this situation is often encountered in systems using interrupts or code to handle hardware malfunctions. For XIP or real-time environments, consider one of the other media managers which may be more suitable for your application.

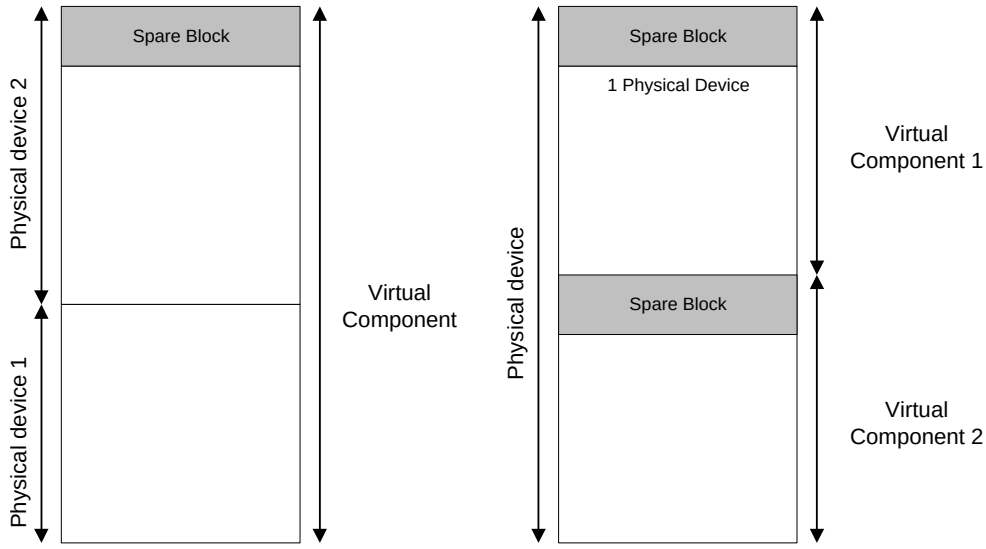
5.3.5 Virtual Components

The VSB Flash Media Manager allows a mechanism to define “virtual components,” which are flexible in size. A virtual component may be any number of erase blocks, from two up to the max size of the flash device, or even larger than one flash device. The minimum size of two blocks includes one for code or data, and a second block used as a spare during reclaim operations. The user defines virtual components easily by specifying the component count, component size, and the starting address offset from 0. In order to ensure efficient management of the flash and the smallest possible VFM code space, all virtual components in an application must be the same size as each other. As described in the examples below, partitions may be used to create different-sized sections of flash space.

One of the key advantages of using virtual components is that only one spare erase block is required for each one, regardless of how many physical flash devices are included. A second advantage is the ability to make virtual components smaller than a physical device. As described above, this scenario facilitates use of VFM with code plus data applications.

In Figure 5-3 on page 5-7, two different uses of virtual components are shown. In the left figure, two physical devices are combined into one virtual component. In the right figure, one physical device is divided into two virtual components. Each virtual component requires use of one spare block for reclaim operations.

Figure 5-3. Two Implementations of Virtual Components



NOTES:

Two physical devices comprise one virtual component; two virtual components reside in one physical device.

The spare block is not fixed in one location; rather, it moves to other locations as a result of the reclaim process. For more information on this topic, see Section 5.9, "Sector Allocation and Wear Leveling" on page 5-38.

5.3.5.1 Virtual Components Versus Partitions

Virtual components are different from partitions because they are defined in the VSB Flash Media Manager, rather than at the higher-level File System Driver. While partitions allow the user to define separate regions of flash that are separately addressable, virtual components are not visible from this level. Instead, partitions and virtual components work in concert to provide maximum flexibility in flash usage. The examples illustrated in the following sections reflect some of these applications, while many more are possible.

Some performance differences may become apparent if using extremely large virtual components. Since the VSB treats the virtual component as one large managed area, file segments may be distributed far apart on different physical devices. Updates to different physical devices or in non-contiguous memory windows must be made using separate write calls to the flash. In contrast, limiting virtual components to a small size could result in fewer separate write calls by the VSB and thus higher system performance.

5.3.5.2 Example Implementations of Exclusion Space

VFM has the ability to start the VFM managed space from an arbitrary erase block at either end of the flash device array or first component. This feature effectively gives you an exclusion area for other information (e.g., code, etc.) and works effectively in systems whose processors boot in either top boot or bottom boot configurations. For standard configurations of exclusion, you only need to change one file, *flashdef.h*. To combine multiple physical devices into one virtual component, simply reflect the combined size in the COMPONENT_SIZE definition.

Listing 5-2. Virtual Component Definition in flashdef.h (no Exclusion Area)

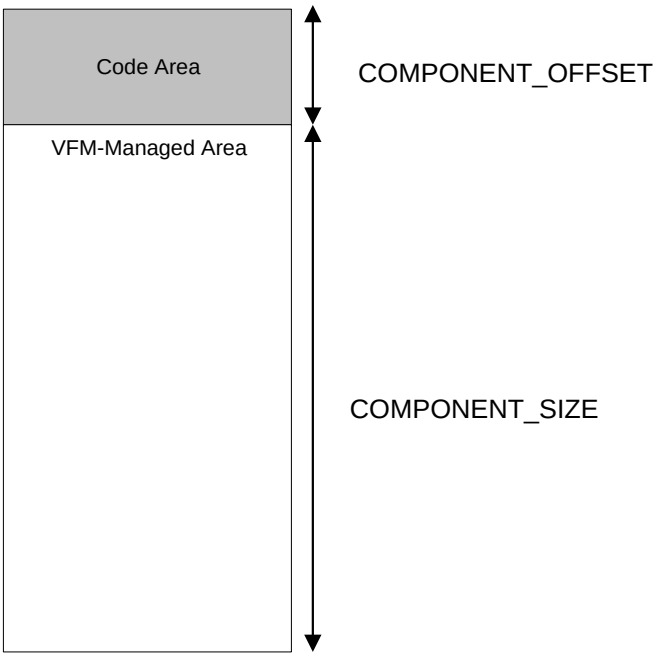
```
#define     COMPONENT_COUNT   1           /* Number of virtual components present */
#define     COMPONENT_SIZE    0x100000L   /* Size of 1 virtual component */

#define     COMPONENT_OFFSET  0x0L        /* Offset into the flash */
#define     BLOCK_SIZE        0x20000L    /* Size of 1 flash block */
#define     BLOCK_COUNT       (COMPONENT_SIZE /BLOCK_SIZE)
                                          /* number of discrete blocks in 1 virtual component */
```

Listing 5-2 is from *vfm\vsblib\flashdef.h*, which is the only code that you should need to change in order to setup an exclusion area. If you need to create a more complicated setup, you may need to change the partition definition in *vfm\vfs\vsbext.h*.

Further, Listing 5-2 shows the definition of one virtual component with no exclusion area. To exclude an area starting from 0x0L you would put that number into COMPONENT_OFFSET and reduce the COMPONENT_SIZE by the size of the exclusion area. See Figure 5-4 through Figure 5-9, “Large Virtual Component Spans Physical Devices” on page 5-13 for illustrations. All examples assume an erase block size of 128 KB. When configuring the offsets and component size for your application, use the erase block size for the flash device used.

If you do not wish to use exclusion space or virtual components, set the COMPONENT #defines corresponding to the physical flash devices in use.

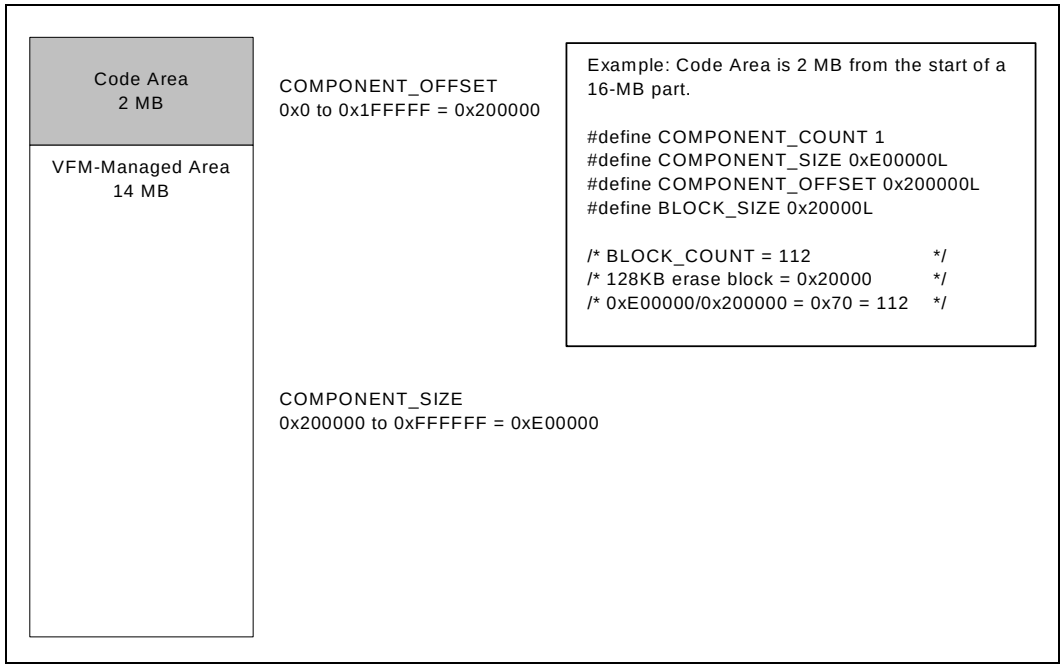
Figure 5-4. Use of Exclusion Space



Bottom-Boot Exclusion Area

Figure 5-5 shows a 16-MB device with a 2-MB exclusion area in the bottom of the device, with one 14-MB virtual component defined for the remainder of the flash device. The term “bottom-boot” reflects the convention that the bottom of the device is that starting with address 0.

Figure 5-5. Bottom-Boot Exclusion Area

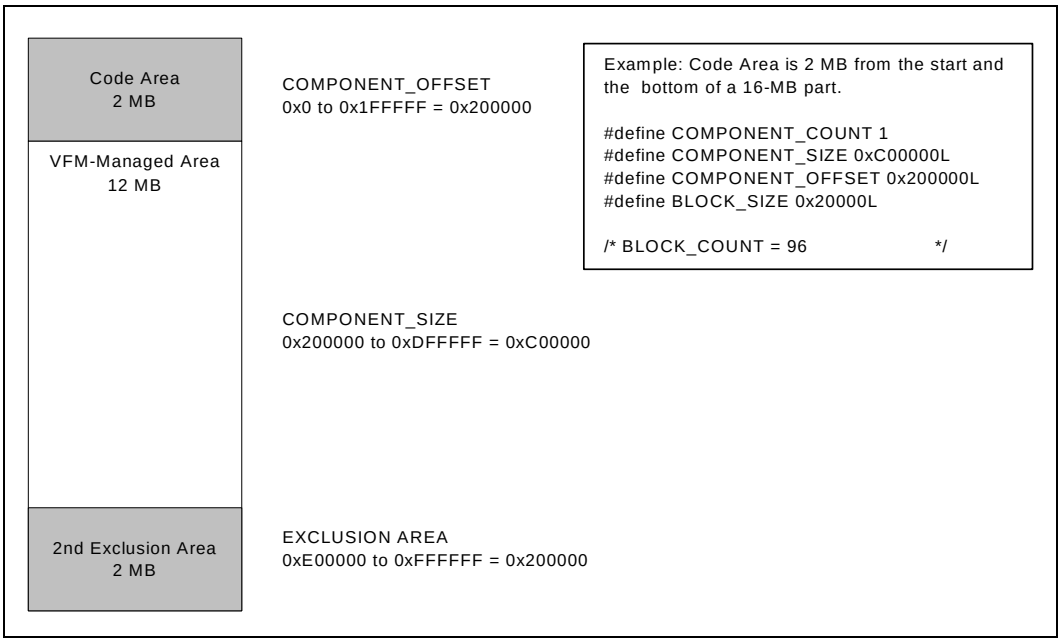


Advantages: This configuration provides space for code in addition to the managed data.

Two Exclusion Areas

Figure 5-6, “Two Exclusion Areas” on page 5-10 illustrates the implementation of two exclusion areas.

Figure 5-6. Two Exclusion Areas



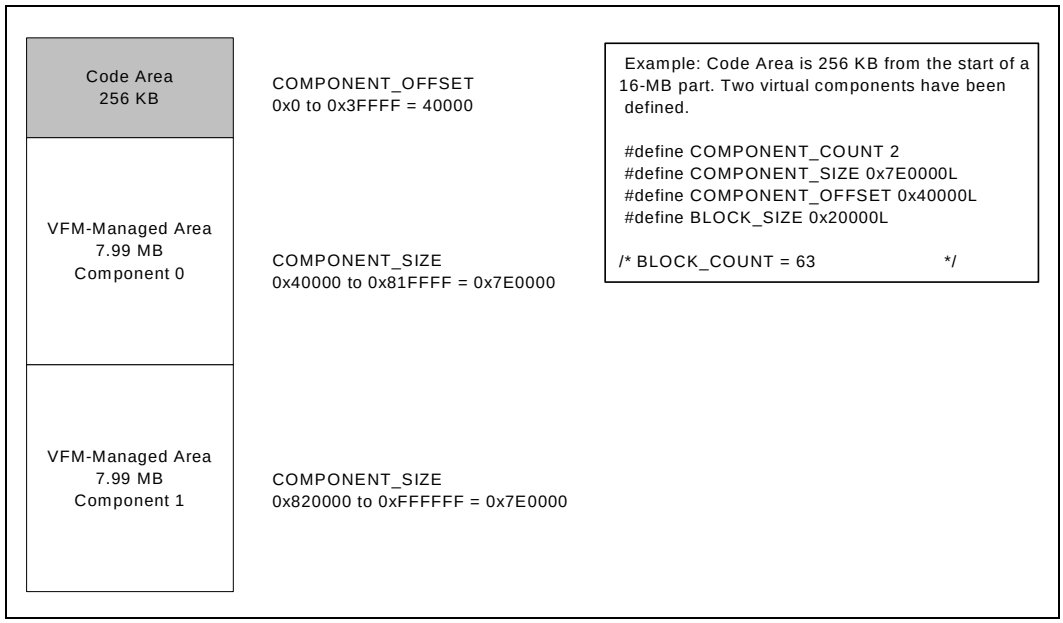
Advantages: This configuration allows two code segments to be stored separately in the device.

Disadvantages: Storing the code segments separately may limit upgradability. Storing them in a contiguous space could increase optimized use of the available space.

Multiple Virtual Components

Figure 5-7 illustrates use of multiple virtual components. Note that virtual components must be the same size.

Figure 5-7. Multiple Virtual Components



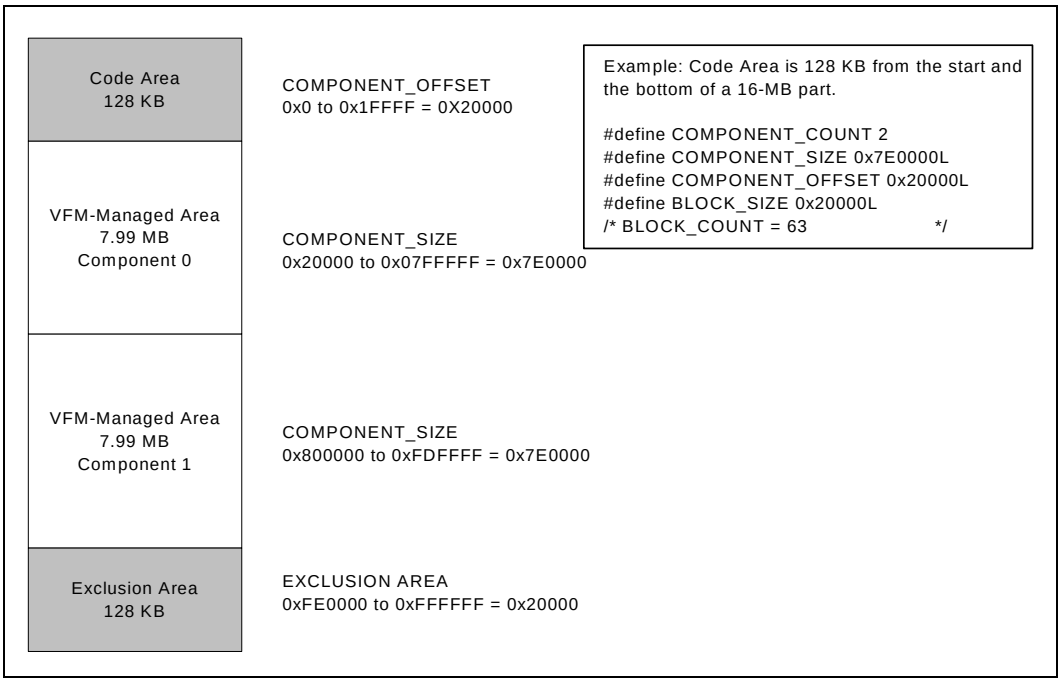
Advantages: Using multiple virtual components provides separate areas for different types of data.

Disadvantages: Using multiple virtual components in one device increases the number of spare blocks required for reclaim operations.

Left-Over Erase Blocks

Figure 5-8, “Left-Over Erase Block” on page 5-12 illustrates the unused block resulting from the specific case of virtual component definition. Since virtual blocks must be equal in size to each other, left-over blocks sometimes result. In this example, the exclusion area uses one block, leaving 15 blocks that could be used by virtual components. The two virtual components will consume seven blocks each, leaving one block left-over. To avoid wasting the extra block, add it to the exclusion area.

Figure 5-8. Left-Over Erase Block



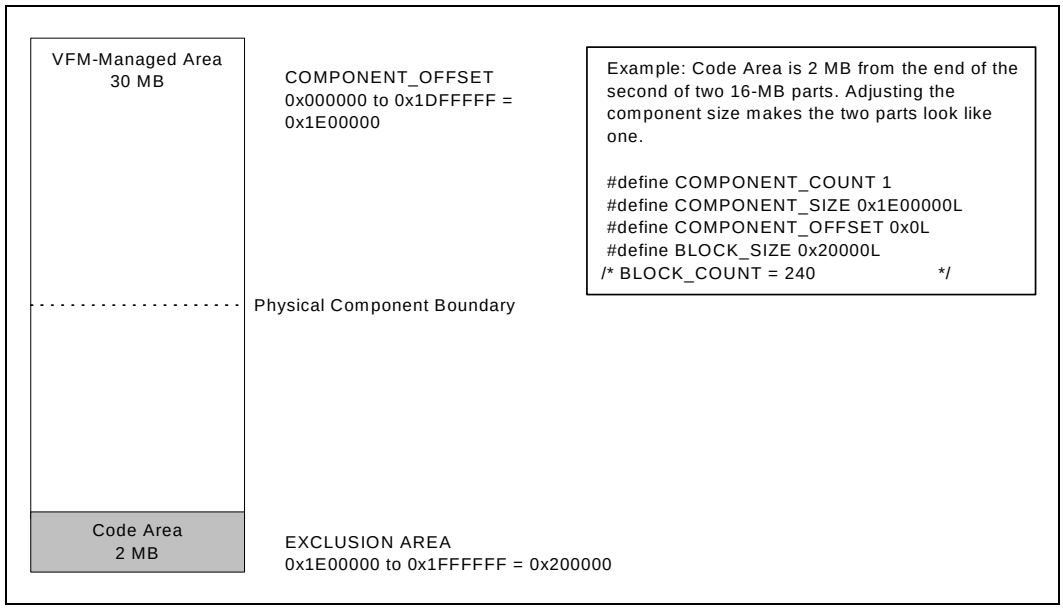
Advantages: This configuration allows multiple virtual components and use of exclusion space which is desirable in many situations. Code and data may be used in the same physical device.

Disadvantages: Because of the combination of device size and exclusion area size, 128 KB of space in the flash device is being unused. A better solution might be to add the 128 KB to the exclusion area so it may be used for future software upgrades.

Large Virtual Component Spanning Physical Devices

Figure 5-9 shows a virtual component that spans multiple physical devices. Note that the COMPONENT_COUNT variable is set to one, reflecting the single virtual component. Adjusting the component size makes the two parts look like one.

Figure 5-9. Large Virtual Component Spans Physical Devices



Advantages: Using a virtual component to span physical boundaries may reduce the number of spare blocks needed in the system. High-level code, including the application, does not need to comprehend the multiple components.

Disadvantages: The large virtual component may contribute to a large VSB size, which may be undesirable in certain systems. Since VSB size is dictated by the size of the component, using multiple virtual components would reduce the VSB size. For more information on VSB calculations, see Section 5.6, “Media Calculations” on page 5-30. There are also potential performance concerns using huge virtual components.

5.3.5.3 Complete Partitioning Example

When partitioning the flash devices, the recommended flow is to start from a high-level of abstraction and work your way down. First, identify your needs in terms of total flash memory size. This is the high level of abstraction. Once this is complete, you can actually choose the flash chips with which you’ll work.

Definition of Application

This example configures the flash for a network printer application. We have 2 MB of code to store, as well as two print images to support simultaneously. These should be stored separately and will require 2 MB per page of the image. Furthermore, we wish to do error and status logging during printing of each image. Logging requires 256-KB space for each image. It is desirable to store logging information for multiple images if sufficient flash space is available.

Summary of Flash Space Requirements

In summary, we need the following areas in flash:

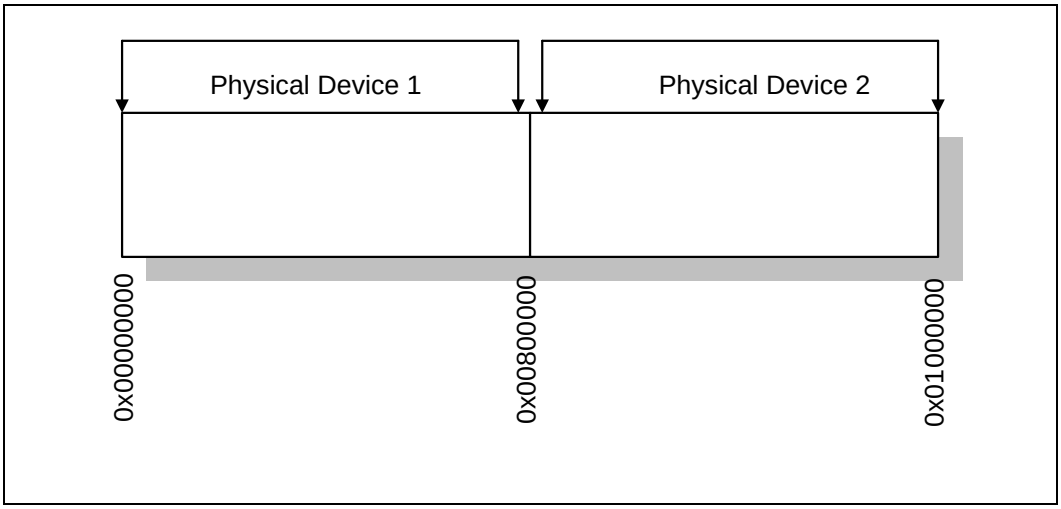
CODE: 2 MB
DATA:
(a) 256 KB logging area per image
(b) 2 MB per page for image 1
(c) 2 MB per page for image 2

TOTAL: 6 MB + 512 KB

Choosing the Flash Devices

In order to handle this code space and the three data areas, we will use an exclusion area and three virtual components. We have a variety of options for actual flash devices. At a minimum, we would need 6 MB + 512 KB in flash. A close match would be one Intel® StrataFlash™ memory chip (8-MB density). However, in order to provide space for multiple pages for each image and sufficient logging area, we will use a total of 16-MB flash, using two Intel StrataFlash memory chips. Each has a x16 interface.

Figure 5-10. Flash Device Configuration



Configuring Virtual Components and Exclusion Area

The next question is how to set up the virtual components and exclusion area. First let's look at the exclusion area size and see how much space is available for the virtual components.

Exclusion area: 2 MB (sixteen 128-KB erase blocks)
Remaining space: 16 MB – 2 MB = 14-MB (112 128-KB erase blocks)

Since we have three distinct types of data that should be accessed separately by the system, we would like to have three virtual components. As we know, all virtual components must be the same size, so the space to be used must be divisible by three. Since 112 is not divisible by 3, we must make an adjustment. We could use 111 erase blocks and divide them into three 37-block virtual components. In order to utilize the flash fully, we will add the remaining one erase block to the exclusion area, giving ourselves space for future feature additions.

Following is the explanation of this calculation:

Number of desired virtual components: 3

Size of each virtual component: (Remaining number of erase blocks/number of virtual components) (112 erase blocks/3 virtual components) = 37.33 erase blocks

Since virtual components must be a whole number of erase blocks, we will round down to 37 erase blocks per virtual component.

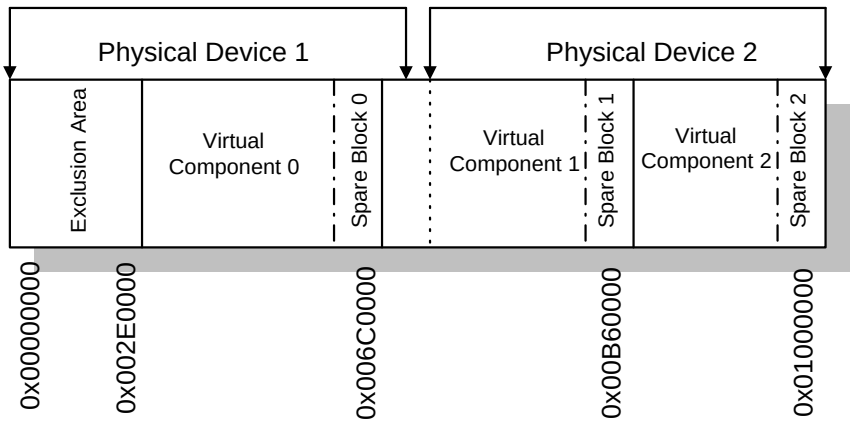
New size of each virtual component: 37 erase blocks
Total space used by virtual components: 37 erase blocks * 3 = 111 erase blocks
Total unused erase blocks: 112 erase blocks -111 erase blocks = 1 erase block

To use the remaining erase blocks, add them to the exclusion area:

Exclusion area: 16 erase blocks + 1 erase block = 17 erase blocks = 2176 KB
Exclusion area (hex): 2176 KB = 0x2E0000

Figure 5-11 gives an illustration of our example.

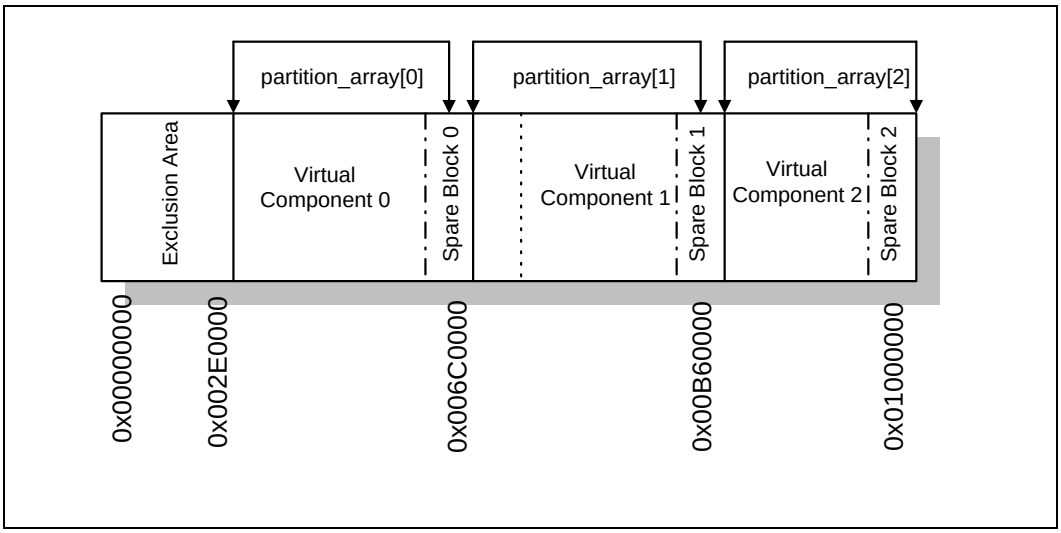
Figure 5-11. Virtual Component Example



NOTE: The spare block is not fixed in one location. Rather, it moves to other locations as a result of the reclaim process. For more information on this topic, see Section 5.9, "Sector Allocation and Wear Leveling" on page 5-38.

The next step is partitioning the flash devices. Follow the procedure described in Section 5.3.2, "Example Media Partitioning" on page 5-4. Keep in mind that the term "component" now refers to "virtual component." In our case, we will simply assign a partition for each virtual component. In some situations, it may be appropriate to define a partition spanning multiple virtual components. This mechanism may be used if different-sized partitions are desired. For example, the user could make one partition including the first virtual component, and another partition including two more virtual components.

Figure 5-12. Partition Illustration



The final level of abstraction is actually specifying our requirements in the appropriate header files. This configuration is simple to specify using the two header files *flashdef.h* and *vsbext.h*. First let's look at the virtual component specification in file *flashdef.h.* Following are the calculations of the required variables.

Component Count: Total number of virtual components used (does not include exclusion area(s))
= 3

Component Size: Total flash memory used – exclusion areas
0x1000000 – 0x220000 = 0xDE0000

Component Offset:

a: if code at bottom: bottom exclusion area size = 0x220000
b: if code at top: 0x00

Block Size: Dependent on flash device used. For Intel StrataFlash memories, the block size is 128 KB.

Block Count: (total – exclusion area blocks)/# virtual components
(128 blocks – 17 blocks)/3 virtual components = 37 erase blocks

The actual definition is done in the file *flashdef.h*, as shown below. The convention used is that the "bottom" of the device is that which starts at address 0.

Listing 5-3. Low-Level Virtual Component, Exclusion Area Definition

```
#define     COMPONENT_COUNT     3           /* Number of flash components present */
#define     COMPONENT_SIZE      0xDE0000    /* Total flash - exclusion areas */
#define     COMPONENT_OFFSET    0x220000    /* end of the exclusion area if placed in the bottom */
#define     BLOCK_SIZE          0x20000     /* media specific */
#define     BLOCK_COUNT         111         /* total - exclusion area blocks */
```

Next, let's define the required information for the partitions. The file *vsbext.h* is updated as follows. Note that the exclusion area is completely ignored in the partition definition. If multiple exclusion areas were made, they would both be ignored in this section. Exclusion areas are relevant only when defining virtual components.

Listing 5-4. FSD Partition Preprocessor Definition

```
#define     MAX_PARTITIONS      3           /* user defined */
#define     BLOCK_SIZE          0x20000     /* media specific */

#define     PARTITION_BEGIN_1   0           /* user defined */
#define     PARTITION_END_1     0           /* Partition 0 */

#define     PARTITION_BEGIN_2   1           /* user defined */
#define     PARTITION_END_2     1           /* Partition 1 */

#define     PARTITION_BEGIN_3   2           /* user defined */
#define     PARTITION_END_3     2           /* Partition 2 */
```

Interactions Between Configuration and Partitioning

Note that some decisions made during configuration may have other effects to consider. For example, using a large virtual component provides a variety of benefits, particularly in minimizing the number of spare blocks required in the system. However, it also contributes to large VSB size, which may not be desirable in some situations. See Section 5.6, "Media Calculations" on page 5-30, for more information on VSB size.

5.4 Required Files

Table 5-1, "File Listing" on page 5-18 gives an overview of which files in the SRC\VFS and SRC\VSBLIB directories are necessary for your implementation. Each file is described more completely below. Files existing in the SRC\LOWLVL directory are the low level files necessary for an example 82365SL PCIC implementation on an MS-DOS PC platform. Your implementation will need to replace the functionality in the low level as indicated in Section 5.11, "Replacing Low-level Driver Interface Functions LOWLVL, PCIC" on page 5-42.

Table 5-1. File Listing (Sheet 1 of 2)

Path	Filename	Necessary?
\PRJ	VFM.PRJ	NO
	VFM_BC5.IDE	NO
\SRC\TSI	TSI.C	NO
	TSILIB.C	NO
	EXEC.C	NO
	TRANS.C	NO
	SCRIPT.C	NO
	GLOBALS.H	NO
	TSI.H	NO
	TIMER.ASM	NO
	TIMER.H	NO
\SRC\VFS	VFM-FS.C	YES
	VFM-SUP.C	YES - adapt to platform and media type
	VSBINT.H	YES
	VSBEXT.H	YES - adapt to platform and media type
\SRC\VSBLIB	CALCADDR.C	YES
	SYSCLNUP.C	YES
	DISCVSB.C	YES
	FINDVSB.C	YES
	FMATVSB.C	YES
	INITVSB.C	YES
	MEMSTAT.C	YES
	MOVEVSB.C	YES
	READVSB.C	YES
	RECLAIM.C	YES
	SCANVAT.C	YES
	WRITESEC.C	YES
	VSBDATA.C	YES
	WRITEVAT.C	YES
	DATATYPE.H	YES - adapt to platform and media type
	FLASHDEF.H	YES - adapt to platform and media type
	VSB.H	YES - adapt to platform and media type
	VSBDATA.H	YES - adapt to platform and media type
	VSBPROTO.H	YES - adapt to platform and media type
\SRC\LOWLVL	LOWLV.C	YES - adapt to platform and media type
	ARCH.C	YES - adapt to platform and media type
	LIBRARY.C	YES

Table 5-1. File Listing (Sheet 2 of 2)

Path	Filename	Necessary?
	LIBRARY.H	YES
	TABLE.C	YES
	LOWLVL.H	YES - adapt to platform and media type
	PCIC.C	YES – for PC Card applications

5.4.1 VFM.PRJ

Borland C++ (3.1) project file for the VFM executable (VFM.EXE). Project not required for VFM.

5.4.2 VFM_BC5.IDE

Integrated Development Environment “project” file for Borland C 5.X.

5.4.3 TSI.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 System file commands into VFM file commands. This *test_script* interface was developed to verify VFM functionality; for use on an MS-DOS PC.

5.4.4 VFM-FS.C

This file contains all of the file system interface functions. This file does not require changes. VFM requires this file to operate.

5.4.5 VFM-SUP.C

This file contains support routines for the file system interface functions. The array to handle media partitioning may require modification for your application requirements. Refer to the *partition_array* variable in the section labeled *Global Variables* at the top of the file. Making changes to the elements in the array will determine how the partitions are laid out on your flash media. Adjust the elements accordingly to correspond to static definitions in the file (*vsbext.h*), listed below. No other changes are required for this file. VFM requires this file to operate.

5.4.6 VSBINT.H

This file contains VFM control structure definitions and function prototypes used internally by the file system driver. This file does not need modification. VFM requires this file to operate.

5.4.7 VSBEXT.H (Contains User-Defined Settings)

This file contains VFM media definitions, control structures, command structures and function prototypes. Both external modules and internal routines require the definitions within this file. Refer to the contents of this file to obtain structure formats needed by external routines to interface with the filing system. Static definitions for media partitioning and extended header assignments are user definable. Adjust those preprocessor definitions accordingly for your application requirements. Refer to the section labeled *Partition Options and Extended Header Options*, at the top of this file. The sections labeled "user definable" are the only areas requiring modification. VFM requires this file to operate.

Note: The following source files are required for either VFM or VSB implementations. The definition files (.h) require modifications to adapt to your application. However, the remaining file extensions (.c) should not require changes.

5.4.8 CALCADDR.C

Calculates the physical flash address.

5.4.9 SYSCLNUP.C

SystemCleanup process used for *PowerOffRecovery. SystemCleanup* validates each VSB against the File Entry Table chain to identify garbage sectors and discards all the garbage sectors. However, writes in these sectors can be done only after reclamation of these sectors.

5.4.10 DISCVSB.C

Marks the Specified VSB as Discarded.

5.4.11 FINDVSB.C

Searches for a specified VSB, beginning at the last location searched.

5.4.12 FMATVSB.C

Erases all blocks in a specified component.

5.4.13 INITVSB.C

Scans all blocks in a specified component, verifies the block attribute and counts free VSBs.

5.4.14 MEMSTAT.C

Analyzes all VSB Allocation Tables in a component and calculates the total number of free, dirty and valid VSBs.

5.4.15 MOVEVSB.C

Moves a sector of data from a source address to a destination address.

5.4.16 READVSB.C

Reads a VSB from flash.

5.4.17 RECLAIM.C

Reclaims discarded VSBs in a specified component.

5.4.18 SCANVAT.C

Scans a VSB Allocation Table for valid entries. Tracks and updates the global mediastatistics with the total number of free, dirty, and clean VSBs in a component.

5.4.19 WRITESEC.C

Searches for the next free VSB, performs reclamation when required and sets up the current pointers.

Writes data to a VSB, previously set up.

5.4.20 VSBDATA.C

Contains the static data declarations for the VSB layer.

5.4.21 WRITEVAT.C

Writes to a specified VSB Allocation Table.

5.4.22 DATATYPE.H (Contains User-Defined Settings)

Contains definitions for data types and error codes. You may need to modify the data type definitions to adapt to your platform and/or compiler requirements. To simplify integration, FSD reports generic errors back to the application. You may modify for more robust error reporting.

5.4.23 FLASHDEF.H (Contains User-Defined Settings)

Contains definitions of the media type and size. You may need to modify the definitions to adapt to the media type used by your application. Refer to those listed as user-defined.

5.4.24 VSB.H (Contains User-Defined Settings)

Contains manifest constants for the VSB layer. The user may set the VSB size, which must comply with certain media restrictions; refer to Section 5.6, “Media Calculations” on page 5-30. Aside from the sector size, the remaining settings do not need to be changed. Modification of the define, VSB_SIZE, will change the sector size throughout VFM.

5.4.25 VSBDATA.H

Contains external data declarations.

5.4.26 VSBPROTO.H

Contains function prototypes for the VSB software layer.

5.4.27 LOWLVL.C

Contains low level media support; flash control for compatibility check, initialization, read, write and erase. The flash control algorithms are required for VFM or VSB to operate. You may need to adapt these routines to suit the type of media used by your application.

5.4.28 ARCH.C

This file contains routines specific to hardware initialization, memory windowing, and power down. Modify this file to suit your low level interface.

5.4.29 LIBRARY.C

Provides memory copy and related routines for the low level software layer. May be used instead of library functions on a variety of platforms. Should not be modified by the user.

5.4.30 LIBRARY.H

Contains header files for file *library.c*. Should not be modified by the user.

5.4.31 TABLE.C

This file is used to define the flash technologies you wish to support with this code. While all Intel flash devices are included, others may be added by adding an entry to the flash technology table and providing the appropriate low level routines. This file is used to support both CFI-compliant and non-CFI-compliant devices.

5.4.32 LOWLVL.H

Contains function prototypes and structure definitions for the low level software layer. Modify accordingly to suit your low level interface.

5.4.33 PCIC.C

Contains PCIC socket control to handle power and memory windowing. The PC socket controller algorithms are not required by VFM or VSB. However, you may need to adapt these routines to accommodate your socket controller, or eliminate them altogether for RFA implementations.

5.5 Modifying Source and Header Files (Compile-Time Options)

This section describes compile-time options, other defines, and other values you need to set appropriately for your system. If you have changed the flash in a current design, be sure to update both files *datatype.h* and *flashdef.h* appropriately. The source and header file modifications set flash device characteristics and masks that define access to the flash device. Note that even if your system is using a linear address configuration, the window masks must be set in file *datatype.h*.

5.5.1 VFM-SUP.C

The array structure described below should be configured to accommodate the number of partitions required by your application.

The array elements are structures, defined by PARTITION_INFO. The partition structure describes the amount of flash to allocate for each partition. Shown below are three partitions. Each contains three values which describe the partition; the erase-block size of the media, the starting component and the ending component comprising the partition. Simply removing a PARTITION_INFO element from this array will eliminate the corresponding partition. When you configure your setup, the partition structure elements must correspond with the preprocessor definitions set up in the header file below (*vsbext.h*). Also, the MAX_PARTITION setting must correspond to the total number of PARTITION_INFO elements in the array.

```
PARTITION_INFO partition_array[ ] =

{

    BLOCK_SIZE,  PARTITION_BEGIN_1,  PARTITION_END_1},    /*partition 0*/

/*{ BLOCK_SIZE,  PARTITION_BEGIN_2,  PARTITION_END_2}, */ /*partition 1*/

/*{ BLOCK_SIZE,  PARTITION_BEGIN_3,  PARTITION_END_3}  */ /*partition 2*/

};
```

5.5.2 VSBEXT.H

#define EXTENDED_HEADER FALSE

To enable the extended header option, set this value to TRUE, FALSE to clear it. The extended header provides a mechanism to store additional file information aside from the file data, like more file attributes. The minimum size for an extended header may be a byte. The maximum size supported by VFM may be up to a full table's worth of entries, many sectors in length. The media type and sector size determine the capacity held in a table; refer to Section 5.6, "Media

Calculations" on page 5-30 for calculations. When set, it in itself enables and configures the default to one sector in length. Together with other preprocessor defines, you may further define the scope and size of this extension for your application. The default setting equals FALSE.

#define EXT_1 FALSE

Set this define to TRUE for an extended header which supports an MS-DOS-like file name. Clear this option by selecting FALSE. Together, with the EXTENDED_HEADER enabled, this provides a mechanism for MS-DOS-like file names to associate with VFM formatted files. The default setting equals FALSE.

#define MAX_PARTITIONS 3

This define establishes the maximum number of partitions configurable and represents the entire flash media in a system. Adjust this according to your application. Multiple partitions are useful if separating file types. Partitions may be treated as a rudimentary sub-directory. Shown in this chapter as an example, the value equals three. The default setting equals one.

#define MAX_FILES_OPEN 5

This define determines the maximum number of files that may be opened concurrently per partition, in read, write or edit modes. 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, like forgetting to close files. The default setting equals five.

#define PARTITION_BEGIN_1 0

This define establishes the first partition and its starting component boundary. The default setting should be zero, the first component in the media. It may, however, be set to any valid logical component number in the media. The partition dimensions are determined by pairing this define with PARTITION_END_1, explained below. Do not overlap logical component numbers between partitions. The logical component numbers must be ordered sequentially in their respective partitions. The default setting equals zero.

#define PARTITION_END_1 0

This define determines the ending component boundary for the first partition. It gets paired with PARTITION_BEGIN_1 to obtain the partition size. Do not overlap logical component numbers between partitions. The logical component numbers must be ordered sequentially in their respective partitions. The default setting equals zero.

#define PARTITION_BEGIN_2 1

This define establishes the second partition and its starting component boundary. The setting should be the next logical component number, following the previous partition. It may, however, be set to any valid logical component number in the media. The partition dimensions are determined by pairing this define with PARTITION_END_2, explained below. Do not overlap logical component numbers between partitions. The logical component numbers must be ordered sequentially in their respective partitions. Shown in this chapter as an example, the value equals one. The default setting omits a second partition.

#define PARTITION_END_2 1

This define determines the ending component boundary for the second partition. It gets paired with PARTITION_BEGIN_2 to obtain the partition size. Do not overlap logical component numbers between partitions. The logical component numbers must be ordered sequentially in their respective partitions. Shown in this chapter as an example, the value equals one. The default setting omits a second partition.

#define PARTITION_BEGIN_3 2

This define establishes the third partition and its starting component boundary. The setting should be the next logical component number, following the previous partition. It may, however, be set to any valid logical component number in the media. The partition dimensions are determined by pairing this define with PARTITION_END_3, shown below. Do not overlap logical component numbers between partitions. The logical component numbers must be ordered sequentially in their respective partitions. Shown in this chapter as an example, the value equals two. The default setting omits a third partition.

#define PARTITION_END_3 3

This define determines the ending component boundary for the third partition. It gets paired with PARTITION_BEGIN_3 to obtain the partition size. The setting should represent the last logical component number in the media. However, it may be set to any valid, logical component number in the media. Do not overlap logical component numbers between partitions. The logical component numbers must be ordered sequentially in their respective partitions. Shown in this chapter as an example, the value equals three. The default setting omits a third partition.

#define DO_EDIT TRUE

This define includes or excludes the Edit source code during compile-time. Setting this define to TRUE enables the code for Editing data; Append, Insert, Replace or Delete bytes. Setting this to FALSE disables all of the Edit options and reduces the code size. Supported at the FSD interface via FSD_Edit. The default setting equals TRUE.

#define DO_APPEND FALSE

This define includes or excludes the Edit-Append source code during compile-time. Setting this define to TRUE enables the code which supports appending bytes. Setting this to FALSE disables the Append option and reduces the code size. Supported at the FSD interface via FSD_Edit. The default setting equals FALSE.

#define DO_INSERT FALSE

This define includes or excludes the Edit-Insert source code during compile-time. Setting this define to TRUE enables the code which supports inserting bytes. Setting this to FALSE disables the Insert option and reduces the code size. Supported at the FSD interface via FSD_Edit. The default setting equals FALSE.

#define DO_REPLACE TRUE

This define includes or excludes the Edit-Replace source code during compile-time. Setting this define to TRUE enables the code which supports replacing bytes. Setting this to FALSE disables the Replace option and reduces the code size. Supported at the FSD interface via FSD_Edit. The default setting equals TRUE.

#define DO_DELETE TRUE

This define includes or excludes the Edit-Delete source code during compile-time. Setting this define to TRUE enables the code which supports deleting bytes. Setting this to FALSE disables the Delete option and reduces the code size. Supported at the FSD interface via FSD_Edit. The default setting equals TRUE.

#define DO_SPACE TRUE

This define includes or excludes the space reporting feature during compile-time. Setting this define to TRUE enables the code which reports free space statistics. Setting this to FALSE disables the space reporting feature and reduces the code size. Supported at the FSD interface via FSD_Special. The default setting equals TRUE.

#define DO_MAN_RECLAIM FALSE

This define includes or excludes the Manual Reclaim feature in the compiled executable. Setting this define to TRUE enables the user do reclaim at any time. Setting this to FALSE would not let the user do reclaim. If this option is set FALSE , automatic reclaim has to be enabled by the user during initialization. Supported at the FSD interface via FSD_Special. The default setting equals FALSE.

#define DO_FITCLEANUP TRUE

This define includes or excludes the File Information Table Cleanup feature during compile-time. Setting this define to TRUE enables the code that provides the table cleanup feature, which allows for defragging and collapsing the file FIT chain. Setting this to FALSE disables the table cleanup feature and reduces the code size. Supported at the FSD interface via FSD_Special calls from your code. The default setting equals TRUE.

#define DO_FETCLEANUP TRUE

This define includes or excludes the File Entry Table Cleanup feature during compile-time. Setting this define to TRUE enables the code that provides the table cleanup, which allows for defragging and collapsing the file FET chain. Setting this to FALSE disables the table cleanup feature and reduces the code size. Supported at the FSD interface via FSD_Special calls from your code. The default setting equals TRUE. The “AUTO_FETCLEANUP” option is a releated option.

#define AUTO_FETCLEANUP TRUE

This define establishes **automatic** File Entry Table cleanup. Setting this define to TRUE enables the code which provides File Entry Table cleanup after every file deletion. Setting this to FALSE disables the automatic table cleanup. Supported at the FSD interface via FSD_Delete. The default setting equals TRUE. Note: this function is available only when the DO_FETCLEANUP define is TRUE.

#define SUB_READBYTES FALSE

This define enables reads of specified number of bytes rather than reading as a whole sector. Setting this to TRUE enables the code which provides partial sector read capability. Setting this to FALSE disables partial sector reads and reduces the code size. The default setting equals FALSE.

5.5.3 DATATYPE.H (Platform and/or Compiler Specific)

typedef unsigned char BIT

The unsigned char should be replaced by any keywords that create an unsigned 8-bit value. If the compiler allows single bit values, set this value accordingly.

typedef unsigned char BYTE

The unsigned char should be replaced by any keywords that create an unsigned 8-bit value.

typedef char CHAR

The char should be replaced by any keywords that create an 8-bit value.

typedef unsigned short WORD

The unsigned short should be replaced by any keywords that create an unsigned 16-bit value.

typedef unsigned long DWORD

The unsigned long should be replaced by any keywords that create an unsigned 32-bit value.

typedef long LONG

The long should be replaced by any keywords that create a signed 32-bit value.

#define MC_ADDR 0xD0000000

This value represents the address of the desired beginning address for flash. Set this address to the beginning address of the physical flash wherever in your system memory map it is location. MC_ADDR is used to define the beginning address for ALL flash interfaces to RFAs (flash on board), as well as flash memory cards. This address MUST be set properly.

#define MC_WIND_MSK 0x00000FFF

This value represents the size of the flash “window” mask. For example, if the MC_WINDMSK is set to **0x00000FFF**, then the mask is set for 4 KB which is typical for a PCMCIA type flash card memory window. However, MC_WIND_MSK is used to define flash interfaces that comprehend RFAs, as well as memory cards. It must be set to represent the largest addressable window into the flash. If the flash is mapped into the system memory map as a full linear array, then this value must likewise be set to encompass the full flash memory size. For example, if you had a 2-Mbyte linear flash array, then MC_WIND_MSK would need to be set to 0x1FFFFF. Neglecting to set this value to reflect your actual flash interface will cause undesirable operation of VFM/VSB and flash.

#define SEGMENTED TRUE

This option configures VFM to work with systems that use segmented memory accesses. For example, these include many Intel architecture systems. In systems that do not use segmented memory, set SEGMENTED to FALSE. Use of this option is **required** in segmented memory architectures.

To use this option, set SEGMENTED to the appropriate value in file *datatype.h.* The default setting is TRUE.

#define MLCDEVICE TRUE

Set to appropriate value based on the flash components being used. MLC refers to “Multi-Level Cell” technology as that used in the Intel StrataFlash component family of components. This compile time option enables a slightly different style of power-loss recovery code. Leaving this set TRUE for non-MLC components is OK, but setting this to FALSE when using a MLC type device is not recommended as power-loss recovery will not function properly on the MLC device.

5.5.4 FLASHDEF.H

#define COMPONENT_COUNT 1

This value represents the total number of flash components in the media that are available to VFM. This example corresponds to a Value Series 100 card configuration, with paired 28F008 devices. Each of the device-pairs counts as one logical component. Thus, an 8-MB card contains four logical components. If the components were RFA and not paired, this value would be eight. Modify this value appropriately. Shown as an example in this chapter, the value equals four. The default setting equals one.

#define COMPONENT_SIZE 0x200000L

This value represents the flash component size. Consider device-pairs as a single entity. Using a flash card as an example, a 2-Mbyte Value Series 100 is comprised of paired 28F008 x8 components which provides an interleaved x16 data path to the flash (two times 1-Mbyte paired components). Each of the device-pairs, also known as logical component numbers, are two megabytes in size. The example shows the hexadecimal value for two megabytes. If the components were RFA and not paired, this value would be one megabyte, hexadecimal 0x100000L. You will need to modify this value, depending upon the type and size of flash media used.

#define COMPONENT_OFFSET 0x40000L

This value defines the size of the exclusion area, setting the offset into VFM managed area. To use exclusion area to hold code, set this define to reflect the size of the exclusion area used. The offset must be a multiple of the block size, reflecting that exclusion areas must be a whole number of blocks. Since the exclusion area may be placed at the bottom or top of memory, the COMPONENT_OFFSET value will be either the size of the exclusion area or zero, respectively. If using an exclusion area at the top of memory, use COMPONENT_SIZE to reflect the device size, less the exclusion area.

This example corresponds to a two-block exclusion area at the bottom of memory, where each block is 0x20000L in size.

If exclusion area is not being used, set this define to 0x00.

#define BLOCK_SIZE 0x20000L

This value represents the erase-block size of the flash component. It must match the actual, physical flash hardware configuration. This example corresponds to a Value Series 100 card configuration, with paired 28F008 devices. The device-pairs are programmed together and are considered one logical component; thus, shown above an erase-block size of 128 KB, represented in hexadecimal form. If the components were RFA and not paired, this value would be 64 KB, hexadecimal 0x10000L. The default setting equals 0x20000L.

#define BLOCK_COUNT (COMPONENT_SIZE/BLOCK_SIZE)

This value represents the total number of discrete blocks in the component. It is calculated based on the settings of COMPONENT_SIZE and BLOCK_SIZE.

5.5.5 VSB.H

#define RECLAIM_THRESHOLD 5

This value represents the reclaim threshold value. When the number of clean sectors reaches this value and write happens, the autoreclamation happens if the *ReclaimEnable* is set to one. This can be done by the user by passing a non-zero value in the count field of the command control structure during initialization or formatting.

#define VSB_SIZE 512L

This value represents the Virtual Small Block (VSB) size, or sector size used by VFM to store files. Determine this setting based on your application requirements, media type and size. Shown as an example in this chapter, the value equals 512L. A sector size of 512 bytes can be accommodated by the example media type; refer to Section 5.6, “Media Calculations” on page 5-30 for capacity calculations. The typical setting equals 2048L which is typically the default for the source provided.

5.5.6 LOWLVL.C

The **ERASE_BACKGROUND** and **PARTIAL** options should not be used in any systems. They are for Intel internal development only.

#define SMART_SLIDE TRUE

This option will optimize management of a sliding “window” to the flash. In systems that do not have the entire flash device in the memory map, use of this window allows access to any area of flash. Using the SMART_SLIDE option not only improves placement of the window, it caches the window information used in recent accesses so it can be used again. This cache feature improves performance by making some window configuration cycles unnecessary. The cache feature manages window configurations in a least-recently-used (LRU) methodology, so that the most recent configurations are always available.

The SMART_SLIDE option should only be used in windowed architectures, such as most personal computer systems. In these architectures, use of this option is **highly recommended**. Also see the “windowed” compile-time option, which must be used as well.

To use this option, keep SMART_SLIDE set to TRUE in file *lowlvl.h*. The default setting is TRUE.

#define WINDOWED TRUE

This option enables use of a “window” to flash memory. It should be used in systems that do not have the entire flash device in the memory map. For best performance, use the SMART_SLIDE option also for optimal window management. In windowed architectures, use of this option is **required**. To use this option, set windowed to TRUE in file *lowlvl.c*. The default setting is TRUE.

5.5.7 LOWLVL.H

#define FALCON FALSE

This option specifically configures VFM to work with the 28F016SA flash memory. However, since this memory is compatible with 28F008SA, use of this option is not required. Use of this option is **not recommended** and will increase code size.

To use this option, set FALCON to TRUE in file *lowlvl.h*. The default setting is FALSE.

#define SOCKET TRUE

This define establishes socket control for card implementations. Setting this define to TRUE enables compilation of the code which supports socket initialization and power control. Selecting FALSE disables compilation of the code which supports the socket. Refer to Section 5.11, "Replacing Low-level Driver Interface Functions LOWLVL, PCIC" on page 5-42 for details about the low level interface functions supporting socket control. The default setting equals TRUE.

#define SRAM_MEDIA FALSE

This option allows the CFI low level commands to work on SRAM PC cards, rather than flash. With minor low level code modifications, this feature can be used during development in order to test code or experiment when flash media is not available. Use of this option in flash systems is **not recommended** and increases code size.

To use this option with SRAM, add the SRAM_MEDIA define to file *lowlvl.h* and set it to TRUE. The default setting is FALSE.

#define SWAP FALSE

This option allows the VFM to switch between big endian and little endian modes. When using a little endian system, such as most Intel architecture machines, set SWAP to FALSE. If using a big endian system, set SWAP to TRUE. Use of this option is **required** in big endian systems.

If using a big endian system, add the SWAP define to the file *lowlvl.h* and set it to TRUE. The default setting is FALSE.

#define WORD_ACCESS FALSE

This option configures VFM to work with systems that cannot access memory on a byte boundary. When this option is enabled, VFM will "steer" bytes of information to the required position in the word. Correct use of this option is REQUIRED for proper operation.

In systems that can access memory only on a word boundary, insert the WORD_ACCESS define in file *lowlvl.h* and set it to TRUE. The default setting is FALSE.

5.6 Media Calculations

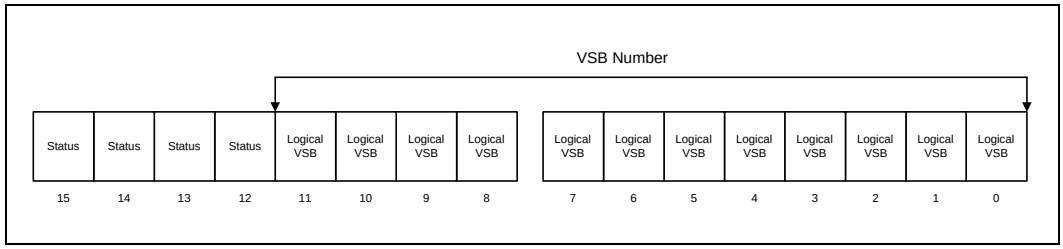
The sector size may be determined by having an understanding of the software layers and their architecture, together with the flash media specifications. Once you have a valid sector size, you may obtain the total number of entries in a File Entry Table and File Information Table.

5.6.1 Sector Size

The user may define the sector size for their VFM implementation, within certain guidelines. The sector size directly correlates to the total number of VSBs which can be managed by the VSB layer. We recommend that you select a sector size based on the guidelines listed below.

Sectors are managed by tables within the VSB layer, called VSB Allocation Tables (VAT). Each erase-block in a component gets configured with a VAT. Every VAT contains entries which handle identification and lookup for logically numbered VSBs. These VAT entries provide a 12-bit field which correspond to a logical VSB number. Thus, the maximum VSB number which can be held in a VAT entry cannot exceed the 12-bit field length. Refer to Figure 2-3, "28F008 Mode with VSB" on page 2-11 and Figure 5-13.

Figure 5-13. Bit Usage for VAT Entry



The maximum VSB number may be calculated by obtaining the component's block count and size, along with selecting a sector size. Both the block count and block size are fixed values, determined by the type of flash component. Because the media definitions are constant, the sector size varies the result of the calculation.

First, obtain the number of VSBs per block by dividing the block size by the sector size. Next, omit the space allocated to each block's VAT by subtracting one from the number of VSBs per block. Then omit the space allocated for the spare block by subtracting one from the block count. Finally, multiply the results of the last two operations together to determine the maximum number of VSBs in a component. See formulas below:

VSB_PER_BLOCK = BLOCK_SIZE / VSB_SIZE

MAXIMUM_VSB = ((VSB_PER_BLOCK - 1) * (BLOCK_COUNT - 1))

Confirm whether the result exceeds the 12-bit size restriction of the VAT entry. If the 12-bit field accommodates the largest VSB number per component, then you have an acceptable sector size configuration. However, if it doesn't fit in the bit range, try increasing the sector size until you have an acceptable configuration. Table 5-2, "Sample Media and Sector Sizes" on page 5-32 shows several example configurations.

In systems where virtual components are used, the VSB calculations are made on each virtual component, not the physical device used. For example, if your system uses a 64-Mb Intel StrataFlash memory device divided into two virtual components, the VSB calculations would be done on the 32-Mb virtual component. This virtual component would have 32 blocks of 128 KB each and a VSB size of 0x400H (1024B). There would be 0x80H VSBs per block, and 3937 maximum VSBs.

If your VSB calculation results in a VSB that is too large, consider using multiple virtual components to reduce the VSB size. Keep in mind, however, that this configuration will require a spare block for each virtual component, rather than just one for the physical component.

Table 5-2. Sample Media and Sector Sizes

Media Type	Block Count	Block Size	Sample VSB Size Sample Sector Size	VSB per Block	Maximum VSB	Valid
28F640J5 RFA	64	0x20000H (128KB)	0x800H (2048B)	0x40H	3969 (0xF81H)	YES
28F320J5 RFA	32	0x20000H (128KB)	0x400H (1024B)	0x80H	3937 (0xF61H)	YES
28F008SA paired (Card)	16	0x20000H (128KB)	0x200H (512B)	0x100H	3825 (0xEF1H)	YES
28F008SA RFA	16	0x10000H (64KB)	0x200H (512B)	0x80H	1905 (0x771H)	YES
28F008SA paired (Card)	16	0x20000H (128KB)	0x400H (1024B)	0x80H	1905 (0x771H)	YES
28F008SA RFA	16	0x10000H (64KB)	0x400H (1024B)	0x40H	945 (0x3B1H)	YES
28F016SA paired (Card)	32	0x20000H (128KB)	0x200H (512B)	0x100H	7905 (0x1EE1H)	NO
28F016SA RFA	32	0x10000H (64KB)	0x200H (512B)	0x80H	3937 (0xF61H)	YES
28F016SA paired (Card)	32	0x20000H (128KB)	0x400H (1024B)	0x80H	3937 (0xF61H)	YES
28F016SA RFA	32	0x10000H (64KB)	0x400H (1024B)	0x40H	1953 (0x7A1H)	YES
28F400BX RFA	4	0x20000H (128KB)	0x200H (512B)	0x100H	765 (0x2FDH)	YES

Note: We recommend you calculate dimensions based on theoretical capacity to protect the integrity of your system. However, the maximum VSB number may never be required for use, depending upon the data model. Most likely, existing data will require updates. When data gets rewritten, logical VSB numbers are reused, reducing the maximum number that gets assigned. All of the “free” VSBs get allocated for use prior to ever reaching the maximum number possible. For example, if the data in logical VSB 12 were to be edited, the physical location containing logical VSB 12 would be marked as “discarded” and the next free physical VSB would assume the logical ID of VSB 12. Due to the reuse of the logical VSB number 12, the maximum count may never be achieved. Due to the variables affecting the frequency of rewrites and reclamation, the “actual” maximum number may be difficult to estimate.

5.6.2 Table Calculations

The control tables for VFM are user-definable, based on the selected sector size. The File Entry Table (FET) and File Information Table (FIT) are the primary control structures used to store files. The total number of entries for each of the tables may be calculated by using the fixed-length structure fields used by these tables. Both the FET and FIT use the following formula to calculate the total number of entries in its table.

$$\frac{\text{(VSB SIZE - FILE LINK SIZE)}}{\text{TABLE ENTRY SIZE}}$$

Each of the FET, FIT and File Link entries is fixed in length. There are 17 bytes in a FET entry, five bytes in a FIT entry and six bytes in a Link entry. The structures are shown below.

```
struct file_entry
{
4       DWORD   type;
4       DWORD   id;
2       WORD    attributes;
4       DWORD   extended_header_size;
1       BYTE    FIT_component;
2       WORD    FIT_VSB;
}FILE_ENTRY;
```

17 Bytes

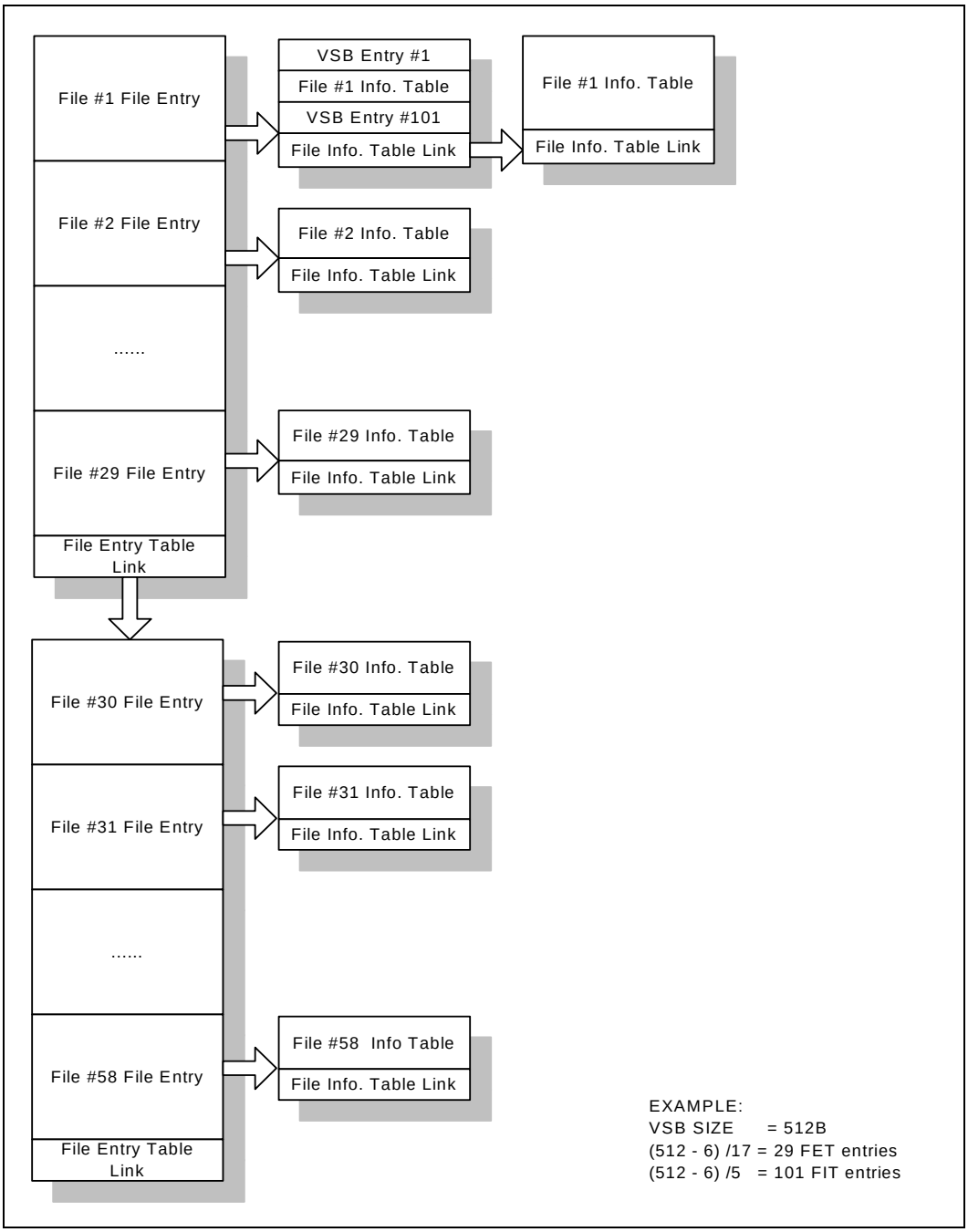
```
struct file _link
{
1       BYTE    next_component;
2       WORD    next_VSB;
1       BYTE    previous_component;
2       WORD    previous_VSB;
}FILE_LINK;
```

6 Bytes

	struct file _info_table		
	{		
1		BYTE	component_number;
2		WORD	VSB_number;
2		WORD	num_valid_bytes;
	}**FILE_INFO_TABLE;**		
5 Bytes			

NOTE: The structure sizes may vary slightly depending on the compiler. For example, a compiler can align on a word boundary. Consequently, FILE_INFO_TABLE would take up six bytes instead of five.

Figure 5-14. VFM Architecture



5.6.3 Media Offset Calculations

Some designs may intend to use some offset within the flash media to begin VFM storage, thus allowing the integration of code (boot, not runtime) or other usage of portions of the flash. The Virtual Small Block media manager accommodates this capability through the use of the

COMPONENT_OFFSET define. This define allows the OEM to set an offset into the component that VSB/VFM uses as the start of the VSB/VFM managed area of flash. COMPONENT_OFFSET is found in the FLASHDEF.H file. More details on this capability are discussed in earlier protions of this chapter.

5.7 Ram Usage

Several VFM control structures are maintained in RAM to provide functionality and improve I/O performance. The following descriptions represent a subset of the fields used by FSD. For a complete description of RAM usage by FSD, VSB and low level routines, refer to a compiler-generated map (.MAP) file. There are several significant factors which affect the amount of RAM usage; component type, total components, sector size and the kind of VFM features required for your application.

5.7.1 Partition Array

As previously described, the *partition_array* provides a mechanism to allocate and identify partitions across the entire flash media. The total RAM usage depends upon the maximum number of partitions you allocate. Multiply the maximum number of partitions by eight bytes, the size of the PARTITION_INFO structure to get the RAM usage: ***(MAX_PARTITIONS * 8)***

5.7.2 Open File Array

VFM tracks open files in an array called *file_array.* The size depends on the total number of partitions and the maximum files, which may be opened concurrently in a partition. Each element of the array contains up-to-date file information, which are structures 30 bytes long, called FILE_INFO. To obtain the total RAM, multiply this value by the total files per partition and total partitions:

(MAX_PARTITIONS * MAX_FILES_OPEN * 30)

5.7.3 Last File Found

VFM maintains a file information structure per partition, representing the last file found in a search. A fixed-length field of 30 bytes per partition, gets allocated from RAM:

(MAX_PARTITIONS * 30)

5.7.4 Last Free Component and VSB

VFM tracks the next free component and VSB for sector allocation. Each fixed-length field,

last_free_component and ***last_free_VSB*** are two bytes long, per partition:

(MAX_PARTITIONS * 2) (per field)

5.7.5 Maximum FET and FIT Entries

The maximum number of entries in a FET and FIT get stored in fixed-length fields. Each is two bytes long, called respectively *max_FET_entries* and *max_FIT_entries* (four bytes total).

5.7.6 I/O & Other Buffers

VFM uses several buffers to manipulate byte transfers to and from flash. These buffers are byte arrays, each a sector in size. The amount of RAM allocated strictly depends upon the sector size. A full-featured version of VFM uses these buffers for data and control structures, in a manner that handles robust power-off recovery. If your application doesn't require all the features, such as edit functionality, you may re-evaluate the need for some of these buffers. The byte arrays are called; *FET_buffer [VSB_SIZE], FIT_buffer [VSB_SIZE], buffer [VSB_SIZE], buffer2 [VSB_SIZE], buffer3[VSB_SIZE].*

5.7.7 Valid Sector Table

An array-of-bytes tracks VSB usage, whether sectors are allocated or remain free for usage. Every logical VSB number corresponds to a bit in this array. A bit gets twiddled to correspond to a free or used VSB number. For example, bit 0 corresponds to VSB 1, bit 1 corresponds to VSB2 and so on. The count spans components, to accommodate multiple device media. Thus, the maximum size of this array must accommodate the total number of VSBs possible for the entire media. The total RAM usage depends upon the total number of VSBs, which directly correlates to the total number of bits. This may be calculated by using the following formula, where BIT_LEN represents 8 bits, or a byte: **(*COMPONENT_COUNT*) * ((*MAXIMUM_VSB / BIT_LEN*)+1).**

There are three variables used by Valid Table, each two bytes long, called *valid, set_VSB_bit, and valid_location.*

5.8 Automatic And Manual Cleanup

VFM incorporates robust cleanup algorithms which recover lost sectors resulting from fragmentation, dangling references and garbage. Also notable, VFM considers power-recovery in its cleanup algorithms during initialization and makes every effort to maintain the integrity of existing data. Refer to Chapter 3 for a discussion of the interfaces for FSD_Special routines; SP_FETCleanup and SP_Reclaim. These routines provide a combination of proactive and manual cleanup processes, optionally set by the user. User-selectable compiler defines are described in earlier sections, which automate some of these algorithms.

5.8.1 FET_CLEANUP and AUTO_FETCLEANUP

The multi-purpose FET_CLEANUP routine may be called via the Special interface, SP_FETCleanup. First, it removes dangling references in the File Entry Table chain. Any file entries identified as FILE_DISCARDING are completely deleted and marked FILE_DISCARDED. Then it proceeds with what's sometimes called a "defrag," which compresses and/or collapses the entire File Entry Table chain. The default algorithm determines when to clean up, based on the fragmentation. Some applications may require that the file entries remain compressed at all times. Both approaches check if there are enough available sectors to

accommodate a chain rebuild. From a power-recovery standpoint, the original chain remains intact while the new chain gets built, then finally the top of the new chain replaces the old one.

An automatic table collapse and/or compression may be enabled to run after every file deletion (FSD_Delete). This may be accomplished by enabling the define called **AUTO_FETCLEANUP** to TRUE. The operation is setup to defragment the File Entry Table chain after every file deletion which will collapse and compress only when fragmentation justifies a collapse. Consider that each operation will use one or more free sectors to rebuild the chain, thus generating additional discarded sectors. Evaluate your application requirements versus the increased overhead. This type of increased sector usage may cause more frequent block reclamation. The default setting equals TRUE. Note: AUTO_FETCLEANUP is available only when the DO_FETCLEANUP define is TRUE.

5.8.2 FIT_CLEANUP

Similar to FET_CLEANUP, FIT_CLEANUP enables a callable function that allows the File Information Table for a file to be collapsed. If a file is edited often using the VFM File Edit functions, the FIT table can grow very large because of many edit updates made to the file. A call to FIT_CLEANUP will collapse the FIT chain by deleting (marking as discarded) any VSBs allocated as FIT entries that contain fully obsoleted/deleted FITs in the file's FIT chain, thus recovering the VSBs. The VSBs are left dirty and a reclaim will be required to clean up the VSB to be usable for other data.

5.8.3 Automatic and Manual Reclaim

While the above routines clean up the filing system control structures, it's most important to note that they **do not** recover the sectors which are marked discarded. The reclamation process at the media manager layer recovers discarded sectors. There are three approaches to handling reclamation; automatically via an enabling variable called *ReclaimEnable*, manually or both. Refer to Chapter 3 for more discussion on reclaim. We have provided a monitoring algorithm which allows the user to poll the usage of all VSBs on the media. A current total of clean, dirty and valid sectors gets reported. Also note that the media manager tracks the count of *clean_sectors* to monitor when to trigger an automatic reclamation.

5.8.4 Global MediaStatistics

The media statistics information is kept global. Media Manager tracks the number of clean, dirty and valid sectors and keeps it up-to-date. Note that the VAT Entry is read only once on initialization.

5.9 Sector Allocation and Wear Leveling

There are several factors inherent with VFM that encourage even wear of flash components. Sector allocation, sector size, media size and the application's data model are variables which impact the wear of the device(s).

First, VFM uses a sector allocation algorithm which makes requests for sectors by spanning components, a sort of "ping-pong" affect. By retrieving free sectors in this manner, data gets spread across the media. A multi-sector file may be spread across several components. Then, depending upon the sector size and the frequency of deletions and edits, sectors are discarded across

components. Dirty sectors are cleaned up by the block reclamation process. This method encourages even wear from a sector allocation perspective. Free sectors are retrieved by ping-ponging the logical component number for every logical VSB number, then incrementing the logical VSB number. The example illustrates a four component media, the logical component number and logical VSB number ping-pong as follows: 0-1, 1-1, 2-1, 3-1; 0-2, 1-2, 2-2, 3-2; 0-3, 1-3, 2-3, 3-3, etc.

If, however, you are using a single device, the logical VSB numbers will be retrieved and allocated sequentially within the single component. Still, the frequency of edits and changes to files will influence the usage of the part. The inherent “sectoring” of the media manager causes data to be moved around. If some amount of data representing a portion of a block never changes, and the remaining data in the block changes, the “stagnant” data will still “move around,” due to block reclamation. That is, when some number of sectors are dirty in the sector, the valid data will be relocated to another block during reclaim. There are no adverse affects for those entire blocks of data which do not change. VFM does not require an algorithm to relocate entire blocks of data which have had no activity, nor does it require a cycling algorithm to count block erasures.

5.10 Platform and Compiler Architecture

The existing VFM code may require unique compiler and/or platform settings which support your system’s hardware and software architecture. VFM was coded and tested using Intel-based architecture under MS-DOS. The filing system and media manager are ANSI-compatible, compiled with Borland C++ 5.0. You may substitute your own adaptation for the low level flash control and omit the code packaged with VFM. A menu-driven interface was designed for testing VFM features. It contains MS-DOS-specific libraries which you may want to adapt to suit your particular platform. Various media types and dimensions were tested using a PC Card Reader/Writer.

The MS-DOS carriage return and new line characters are throughout the source; translated in hexadecimal format (0Dh) (0Ah). Some operating systems may require adjustments which omit or change these end-of-line character sequences. You may need to adjust your editor by selecting appropriate margins and tab settings which properly display the source code. Tab characters are omitted and replaced by spaces, which are column aligned every three spaces. The right margin setting extends to column 80.

The filing system, media manager and low level code support 16-bit offset addressing. The modules are compiled and linked together under a small memory model (Borland option **-ms**). Imposed are byte aligned data (Borland **-a-**) in the VFM-FS control structures listed below. To comply with alignment requirements, you should set your compiler to accommodate byte aligned code. Your compiler may provide a special directive designed for this purpose. The VFM control structures are throughout the source code, so you may prefer to set your compiler directives to encompass all the VFM routines. The code was designed in a way to simplify porting by accommodating relative offsets, based on the data structure sizes and their corresponding *typedef*. Most likely, your compiler will treat the *typedef* structures as a single “entity,” providing offsets to each structure “entity” in accordance with their structure size. For example, incrementing a FILE_INFO_TABLE variable will result in an address offset representative of the size of the structure, five bytes in length. VFM uses indirect addressing methods and does not contain any absolute addresses.

However, accessing individual elements contained within these structures must be considered and may be treated differently, depending upon the compiler. If your platform doesn't support a byte-align directive, we recommend you review all the structures and I/O functionality to verify a transparent conversion. For optional structures which are not required by the filing system, like EXT_HEADER, you may omit the structure entirely via a compile time option.

Some architectures may have difficulties attempting to access a word or a long, which falls on odd address boundaries. Such may be the case, given that some structures contain byte-defined elements, followed by words or longs. In this case, we recommend either padding byte-defined elements to even address boundaries and/or reordering the elements in the structures to relocate byte-defines as the last element. Padding the data structures to even offsets will result in aligning member accesses to even offsets. Relocating byte-defines to be last may eliminate the odd addressing of word or long data types. Review the table structures and your defined sector size to understand their boundaries, then determine how to integrate them for your compiler.

For example, the FILE_INFO_TABLE structure contains the byte-defined element *component_number.* A WORD access on an odd address boundary may occur when attempting to access the field which follows it. By adding the byte-defined element called "*pad,*" the structure alignment changes and *VSB_number* now falls on an even offset. This approach can eliminate the platform dependent requirements described above. Keep in mind that increasing the overall size of these control structures will impact the total number contained in their corresponding tables. For example, the FILE_INFO_TABLE structure now represents six bytes. The total number of FILE_INFO_TABLE structures which can be stored in a File Information Table may change, according to the increased size of each structure. Alternatively, simply reordering elements may eliminate the odd addressing problem; refer to the example shown below. Either of these approaches should not impact VFM functionality.

5.10.1 VSBINT.H

```
typedef  struct          file_info_table
{
        BYTE             component_number;
        WORD             VSB_number;
        WORD             num_valid_bytes;
}FILE_INFO_TABLE;

typedef  struct          delete_fit_cmd
{
        WORD             cmd;
        BYTE             FIT_component;
        WORD             FIT_VSB;
}DELETE_FIT_CMD;

typedef  struct          mark_valid_fit_cmd
{
        WORD             cmd;
        BYTE             FIT_component;
        WORD             FIT_VSB;
}MARK_VALID_FIT_CMD;

typedef  struct
{
        WORD             cmd;
        BYTE             FIT_component;
        WORD             FIT_VSB;
        DWORD            size;
}GET_SIZE_CMD;
```

EXAMPLE:

```
typedef  struct          file_info_table
{
        BYTE             component number;
        BYTE             pad;
        WORD             VSB_number;
        WORD             num_valid_bytes;
}FILE_INFO_TABLE;
```

EXAMPLE:

```
typedef  struct          delete_fit_cmd
{
        WORD             cmd;
        WORD             FIT_VSB;
        BYTE             FIT_component;
}DELETE_FIT_CMD;
```

5.10.2 VSBEXT.H

```
typedef  struct          file_entry
{
         DWORD           type;
         DWORD           id;
         WORD            attributes;
         DWORD           extended_header_size;
         BYTE            FIT_component;
         WORD            FIT_VSB;
}FILE_ENTRY;

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

5.11 Replacing Low-level Driver Interface Functions LOWLVL, PCIC

Due to the unique characteristics of the hardware interface to flash created by individual OEMs, the following functions must be supplied by your low-level driver. They must adhere to FSD data structure error handling. VFM accepts a successful return value of zero (0) from the low level layer and labels it ERR_NONE. Only the power control routine returns void. Non-zero values are reported through the filing system with an appropriate descriptive error code. In attempt to simplify integration, the filing system provides generic types of error codes, which describe the failure. All the error codes for the filing system and media manager are implemented via an enumerator list. Both the enumerator list and filing system may be easily modified to provide enhanced error reporting if needed.

The following prototypes for the low level interface may be found in the "*lowlvl.h*" file. If you are implementing an RFA implementation, you will need the following flash control routines; compatibility check, read, write and erase. Additionally, you must consider your approach to handling power control. If you are implementing non-removable cards requiring socket control, you will need flash control, initialization and power routines. If you are implementing removable media, you will need all the mentioned routines, plus card detect and partition mounting. Refer to the interfaces listed below.

We have provided sample source written in ANSI 'C,' in the VFM package. However, you may obtain samples written in C, which you may adapt to your platform and media type.

5.11.1 BYTE Initialization (DWORD memory_address)

Enables a memory window which manages flash I/O via the 82365 PCIC socket; default address starting at 0xD0000H. The current implementation establishes memory windowing and socket control from the FMM layer during the initialization phase. This Initialization routine gets called from the InitializeVSB routine, located in "*initvsb.c*."

5.11.2 Void PowerUpSocket (BYTE Socket, WORD V_{PP})

Apply power to the socket; default socket zero, 12 V V_{PP}. There are several approaches to handling power control. The current implementation powers up the socket at initialization time and power remains applied to the socket. PowerUpSocket gets called from the InitializeVSB routine in the FMM layer, located in the file called "*initvsb.c*." For power sensitive designs, you may prefer to control power from the filing system layer. Upon entry to an I/O operation, power may be applied, then removed upon exiting the operation. Perhaps you may want to control power from the application layer, prior to requesting a file read or write from the filing system. PowerUpSocket, together with PowerDownSocket may be used by any layer to manipulate power control.

5.11.3 Void PowerDownSocket (BYTE Socket) (not implemented in provided reference code)

Remove power to socket; default socket 0. The current implementation **does not** include this routine. To reserve power, battery operated designs may consider power control which reduces power to save resources. See above, *PowerUpSocket*.

5.11.4 BYTE FlashDevCompatCheck (MEDIA_INFO *)

Verifies compatibility with Intel Flash. First makes a CFI query to identify the flash physical component. If no query information is returned, uses a JEDEC table to identify media size, block size, and block count using the legacy method. This function may be used to initialize a media structure to handle mounting different types of Intel Flash media. Query or table lookup on the Miniature Card flash components supplies the structure with the media size, block size and block count. The Media structure is not required for an RFA or non-removable card implementation. Reports an error if components are non-Intel devices.

5.11.5 BYTE FlashDevRead (DWORD Address, DWORD Length, BYTE *Buffer)

Read fills the specified buffer with the number of bytes defined by length, from the device's absolute physical address. The VSB layer determines the physical address by using its logical component and VSB based mechanism.

This function must access the flash through the hardware interface on your system. If your system uses memory windows (e.g., 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. The low level incorporates a sliding window algorithm.

5.11.6 BYTE FlashDevWrite (DWORD Address, DWORD Length, BYTE ∗Buffer)

This function writes length bytes of data from a specified buffer to the destination address within the device. The VSB layer determines the physical address by using its logical component and VSB based mechanism.

This function must access the flash through the hardware interface on your system. If your system uses memory windows (e.g., 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. The low level incorporates a sliding window algorithm.

5.11.7 BYTE FlashDevEraseBlock (DWORD erase_block_address)

In order to reuse the flash media, an erase command must be provided for the filing system driver. This command erases a single flash erase-block beginning at the address specified. The VSB layer determines the physical address by using its logical component and VSB based mechanism.

This function must access the flash through the hardware interface on your system. If your system uses memory windows (e.g., 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. The low level incorporates a sliding window algorithm.

5.11.8 Void FlashDevMount() (not implemented)

If your application requires removable media, a mounting routine provides the means to identify different types of media upon card insertion. The current implementation **does not** include this routine. However, VFM may be adapted to handle this type of mechanism. It determines the presence of a device, the type, size and initializes partitioning. This routine can call FlashDevCompatCheck(), which obtains the media information identifying the unique flash characteristics. This information gets passed to the VFM control structures.

Non-removable media does not require this mechanism, due to its static media definitions.

5.11.9 Void CardDetect (WORD Socket, DWORD Partition) (not implemented)

For removable media, this function will provide a way to indicate when a device has been removed or inserted. An interrupt handler can call FSD. The filing system will then clean up all internal structures which are applicable (open file structures), and the low level function can re-initialize the partitioning structures. The current implementation **does not** include this routine.

5.12 Creation of an O/S Interface or Direct Application Access

The VFM file system driver can be used in two different ways: directly as an application or through an O/S file interface.

5.12.1 Direct Application Access

The VFM file system can be treated as a library of file system functions. Many embedded systems do not have an operating system. These systems should directly call the FSD functions. Using the direct call approach allows the software to have a simple interface to the library.

5.12.2 Creation of an O/S Interface

Many operating systems allow multiple file system drivers to be installed and provide a mechanism to accomplish this. These operating systems then provide a way to perform file reads and writes through the system I/O. If this type of interface is chosen, you must identify what functions must be supported by the Operating System Interface. Create those functions, and use the functions to translate the information the operating system provides to the VFM FSD interface. For example, a partition gets installed into the system I/O tables. When a read to this partition occurs, the system I/O transfers any necessary information to a driver's read interface function. This function should translate the information passed to it into information the VFM FSD understands, then call the read function provided in the FSD. Any post processing would be done after this call completes and control returns to the system I/O interface which transfers control back to the calling application.

Appendix A
Frequently Asked Questions (FAQ)

Frequently Asked Questions (FAQ) A

A.1 Frequently Asked Questions (and Answers)

Q. ***What portion of the VFM code will I need to modify for my application?***

A. Your implementation will need to adapt or replace the functionality in the low level directory. This is the routine that handles the direct manipulation of the flash. You may also need to adapt the higher level file system interface to more easily adapt to your application of particular operating system. You will also need to set a number of VFM compile time options and defines. It will be very helpful to have an understanding of the specs of your particular flash devices that you are using, how they are interfaced in your platform, and details on the particular processor you are using. Your hardware designer will be a helpful resource during the initial porting stages. After that, of course it will be the responsibility of the software to fix most of the hardware problems.

Q. ***How much memory will my system need to store the VSB/VFM code?***

A. The memory required to store the VFM code will vary from system to system, due to differences between the low level driver interface and the operating system interface. The size of the file system will also vary with different compilers. A full implementation of VFM code is usually about 16K-24K on a typical platform.

Q. ***Does VFM support power-loss recovery? How well?***

A. VFM has very robust power-loss recovery. We have put much effort into assuring existing data will not be lost or corrupted if a power-loss event happens, even at the worst possible instant. Extensive testing by Intel and OEMs assures your data is safe. The most that will be lost is the particular new data that is being written to a VSB. Any existing data in or around the sector being updated will be retained. This is handled through a very sophisticated power-loss recovery mechanism that uses redundancy and status bits to allow the code to recover from an unfortunate power-loss event.

Q. ***Does VFM support flash wear leveling?***

A. The virtualization of flash (virtual sectors) and the allocation algorithm that handles component spanning combine to keep data moving in the flash. In our experience, it is nearly impossible to accidentally “stall” data in one spot. It is the nature of VSB such that files are constantly moving about. Whenever a file is updated, its current position is marked as dirty, and the file is re-written elsewhere. When it becomes necessary to reclaim the block containing the old file (as the media fills up), the valid data that is in that block is moved to a spare block, and the old block erased to allow re-use. Files that do not get updated will not move to other blocks within the media unless other files within that block get updated and re-written elsewhere. A cycling utility exists which can help you estimate how long it will take your application to reach a given block erase count:

http://developer.intel.com/design/builder/FLBLDR/SWB/cycling/content.htm#vsb

Q. ***Does VFM support code execution and data storage in the same component?***

A. For the most part, the answer is currently “No.” However, it does depend on your usage model:

Single partition flash CODE+DATA-

VFM, as provided, does not support software "Read-While-Write" (RWW), thus cannot support direct code execution (XIP) from one portion of a single partition flash component while storing data (files, etc.) to another portion of the same part. There are plans to support some level of software managed "RWW" in a future version of VFM.

OEMs also have been successful at adapting VFM to run the all flash routines that write or set status on the chip (such as the low-level routines) from RAM with interrupts disabled to allow writes back to the single flash chip.

Of course if there are multiple chips, VFM can run from one and manage another physical component.

Boot code from flash and O/S and/or VFM in RAM

VFM does allow a portion of the component to be reserved for "other uses" such as direct execute boot code. It is assumed that at the time the rest of the flash component begins to be managed by VFM, that VFM or its low-level will be executing from some other memory such as RAM.

Dual partition flash-

VFM will support running its code from one partition while managing data in the other partition(s).

Q. ***Does VFM support multitasking or multithreaded environments?***

A. Not directly as provided. The issue is that the VFM code modules/routines were not designed to be re-entrant. Most all of the data structures used by VMF are global and shared. If one task was operating in VFM and another task interrupted it and did something else in VFM, upon return most all of the structures will have been changed by the interrupting task. There are a couple of techniques you can use to make VFM more multitasking aware that other OEMs have implemented in VFM. The first, and simplest, is to set up a semaphore flag at the entry point to the VFM API:

A task sets it "busy" on entry and clears it on exit. If another task interrupts and attempts to access VFM, it is blocked by the busy flag and must wait for it to clear before accessing the VFM volume.

The next technique requires more understanding of the VFM operation. Instead of blocking access for any task, the semaphore can be more sophisticated and only block certain operations if certain others are in process. For example, a read is likely OK if another read is in process (as long as the global structures integrity are maintained). It is recommended to block accesses if a cleanup or erase are in process. You may also need to block access if a delete or write is in process, but may be OK if tasks do not share the same files.

The most drastic change is to implement a full context switch between tasks. It is still recommended to block access if a cleanup (and potentially other) functions are in process.

Q. ***My first access to VFM returns an error, what is wrong?***

A. Well, assuming you have all flash component and partition settings correct, the two most common reasons for an error on first access are:

1. You need to add an initialization call to VFM into your system initialization code. VFM requires you call the VFM Init function, FSD_Special subfunction SP_INIT before accessing the VFM partition. This will ID the flash, set up global variables and structures, check the partition and data integrity, and do any 'housekeeping' functions, such as power-loss recovery.

2. You need to FORMAT the array on the very first time boot with VFM to prep the flash for access by VFM. SP_INIT will return an error indicating the array is not formatted. You will need to call FSD_Special subfunction SP_FORMAT to format the flash.

After these, the next most common cause of a first access error is a problem with your settings that characterize the flash. You might want to look at ALL of the flash settings carefully. Drawing a flash usage memory map, and double checking the specs for the particular flash components that you are using, generally helps.

Q. ***I see references to flash card or PC Card settings in the user settings. I'm using an RFA (flash on board); do I need to set those values?***

A. YES. The settings for MC_ADDR, MC_WIND_MASK, etc. are used for whatever style of flash you are using and must be set appropriately.

Q. ***My flash is fully mapped in a linear fashion and does not use a window to access the flash; do I need to set the MC_WIND_MSK?***

A. YES! MC_WIND_MSK must be set to the maximum range of flash in memory as it is used to access flash no matter what style it is.

Q. ***Does VFM support Boot Block architecture components?***

A. Yes, with qualifications. VFM, as provided, only manages blocks that are symmetrical in size. Thus, VFM can be set to manage the boot/parameter blocks, or the symmetrical data blocks in the data area of the component, but typically not the entire component.

If the support has not yet been added to the base code, Boot Block components such as the Intel® Advanced Boot Block (B3) or Intel® Advanced+ Boot Block (C3), or other Boot Block components can easily be added to the code by adding a "technology table" to the VSB low-level. (It was designed for this type of addition) – details available upon request. C3 is supported via Intel Basic Command Set (BCS) but requires additional coding to support the Instant Individual Block Locking that the C3 device uses. The Intel VSB/VFM managed blocks will need to be unlocked by extra code (not provided in Intel VSB/VFM) before Intel VSB/VFM initialization. Note: as previously indicated, IVFM operates only on symmetrical (same size) blocks, so either the "parameter blocks" or the "data" blocks can be managed with IVFM on a Boot Block architecture component, but not both simultaneously unless extensive changes are made to the low-level routines.

Appendix B
Documentation and Technical Support

Documentation and Technical Support B

B.1 Supporting Documentation

Ref / Order Number	Document/Tool
Contact your local Intel Sales Representative	Intel® VFM Developer's Kit
Contact your local Intel Sales Representative	Test Script Interpreter (TSI) User's Guide
Intel Developer's Web Site for Flash	http://developer.intel.com/design/flash/
290667	3 Volt Intel® StrataFlash™ Memory; 28F128J3A, 28F640J3A, 28F320J3A
290606	5 Volt Intel® StrataFlash™ Memory; 28F320J5, 28F640J5 (x8/x16)
290429	5 Volt FlashFile™ Memory; 28F008SA (x8)
290608	3 Volt FlashFile™ Memory; 28F160S3, 28F320S3 (x8/x16)
290645	3 Volt Advanced+ Boot Block Flash Memory; 28F800C3, 28F160C3, 28F320C3, 28F640C3 (x16)
290580	3 Volt Advanced Boot Block Flash Memory; 28F004/400B3, 28F008/800B3, 28F016/160B3, 28F320B3, 28F640B3 (x8/x16)
292204	AP-646 Common Flash Interface (CFI) and Command Sets
297665	AP-686 Flash File System Selection Guide

B.2 Technical Support

Intel provides extensive customer support for all of its products. For technical assistance on flash memory products, please contact Intel's Customer Support (ICS) team at 1-800-628-8686. You may also send electronic mail to the ICS team [e-mail* address: ICS_Flash@intel.com]. The e-mail* must be of the following format (Note: the colon following each keyword is required):

Company: *[Enter your company's name]*

Name: *[Enter your name]*

Product: *[Enter the specific flash product name, e.g. TE28F160B3-T120]*

Question: *[Enter your question]*

Documentation is available on Intel's Flash memory website: (http://www.intel.com/design/flash) page. For specific software questions, send e-mail to the flash software team [e-mail address: flash@inside.intel.com]. Also, the flash software team can be reached at 1-916-356-8922.

Appendix C
Glossary

Glossary C

C.1 Definitions and Conventions

Term	Definition.
BYTE	8-bit value.
CFI	Common Flash Interface. Allows software to “query” a device in order to learn about the device’s physical characteristics. New commands are implemented via the Scalable Command Set (SCS).
deallocated	When a file is marked as deleted, its flash space cannot be reused until a reclaim occurs. This space is called “deallocated” or “dirty” space.
device	A flash component.
DWORD	32-bit value.
endian	The order in which hardware writes individual bytes of a multiple byte field. At this time, the VFM software does not support removable media and it is not necessary to manipulate bytes into a particular order. This may need to occur as part of the transition to support removable media.
FET	File Entry Table; a sequentially ordered control structure supporting VFM file entries.
FET Link	The structure which chains sequential tables together via logical pointers in a pseudo-linked list architecture.
FIT	File Information Table; a sequentially ordered control structure supporting a particular file’s data entries.
FIT Link	See FET Link.
flat file system	A non-hierarchical filing system that does not support directories and/or sub-directories.
FMM	Flash Media Manager.
FNULL	0xFFFFFFFFh. Only used to refer to media pointers.
FSD	File System Driver, the core driver that manages storage of “file” objects on the flash media.
FULL header	Structure which contains a VFM header and VFM extended header, if the extended header variable is defined.
LONG	64-bit value.

media	The physical storage media–typically a Flash Card (comprised of one or more flash chips) or a Resident Flash Array (RFA) also consisting of one or more flash chips.
NULL	Zero.
partition	An integral number of contiguous flash erase blocks formatted in a particular way (to support VFM control/data structures)
PCIC	PC Interface Controller (Intel® 82365sl, or compatible); acts as the interface between the PC and a flash card via an MS-DOS memory window.
reclaim	The process by which deallocated flash media is made available for use.
RFA	Resident Flash Array, a fixed array of flash memory components within an embedded computer.
SCS	Intel Scalable Command Set. Contains additional commands related to the Common Flash Interface (CFI). These commands allow VFM to choose the correct read, write, and erase routines for any Intel® CFI-enabled device.
sector	Used interchangeably with VSB. User-defined size.
socket	Device with an onboard PCIC which supports the pinout of the flash card connector.
spare block array	One block per component, reserved for the reclaim process.
TSI	Test Script Interpreter, used to simulate software interaction with VFM.
VAT entry	**V**irtual Small Block **A**llocation **T**able entry. The VAT determines the logical to physical translation for each VSB within a block and the state of each VSB within that block.
VFM	**V**irtual Small Block **F**ile **M**anager.
VFM extended header	Additional descriptive data exists in a VFM header field, but contributes to the overall size of the file object.
VFM header	FILE_ENTRY structure describing a particular file's characteristics.
Virtual Component	User-defined driver-level logical component which could comprise a subset of a physical device or encompass multiple physical devices.
VFS	**V**FM **F**ile **S**ystem Driver.
VSB	Virtual Small Block, or sector. A logical storage unit consisting of a user defined number of bytes. The media is divided up into a number of VSBs.
VSB layer	Flash Media Manager layer for VFM.
WORD	16-bit value.