Managing Data in an Embedded System Utilizing Flash Memory

Authors:
Deborah See **Clark Thurlo**
Senior Software Engineer **Flash Software Development Manager**

Flash Software Development Group, Intel Corporation

Document Rev 1.01
June 30, 1995

1. Abstract

Determining the optimum Flash solution for an embedded application tends to be more of an art than a science. While several potential solutions exist, determining which solution to use can often be unclear. This document discusses data storage characterization, along with the options available to embedded systems that wish to store system data in Flash memory. This data may take the form of file information, code, or system parameters.

2. Introduction

As Flash memory continues to become a memory of choice, systems are beginning to demand more capabilities and functionality. Embedded Systems are beginning to use Flash memory in many ways other than code storage. New uses include file storage, parameter storage, and DRAM replacement. Embedded systems are taking advantage of these capabilities to reduce system cost as well as improving data reliability, providing easy update capabilities, increasing battery life, and providing stability after power loss. By using the software techniques described in this paper, many applications can replace existing data storage memory with Flash memory. For this paper, we will concentrate on the particular characteristics of NOR style flash. Although fundamental differences from other flash technologies exist, the general concepts in this paper still apply.

3. Flash Fundamentals

3. 1. Review

Flash technology brings unique attributes to system memory. Like RAM, Flash memory is electrically modified in-system. Like ROM, Flash is nonvolatile, retaining data after power is removed. Flash memory reads and writes on a byte-by-byte basis, while adding non-volatility: the flash bits retain their state even when the power is turned off. In general, bytes may be re-written, provided each bit changes from erase state to a non-erase state only.

During the “write”, or “program”, process, the Flash cells are changed from one binary voltage level to another (i.e. “1”s to “0”s). In the “erase” process, the Flash cells are set back to their original binary voltage level, or erase state (i.e. “0”s to “1”s). This erase process occurs on a block basis, where blocks are defined by a fixed address range. When a block is erased, all address locations within a block are erased in parallel, independent of other blocks in the Flash memory device. The figure below provides several examples that show the results of different program/erase commands on a Flash byte (using NOR Flash).

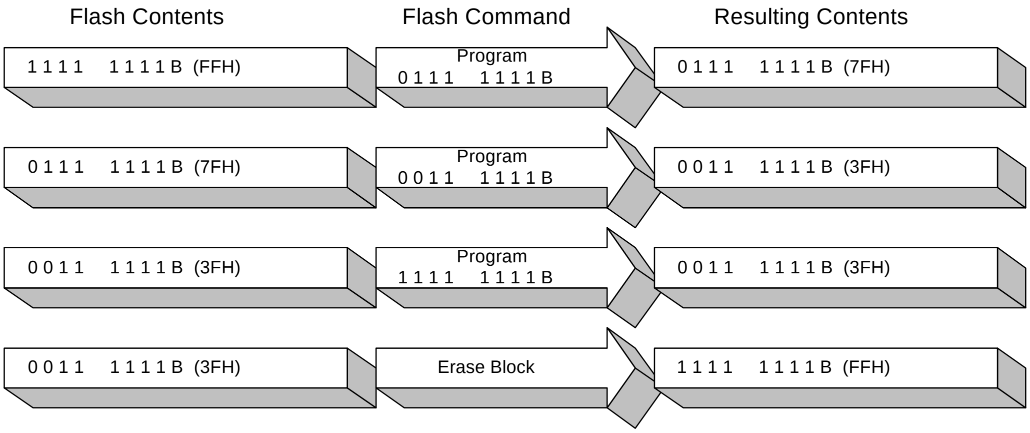

Flash Contents
Flash Command
Resulting Contents
1 1 1 1   1 1 1 1 B (FFH)
Program
0 1 1 1   1 1 1 1 B
0 1 1 1   1 1 1 1 B (7FH)
0 1 1 1   1 1 1 1 B (7FH)
Program
0 0 1 1   1 1 1 1 B
0 0 1 1   1 1 1 1 B (3FH)
0 0 1 1   1 1 1 1 B (3FH)
Program
1 1 1 1   1 1 1 1 B
0 0 1 1   1 1 1 1 B (3FH)
0 0 1 1   1 1 1 1 B (3FH)
Erase Block
1 1 1 1   1 1 1 1 B (FFH)

Figure 1: Example of Flash Commands/Content Changes

Flash memory products carry a “cycling” specification. This value indicates the amount of usage a Flash component can withstand before performance degradation begins. For NOR based flash, a “cycle” is equivalent to an erase operation. For example, if each of the bytes in one block are successively programmed and then the block is erased, 1 cycle has completed. Cycling is useful for determining the lifetime of a product, given a known data usage pattern.

3. 2. Flash Characteristics

3. 2.1. Cycling

Each block within a Flash component typically has a specific lifetime based on the maximum number of recommended erase cycles. This lifetime may differ based on the operational temperature range of the component. System designers must evaluate the data they wish to store in Flash to determine the average data sizes and frequency of data update to determine if cycling is an issue for their system. Systems that have frequent data update should consider using “wear leveling”. Wear leveling can be implemented into file systems to balance cycles evenly across all blocks in the Flash media.

Often, the actual number of cycles needed for a particular design is often over-estimated. This is generally due to the difficulty of accurately estimating the true number needed. That is, in applications where the customer has the ability to update Flash, it may be difficult to accurately predict the maximum erase cycles needed. Either a model is developed - and usually “guardbanded” - or a data sheet claim is chosen and somehow justified as needed. In order to shed some light into the outcome of either method, a basic cycling usage model is presented in Figure 2.

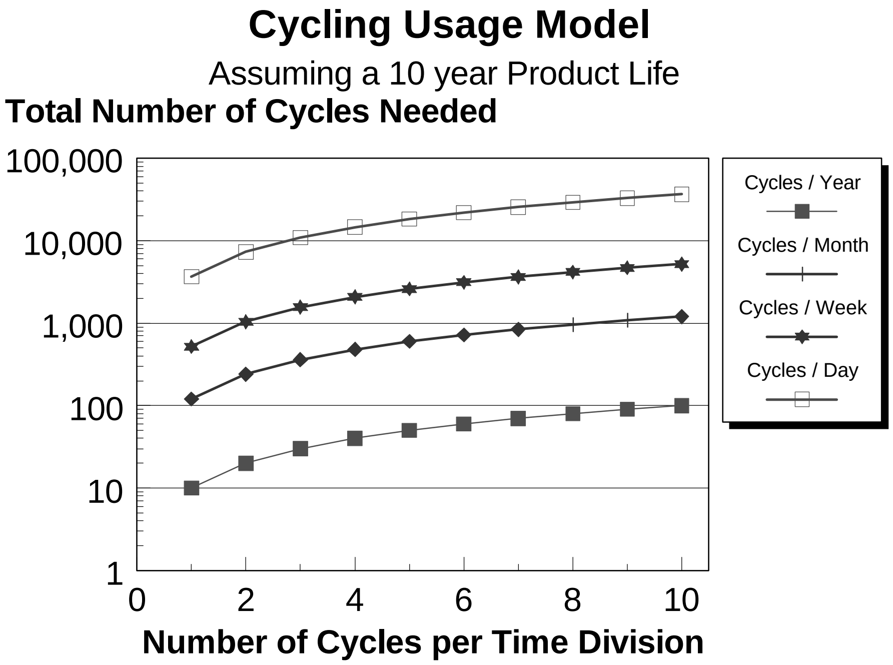


Figure 2: Cycling Usage Model

Essentially, the model allows the user to choose the number of cycles per time division (year, month, week, or day) and determine the total number of erase cycles needed (assuming a 10 year life cycle). For example, if a design needed to erase an entire Flash component 2 times a day for 10 years, the total number of erase cycles needed would be 7,300. On the other hand, an application requiring 2 times a week for 10 years would only require 1,040 cycles. Finally it is noted that 10 erase cycles per day for 10 years would only require 36,500 erase cycles. These estimates assume wear leveling is occurring. So, for any file systems that do not utilize wear leveling, you must evaluate the potential impact.

3. 2.2. Program / Erase Latency

Systems that are very time intensive and have little time to write data to Flash or erase Flash must carefully analyze their timing requirements. Flash components take a significant amount of time to erase a block or program when compared to RAM. Some systems may require RAM buffering of incoming data to occur while data in an alternate RAM buffer is being programmed into Flash. Some Flash vendors have components that contain dual page buffers, internal to the component, to assist with buffering and to increase programming speed.

3. 2.3. Power Off Requirements

As with most system memory, where status structures are stored on the media, it is important to evaluate system power off requirements. For systems in which power is continuously available, power off issues are not as important as in systems that are continuously receiving system resets. Any system which receives frequent system resets must utilize data storage software that adheres to power off restrictions. This means that all structures which store status or data management information must utilize bit modifications to indicate status changes, and must perform data management structure modifications in an appropriate order. Losing an update in progress should be the worst case scenario for a power loss during an update to file management structures, leaving all of the current structures intact.

3. 2.4. Read While Write

Many embedded systems designers wish to use a single Flash component for code execution as well as file storage. These systems must execute Flash program and erase algorithms from RAM. It is not possible, with many of today’s Flash components, for the processor to read (or execute) from a Flash component while simultaneously programming or erasing the same Flash component. There are several basic reasons for this limitation:

a) Address and data lines in a Flash component are not dual ported and physically cannot address two separate sections of the component at the same time. Adding this feature would result in prohibitive die size and silicon costs.

b) Many Flash components contain automatic state machines. After a Flash component receives a program or erase command, it automatically transitions to a status mode where reads to the component provide status register information instead of data. From the processor’s perspective, once the program or erase command has been sent, the processor can no longer read the machine code from the flash array.

System designers that wish to use Flash for code execution and file storage can do so by having the Flash program and erase algorithms execute from RAM. Figure 3 gives a simple Flash memory update algorithm example. It shows portions of the code that must be executed off-chip, with shaded areas showing ‘windows’ where the Flash memory array can be accessed if needed (by issuing the Read Array command and then reading from desired locations). The amount of code executed in RAM to actually program/erase the Flash memory is very small. Care must be taken to disable interrupts during program/erase algorithms in RAM to ensure accesses to the Flash do not occur while the component is in status mode. If interrupt vectors and handlers are maintained in RAM and do not access the Flash component, it is unnecessary to disable interrupts.

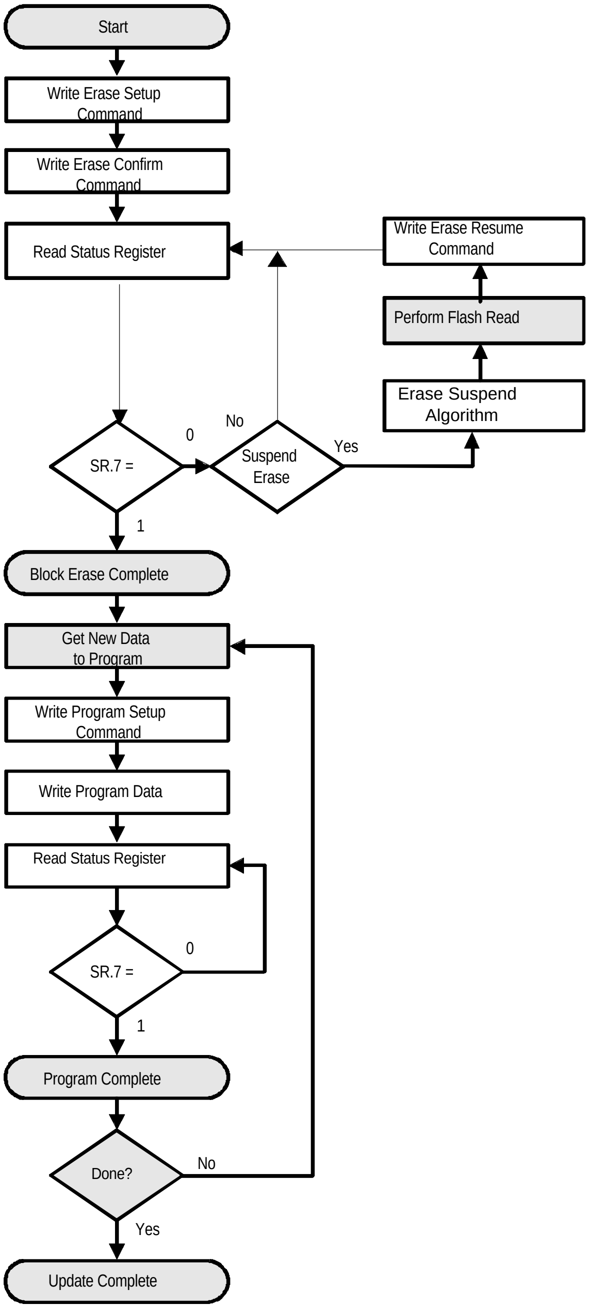


Figure 3: Read while Write emulation example

3. 2.5. Reclaim

Since Flash is erased on a per block basis, a method must exist to manipulate data to allow discarded information to be erased while valid information within the same block is retained. This process is called reclaim or garbage collection. Reclaim occurs on a block basis or on a partition basis, depending on the file system and the format of the media. Block reclaim typically consists of copying valid data to a spare or unused Flash block, then erasing the original block. Partition reclaim consists of copying each block to a spare block, erasing the original block, then copying valid data to an appropriate location.

3. 2.6. Block Architecture

Flash components generally have two different architecture styles: symmetrically blocked components and asymmetrically blocked components. The architecture style chosen depends on how the system will use the flash.

3. 2.6.1. Symmetrically Blocked Components

Symmetrically blocked components contain multiple erase blocks, where each block is identical in size as shown in Figure 4. This style of component architecture is ideal for systems that wish to use components as a resident flash array filing system or as a Flash card filing system. This style of component can be used as code storage or data/code storage as well.

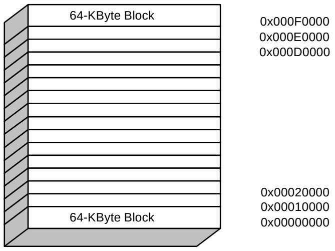


Figure 4: Symmetrically blocked component (e.g. Intel 28F008)

3. 2.6.2. Asymmetrically Blocked Components

Asymmetrically blocked components contain multiple erase blocks, where each block is identical in size except for the top or bottom block. This “special” block is usually broken down into several blocks. This breakdown typically consists of a “boot block” , two “parameter blocks” and the remaining portion becomes a smaller version of a “main block” as shown in Figure 5. The boot block is designed to contain the boot code for a system. It usually has some level of protection from accidental overwrite. The parameter blocks are designed to maintain system parameters. The remaining space can be utilized as storage for code, data, or code/data. Asymmetrically blocked components can be used as they are designed, or can be treated as a symmetrically blocked component through the use of software.

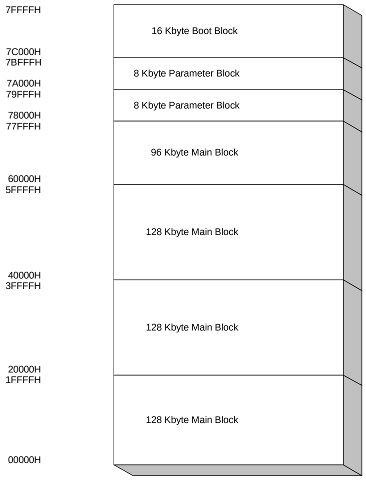


Figure 5: Asymmetrically blocked component (e.g. Intel 28F400 BX-T)

4. Defining Flash Usage

Before selecting a method for storing data in Flash, it is important to determine how the Flash will be used. This assists in determining if a solution already exists that you may use, or if you must create a proprietary solution.

4. 1. Characterizing Data Type

4. 1.1. ROM Replacement

Flash is ideal for ROM replacement due to its upgradeability. Storing object code in Flash is much more straightforward than storing code and file data. Wear leveling is typically not an issue in this circumstance. The system designer must determine if the object code is stored in one single object file or if multiple files are required. If one single object is desired, the data can be later upgraded in a raw object file format via a PCMCIA Flash card or through an I/O interface. If multiple objects are desired, they can be treated as raw information on a PCMCIA Flash card with each file beginning at a predetermined offset, or transferred through an I/O interface. If predetermined offsets are not known, a jump table can be created. Information can be placed onto Flash through PROM programmers, automated handlers, or a PC/host interface.

4. 1.2. File Storage

There are several potential solutions for file storage in an embedded system. To determine which solution is appropriate, system file storage analysis is required. It is necessary to determine how much memory is available in the system for file system object code storage. It is also useful to determine average file size and average file update frequency (i.e. once per week, once per day, etc.). If an average value is difficult to evaluate due to a variety of file sizes/types, it is best to evaluate a file system based on the attributes of the files that are updated most frequently. It is also useful to evaluate total memory size of files that do not change under normal circumstances, if such files exist. Section 4.2 assists in defining this information.

4. 1.3. ROM Replacement and File Storage

ROM replacement in conjunction with file storage requires another evaluation: if code is being executed from Flash and file data is stored in the same component, the Flash program and erase algorithms must be placed in RAM before execution as described in section 3.2.4. Systems that utilize Flash for both of these functions must also evaluate wear leveling more closely than those that do not.

4. 2. Characterizing Flash usage

Many embedded systems need to utilize Flash for a filing system, but are unsure of whether the lifetime of the Flash components will last beyond the estimated product lifetime, given their required use of the Flash.

4. 2.1. Defining Average File Size

Defining average file size is difficult for many applications. While determining this information is tedious, the information is extremely useful for evaluating the lifetime of the Flash components in the system. If many file sizes exist in the system, the files that are updated more frequently should factor the most in average file size determination.

4. 2.2. Characterizing Update Occurrence of Information

Characterizing the frequency of updates assists in determining the number of Flash cycles required to meet the required product lifetime. Although this is often difficult to assess in systems that have a variety of files, the files that are updated the most often are the files that should be evaluated the closest. A worst case evaluation ensures the product meets the product lifetime expectations.

4. 2.3. Evaluating Product Lifetime

Many embedded products have an estimated product lifetime. This information, when combined with average file size, frequency of updates, block size, and the number of erase blocks, help verify that Flash is an appropriate solution. The example product lifetime calculation assumes the use of wear leveling in the implemented file system. Those file systems that do not utilize wear leveling must divide the estimated lifetime by the number of blocks that exist in the system.

Example 1 below shows an example product lifetime calculation for systems with wear leveling as well as those without.

Number of Blocks = 16
Block Size = 64 KB
Average File Size = 100 KB
File update frequency = 100 / day
Component block erase cycles = 100,000
Total memory = 1024 KB = 1 M

$$\frac{\text{1024 KB}}{\text{1 erase cycle}} * \frac{\text{1 file update}}{\text{100 KB}} = \frac{\text{10.24 updates}}{\text{cycle}}$$

$$\frac{\text{1 cycle}}{\text{10.24 updates}} * \frac{\text{100 updates}}{\text{day}} = \frac{\text{9.77 cycles}}{\text{day}}$$

$$\frac{\text{1 day}}{\text{9.77 cycles}} * \frac{\text{100,000 cycles}}{\text{lifetime}} = \frac{\text{10235.4 days}}{\text{lifetime}}$$

$$\frac{\text{10235.4 days}}{\text{lifetime}} * \frac{\text{1 year}}{\text{365 days}} = \frac{\text{~28 years}}{\text{lifetime}} \quad \text{(with wear leveling)}$$

For those systems that reclaim an entire partition each reclaim with a stationary spare block, the number of years is divided by the number of blocks that are erased per reclaim. The example given would provide a worst case of 28/16 = 1.75 years / lifetime. These values must be compared to the product lifetime to determine if they are realistic. If the example product lifetime is 5 years, this would indicate that a Flash filing system with wear leveling is more than adequate, however, a filing system that reclaims the entire partition each reclaim with a stationary spare block is not an option.

Example 1: Example Product Lifetime Estimate Calculations

5. Defining File System Needs

Pre-determining the file system characteristics your system will utilize is important and assists in determining which file system is appropriate for your needs. Several file system characteristics are of interest here, and vary from file system to file system. The characteristics are as follows:

- **Hierarchical Directory Support:**

Hierarchical directories include the capability of defining directories with sub-directories, or a directory tree. Hierarchical directories provide a method to classify files, and provide a method to speed up the search for files in systems that maintain a substantial number files. Some file systems support hierarchical directory structures, while others do not. Although this may seem like a limitation, many systems do not utilize the hierarchy and would prefer a smaller code footprint.

- **File Editing:**

It is important to determine if the files stored in Flash are written and read once before erasure, or if files are updated (edited) on an ongoing basis. While some file systems allow direct editing of files, others target systems do not require updates of files and prefer a more simplistic file system. Also, some systems have enough RAM available to perform file editing in RAM. In other words, these systems read the entire file into RAM, modify/edit the file, write the file back to non-volatile memory, and then delete the old file.

- **File Access:**

Many applications have only one interface to a file system, and require write access to one file at any given time, while other applications utilize a multi-tasking operating system, and require access to several files at any given time. Some file systems have limited access to multiple files.

- **Data Transfer:**

Some file systems support data transfer to a host PC, while others do not. A particular Flash file system may only be available on the embedded system or use a proprietary format. If data transfer to a host system is a requirement for your application, this should be evaluated closely. Some Flash media managers or file systems have DOS device drivers or DOS redirectors available. These perform a direct interface to the Flash, but are loaded at power on and reside permanently in system memory. Some file systems may have filter applications that serve as a transfer mechanism to place files onto the DOS system where the file may then be manipulated.

- **Wear Leveling:**

Many sector based or block based Flash filing systems provide inherent wear leveling. However, other filing systems which are intended for systems with infrequent updates do not support wear leveling.

6. File System Options for the Embedded Market

File system options for the embedded market range from sector-based translation layers (that manipulate logical to physical addressing for an existing sector based file system) to robust linked-list filing systems. Each option has several advantages and disadvantages. While a full filing system may seem ideal for one embedded application, its code footprint may not be suitable for another embedded application. This section identifies several available options, and their capabilities and limitations.

6. 1. Flash Translation Layer (FTL)

FTL is a sector based Flash Translation Layer that provides logical to physical sector translation. FTL performs sector mapping to allow Flash to appear as a rotating disk drive. While the host file system sees the Flash card or resident Flash array (RFA) as a continuous sectored medium, FTL relocates these sectors and tracks the logical-to-physical relationship. Figure 6 below provides a simplified graphical representation of the sector translation and relocation that occurs with FTL.

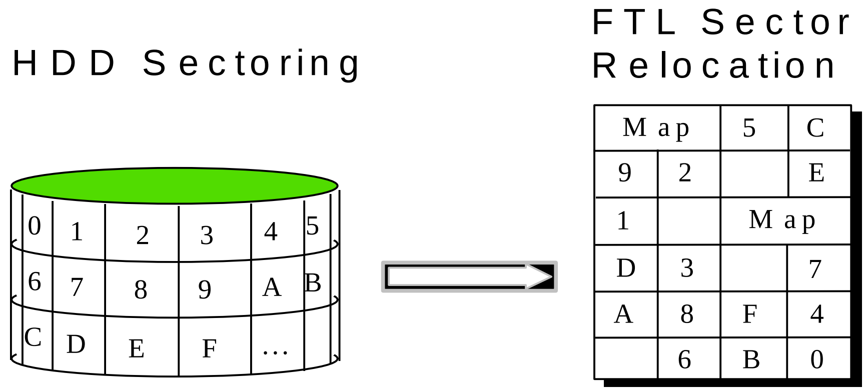


Figure 6: FTL Sector Relocation

This logical-to-physical mapping allows the Operating System to concern itself with only file operations. Because the O/S already oversees these file operations, the FTL solution can provide compatibility with existing applications and media utilities while presenting a small code footprint. Figure 7 demonstrates this layered approach, where FTL works with the existing File System to control the Flash media. Of important note here is the fact that FTL is able to relocate sectors to any position in the Flash media, making large Flash blocks appear as smaller erasable sectors.

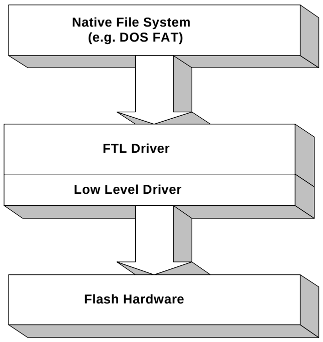


Figure 7: Software architecture using FTL

6. 1.1. FTL Format

FTL first breaks the Flash Array down into units called erase units. Each erase unit is the size of a Flash Erase Block. The size of the Erase Block depends on the component being used, as well as the mode the component is operating in.

Each Erase Unit is evenly divided into Read/Write Blocks (or sectors) of equal size. These Read/Write blocks are the same size as a sector presented by the host file system. Under DOS, a Read/Write block is 512 bytes. Although presented to the Operating System in sequential order, Read/Write blocks are not stored in sequential order on the Flash media; they are continually being re-mapped so that block erases and wear-leveling may take place.

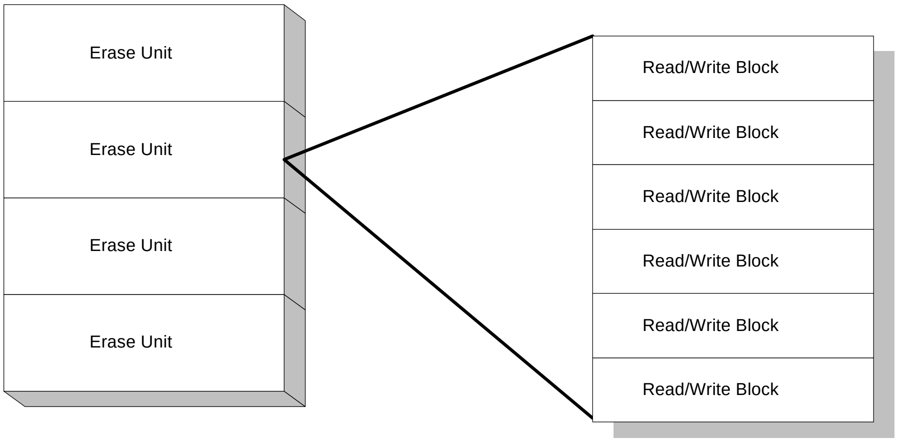


Figure 8: Erase Units divided into Read/Write Blocks

6. 1.1.1. Erase Unit Structure

At the beginning of each Erase Unit (physical offset 0x00) is the Erase Unit Header (EUH), that containing data both about the entire FTL partition and the particular Erase Unit in question. Each Erase Unit has a unique number assigned to it, called the Erase Unit Number (EUN), that can only be found in the Erase Unit Header. It is important to note that, like the Read/Write blocks within the Erase Unit, the Erase Units are often re-mapped by FTL for performance reasons and are not stored in sequential order. To assist with tracking, a logical erase unit number (LogicalEUN) is stored in the EUH for each Erase Unit which allows the FTL to keep track of the mappings.

Another structure kept within the Erase Unit is the Block Allocation Map (BAM) that contains allocation information for each Read/Write block within the Erase Unit. The BAM can be found by checking the Block Allocation Map Offset (BAMOffset) field within the EUH and examining the region starting at that offset into the Erase Unit. The BAM usually begins at offset 0x44 within the Erase Block (immediately after the EUH). The remainder of the Erase Unit is taken up by file information described by the BAM. All fields within the Erase Unit Header Structure are shown below in Table 1.

Offset	Field	Offset	Field
0	LinkTargetTuple	32	FirstVMAddress
5	DataOrganizationTuple	36	NumVMPages
15	NumTransferUnits	38	Flags
16	EraseCount	39	Code
20	LogicalEUN	40	Serial Number
23	EraseUnitSize (Log$_2$ N)	44	AltEUHOffset
24	FirstPhysicalEUN	48	BAMOffset
26	NumEraseUnits	52	Reserved
28	FormattedSize		

Table 1: Erase Unit Header Structure

6. 1.1.2. Block Allocation Map (BAM) Structure

The Block Allocation Map (BAM) is composed of signed 32 bit entries. There is an entry in the BAM for every Read/Write block present within the Erase Unit. These entries indicate the use of each Read/Write block, including use for data, Virtual Map Pages, Control Structures, and Replacement Pages.

The Virtual Map Page entries in the BAM are later used to construct the Virtual Page Map which is a page table for the Virtual Block Map.

32Bit Entry	Meaning	Description
0xFFFFFFFF	Free	Read/Write Block is available to be written to.
0x00000000	Deleted	Data in block is invalid. This block has data in it and must be erased before it may be reused.
0x00000070	Bad	Block is unusable. This block may not be read from or written to.
0x00000030	Control	FTL control structure (BAM, EUH, ECC, etc...)
0xXXXXXX40	DATA or Map Page	Contains either DATA or a Virtual Map Page
0xXXXXXX60	Replacement Page	Contains a Replacement Page

Table 2. Block Allocation Map entry information

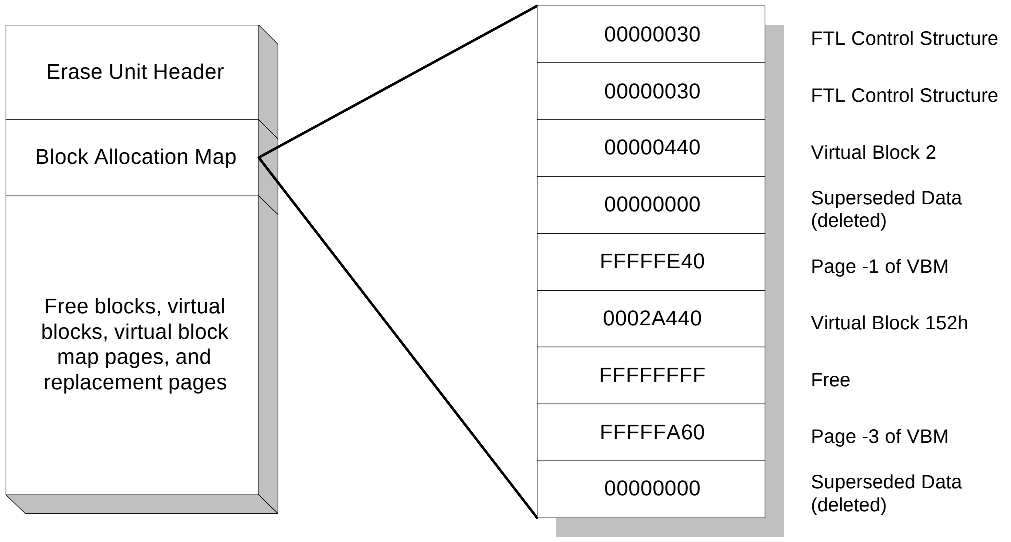


Figure 9: Example contents of a Block Allocation Map

6. 1.1.3. The Virtual Block Map (VBM)

The Virtual Block Map consists of several pieces or “pages”. Each page is identified through values in the Block Allocation Map: if the BAM entry is negative, it represents a Virtual Block Map Page. As an example, assuming a Read/Write block size of 512 bytes, each Virtual Block Map Page represents 128 Virtual Blocks or 64 Kbytes of data storage. The Virtual Block Map Pages are used to construct a table called the Virtual Block Map (VBM) that is the fundamental table for translating Operating System requests into logical locations. The Virtual Block Page Maps contain translation information to get to any virtual location on the card.

6. 1.2. Reclaim

Reclaim for the FTL media manager is done on an individual erase block basis. Each block is evaluated based on the amount of discarded data and a selection is made for which block to reclaim. During a reclaim, the Erase Unit Header information, Block Allocation Map, and valid sectors are transferred to a spare block, and the invalid block is then erased.

6. 1.3. Capabilities vs. Limitations

Table 3 below identifies the capabilities and limitations for the Flash Translation Layer. FTL is an industry standard with multiple vendor sources. It is a translation layer geared toward systems that have a sector-based filing system already in place and need to add flash support with a minimal code footprint. This software is also geared toward applications with a high frequency of file update.

File System Feature	Capabilities	Limitations
Media Efficiency	Requires one spare block for cleanup (if partition is updateable). Allows a percentage of the card to be labeled as overhead to improve performance.	
Power off recovery algorithms		Filing system power off algorithms must also be considered.
File editing capabilities	Dependent upon file system layer that is used with this translation layer.	
Partitioning capabilities		Current implementations do not support multiple partitions.
Multiple File Access capabilities	Dependent upon file system layer that is used with this translation layer.	
XIP capabilities		Not applicable.
Code Size	~24 KB (DOS implementation) -varies based on vendor, file system layer, processor, etc.	
Standard adherence	PC Card (PCMCIA) accepted standard	
Reclaim Speed	Reclaim occurs on a per block basis.	
File Seek Speed	File seek speed is dependent on the file system used with FTL.	
Portability	Must be adapted to each file system and processor by third party vendor.	
Data Transfer to the PC	DOS block device drivers exist	
Media Types Supported	Flash RFA/ Flash Card	
Directory Structure Support	Dependent upon file system layer that is used with this translation layer.	
Wear Leveling Support	Dependent on vendor supplying FTL. Inherent wear leveling occurs automatically.	
Other		FTL is a sector based media manager. It is not a filing system. FTL must co-exist with a sector based filing system, such as DOS.

Table 3: FTL Capabilities and Limitations

6. 2. Linear File Store File Manager (LFM)

Unlike FTL, the Linear File Store File Manager is an entire file manager. The Linear File Store (LFS) structure is a PCMCIA defined data structure which provides a method for storing variably sized file objects. LFS File Manager (LFM) is a particular File Manager implementation that utilizes this data structure. File objects are stored contiguously in a partition and are arranged in a one-way linked list. The topmost 32 bytes of each file object contain an LFS Header. The header contains basic file information including a link to the next file object, which begins immediately after the current file. See Figure 10.

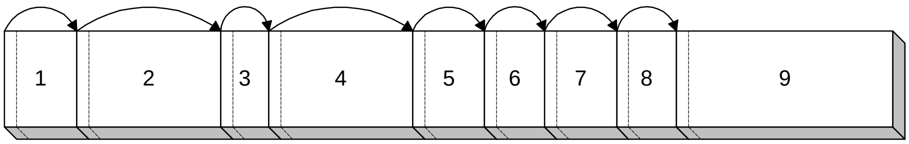


Figure 10. Linear File Store Overview

The LFM software has certain restrictions due to its original intent: to provide a file storage system that stores files contiguously for execute-in-place capability. These restrictions include: no hierarchical directory structures, no direct file editing, and one file open for writes on a per partition basis.

6. 2.1. LFS Format

There are two sets of format information that can illustrate the functionality of LFM. These are the file format and the media format. Figure 11 shows the file format. Each file contains a link to the next file, flags indicating file status, a type and id used for file identification, and an optional extended header to track user specific file information.

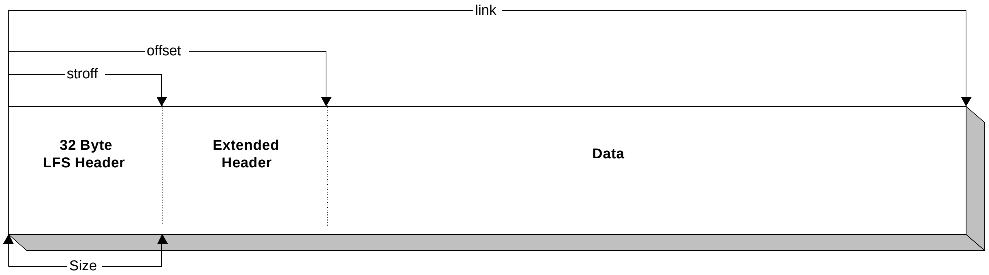


Figure 11. LFS Header Offsets

```
typedef struct lfs_header {
   ULONG   link;
   ULONG   size;
   ULONG   type;
   ULONG   offset;
   ULONG   flags;
   ULONG   stroff;
   ULONG   id;
   ULONG   reserved;
} LFS_HEADER;
```

Listing 1. LFS Header Offsets Structure

Field	Description
link	This field contains the offset from the start of this header to the next LFS header in the partition. If each bit in this field is equal to bit D0 of the flags field, this is the last entry in the partition.
size	This is the actual size of the LFS header.
type	PCMCIA requires that LFS headers contain a stamp indicating the type of the header. For our implementation, this field has been assigned ZERO (0), indicating the 32 byte header above.
offset	This field indicates how far from the start of this header into the entry the file data begins. This is allows LFS implementations that use extended headers to be compatible with drivers that can’t read the extended header.
flags	This is a bit-mapped flags field. Bit D0 indicates the nature of the Flash (1 erase or 0 erase). Bit D1 indicates whether or not this file entry is valid or deleted: if D1 matches D0 the file is valid, if D1 differs, than that file is deleted.
stroff	This field points to an extended header. The actual location of the extended header is determined by adding the value in this field to the address of the LFS_HEADER.
id	This value is unique to each file object.

The example LFM format shown below in Figure 12 shows an Intel 28F008 component, divided into 2 partitions. This provides an example of multiple partitions as well as demonstrating the first block's use for power off recovery. Systems can optionally use the entire media as one partition. Partitions allow the user to emulate directories. These partitions allow multiple files to be open for write since each partition can allow one file open for write. The limitation of one file open for write per partition is due to the LFS standard, requiring the placement of a file immediately after its predecessor.

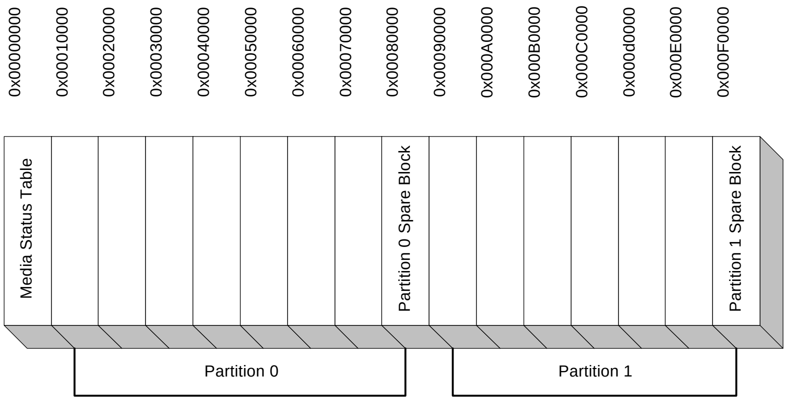


Figure 12. Example Multi-partition LFM Media Format on Intel 28F008

6. 2.2. Reclaim

Performing a reclaim on a linked list file system such as LFM is much more complex than reclaiming a file system format that maintains structures and data within block boundaries. Recovering file space which has been marked as discarded requires copying each block to the spare block, erasing the original block, determining which information is still valid within the spare block, and copying the valid information back to the current end of the linked list chain. Since file headers and data are not limited to block boundaries, the reclaim algorithm must track information that indicates structures that cross block boundaries. Reclaim for this filing system must occur on an entire partition. This ensures that the linked list remains intact. Reclaims may be time consuming and are most effective when they can be performed in the background. Due to the nature of the simplistic linked list, the spare block permanently remains the last block in the partition. Wear leveling will not occur with the simple linked list file system. However, this should not be required for systems with infrequent file updates.

6. 2.3. Power off recovery

The structures within LFM are designed to handle the possibility of power loss. During initialization, LFM attempts two types of power off recovery. The first step of power off recovery is to determine if any partition in the media was in the process of undergoing reclamation. To assist with this, reclaim uses the first block of the media to track the state of reclaim. This block is used by all partitions and allows one partition to undergo reclaim at any given time. On initialization, this reclaim state information is used to determine if any partition was currently being reclaimed, and if so, provides the information to continue and complete this process. The second step to power off recovery is to determine if any files were left open for write at power off. These files are closed and marked as deleted (discarded) during initialization. This is done by searching from the end of a partition backwards to the last data that does not equal FF Hex. This address is then compared against the last file's address to determine if any files were left open for write during power off.

6. 2.4. Capabilities vs. Limitations

Table 4 below identifies the capabilities and limitations for the LFM filing system. This filing system is geared toward systems that store files with low update frequency, or systems that need XIP capabilities.

File System Feature	Capabilities	Limitations
Media Efficiency		Requires one spare block per partition plus one spare block per entire Flash media for power off recovery.
Power off recovery algorithms	Handles closing and deleting files left open for write as well as having robust recovery algorithms for power off during reclaim.	Potential of losing entire current file open for write if file is not closed at power off.
File editing capabilities		User could incorporate file editing capabilities into the extended header but this would be a user option only.
Partitioning capabilities	Allows multiple partitions with a pre-defined maximum. Useful for directories, code/data separation, and multi-tasking systems that need multiple files open for write.	
Multiple File Access capabilities	Allows multiple files open for read and one file open for write per partition at any given time.	
XIP capabilities	Generic interface for XIP	
Code Size	~ 13-14 Kbytes	
Standard adherence	File headers adhere to PCMCIA LFS standard	
Reclaim Speed		Reclaim must occur over an entire partition which may be slower than a sector based file system.
File Seek Speed		Seeks may be slow for systems which have many small files due to lack of hierarchical directory structure.
Portability	OEM needs to modify O/S interface and low level hardware interface	
Data Transfer to the PC	DOS redirector exists, example filter application and source code exists.	
Media Types Supported	Flash Card/Flash RFA	
Directory Structure Support		None.
Wear Leveling Support		None. Entire partition is reclaimed through a stationary spare block.
Other		

Table 4: LFM Capabilities and Limitations

6. 3. Flash Translation Layer Logger (FTL Logger)

The flexibility of the Flash Translation Layer(FTL) data structures makes it possible to arrange data in a symmetric layout on a Flash PCMCIA card and still maintain FTL compatibility. Certain embedded applications log data for foreign post-processing (on a host system). Such systems would benefit from the exchangeability of FTL, and could avoid its overhead through this logging concept. The FTL Logger provides a method of formatting a card to maintain FTL compatibility without using FTL in the embedded system.

6. 3.1. Pre-formatting

A utility exists on the desktop which pre-formats Flash memory cards. This utility queries the user to determine the number of files that should be created. The utility calculates the amount of overhead required by the filing system (e.g. DOS FAT/FTL) to pre-format the card. It then writes all of the filing system overhead to the card, laying these structures down in such a way that it symmetrically utilizes a portion of each erase block. All data areas are untouched (left in the original Flash erase state). The utility informs the user of the areas which are left untouched on the card (I.E. Bytes 0 - 1000 Hex in each erase block). In one implementation, this format information is stored in reserved areas on the Flash media, for the embedded system's data logger to reference.

6. 3.2. Data Logging

The embedded system needs to know the pre-defined overhead area which it must leave intact. It begins logging structured data at the first area allowed, and continues logging until it reaches the end of the area allowed (for instance, FTL keeps its overhead at the beginning of each block). Once the end of the erase block is reached, it skips the next structure area, and continues logging. This continues until the end of the card is reached, or the data logging is complete.

6. 3.3. Post Processing

Once the embedded system has finished logging data to the card, the card is transferred to an FTL-enabled desktop system. The files are accessed via standard file access methods, possibly copied to the desktop for further analysis.

The embedded system has two options to determine parameters such as card size, file size, number of files, and overhead size. The first option is to assume certain values for these parameters. If this option is chosen, the embedded system can pre-define these parameters. The second option is to have the preformatter store these parameter values into reserved space (available for FTL, for instance, stored in the reserved area of the first FTL structure in block zero). Using this option, the embedded system can dynamically read these values from a known location on the card, providing greater flexibility.

7. Other Flash Needs

7. 1. EEPROM replacement

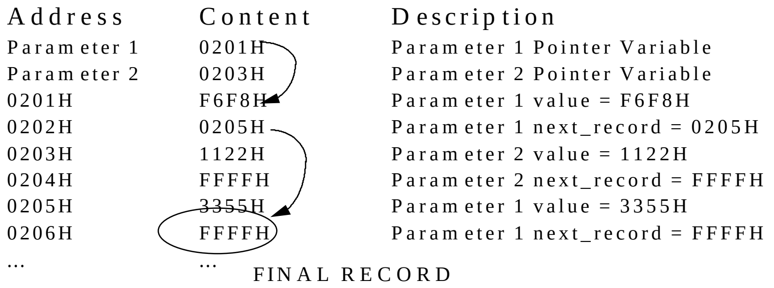
Many systems currently contain Flash for upgradeable software, along with an EEPROM for parameter storage. These systems can now remove EEPROM from the system by storing parameters in Flash. This capability can reduce board space, lower system cost, reduce parameter write time, and allow systems to maintain lower program and erase voltages. A comparison of Flash and EEPROM is shown in Table 6.

	Flash	EEPROM
Byte Write Time (typical)	10 us	10 ms
Erase Time (typical)	800 ms / 8 KB block	N/A
Cell Size Cost	1 transistor / cell	2 transistors / cell
Program/Erase Voltage (int/ext)	5V / 12V	5V / 21 V
Cycling	100,000 **Erase** cycles / block	100,000 **Write** cycles / block

Table 6: Flash vs. EEPROM comparison

7. 1.1. Parameter Tracking

Two Flash blocks are required for EEPROM replacement. The two 8 KB parameter blocks in asymmetrically blocked Flash components are ideal for this function. Parameters are written to the first block in a linked list fashion until the block becomes full. At this point, valid parameters are transferred to the second block and parameter updates can continue. Once this block is full, parameters are transferred back to the first block, and so on. The basic element of EEPROM replacement is a linked list of parameter structures that track the parameter value, and a pointer to the next parameter instance. Figure 13 illustrates a simplified example of a linked list of parameters.



Address	Content	Description
Parameter 1	0201H	Parameter 1 Pointer Variable
Parameter 2	0203H	Parameter 2 Pointer Variable
0201H	F6F8H	Parameter 1 value = F6F8H
0202H	0205H	Parameter 1 next_record = 0205H
0203H	1122H	Parameter 2 value = 1122H
0204H	FFFFH	Parameter 2 next_record = FFFFH
0205H	3355H	Parameter 1 value = 3355H
0206H	FFFFH	Parameter 1 next_record = FFFFH
…	…	

FINAL RECORD

Figure 13: Simplified Example

7. 1.2. System Requirements

The following system requirements must be met in order for EEPROM replacement to work in your system:

- Approximately 8 KB of memory storage for software library.
- Approximately 1 KB of RAM storage for Flash program and erase algorithms. These must execute from RAM due to Read-while-Write limitations.
- Either the interrupts must be disabled during program and erase algorithms, or the interrupt handlers and interrupt vectors must exist in RAM. Interrupt handlers existing in RAM cannot allow access to the Flash to occur during program and erase algorithms. Flash contents are inaccessible during program and erase commands.

7. 2. eXecute In Place (XIP)

XIP (eXecute In Place) is a method by which an operating system runs the read-only code of an application, library, or driver directly from non-volatile memory mapped into the CPU address space rather than copying those objects into RAM and then executing them.

7. 2.1. XIP Overview

Programs have long been stored in ROM. Flash memory, based on a NOR interface, features fast random read times, RAM/ROM-like interfaces, and non-volatility, making it particularly useful for XIP program storage. Flash used for program storage can be present in a system as a resident array of memory on the motherboard or in a removable media such as a PCMCIA card. Usually, ROM-based applications must be custom coded and essentially hand-located in memory to execute correctly. This labor-intensive approach has created strong barriers which prevent most applications from running in ROM. A converter can assist in the automatic conversion of an application to an XIP image.

XIP is best suited to environments where minimal weight, cost, power consumption and the ability to change the application mix in the field are important. Power conservation dictates Flash for storage and large amounts of DRAM are impractical for reasons of cost and power. The applications are limited to a small set which may be altered by the users .

7. 2.2. XIP for MS Windows 3.X

This concept has been demonstrated with Windows 3.x applications. However, DOS applications must be written ‘ROM-able’ for XIP to function. There are several major steps to make a Windows 3.x XIP application work: conversion of a Windows 3.x application to XIP, installation of the XIP image into Flash, XIP runtime environment creation, XIP application launching/ loading, and XIP application context management.

7. 2.2.1. XIP Conversion

Instead of relying on hand-crafted ROM applications, an automatic conversion tool can take an existing Windows 3.x application and convert it for execution from ROM. This conversion must handle several assumptions which RAM resident applications make about their runtime environment.

All applications have external memory references of some sort to the Operating System, other system support modules/libraries and different parts of the application itself. For example, such a reference is used to call the O/S file open entry point. Whenever a RAM resident application is constructed, these external references are stored in a form that can be resolved at load time since the specific memory address won't be known until the application is run. When the application is loaded, these references are modified in the code segments of the application and "fixed-up" to be the real, current memory addresses. This load time fixup technique, while fine for RAM resident execution, doesn't work when the code for the application is in ROM or Flash.

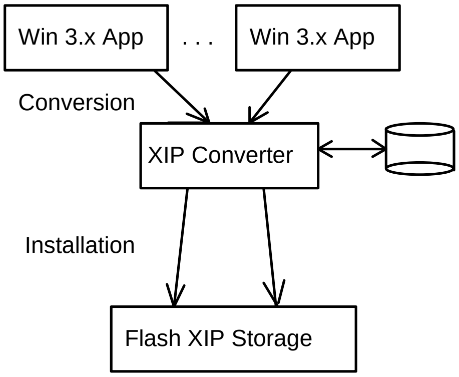


Figure 14: Off-line XIP Processing Flow

An XIP image can only have references requiring fixups (relocation references) for its R/W data segments since these segments are the only ones that will reside in RAM at execution time. Relocation references therefore must be resolved at image creation time. All relocation references must either be eliminated by fixing them at conversion time or by moving them to a R/W data segment. XIP uses a combination of several methods to resolve relocation references to accomplish application conversion. The methods are:

- Indirect far calls through the data segment
- Selector pre-allocation

7. 2.2.1.1.Indirect far calls

PTR references (far calls) can be changed into indirect far calls through the default data segment. The data segment resides in RAM and therefore can be patched. Indirect far calls have the benefit of retaining the runtime dynamic link capability of standard fixups. This is advantageous for calls to system DLLs whose entry points may shift from one version or OEM to another. To make a PTR reference into an indirect far call the instruction must be modified from **CALL *immediate ptr16:16*** to **CALL *memory address m16:16*** and a far address data item added to the default data segment. Note that the new instruction is one byte shorter so the code must be padded with a NOP (90h). The NOP can also be replaced with a SS override prefix to allow indirect far calls through the stack segment.

Since a reference can be from any segment of the application, the converter moves the relocation information for the reference from the segment being processed to the data segment's relocation information area. At image load time the Windows loader will patch the relocation references in the data segment with the correct SEL:OFF pair for the requested code. The .EXE fix-up table (retained as part of the XIP image) contains the symbolic references (library name, ordinal number) along with the SEG:OFF pair where the image must be patched. There may be apps where DS can't be expanded. In these situations the conversion tool can convert with selector pre-allocation. While less desirable this method will still work as long as the target of the reference doesn't change, e.g. by having a DLL be updated with a newer version.

7. 2.2.1.2.Selector Pre-allocation

Conversion by indirection alone won't resolve all fixups since some are not PTRs. In particular, BASE references require selector values that can be determined at conversion time, written to ROM and then used at execution time, possibly on different machines. This kind of selector used during XIP conversion is called a *preallocated* selector.

Pre-allocation is done out of a reserved range of selectors set aside in the XIP environment At XIP load time the pre-allocated selectors are aliased to the real selectors they represent. Aliasing makes the pre-allocated selectors refer to the same physical memory as the real selectors (i.e. same base and limit field value in the descriptor).
The result of the conversion is an application image that has 1 or more code and read-only data segments suitable to be executed from ROM, and 1 or more Read/Write data segments that will be copied into RAM at execution time. By making use of a Windows feature called a "self loader", the XIP application image file format remains a new executable (NE) style .EXE file. However, this EXE file will no longer run as a non XIP program. The image must be resident in memory so that it can be directly executed by the CPU. The created image also depends on the XIP runtime infrastructure in order to execute.

7. 2.2.2.   XIP Application Installation

Once the application has been converted to a ROMable form, it must be installed into Flash. This installation can take several forms as discussed later, however, any form must ensure that each code segment composing the application is stored contiguously in memory. This contiguity must be preserved since the CPU will be directly executing code from the Flash storage. If the code segment contents are disturbed during storage, an incorrect instruction stream will result causing the application or system to misbehave. For example, some storage techniques manipulate the data being stored (i.e. compression techniques) while others may not store the data precisely as requested so as to deal with peculiarities of the storage media such as breaking a file into file system allocation units.

7. 2.2.3. XIP Environment Creation

Windows segmented applications all require some number of selectors for the code and data segments composing the application. An application has those selector images embedded in its code and data segments at runtime, i.e. in instruction streams. These selectors are used to reference parts of the application or the DLLs or OS that are used during runtime. Therefore, in order to ROM an application, selector values are determined and assigned no later than when the application is stored in ROM. Some clever technique was needed to determine the selector values. This needed to be done in such a way to allow XIP applications on FLASH PCMCIA cards to be run on different machines or on the same machine with different system configurations. Also, there should be no arbitrary limits on the number of XIP applications that can exist (and be run) in the system. Selector values couldn't simply be assigned at conversion or installation time based on unused selectors available in the currently running system. Windows allocates selectors on demand and they change from system to system and boot to boot. Also specific selectors couldn't be dedicated to each XIP application since there are a very limited number of selectors in the system (less than 8K) and each application could consume up to 255 selectors (limited by the NE file format).

The technique selected was to pre-allocate a range of selectors for shared usage by all XIP applications. Each application would have its segments accessed via these selectors. Whenever an XIP application task switches, the segment descriptors are modified to reflect the new XIP task. Therefore, an XIP VXD allocates the block of selectors at windows startup time and sets up internal data structures that will be used to track the XIP applications presently running in the system. The XIP application converter knows the range of selectors to use for the system and allocates these selectors to segments as it is processing an application and eliminating fixup records. Since each XIP application reuses the pre-allocated selectors, any number of XIP applications can run in a system concurrently.

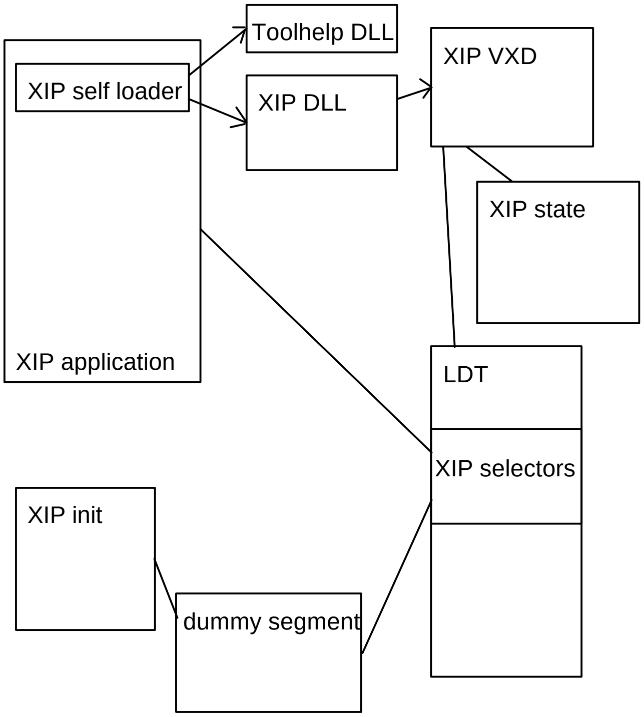


Figure 15. Runtime XIP Functional Blocks

The XIP VXD also maps the physical address space used to access Flash memory into linear address space. When an application is running, the appropriate address is mapped into virtual address space for the application.

In the current implementation, the XIP application is assumed to be stored on a PCMCIA Flash card. A fixed amount of the card address space is mapped to a fixed range of the system's physical address space. This space is mapped to a fixed size linear address range. In a real product, these limits would need to be more flexible so that mapping of the card to the system could accommodate larger Flash cards and different sized linear system address ranges.

7. 2.2.4.   XIP Application Launching/Loading

There must be some way to identify an XIP application in order to get it to execute. In DOS/Windows, any EXE file in a file system can be launched to run the application. When applications are stored in ROM, the system shell and application loader frequently must be modified in order to recognize the applications in ROM. We chose to use the Linear File Manager file system as the XIP storage medium so that no changes were required to support XIP program launching.

XIP applications appear as normal EXE files in a file system. When the EXE file is run, the XIP self loader is invoked instead of the normal Windows loader. This loader calls an XIP DLL which loads the data segment of the application into DRAM and applies the required fixups. The XIP DLL then calls the XIP VXD to finish setting up the application's segment descriptors and address mapping. The bulk of the self loader code was located in the XIP DLL so that XIP applications wouldn't redundantly contain the same code. The self loader code in each XIP application essentially contains far calls to the XIP DLL.

7. 2.2.5.   XIP Context Management

Essentially, XIP introduces another layer of virtualization into the system; XIP application code and data segment selectors are reused for each XIP application. The segment descriptor information for a particular XIP application is changed on a task switch along with any other task specific state. Whenever an XIP application is started, the XIP VXD adds information about the task to internal data structures and performs the appropriate task switch processing as required. When an XIP task terminates, the XIP VXD cleans up its internal state to make room for another XIP task. The state required by the VXD per task is around 30 bytes plus one 4KB page for the task's XIP segment descriptor information. This small overhead per XIP task allows a large number of XIP applications to run concurrently. Since most

Windows applications have more than 4KB of code, running it as an XIP application results in RAM savings.

In order to manage the sharing of the pre-allocated selectors for XIP applications, we needed a mechanism to notify the XIP DLL/VXD support code when a task switch (in or out) was occurring. The toolhelp.dll provides a mechanism to get a callback on task switch, creation and destruction. The XIP DLL receives this notification and then calls the XIP VXD to actually do the XIP relevant task switch processing.

7. 3. Raw Image Transfer to/from Host PC

Many embedded systems wish to use PCMCIA Flash cards to transfer object code or data between a host system and an embedded system as a raw image. This requires two items: a host transfer utility that reads data from or writes data to the PCMCIA Flash card and software in the embedded system to interpret/write information. This type of transfer is extremely useful when trying to transfer one file. It is also useful to transfer multiple files if those files are predictable in size and both systems always know where to find them. Another option for multiple files would be to create a jump table at the beginning of the PCMCIA card indicating where each file begins and its length. This simplistic transfer is useful for systems that wish to transfer well defined information to or from a host system. An example for use of this type of interface would be a system that wishes to upgrade software on an embedded system and the code image is one file. Another example might be to update printer fonts in a printer.

8. Conclusion

Determining your system needs is the first step toward defining what Flash software is necessary for your system. Many software solutions for integrating Flash into systems currently exist and are available through Flash component/card vendors or through third party vendors.

9. For More Information

Intel’s 1995 Memory Handbook set (order number 210830) provides specifications on Flash memory products and software algorithms. It is available by contacting your local Intel or distribution sales office, or by calling the Intel Literature Department at 1-800-548-4725.