AP-619

APPLICATION NOTE

FTL Logger Exchanging Data with FTL Systems

KIRK BLUM
TECHNICAL MARKETING ENGINEER

PETER LAM
SOFTWARE ENGINEER

August 1995

Order Number: 292174-001

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

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

or call 1-800-879-4683

1.0 INTRODUCTION

Recently, quite a few companies have developed software that allows Intel Flash products to emulate the functionality of disk drives. This allows flash to be used for mass storage of both code and data in PC applications. Several of the companies have “joined forces” with Intel and PCMCIA to standardize the flash media format used for this type of sector-based disk emulation. This makes it easy for the OEM or end-user to transfer data across a variety of PCs equipped with PCMCIA slots. This flash media format was recently approved by PCMCIA as the Flash Translation Layer (FTL) format. The FTL specification is now available from PCMCIA.

The flexibility of the Flash Translation Layer (FTL) data structures makes it possible to arrange data in a symmetric layout on an Intel Flash PCMCIA card and still maintain FTL compatibility. Embedded applications that would like to have the exchangeability of FTL without the overhead of a FULL FTL flash media manager on the embedded system can use this concept to their advantage. The FTL Logger provides a method of formatting a card to maintain FTL compatibility without using FTL in the embedded system.

Symmetric Block Formatting (SBF) is a method used to store data on a flash PCMCIA card by placing it on the flash media so that standard FTL software will be able to recognize it. This is accomplished by formatting a card with FTL-type data structures for a predetermined number of files. By rearranging and re-mapping of the various file system structures and the virtual maps, we can alter the data contained within that space without upsetting the FTL format. The format operates on one assumption: the number of files must be known before formatting, and the size of a file is determined by that number. This is where a Symmetric Block Format differs from standard FTL software: after the card is formatted, there can be no deviation from the size and number of files that were created unless the card is erased and reformatted. The files that will be logged onto the card will become read-only files. No editing, deleting, or addition of files are allowed from the embedded/target system.

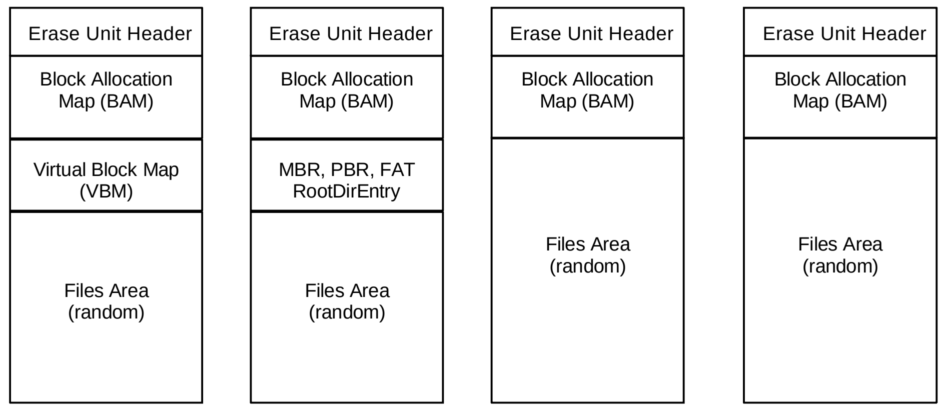
Symmetric Block Formatting fulfills the requirements of many embedded applications by providing a way to store data from an embedded system onto a removable flash card, insert it into a PC running FTL, and be able to retrieve the data as one or many standard DOS files through FTL. The only variation, or implementation specific detail is the format, which is derived from the nature of the data.

Before explaining how SBF works, the following sections will introduce a basic contrast between the regular FTL format and the symmetrical block format, and then explain how standard FTL manages the flash media.

2.0 THE SYMMETRIC FORMAT

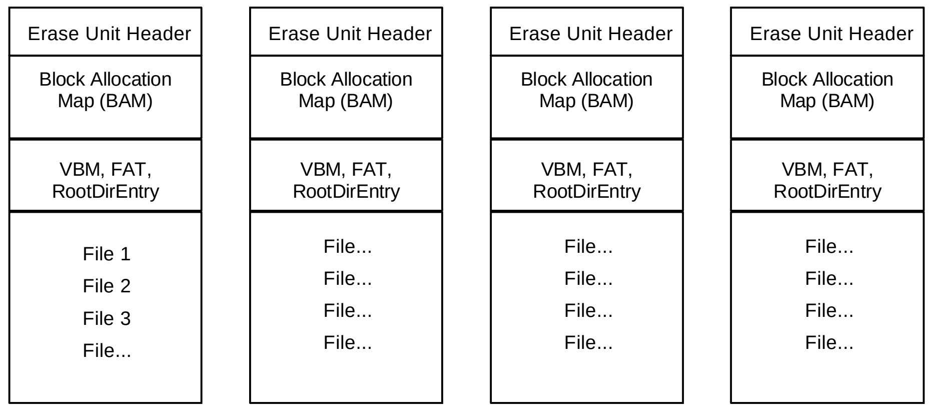
When a standard FTL format occurs, it puts all the file management overhead such as Virtual Block Map (VBM), FAT table, RootDirEntries, and any other structures in the first two logical blocks of the flash media. These first two blocks, therefore, have less space than the rest of the blocks to store file data. When adding or deleting a file, FTL looks for the available space on the card and set up the pointers and maps to allocate this space accordingly. Thus files can be scattered anywhere on the card in this fashion. There is no way for the user to know where the file is stored on the card, and, therefore, no way to log data into specific locations. Furthermore, the allocated space on the card cannot be immediately reused even after the file has been deleted until a special reclamation process called “clean-up” has been run. That makes the files even more scattered around the card.

For the symmetrical block format, we will format the card so that the file management overhead is laid out evenly, or symmetrically, in each block. In other words, the available space in each block for writing the data will be the exactly same for each block and will be maximized. Each block header will contain the information needed for FTL to recognize the card and find out where the files are on the card. A card formatted to the above conditions is just a card with certain number of files each block. However, also notice that the file content is contiguous in each block, i.e., the content of file 2 follows immediately after the content of file 1 and so on. So if there are x files per block, all these x files have their content contiguously stored on the card. The content of file (x+1) starts at the same offset into the next block as the first file into the first block. This is useful when the logged data needs to be contiguous but will span more than one file. The target/logging system only needs to know where the file area starts and it can immediately start to log the data. If we pop the card out and put it into a PC with standard FTL loaded, the logged data will show up in standard DOS files and can be accessed using standard DOS tools.



NOTE:
VBM in first block, FAT in second, and the files are stored randomly.

Figure 1. Normal FTL Formatting



NOTE:
Each block has same size for the files area and the files. The number of files can vary from one across the entire media, to the maximum number of files per block allowed for the particular media.

Figure 2. Symmetrical Block Structure for FTL Pre-Formatting

3.0 FTL FUNDAMENTALS

Traditional block storage devices read and write data in small blocks sized in power of two multiples of 128 bytes (i.e., 200hx). In the case of traditional rotating media disk drives, their blocks are typically 512 bytes. Today’s most popular flash devices, such as Intel’s ETOX™ III FlashFile™ memory parts, use a 64-KB erase block or erase unit. This erase unit must be erased all at once before data can be changed in the block. Thus, for flash, the Erase Unit is typically 20000 hex in size. The Erase Unit is the smallest area that can be reclaimed all at once. Because of the different block size, and of this special characteristic of flash, a translation layer is needed to facilitate communication between the standard OS-level software (such as DOS FAT file system) and flash devices. From the perspective of higher level layers, a block storage device is a contiguous array of blocks which are writable at will, without any regard for the need to first erase the media and certainly without any need to erase an area that exceeds the size of the block being written. The FTL delivers this capability to the higher level software layers by re-mapping requests to write blocks to unallocated or free areas of the media and invalidating the area of the media previously containing the block’s data. The FTL also records where the re-mapped block is physically placed on the media to allow subsequent read accesses to return the data written. In effect, the FTL presents a virtual block storage device to the higher level software layers that reliably manages the logical to physical mapping of blocks or sectors.

3.1 Flash Characteristics

A unique characteristic of flash media is its data content after erasure. If erased, flash media data bytes are all 1’s or all 0’s. Once a flash bit has been set to a value other than its erased state, an entire Erase Unit must be erased to return the bit to its erased state. However, single bits may be set from “erased” state to “programmed” state at any time. FTL use this ability to modify fields in the media control structures.

3.2 Erase Unit Header and Block Allocation Information

For allocation purposes, an Erase Unit is evenly divided into arrays of read/write blocks of equal size (see Figure 3). The size of a read/write block is the same as a virtual block viewed by FAT. At the beginning of each Erase Unit is a Erase Unit Header (see Table 1) which includes specific information about the Erase Unit and global information about the format of the FTL partition. Each Erase Unit also contains allocation information for all of the read/write blocks within the unit.

For each read/write block, a 4-byte value tracks the block’s current state. This is the case for all read/write blocks in that Erase Unit. This section is located right after the EUH and is called Block Allocation Map (BAM). At any point in time, a read/write block in an Erase Unit is either free, deleted, bad or allocated. (see Table 2 and Figure 4). Allocated means the block is been mapped by FTL to be used as virtual block data, VBM pages or replacement pages.

Table 1. Erase Unit Header Fields

Offset	Field
0	LinkTarget Tuple
5	DataOrganization Tuple
15	Number of Transfer Units
16	Erase Count
20	Logical EU Number
22	Read/Write (Sector) Size
23	Erase Unit Size (in Log2 Form)
24	First Physical EU Where Partition Starts
26	Number of Erase Units
28	Formatted Size
32	First Virtual Map Address on the Media
36	Number of Virtual Map Pages
38	Flags
39	Code
40	Serial Number
44	Alternate Erase Unit Header Offset
48	BAM Offset
52	Reserved

Table 2. The Entries of BAM and Their Meaning

Value	Meaning
FFFFFFFF	Free
00000000	Deleted
00000070	Bad
00000030	Control
xxxxxx40	Data of Map Page
xxxxxx60	Replacement

The BAMs for VBM are negative numbered while the BAMs for virtual block data are positive numbered. This is the only way to distinguish between the two. For example, 00000440 is virtual block data number 2 (each block is 200hx) while FFFFFE40 is the last page of the VBM.

The Control Units are read/write blocks used to store the BAM. The number of Control Units depends on the size of the BAM which depends on the ratio of Erase Unit size to the read/write block size.

3.3 The Virtual Block Map

The FTL uses a data structure known as the Virtual Block Map (VBM) to map requests for virtual blocks from higher level software layers to logical addresses on the media. The VBM is an array of 32-bit entries, each of which represents a logical address on the media where a virtual block's data is stored. The virtual block number requested by higher level software layers is used as an index into this array (see Figure 5).

The VBM is subdivided into pages. Each page of the VBM is the same size as a virtual sector of the FAT file system. Since each entry of VBM is 4 bytes, each page holds (virtual blocks/4 number of entries), and from that we can figure out how much virtual space each map represents and how many pages we need to map the whole virtual space.

The flash media is divided into Erase Units. Each Erase Unit is evenly divided into read/write blocks for allocation purposes. Each of these read/write blocks is the same size as the virtual blocks presented to FAT file system.

Space is always reserved on the media to store a VBM large enough to track the allocation of all the virtual blocks on the card. However, when the card is formatted, the FTL may choose to only keep a portion of VBM on the media, and the rest of it can be stored in RAM. The amount of VBM stored on the media is indicated by the FirstVMAddress field of the Erase Unit Header. If the first VMAddress is set to 0, the FTL maintains all of the VBM entries on the media. If the FirstVMAddress exceeds the FormattedSize, none of the VBM entries are maintained on the media by the FTL.

When all or a portion of the VBM is not maintained on the media, it has to be reconstructed in RAM every time a card is re-inserted. It uses the BAM information to fill out the entries of VBM in RAM. Although it uses more system RAM to contain the VBM, this approach does have two advantages. First, it uses less flash as it reduces the virtual block management overhead on the flash media. Second, it can tend to increase performance as the first level virtual sector map lookup occurs from the fast system RAM.

If a VBM entry is all ones, the virtual block does not exist on the media. If it is all zeroes, the logical address of the virtual block is described on a replacement page.

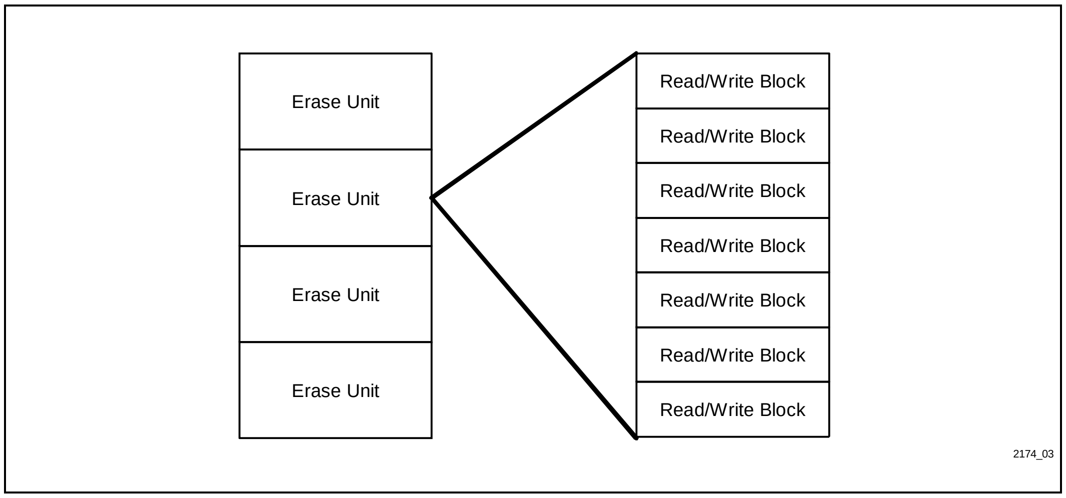


Figure 3. Erase Unit Format

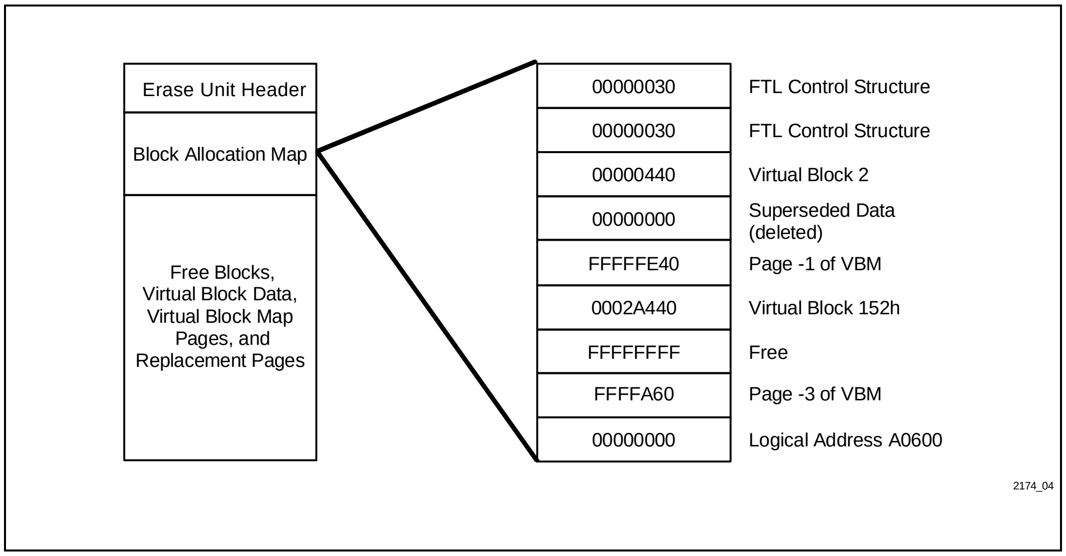


Figure 4. Block Allocation Map Entries

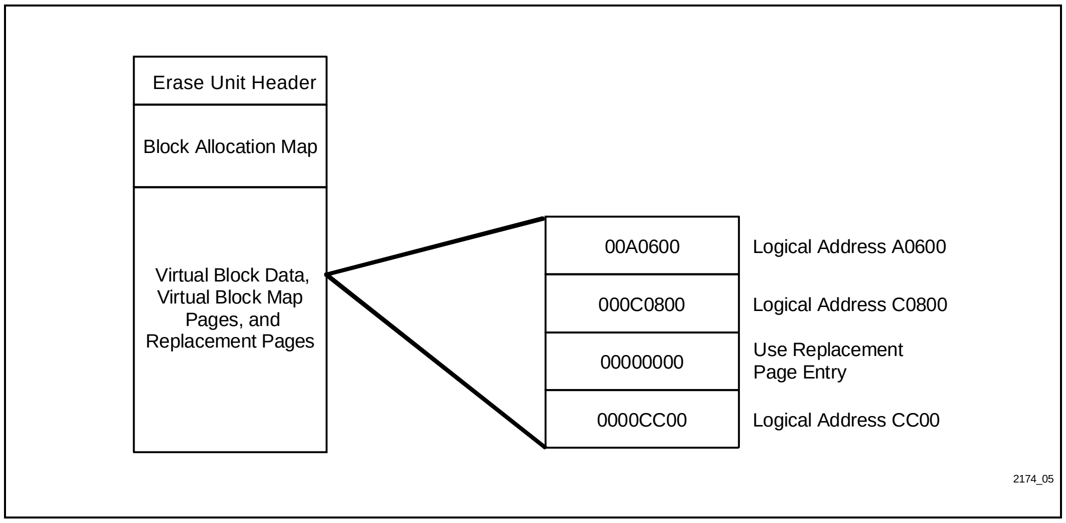


Figure 5. Virtual Block Map Entries

3.4 Replacement Pages

Each page of the VBM may have a replacement page. Values in a replacement page override entries in the original VBM pages. Replacement pages are allocated from free read/write blocks in any Erase Unit. The FTL locates allocated replacement pages by scanning the block allocation information on the media. This scan may be performed when the media is inserted in the host system or when a VBM entry of zero is encountered. Replacement pages cannot be replaced. The block allocation information entry for a replacement page uses the same virtual address as the original VBM page. FTL distinguishes between the two by looking at the last byte of the entry: 40H for VBM pages and 60H for replacement pages.

3.5 Summary

Hopefully this provides a brief description of the standard FTL format. FTL provides a translation layer between a high-level file system such as DOS FAT and flash media. The whole purpose of the mapping is to be able to use a standard, sector-based file system with flash media and not violate flash characteristics.

4.0 SYMMETRICAL BLOCK PRE-FORMATTER

FTL Logger Pre-Formatter is a utility tool which formats a flash card symmetrically so that data can be written to the card at pre-determined addresses for all blocks. It uses FTL to map between FAT and flash media. Map structures are located evenly among the blocks so that each block will be symmetrical.

4.1 The User Interface

The user will be prompted to determine if the card needs to be erased after the utility gets the media information. If the user decides to go ahead and erase the card, the whole card will then be erased and cannot be recovered to its original format. After erasing, the user will be able to choose to either format the entire card as one file, or input the number of files they want to put in each block. If the entire card option is chosen, the utility will then format the card as one file and display the size of the file and where in each block the file pieces start. If the second option is chosen, the user will then be notified the size of each file after the number of files per block is entered. The user can have the option of keeping this number of files or change to another number. After the user has confirmed the number of files per block, the utility will then go ahead and pre-format the card and display the size of each file, number of files per block and where in each block the first file starts.

4.2 The Structure of the Format

As shown in Figure 2, the pre-formatter tries to allocate all the structures as evenly as possible among the blocks. The required structures are Erase Unit Header, Block Allocation Map, Virtual Block Map, and FAT structures which includes Master Boot Record, Partition Boot Record, FAT tables and RootDirEntries. The rest of the space will be used as the files area.

4.2.1 ERASE UNIT HEADER (EUH)

The EUH is identical for every Erase Unit except for the logicalEUN field which assigns a logical counter to each Erase Unit. For convenience, Pre-Formatter assigns each logical unit in the order of its physical location. Also, the NumTransferUnits (spare blocks) and the EraseCount is assigned to one. The EUH still stays at the beginning of each Erase Unit.

4.2.2 BLOCK ALLOCATION MAP (BAM

BAM still exists in each Erase Unit after the EUH. Since the space BAM takes in each Erase Unit is identical, there is no need to balance it among the blocks.

4.2.3 VIRTUAL BLOCK MAP (VBM)

For standard FTL, the VBM is normally put in the first Erase Unit. However for FTL Logger, the VBM is spread evenly among all the Erase Units. The number of pages in each Erase Unit is calculated by dividing the total VMPages by the number of Erase Units (excluding transfer units). The corresponding entries in the BAM are also updated to reflect this allocation.

4.2.4 MASTER BOOT RECORD (MBR)

The MBR is located in the first Erase Unit which has one fewer VBM pages. Since the number of pages per Erase Unit is mostly likely not an integer, there will be some Erase Units with one more page than the others. Therefore, the extra space in the other Erase Units can be used for storing the MBR. In MBR, there is a field which indicates where the FAT partition starts. The PBR is located at the beginning of that partition.

4.2.5 PARTITION BOOT RECORD (PBR)

The PBR is placed on the card in the Erase Unit following the Erase Unit containing the MBR at the same offset. The NumFAT field in PBR is set to 1. Since the user will not be updating the FAT, there is no danger of destroying the FAT in the read/write process.

4.2.6 THE FAT TABLE

Unlike standard FTL which places the entire FAT table in one Erase Unit, the FTL LOGGER spreads it evenly among all the Erase Units. The number of sectors FAT takes is defined in PBR and by dividing that number by the number of Erase Units (excluding transfer units) gives the number of sectors in each Erase Unit that would be used to store FAT. The FAT entries are then updated to point to the files area. Since all the files have the same size and are stored consecutively in virtual memory, the FAT entries can be very easily set up. All the entry numbers in FAT should be consecutive and there should be an equal number of entries in between each pair of end-of-file sign (FFFF).

The remainder of the FAT table should be zeroed out after all the file entries have been filled in.

4.2.7 ROOT DIRECTORY ENTRIES

The size of the Root Directory Entries (RootDirEntries) field and the number of root directory entries is dependent on the number of files chosen. Once this is done, the RootDirEntries field will be divided evenly among Erase Units. They are placed right after the FAT sectors in each Erase Unit.

All the entries in this field are identical except for the name field. The name is assigned as the number of the file, e.g., 00000001.fil, 00000002.fil, etc. The time and date fields are set to predetermined numbers.

After all the file entries are filled in, the rest of the RootDirEntries field is then cleared.

4.2.8 THE FILES AREA

The File Area is maximized by evenly placing the above structures. Since all the files must be the same size, some of the file area may go unused. For example, if there are 200 sectors left in each Erase Unit and the user wants 15 files per block, each files will then be allocated 13 sectors and there will be five unused sectors left in each unit. In this case, the files area are allocated from bottom up, i.e., the last $15 \times 13 = 195$ sectors will be allocated for files storage. The corresponding entries in the BAM are updated to reflect the allocation of these files sectors.

4.2.9 REPLACEMENT PAGES

There is no need to implement replacement pages for FTL LOGGER since the files will be read-only on the host afterwards and no updates are needed for VBM or BAM entries.

4.3 Pre-Formatter Summary

Because of the nature of FTL mapping, we are able to relocate the structures such as VBM, PBR, MBR, FAT and RootDirEntries. We arrange them in such a way so that the space leftover for file storage is maximized given the number of files. After FTL Logger pre-formatting, there is an equal amount of space left in each Erase Unit where the file data can be written. The size to skip, the size of each file and the number of files per block will be displayed for the user after the formatting.

The reference code can be found in Appendix A.

5.0 EMBEDDED SYSTEM REQUIREMENTS

The embedded system does not need to understand the structures of FTL or FAT. All it needs to know is the starting address at which to write and the data will be in its proper place.

5.1 Writing to the Formatted Card

Since all of the overhead space created by the formatter exists in a contiguous region of flash, the embedded system needs to know how to avoid this forbidden region. This can be done in two ways, depending on the read/write mechanism of the embedded system.

5.1.1 BYTE-WRITE PROTECTION ALGORITHM

The first way is to use a Byte-Write Protection Algorithm which writes to the card byte by byte. Typically, a generic write algorithm would look something like Figure 6. When the embedded system decides to write a byte, it obtains the current address, calls the flash programming algorithm, shown in Figure 7, and avoids the region of space at the top of the block reserved for the overhead. If the current address is less than the overhead size, skip over the overhead region, if not, then just write the byte. This procedure requires pre-defining the following variables: card size, block size, overhead size. Of course, we always have to subtract the address by the block base first in order to get its relative position into each Erase Unit.

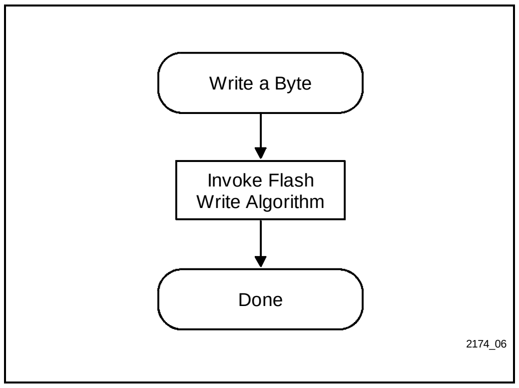


Figure 6. Conventional Byte-Write Process

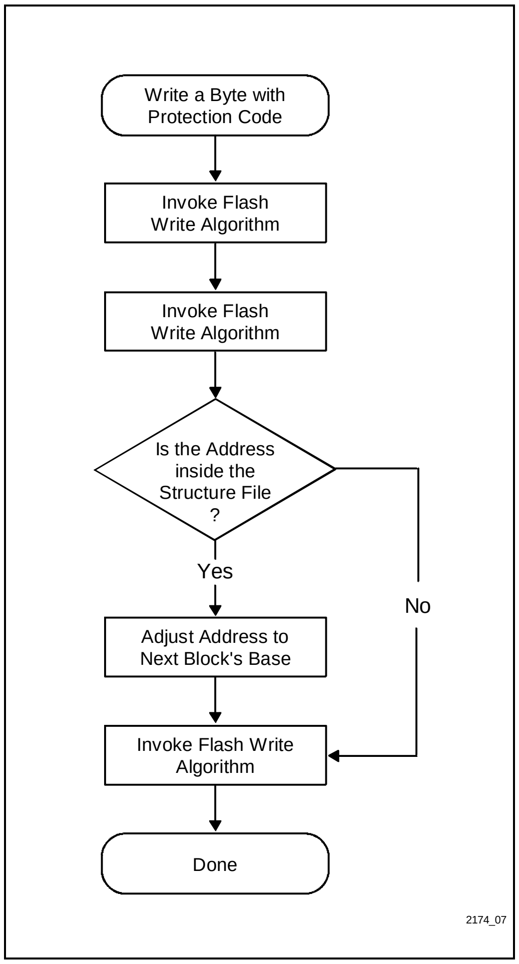

Write a Byte with Protection Code
Invoke Flash Write Algorithm
Invoke Flash Write Algorithm
Is the Address inside the Structure File ?
Yes
No
Adjust Address to Next Block's Base
Invoke Flash Write Algorithm
Done
2174_07

Figure 7. Byte-Write Protection Algorithm

5.1.2 ALTERNATE WRITING METHOD

Another way to write to the formatted card is to set certain bits on the card to tell the target system the information that needs to be otherwise pre-defined. This is the method currently used by FTL Logger. First, the card size and block size can be determined from the fields inside the EUH. From the FTL specifications, there exists a reserved field at the end of the EUH which is currently unused. This field is now used for storing file size, number of files, and the overhead size. There is also a byte that would indicate if the card has been formatted as one entire file or certain number of files per block. The exact location of these fields are as follows:

Files per block formatting:

34–37 H	size to skip
38–39 H	# of files/block
3A–3D H	size of each file
3F H	FF

One entire file formatting:

34-37 H	size to skip
3F H	F0

NOTE:

These fields are not used by FTL currently, but in case FTL decides to use them later, we can still set these fields after the reserved field and increase the BAMOffset field in the EUH to accommodate these fields.

Now the target system just has to look at these fields and use these values in place of pre-defines. The advantage with this method is that these fields are set at the time of pre-formatting, so they exist for any card and there is no need for the target system to pre-define card information every time.

```
             ;Values needed to be predefined

             define Card_Size = Card Size;

             define Block_Size = Block Size-1

             define overhead   = size of overhead

             assume addr       = address in the card about to be written to
               :
      protection check:

           mov   addr1,addr          ;grab the current address
           and   addr1, block_Size   ;get the offset of that adress in a block
           sub   overhead, addr1     ;subtract address by the overhead size
           js    write_flash         ;if the overhead is smaller, then we are safe
           mov   addr, overhead      ;otherwise, set the address to after the overhea

      write_flash

           call  write_flash_byte
           :
           :

           :
```

2174_08

Figure 8. Protection Code for Byte-by-Byte Writing

5.2 Reading from the Formatted Card

After the card has been formatted and the data has been logged to the flash card, we can read the files with any standard FTL-based PC. The file data will be contiguous, i.e., the content of file 2 follows the content of file 1.

5.3 Erasing and Modifying Files

Since the card is pre-formatted and all the space on the card has been allocated in a certain fashion, no erasing or modifying of files on the card is allowed. The card does not have the standard FTL format, it is just formatted in such a way that FTL will understand and be able to recognize the files. We can only use a logger to fill in the data, but cannot use edit or any other file manipulating functions besides just reading the data. Once the files have been modified or new files have been saved onto the card, the FTL Logger format will be lost.

6.0 REFERENCE CODE FOR FTL LOGGER

The most important initial function upon program execution is the compatibility module which checks if the card is an Intel Flash product. It exits right away if it is not. The next function is to obtain the card's geometry: size, number of blocks, size of block, Jedec, etc. This information will be used later for calculating the overhead. Then the program prompts the user that the card is going to be erased and the user has the option of escaping out or continuing. Next, the program will ask the user for an option of formatting the entire card as one file or number of files per block. If the latter option is selected, the program asks for the number of files per block and then calculates the overhead it would take to accommodate these files and asks the user if he wants to go ahead with this file size. Once the user has confirmed his option, the program then goes ahead and formats the card according to the user's option. For both kinds of formatting, the program writes the structures onto the card in the following order: EUH, BAM, VBM, MBR, PBR, FAT, RootDirEntries and Files Area. Every time a structure is put onto card, the corresponding fields in the BAM are also set.

After the formatting has been completed, the program will display the file size, number of files and the range in each block that data can be logged. As mentioned before, the card and format geometry information will also be in the reserved field of the EUH which would be useful for the embedded side.

If the card is inserted into a system using a standard FTL driver, a regular “dir” command will show that the card has the number of files the user requested with the names of the files in increasing order.

7.0 ADDITIONAL INFORMATION

7.1 Revision History

Number	Description
-001	Original Version

7.2 Glossary of Terms

Block Allocation Map (BAM)	An FTL control structure that is used to store Erase Unit block allocation information when hidden areas are not used to store this information.
Erase Unit	The area of flash media that is handled as a single erasable unit by the FTL. All Erase Units in an FTL partition are the same size.
Erase Unit Header (EUH)	An FTL data structure that describes an Erase Unit.
FAT	Acronym for File Allocation Table. It is the primary file system DOS uses.
FTL	Abbreviation for Flash Translation Layer.
Logical Address	An address based on accessing the media in Logical Erase Unit order.
Partition	An integral number of contiguous flash erase components formatted in a specific way.
Read/Write Block	A subdivision of an Erase Unit. Used by the FTL to track media allocation. The FTL maintains the allocation state of each read/write block.
Replacement Page	Values in a replacement page override values in the original page of the Virtual Block Map or BAM. They are used whenever a file is added, deleted, or modified.
Transfer Unit	Also called spare block. An Erase Unit reserved for block/drive reclamation process.
Virtual Address	The address recorded in a read/write block's allocation information representing where the stored data appears in the virtual image presented to the host system.
Virtual Block	The unit of information used by the file system layer above the FTL to read and write data to the media. It is also called a read/write block or a sector.
Virtual Block Map (VBM)	An array of 32-bit entries used to map a virtual block number to a logical address on the media. Space is always reserved on the media to store the entire VBM. The FirstVMAddress field describes how much of the VBM is maintained on the media by the FTL.

Filename:	292174_1.DOC
Directory:	C:\TESTDOCS\DOCS
Template:	C:\WINDOWS\WINWORD6\TEMPLATE\ZAN____1.DOT
Title: E	
Subject:	
Author:	Mary Ann Hooker
Keywords:	
Comments:	
Creation Date:	08/04/95 2:03 PM
Revision Number:	34
Last Saved On:	12/06/95 10:12 AM
Last Saved By:	Ward McQueen
Total Editing Time:	378 Minutes
Last Printed On:	12/06/95 10:13 AM
As of Last Complete Printing	
Number of Pages:	14
Number of Words:	4,231 (approx.)
Number of Characters:	24,119 (approx.)