Fedit Plus

A File And Disk Editor

MacMaster Systems
108 E. Fremont Avenue, Suite 37
Sunnyvale, CA 94087-3201
(408) 773-9834

Table of Contents

An Overview Of Fedit Plus

Fedit Plus is a file and disk edit utility program for the Macintosh patterned after the ZAP type programs available on many other systems. It is intended to be a powerful and easy to use utility for use by average to highly technical users. It is not intended for the uninitiated user. The program allows the user low level, direct access to disk volumes for both reading and updating.

Fedit Plus is also designed to operate with hard disk drives connected to the serial, external disk or SCSI ports. Efforts have been made to test Fedit Plus with as many varieties of hard disk as possible, but no warranty can be provided that it will operate with all disks. In particular, hard disks connected through Appletalk may experience some difficulties, particularly when operating in volume mode.

Some words of caution are in order. Careless use of this program can seriously damage a file or disk. If you need to modify data on a disk, and you can do so with a more secure program, then you should not use Fedit Plus to make the modifications. Fedit Plus has no protection to prevent you overwriting critical system areas of the disk such as the Volume Allocation Table and the disk directory, and you cannot undo changes once they have been written to disk.

I would suggest that you follow these two strategies when using Fedit Plus:

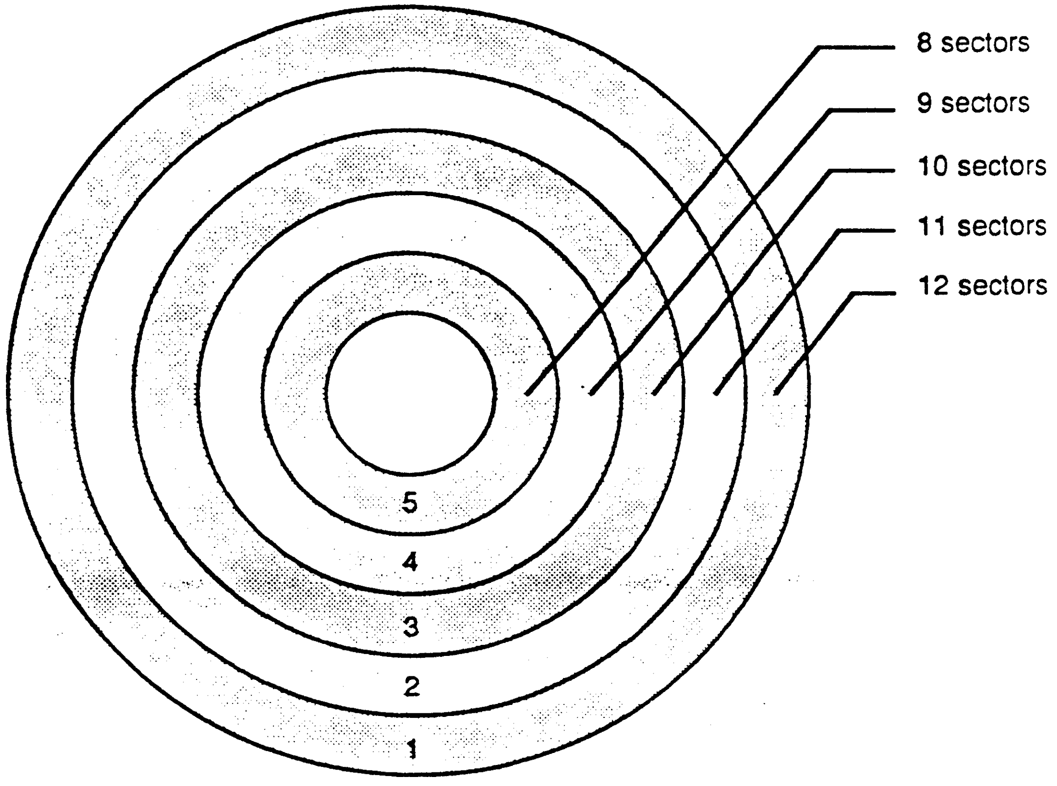
- Make absolutely sure that you have at least one backup copy of the disk you are intending to modify. This may seem like an elementary precaution, but it is surprising how often it is omitted. It really does seem that there is no substitute for experience when learning this lesson.

- When writing modified data to a disk, always check your changes twice. Then check them again. You cannot undo changes that you make after they have been written to disk. If you are unsure that the changes you are making are correct, then don't commit them to disk. It isn't only marriage that presents an opportunity to do something in haste and repent at leisure.

An Overview of the Physical Disk Structure

The diagram below shows the basic structure of a Macintosh diskette. The Macintosh disk drives are unusual because they access different portions of the disk at different motor speeds (this is the cause of the varying pitch of the single sided disk drive motor while accessing diskettes).

Each diskette is formatted into 80 tracks, and each track is divided into a number of sectors. The sector count in any given track varies according to the position of the track on the disk. Since the outer tracks are longer, they can contain a larger number of sectors. Because the speed of the disk surface passing under the disk head must be kept within fairly close tolerances, it is necessary to vary the rotation speed of the disk according to which track is being accessed. The speed is lower for the outside tracks getting progressively higher as the disk head moves towards the center of the disk.



To simplify matters somewhat, the diskette is divided into 5 bands. Each band is 16 tracks wide and has a varying number of sectors in each track as shown in the diagram. By convention, track 1 is at the outside of the diskette and track 80 is closest to the center.

Double sided disks follow the same principles as above except that both sides of the disk are used to store data. Head 0 (corresponding to the only head on a single sided disk) is on the underside of the disk as it is presented to the drive, and head 1 is on the top side of the disk.

Each sector on a floppy diskette contains 512 bytes of data plus 12 bytes of tag information. The tag data can be used for recovering data accidentally erased or lost on damaged disks. The format of this data is described later in this manual.

From the above data we can draw the following table:

Band	Sectors/Track		Sectors	
	SS	DS	SS	DS
1	12	24	0-191	0-383
2	11	22	192-367	384-735
3	10	20	368-527	736-1055
4	9	18	528-671	1056-1343
5	8	16	672-799	1344-1599

Thus a total of 409,600 bytes can be stored on a single-sided diskette and 819,200 bytes can be stored on a double sided diskette.

Not all the data area on a diskette can be used for data. Specific portions of the disk have been put aside for the data structures that enable data to be stored and retrieved from the disk. These structures are discussed later in this manual.

Hard Drives

Hard disks come in many varieties from a number of different manufacturers. They vary in size from 5 megabytes to 125 megabytes or larger. They can be connected to your Macintosh through a serial port, the external disk port, the SCSI port of a Macintosh Plus or through an AppleTalk network. Fedit Plus supports all these disk types except for disks connected through AppleTalk.

The physical layout of a hard disk volume varies according to size, type and manufacturer. There is no standard layout. One of the differences between hard disks and floppy disks is the treatment of tag bytes. Most manufacturers do not support tag bytes on each sector. The Apple Profile and Widget hard disks on the Macintosh XL support 12 bytes of tags, and the Hard Disk 20 supports 20 bytes of tags. The layout of these tag bytes is described in detail later in this manual.

Hard disks are often segmented into a number of logical volumes. Each volume has its own directory of files and its own space to manage. Each of these volumes is considered a separate volume by the Macintosh operating system and by Fedit Plus.

Logical Structure of File Volumes

When the Macintosh was introduced, each disk volume was organized as a number of files with no particular structure. The disk directory consisted of a number of files in a simple unordered list. When hard disks were introduced this organization was found to be inadequate; different packages would use files of the same name but with different content, the time taken to search for files became excessive, and so on. As a result Apple introduced a hierarchical file system with the files in it organized rather like a tree structure. This was given the imaginative name of the "Hierarchical File System" to distinguish it from the orginal "Macintosh File System" with flat directories. These disk structures are described below. The terms HFS and MFS are used to describe the different systems.

The two systems share a number of concepts. Each disk volume contains a file directory that contains information describing the files on the disk. The volumes are formatted into sectors each containing 512 bytes of data. Files are stored on disk in units called allocation blocks, and each allocation block contains a multiple number of sectors. The size of an allocation block is determined at the time the volume is initialized, and is fixed for the lifetime of the volume.

The Macintosh File System

This is the original system of disk organization with flat directories. Older Macintosh systems only know about this type of organization, and initialize all disks with an MFS directory structure. Newer systems which have the 128K ROM version of the file manager will only initialize single sided diskettes with an MFS directory. Under this system, the placing of files into folders has no effect on the actual disk structure.

As mentioned earlier, each single sided disk holds 409,600 bytes of information, but not all this area can be used for data. Specific portions of the disk are put aside for the data structures that enable data to be stored and retrieved from the disk in an orderly manner. These areas include:

Bootstrap loader	2 sectors	1024 bytes
Volume Access Table	2 sectors	1024 bytes
Disk directory	12 sectors	6144 bytes
Alternate Master directory	2 sectors	1024 bytes

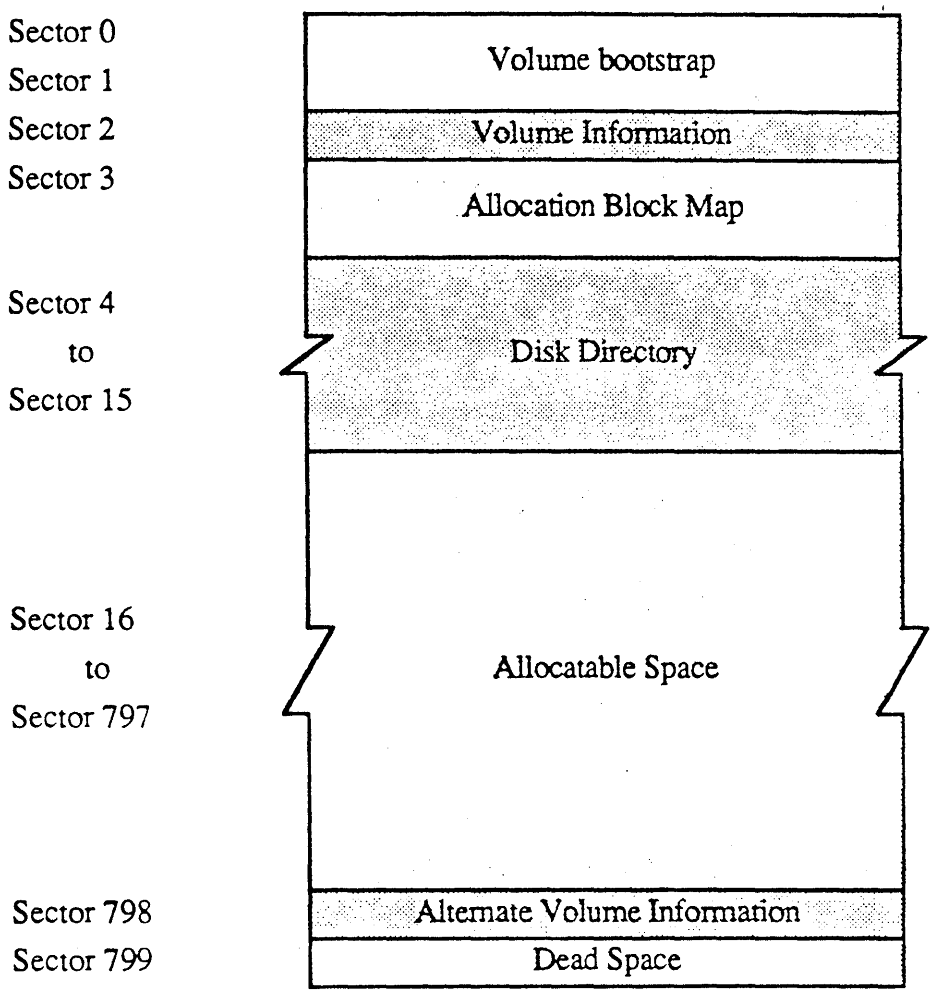
After the spaces for these areas is deducted, about 400,000 bytes is left for data storage. In the Finder prior to version 5.0, the references to space used and space available were also in terms of "K Bytes" where 1K bytes was equal to 1000 bytes. Since version 5.0 this has been changed to use the more usual value of 1024 bytes for 1K.

The volume map, overleaf, shows the various areas of a single sided diskette, and where they are normally located. Some areas (volume bootstrap and the volume information) have fixed locations, but other areas (like the disk directory) are capable of being moved to another place on the disk.

The diagram below shows the layout of a single sided MFS diskette.

MFS Volume Map

Single Sided Diskette



MFS Volume Bootstrap

Disk sectors zero and one contain the boot code if the volume is capable of bootstrapping the system. These sectors are the same for both MFS and HFS and are described later in this manual.

MFS Volume Information Table

The volume information table is contained in the first 64 bytes of sector 2 on every properly initialized volume. It is written to the disk when the volume is initialized.

The layout of this table is as follows (addresses are in hexadecimal):

0000	$D2D7 - the ASCII equivalent is "RW" with the high bit set on, in good Apple 2 tradition. This stands for Randy Wiggington. These two bytes serve as a signature to identify this block.
0002	The date and time of volume initialization in the standard format; the number of seconds since midnight on January 1st., 1904 (4 bytes).
0006	The documentation describes this field as the date and time of the last backup. Actually, it is the date and time that the volume was last changed or updated (4 bytes).
000A	The volume attributes (2 bytes). If bit 14 is set, this volume is copy protected.
000C	The number of file entries in the disk directory (2 bytes).
000E	The starting sector number for the disk directory. Normally equal to 4 on a standard diskette (2 bytes).
0010	Number of sectors that the file directory occupies (2 bytes).
0012	Total number of allocation blocks on the volume (2 bytes). Allocation blocks are described below.
0014	Size of each allocation block in bytes (4 bytes).
0018	Allocation clump size in bytes (4 bytes). See below.
001C	Sector number of first sector in the volume data area (2 bytes).
001E	The next unused file number (4 bytes). See below.
0022	The number of unused allocation blocks on the volume (2 bytes).
0024	The length of the volume name (1 byte).
0025	The volume name in ASCII characters (27 bytes).

In talking about the volume information, it is necessary to describe in more detail how sectors are allocated to files on a volume.

The allocatable space where the file contents are stored is divided into a number of allocation blocks. Each allocation block is a multiple number of 512 byte sectors. On a standard MFS floppy disk the allocation block size is 1024 bytes or 2 sectors. For reasons described below the allocation size on MFS hard disks is usually considerably larger (8 to 20 sectors are typical sizes).

An allocation block is the minimum amount of disk space that can be allocated to a file. All space is allocated and deallocated to files in terms of allocation blocks exclusively. The volume clump size specifies the amount of space (in bytes) for file allocation. The clump size must be a multiple of the allocation blocksize. When a file is first written, MFS allocates an amount of space equal to the clump size to that file. If the size of the file becomes so large that this space is insufficient then another space of clump size bytes is allotted to the file. When the file is eventually closed, MFS will deallocate any allocation blocks not used at the end of the file.

An example might help. The usual size for the clump size is 8K bytes (16 sectors), and for the allocation blocksize 1K bytes (2 sectors). When an application program opens an output file, the file will be allocated 16 sectors on the disk. Assume that the application then writes 700 bytes to the file and closes it. The allocation blocks that contain data (in this case only the first) will remain allocated to the file, but the File Manager will deallocate the space for the remaining 7 allocation blocks that were unused. The file entry for that file will show a physical end of file at byte 1024 (the first byte after the end of the allocation block), and a logical end of file after the 700 bytes that were written by the application.

Unfortunately for us humans (or at least non-computers), a number of fields are expressed in terms of allocation blocks rather than absolute sector numbers, so it may occasionally be necessary to understand how to translate one to another (although Fedit Plus will convert the fields in the volume and file headers for you). The relationship is:

STNR = (BKNR - 2) * ABSZ DIV 512 + FSTN

where STNR is an absolute sector number,
BKNR is an allocation block number,
ABSZ is the size of each allocation block in bytes,
FSTN is the number of the first sector in the data space.

The last two quantities are taken from the volume information table.

When a file is created on a volume, that file is given a file number. The principal use for the file number on MFS volumes is in file and volume reconstruction. You may recall from earlier in this discussion that the tag data on each sector contains the file number of the file to which it belongs. File numbers are never reused, and since 32 bits have been allocated to hold them, it is unlikely that they will ever wraparound.

MFS Volume Allocation Map

The Volume Allocation Block Map starts at byte 64 on sector 2 of the volume (immediately after the Volume Information Table) and continues for as many sectors as are required. There is one entry in the block map for each allocation block on the volume.

Each entry is 12 bits in size. It indicates whether the allocation block is used or unused. If the block is in use, the value is a pointer to the next allocation block in the file. A value of 1 indicates that this is the last allocation block in the file, and a zero value indicates that the block is unused. Because of the special values attached to zero and one, the first allocation block on a volume is number 2.

As an example, if you assume that a file is resident in allocation blocks 5, 9 and 12 of a volume, and that there are no other files present. The first few entries in the volume allocation block map would look like this:

```
0  0  0  9  0  0  0  12  0  0  1  0  0  0  0
```

The file entry in the disk directory has a field pointing to the first entry in the block map for the file. In the above case, it would have a value of 5.

The size of the volume allocation map is dependent on the number of allocation blocks on the disk. For example, a 5 megabyte Profile disk initialized under MacWorks has a four sector map using an allocation block size of 4K bytes (8 sectors). This will give an average wastage of 2K bytes per file. If the allocation block size had been set to 10K bytes, the volume allocation map would be 2 sectors long and there would be an average wastage of 5K bytes per file.

MFS Disk Directory

The disk directory on standard volumes is located immediately after the last sector of the Volume Allocation Map at sector 4. There is also a pointer in the Volume Information Table to the starting sector of the directory.

The directory contains one entry for each file on the volume. Each entry consists of a 50 byte fixed length portion plus a string for the filename. There are a variable number of entries in a sector, but no entry ever crosses a sector boundary. If a file entry will not fit in a sector it is placed into the next sector. If all sectors are full, then a "directory full" error is returned by the File Manager.

On a standard disk there 12 sectors allocated to the file directory, and depending on filename length, each sector will probably contain 6 to 9 file entries. This suggests a maximum of 72 to 108 files on a diskette volume. In the unlikely event of you getting a directory full condition, it is possible that you may be able to overcome it by reducing the size of the filenames for entries already present on disk. Hard disks have more space allocated for the file directory and can accomodate many more files.

There are two possible divisions in a file, called the data fork and the resource fork. These can be thought of as two completely separate files that share a common filename. Either or both of the forks may be present for any file. When an application program opens, reads or writes a file it will usually be dealing with the data fork. A program can also open the resource fork, but will use a slightly different command to do so. Both forks cannot be open at the same time.

From the File Manager point of view, the main difference between the two forks is their organization. The format of the resource fork is closely defined by the Resource Manager, and the types of data are usually well defined also - things such as menus, fonts, icons and dialogs - all of which are designed to be accessed through the Resource Manager. On the other hand, the data fork has no defined structure and is only accessible through the File manager.

The format of each file directory entry is as follows:

0000	File system attributes byte. Bit 7 is always set to indicate a valid directory entry. Bit 6 is set if the file is copy protected (1 byte).
0001	Version number of the file. This field is unused and should always be set to zero (1 byte).

0002	The file type of the file. This is a four character field with values such as "APPL" for an application, "TEXT" for text file (4 bytes).
0006	The creator of the file. This is also a four character field. The values in this field are purely arbitrary and are used for matching applications to documents belonging to that application (4 bytes).
000A	Finder attributes word. This field is described in detail below (2 bytes).
000C	The location of this file on the desktop. Used only by the Finder (4 bytes).
0010	The folder within which this file resides. Used only by the Finder (2 bytes).
0012	The file number of this file. Each file on the disk is given a file number which is unique to that file. This number is present in the tag field of all sectors belonging to the file (4 bytes).
0016	The number of the first allocation block in the data fork of this file. If this field is zero, this file has no data fork (2 bytes).
0018	The logical end of file for the data fork. This is a count of the number of valid bytes between the start of the fork and its logical end of file (4 bytes).
001C	The physical end of file for the data fork. This is a count of the number of bytes on disk allocated to this fork of the file. It is always a multiple of 512 bytes (4 bytes).
0020	The number of the first allocation block in the resource fork of this file. If this field is zero, this file has no resource fork (2 bytes).
0022	The logical end of file for the resource fork. This is a count of the number of valid bytes between the start of the fork and its logical end of file (4 bytes).
0026	The physical end of file for the resource fork. This is a count of the number of bytes on disk allocated to this fork of the file. It is always a multiple of 512 bytes (4 bytes).
002A	The timestamp when this file was created. This is held in seconds since midnight on the 1st. January, 1904 (4 bytes).
002E	The timestamp when this file was last modified. This is held in seconds since midnight on the 1st. January, 1904 (4 bytes).
0032	Length of file name (1 byte).
0033	Characters of the file name (1 to 63 bytes, variable length).

The file entry is required to start on a word boundary, so there may be one additional byte after the file name, and before the start of the next file entry.

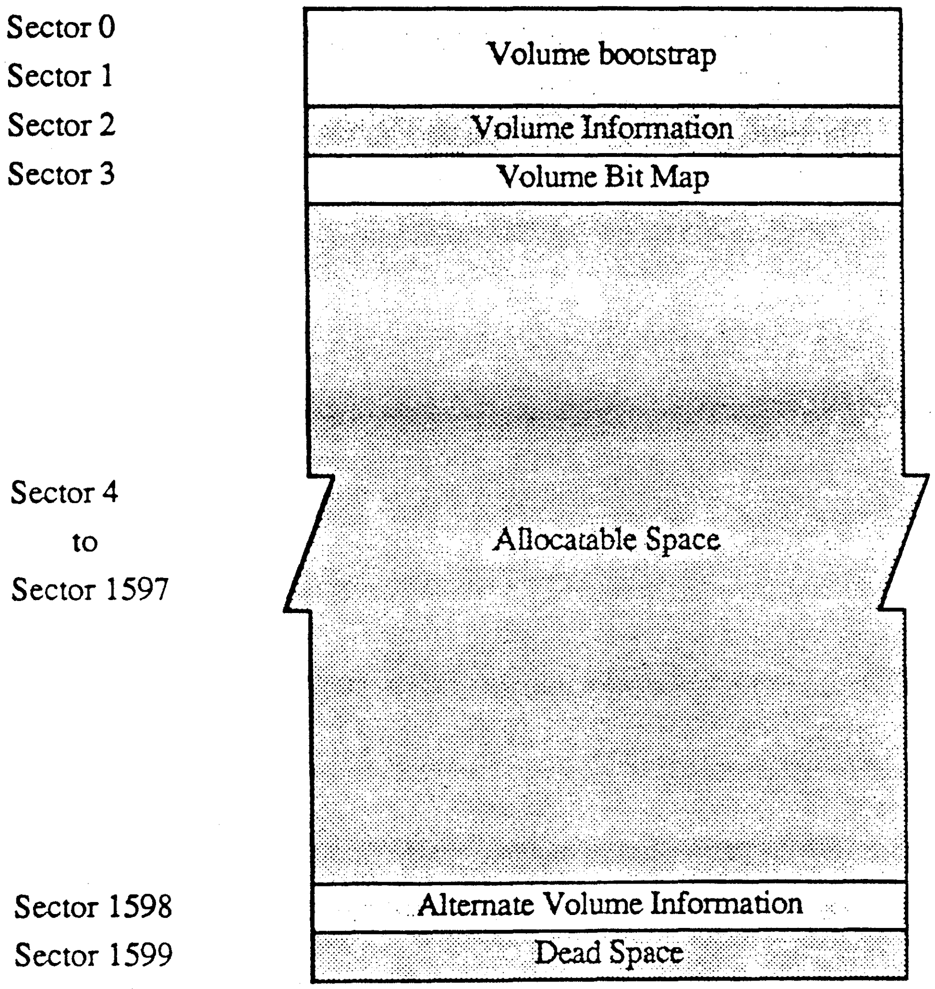
Alternate Volume Information and Dead Space

The penultimate sector of the volume is used to store a second copy of the volume information block in its virgin state when the volume was created. This alternate block is not updated, but can be useful for indicating the default sizes for various data areas during volume recovery.

The last sector on the volume is never used by the file system.

The Hierarchical File System

This file system is used for double sided disks, hard disks connected though the SCSI port and the Apple Hard Disk 20. Some disks from other manufacturers support HFS, and some do not. There is at least one variety of hard disk where a single HFS volume can have number of MFS volumes within it's data space.



The diagram above shows the layout of a standard double sided floppy disk. As with MFS, an HFS volume can be divided into several parts:

- Volume Bootstrap
- Volume Information Block
- Volume Bitmap
- Allocatable Space
- Alternate Volume Information and Dead Space

HFS Volume Bootstrap

Disk sectors zero and one contain the boot code if the volume is capable of bootstrapping the system. These sectors are the same for both MFS and HFS and are described later in this manual.

HFS Volume Information Block

Sector number 2 on the volume contains the data about the volume that has to be saved while the volume is dismounted (not in use). When the volume is mounted, this data is read into memory.

The first 64 bytes of this information is very similar to the MFS Volume Information Table described previously. The first two bytes are used to distinguish the volume type; $D2D7 for MFS or $4244 for HFS.

The layout of the sector is as follows:

0000	$4244 - the ASCII equivalent is "BD" which stands for Big Disk.
0002	The date and time of volume initialization in the standard format: the number of seconds since midnight, January 1st., 1904 (4 bytes).
0006	The date and time that the volume was last modified; the number of seconds since midnight on January 1st., 1904 (4 bytes).
000A	The volume attributes (2 bytes). If bit 14 of this word is set, this volume is copy protected. Bit 8 is the clean un-mount bit which is set by using the Finder Shutdown command. If this bit is set on boot, less checking of volume integrity is performed by the system and the boot is performed much faster.
000C	The number of file entries in the root directory (2 bytes).
000E	The sector number of the first sector in the volume bitmap. Normally equal to 3 (2 bytes).
0010	New file allocation pointer (2 bytes).
0012	Total number of allocation blocks on the volume (2 bytes).

0014	Size of each allocation block in bytes (4 bytes).
0018	Allocation clump size in bytes (4 bytes). See below.
001C	Sector number of first sector in the volume data area (2 bytes).
001E	The next unused catalog node identifier (4 bytes).
0022	The number of unused allocation blocks on the volume (2 bytes).
0024	The length of the volume name (1 byte).
0025	The volume name in ASCII characters (27 bytes).
0040	The date and time that this volume was last backed up expressed as the number of seconds since midnight on January 1st., 1904 (4 bytes).
0044	The sequence number of this volume in a set of backup disks (2 bytes).
0046	The number of writes performed to this volume (4 bytes).
004A	The clump size of the extents tree file (4 bytes).
004E	The clump size of the catalog tree file (4 bytes).
0052	The number of directories (folders) in the root directory (2 bytes).
0054	The total number of files in all directories on this volume (4 bytes).
0058	The total number of directories on this volume (4 bytes).
005C	Finder information for this volume (32 bytes). The first four bytes contain the directory identifier of the folder that contains the System file - the "blessed" folder. The second four bytes contain the directory identifier of the directory that contains the startup application if it is not in the blessed folder.
007C	The size of the volume cache (2 bytes).
007E	The size of volume bitmap cache in memory (2 bytes).
0080	The size of the common cache for the volume in memory (2 bytes).
0082	The length of the Extent B-Tree (4 bytes).
0086	The extent record for the Extent B-Tree (12 bytes).
0092	The length of the Catalog B-Tree (4 bytes).
0096	The first extent record for the Catalog B-Tree (12 bytes).

HFS Volume Bitmap

The volume bitmap is stored on disk immmediately after the volume information block. The size of the volume bitmap is dependent on the number of allocation blocks on the disk. On a standard double sided floppy diskette it occupies one sector.

The bitmap is organized as a series of bits, one bit for each allocation block on the volume. The most significant bit of the first word of the first sector corresponds to the first allocation block, the next bit corresponds to the second block, and so on. If an allocation block is not in use by a file then its allocation bit is set to zero, and if the block is in use the bit is set to one.

HFS Data Space

The space from the end of the volume bitmap to the end of the disk is available for data storage. All data files are placed in this area as are the Extent and Catalog B-Trees which describe the volume structure.

The B-Tree structures are described later in this manual.

Alternate Volume Information and Dead Space

The penultimate sector of the volume is used to store a second copy of the volume information block in its virgin state when the volume was created. This alternate block is not updated, but can be useful for indicating the default sizes for various data areas during volume recovery.

The last sector on the volume is never used by the file system.

The Structure of HFS Data Files

Unlike MFS, the disk directory does not occupy a specific portion of the disk but is contained in two files in the allocatable space of the volume. The Extents B-Tree file is used by the system to keep track of file mapping on the volume, a function performed by the volume allocation map under MFS. The Catalog B-Tree file is used to maintain the hierarchical file structure on the disk and corresponds to the disk directory under MFS.

Every file on the volume has an entry in the catalog file. As with MFS, each file can contain a data fork, a resource fork, or both, or neither. The last case is an empty file which needs no further description. Each fork can be considered as a separate file, with no relationship between them. Apart from the restriction that both forks cannot be open at the same time, HFS considers them as independent entities on the disk that share a common file entry in the catalog.

The primary difference between the two forks is their organization. The format of the resource fork is defined by the Resource Manager, and the types of data are usually well defined also - fonts, menus, icons and dialogs, for example - which are designed to be accessed through the Resource Manager. The data fork has no defined structure and is only accessible through the File Manager.

Each fork in a file will consist of one or more extents. An extent is a contiguous area on disk that is allocated to a file, and is specified by a starting allocation block number and a length field indicating how many allocation blocks are in the extent. If a file is contiguously on disk then it will have a single file extent. Access to a file is considerably faster if all the sectors in the file are contiguous, so the fewer extents in a file the better.

As files are allocated space when required, and several files can be written simultaneously, the disk might become very fragmented if space was allocated a single block at a time. The concept of a file clump was introduced to reduce this possibility. Space is allocated to a file not in terms of allocation blocks but in file clumps. A clump is a group of contiguous allocation blocks, and therefore the clump size is always a multiple of the allocation block size. Since space is assigned to a file in clumps, the worst fragmentation of a file is into clump size pieces. To reiterate, the object is to reduce the number of file extents.

The default size of a file clump is determined from the volume information block described above. An application can also specify the clump size to be used for an individual file when the file is created.

The following strategy is used when searching for space to allocate to a file. When a block of a certain size is requested by an application, it is first rounded up to the next multiple of the clump size. The volume bit map is scanned forward from the block specified by the new file allocation pointer (in the volume information block) until a free area is found that is large enough to accomodate the requested block. The block is allocated at the start of the located area and the rest of the area is left free. The new file allocation pointer is updated to point to the location where the scan finished. This means the next scan for space will start where this one left off. The scan is unaffected by files that are deleted from lower disk addresses (until the new file pointer wraps around), and disk fragmentation is kept fairly low.

If the requesting application asks for a contiguous area, and no area large enough is found, an error is returned to the program. However, if a contiguous area was not requested then a number of passes may be made though the volume bitmap to locate a number of smaller spaces that will satisfy the request.

If an application requests extra space for an existing file, an attempt is made to extend the last file extent already allocated as far as is permitted by the availability of contiguous free allocation blocks. If this does not provide enough space to satisfy the request then a search is made from that point to locate the further space required.

HFS File Numbers

Each file and directory on an HFS volume is assigned a number called the Catalog Node Identifier (CNID) or file number. The CNID for a new file or directory is determined by adding 1 to the last CNID that was allocated. The next CNID to be allocated is held in the volume information table in sector 2. When a volume is created, this number is set to an initial value of 16.

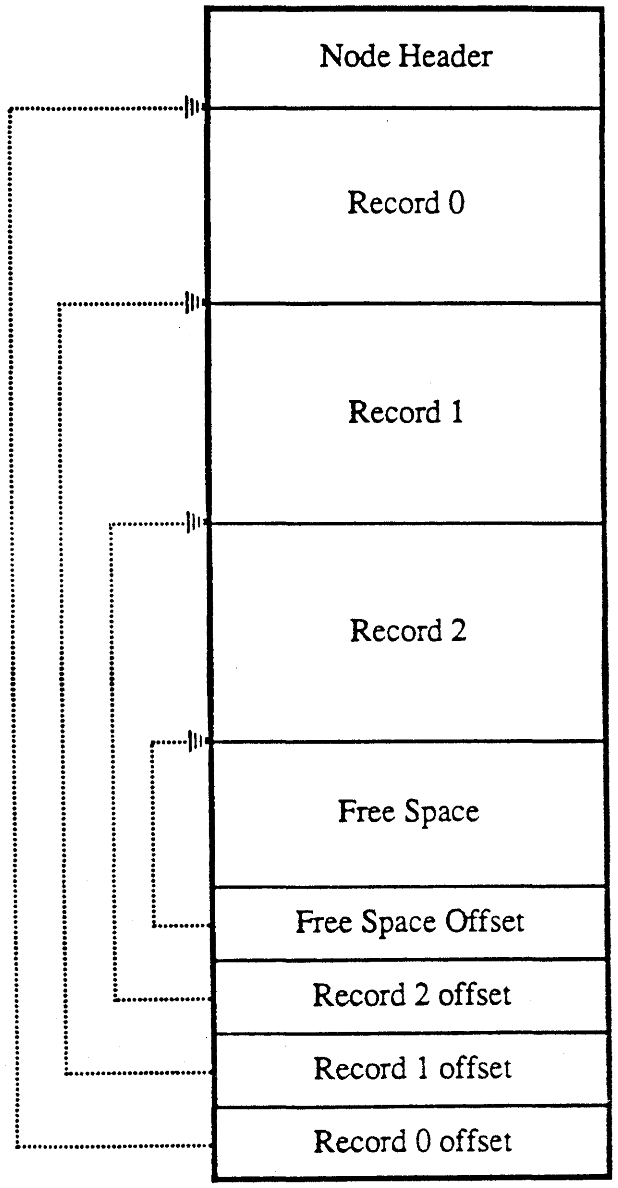
System nodes are allocated fixed node numbers below 16 as follows:

1	The volume bitmap
2	The root directory of the volume
3	The Extents B-Tree file
4	The Catalog B-Tree file.

The Structure of HFS B-Tree Files

File extents and the file catalog are maintained by two special HFS files which are organised as B-Trees. B-Trees are based on the concept of a hierarchical tree structure. Each record in the file consists of two portions; a key that is used to locate the record, and the data assigned to the record. Each data record is located in a file node for convenience in searching. Also part of the file are a number of index records that are used for speedy access to individual records.

A B-Tree file contains of a number of nodes, and each node contains a number of records. Each node is 512 bytes long; the more astute will realise that this is also the length of one data sector. At the start of each node is a 14 byte area called the node header. The layout of a B-Tree node is shown below:



The layout of the node header is as follows:

0000	A forward link to the next node at the same level in the B-Tree. This forms a sequential chain though each level (4 bytes).
0004	A backward link to the previous node at the same level in the B-Tree. (4 bytes).
0008	The type of node (1 byte). Here are the possible types:

0	index node
1	header node
2	map node
255	leaf node

0009	The level of the node in the B-Tree. Leaf nodes are always at level 1, the first level of index nodes above them are level 2, and so on up to the header node (1 byte).
000A	The number of records currently located in this node (2 bytes).
000C	An unused field (2 bytes).

Following the node header are a number of variable length data records. At the end of the node are two byte pointers to each of these records. The area between the last record and the offset pointers is designated as free space. There is also a pointer to the free space at the end of the node.

B-Tree Header Node

The first record in every B-Tree is the header node with a node type of 1. This record always contains three records: the B-Tree header block, a user area, and a bitmap. When the B-Tree file is opened, the B-Tree header block is read into memory where it forms the nucleus of the B-Tree Control Block. The layout of the header node is as follows:

000E	Header block (104 bytes).
0078	User area (128 bytes).
00F8	B-Tree bit map (256 bytes).

The layout of the 104 byte header block is as follows:

0000	Current depth of the B-Tree (2 bytes).
0002	The logical sector number of the B-Tree root node (4 bytes).
0006	The number of leaf nodes in the B-Tree (4 bytes).
000A	The logical sector number of the first leaf node (4 bytes).

000E	The logical sector number of the last leaf node (4 bytes).
0012	The size of each B-Tree node. This field is always equal to 512 (2 bytes).
0014	Maximum size of a key in the B-Tree. All the keys in any given B-Tree have the same length (2 bytes).
0016	The total number of nodes in the B-Tree (4 bytes).
001A	The number of free nodes in the B-Tree (4 bytes).
001E	Unused area of the header (76 bytes).

The user area of the header node is not used by the system and is available for use in any manner designated by the owner of the B-Tree.

The bit map is very similar in concept to the volume bit map. It is used to indicate which sectors are in use and which sectors are free within the B-Tree. One bit is designated for each sector in the file. When the sector contains valid data the bit is set to one. As the file expands, new sectors will be required to store data and the bit map is used to quickly locate a free sector.

B-Tree Map Node

A map node has a type field of 2. This type of node is very infrequently found in B-Tree files. When a B-Tree file exceeds 256 in-use sectors, the bit map in the file header will overflow its designated area in the header node and a map sector is then allocated to hold the overflow portion of the bit map.

The forward link pointer in the node header (see above) is used to form a linked list of map nodes. The link pointer in the B-Tree header node will point at the first map nodes. If no map nodes have been allocated, this field will be zero. Note that the backward link pointer field is not used.

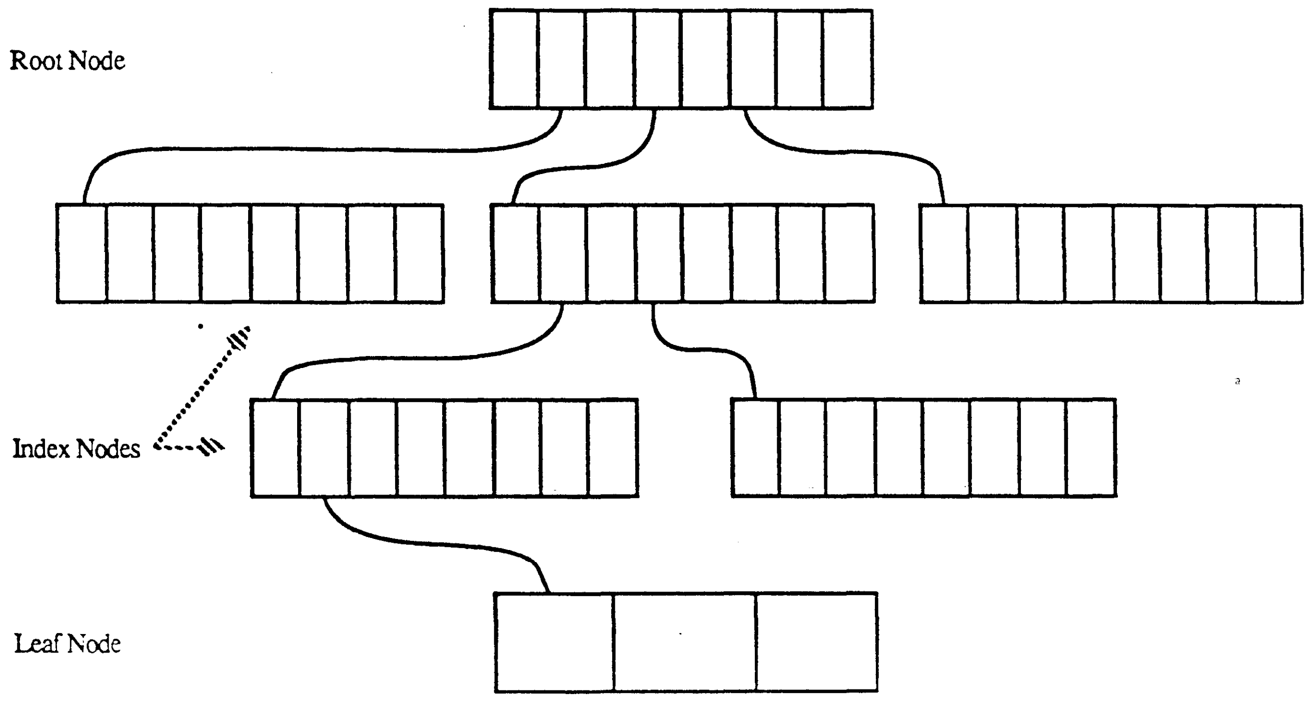
B-Tree Index Node

An index node has a type of 0. This type of node contains a series of records that point to other nodes in the B-Tree file. Each record consists of a fixed length key followed by a four byte pointer to another node (which could be another index node or a leaf node). The index nodes are used to speedily search though the file to locate a specific data record in a leaf node.

The diagram below shows how a search of a B-Tree proceeds from the root node to the leaf nodes that contain the data. The search starts by reading the root node (whose sector number can be found in the B-Tree header node) and searching linearly through the node until a record is found whose key is equal to or greater than the key of the record that is to be located. When this record is found, the pointer in the record is used to read another node from the file. If the newly read sector is also an index node, the above search pattern is repeated.

Eventually the record accessed will be a leaf node which is then searched for the required record. If the record is not located in the leaf node, it is not in the file.

This is a fairly complex structure. An insertion or deletion can result in index nodes being split or removed from the file. An operation such as moving a large number of files into, out of or between directories can result in a substantial number of sectors in the Catalog B-Tree being changed.



B-Tree Leaf Node

The leaf node is the only type of node that actually contains data records. Each record in a leaf node is laid out as shown below:

Record Key (4 to 254 bytes)	Record Data Area (4 to 502 bytes)

The key fields and the data fields are considerably different in the Extents B-Tree and the Catalog B-Tree so each will be described in turn.

Extents B-Tree Structure

The Extents B-Tree contains information about each file extent on the disk. Each record contains information about three extents on disk, each extent being described by a extent descriptor, and is preceded by the record key as follows:

0000	Length of the record key, always equal to $07 (1 byte).
0001	The fork to which this file belongs, $00 for the data fork or $FF for the resource fork (1 byte).
0002	The Catalog Node Identifier (file number) of the file (4 bytes).
0006	The relative offset of the first allocation block that is described by the first extent descriptor in the record (2 bytes).
(End of record key and start of record data)	
0008	First extent descriptor (4 bytes).
000C	Second extent descriptor (4 bytes).
0010	Third extent descriptor (4 bytes).

Each extent descriptor has two fields as follows:

0000	The relative offset of the first allocation block in the extent (2 bytes).
0002	The number of allocation blocks described in the extent (2 bytes).

Not all extent descriptors will necessarily contain valid data. Those that do not will contain zeroes.

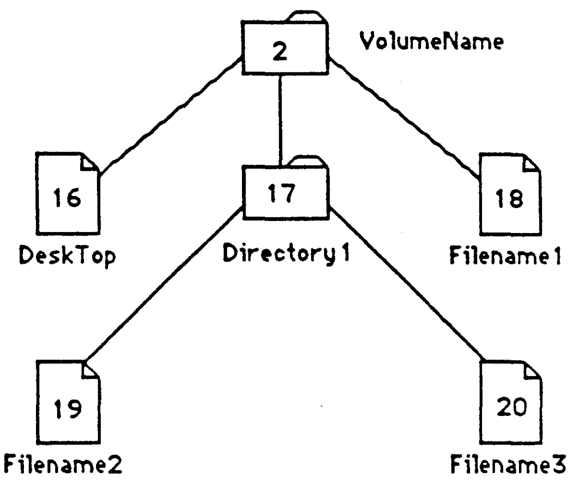
Not all files have entries in the Extents B-Tree because the first three extents in each fork are specified in extent descriptors within the file's entry in the Catalog B-Tree (described below).

Catalog B-Tree Structure

The Catalog B-Tree contains an entry for each file and directory on the disk as well as control information that defines the hierarchical structure on the disk. There are three types of record in the Catalog B-Tree; directory records, file records and thread records. Each record is preceded by its key.

As mentioned earlier, each file and directory on the volume is allocated an identifying number. This might be called the file number, the directory ID, the Catalog Node ID or the Cnode number, depending on which of Apple's documentation you are reading, but they all refer to the number that is assigned to the file or directory when it is created in the Catalog B-Tree. The system uses these node identifiers together with the file and directory names to navigate around the Catalog B-Tree.

In the Finder display, directories are shown as folders, and can be opened to show other directories and files that are enclosed in each folder. The following diagram shows a tree structure respresenting a volume called VolumeName with one directory called Directory1, three user files and the Desktop file. Two of the files are enclosed in Directory1. The numbers inside each node are the Cnode identifiers for each entity.



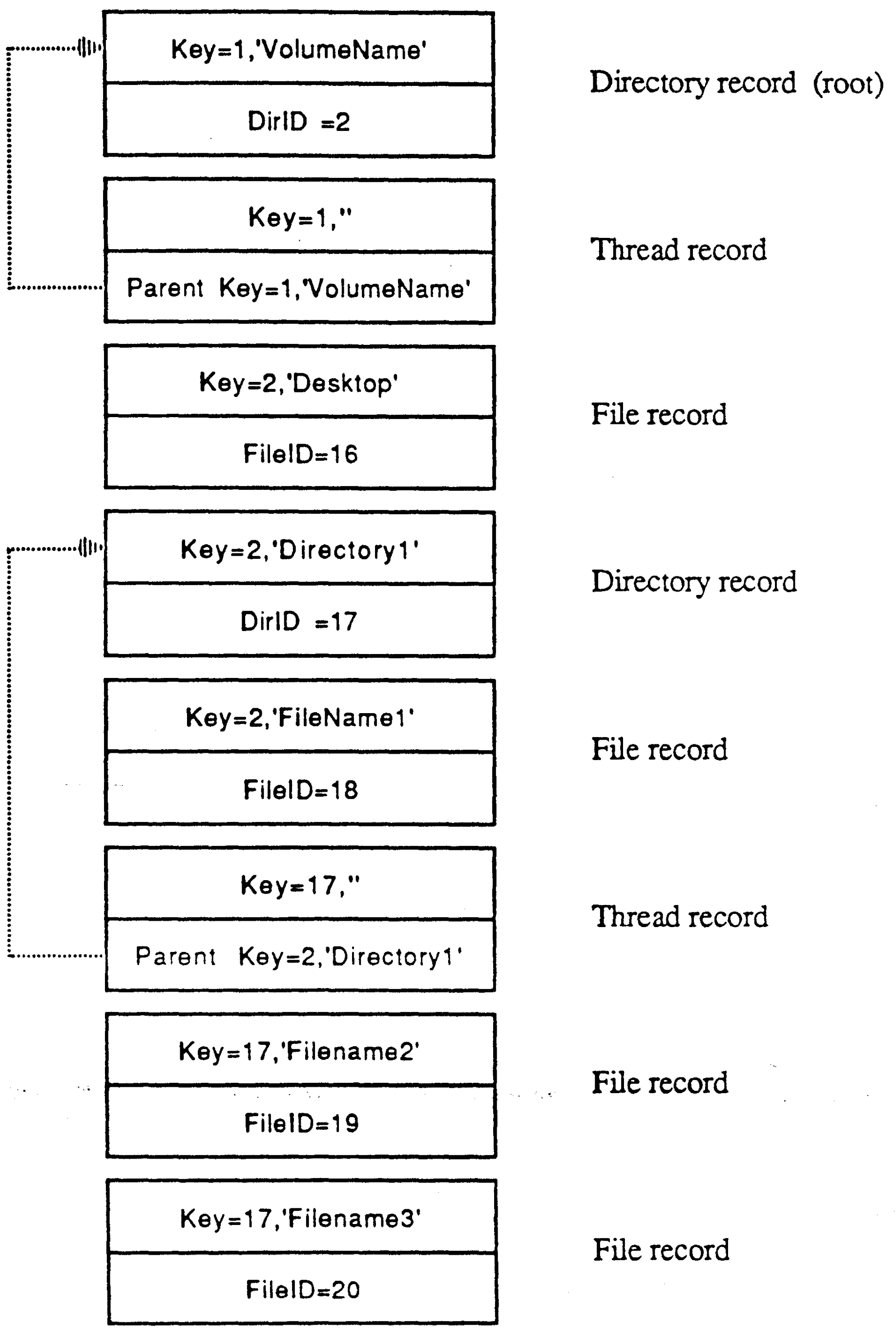
If you examine the node records of a Catalog B-Tree containing the above structure you would see two directory entries (for nodes 2 and 17) and four file entries (for nodes 16, 18, 19 and 20). In addition there would be two thread records, one associated with each directory record. The root directory for every volume always has a node identifier of 2.

Every node has a parent node which is defined as the node immediately above it in the hierarchy. The root node would appear from the above diagram not to have one, but in fact the parent node of the root directory is a non-existent node with a node identifier of 1.

Every node in the Catalog B-Tree has a key which is defined as the node identifier of its parent plus the name of the node, for example, the key for the Desktop file would be 2,'DeskTop'. Pathnames are defined by concatenating node names together separated by colons, for example, the pathname for Filename2 would be 'Volumename:Directory1:Filename2'.

As mentioned previously, there is one thread record in the Catalog B-Tree for every directory record. All directory and file records that have a common parent are given contiguous entries in the Catalog B-Tree. Preceding these records is a thread record which specifies the key of the parent node, that is, provides a link to the parent, thus allowing a backwards chain from any file record through each parent directory to the root directory.

The diagram on the next page shows the linkages that would be found for the Catalog B-Tree whose diagram is shown above.



All the records in the Catalog B-Tree share a standard format for their key, the directory identifier of their parent concatenated with their node name.

The layout for each Catalog B-Tree key is as follows:

0000	The length of the key in bytes (1 byte).
0001	One unused byte, always zero (1 byte).
0002	The directory identifier of the parent directory for this entry (4 bytes).
0006	The name of this entry in Pascal string format (up to 32 bytes).

There may be one additional byte following the key to make the total length of the key an even number. Sometimes this extra byte is included in the key length and sometimes it isn't!!

A directory record has the following data following its key:

0000	The directory record signature of 1 (1 byte).
0001	A single byte equal to zero (1 byte).
0002	The Finder flags for the directory entry. Not all the flags defined for files are valid for directories, but you can achieve some interesting effects by setting them. For example, invisible directories (2 bytes).
0004	The number of file and directory entries that are contained in the hierarchical level directly below this directory. This is sometimes referred to as the entry's valance (2 bytes).
0006	The node identifier for this entry (4 bytes).
000A	The timestamp when this directory was created. This is held in seconds since midnight on the 1st. January, 1904 (4 bytes).
000E	The timestamp when this directory was last modified. This is held in seconds since midnight on the 1st. January, 1904 (4 bytes).
0012	The timestamp when this directory was last backed up. This is held in seconds since midnight on the 1st. January, 1904 (4 bytes).
0016	Bytes for use by the Finder (32 bytes).
0036	Unused space (16 bytes).

A file record has the following data following its key:

0000	The file record signature of 2 (1 byte).
0001	Unused byte set equal to zero (1 byte).
0002	File system attributes byte. Bit 6 is set if the file is copy protected (1 byte).
0003	Unused byte set equal to zero (1 byte).

0004	The file type of the file. This is a four character field with values such as "APPL" for an application, "TEXT" for text file (4 bytes).
0008	The creator of the file. This is also a four character field. The values in this field are purely arbitrary and are used for matching applications to documents belonging to that application (4 bytes).
000C	Finder attributes word. This field is described in detail below (2 bytes).
000E	The location of this file on the desktop. Used only by the Finder (4 bytes).
0012	The number of the directory identifier in which this file resides (2 bytes!!).
0014	The node number of this file entry (4 bytes).
0018	Unused bytes set equal to zero (2 bytes).
001A	The logical end of file for the data fork. This is a count of the number of valid bytes between the start of the fork and its logical end of file (4 bytes).
001E	The physical end of file for the data fork. This is a count of the number of bytes on disk allocated to this fork of the file. It is always a multiple of 512 bytes (4 bytes).
0022	Unused bytes set equal to zero (2 bytes).
0024	The logical end of file for the resource fork. This is a count of the number of valid bytes between the start of the fork and its logical end of file (4 bytes).
0028	The physical end of file for the resource fork. This is a count of the number of bytes on disk allocated to this fork of the file. It is always a multiple of 512 bytes (4 bytes).
002C	The timestamp when this file was created. This is held in seconds since midnight on the 1st. January, 1904 (4 bytes).
0030	The timestamp when this file was last modified. This is held in seconds since midnight on the 1st. January, 1904 (4 bytes).
0034	The timestamp when this file was last backed up. This is held in seconds since midnight on the 1st. January, 1904 (4 bytes).
0036	Additional space for use by the Finder (16 bytes).
0048	The clump size of this file. Set equal to zero if the volume default size is to be used (2 bytes).
004A	The first extent descriptor for the data fork. Zero if unused (4 bytes).
004E	The second extent descriptor for the data fork. Zero if unused (4 bytes).

0052	The third extent descriptor for the data fork. Zero if unused (4 bytes).
0056	The first extent descriptor for the resource fork. Zero if unused (4 bytes).
005A	The second extent descriptor for the resource fork. Zero if unused (4 bytes).
005E	The third extent descriptor for the recource fork. Zero if unused (4 bytes).
0062	Unused space set to zero (4 bytes).

A thread record has the following data following its key:

0000	The thread record signature of 3 (1 byte).
0001	Unused bytes set equal to zero (9 bytes).
000A	The directory identifier of the parent of the file entries which follow this record in the catalog (4 bytes).
000E	The name of the parent of the file entries which follow this record in the catalog (always 32 bytes, zero filled on the right).

Tag Data

Each data sector when on disk consists of two portions, the data itself plus 12 bytes of tag data. The declared purpose of the tag data is to assist in the reconstruction of damaged disks. The tag data consists of the following areas (addresses are in hexadecimal):

0000	The number of the file to which this sector is allocated (4 bytes).
0004	Fork Indicator. If bit 1 of this byte is zero the sector belongs in the data fork, otherwise it is located in the resource fork (1 byte).
0005	For MFS volumes, the version number of file that last wrote this sector. On HFS volumes this field forms part of the following field to form a 24 bit relative sector number (1 byte).
0006	Relative sector number of this sector within its fork (2 bytes).
0008	Timestamp set when this sector is written to disk in seconds since midnight on January 1st., 1904 (4 bytes).
000C	Creator of the file as specified in the file's entry in the HFS disk catalog (4 bytes).
0010	The HFS directory identifier of the catalog entry which is the parent of the file in which this sector resides (4 bytes).

The last two fields are only applicable to HFS, and are only found on hard disks that support the 20 byte tag fields.

Most non-Apple produced hard disks do not support tags. In these cases the tag data fields are ignored on both input and output.

The Volume Bootstrap

The first two sectors on each volume are reserved for a volume bootstrap loader. These sectors are read into memory whenever an attempt is made to boot the Macintosh from that disk. Currently the boot sectors are the same for both MFS and HFS. The first part of the sector zero is data defining some of the standard system names and sizes and the rest is code to get the system running.

The layout of the data areas at the start of sector zero are as follows (addresses are in hexadecimal):

0000	$4C4B - 'LK' in ASCII, for Larry Kenyon - the principal architect of the file system.
0002	A branch to the code starting point (4 bytes).

0006	The version number of the bootblocks (2 bytes) - see below.
0008	Page flags (2 bytes).
000A	Name of the system file (16 bytes).
001A	Name of program to run when exiting application (16 bytes).
002A	Name of the debugger program to use (16 bytes).
003A	Name of disassembler to use if debugger is loaded. This is pretty much historical only (16 bytes).
004A	Name of the file containing the startup screen (16 bytes).
005A	Name of the first application to run after the bootstrap is completed (16 bytes).
006A	Name of the clipboard file (16 bytes).
007A	Number of file control blocks to allocate. This field defines the maximum number of files that can be open (2 bytes).
007C	Max number of elements for the event queue (2 bytes).
007E	System heap size for 128K system (4 bytes).
0082	System heap size for 256K system (4 bytes).
0086	System heap size for 512K and larger systems (4 bytes).

When a disk is initialized, no boot blocks are written into sectors zero and one. In order to get this task accomplished, it is necessary to copy another disk onto the new disk (perhaps by dragging the icon for the old disk onto the new disk), copy a file called "System" from another disk of the same type or to use a program designed to write boot blocks such as the Disk Utility program available from Apple or Fedit Plus.

The current boot block version number (June 1986) is $0012. This is the version that is written to the disk by the latest version of the Apple Disk Utility program. Incidentally, there is a feature in the Disk Utility program that may be of use to developers using 512K systems who wish to test that their programs will work correctly when executed on a 128K system. If the command and option keys are held down while the Write Boot Blocks option is selected, a special version of the boot blocks is written to the disk. The effect of these blocks is to cause a 512K system to be initialized as if it contained only 128K of memory. If you are looking at the boot blocks using Fedit Plus, you can recognize this special bootstrap because it has a version number of $0014.

On systems with 128K ROMs, the system does not boot from the bootstrap loader as it has the necessary code built into the ROM. One effect of this is to cause a change that you make to the size of the system heap or other default parameters to be ignored, as the values are taken from ROM. In order to make an effective change in the size of the system heap you should change the version number of the boot blocks to a value $0015. This will cause the values in the bootstrap sector to override the values in ROM.

If you wish to force the system to call the boot block code as a procedure during the system boot process, you should set the value of the version number to $4400.

The Finder Flags

This is a two byte area in the directory entry for each file on both MFS and HFS volumes. The flags in this field describe various attributes of the file to the Finder. The meanings of each bit (starting from the left) is as follows:

15	Locked. This bit is set if the file is locked.
14	Invisible. This bit is set if the file is not to be displayed by the Finder.
13	Bundle. This bit is set if the file has one or more icon lists, file references or version data. If a file entry has this bit set, the Finder will copy this information into the "desktop" file that it maintains on each disk. The most common reason for setting this bit is to get the Finder to recognize non-standard icons.
12	System. This bit is set if the file is a system file.
11	Bozo. This is a protection scheme so simple, only a bozo would be deterred by it.
10	Busy. Set if the file is currently busy.
9	Changed. The file has been changed, and needs to be updated on disk.
8	Inited. The file has been seen by the Finder and allocated a location in its assigned folder.
7	Shared. Indicates that the file is open by multiple applications for shared access.
6	Cached. This bit is currently unused.
5	On desk. Specifies that this file is on the desktop, not in its folder.
4	Always switch launch. Normally, when the boot volume is a hard disk and an application on another disk is launched, the hard disk remains as the boot volume. When this attribute is set, launching the file will always cause the volume on which it resides to become the boot volume provided there is a valid system and Finder file on the disk.

The remaining bits in this word are not currently used.

Using The Program

This concludes the description of the disk data structures. The rest of this manual describes the Fedit Plus program and how the various commands are used to examine and manipulate the disk structures.

Fedit Plus is not a difficult program to understand, but it is very easy to misuse it and accidentally change data on the disk in a way that you did not anticipate. Now would be a good time to re-read the warnings on the first page of this manual about data security.

Fedit Plus is essentially a disk editor. The principal difference between Fedit Plus and more conventional edit programs it that Fedit Plus edits data as it exists on disk without regard to its logical structure, and conventional editors always make some assumptions about the data structure.

With Fedit Plus there is no concept of inserting and deleting data. As it operates on disk sectors of fixed size, you can only replace data in each sector. You can never increase or decrease the amount of data in a sector; it is always fixed at 512 bytes (plus the 12 or 20 tag bytes if supported).

File And Volume Modes

Fedit Plus operates in two modes - file mode and volume mode. In file mode you can open a file and read each disk sector that is allocated to the file. You can also examine the file entry in the disk directory and make changes to some portions of that entry. In volume mode, you can read all the sectors in a volume without regard to which file (if any) each sector belongs. This mode is most useful for examining and changing the system areas of a disk such as the boot blocks, the volume information table and the disk directory.

Lisa Diskettes

If a diskette initialized by the Lisa Workshop or the Lisa Office System is placed in a Macintosh, it normally cannot be read. The Macintosh will give you the option of either ejecting the diskette or initializing it.

If a Lisa diskette is placed in the Macintosh while Fedit Plus is running, it will report that the diskette was not initialized on a Macintosh and will give you the option of ejecting the disk or mounting it. If you choose to mount the diskette, then Fedit Plus can edit it as normal, but only in the volume mode.

ASCII versus Hexadecimal display

You may choose to display sector data either as ASCII characters only or as hexadecimal characters with the ASCII translation beside them. Which display you use is primarily a matter of personal preference. The ASCII mode is useful because a full sector can be displayed, and it is easy to scan though a large number of sectors quickly. The hexadecimal display is often more useful for detail work on a sector.

Running The Program

As with all other Macintosh programs, Fedit Plus is run by double clicking on the Fedit icon (unless you are using the minifinder or another program to replace the Finder). This loads the entire program into memory including all system resources that may be required by Fedit Plus. After this load has been completed, you may remove the disk containing Fedit Plus and the system disk from the system as no resources should ever be required to be loaded during program execution. This is useful if you have a single drive system, as you will not have to perform any disk swapping in the middle of the program - a definite problem for a disk editor.

When the program is first run, you are presented with four menus across the top of the screen. These are described in detail below. Normally, the first action you will take will either be to open a file or a volume from the "File" menu. When you do this, you will be presented with the first 512 byte sector of the volume or file that you have selected. This will either be in ASCII or hexadecimal format depending on the default chosen in the "Configure" item of the file menu.

ASCII Mode Display

The ASCII format shows an entire 512 byte sector as a series of ASCII characters in 8 rows each of 64 characters. Characters that are not valid for printing are displayed as periods. A blinking cursor is displayed beneath the current character, marking the insertion point. In order to change the position of the character cursor, you should move the mouse cursor on top of a new character and press the mouse button.

Beneath the character display is a line showing the hexadecimal value of the character at the insertion point, the offset of the insertion point from the start of the sector, and the offset of the current sector from the start of the file or volume currently being displayed.

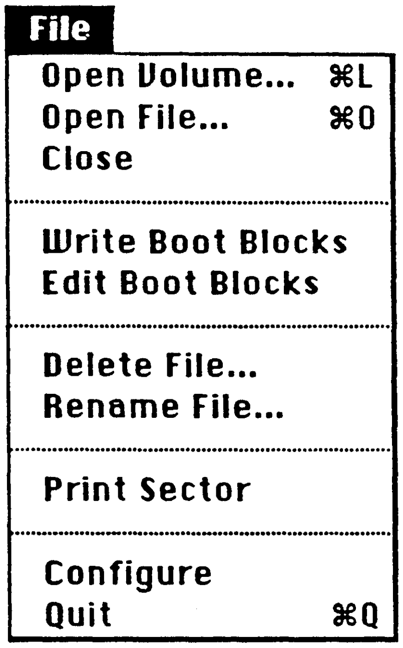
Hexadecimal Mode Display

In this mode, the data in a sector is displayed in hexadecimal characters with the ASCII representation beside them. The current insertion point is shown by one inverted hex character. Because this mode requires more space, the entire sector cannot be displayed on one page. A vertical scroll bar on the right side of the window allows a user to switch between the first and second portions of the sector. At the start of the sector is displayed the sector tags and the offset of the sector from the start of the file or volume, both in hexadecimal.

Changing the Current Character Position

In both ASCII and hexadecimal display modes, the current character may be changed by moving the mouse pointer to the required character, and clicking the mouse. Fedit Plus also supports moving the current character pointer by using the cursor keys on the Macintosh Plus. The Tab key will move the current character pointer forward to the next multiple of eight characters in the sector, and the shift/tab will move the current character position backwards in the sector to a multiple of eight characters.

The File Menu



This menu contains items applicable to file and volume opening and closing, the use of the printer, and some housekeeping items.

Open Volume

This command requests that a volume be opened for input and updating. This is most useful if you wish to examine the system areas of a disk such as the boot blocks, volume allocation tables or disk directory.

When a Lisa disk or a disk with errors has been mounted, these can also be examined using the Open Volume command. In these cases Fedit Plus will not be able to determine the volume name, and the disks will be referenced using names such as "(Internal Drive)" and "(External Drive)".

For hard disks that are partitioned, each partition is considered a separate volume.

Open File

This command is used when you wish to open a file on a volume. This calls the standard Macintosh file selection dialog which passes the name of the requested file to Fedit Plus.

One point to be aware of is that the normal system action will occur if a Lisa or other non-standard disk (or a disk with directory errors) is inserted while the file selection dialog is active. This action consists of asking you whether you wish to initialize or eject the disk. Within the rest of Fedit Plus, the normal action is to describe the error condition found, and give you the choice of mounting or ejecting the disk. By mounting the disk, you are able to read it using the Open Volume command.

Close

This command closes the volume or file that is currently open. If a desk accessory is currently active, this is closed instead.

Write Boot Blocks

This command writes boot data in sectors zero and one of the volume currently open. Version 1.2 of the boot data (see previous discussion) is always written. The data is taken from a resource in the Fedit Plus code file. If you wish to change the boot data written you should modify this resource. It is BOOT resource number 128.

Edit Boot Blocks

This command allows you to edit the fields in the boot blocks that define the names of the various system and startup files and the sizes of various system options. On a system that has the 128K ROMs, certain options such as the size of the event queue and the system heap size are actually taken from code in the ROM. It is necessary to change the version number of the boot blocks to 21 (hexadecimal $15) for the size specified in the boot blocks to override the ROM values at the time the system is booted.

Delete File

You may delete a file in the current volume using this command.

Rename File

You may rename a file in the current volume using this command.

Print Sector

This command copies the current disk sector to the system printer. The format of the sector is similar to the hexadecimal display. Data related to the position of the sector in its file on the disk is also printed if applicable.

Configure

This command allows you to reconfigure the default mode for displaying data (either ASCII or hexadecimal) and to choose the type of cursor to be used in each display mode. Six standard cursors are provided, but if your favorite cursor is not among them, you can modify or substitute the supplied cursors using a resource editor. The cursors are resource type CRSR with values of 256 to 261.

The configure command also gives you some control over the input of numbers. This is how Fedit Plus examines any numbers that you input to a dialog:

> if the number starts with a dollar sign ($) it is considered to be hexadecimal,
>
> if the number starts with a pound sign (#) it is considered to be decimal,
>
> if there is a hexadecimal character (A to F, or the lower case equivalents) in the number it is assumed to be hexadecimal,
>
> and if none of the above is true, the number is assumed to be decimal or hexadecimal according to the setting of the default in the Configure dialog.

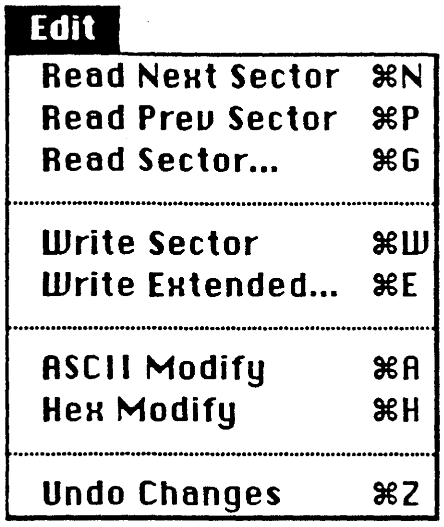
An occasional problem with the hexadecimal display screen is that the whole of the sector is not displayed at once. If you are scrolling though a file or volume you may wish to show the top and bottom halves of each sector in sequence to ensure that you have visually examined all the data in each sector. This may be done by selecting the option that allows half sector moves in this dialog.

Characters in a file are normally displayed as their standard ASCII equivalents. Characters that are outside the range of $20 to $7E (space to "~") are displayed as periods. The Configure dialog also allows you to change this setting. Using the Extended character set, characters in the range $A0 to $D9 will also be displayed. If this setting does not fit your needs there is also a Custom character set that you can set up as you require. These three sets can be found in the code file as resource type FXTL numbers 256 (Standard), 257 (Extended), and 258 (Custom).

The configure dialog also allows you to modify one of the Fedit Plus safety features. Normally each write to the disk has to be confirmed by you, but if you prefer this can be bypassed. This is not recommended unless you are confident in your abilities and never make mistakes.

Quit

This command allows you to exit the program.



This menu contains items relating to the selection of sectors for editing, the type of edit to use, discarding changes and writing changed data back to disk.

Read Next Sector

This command reads the next higher logical sector for the file or volume currently open. This is the same action that is taken if the mouse is clicked in the down arrow or down page regions of the horizontal scroll bar at the bottom of the display window.

Read Last Sector

This command reads the next lower logical sector for the file or volume currently open. This is the same action that is taken if the mouse is clicked in the up arrow or up page regions of the horizontal scroll bar at the bottom of the display window. No action is taken if you attempt to use this command while the first sector of the file or volume is displayed.

Read Sector

This command allows you to enter the number of the sector that you wish to display. The number is a decimal value between zero and the last sector number of the file or volume currently open. This action is very similar to the action taken by moving the thumb of the horizontal scroll bar at the bottom of the display window.

Write Sector

After a sector has been modified, you can write the sector back to the disk by selecting this command. After confirming that you really do wish to update the disk, the sector will be written back to the disk at the location from which it was read.

Write Extended

This command presents you with a dialog box allowing you to select a different volume and position to re-write the current sector. The default selection is always the original position of the sector on the volume currently open. The sector number is always relative to the start of the disk volume, even when a file is open.

You should be very careful when using this command as there are no checks to prevent overwriting any portion of any disk volume on the system. This command can be very useful for reading portions of a disk that have been destroyed and writing them to another disk.

ASCII Modify

This command allows you to modify the data in the sector. When this command is active, a warning message is displayed in the menu bar. The command is deselected whenever a new sector is read from disk unless the Configure dialog has been set to always enable the command.

Pressing any non-control key will cause the character at the current insertion point to be overwritten and the insertion point to be moved forward one character. If the insertion point is located at the last character of the sector, it is wrapped around to the first character.

Pressing a control key (cursor key, backspace or tab) will cause the normal for that key to occur; that is, the position of the insertion point in the sector will be altered.

Any changes that you make to the sector are not written to disk until the Write Sector command is selected.

Hex Modify

This command allows you to modify the data in the sector. It may be selected from the Hexadecimal or ASCII display modes. When this command is active, a warning message is displayed in the menu bar. The command is automatically deselected whenever a new sector is read from the disk unless the Configure dialog has been set to always enable the command.

If in Hex display mode, pressing any valid hexadecimal key (0-9, A-F) will cause the character at the current insertion point to be overwritten and the insertion point to be moved forward one character. If the insertion point is at the last character of the sector, it wraps around to the beginning of the sector.

Pressing a control key (cursor key, backspace or tab) will cause the normal for that key to occur; that is, the position of the insertion point in the sector will be altered.

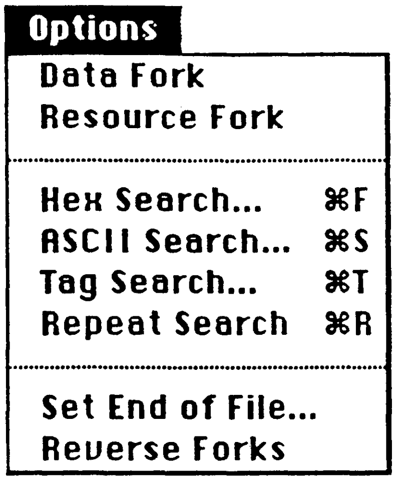
In ASCII display mode, two hexadecimal characters are required to overwrite one ASCII character. After you have typed the first hexadecimal character, you must type another one; this is the only action that Fedit Plus will allow at this point. If you suddenly find that menu commands won't operate correctly, check that the program is not waiting for a second hexadecimal character.

Any changes that you make to the sector are not written to disk until the Write Sector command is selected.

Undo Changes

Using this command, you can undo any changes you have made to the current sector using the Hex or ASCII modify commands. You will be asked to verify that you wish to revert to the original data.

Options Menu



This menu contains various items that modify the normal progression for reading a file or volume, such as selecting a different fork or searching for a particular string.

Data Fork

This command is only valid when displaying a file. It allows you to select sectors from the data fork for display. If the data fork is empty, an appropriate error message is displayed.

Resource Fork

This command is only valid when displaying a file. It allows you to select sectors from the resource fork for display. If the resource fork is empty, an appropriate error message is displayed.

Hex Search

This command allows you to search the file or volume currently open for a specific hexadecimal string. Up to 32 hex characters (between 0-9 and A-F) may be specified. Spaces may also be included to make the string more readable, but these are ignored during the search.

ASCII Search

Using this command you may search for a specific ASCII string of up to 32 characters. Upper and lower case are considered separate characters unless the "Ignore Case" box is checked.

Tag Search

This command allows you to search though the tag bytes of each block looking for specific patterns. You should refer to the layout of the tags described earlier for more details of what this command can do for you.

Repeat Search

This command repeats the last search specified, starting one character beyond the current character in the sector.

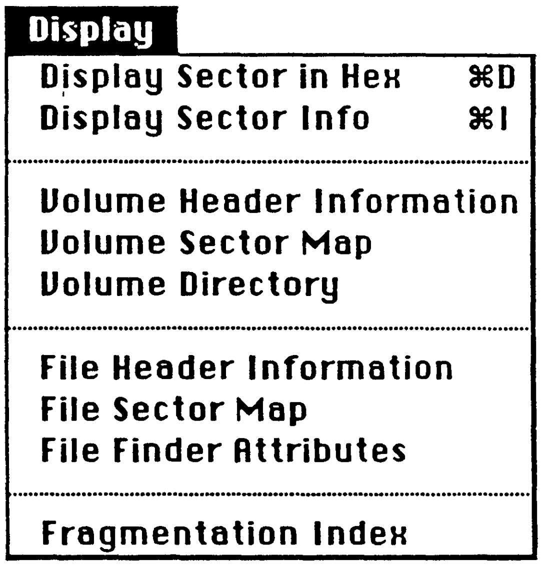
Set End of File

Using this command you can set the current end of file position to the start of the file (effectively creating an empty file), to be equal to the end of the physical file, or to any position in between.

Reverse Forks

This is a command that switches the pointers in the disk directory for the data and resource forks. It is as if the entire data fork of the currently open file is replaced by the resource fork, and vice versa.

The Display Menu



This menu allows you to modify the method of displaying data, or to display special areas of the disk.

Display sector in Hex/ASCII

Using this command you can switch between the hexadecimal and ASCII display modes.

Display Sector Info

This command gives access to extra information about the current sector. It displays the absolute sector number of the current sector, together with the position of the sector on the diskette in terms of track and sector numbers (but only if the sector is on a diskette in a Sony drive).

The information in the data tags for the sector is also displayed, interpreted and verified. Any errors found are flagged.

The file name, fork and position in the fork of the file are displayed. This can be very useful when reading a volume, as you can tie the data in the sector directly back to the file to which it belongs. On HFS volumes, a sequential search of the disk directory may have to be made to locate the filename and build the full pathname; this can take a number of seconds if there are many files on the volume.

Volume Header Information

This command intreprets the various fields in the volume header in sector 2 of the volume. The various fields are identified and the values are displayed in hexadecimal and decimal where appropriate. Timestamps located in the header are also interpreted.

Volume Sector Map

This command displays a graphical interpretation of the status of each sector in the currently selected volume and shows which sectors on the disk are allocated to a file and which are available. This command is currently only available for single-sided MFS diskettes.

Volume Directory

The volume directory command shows selected information from the disk directory of the current volume. Information is presented on a sector by sector basis so that you can easily correlate information between this display and a direct view of the disk directory sectors. From this display you may select a file (by clicking the mouse on the filename), open a file (by double clicking the mouse on the filename, or selecting the Open button), or modify the directory attributes (by selecting the SetAttrs button).

You may move forward or backward in the disk directory sectors by selecting the appropriate button, or return to the standard display mode by clicking on the End button.

File Header Information

This command is only available when displaying a file. It interprets all the fields in the disk directory heading (as opposed to the disk directory listing that only interprets some of the fields). The various fields are identified and the values are displayed in hexadecimal and decimal where appropriate. Timestamps in the file header are also interpreted.

File Sector Map

This command is only available when displaying a file. It displays the physical location of each sector in the current file and (if the file is on a single-sided MFS diskette) displays a graphical interpretation of the same data.

File Finder Attributes

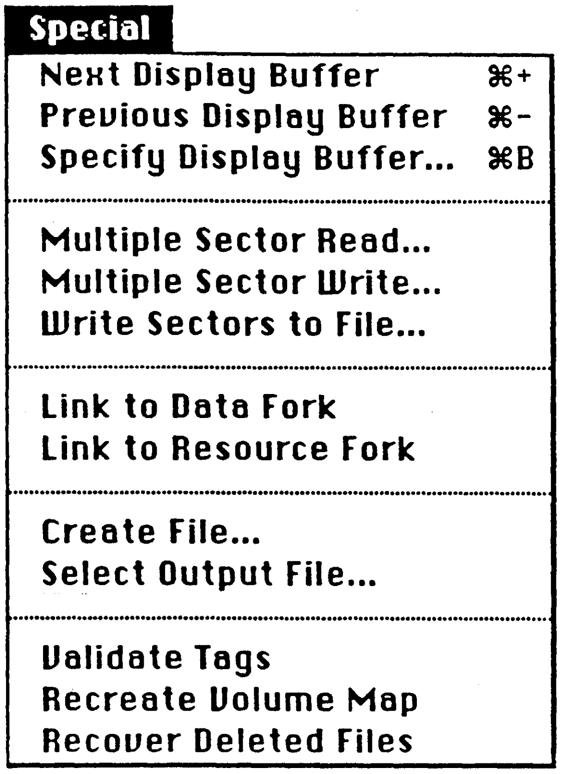
This command is only available when displaying a file. It allows you to update the various fields of the Finder attributes in the disk directory.

Fragmentation Index

This command will examine the disk organization of an HFS organized volume and will report on the degree of fragmentation found on files on that volume. If all files on the volume have no more than one extent for each fork that exists then the index will be 0.00, but as forks get split into multiple extents the index will rise. The index actually measures the amount of fragmentation that occurs on each fork and combines those figures into a composite for the whole volume. An index of 1.00 would indicate that every fork of every file had more than one extent and that the first extent of each fork was extremely small - less than one percent of the fork size.

This index is only a rough indication of the performance degradation that could occur due to fragmentation. It is suggested that volumes having a fragmentation index greater than 0.4 should be reorganized by copying all files to floppy diskettes (or another backup medium), reinitializing the disk and restoring the files. After this has been done the fragmentation index should return to 0.00.

Special Menu



This menu contains items for use in special circumstances. One of the most difficult procedures in disk handling is recovering from disk problems. As discussed previously, the tags on each sector can be used for data recovery, but most hard disk manufacturers do not support tags. The items in this menu can help make recovery possible on these types of disk as well as on floppy diskettes. Unfortunately, due to space restrictions none of the items in this menu will run in a 128K system. If you are running Fedit Plus under Switcher you may also want to play with the partition size until you have the number of buffers that you require.

When Fedit Plus is initialized, the program will allocate a number of buffers for use in storing data in memory. This will vary between 1 and 400 buffers depending on how much memory is available after making an allowance for desk accessories that you may wish to use. Using the items in this menu you may read data from any sector on disk into any buffer and then write the data out into another area of the disk or into a new file.

A second method of file recovery without using tags is also available. This method allows you to select sectors and tell Fedit Plus to add them to the end of a file. This method can be more convenient that using buffer copies, but cannot always be used, for example if there are problems with the disk directory or the volume allocation tables. In these cases you will have to copy the file into one or more buffers before writing them to a different disk. Both the above methods have advantages and disadvantages. They are somewhat crude methods of recovering data, but without tags there appears to be no better methodology available. The real answer is for disk manufacturers to support the Apple tag storage areas on each sector.

For users who have tags available, the above methods can still be used as well as the tag specific file recovery methods.

Unfortunately, most of the methods described below do not yet operate under HFS. These will be provided in future versions.

Next Display Buffer

This routine allows you to select the next buffer to be displayed on the screen.

Previous Display Buffer

This routine allows you to select the previous buffer to be displayed on the screen.

Specify Display Buffer

Using this routine you may select a buffer to be displayed on the screen. The currently displayed buffer is used for all single sector disk operations.

Multiple Sector Read

This command allows you to read a number of sectors from the currently open file or volume into memory. Each sector read from disk will be stored in a separate buffer starting with the buffer selected in this menu. Any sector in any buffer can be edited before writing it back to disk.

Multiple Sector Write

This command allows to to write data from memory buffers onto the current volume. The data is written directly to disk and not as part of any file that may be open. The volume allocation table is not updated to reflect any changes that may occur. This command can be useful if you need to write into the system areas of a disk.

Write Sectors to File

This command is a great tool for trashing disks. Be careful using it. It permits you to write data from the buffers onto a selected file on disk. No validity checks whatsoever are made on the data, so you are able to create some very strange files. All that is guaranteed is that the resultant disk file will conform to the requirements of the Macintosh file manager. This command is different from all others in Fedit Plus because it permits you to write to only part of a sector, thus allowing great flexibility in what data you change. By changing the current end of file pointer you can write to any location in a file.

Link to Data Fork

This command is designed for file recovery. When invoked, the allocation block to which this sector belongs is linked in the Volume Allocation Table at the end of the Data Fork of the currently selected output file. You should note that the complete allocation block is linked in, not just the displayed sector. On a standard MFS diskette this will be two sectors, but the number of sectors will vary on a hard disks. Typical values are between eight and twenty sectors on MFS organized hard disks, and one to three on HFS organized disks.

Link to Resource Fork

This command is exactly the same as the previous command, except the allocation block is linked into the resource fork of the currently selected file.

Create File

This command allows you to create a file on a volume. The created file is automatically selected for output from the Write Data command or Link commands. The file is created with the type and creator fields set to question marks, but these items can be changed using the Display File Attributes command.

Select Output File

This command allows you to select a file for output from the Write Sectors to File command or for data to be linked in using the Link to Data Fork or Link to Resource Fork commands.

Validate Tags

This command reads each sector that is allocated to a file and validates the various fields in the tag bytes. Any sectors that are found to have incorrect tags will be rewritten with corrected fields. This command is essential if you are using MacWorks on a Macintosh XL. This is a problem with current versions of the file that results in invalid tag fields being written on hard disks connected to the system. The tag fields are not corrected when a file is copied from a hard disk to a floppy diskette, so any diskette that has been used on a Macintosh XL should be considered suspect.

Recreate Volume Map

This command will totally rebuild the Volume Allocation Map and the Volume Information Table on a hard disk or floppy diskette using the MFS file organization. This is done by reading all the tags on a volume and determining the file structure from the tags. As a side effect, any files that are found on the disk but are not located in the disk directory are also recovered.

Recovered files are given a name with the format "REC.nnnnn" where nnnnn is the internal number of the file. The type and creator fields are given the value "????" and the logical end of file for each fork is set to the physical end of file. All the file system and Finder attributes are reset. You will have to restore the type and creator fields yourself by examining the data and determining what the correct values should be.

Recover Deleted Files

This command is very similar to the previous one, except that the volume Allocation Table is not rebuilt. Files are recovered in the same manner and are given the same type of names and file attributes. Currently this command only works for MFS organized volumes.