File Manager for Mac OS 9 Preliminary Documentation

Preliminary Draft

Technical Publications

1. Summary

The goal for HFS Plus in Mac OS 9 is to add basic API support for the volume format features not supported in the Mac OS 8.1 release. This includes support for large files (forks over 2GB), long Unicode names, and named forks.

There are also a few fields in the HFS Plus catalog records that are not found in HFS catalog records. These fields exist primarily to ease support for Mac OS X. These fields include the `contentModDate` and `modifyDate`, permissions, and text encoding hint.

Note

HFS Plus APIs in Mac OS 9 does not include implementing named forks on HFS Plus volumes. The API parameters will remain flexible enough to allow the complete implementation of named forks at a later date. ◆

2. Feature Set

The primary purpose for HFS Plus in Mac OS 9 is to finish support for the volume format features found in HFS Plus, particularly large (>2GB) files, long Unicode filenames, and named forks.

3. Compatibility Requirements

3.1 Hardware Requirements

HFS Plus for Mac OS 9 will have the same hardware requirements as Mac OS 8.5. It requires PowerPC because it loads some shared libraries very early in boot (long before CFM-68K is initialized). The statically-linked 68K version of the Text Encoding Converter is no longer used.

3.2 Software Requirements

There are no planned changes to the behavior of existing APIs; there are new APIs for Mac OS 9. Any program that wants to take advantage of the new features (large files, long Unicode names, named forks) introduced in Mac OS 9 will have to be revised to use the new APIs.

4. Architectural Requirements

4.1 Aspects of Long-term Architecture

In order to take advantage of preemptive multitasking, applications need to be able to make File Manager calls that execute asynchronously with respect to the application. In the past, the solution has been asynchronous parameter block-based calls that return control to the caller before the call completes, and cause the caller's completion routine to execute after the call completes.

But asynchronous calls (especially a series of them) are cumbersome and error prone to implement. A more convenient programming model is to allow the caller's thread of execution to be suspended while the call is completing. Further, the parameters to the call should be able to be passed as function arguments, rather than as fields of a parameter block structure.

This implies that the high-level functions should support the full functionality of the API. This is not the case in the original File Manager, where the high-level calls are often simplified versions of the low-level parameter block calls. In the HFS Plus API, however, the high-level calls provide complete functionality.

5. Software Architecture and Design

The API descriptions presented here are preliminary. They are intended primarily to give you an understanding of the kind of functionality that is available.

5.1 Component Description

The APIs described here provide access to three major features introduced in the HFS Plus volume format: long Unicode filenames, forks larger than 2GB, and named forks. There are also new informational APIs that combine all three new features.

While these APIs were added because of the HFS Plus volume format, they are intended to be suitable for other volume formats with corresponding features, such as UDF and NTFS. To simplify and encourage adoption of the Unicode and large file APIs, the APIs can be used with all volumes, even if the volume format's implementation has not yet added explicit support for these APIs. If the volume format's implementation does not explicitly support these APIs, then the functionality is limited to the functionality available through the older APIs.

For example, let's assume that the ISO 9660 implementation is not updated to support the new APIs. You will still be able to use the new APIs to access files on ISO 9660 volumes, but you will not be able to grow files beyond 2GB in size, and will not be able to create or rename files with names longer than 31 characters. You will only be able to create data and resource forks, for files only; you will not be able to create additional named forks.

Only programs that have been revised to use these APIs will be able to open files larger than 2GB (or be able to grow a file to be larger than 2GB). Eventually, programs that have not been revised will attempt to access these large files. This will cause the programs to receive an error code (`fsDataTooBigErr` or `fileBoundsErr`) at the time they try to open a large file or extend the file beyond 2GB.

It is anticipated that named forks will be used (in part) to enhance the UI of other applications such as the Finder by letting them store out-of-band information with files and directories.

5.2 Programmer Feature Set

These new APIs introduce three general new concepts: long Unicode filenames, large files (access to forks larger than 2GB), and named forks. To support these new concepts, several new data structures are introduced.

Long Unicode filenames and large files are really just modern variations on concepts already present in the File Manager API. The new APIs present functionality that is quite similar to older APIs, but using different data structures. To ease developer adoption and usage of the new APIs, the long Unicode filename and large file APIs can be used (limited to the functionality available through the old APIs) on all volume formats, even if the filesystem code handling that volume format has not been updated to directly support the new APIs. For those volume formats, the File Manager will automatically emulate the new APIs in terms of the existing APIs. This is called the compatibility layer.

For example, the `FSOpenFork` call allows you to open a file and access forks larger than 2GB. If a particular volume does not support the `FSOpenFork` call directly, the File Manager will automatically pass a `PBHOpenDF` or `PBHOpenRF` call to the volume's filesystem code and translate the inputs and outputs appropriately. An application need only test for the presence of the `FSOpenFork` call itself, not whether a particular volume supports it. (Note: if the volume does not support the `FSOpenFork` directly, forks will still be limited to 2GB in size and only data and resource forks may be created—for files only.)

Most of the new APIs are available in two forms: as a high-level API or as a low-level API. A low-level API uses a single structure called a parameter block to contain all of the input and output parameters for the call; a pointer to the parameter block is passed to the actual API function. A call to a low-level API may be synchronous or asynchronous. It is typical for several related low-level APIs to share a single parameter block structure; some of the fields will be unused for some of those APIs. All of the new API functionality is available via low-level APIs.

A high-level API gets all of its input and output parameters as separate parameters to the API function. A call to a high-level API is always synchronous. A high-level API is implemented internally using calls to the equivalent low-level API. The high-level API provides all of the functionality available via the low-level API (not true of the old APIs). The use of the high-level API is greatly encouraged; the low-level API should only be needed for async calls.

Many outputs in the new APIs are optional. This means that a client does not have to allocate space for an output value they will never use. It also means that the File Manager can avoid doing the work of computing or returning that output, which can make some operations faster. See the descriptions of individual APIs to determine which outputs are optional.

Several new error codes are introduced. The older File Manager APIs overloaded `paramErr` to mean both "some parameter was invalid" and "unrecognized call/selector". The intent with these new APIs is to introduce separate error codes for each kind of parameter that might be invalid. This is very helpful during initial debugging of code that uses the new APIs. If a call takes multiple fields of the same kind (for example, source and destination FSRef), the error code does not tell you which one was invalid.

The following subsections describe the new concepts and data structures specific to individual APIs. They also describe new or changed data structures and utility routines for implementing these new APIs in third-party filesystems.

5.2.1 Unicode

The HFS Plus volume format stores filenames as strings of up to 255 Unicode characters. In Mac OS 8.1, a name that is not representable in the text encoding used by the Finder (because it is too long, or contains characters not in that text encoding) is translated into a name that is representable, including the catalog node ID embedded within the name. This allows all files to be accessed by Mac OS 8.1, but does not allow clients to determine or change the original Unicode name.

The `FSCreateFileUnicode` and `FSCreateDirectoryUnicode` calls create a file or directory using a long Unicode name. The `FSGetCatalogInfo` call can return the long Unicode name of a file or directory (and other information as well). The `FSRenameUnicode` call is used to change the name of an existing file or directory; the new name is a long Unicode string. The `FSCatalogSearch` call can be used to find files and directories based on long Unicode names.

The new APIs do not explicitly pass file or directory names to identify objects. They use an opaque type, `FSRef`. This means that you must make an API call to determine an object's name; it cannot be copied out of the `FSRef`. See Section 5.2.4, "Identifying Files and Directories" for more information.

A large number of pre-existing interfaces in the File Manager and many other system software components take a filename as input or produce a filename as output. It would be impractical to provide new versions of each of these entry points to allow long Unicode names to be passed. For interchange with existing APIs, routines are provided to convert between an `FSSpec` and an `FSRef`.

When a client passes a Unicode name in a call to a volume that does not directly support Unicode, the name is converted to a PString in order to call the equivalent PString-based API. The inverse is true for APIs that return a Unicode name; the compatibility layer converts the returned PString into a Unicode string. The strings are converted using the File Manager's current text encoding.

If for some reason a Unicode name cannot be converted to a PString, the compatibility layer will return a `badNamErr` error from the call. Such a failure is generally caused when the Unicode name contains characters that are not representable in the desired text encoding. This could happen if the text encoding was incorrect, or if the Unicode string contained characters from more than one Mac OS encoding (or from non-Mac OS encodings).

▲ **WARNING**

Programs using the new APIs must check for and properly handle the `badNamErr` error. ▲

5.2.2 Named Forks

Prior to this release, the Mac OS File Manager has had a very limited notion of what constitutes a file or directory. Files have two well-known forks called the data fork and the resource fork. Directories do not have a data fork, nor a resource fork. Both files and directories contain certain metadata managed by the File Manager (such as creation and modification dates, and file and directory IDs). Files and directories also contain a small amount of metadata stored but not interpreted by the File Manager (such as Finder info or the backup date). These two types of metadata are collectively known as catalog information or catalog info.

Modern volume formats, including NTFS, UDF and HFS Plus, are capable of storing additional user data besides the data and resource forks and catalog info. While the details vary by volume format, they are generally able to store multiple, named pieces; they are often stored in the same way as a data or resource fork. These new APIs provide access to this extra user data, as well as unifying it with the data and resource forks, through a concept of named forks.

With the exception of catalog info, all other user data is stored in a fork. Forks have names composed of up to 255 Unicode characters. There are two fork names reserved for the data and resource forks: the empty string (i.e. length equals zero) and "Resource Fork". This allows the same APIs to easily access the traditional data and resource forks as well as other named forks.

The contents of a fork are a sequence of zero or more bytes.

5.2.3 Large Files

The HFS Plus volume format uses 64-bit numbers for the size of a fork. The `IOParam` parameter block uses a signed 32-bit field for offsets and other parameter blocks use signed 32-bit fields for fork physical and logical sizes. This means that the existing APIs are only able to represent forks up to 2GB. In fact, in Mac OS 8.1, the File Manager will return an error when attempting to open any fork that is 2GB or larger.

It is assumed that applications would get confused and possibly crash or corrupt data if they were allowed to open large files with the existing Open calls. Similarly, they should not be allowed to extend an open file beyond 2GB if they use the existing APIs.

Therefore a new Open call is required so that the File Manager knows that it is safe to open a large file or let the file be extended beyond 2GB. The other calls that use file refnums would need to change; even though they could potentially use `XIOParam` to pass their parameters, there would be no way to make sure they were really using `XIOParam` instead of `IOParam`.

The new file I/O APIs in this section are a replacement for the existing (old) file I/O calls.

When a fork is opened for I/O, an **access path** is created. This access path contains state information, including the **current position** (or **mark**), and the **permissions**. Both the old and new APIs identify an access path using a 16-bit **file reference number**. Note that a file reference number obtained from the large file APIs cannot be passed to the older file I/O APIs; you'll get an error if you try.

5.2.4 Identifying Files and Directories

Files and directories are identified using an opaque type, the `FSRef`. In concept, it is similar to an `FSSpec`. Unlike an `FSSpec`, the fields within an `FSRef` are not publicly defined; in fact, they may vary from version to version, between volumes, and even between files or directories on a single volume.

An `FSRef` is intended to identify the same file or directory, even if it is moved or renamed (like a FileID reference). Clearly some volume formats will not be able to identify an object after it has moved or been renamed; in this case, an `FSRef` will identify the object at least as reliably as an `FSSpec`. Since HFS Plus volumes always have file and directory thread records, they fully support the desired behavior. HFS volumes will also fully support this behavior, but possibly with a slight performance degradation if the volume is locked and the file has no thread record (since the File Manager will attempt to find the file first by FileID, and then by parent directory ID and name).

Since the contents of an `FSRef` are opaque, clients must use APIs (such as `FSMakeFSRefUnicode`) to create them or to get information about the file or directory they identify. Since an `FSRef` may contain information about an object's name or location, the `FSMoveObject` and `FSRenameUnicode` calls return an updated `FSRef` that may differ from the one passed as input.

A client may not compare two `FSRef`s for equality by comparing the bytes of the `FSRef`. This allows volume format implementations to cache data in the `FSRef` to speed up future resolution. It also lets volume format implementations store text (such as a filename) in the `FSRef`, and lets them use whatever text comparison is appropriate for their volume format.

5.2.5 File Reference Numbers

The file refnum is an opaque 16-bit quantity. The value zero (0) is never a valid file refnum. You should not assume that file refnums fit any particular pattern. In particular, they are not byte offsets into a global FCB table. The only way to use or get information about a fork refnum is to use the APIs.

The same fork may in fact be opened simultaneously by both the new and old APIs (as long as the permissions allow). This results in two distinct access paths, just like using `PBHOpen` twice on the same fork.[1]

5.2.6 Fork Sizes and Offsets

The new file I/O APIs use a signed 64-bit value for fork sizes and offsets. Because this value is signed, fork sizes are limited to 2^{63} bytes, or 8 billion terabytes. Note that the existing Mac OS 8 driver and disk cache model imposes a maximum volume size limit of 2^{41} bytes, or 2 terabytes, for local volumes.

5.2.7 Informational Calls

The `FSGetCatalogInfo` call is an enhanced version of `PBGetCatInfo` that returns 64-bit fork sizes, Unicode filename strings, and more catalog information (including both the `contentModDate` and `attributeModDate`, permissions, etc.). There is a corresponding enhancement of `SetCatInfo`, called `FSSetCatalogInfo`.

`FSCatalogSearch`, a new version of `PBCatSearch`, allows additional search criteria. It can also return the same kind of file information as `FSGetCatalogInfo` for the matching files or folders. There is a `FSGetCatalogInfoBulk` call which can return information about multiple items in a single directory via a single call.

These new calls use a new structure, `FSCatalogInfo`, that encapsulates the fixed-length catalog information about a file or folder. Other information (such as names and `FSRef`) are returned as separate parameters.

5.2.8 Catalog Iterators

The calls that iterate over a directory or volume's contents (such as `FSGetCatalogInfoBulk` and `FSCatalogSearch`) may require several calls to return all of the items. Some state information must be stored across the sequence of operations. The HFS Plus APIs introduce catalog iterators to maintain this state.

1. However, the File Manager will prevent the case where a fork is opened for writing via the large file APIs, and simultaneously opened for reading or writing by the older APIs. If the large file APIs are used to grow the fork beyond 2GB, the programs accessing the fork via the old APIs might behave badly.

The current PBGetCatInfo and PBCatSearch calls require the caller to maintain all state information. For PBGetCatInfo, the state is a simple numerical index of the item within the directory. For PBCatSearch, the state is a mostly opaque structure of 16 bytes, which requires the caller to initialize the first 4 bytes before the first call.

In the HFS Plus APIs, this state information is stored by the File Manager. The client must explicitly create this state with FSOpenIterator and dispose the state with FSCloseIterator. A reference to this state, of type FSIterator, is returned by FSOpenIterator and passed as input to the other calls.

Conceptually, a catalog iterator is like an open fork. Iterating over the contents of a directory or volume is like reading from a fork, except that you are "reading" catalog entries. The catalog iterator maintains its position within the catalog just like an open fork maintains its current position within the fork.

When a catalog iterator is created, you supply an FSRef for a directory and some flags. Currently, there is only one defined flag. It determines whether the iterator iterates over just the contents of that directory ("flat") or the entire subtree rooted at that directory.

5.2.9 Gestalt Bits

Two new bits have been defined for the gestaltFSAttr selector:

```
enum {
  gestaltHasHFSPlusAPIs       = 12, /* file system supports HFS Plus APIs */
  gestaltMustUseFCBAccessors  = 13  /* FCBSPtr and FSFCBLen are invalid - must
                                       use FSM FCB accessor functions*/
};
```

If the gestaltHasHFSPlusAPIs bit is set, then the File Manager supports the HFS Plus APIs. Individual file systems may or may not implement the HFS Plus APIs; however, the File Manager provides compatibilty support (emulation) for file systems that do not implement the HFS Plus APIs. Applications can call PBHGetVolParms to see which HFS Plus API features are supported on a particular volume.

The gestaltMustUseFCBAccessors bit indicates that the File Manager no longer supports the private low memory globals FCBSPtr and FSFCBLen, and that all access to File or Fork Control Blocks must be made with the File System Manager's utility functions.

5.3 Data Structures and File Formats

5.3.1 FSVolumeRefNum

```
typedef SInt16 FSVolumeRefNum;
```

The `FSVolumeRefNum` type is used to identify a particular mounted volume. This is the same as the 16-bit volume refnum previously passed in the `ioVRefNum` fields of a parameter block; we've simply introduced a new type name for the old data type.

5.3.2 FSRef

```
struct FSRef {
  UInt8    hidden[80];  /* private to File Manager */
};
typedef struct FSRef FSRef;
typedef FSRef *FSRefPtr;
```

The `FSRef` type is used to identify a directory or file (including a volume's root directory). It's purpose is similar to an `FSSpec` except that an `FSRef` is completely opaque. An `FSRef` contains whatever information is needed to find the given object; the internal structure of an `FSRef` is likely to vary based on the volume format, and may vary based on the particular object being identified.

The client of the File Manager cannot examine the contents of an `FSRef` to extract information about the parent directory or the object's name. Similarly, an `FSRef` cannot be constructed directly by the client; the `FSRef` must be constructed and returned via the File Manager. There is no need to call the File Manager to dispose an `FSRef`.

To determine the volume, parent directory and name associated with an `FSRef`, or to get an equivalent `FSSpec`, use the `FSGetCatalogInfo` call.

5.3.3 FSAllocationFlags

```
typedef UInt16 FSAllocationFlags;

enum {
   kFSAllocDefaultFlags    = 0x0000, /* as much as possible, not contiguous */
   kFSAllocAllOrNothingMask= 0x0001, /* allocate all of the space, or none */
   kFSAllocContiguousMask  = 0x0002, /* allocate a single contiguous piece */
   kFSAllocNoRoundUpMask   = 0x0004, /* don't round up to clump size */
   kFSAllocReservedMask    = 0xFFF8  /* reserved; set to zero */
};
```

The `FSAllocationFlags` type is a set of bit flags. It is passed to the `FSAllocateFork` call to control how the space is allocated.

The kAllocContiguousMask is set when an allocation should allocate one contiguous range of space on the volume. If this bit is clear, multiple discontiguous extents may be allocated to fulfill the request.

The kAllocAllOrNothingMask is set when an allocation must allocate the total requested amount, or else fail with nothing allocated; when this bit is not set, the allocation may complete successfully but allocate less than requested.

If kFSAllocNoRoundUpMask is clear, then additional space beyond the amount requested may be allocated; this is done by some volume formats (including HFS and HFS Plus) to avoid many small allocation requests. If the bit is set, no additional allocation is done (except where required by the volume format, such as rounding up to a multiple of the allocation block size).

5.3.4 FSPositionMode

```
typedef UInt32 FSPositionMode;

enum {
  kFSAtMark         = 0,       /* At current position; offset ignored */
  kFSFromStart      = 1,       /* Offset from start of fork */
  kFSFromEnd        = 2,       /* Offset from end of fork */
  kFSFromMark       = 3        /* Offset from current position */

  pleaseCacheBit    = 4,       /* please cache this request */
  pleaseCacheMask   = 0x0010,
  noCacheBit        = 5,       /* please don't cache this request */
  noCacheMask       = 0x0020,
  forceReadBit      = 6,       /* force read from disk, bypassing all caches */
  forceReadMask     = 0x0040,
};
```

The FSPositionMode type is used in conjunction with an offset parameter (of type SInt64) to specify a position within a fork. Use one of kFSAtMark, kFSFromStart, kFSFromEnd, kFSFromMark.

kFSAtMark	Starting point is the access path's current position. The offset is ignored.
kFSFromStart	Starting point is offset bytes from the start of the fork. The offset must be non-negative.
kFSFromEnd	The starting point is offset bytes from the logical end of the fork. The offset must be non-positive.
kFSFromMark	The starting point is offset bytes from the access path's current position. The offset may be positive or negative.

For the FSReadFork and FSWriteFork calls, you may also add either of the pleaseCacheMask or noCacheMask constants to hint whether the data should be cached or not. Adding the pleaseCacheMask constant tells the disk cache that you think that data will be accessed again soon; the data is more likely to be placed in the disk cache.

Adding the `noCacheMask` tells the disk cache that you think the data will not be accessed again soon; whole sectors in the middle of the I/O are less likely to be placed in the disk cache.

Setting the `forceReadBit` will force reads directly from disk, bypassing any data in the cache. Clients can use this to verify that data is stored correctly on the media (eg., to verify after writing) by reading the data into a different buffer while setting the bit, and then comparing the newly read data with the previously written data.

Note

The `forceReadBit` is the same as the `rdVerifyBit` used in the older APIs. The actual implementation of the `rdVerifyBit` in the older APIs actually caused the "force read" behavior, and only compared the data in partial sectors. `FSReadFork` cleans up this behavior by always letting the client do all of the compares. ◆

New-line mode is not supported by `FSReadFork`.

5.3.5 FSCatalogInfoBitmap

```
typedef UInt32 FSCatalogInfoBitmap;

enum {
    kFSCatInfoNone            = 0x00000000,
    kFSCatInfoTextEncoding    = 0x00000001,
    kFSCatInfoNodeFlags       = 0x00000002,  /* Locked (bit 0) and */
                                             /*  directory (bit 4) only */
    kFSCatInfoVolume          = 0x00000004,
    kFSCatInfoParentDirID     = 0x00000008,
    kFSCatInfoNodeID          = 0x00000010,
    kFSCatInfoCreateDate      = 0x00000020,
    kFSCatInfoContentMod      = 0x00000040,
    kFSCatInfoAttrMod         = 0x00000080,
    kFSCatInfoAccessDate      = 0x00000100,
    kFSCatInfoBackupDate      = 0x00000200,
    kFSCatInfoPermissions     = 0x00000400,
    kFSCatInfoFinderInfo      = 0x00000800,
    kFSCatInfoFinderXInfo     = 0x00001000,
    kFSCatInfoValence         = 0x00002000,  /* Folders only, zero for files */
    kFSCatInfoDataSizes       = 0x00004000,  /* Data logical & physical size */
    kFSCatInfoRsrcSizes       = 0x00008000,  /* Resource logical and */
                                             /*  physical size */
    kFSCatInfoSharingFlags    = 0x00010000,  /* sharingFlags */
    kFSCatInfoUserPrivs       = 0x00020000,  /* userPrivileges */
    kFSCatInfoAllDates        = 0x000003E0,
    kFSCatInfoGettableInfo    = 0x0003FFFF,
    kFSCatInfoSettableInfo    = 0x00001FE3,  /* flags, dates, permissions, */
                                             /*  Finder info, text encoding */
    kFSCatInfoReserved        = (long)0xFFFF0000    /* reserved */
};
```

5.3.6 FSCatalogInfo

```
/*  Constants for nodeFlags field of FSCatalogInfo */
enum {
    kFSNodeLockedBit          = 0,
    kFSNodeLockedMask         = 0x0001,
    kFSNodeIsDirectoryBit     = 4,
    kFSNodeIsDirectoryMask    = 0x0010
};

struct FSCatalogInfo {
    UInt16            nodeFlags;         /* node flags */
    FSVolumeRefNum    volume;            /* object's volume ref */
    UInt32            parentDirID;       /* parent directory's ID */
    UInt32            nodeID;            /* file/directory ID */
    UInt8             sharingFlags;      /* kioFlAttribMountedBit and */
                                         /*  kioFlAttribSharePointBit */
    UInt8             userPrivileges;    /* user's effective AFP privileges */
                                         /*  (same as ioACUser) */
    UInt8              reserved1;
    UInt8              reserved2;
    UTCDateTime        createDate;       /* date and time of creation */
    UTCDateTime        contentModDate;   /* date/time of fork modification */
    UTCDateTime        attributeModDate; /* attribute modification date/time*/
    UTCDateTime        accessDate;       /* date and time of last access */
                                         /*  (for Mac OS X) */
    UTCDateTime        backupDate;       /* date and time of last backup */

    UInt32             permissions[4];   /* permissions (for Mac OS X) */

    UInt8              finderInfo[16];   /* Finder information part 1 */
    UInt8              extFinderInfo[16];/* Finder information part 2 */

    UInt64             dataLogicalSize;  /* files only */
    UInt64             dataPhysicalSize; /* files only */
    UInt64             rsrcLogicalSize;  /* files only */
    UInt64             rsrcPhysicalSize; /* files only */

    UInt32             valence;          /* folders only */
    UInt32             textEncodingHint;
};
typedef struct FSCatalogInfo FSCatalogInfo;
typedef FSCatalogInfo *      FSCatalogInfoPtr;
```

The `FSCatalogInfoBitmap` type is used to indicate which fields of the `FSCatalogInfo` should be set or retrieved. If the bit corresponding to a particular field is not set, then that field is not changed if the `FSCatalogInfo` is an output parameter, and that field is ignored if the `FSCatalogInfo` is an input parameter.

The `FSCatalogInfo` structure holds basic information about a file or directory. The `nodeFlags` field contains has two defined bits that indicate whether an object is a file or folder, and whether a file is locked (constants `kFSNodeIsDirectoryMask` and `kFSNodeLockedMask`).

There are several date fields that describe when the object was created (`createDate`), when the data or resource fork was last modified (`contentModDate`), when any other fork was last modified (`attributeModDate`), when the object was last accessed (`accessDate`), and when the object was last backed up (`backupDate`).

The Mac OS 8 File Manager does not automatically update the `accessDate` field; it exists primarily for use by other operating systems. The `backupDate` field is not updated by the File Manager; a backup utility may use this field if it wishes.

The `permissions` field contains user and group permission information. The Mac OS 8 File Manager does not use or enforce this permission information. It could be used by a file server program or other operating system.

The `finderInfo` and `extFinderInfo` fields contain basic Finder information and extended Finder information for the object. This information is available in the catalog info, instead of a named fork, for historical reasons. The File Manager does not interpret the contents of these fields.

The `valence` field is used for directories. It is the number of items (files plus directories) contained within the directory. For files, it is set to zero.

▲ **WARNING**

Many volume formats do not store a field containing a directory's valence. For those volume formats, this field is very expensive to compute. Think carefully before you ask the File Manager to return this field. ▲

The `dataLogicalSize` is the size of the data fork in bytes. The `rsrcLogicalSize` is the size of the resource fork. The `dataPhysicalSize` is the amount of disk space (in bytes) occupied by the data fork. The `rsrcPhysicalSize` is the amount of disk space occupied by the resource fork.

The `parentDirID` field contains the ID of the directory that contains the given object. The root directory of a volume always has ID `fsRtDirID` (`2`); the parent of the root directory is ID `fsRtParID` (`1`). Note: there is no object with ID `fsRtParID`; this is merely used when the File Manager is asked for the parent of the root directory.

The `textEncodingHint` field is used in conjunction with the Unicode filename. It is an optional hint that can be used by the volume format when converting the Unicode to some other encoding. For example, HFS Plus stores this value and uses it when converting the name to a Mac OS encoding, such as when the name is returned by `PBGetCatInfo`. As another example, HFS volumes use this value to convert the Unicode name to a Mac OS encoded name stored on disk. If the entire Unicode name can be converted to a single Mac OS encoding, then that encoding should be used as the `textEncodingHint`; otherwise, a text encoding corresponding to the first characters of the name will probably provide the best user experience.

Note

If a `textEncodingHint` is not supplied when a file or directory is created or renamed, the volume format will use a default value. This default value may not be the best possible choice for the given filename. Whenever possible, a client should supply a `textEncodingHint`. ◆

5.3.7 FSVolumeInfoBitmap

```
typedef UInt32 VolumeInfoBitmap;

enum {
    kFSVolInfoNone           = 0x0000,
    kFSVolInfoCreateDate     = 0x0001,
    kFSVolInfoModDate        = 0x0002,
    kFSVolInfoBackupDate     = 0x0004,
    kFSVolInfoCheckedDate    = 0x0008,
    kFSVolInfoFileCount      = 0x0010,
    kFSVolInfoDirCount       = 0x0020,
    kFSVolInfoSizes          = 0x0040, /* totalBytes and freeBytes */
    kFSVolInfoBlocks         = 0x0080, /* blockSize, totalBlocks, freeBlocks*/
    kFSVolInfoNextAlloc      = 0x0100,
    kFSVolInfoRsrcClump      = 0x0200,
    kFSVolInfoDataClump      = 0x0400,
    kFSVolInfoNextID         = 0x0800,
    kFSVolInfoFinderInfo     = 0x1000,
    kFSVolInfoFlags          = 0x2000,
    kFSVolInfoFSInfo         = 0x4000, /* filesystemID, signature */
    kFSVolInfoDriveInfo      = 0x8000, /* driveNumber, driverRefNum */
    kFSVolInfoGettableInfo   = 0xFFFF,
    kFSVolInfoSettableInfo   = 0x3004  /* backup date, Finder info, flags */
};
```

5.3.8 FSVolumeInfo

```
struct FSVolumeInfo {
    /* Dates -- zero means "never" or "unknown" */
    UTCDateTime   createDate;
    UTCDateTime   modifyDate;
    UTCDateTime   backupDate;
    UTCDateTime   checkedDate;

    /* File/Folder counts -- return zero if unknown */
    UInt32        fileCount;        /* total files on volume */
    UInt32        folderCount;      /* total folders on volume */
                                    /* Note: no root directory counts */
    UInt64        totalBytes;       /* total number of bytes on volume */
    UInt64        freeBytes;        /* number of free bytes on volume */

    /* HFS and HFS Plus specific.  Set fields to zero if not appropriate */
    UInt32        blockSize;   /* size (in bytes) of allocation blocks */
    UInt32        totalBlocks; /* number of allocation blocks in volume */
    UInt32        freeBlocks;  /* number of unused allocation blocks */
    UInt32        nextAllocation;   /* start of next allocation search */
    UInt32        rsrcClumpSize;    /* default resource fork clump size */
    UInt32        dataClumpSize;    /* default data fork clump size */
    UInt32        nextCatalogID;    /* next unused catalog node ID */
    UInt8         finderInfo[32];   /* information used by Finder */
    /* Identifying information */
    UInt16          flags;            /* ioVAtrb */
    UInt16          filesystemID;     /* ioVFSID */
    UInt16          signature;        /* ioVSigWord, unique within an FSID */
    UInt16          driveNumber;      /* ioVDrvInfo */
    short           driverRefNum;     /* ioVDRefNum */
};
typedef struct FSVolumeInfo  FSVolumeInfo;
```

The `FSVolumeInfo` structure is used when getting or setting information about a volume (as a whole; information about a volume's root directory would use the `FSCatalogInfo` type).

The `createDate` is the date/time when the volume was created. The `modifyDate` is the last time when the volume was modified in any way. The `backupDate` is for use by backup utilities to indicate when the volume was last backed up. The `checkedDate` is the last date/time that the volume was checked for consistency.

The `fileCount` and `folderCount` fields are the total number of files and folders on the volume, respectively.

The `totalBytes` is the the size of the volume in bytes. The number of bytes of free space on the volume is `freeBytes`.

The `blockSize` is the size of an allocation block. There are `totalBlocks` allocation blocks on the volume. There are `freeBlocks` unused allocation blocks on the volume. The `nextAllocation` is a hint for where to start searching for free space during an allocation. These four fields are only appropriate for volume formats (such as HFS and HFS Plus) that allocate space in fixed-size pieces; other volume formats may not have a similar concept, and will set some or all of these fields to zero.

The `rsrcClumpSize` and `dataClumpSize` are the clump sizes for the resource and data forks. When a fork is automatically grown as it is written, the File Manager attempts to allocate space that is a multiple of the clump size. These fields are zero for volume formats that don't support the notion of a clump size.

Some volume formats (such as HFS and HFS Plus) use a monotonically increasing number for the catalog node ID (i.e. File ID or Directory ID) of newly created files and directories. For those volume formats, the `nextCatalogID` is the next file/directory ID that will be assigned. For other volume formats, this field will be zero.

The `finderInfo` is information used by the Finder, such as the Directory ID of the System Folder. Some volume formats do not support Finder info for a volume and will set this field to all zeroes.

The `flags` field contains bit flags about the volume .

The `fileSystemID` identifies the filesystem implementation that is handling the volume; this is zero for HFS and HFS Plus volumes. The `signature` is used to distinguish between volume formats supported by a single filesystem implementation.

The `driveNumber` and `driverRefNum` fields contain the drive number and driver reference number for the drive (drive queue element) associated with the volume.

5.3.9 FSIterator

```
typedef struct OpaqueFSIterator*  FSIterator;
```

An `FSIterator` refers to a position within the catalog, used when iterating over files and folders in a directory. It is like a file reference number because it maintains state internally to the File Manager and must be explicitly opened and closed.

An `FSIterator` is returned by `FSOpenIterator` and passed as input to `FSGetCatalogInfoBulk`, `FSCatalogSearch` and `FSCloseIterator`.

5.3.10 HFSUniStr255

```
struct HFSUniStr255 {
  UInt16   length;          /* number of unicode characters */
  UniChar  unicode[255];    /* unicode characters */
};
typedef struct HFSUniStr255   HFSUniStr255;
typedef const HFSUniStr255 *  ConstHFSUniStr255Param;
```

The `HFSUniStr255` type is used by the File Manager to return Unicode strings. It is a string of up to 255 16-bit Unicode characters, with a preceding 16-bit length (number of characters). Note that only the first `length` characters have meaningful values; the remaining characters may be set to arbitrary values. A caller should always assume that the entire structure will be modified, even if the actual string length is less than 255.

5.4 New Error Codes

```
enum {
    errFSBadFSRef              = -1401, /* FSRef parameter is bad */
    errFSBadForkName           = -1402, /* Fork name parameter is bad */
    errFSBadBuffer             = -1403, /* A buffer parameter was bad */
    errFSBadForkRef            = -1404, /* A ForkRefNum parameter was bad */
    errFSBadInfoBitmap         = -1405, /* A CatalogInfoBitmap or */
                                        /*  VolumeInfoBitmap has reserved */
                                        /*  or invalid bits set */
    errFSMissingCatInfo        = -1406, /* A CatalogInfo parameter was NULL */
    errFSNotAFolder            = -1407, /* Expected a folder, got a file */
    errFSForkNotFound          = -1409, /* Named fork does not exist */
    errFSNameTooLong           = -1410, /* File or fork name is too long */
                                        /*  to create or rename */
    errFSMissingName           = -1411, /* A Unicode name parameter was NULL */
    errFSBadPosMode            = -1412, /* Newline bits set in positionMode */
    errFSBadAllocFlags         = -1413, /* Reserved or invalid bits set in */
                                        /*  an FSAllocationFlags parameter */
    errFSNoMoreItems           = -1417, /* Iteration done; no more items */
    errFSBadItemCount          = -1418, /* maximumItems was zero */
    errFSBadSearchParams       = -1419, /* Search criteria passed to */
                                        /*  CatalogSearch is invalid */
    errFSRefsDifferent         = -1420, /* Refs are for different objects */
    errFSForkExists            = -1421, /* Named fork already exists. */
    errFSBadIteratorFlags      = -1422, /* Flags passed to FSOpenIterator */
                                        /*  are bad */
    errFSIteratorNotFound      = -1423, /* Passed FSIterator is not an open */
                                        /*  iterator */
    errFSIteratorNotSupported= -1424    /* The iterator's flags or container */
                                        /*  are not supported by this call */
};
```

The following are new error codes that may be returned by the new APIs.

Note

During the development process, the constants assigned to these error codes may change. ◆

5.4.1 errFSBadFSRef

An `FSRef` parameter was invalid. There are several possible causes:

- The parameter was not optional, but the pointer was `NULL`.
- The volume refnum contained within the `FSRef` does not match a currently mounted volume. This can happen if the volume was unmounted after the `FSRef` was created.
- Some other private field inside the `FSRef` contains a value that could never be valid. If the field value *could* be valid, but doesn't happen to match the existing

volume or in-memory structures, a "not found" error would be returned instead.

5.4.2 errFSBadForkName

A supplied fork name was invalid (syntactically illegal for the given volume). For example, the fork name might contain characters that cannot be stored on the given volume (such as a colon on HFS volumes).

Some volume formats do not store fork names in Unicode. Those volume formats will attempt to convert the Unicode name to the kind of encoding used by the volume format. If the name could not be converted, `errFSBadForkName` is returned.

Some volume formats only support a limited set of forks (such as the data and resource forks on HFS volumes). For those volumes, if any other fork name is passed, `errFSBadForkName` is returned.

5.4.3 errFSBadBuffer

A non-optional buffer pointer was `NULL`, or its size was invalid for the type of data it was expected to contain. In a protected memory system, this could also mean the buffer space is not part of the address space for the calling process.

5.4.4 errFSBadForkRef

A file reference number does not correspond to a fork opened with `FSOpenFork`. This could be because that fork has already been closed. Or, you may have passed a refnum created with the older APIs (eg., by PBHOpenDF). A value of zero is never a valid file reference number.

5.4.5 errFSBadInfoBitmap

A `FSCatalogInfoBitmap` (see Section 5.3.6, "FSCatalogInfo") or `FSVolumeInfoBitmap` (see Section 5.3.8, "FSVolumeInfo") has one or more reserved/undefined bits set. Can also be returned if a defined bit is set, but the corresponding `FSCatalogInfo` or `FSVolumeInfo` field cannot be operated on with that call (for example, trying to use `FSSetCatalogInfo` to set the valence of a directory).

5.4.6 errFSMissingCatInfo

A `FSCatalogInfo` pointer is `NULL`, but is not optional. Or, if the `FSCatalogInfo` is optional and `NULL`, but the corresponding `CatalogInfoBitmap` is not zero (that is, the bitmap says that one or more of the `FSCatalogInfo` fields is being passed, but the supplied pointer was `NULL`).

5.4.7 errFSNotAFolder

A parameter was expected to identify a folder, but it identified some other kind of object (eg., a file) instead. This implies that the specified object exists, but is of the wrong type. For example, one of the parameters to `FSCreateFileUnicode` is an `FSRef` of the directory where the file will be created; if the `FSRef` actually refers to a file, this error is returned.

5.4.8 errFSForkNotFound

An attempt to specify a fork of a given file or directory, but that particular fork does not exist.

5.4.9 errFSNameTooLong

A file or fork name was too long. This means that the given name could never exist; this is different from a "file not found" or `errFSForkNotFound` error.

5.4.10 errFSMissingName

A required file or fork name parameter was a `NULL` pointer, or the length of a filename was zero.

5.4.11 errFSBadPosMode

Reserved or invalid bits in a `positionMode` field were set. For example, the `FSReadFork` call does not support newline mode, so setting the newline bit or a newline character in the `positionMode` would cause this error.

5.4.12 errFSBadAllocFlags

Reserved or invalid bits were set in an `FSAllocationFlags` parameter.

5.4.13 errFSNoMoreItems

There are no more items to return when enumerating a directory or searching a volume. (Note that `FSCatalogSearch` returns `errFSNoMoreItems` whereas `PBCatSearch` would return `eofErr`.)

5.4.14 errFSBadItemCount

The `maximumObjects` parameter to `FSGetCatalogInfoBulk` or `FSCatalogSearch` was zero.

5.4.15 errFSBadSearchParams

The search criteria to `FSCatalogSearch` are invalid or inconsistent.

5.4.16 errFSRefsDifferent

The two `FSRef`s passed to `FSCompareFSRefs` are for different files or directories. Note that a volume format may be able to compare the `FSRef`s without searching for the files or directories, so this error may be returned even if one or both of the `FSRef`s refers to non-existant objects.

5.4.17 errFSForkExists

An attempt to create a fork (see Section 5.6.11, "FSCreateFork"), but that fork already exists.

5.4.18 errFSBadIteratorFlags

The flags passed to `FSOpenIterator` are invalid, such as setting a bit that is currently reserved.

5.4.19 errFSIteratorNotFound

The value of an `FSIterator` parameter does not correspond to any currently open iterator.

5.4.20 errFSIteratorNotSupported

The iterator flags or container of an `FSIterator` are not supported by that call. For example, in the initial release, the `FSCatalogSearch` call only supports an iterator whose container is the volume's root directory and whose flags are `kFSIterateSubtree` (i.e. an iterator for the entire volume's contents). Similarly, in the initial release, `FSGetCatalogInfoBulk` only supports an iterator whose flags are `kFSIterateFlat`.

5.5 Parameter Blocks

The following sections list the parameter blocks used by the low-level (parameter block based) APIs. The Interfaces sections describe which fields are actually used for individual calls.

5.5.1 FSRefParam

```
struct FSRefParam {
    QElemPtr                qLink;              /*queue link in header*/
    short                   qType;              /*type byte for safety check*/
    short                   ioTrap;             /*FS: the Trap*/
    Ptr                     ioCmdAddr;          /*FS: address to dispatch to*/
    IOCompletionUPP         ioCompletion;       /*completion routine addr */
    volatile OSErr          ioResult;           /*result code*/
    ConstStringPtr          ioNamePtr;          /*ptr to Vol:FileName string*/
    short                   ioVRefNum;          /*volume refnum */

    SInt16                  reserved1;          /* was ioRefNum */
    UInt8                   reserved2;          /* was ioVersNum */
    UInt8                   reserved3;          /* was ioPermssn */

    const FSRef *           ref;                /* Input ref; */
                                                /*  the target of the call */
    FSCatalogInfoBitmap     whichInfo;
    FSCatalogInfo *         catInfo;
    UniCharCount            nameLength;         /* input name length */
                                                /*  for create/rename */
    const UniChar *         name;               /* input name */
                                                /*  for create/rename */
    long                    ioDirID;
    FSSpec *                spec;               /* target (source) FSRef */
    FSRef *                 parentRef;          /* secondary (dest) FSRef */
    FSRef *                 newRef;             /* output ref */
    TextEncoding            textEncodingHint;   /* for Rename
                                                /*  and MakeFSRefUnicode */
    HFSUniStr255 *          outName;            /* Output name
                                                /*  for GetCatalogInfo */
};
typedef struct FSRefParam   FSRefParam;
typedef FSRefParam *        FSRefParamPtr;
```

5.5.2 FSForkIOParam

```
struct FSForkIOParam {
    QElemPtr                  qLink;              /*queue link in header*/
    short                     qType;              /*type byte for safety check*/
    short                     ioTrap;             /*FS: the Trap*/
    Ptr                       ioCmdAddr;          /*FS: address to dispatch to*/
    IOCompletionUPP           ioCompletion;       /*completion routine addr */
    volatile OSErr            ioResult;           /*result code*/
    void *                    reserved1;          /* was ioNamePtr */
    SInt16                    reserved2;          /* was ioVRefNum */
    SInt16                    forkRefNum;         /* same as ioRefNum */
    UInt8                     reserved3;          /* was ioVersNum */
    SInt8                     permissions;        /* DO NOT MOVE THIS FIELD */
    const FSRef *             ref;                /* which object to open */
    Ptr                      buffer;          /*data buffer Ptr*/
    UInt32                   requestCount;    /*requested byte count*/
    UInt32                   actualCount;     /*actual byte count completed*/
    UInt16                   positionMode;    /*initial file positioning*/
    SInt64                   positionOffset;  /*file position offset*/

    FSAllocationFlags        allocationFlags;
    UInt64                   allocationAmount;

    UniCharCount             forkNameLength;  /* input; fork name length */
    UniChar *                forkName;        /* input; fork name */

    CatPositionRec           forkIterator;
    HFSUniStr255 *           outForkName;     /* output; fork name */
};
typedef struct FSForkIOParam FSForkIOParam;
typedef FSForkIOParam *      FSForkIOParamPtr;
```

5.5.3 FSCatalogBulkParam

```
struct FSCatalogBulkParam {
    QElemPtr                 qLink;        /* queue link in header */
    short                    qType;        /* type byte for safety check */
    short                    ioTrap;       /* FS: the Trap*/
    Ptr                      ioCmdAddr;    /* FS: address to dispatch to */
    IOCompletionUPP          ioCompletion;     /* completion routine addr */
    volatile OSErr           ioResult;         /* result code */
    Boolean                  containerChanged; /* true if container changed */
                                               /*  since last iteration */
    UInt8                    reserved;         /* align following fields */
    FSIteratorFlags          iteratorFlags;
    FSIterator               iterator;
    const FSRef *            container;     /* directory/volume to iterate */
    ItemCount                maximumItems;
    ItemCount                actualItems;
    FSCatalogInfoBitmap      whichInfo;
    FSCatalogInfo *          catalogInfo;      /* returns an array */
    FSRef *                  refs;             /* returns an array */
    FSSpec *                 specs;            /* returns an array */
    HFSUniStr255 *           names;            /* returns an array */
    const FSSearchParams *   searchParams;
};
typedef struct FSCatalogBulkParam  FSCatalogBulkParam;
typedef FSCatalogBulkParam * FSCatalogBulkParamPtr;
```

5.5.4 FSForkCBInfoParam

```
struct FSForkCBInfoParam {
    QElemPtr                 qLink;‘          /* queue link in header */
    short                    qType;           /* type byte for safety check */
    short                    ioTrap;          /* FS: the Trap */
    Ptr                      ioCmdAddr;       /* FS: address to dispatch to */
    IOCompletionUPP          ioCompletion;    /* completion routine addr */
    volatile OSErr           ioResult;        /* result code */
    SInt16                   desiredRefNum;   /* 0 to iterate, non-0 for */
                                              /*  specific refnum */
    SInt16                   volumeRefNum;    /* volume to match, */
                                              /*  or 0 for all */
    SInt16                   iterator;        /* 0 to start iteration */
    SInt16                   actualRefNum;    /* actual refnum found */

    FSRef *                  ref;
    FSForkInfo *             forkInfo;
    HFSUniStr255 *           forkName;
};
typedef struct FSForkCBInfoParam   FSForkCBInfoParam;
typedef FSForkCBInfoParam *  FSForkCBInfoParamPtr;
```

5.5.5 FSVolumeInfoParam

```
struct FSVolumeInfoParam {
    QElemPtr                 qLink;        /* queue link in header */
    short                    qType;        /* type byte for safety check */
    short                    ioTrap;       /* FS: the Trap */
    Ptr                      ioCmdAddr;    /* FS: address to dispatch to */
    IOCompletionUPP          ioCompletion; /* completion routine addr */
    volatile OSErr           ioResult;     /* result code */
    StringPtr                ioNamePtr;    /* unused */
    FSVolumeRefNum           ioVRefNum;    /* volume refnum */

    UInt32                   volumeIndex;  /* index, or 0 to use ioVRefNum */
    FSVolumeInfoBitmap       whichInfo;    /* which volumeInfo to get/set */
    FSVolumeInfo *           volumeInfo;   /* information about the volume */
    HFSUniStr255 *           volumeName;   /* output; ptr to volume name */
    FSRef *                  ref;          /* volume's FSRef */
};
typedef struct FSVolumeInfoParam   FSVolumeInfoParam;
typedef FSVolumeInfoParam *  FSVolumeInfoParamPtr;
```

5.6 Interfaces

The calls listed here are the high-level variants. Each call has two corresponding parameter block-based calls (one sync, one async). The parameter block-based calls take the same parameters, but they are fields in a larger parameter block instead of separate function arguments. In many cases, a single parameter block type is shared by multiple calls; some fields are unused by some calls. Some of the parameter block fields are input/output, so a single field is used where there would be separate input and output parameters for the corresponding high-level call.

5.6.1 PBHGetVolParms

```
OSErr PBHGetVolParms          (HParmBlkPtr            paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ioNamePtr       A pointer to the volume’s name
    ->  ioVRefNum       A volume specification
    ->  ioBuffer        A pointer to the GetVolParmsInfoBuffer record
    ->  ioReqCount      The size of the bufffer area, in bytes
    <-  ioActCount      The size of the data actually returned
```

The `PBHGetVolParms` function returns information about the characteristics of a volume. You specify a volume (either by name or by volume reference number) and a buffer size, and `PBHGetVolParms` fills in the volume attributes buffer, as described in this section.

You can use a name (pointed to by the `ioNamePtr` field) or a volume specification (contained in the `ioVRefNum` field) to specify the volume. A volume specification can be a volume reference number, drive number, or working directory reference number. If you use a volume specification to specify the volume, you should set the `ioNamePtr` field to `NULL`.

You must allocate memory to hold the returned attributes and put a pointer to the buffer in the `ioBuffer` field. Specify the size of the buffer in the `ioReqCount` field. The `PBHGetVolParms` function places the attributes information in the buffer pointed to by the `ioBuffer` field and specifies the actual length of the data in the `ioActCount` field.

```
struct GetVolParmsInfoBuffer {
  short  vMVersion;       /*version number*/
  long   vMAttrib;        /*bit vector of attributes (see constants below)*/
  Handle vMLocalHand;     /*handle to private data*/
  long   vMServerAdr;     /*AppleTalk server address or zero*/

  /*  vMVersion 1 GetVolParmsInfoBuffer ends here  */

  long   vMVolumeGrade;   /*approx. speed rating or zero if unrated*/
  short  vMForeignPrivID; /*foreign privilege model supported, zero if none*/

  /*  vMVersion 2 GetVolParmsInfoBuffer ends here  */

  long   vMExtendedAttributes;  /*extended attribute bits (see below)*/

  /*  vMVersion 3 GetVolParmsInfoBuffer ends here */
};
typedef struct GetVolParmsInfoBuffer    GetVolParmsInfoBuffer;
```

The `vMVersion` is the version of the attributes buffer structure, as returned by the volume format’s implementation. Currently, this field returns 1, 2, or 3. Version 3 was introduced to support the HFS Plus APIs.

The `vMAttrib` field consists of many bits that describe the volume's attributes. For details on these bits, see *Inside Macintosh: Files* and its errata (available on the Apple Developer web site).

The `vMLocalHand` field contains a handle to private data for shared volumes. On creation of the VCB (just after mounting), this field is a handle to a 2-byte block of memory. The Finder uses this for its local window list storage, allocating and deallocating memory as needed. It is disposed of when the volume is unmounted. Your application should treat this field as reserved.

For AppleShare server volumes, the `vMServer` field contains the network address of an AppleTalk server volume. Your application can inspect this field to tell which volumes belong to which server; the value of this field is 0 if the volume does not have a server.

The `vMVolumeGrade` is the relative speed rating of the volume. The scale used to determine these values is currently uncalibrated. In general, lower values indicate faster speeds. A value of 0 indicates the volume's speed is unrated. The buffer version returned in `vMVersion` must be greater than 1 for this field to be meaningful.

The `vMForeignPrivID` is ain integer representing the privilege model supported by the volume. Currently two values are defined for this field: 0 represents a standard HFS or HFS Plus volume that might or might not support the AFP privilege model; `fsUnixPriv` represents a volume that supports the A/UX privilege model. The buffer version returned in `vMVersion` must be greater than 1 for this field to be meaningful.

The `vMExtendedAttributes` field, like the `vMAttrib` field, contains bits that describe a volume's attributes. For this field to be meaningful, the `vMVersion` must be greater than 2. The bits currently defined in `vMExtendedAttributes` are:

```
enum {
  bIsEjectable                = 0, /* volume is in an ejectable disk drive */
  bSupportsHFSPlusAPIs        = 1, /* volume supports HFS Plus APIs directly
                                      (not through compatibility layer) */
  bSupportsFSCatalogSearch    = 2, /* volume supports FSCatalogSearch */
  bSupportsFSExchangeObjects  = 3, /* volume supports FSExchangeObjects */
  bSupports2TBFiles           = 4, /* volume supports 2 terabyte files */
  bSupportsLongNames          = 5, /* volume supports file/directory/volume
                                      names longer than 31 characters */
  bSupportsMultiScriptNames   = 6, /* volume supports file/directory/volume
                                      names with characters from multiple
                                      script systems */
  bSupportsNamedForks         = 7, /* volume supports forks beyond the data
                                      and resource forks */
  bSupportsSubtreeIterators   = 8, /* volume supports recursive iterators
                                      not at the volume root */
  bL2PCanMapFileBlocks        = 9  /* volume supports Lg2Phys SPI correctly */
};
```

Volumes that implement the HFS Plus APIs must version 3 (or newer) of the `GetVolParmsInfoBuffer`. Volumes that don't implement the HFS Plus APIs may still implement version 3 of the `GetVolParmsInfoBuffer`. If the version of the `GetVolParmsInfoBuffer` is 2 or less, or the `bSupportsHFSPlusAPIs` bit is clear (zero), then the volume does not implement the HFS Plus APIs, and they are being emulated for that volume by the File Manager itself.

If a volume does not implement the HFS Plus APIs, and supports version 2 or earlier version of the `GetVolParmsInfoBuffer`, it cannot itself describe whether it supports the `FSCatalogSearch` or `FSExchangeObjects` calls. The compatibility layer will implement the `FSCatalogSearch` call if the volume supports the `PBCatSearch` call (i.e. the `bHasCatSearch` bit of `vMAttrib` is set). The compatibility layer will implement the `FSExchangeObjects` call if the volume supports `PBExchangeFiles` (i.e. the `bHasFileIDs` bit of `vMAttrib` is set).

5.6.2 FSMakeFSRefUnicode

```
OSErr FSMakeFSRefUnicode      (const FSRef *          parentRef,
                               UniCharCount           nameLength,
                               const UniChar *        name,
                               TextEncoding           textEncodingHint,
                               FSRef *                newRef);

OSErr PBMakeFSRefUnicodeSync  (FSRefParam *           paramBlock);
OSErr PBMakeFSRefUnicodeAsync (FSRefParam *           paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             A pointer to the parent directory FSRef
    ->  nameLength      The length of the Unicode Name
    ->  name            A pointer to Unicde name
    ->  textEncodingHint A suggested text encoding to use for the name
    <-  newRef          A pointer to an FSRef
```

The `FSMakeFSRefUnicode` call lets you construct an `FSRef` given an `FSRef` for the parent directory, and Unicode name. The Unicode name must be a leaf name; partial or full pathnames are not allowed.[2]

An `FSRef` for the directory containing the object is passed in `parentDirectory`. The name of the object is specified with Unicode; the length (in Unicode characters) is passed in `nameLength`; `name` points to the Unicode characters. A text encoding hint for the name is passed in `textEncodingHint`; if you pass `kTextEncodingUnknown`, the File Manager will use a default value.

If the result is `noErr`, an `FSRef` for the object is returned in `ref`.

5.6.3 FSpMakeFSRef

```
OSErr FSpMakeFSRef            (const FSSpec *         source,
                               FSRef *                newRef);

OSErr PBMakeFSRefSync         (FSRefParam *           paramBlock);
OSErr PBMakeFSRefAsync        (FSRefParam *           paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ioNamePtr       A pointer to a pathname
    ->  ioVRefNum       A volume specification
    ->  ioDirID         A directory ID
    <-  newRef          A pointer to an FSRef
```

The FSpMakeFSRef call lets you convert an `FSSpec` to an `FSRef`. (To obtain an `FSSpec` from an `FSRef`, use the `FSGetCatalogInfo` call.)

2. If you have a partial or full pathname in Unicode, you will have to parse it yourself and make multiple FSMakeFSRef calls.

The `fsSpec` parameter must contain a valid `FSSpec` for an existing file or directory; if not, the call will return `fnfErr`. The File Manager will contstruct and return, in the `ref` parameter, an `FSRef` for the same file or directory.

Note

For the parameter block based calls, the fields of the source `FSSpec` are passed as separate parameters. This allows the call to be dispatched to external filesystems the same way as other `FSp` calls are. ◆

5.6.4 FSCompareFSRefs

```
OSErr FSCompareFSRefs          (const FSRef *           ref1,
                                const FSRef *           ref2);

OSErr PBCompareFSRefsSync      (FSRefParam *            paramBlock);
OSErr PBCompareFSRefsAsync     (FSRefParam *            paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             A pointer to the first FSRef (ref1)
    ->  parentRef       A pointer to the second FSRef (ref2)
```

The FSCompareFSRefs call lets you determine whether two `FSRef`s refer to the same file or directory. It is not possible to compare the `FSRef` structures directly since some bytes may be uninitialized, case-insensitive text, or contain hint information.

If the two `FSRef`s refer to the same file or directory, then `noErr` is returned. If they refer to objects on different volumes, then `diffVolErr` is returned. If they refer to different files or directories on the same volume, then `errFSRefsDifferent` is returned. This call may return other errors, including `nsvErr`, `fnfErr`, `dirNFErr`, and `volOffLinErr`.

Note: some volume formats may be able to tell that two `FSRef`s would refer to two different files or directories, without having to actually find those objects. In this case, the volume format may return `errFSRefsDifferent` even if one or both objects no longer exist. Similarly, if the refs are for objects on different volumes, the File Manager will return `diffVolErr` even if one or both volumes are no longer mounted.

5.6.5 FSCreateFileUnicode

```
OSErr FSCreateFileUnicode  (const FSRef *          parentRef,
                           UniCharCount           nameLength,
                           const UniChar *        name,
                           FSCatalogInfoBitmap    whichInfo,
                           const FSCatalogInfo *  catalogInfo, /*can be NULL*/
                           FSRef *                newRef,      /*can be NULL*/
                           FSSpec *               newSpec);    /*can be NULL*/

OSErr PBCreateFileUnicodeSync (FSRefParam *           paramBlock);
OSErr PBCreateFileUnicodeAsync(FSRefParam *           paramBlock);
    ->  ioCompletion   A pointer to a completion routine
    <-  ioResult       The result code of the function
    ->  ref            The directory where the file is to be created
                       (parentRef)
    ->  nameLength     Number of Unicode characters in the file's name
    ->  name           A pointer to the Unicode name
    ->  whichInfo      Which catalog info fields to set; may be NULL
    ->  catInfo        The values for catalog info fields to set; may be NULL
    <-  newRef         A pointer to the FSRef for the new file; may be NULL
    <-  spec           A pointer to the FSSpec for the new file; may be NULL
```

The `FSCreateFileUnicode` call will create a new file in the directory specified by `parentRef`. The file's new name is passed in `name`; the length of the name (in Unicode characters) is passed in `nameLength`. You may optionally set catalog information for the file using the `whichInfo` and `catalogInfo` parameters; this is equivalent to calling `FSSetCatalogInfo` after creating the file. If possible, you should set the `textEncodingHint` in the `catalogInfo`.

An `FSRef` for the new file is (optionally) returned in `newRef`. An `FSSpec` for the new file is (optionally) returned in `newSpec`.

5.6.6 FSCreateDirectoryUnicode

```
OSErr FSCreateDirectoryUnicode
```

```
                        (const FSRef *          parentRef,
                         UniCharCount           nameLength,
                         const UniChar *        name,
                         FSCatalogInfoBitmap    whichInfo,
                         const FSCatalogInfo *  catalogInfo, /* can be NULL */
                         FSRef *                newRef,      /* can be NULL */
                         FSSpec *               newSpec,     /* can be NULL */
                         UInt32 *               newDirID);   /* can be NULL */

OSErr PBCreateDirectoryUnicodeSync    (FSRefParam *           paramBlock);
OSErr PBCreateDirectoryUnicodeAsync   (FSRefParam *           paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             The parent directory where the new directory
                        is to be created
    ->  nameLength      Number of Unicode characters in the directory's name
    ->  name            A pointer to the Unicode name
    ->  whichInfo       Which catalog info fields to set; may be NULL
    ->  catInfo         The values for catalog info fields to set; may be NULL
    <-  newRef          A pointer to the FSRef for the directory; may be NULL
    <-  spec            A pointer to the FSSpec for the directory; may be NULL
    <-  ioDirID         The directory ID of the directory
```

The `FSCreateDirectoryUnicode` call will create a directory (folder) inside the directory specified by the `parentRef` parameter. The new directory's new name is passed in `name`; the length of the name (in Unicode characters) is passed in `nameLength`. You may optionally set catalog information for the new directory using the `whichInfo` and `catalogInfo` parameters; this is equivalent to calling `FSSetCatalogInfo` after creating the directory. If possible, you should set the `textEncodingHint` in the `catalogInfo`.

An `FSRef` for the new directory is (optionally) returned in `newRef`. An `FSSpec` for the directory is optionally returned in `newSpec`. The directory's ID is optionally returned in `newDirID` (but always returned in the parameter block form of the call).

5.6.7 FSDeleteObject

```
OSErr FSDeleteObject          (const FSRef *          ref);

OSErr PBDeleteObjectSync      (FSRefParam *           paramBlock);
OSErr PBDeleteObjectAsync     (FSRefParam *           paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             The file or directory to be deleted
```

The `FSDeleteObject` call will delete a file or an empty directory (one that contains no files or folders). The object is specified by the `ref` parameter.

5.6.8 FSMoveObject

```
OSErr FSMoveObject     (const FSRef *          ref,
                        const FSRef *          destDirectory,
                        FSRef *                newRef);     /* can be NULL */

OSErr PBMoveObjectSync        (FSRefParam *           paramBlock);
OSErr PBMoveObjectAsync       (FSRefParam *           paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             The file or directory to be moved
    ->  parentRef       The file or directory will be moved into
                        this directory
    <-  newRef          A new FSRef for the file or directory in its
                        new location; optional, may be NULL
```

The `FSMoveObject` call moves the file or directory specified by `ref` into the directory specified by `destDirectory`. The new `FSRef` for the file or directory is (optionally) returned in `newRef`.

If `destDirectory` specifies a non-existent object, `dirNFErr` is returned. If `destDirectory` refers to a file, then `errFSNotAFolder` is returned. If `destDirectory` is on a different volume than `ref`, `diffVolErr` is returned.

5.6.9 FSRenameUnicode

```
OSErr FSRenameUnicode         (const FSRef *          ref,
                               UniCharCount           nameLength,
                               const UniChar *        name,
                               TextEncoding           textEncodingHint,
                               FSRef *                newRef); /*can be NULL*/

OSErr PBRenameUnicodeSync     (FSRefParam *           paramBlock);
OSErr PBRenameUnicodeAsync    (FSRefParam *           paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             The file or directory to be moved
    ->  nameLength      Number of Unicode characters in the new name
    ->  name            A pointer to the new Unicode name
    ->  textEncodingHint A suggested text encoding to use for the name
    <-  newRef          A new FSRef for the file or directory; may be NULL
```

The `FSRenameUnicode` call allows you to rename a file or folder. The object to rename is specified by the `oldRef` parameter. The new name is specified by the `name` and `nameLength` parameters. The text encoding hint for the new name is passed in `textEncodingHint`; pass `kTextEncodingUnknown` to use a default value.

A new `FSRef` for the renamed object is (optionally) returned in `newRef`.

5.6.10 FSExchangeObjects

```
OSErr FSExchangeObjects        (const FSRef *           ref,
                                 const FSRef *           destRef);

OSErr PBExchangeObjectsSync    (FSRefParam *           paramBlock);
OSErr PBExchangeObjectsAsync   (FSRefParam *           paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             The first file
    ->  parentRef       The second file (destRef)
```

The `FSExchangeObjects` call allows programs to implement a “safe save” operation by creating and writing a complete new file and swapping their contents (which means that an alias, `FSSpec` or `FSRef` that refers to the old file will now access the new data).

The corresponding information in in-memory data structures are also exchanged. Either or both files may have open access paths. After the exchange, the access path will refer to the opposite file’s data (that is, to the same data it originally referred, which is now part of the other file).

5.6.11 FSCreateFork

```
OSErr FSCreateFork      (const FSRef *           ref,
                         UniCharCount            forkNameLength,
                         const UniChar *         forkName); /* can be NULL */

OSErr PBCreateForkSync         (FSForkIOParam *        paramBlock);
OSErr PBCreateForkAsync        (FSForkIOParam *        paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             The file or directory
    ->  forkNameLength  The length of the fork name (in Unicode characters)
    ->  forkName        The name of the fork to open (in Unicode)
```

The `FSCreateFork` call lets a program create a named fork for a file or directory. The file or directory is specified by the `ref`. The name of the fork is specified by `forkNameLength` and `forkName`. A newly created fork has zero length (i.e. its logical end-of-file is zero).

If the named fork already exists, `errFSForkExists` is returned. If the fork name is syntactically invalid or otherwise unsupported for the given volume, `errFSBadForkName` or `errFSNameTooLong` is returned.

The data and resource forks of a file are automatically created and deleted as needed (for compatibility with older APIs, and because they’re often handled specially). If a given fork always exists for a given volume format (such as data and resource forks for HFS and HFS Plus, or data forks for most other volume formats), an attempt to create that fork when a zero-length fork already exists should return `noErr`; if a non-empty fork already exists then `errFSForkExists` should be returned.

5.6.12 FSDeleteFork

```
OSErr FSDeleteFork      (const FSRef *           ref,
                         UniCharCount            forkNameLength,
                         const UniChar *         forkName);   /* can be NULL */

OSErr PBDeleteForkSync         (FSForkIOParam *        paramBlock);
OSErr PBDeleteForkAsync        (FSForkIOParam *        paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             The file or directory
    ->  forkNameLength  The length of the fork name (in Unicode characters)
    ->  forkName        The name of the fork to open (in Unicode)
    --  permissions     Field may be modified
    --  forkRefNum      Field may be modified
    --  positionMode    Field may be modified
    --  positionOffset  Field may be modified
```

The `FSDeleteFork` call allows a program to delete a named fork of a file or directory. Any storage allocated to that fork is released. The file or directory is specified by `ref`. The name of the fork is specified by `forkNameLength` and `forkName`. If the named fork does not exist, `errFSForkNotFound` is returned.

If a given fork always exists for a given volume format (such as data and resource forks for HFS and HFS Plus, or data forks for most other volume formats), this is equivalent to setting the logical size of the fork to zero.

5.6.13 FSIterateForks

```
OSErr FSIterateForks   (const FSRef *    ref,
                        CatPositionRec * forkIterator,
                        HFSUniStr255 *   forkName,          /* can be NULL */
                        UInt64 *         forkSize,          /* can be NULL */
                        UInt64 *         forkPhysicalSize); /* can be NULL */

OSErr PBIterateForksSync    (FSForkIOParam *    paramBlock);
OSErr PBIterateForksAsync   (FSForkIOParam *    paramBlock);
    ->  ioCompletion    A pointer to a completion routine.
    <-  ioResult        The result code of the function.
    ->  ref             The file or directory containing the forks.
    <>  forkIterator    Maintains state between calls for a given FSRef.
                        Before the first call, set the initialize field to 0.
    <-  outForkName     The name of the fork in Unicode.
    <-  positionOffset  The length of the fork, in bytes (forkSize).
    <-  allocationAmount The space allocated to the fork (forkPhysicalSize).
```

The `FSIterateForks` call allows a program to determine the name and size of every named fork belonging to a file or directory. The file or directory is specified by `ref`. Since information is returned about one fork at a time, several calls may be required to iterate through all the forks. The `forkIterator` field maintains state between a series of calls; set the `initialize` field to zero before the first call. The `forkIterator` will be updated after the call completes; the updated iterator should be passed into the next call.

The name of the fork is (optionally) returned in the structure pointed to by `forkName`.

The fork's logical size in bytes is returned in `forkSize`. The fork's physical size (i.e. the amount of space allocated on disk, in bytes) is returned in `forkPhysicalSize`. Both of these outputs are optional; if you don't want the value, set the pointer to `NULL`.

There is no guarantee about the order in which forks are returned. The order may vary between iterations.

5.6.14 FSGetDataForkName

```
OSErr FSGetDataForkName      (HFSUniStr255 *         dataForkName);
```

Returns a Unicode string constant for the name of the data fork (currently an empty string) in the structure pointed to by `dataForkName`.

Note

There is no parameter block-based form of this call since it is not dispatched to individual volume formats, and does not require any I/O. ◆

5.6.15 FSGetResourceForkName

```
OSErr FSGetResourceForkName   (HFSUniStr255 *         resourceForkName);
```

Returns a Unicode string constant for the name of the resource fork (currently "RESOURCE_FORK") in the structure pointed to by `resourceForkName`.

> **Note**
>
> There is no parameter block-based form of this call since it is not dispatched to individual volume formats, and does not require any I/O. ◆

5.6.16 FSOpenFork

```
OSErr FSOpenFork       (const FSRef *           ref,
                        UniCharCount            forkNameLength,
                        const UniChar *         forkName,    /* can be NULL */
                        SInt8                   permissions,
                        SInt16 *                forkRefNum);

OSErr PBOpenForkSync          (FSForkIOParam *        paramBlock);
OSErr PBOpenForkAsync         (FSForkIOParam *        paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             The file or directory containing the fork to open
    ->  forkNameLength  The length of the fork name (in Unicode characters)
    ->  forkName        The name of the fork to open (in Unicode)
    ->  permissions     The access (read and/or write) you want
    <-  forkRefNum      The reference number for accessing the open fork
```

The `FSOpenFork` call is used to open any fork of a file or directory for streaming access (using the `FSReadFork`, `FSWriteFork`, `FSCloseFork` and related calls). The fork may be larger than 2GB, and is allowed to grow to 2GB or larger.

The file or directory is specified by the `ref` parameter. The fork to open is specified by `forkName` and `forkNameLength`. You can obtain the string constants for the data fork and resource fork using the `FSGetDataForkName` and `FSGetResourceForkName` calls.

The `permission` parameter controls the way the file can be accessed via the returned fork reference (read-only or read/write). It is the same as the `permission` parameter passed to `FSpOpenDF` and `FSpOpenRF`.

A fork reference number is returned in `forkRef`.

5.6.17 FSReadFork

```
OSErr FSReadFork        (SInt16                 forkRefNum,
                         UInt16                 positionMode,
                         SInt64                 positionOffset,
                         ByteCount              requestCount,
                         void *                 buffer,
                         ByteCount *            actualCount); /* can be NULL */

OSErr PBReadForkSync           (FSForkIOParam *        paramBlock);
OSErr PBReadForkAsync          (FSForkIOParam *        paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  forkRefNum      The reference number of the fork to read from
    ->  positionMode    The base location for start of read
    ->  positionOffset  The offset from base location for start of read
    ->  requestCount    The number of bytes to read
    <-  buffer          Pointer to buffer where data will be returned
    <-  actualCount     The number of bytes actually read
```

The `FSReadFork` call is used to read data from a fork opened using the `FSOpenFork` call. Data is read starting at the position specified by `positionMode` and `positionOffset`. Up to `requestCount` bytes will be read into the buffer pointed at by the `buffer` parameter. The actual number of bytes read is (optionally) returned in `actualCount`.

The `actualCount` will be equal to `requestCount` unless there was an error during the read operation. If there are fewer than `requestCount` bytes from the specified position to the logical end-of-file, then all of those bytes are read, and `eofErr` is returned. The `actualCount` output is optional; if you don't want it, set `actualCount` to `NULL`.

The fork's current position is set to point immediately after the last byte read (that is, the initial position plus `actualCount`).

The caller can hint to the File Manager whether the data being read should or should not be cached. Set the appropriate bits in `positionMode`. (This is the same behavior as `PBReadSync`.) See Section 5.3.4, "FSPositionMode" for more information.

You may also add `forceReadMask` to `positionMode`. This tells the File Manager to force the data to be read directly from the disk. This is different from `noCacheMask` since `forceReadMask` will flush the appropriate part of the cache first, then ignore any data already in the cache. However, data that is read may be placed in the cache for future reads. The `forceReadMask` is also passed to the device driver, indicating that it should avoid reading from any device caches.

To verify that data previously written has been correctly transferred to disk, read it back in using `forceReadMask` and compare it with the data you previously wrote.

5.6.18 FSWriteFork

```
OSErr FSWriteFork       (SInt16                 forkRefNum,
                         UInt16                 positionMode,
                         SInt64                 positionOffset,
                         ByteCount              requestCount,
                         void *                 buffer,
                         ByteCount *            actualCount); /* can be NULL */

OSErr PBWriteForkSync          (FSForkIOParam *        paramBlock);
OSErr PBWriteForkAsync         (FSForkIOParam *        paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  forkRefNum      The reference number of the fork to write to
    ->  positionMode    The base location for start of write
    ->  positionOffset  The offset from base location for start of write
    ->  requestCount    The number of bytes to write
    ->  buffer          Pointer to data to write
    <-  actualCount     The number of bytes actually written
```

The `FSWriteFork` call is used to write data to a fork opened using the `FSOpenFork` call. Data is written starting at the position specified by `positionMode` and `positionOffset`. The call will attempt to write `requestCount` bytes from the buffer pointed at by the `buffer` parameter. The actual number of bytes written is (optionally) returned in `actualCount`.

The `actualCount` will be equal to `requestCount` unless there was an error during the write operation. If there is not enough space on the volume to write `requestCount` bytes, then `dskFulErr` is returned and `actualCount` contains the number of bytes actually written to the fork. The `actualCount` output is optional; if you don't want it, set `actualCount` to `NULL`.

The caller can hint to the File Manager whether the data being written should or should not be cached. Set the appropriate bits in `positionMode`. See Section 5.3.4, "FSPositionMode" for more information.

The fork's current position is set to point immediately after the last byte written (that is, the initial position plus `actualCount`).

5.6.19 FSGetForkPosition

```
OSErr FSGetForkPosition       (SInt16                 forkRefNum,
                               UInt64 *               position);

OSErr PBGetForkPositionSync   (FSForkIOParam *        paramBlock);
OSErr PBGetForkPositionAsync  (FSForkIOParam *        paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  forkRefNum      The reference number of the fork
    <-  positionOffset  The current position of the fork (position)
```

The `FSGetForkPosition` call returns the current position of the open fork specified by `forkRef` in the `forkPosition` parameter. The returned fork position is relative to the start of the fork (that is, is an absolute offset in bytes).

5.6.20 FSSetForkPosition

```
OSErr FSSetForkPosition     (SInt16       forkRefNum,
                             UInt16       positionMode,
                             SInt64       positionOffset); /* can be NULL */

OSErr PBSetForkPositionSync   (FSForkIOParam *        paramBlock);
OSErr PBSetForkPositionAsync  (FSForkIOParam *        paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  forkRefNum      The reference number of the fork
    ->  positionMode    The base location for the new position
    ->  positionOffset  The offset of the new position from the base
```

The `FSSetForkPosition` call sets the current position of the open fork specified by `forkRef` to the position indicated by `positionMode` and `positionOffset`. If `positionMode` is equal to `fsAtMark`, then the `positionOffset` parameter is ignored. See Section 5.3.4, "FSPositionMode" for a description of the constants that may be used in `positionMode`.

5.6.21 FSGetForkSize

```
OSErr FSGetForkSize           (SInt16                 forkRefNum,
                               UInt64 *               forkSize);

OSErr PBGetForkSizeSync       (FSForkIOParam *        paramBlock);
OSErr PBGetForkSizeAsync      (FSForkIOParam *        paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  forkRefNum      The reference number of the fork
    <-  positionOffset  The logical size of the fork, in bytes (forkSize)
```

The `FSGetForkSize` call returns the size (logical end-of-file) of the open fork specified by `forkRef` in the `forkSize` parameter. The size returned is the logical size (that is, the total number of bytes that can be read from the fork); the amount of space actually allocated on the volume (the physcial size) will probably be larger.

5.6.22 FSSetForkSize

```
OSErr FSSetForkSize         (SInt16        forkRefNum,
                             UInt16        positionMode,
                             SInt64        positionOffset);

OSErr PBSetForkSizeSync       (FSForkIOParam *        paramBlock);
OSErr PBSetForkSizeAsync      (FSForkIOParam *        paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  forkRefNum      The reference number of the fork
    ->  positionMode    The base location for the new size
    ->  positionOffset  The offset of the new size from the base
```

The `FSSetForkSize` call changes the size of the open fork specified by `forkRef`. The logical end-of-file will be set to the position indicated by the `positionMode` and `positionOffset` parameters.

The fork’s new size may be less than, equal to, or greater than the fork’s current size. If the fork’s new size is greater than the fork’s current size, then the additional bytes (between the old and new size) will have an undetermined value.

If there is not enough space on the volume to extend the fork, then `dskFulErr` is returned and the fork’s size is unchanged.

If the fork’s current position is larger than the fork’s new size, then the current position will be set to the new fork size. That is, the current position will be equal to the logical end of file.

5.6.23 FSAllocateFork

```
OSErr FSAllocateFork   (SInt16                forkRefNum,
                        FSAllocationFlags      flags,
                        UInt16                 positionMode,
                        SInt64                 positionOffset,
                        UInt64                 requestCount,
                        UInt64 *               actualCount); /* can be NULL */

OSErr PBAllocateForkSync       (FSForkIOParam *        paramBlock);
OSErr PBAllocateForkAsync      (FSForkIOParam *        paramBlock);
    ->  ioCompletion      A pointer to a completion routine
    <-  ioResult          The result code of the function
    ->  forkRefNum        The reference number of the fork
    ->  allocationFlags   Controls how new space is allocated
    ->  positionMode      The base location for start of allocation
    ->  positionOffset    The offset of the start of allocation
    <>  allocationAmount  The number of bytes to allocate (requestCount)
                          On output, the number of bytes actually added
                          (actualCount)
```

The `FSAllocateFork` call is a hint to reserve space on a volume for use by the open fork specified by `forkRef`. This call will attempt to allocate `requestCount` bytes of physical storage starting at the offset specified by `positionMode` and `positionOffset`. For volume formats that support preallocated space, you can later write to this range of bytes (including extending the size of the fork) without requiring an implicit allocation.

The `flags` parameter controls how the space is allocated. If the `kAllocContiguousMask` is set, then then any newly allocated space must be in one contiguous extent (preferably contiguous with any space already allocated). If `kAllocAllOrNothingMask` is set, then the entire requestCount bytes must be allocated for the call to succeed; if not set, as many bytes as possible will be allocated (without error). If `kFSAllocNoRoundUpMask` is set, then no additional space is allocated (such as rounding up to a multiple of a clump size); if clear, the volume format may allocate more space than requested as an attempt to reduce fragmentation. See Section 5.3.3, "FSAllocationFlags".

The actual number of bytes allocated to the file (starting at the given starting position) is returned in `actualCount`. The `actualCount` may be smaller than `requestCount` if some of the space was already allocated. The `actualCount` output is optional; if you don’t want it, set `actualCount` to `NULL`.

Note

The value returned in `actualCount` does not reflect any additional bytes that may have been allocated because space is allocated in terms of fixed units such as allocation blocks, or the use of a clump size to reduce fragmentation. ◆

Any extra space allocated but not used will be deallocated when the fork is closed (via FSCloseFork) or flushed (via FSFlushFork).

5.6.24 FSFlushFork

```
OSErr FSFlushFork             (SInt16                 forkRefNum);

OSErr PBFlushForkSync         (FSForkIOParam *        paramBlock);
OSErr PBFlushForkAsync        (FSForkIOParam *        paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  forkRefNum      The reference number of the fork to close
```

The `FSFlushFork` call causes all data written to the open fork specified by `forkRef` to be written to disk. The actual fork contents are written, as well as any other volume structures needed to access the fork.[3]

Note

On volumes that do not support `FSFlushFork` directly, the entire volume is flushed to be sure all volume structures associated with the fork are written to disk. ◆

5.6.25 FSCloseFork

```
OSErr FSCloseFork             (SInt16                 forkRefNum);

OSErr PBCloseForkSync         (FSForkIOParam *        paramBlock);
OSErr PBCloseForkAsync        (FSForkIOParam *        paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  forkRefNum      The reference number of the fork to close
```

The `FSCloseFork` call causes all data written to the open fork specified by `forkRef` to be written to disk (the same as `FSFlushFork`), and then closes the open fork (making `forkRef` invalid).

3. On HFS and HFS Plus, this includes the catalog, extents, and attribute B-trees; the volume bitmap; and volume header and alternate volume header (MDB, alterhate MDB), as needed.

5.6.26 FSGetForkCBInfo

```
struct FSForkInfo {
    SInt8                    flags;           /* copy of FCB flags */
    SInt8                    permissions;
    UInt16                   reserved1;
    UInt32                   reserved2;
    UInt32                   nodeID;          /* file or directory ID */
    UInt32                   forkID;          /* fork ID */
    UInt64                   currentPosition;
    UInt64                   logicalEOF;
    UInt64                   physicalEOF;
    UInt64                   process;     /* should be ProcessSerialNumber */
};
typedef struct FSForkInfo    FSForkInfo;
typedef FSForkInfo *         FSForkInfoPtr;

OSErr FSGetForkCBInfo  (SInt16                desiredRefNum,
                        FSVolumeRefNum        volume,
                        SInt16 *              iterator,
                        SInt16 *              actualRefNum,
                        FSForkInfo *          forkInfo,      /* can be NULL */
                        FSRef *               ref,           /* can be NULL */
                        HFSUniStr255 *        outForkName);  /* can be NULL */

OSErr PBGetForkCBInfoSync   (FSForkCBInfoParam *  paramBlock);
OSErr PBGetForkCBInfoAsync  (FSForkCBInfoParam *  paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  desiredRefNum   If non-zero on input, then get information for
                        this refnum; unchanged on output.  If zero on input,
                        iterate over all open forks (possibly limited to a
                        single volume); on output, contains the fork's refnum.
    ->  volumeRefNum    Used when desiredRefNum is zero on input.  Set to 0 to
                        iterate over all volumes, or set to a FSVolumeRefNum
                        to limit iteration to that volume.
    <>  iterator        Used when desiredRefNum is zero on input.  Set to 0
                        before iterating.  Pass the iterator returned by the
                        previous call to continue iterating.
    <-  actualRefNum    The refnum of the open fork.
    <-  ref             The FSRef for the file or directory that contains
                        the fork.
    <-  forkInfo        Various information about the open fork.
    <-  outForkName     The name of the fork.
```

The `FSGetForkCBInfo` call returns information about a specified open fork, or for all open forks (optionally limited to a given volume). If you want information on a specific refnum, set `desiredRefNum` to that value, and pass `NULL` for `iterator`.

If you want to iterate over all open forks (or all open forks for a specific volume), set `volume` to the desired volume refnum, or `kFSInvalidVolumeRefNum` for all volumes. Set `iterator` to `0` before the first call. Set `desiredRefNum` to zero. Upon completion of the call, iterator will be updated and that updated value should be passed into the next call. The call returns errFSNoMoreItems if there are no more open forks to return.

The actualRefNum is not NULL, the refnum of the fork will be returned. If forkInfo is not NULL, the FSForkInfo for the open fork will be returned. If ref is not NULL, an FSRef for the file or directory containing the fork will be returned. If outForkName is not NULL, then the fork name will be returned.

The well-known data fork (i.e. the fork whose name is the empty string, returned by FSGetDataForkName) will always have a forkID of zero. You should not make any assumption about the forkID of the well-known resource fork (whose name is returned by FSGetResourceForkName).

5.6.27 FSGetCatalogInfo

```
OSErr FSGetCatalogInfo (const FSRef *          ref,
                        FSCatalogInfoBitmap    whichInfo,
                        FSCatalogInfo *        catalogInfo, /* can be NULL */
                        HFSUniStr255 *         outName,     /* can be NULL */
                        FSSpec *               fsSpec,      /* can be NULL */
                        FSRef *                parentRef);  /* can be NULL */

OSErr PBGetCatalogInfoSync    (FSRefParam *           paramBlock);
OSErr PBGetCatalogInfoAsync   (FSRefParam *           paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             The file or directory whose information is to
                        be returned
    ->  whichInfo       Which catalog info fields to get; may be NULL
    <-  catInfo         The returned values of catalog info fields;
                        may be NULL
    <-  spec            A pointer to the FSSpec for the object; may be NULL
    <-  parentRef       A pointer to the FSRef for the object's parent
                        directory; may be NULL
    <-  outName         The Unicode name is returned here; may be NULL.
```

The FSGetCatalogInfo call returns general information about the file or directory specified by ref.

General information about the file or directory is optionally returned in catalogInfo. Only the information specified by whichInfo is returned. If you don't want any catalog info, set whichInfo to kFSCatInfoNone and set catalogInfo to NULL.

If spec is not NULL, an FSSpec for the file or directory is returned. If parentRef is not NULL, an FSRef for the object's parent directory is returned. If outName is not NULL, the file or directory name is returned.

If the object specified by `ref` is a volume's root directory, then the ref returned in `parentRef` will not be a valid `FSRef` (since the root directory has no parent object).

5.6.28 FSSetCatalogInfo

```
OSErr FSSetCatalogInfo        (const FSRef *           ref,
                               FSCatalogInfoBitmap     whichInfo,
                               const FSCatalogInfo *   catalogInfo);

OSErr PBSetCatalogInfoSync    (FSRefParam *            paramBlock);
OSErr PBSetCatalogInfoAsync   (FSRefParam *            paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ref             The file or directory whose information is to
                        be changed
    ->  whichInfo       Which catalog info fields to set
    ->  catInfo         The new values of catalog info fields
```

The `FSSetCatalogInfo` call sets the general information about the file or directory specified by `ref`.

The general information to be set is passed in `catInfo`. The `whichInfo` parameter determines which fields of the `catInfo` should be set. The following fields may be set:

- `createDate`
- `contentModDate`
- `attributeModDate`
- `accessDate`
- `backupDate`
- `permissions`
- `finderInfo`
- `extFinderInfo`
- `textEncodingHint`

5.6.29 FSOpenIterator

```
typedef struct OpaqueFSIterator*  FSIterator;
enum {
  kFSIterateFlat      = 0,         /* Immediate children of container only */
  kFSIterateSubtree   = 1,         /* Entire subtree rooted at container */
  kFSIterateReserved  = (long)0xFFFFFFFE
};
typedef OptionBits    FSIteratorFlags;

OSErr FSOpenIterator      (const FSRef *          container,
                           FSIteratorFlags        iteratorFlags,
                           FSIterator *           iterator);

OSErr PBOpenIteratorSync  (FSCatalogBulkParam *   paramBlock);
OSErr PBOpenIteratorAsync (FSCatalogBulkParam *   paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    <-  iterator        The returned FSIterator
    ->  iteratorFlags   Controls whether the iterator iterates over subtrees
                        or just the immediate children of the container.
    ->  container       An FSRef for the directory to iterate (or root of
                        the subtree to iterate).
```

The `FSOpenIterator` call creates a catalog iterator that can be used to iterate over the contents of a directory or volume. The set of items to iterate over can either be the objects directly contained in a directory, or all items directly or indirectly contained in a directory (i.e. recursive or subtree).

The `container` parameter specifies the directory to iterate over (or root of the subtree).

The `iteratorFlags` parameter is a set of flags. At this time, only one bit (bit 0) is defined. If the bit is clear (i.e. `kFSIterateFlat` is passed), then the set of items to iterate over is the set of files and folders directly contained by the `container` directory; that is, the files and folders whose parent directory is `container`. If the bit is set (i.e. `kFSIterateSubtree` is passed), then the set of items are all files and folders directly or indirectly contained by `container`; that is, files or folders in the subtree rooted at `container`.

The newly created catalog iterator is stored in the location pointed to by the `iterator` parameter. The iterator is automatically initialized so that the next use of the iterator returns the first item. The order that items are returned is volume format dependent and may be different for two different iterators created with the same container and flags.

Catalog iterators must be closed when you are done using them, whether or not you have iterated over all the items. Iterators are automatically closed upon process termination (just like open files). However, you should use the `FSCloseIterator` call to close an iterator to free up any system resources allocated to the iterator.

5.6.30 FSCloseIterator

```
OSErr FSCloseIterator       (FSIterator             iterator);

OSErr PBCloseIteratorSync   (FSCatalogBulkParam *   paramBlock);
OSErr PBCloseIteratorAsync  (FSCatalogBulkParam *   paramBlock);
```

The `FSCloseIterator` call closes a catalog iterator. It releases memory and other system resources used by the iterator. The iterator becomes invalid.

5.6.31 FSGetCatalogInfoBulk

```
OSErr FSGetCatalogInfoBulk
                       (FSIterator             iterator,
                        ItemCount              maximumObjects,
                        ItemCount *            actualObjects,
                        Boolean *              containerChanged,/*can be NULL*/
                        FSCatalogInfoBitmap    whichInfo,
                        FSCatalogInfo *        catalogInfos,   /*can be NULL*/
                        FSRef *                refs,           /*can be NULL*/
                        HFSUniStr255 *         names);         /*can be NULL*/

OSErr PBGetCatalogInfoBulkSync  (FSCatalogBulkParam * paramBlock);
OSErr PBGetCatalogInfoBulkAsync (FSCatalogBulkParam * paramBlock);
    ->  ioCompletion     A pointer to a completion routine
    <-  ioResult         The result code of the function
    ->  iterator         The iterator
    ->  maximumItems     The maximum number of items to return
    <-  actualItems      The actual number of items returned
    <-  containerChanged Set to true if the container's contents changed
    ->  whichInfo        The catalog information fields to return for
                         each item
    <-  catalogInfo      An array of catalog information; one for each
                         returned item
    <-  refs             An array of FSRefs; one for each returned item
    <-  names            An array of filenames; one for each returned item
```

The `FSGetCatalogInfoBulk` call returns information about one or more objects from a catalog iterator; it can return information about multiple objects in a single call.

The `iterator` parameter specifies the iterator to use.

The maximum number of objects to return is passed in `maximumObjects`. The actual number of objects found for this call is returned in `actualObjects`.

The `containerChanged` output is set to true if the container changed since the previous `FSGetCatalogInfoBulk` call. Objects may still be returned even though the container changed. This output is optional; if you don't want it, pass a `NULL` pointer. Note that if the container has changed, then the total set of items returned may be incorrect; some items may be returned multiple times, and some items may not be returned at all.

The rest of the parameters allow you to obtain the `FSRef`, Unicode name, and catalog information for every item found. To get catalog information, set the desired bits in `whichInfo`; an array of `FSCatalogInfo` structures with the corresponding fields are returned, one for each item found. The `catalogInfos` parameter should point to an array of `maximumObjects` `FSCatalogInfo`s. (See also Section 5.3.6, "FSCatalogInfo".) If you don't want any catalog information, set `whichInfo` to `kFSCatInfoNone` and `catalogInfos` to `NULL`.

If you want an `FSRef` for each item found, set `fsRefs` to point to an array of `maximumObjects` `FSRef`s. Otherwise, set `fsRefs` to `NULL`.

If you want the Unicode filename for each item found, set `names` to point to an array of `maximumObjects` `HFSUniStr255`s. Otherwise, set `names` to `NULL`.

The `FSGetCatalogInfoBulk` call may complete and return `noErr` with fewer than `maximumObjects` items returned. This may be due to various reasons related to the internal implementation. In this case, you may continue to make `FSGetCatalogInfoBulk` calls using the same `iterator`.

When all of the iterator's objects have been returned, the call will return `errFSNoMoreItems`.

5.6.32 FSCatalogSearch

```
enum {
  /* CatalogSearch constants */
  fsSBNodeID            = 0x00008000, /* search by range of nodeID */
  fsSBAttributeModDate = 0x00010000, /* search by range of attributeModDate */
  fsSBAccessDate        = 0x00020000, /* search by range of accessDate */
  fsSBPermissions       = 0x00040000, /* search by value/mask of permissions */

  fsSBNodeIDBit              = 15,
  fsSBAttributeModDateBit   = 16,
  fsSBAccessDateBit          = 17,
  fsSBPermissionsBit         = 18
};


struct FSSearchParams {
  SInt32           searchTime;             /* a Time Manager duration */
  OptionBits       searchBits;             /* which fields to search on */
  UniCharCount     searchNameLength;
  UniChar *        searchName;
  FSCatalogInfo *  searchInfo1;            /* values and lower bounds */
  FSCatalogInfo *  searchInfo2;            /* masks and upper bounds */
};
typedef struct FSSearchParams  FSSearchParams;
typedef FSSearchParams *       FSSearchParamsPtr;

OSErr FSCatalogSearch (FSIterator             iterator,
                       const FSSearchParams * searchCriteria,
                       ItemCount              maximumObjects,
                       ItemCount *            actualObjects,
                       Boolean *              containerChanged,/*can be NULL*/
                       FSCatalogInfoBitmap    whichInfo,
                       FSCatalogInfo *        catalogInfos,  /* can be NULL */
                       FSRef *                refs,          /* can be NULL */
                       HFSUniStr255 *         names);        /* can be NULL */

OSErr PBCatalogSearchSync   (FSCatalogBulkParam *   paramBlock);
OSErr PBCatalogSearchAsync  (FSCatalogBulkParam *   paramBlock);
    ->  ioCompletion     A pointer to a completion routine
    <-  ioResult         The result code of the function
    ->  iterator         The iterator
    ->  searchParams     The criteria that controls the matching, including
                         timeout, a bitmap controlling the fields to compare,
                         and the (Unicode) name to compare.
    ->  maximumItems     The maximum number of items to return
    <-  actualItems      The actual number of items returned
    <-  containerChanged Set to true if the container’s contents changed
    ->  whichInfo        The catalog information fields to return for each
                         item
    <-  catalogInfo      An array of catalog information; one for each
                         returned item
    <-  refs             An array of FSRefs; one for each returned item
    <-  names             An array of filenames; one for each returned item
```

The FSCatalogSearch call searches for objects from a catalog iterator that match a given set of criteria. A single search may span more than one call to FSCatalogSearch. Various information about each matching file or directory is returned.

The iterator parameter specifies the iterator to use. Objects traversed by that iterator are matched against the criteria specified by searchCriteria. You can match against the object's name (in Unicode) and by the fields in a FSCatalogInfo. You may use the same search bits as passed in ioSearchBits to PBCatSearch; they control the corresponding FSCatalogInfo fields. The fsSBPartialName and fsSBFullName constants control matching against the Unicode name specified by searchNameLength and searchName.

There are a few new search criteria supported by FSCatalogSearch but not PBCatSearch. The fsSBNodeID, fsSBAttributeModDate, and fsSBAccessDate bits let you search by a range of nodeID, attributeModDate, and accessDate, respectively. The fsSBPermissions bit lets you search by mask/value match against the permissions.

Set maximumObjects to the maximum number of objects to return. The actual number of objects found in this call is returned in actualObjects.

If searchTime is non-zero, it is interpretted as a Time Manager duration (milliseconds if positive, or microseconds if negative); the search may terminate after this duration even if maximumObjects objects have not been returned and the entire catalog has not been scanned. If searchTime is zero, there is no time limit for the search.

If you are searching by partial or full name (fsSBPartialName or fsSBFullName), then searchName points to searchNameLength Unicode characters to match against. If you are searching by any other criteria, you must set searchInfo1 and searchInfo2 to point to FSCatalogInfo structures containing the values to match against. For fields that match against a range (such as dates), the minimum matching value should be placed in the searchInfo1 structure and the maximum matching value in the searchInfo2 structure. For fields that match by mask/value (such as permissions and Finder info), set the bits you wish to compare in the searchInfo2 structure; set the corresponding matching value in the searchInfo1 structure.

The call may complete with no error before scanning the entire volume. This typically happens because the time limit (searchTime) has been reached or maximumObjects items have been returned. When the entire volume has been searched, errFSNoMoreItems is returned.

The containerChanged output is set to true if the container changed since the previous FSCatalogSearch call. Objects may still be returned even though the container changed. This output is optional; if you don't want it, pass a NULL pointer. Note that if the container has changed, then the total set of items returned may be incorrect; some items may be returned multiple times, and some items may not be returned at all.

You can get various information about all the matching files or directories. The `whichInfo`, `catalogInfos`, `fsRefs`, `nameLengths`, and `names` parameters are used in the same way as for the `FSGetCatalogInfoBulk` call.

5.6.33 FSGetVolumeInfo

```
OSErr FSGetVolumeInfo  (FSVolumeRefNum       volume,
                        ItemCount            volumeIndex,
                        FSVolumeRefNum *     actualVolume,   /* can be NULL */
                        FSVolumeInfoBitmap   whichInfo,
                        FSVolumeInfo *       info,           /* can be NULL */
                        HFSUniStr255 *       volumeName,     /* can be NULL */
                        FSRef *              rootDirectory); /* can be NULL */

OSErr PBGetVolumeInfoSync   (FSVolumeInfoParam *  paramBlock);
OSErr PBGetVolumeInfoAsync  (FSVolumeInfoParam *  paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    <>  ioVRefNum       The volume whose information is to be returned
                        (if volumeIndex is 0).  On output, the volume
                        reference number of the volume (actualVolume).
    ->  volumeIndex     The index of the desired volume, or 0 to use ioVRefNum
    ->  whichInfo       Which volInfo info fields to get; may be NULL
    <-  volumeInfo      The returned values of Volume info fields; may be NULL
    <-  volumeName      The Unicode name is returned here; may be NULL.
    <-  ref             The FSRef for the root directory; may be NULL
```

The `FSGetVolumeInfo` call returns information about a volume as a whole (as opposed to the root directory of a volume, for which you can use `FSGetCatalogInfo`).

You can specify a particular volume or index through the list of mounted volumes. To get information on a particular volume, set `volume` to the desired `FSVolumeRefNum` and set `volumeIndex` to zero. To index through the list of mounted volumes, set volume to `kFSInvalidVolumeRefNum` and set `volumeIndex` to the index (starting at 1).

The volume refnum of the volume is returned in `actualVolume`; this is useful when indexing over all mounted volumes. If you don't want this information (because you supplied a particular volume refnum in `volume`, for example), set `actualVolume` to `NULL`.

Information about the volume is returned in `info`. If you don't want this output, set `info` to `NULL` and `whichInfo` to `kFSVolInfoNone`. The `whichInfo` parameter controls which fields of `info` are returned. For more information, see Section 5.3.8, "FSVolumeInfo".

If `volumeName` is not `NULL`, the volume's name is returned.

If rootDirectory is not NULL, an FSRef for the volume's root directory is returned.

5.6.34 FSSetVolumeInfo

```
OSErr FSSetVolumeInfo         (FSVolumeRefNum         volume,
                               FSVolumeInfoBitmap     whichInfo,
                               const FSVolumeInfo *   info);

OSErr PBSetVolumeInfoSync     (FSVolumeInfoParam *    paramBlock);
OSErr PBSetVolumeInfoAsync    (FSVolumeInfoParam *    paramBlock);
    ->  ioCompletion    A pointer to a completion routine
    <-  ioResult        The result code of the function
    ->  ioVRefNum       The volume whose information is to be changed
    ->  whichInfo       Which catalog info fields to set
    ->  volumeInfo      The new values of volume info fields
```

The FSSetVolumeInfo call is used to set information about a volume as a whole (as opposed to the root directory of a volume, for which you can use FSSetCatalogInfo).

Set volume to the appropriate FSVolumeRefNum.

The volume information to be set is passed in info. The particular fields to set are specified by whichInfo. The following fields may be set:

- backupDate
- finderInfo
- flags

See Section 5.3.8, "FSVolumeInfo" for more information about these fields.

5.7 Implementing Foreign Filesystems

To allow developers to support the HFS Plus APIs in their filesystems, some internal data structures and utility routines are enhanced.

The most significant change is for File Control Blocks. File refnums used to be byte offsets into an array of FCBs. Since refnums are positive 16-bit values, this limits both the size and number of FCBs. As of the Mac OS 9 release, the way FCBs are stored will change. This makes it very important that external filesystems use the FSM accessor functions to access an FCB. These routines must be used even if the external filesystem doesn't use FSM for dispatching requests.

To accommodate 64-bit file sizes and named forks, several fields will be added to the FCB. The new FCB structure is called ForkControlBlock.

To support large forks, the logical-to-physical block mapping used by the disk cache has been extended. Forks that are opened with the new APIs should have `fcbLargeFileBit` set in the `fcbFlags`. When this bit is set, the cache routines pass a file offset as a number of 512-byte sectors instead of a byte offset. This allows forks and volumes up to 2 terabytes.

To support multiple named forks, the FCB contains a `forkID`. When the `fcbLargeFileBit` is set, the `forkID` is used to determine the fork number. Otherwise, the `fcbResourceBit` is used: if clear, the fork number is zero; if set, the fork number is `$FFFFFFFF`. If an external filesystem supports `forkID`s, it must use zero for the `forkID` of the data fork.

Since external filesystems can no longer map between file refnums and FCBs themselves, two new support routines have been added to FSM:

```
OSErr UTResolveFileRefNum  (FCBRecPtr fileCtrlBlockPtr, short *fileRefNum);
OSErr UTCheckFCB           (FCBRecPtr fileCtrlBlockPtr);
```

The `UTResolveFileRefNum` routine returns the file refnum for a given FCB pointer; it is the inverse of `UTResolveFCB`. The `UTCheckFCB` routine lets a caller determine whether an externally-supplied FCB pointer actually corresponds to a real FCB (whether the FCB is currently allocated or not); it is like `UTCheckFileRefNum` except that it uses a FCB pointer instead of a refnum. Both calls will return `badFCBErr` if the FCB pointer does not correspond to an actual FCB.

5.7.1 ForkControlBlock

```
struct ForkControlBlock {
    unsigned long        fcbFlNm;      /* FCB file number.  Zero if unused */
    SignedByte           fcbFlags;     /* FCB flags */
    SignedByte           fcbTypByt;    /* File type byte */
    unsigned short       fcbSBlk;  /* File start block (in alloc size blks) */
    unsigned long        fcbEOF;       /* Logical length or EOF in bytes */
    unsigned long        fcbPLen;      /* Physical file length in bytes */
    unsigned long        fcbCrPs;      /* Current position within file */
    VCBPtr               fcbVPtr;      /* Pointer to the corresponding VCB */
    Ptr                  fcbBfAdr;     /* File's buffer address */
    unsigned short       fcbFlPos;     /* Directory block this file is in */

    /* FCB Extensions for HFS */
    unsigned long        fcbClmpSize;  /* Number of bytes per clump */
    Ptr                  fcbBTCBPtr;   /* B*-Tree control block for file */
    HFSExtentRecord      fcbExtRec;    /* First 3 file extents */
    OSType               fcbFType;     /* File's 4 Finder Type bytes */
    unsigned long        fcbCatPos;    /* Catalog hint for use on Close */
    unsigned long        fcbDirID;     /* Parent Directory ID */
    Str31                fcbCName;     /* CName of open file */

    /* New fields for HFS Plus APIs */
    unsigned short       moreFlags;    /* more flags, align following fields*/

    /*  Old ExtendedFCB fields*/
    UInt32               processID1;  /* these two fields are the Process */
                                      /* serial number that opened the file */
    UInt32               processID2;  /* (used to clean up at process death)*/

    HFSPlusExtentRecord  extents;     /*  extents for HFS+ volumes */

    /*  New fields for HFS Plus APIs*/
    UInt64               endOfFile;        /* logical size of this fork */
    UInt64               physicalEOF;      /* amount of space physically */
                                           /* allocated to this fork */
    UInt64               currentPosition;  /* default offset for next */
                                           /* read/write */
    UInt32               forkID;
    UInt32               reserved1;
    UInt64               reserved2;     /* make size a multiple of 16 bytes */
};
typedef struct ForkControlBlock  ForkControlBlock;
typedef ForkControlBlock *       ForkControlBlockPtr;
```

The File Control Block (FCB) structure has been extended to support forks larger than 2GB, and multiple named forks. The fields up through `fcbCName` are the same as in previous versions of Mac OS; new fields were added at the end. A new data type, `ForkControlBlock`, containing those new fields has been added; the `FCBRec` structure retains its original definition, for backward compatibility (and Mac OS ROM builds).

In previous versions of Mac OS, all FCBs were stored as an array in a single block of memory. This block of memory was pointed to by the low memory global `FCBsPtr` (`$34E`). The size of an FCB was stored in the low memory global `FSFCBLen` (`$3F6`). A file refnum was merely a byte offset from `FCBsPtr`, so you could compute the address of an FCB by adding its refnum to `FCBsPtr`. This severly limited the maximum number of FCBs.

Starting with the Mac OS 9 release, FCBs are stored differently. An external filesystem *must* use the FSM accessor functions (`UTAllocateFCB`, `UTReleaseFCB`, `UTLocateFCB`, `UTLocateNextFCB`, `UTIndexFCB`, `UTResolveFCB`) to access FCBs. You can use these functions even if you don't dispatch calls through FSM. See "Guide to the File System Manager" (part of the File System Manager SDK) for more descriptions of these calls.

Since several fields in the old FCB structure are too small to support forks larger than 2GB, larger replacement fields have been added. A new flag bit (`fcbLargeFileBit`) in the `fcbFlags` field has been defined to indicate whether the old, shorter fields or new, larger fields are being used. If the bit is set, then the `endOfFile` field is used instead of `fcbEOF`, `physicalEOF` is used instead of `fcbPLen`, and `currentPosition` is used instead of `fcbCrPs`. The `fcbLargeFileBit` should be set when a fork is opened via `FSOpenFork` and cleared when opened via the older APIs (`PBHOpenDF`, `PBHOpenRF`, etc.). The bit also implies that a fork may grow beyond 2GB (if the volume format supports it).

The `moreFlags` field is reserved for additional flag bits to be defined by Apple. The processID1 and processID2 fields are used by the File Manager to store the ID of the process that opened the fork. The `extents` field is used to store the first eight extents for forks on an HFS Plus volume; your filesystem may use this field for its own purposes.

The `forkID` field is used to differentiate between multiple forks for a file or directory. In effect, it is a replacement for the `fcbResourceBit` in `fcbFlags`. If you use the Disk Cache, you must set this field as described in the following sections; otherwise, your filesystem may use this field for its own purposes.

6. Glossary

6.1 compatibility layer

The compatibility layer is a part of the File Manager that allows the HFS Plus APIs to be used to access files on a volume format whose implementation does not directly implement the HFS Plus APIs. This allows applications to adopt the HFS Plus APIs without having to determine whether a particular volume format supports them.

When an application uses one of the HFS Plus APIs to access a volume that does not directly implement the APIs, the File Manager provides a default implementation.

This default implementation actually uses the equivalent older APIs to provide the functionality. For example, a call to `FSDeleteObject` would result in a `PBHDelete` call being passed to the volume format's implementation.

Since the compatibility layer uses the older APIs to provide the implementation for the HFS Plus APIs, it can't actually provide any functionality beyond what the older APIs allow. For example, the compatibility layer can't actually allow files to grow larger than 2GB. Similarly, it can't cause filenames to be stored differently (though it internally converts between Unicode and the File Manager's default Mac OS text encoding).

6.2 high-level API

A high-level API takes all of its parameters as separate function arguments. It is synchronous only, which means the caller is blocked until the call completes.

For many of the older File Manager APIs, the high-level APIs were simplified versions; they did not always provide the same variety of functionality that the low-level APIs did. With the HFS Plus APIs, the high-level APIs provide the same level of functionality as the low-level APIs.

6.3 low-level API

A low-level API is characterized by its use of a parameter block. The parameters to the API are stored as fields in a structure called a parameter block. The caller then passes a pointer to the parameter block as the only argument to the function. A low-level API can be made synchronously or asynchronously. If the call is made synchronously, the caller blocks until the call completes. If the call is made asynchronously, the caller may regain control before the call completes; when the call does complete, a caller-supplied routine (called a completion routine) is called asynchronously with respect to the caller's thread of execution.

6.4 parameter block

A parameter block is a data structure that is used to collect the input and output parameters for a low-level API. It may also contain state information used during the execution of the call.

6.5 synchronous

As in “synchronous call”. Sometimes shortened to “sync”. A synchronous call completes its work while the calling routine is blocked or suspended. Once the work is complete, the calling routine resumes. An ordinary function call is synchronous.

6.6 asynchronous

As in “asynchronous call”. Sometimes shortened to “async”. An asynchronous call completes its work while the calling routine continues execution. When the work is complete, the caller is interrupted and a routine in the caller’s program (called a completion routine) is executed; when the completion routine returns, the caller resumes from where it was interrupted.

Note that an asynchronous call involves an ordinary function call to begin the work. That function call may return control to the caller before the work is complete.

6.7 catalog information

Also called “metadata”. Most volume formats store some kind of information about a file or directory, besides just their contents. For example, HFS and HFS Plus store a 4-byte file type and 4-byte creator type for all files; this information is used by the Mac OS Finder. Collectively, this kind of information is called “catalog information” because it is stored in a structure known as the “catalog”.

6.8 forks

A fork is a set of bytes that are contained by a file (or possibly a directory). A fork has a size or length (the number of bytes). It is characterized by the ability to randomly access those bytes, or a contiguous subset of the bytes. You can also increase or decrease the number of bytes in a fork.

Many older volume formats assume that a file contains only a single fork. This is what we would call the data fork. In this case, the fork is usually considered synonymous with the file itself. That is, the file’s contents are the set of bytes in the data fork.

The HFS volume format introduced a second fork for every file: the resource fork. The set of bytes in the data fork are logically separate from the bytes in the resource fork. Accessing, changing, increasing or decreasing the bytes of one fork does not affect the other fork.

6.9 named forks

Previously known as "attributes". Named forks are a generalization of the two forks supported by the older File Manager APIs and the HFS volume format. With the older APIs, the caller identified which fork they wanted to access by using a different call for each fork (eg., PBHOpenDF and PBHOpenRF to open the data or resource fork, respectively).

To support more than two well-known forks per file (or directory), there needs to be a way to indicate which fork you want to access. The HFS Plus APIs use a name (a Unicode string) to identify a fork. This is similar to the way that files in a directory have names that identify them. And similar to file names, a fork's name must be unique with respect to the set of all forks of a file.

Since the data and resource forks have not previously had identifiable names, and we would like to use the same APIs to access data and resource forks as well as other named forks, we must pick some constants to use as the names for the data and resource forks. An empty string (i.e. a string of zero length) is used for the data fork since it is often considered the primary contents of a file. For the resource fork, we have defined a constant in the APIs, `kFSResourceForkName`.