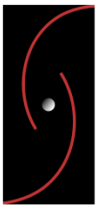

O S R
open
systems
resources
inc.

FSDK

Programming Interface V2.5

© 2009 OSR Open Systems Resources, Inc.

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means -- graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems -- without written permission of:

OSR Open Systems Resources, Inc.
105 Route 101A Suite 19
Amherst, New Hampshire 03031
+1 (603) 595-6500

OSR, the OSR logo, “OSR Open Systems Resources, Inc.”, and “The NT Insider” are trademarks of OSR Open Systems Resources, Inc. All other trademarks mentioned herein are the property of their owners.

Printed in the United States of America

Document Identifier: UL219

LIMITED WARRANTY

OSR Open Systems Resources, Inc. (OSR) expressly disclaims any warranty for the information presented herein. This material is presented “as is” without warranty of any kind, either express or implied, including, without limitation, the implied warranties of merchantability or fitness for a particular purpose. The entire risk arising from the use of this material remains with you. OSR’s entire liability and your exclusive remedy shall not exceed the price paid for this material. In no event shall OSR or its suppliers be liable for any damages whatsoever (including, without limitation, damages for loss of business profit, business interruption, loss of business information, or any other pecuniary loss) arising out of the use or inability to use this information, even if OSR has been advised of the possibility of such damages. Because some states/jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

U.S. GOVERNMENT RESTRICTED RIGHTS

This material is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Right in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software--Restricted Rights 48 CFR 52.227-19, as applicable. Manufacturer is OSR Open Systems Resources, Inc. Amherst, New Hampshire 03031.

© | OSR Open Systems Resources, Inc.

© | OSR Open Systems Resources, Inc.

© | OSR Open Systems Resources, Inc.

INTRODUCTION

Executive Overview

This is the Interface Definition provided as part of the OSR Open Systems Resources, Inc. File Systems Development Kit (FSDK). As such, this guide serves as the principal definition of the interface to use when implementing a File System Driver with the File Systems Wrapper provided as part of the OSR FSDK.

Conventions

In this document, we have adopted the following conventions:

We will use the name "Windows" when referring to Windows 2000, Windows XP, Windows Server 2003 or Windows Vista. This release of the FSDK will support Windows 2000, Windows XP, Windows Server 2003 and Windows Vista. If your file system does not rely upon O/S specific features, it should be binary compatible with the three separate versions of the FSDK wrapper, although the FSDK Wrapper library will differ between the systems.

Parameters to functions are shown in *italics*. The names used are intended to be demonstrative of what a given parameter represents.

We refer to the OSR-provided FSDK library as the Wrapper.

We refer to the file system specific code as the FSD (for "File System Driver").

In naming the various functions, those with an FS_ prefix are routines that are to be provided by the FSD as entry points to be called by the Wrapper. The names used are those of their actual type declaration. The headers we have used are intended to be descriptive of the function provided by the given FSD entry point.

Routines beginning with Ow are provided by the OSR Wrapper and are intended for use by the FSD when using Wrapper-provided services.

The general model for all operations is to describe the entry point, the parameters, and the general function of the operation, as well as a general discussion of the potential return parameters.

Finally, we note the status of a particular interface call, that is, if implementing the function in the FSD is optional or mandatory.

Status

While every attempt has been made to ensure that the information is correct in every detail, the nature of the complex technical material herein may result in errors or inconsistencies. As they are found and brought to our attention, this information will be revised and updated. We strongly encourage anyone using this information to send us their questions or suggestions for improvements. Updated versions will be made available from time to time via the normal support channels: either as part of your update or via the OSR secure FTP server, ftp://ftp.osr.com. Please contact OSR for information on obtaining an account on this server.

Comments should be sent to fsdkbugs@osr.com. This mechanism ensures optimal response to any problems or issues. Issues raised via any other channel may experience unnecessary delay times or even be inadvertently overlooked. This describes the interface for the FSDK Wrapper. Fundamentally, the Wrapper component of the FSDK is the core portion which implements file system functionality specific to the Windows operating system.

CHANGES IN THE V2.5 RELEASE

Overview

The primary focus in FSDK V2.5 was to improve performance, notably in the area of parallel activity. Prior to this version of the FSDK, a number of coarse-granularity locks were being used internally that effectively made key operations largely serial in nature, leading to bottlenecks for some applications. Our goal was to remove these bottlenecks by increasing parallel activity.

Generic Lock Package

Our first task was to improve the internal locking behavior of the FSDK, which was supported by the *generic lock package.* This package provides an infrastructure for tracking all locks used by a given thread within the FSDK and enforces a locking hierarchy. Such a strict ordering model allowed us to change the internal FSDK locking and ensure that it was done so in a deadlock free fashion. Any problems observed will immediately cause the FSDK to halt and report the issue, providing valuable information about the current lock state. Generally, bug reports for lock issues that report the current call stack and the debug output (the lock list printed just before the breakpoint) are sufficient for us to debug any specific issues that you might observe.

Parallel Locking

Internally, the FSDK now uses a parallel table of locks to protect its internal file control block (FCB) lookup table. Correctly supporting this involves calling into your FSD to obtain a handle, acquiring the correct lock, calling into your FSD a second time to ensure the file handle has not changed and then locating the correct cached FCB data structure (if the handle has changed, the FSDK will acquire the correct lock for the new handle and repeat this process.) Accordingly, you may find it is performant to cache information about the last file looked up in a given directory.

Impact on FSDK-based File Systems

These revisions do not change the FSDK interface into the file system. Rather, they change the behavior characteristics of the FSDK and how the FSDK interacts with your file system. Where previously you would have seen serialized activity in your FSD, now you will observe potentially parallel activity within your file system driver.

These performance improvements should allow you to further tune the performance of your own file system products. While this will benefit all of our customers, we anticipate the greatest gains will be observed by our customers whose products are used in high end workstation and all server environments. The increased levels of concurrent behavior now present throughout the FSDK, may uncover latent race conditions or other bugs within your FSD. Thus, for those moving their FSD to FSDK V2.5, we strongly encourage you to budget additional time for testing/validation, particularly for parallel file activity.

COMMON DEVELOPMENT ISSUES

Overview

We have attempted to codify and discuss issues that commonly arise when using the FSDK for development. Please note that while we have attempted to identify these common issues this list should not be considered as definitive. Indeed, our experience is that each customer utilizes the FSDK in a slightly different fashion and as a result each customer has unique issues.

Handles and Handle Management

Perhaps one of the most confusing issues with respect to the FSDK is the use of "handles". Because the FSDK allows upcalls using these handles, the FSDK internally maintains lookup tables that are indexed by handle.

While this approach seems simple, it has some unfortunate side effects.

First, it requires that handles be unique. Otherwise, when we receive a call from a file system to perform an operation we do not know precisely what is required (imagine a call to change caching characteristics, for example.)

While uniqueness might seem to be a simple property, it turns out to be considerably more complicated, mostly in the "edge condition" cases. For example, when a user performs a forced dismount or forced disconnection, the FSDK must continue to maintain sufficient state to manage calls from the application programs (even though the backing file system is no longer present.) Accordingly, these handles (which are now released from the perspective of the file system) are still active and in use (from the perspective of the FSDK.) Until the files are closed, the FSDK must continue to maintain that state.

Over the development cycles of the FSDK we have strived to "clean up" all of these edge cases so that file handles can be safely reused. However, *volume* handles are not safe for reuse. Thus, we strongly suggest that you attempt not to reuse them whenever possible.

Typically, a file system uses nothing more than a pointer structure (an address as it were) for tracking these handles. This approach, while simple, does not lend itself well to preventing handle re-use. A simple approach that substantially minimizes the likelihood of handle re-use is to exploit the nature of the Windows memory allocator. As documented, the memory allocator never allocates addresses with the low three bits set. Thus, a simple scheme that greatly minimizes the likelihood of handle re-use is to maintain a global variable within your driver. Each time you allocate a new volume handle, you increment that global value, OR the low three bits into your handle (the address returned by the memory allocator) and return that result as the handle to the FSDK.

Then, when the FSDK calls your file system using that handle, you mask off the low three bits and use the corresponding address.

Note: this behavior for the memory allocator is documented in the Device Driver Kit and is architecturally required for some (no longer supported) hardware platforms (due to alignment constraints on 64-bit data

value access.) While the reasons for this have been eliminated, this restriction is sufficiently ingrained in the operating system that it can be used reliably.

Accessing the IRP

In earlier versions of the FSDK, we made the IRP available during the IRP_MJ_CREATE processing. As of V2.0 of the FSDK, we store the IRP in thread local storage so that your file system may identify and extract information from the IRP that is currently being processed by the FSDK. In cases where multiple IRPs are being processed, the FSDK will return the last IRP initiated. For example, it is common for an IRP_MJ_CLEANUP to cause an IRP_MJ_CLOSE to occur with respect to another file object. If this requires a call into your file system, the FSDK will provide you with the IRP_MJ_CLOSE.

> *This feature is only partially implemented. If you have specific cases that have not yet been converted to this new model, advise us and we will prioritize the conversion of that portion of the FSDK to include this feature.*

In addition to the **OwGetTopLevelIrp** API, we have also introduced a separate filtering mechanism. In this model, you register callback functions which are called as each IRP arrives into the FSDK and again just before the FSDK completes the I/O request. Our goal was to eliminate the need for the use of a file system filter driver in conjunction with the FSDK.

The general convention for FSDK-style filtering is:

- If your dispatch function returns anything other than STATUS_SUCCESS, the FSDK will assume that the I/O operation has been "claimed" and it will **not** be processed by the FSDK. Thus, your file system is responsible for processing and completing the request.
- If your completion function returns **STATUS_MORE_PROCESSING_REQUIRED** then the FSDK will not complete the request. Thus, your file system is responsible for processing and completing the request.

We note that if your file system "claims" the I/O request in a dispatch function, the FSDK will not provide a callback for completion processing (this is because your file system will be performing the completion, not the FSDK.)

Most customers using the FSDK will not find it necessary to utilize this functionality. If you do wish to use this functionality, please make sure that you communicate with the OSR support team to ensure your usage is fully supported during the development cycle.

Managing Upcalls

Calling from your file system back into the FSDK is one of the most complicated functions that you can perform. The FSDK itself must maintain and control serialization in order to ensure correct (deadlock-free) behavior. The upcalls can, in turn, trigger additional calls down into your file system.

Thus, when using upcalls you must take into account:

- FSDK Locking state. In general, it is safe to call back into the FSDK to perform operations on the **same** file handle, but not to perform operations on a different file handle. Otherwise, you will introduce lock cycles (which cause deadlocks under the right circumstances). Although we have gone to great lengths to avoid deadlocks, in general it is not good practice to call back into the FSDK to perform operations that change the state of a file/directory/volume.
- Reentrancy state. Some calls into your file system are not safe to use for upcalls. The notable examples of this would be **FS_READ** and **FS_WRITE** because these operations can be performed as part of page fault processing and/or at high priority. For example, I/O operations cannot be completed when the system is running at **APC_LEVEL** – and there are some paging I/O operations that run at **APC_LEVEL** to eliminate reentrancy.
- File System Locking State – as we noted previously, the FSDK may perform operations that trigger a call back into your file system. If this occurs, you will need to ensure that this does not cause a deadlock situation. Accordingly, you must either use re-entrant locks, such as ERESOURCES, thread level state tracking, or some other mechanism for managing reentrancy.

Caching

There is considerable confusion about the effect of caching with respect to file systems using the FSDK. Perhaps one of the most important considerations here is that caching causes the file to remain open for a potentially indefinite period of time. Thus, normally, customers will report that their files are "never" released. Normally, however, we find that the file is in fact open because it is still cached. Thus, the FSDK must maintain the open reference. This in turn tells the file system that the file state must be tracked.

A file system that wishes to manipulate caching has a number of options when using the FSDK:

- Disable caching. While this will "solve" the problem, it yields very poor performance. Further, the FSDK will cache certain files (essentially anything that is memory mapped) because the alternative is sacrificing correct function.
- Disable write-back caching. In this case, all files are marked as "write through" and thus any dirty data is flushed immediately. Clean (that is, unmodified with respect to the state in the file system) data can be served from the cache.
- Invalidate caching as necessary. There are upcalls (OwFlushCache, OwPurgeCache) available for invalidating the cache as necessary. These calls must be made when it is safe to perform such upcalls. The FSDK, as of V2.0, greatly improves the ability of these calls to succeed by posting portions of the necessary work, thereby eliminating lock contention issues that have been a problem in earlier versions.
- Allow caching. This is certainly the simplest in terms of implementation, although it is not suitable for all projects.

The ramifications of caching decisions in your file system are very important. This issue can impact the design and implementation of your file system. If you have any questions about this, you should submit them to fsdkbugs@osr.com as part of your normal support.

Drive Letter Management

Drive letter management turns out to be a complicated part of Windows. Further, the manner in which the drive letters are managed for Windows 2000 and beyond has been completely changed.

In Windows 2000 and above, the drive letter management becomes more complicated because of the introduction of support for dynamic ("plug and play") devices. Thus, static assignment when the system is initialized is insufficient. Windows 2000 introduces the "mount manager" in order to handle this. The APIs necessary to communicate with the Mount Manager are documented in the Win32 API documentation (part of the Platform SDK.)

Locking Considerations

In developing an FSD, careful consideration must be made to the locking being performed by the FSDK when these wrapper support routines are called. Specifically, if an FSD is performing a call into a wrapper support routine it is imperative that only operations on that file and the volume containing that file are performed.

This is due to the locking model supported within the FSDK wrapper itself. Because there is no way for the FSDK to ensure locks are acquired in the proper order under all circumstances, an FSD must obey these restrictions. A failure to do so will lead to internal FSDK deadlocks.

Attribute Operations

File attribute management and the associated calls for it make up a substantial part of the overall interface exported by an FSD. These attributes include such things as security information, extended attributes, as well as more traditional attribute information (such as archive, read-only, etc.)

For each of these operations we utilize a *Get* interface to retrieve the information from the FSD. The FSDK Wrapper is then responsible for handling all presentation issues with respect to the system. The corresponding *Set* interface is responsible for translating information from the system, formatting that information into a canonical form, and providing it to the FSD for storage.

Thus, the general model here is that an FSD provides the *storage* for attributes, but does not provide the interpretation of those attributes.

Of course, there are exceptions to this model. The OS has a set of "attributes" which are always assumed to be available for each file (access, modify, and create times, data length, allocation size, etc.). These attributes are manipulated via a get/set interface call for these "generic" attributes.

As with file data, the Wrapper may cache file and directory attributes to attempt to minimize the number of times they are modified on-disk. Thus, when the Wrapper calls to the underlying FSD requesting modification of an attribute, the FSD should write the attributes to persistent storage. Similarly, if these values are externally modified the Wrapper cache must be purged using the appropriate functions.

FSDK SUPPORT INTERFACE

Overview

This section describes, in alphabetical order, the support routines exported by the FSDK and available for use within a file system driver.

Note that not all functions are required for all file systems. Some functions, such as those exported for media file systems, are optional use and appropriate only to a specific type of file system.

Wrapper Support Routines

The Wrapper provides the routines described in this section for use in FSD implementations. In general, the model for the Wrapper is *not* to isolate the FSD from the standard Windows DDK primitives. Thus, the routines in this section are here to export file systems specific functionality being provided by the Wrapper but requiring some form of access for the FSD.

OwAllocateSDBuffer

```
NTSTATUS
OwAllocateSDBuffer(
    IN ULONG BufferSize,
    OUT PVOID Buffer);
```

Parameters:

BufferSize — The size of the buffer needed for the Security Descriptor (SD)

Buffer — This is the allocated buffer, which can be NULL

Description:

This routine is used to allocate a security descriptor buffer.

There is no equivalent "free" because the FSDK will handle freeing the security descriptor.

Returns:

STATUS_SUCCESS - The buffer has been allocated

STATUS_INSUFFICIENT_RESOURCES - The buffer could not be allocated

STATUS_INVALID_BUFFER_SIZE - The buffer size was zero

STATUS_INVALID_PARAMETER-2 – The buffer is invalid

OwDeregister

```
NTSTATUS
OwDeregister(
    IN PVOID RegistrationHandle);
```

Parameters:

RegistrationHandle — The handle that was returned when the FSD was registered using OwRegister.

Description:

This indicates the FSD will no longer be using the Wrapper functions. Subsequent calls into the Wrapper will fail.

This call will fail whenever there are outstanding references to the FSD or volumes being managed by the FSD. Note that this call is optional and need not be used by an FSD. Typically, Windows file systems do not dynamically unload.

This should not be called as part of shutdown processing – it is implemented to allow for the dynamic unloading of FSDs. Use during shutdown processing may lead to unpredictable results.

For a network file system, this call must be made in the same process context as the original call to OwRegister. This is because the Wrapper will register the FSD as a UNC provider; this in turn generates a handle that must be valid when OwDeregister is called. To ensure the handle is valid, OwDeregister must be called in the same process context (since handles are process context specific.)

Thus, if your network file system uses a service thread context to issue the OwRegister call, you must also use the service thread context to issue the OwDeregister call. Alternatively, if you register in your FSD's DriverEntry routine, you must deregister in the system process context, typically using a system worker thread. You can do this by creating a work item and using the Executive worker routines, such as ExQueueWorkItem (documented in the Windows DDK.)

Returns:

STATUS_SUCCESS - the deregistration was completed successfully

STATUS_INVALID_DEVICE_REQUEST – there are still active volumes for this file system; deregistration is not permitted

OwDisableFileCache

```
NTSTATUS
OwDisableFileCache (
    IN FS_FILE_HANDLE FileHandle,
    IN BOOLEAN DisableReadCaching);
```

Parameters:

FileHandle —The handle provided by the FSD to identify a given file or directory instance.

DisableReadCaching —If TRUE, read caching is disabled as well as write caching.

Description:

This routine is called by the FSD to indicate that data for this file (or directory) should not be cached. If the *DisableReadCaching* variable is TRUE, no caching will be performed. If the *DisableReadCaching* variable is FALSE, data may be cached, but all write operations will be done via a write-through mechanism.

This approach allows the FSD to tightly control the caching semantics on a per file basis.

The Wrapper is not allowed to cache attribute information for a file or directory that has caching disabled. For a directory, the Wrapper is not allowed to cache the directory enumeration.

Returns:

STATUS_SUCCESS – the cache as been successfully purged and the file/directory is now marked so that it cannot be cached.

STATUS_INSUFFICIENT_RESOURCES – a work item could not be allocated; the state of the file has not been modified.

STATUS_INVALID_HANDLE – the handle passed in by the FSD is not valid; the state of the file has not been modified.

STATUS_USER_MAPPED_FILE – the file in question is mapped by a user application and caching cannot be disabled for this file; the state of the file has not been modified.

STATUS_OPLOCK_BREAK_IN_PROGRESS – the file in question may have had an outstanding oplock. A break of that oplock is in progress. The state of the file has been updated to disable caching.

STATUS_RETRY – due to lock state within the FSDK, the operation could not be completed immediately. The request has been posted and the FSD should retry this at a later time. The state of the file has not yet been updated.

OwDisableVolumeCache

```
NTSTATUS
OwDisableVolumeCache (
    IN  FS_VOL_HANDLE FsdVolumeHandle,
    IN BOOLEAN DisableReadCaching);
```

Parameters:

FsdVolumeHandle —The handle provided by the FSD describes a volume instance.

DisableReadCaching —If TRUE, read caching is disabled as well as write caching.

Description:

This routine is called by the FSD to indicate that data for any file on the specified volume should not be cached. If the *DisableReadCaching* variable is TRUE, no caching will be performed. If the *DisableReadCaching* variable is FALSE, data may be cached, but all write operations will be done via a write-through mechanism.

These caching state changes are effective only for files opened subsequent to this change.

The Wrapper is not allowed to cache attribute information for a file or directory that has caching disabled. For a directory, the Wrapper is not allowed to cache directory enumerations.

Returns:

STATUS_INVALID_HANDLE – the handle passed in by the FSD is not valid; the state of the volume has not been modified.

STATUS_SUCCESS – future files opened on this volume will be marked so that their data and attributes may not be cached.

OwEnableFileCache

```
NTSTATUS
OwEnableFileCache (
    IN PVOID FileHandle,
    IN BOOLEAN EnableWriteCaching);
```

Parameters:

FileHandle —The handle provided by the FSD to identify a given file or directory instance.

EnableWriteCaching —Indicates if write caching should be enabled, in addition to read caching.

Description:

This routine is called by the FSD to indicate that data for this file should be cached. If the *EnableWriteCaching* variable is FALSE, data may be cached, but all write operations will be done via a write-through mechanism.

This approach allows the FSD to tightly control the caching semantics on a per file basis.

The Wrapper is not allowed to cache attribute information for a file or directory that has caching disabled.

Returns:

STATUS_INVALID_HANDLE – the handle passed in by the FSD is not valid; the state of the file has not been modified.

STATUS_SUCCESS – this file has been marked so that its data and attributes may be cached.

OwEnableVolumeCache

```
NTSTATUS
OwEnableVolumeCache (
    IN FS_VOL_HANDLE FsdVolumeHandle,
    IN BOOLEAN EnableWriteCaching);
```

Parameters:

FsdVolumeHandle —The handle provided by the FSD describes a volume instance.

EnableWriteCaching —Indicates if write caching should be enabled, in addition to read caching.

Description:

This routine is called by the FSD to indicate that data for any file on the specified volume should be cached. If the *EnableWriteCaching* variable is TRUE, full data caching will be performed. If the *EnableWriteCaching* variable is FALSE, data may be cached, but all write operations will be done via a write-through mechanism.

These caching state changes are effective only for files opened subsequent to this change.

The Wrapper is not allowed to cache attribute information for a file or directory that has caching disabled.

Returns:

STATUS_INVALID_HANDLE – the handle passed in by the FSD is not valid; the state of the volume has not been modified.

STATUS_SUCCESS – future files opened on this volume will be marked so that their data and attributes may be cached.

OwFlushCache

```
NTSTATUS
OwFlushCache (
    IN FS_FILE_HANDLE FileHandle,
    IN BOOLEAN Purge);
```

Parameters:

FileHandle —The handle provided by the FSD to identify a given file or directory instance.

Purge —If this value is TRUE, it indicates that the cache should be purged after the data has been written back.

Description:

This routine would be called to request that the wrapper flush cached data for the given file. Thus, if there is any *dirty* data stored in the VM system, it will be written back to the FSD (via the FS_WRITE entry point in the FSD.)

This function, while available to any file system, is likely to be of use only to network file systems to aid in their implementation of a cache consistency mechanism. Certain restrictions imposed by Windows are inherent in the Wrapper support of this interface – notably, the inability of the VM system to purge cached data for files mapped into a user address space.

The *Purge* parameter may be used to avoid a subsequent call to OwPurgeCache once the data has been written back.

This routine returns STATUS_SUCCESS if the flush and purge (if requested) was successful. Otherwise it returns an appropriate error code.

One very common error code is STATUS_USER_MAPPED_FILE that indicates the file is mapped into a user application space and the caller specified that the file should be *purged.* In such cases, it is not possible for the Wrapper to purge the data in the cache.

The purpose of this call is to allow an FSD to implement its own cache consistency protocol. Thus, typically, this call will only be used by network file systems, rather than media file systems. However, it is permissible for any file system to access this operation.

Calling this routine requires the Wrapper to discard any cached attribute information regarding the file.

Returns:

STATUS_SUCCESS - the flush operation was successful

STATUS_FILE_LOCK_CONFLICT – the operation could not be performed, as it was not possible to acquire the necessary locks.

STATUS_USER_MAPPED_FILE - the purge could not be done because a user application has mapped the file.

© | OSR Open Systems Resources, Inc.

OwGetCharacteristics

```
VOID
OwGetCharacteristics(
    IN PVOID MediaHandle,
    OUT PULONG DeviceType,
    OUT PULONG DeviceCharacteristics);
```

Parameters:

MediaHandle — The media-based handle provided to the FSD by the Wrapper.

DeviceType — Indicates the *type* of device.

DeviceCharacteristics — The characteristics of the media device.

Description:

This routine is used by the FSD to obtain information about the underlying media device. The *DeviceType* corresponds to the type of device object (the value provided to *IoCreateDevice*) and the *DeviceCharacteristics* correspond to the features of the device object (e.g., *removable media)*. These correspond exactly to the values described in the Windows DDK.

Returns:

None.

OwGetDirectorySearchString

```
NTSTATUS
OwGetDirectorySearchString (
    IN OUT PUNICODE_STRING SearchString);
```

Parameters:

SearchString —The specific regular expression being used for the current directory enumeration.

Description:

This call is provided to allow an FSD that does not support directory caching to retrieve additional information about the directory enumeration operation currently active. Because this information is not being cached, a subset of the actual directory may be returned to the FSDK Wrapper.

Note that an FSD that relies upon directory caching is prohibited from using this interface. Further, an FSD that uses this interface must be able to handle standard file system regular expressions:

* - indicates zero or more characters

? – indicates exactly one character

“ – indicates the value “.”

> - indicates exactly one character, but may not be a space character either immediately before a “.” or as the last character of the string.

< - indicates zero or more characters but may not be a space character either immediately before a “.” or as the last character of the string.

The last two characters are unique to the Virtual Dos Machine (VDM) implementation on Windows and indicate that *DOS* file name matching semantics are to be used. The double quotation mark is used to distinguish a DOS period.

Returns:

STATUS_UNSUCCESSFUL – the request could not be honored because the string information was not valid for this request.

STATUS_BUFFER_TOO_SMALL – the provided string buffer was not large enough to contain the search string.

STATUS_ACCESS_VIOLATION – the provided string buffer was not valid.

STATUS_SUCCESS – the string has been returned in the provided buffer.

© | OSR Open Systems Resources, Inc.

OwGetFsdHandleForFileObject

```
NTSTATUS
OwGetFsdHandleForFileObject (
    IN PFILE_OBJECT FileObject,
    IN OUT PFS_FILE_HANDLE FileHandle);
```

Parameters:

FileObject —A file object provided by the FSD to identify a given open application file instance.

FileHandle —The handle provided by the FSD to identify a given file instance.

Description:

This function is provided to allow an FSD developer to pass file handles directly to their FSD (via the I/O control interface). When this is done, an FSD may use the Windows DDK function *ObReferenceObjectByHandle* and pass the resulting file object to the FSDK Wrapper.

The FSDK Wrapper will extract the FSD's internal handle from the file object and return it.

Returns:

STATUS_SUCCESS – the returned file handle is valid

STATUS_OBJECT_TYPE_MISMATCH – the provided object was not a file object

STATUS_INVALID_HANDLE – the provided file object was not created by the FSDK.

STATUS_ACCESS_VIOLATION – the *FileHandle* pointer was not valid. The FSD is responsible for allocating the storage for this pointer.

OwGetMediaDeviceObject

```
NTSTATUS
OwGetMediaDeviceObject(
    IN PVOID MediaHandle,
    OUT PDEVICE_OBJECT *DeviceObject);
```

Parameters:

MediaHandle — The media-based handle provided to the FSD by the Wrapper.

DeviceObject — The Wrapper will set this to point to the underlying device object containing the specified media volume.

Description:

This entry point is provided to allow an FSD to implement its own lower edge interface rather than relying upon the Wrapper to implement it. This call, given a *MediaHandle,* returns a pointer to the device object representing this volume. The FSD may then build its own I/O requests and send them to the underlying media device driver.

Mixing the Wrapper implementation and the FSD implementation is possible but should be used with extreme care as this may lead to unpredictable results.

Returns:

STATUS_SUCCESS – the returned device object is valid STATUS_INVALID_HANDLE – the provided media handle is not a valid FSDK media handle.

OwGetPseudoDeviceObject

```
NTSTATUS
OwGetPsuedoDeviceObject(
    IN PVOID PseudoVolumeHandle,
    OUT PDEVICE_OBJECT *DeviceObject);
```

Parameters:

PseudoVolumeHandle— The pseudo volume handle provided to the FSD by the Wrapper.

DeviceObject — The Wrapper will set this to point to the underlying device object representing the specified pseudo volume.

Description:

This entry point is provided to allow an FSD to implement its own lower edge interface rather than relying upon the Wrapper to implement it. This call, given a *PseudoVolumeHandle,* returns a pointer to the device object representing this volume. The FSD may then build its own I/O requests and send them to the underlying device driver.

Mixing the Wrapper implementation and the FSD implementation is possible but should be used with extreme care as this may lead to unpredictable results.

Returns:

STATUS_SUCCESS – the returned device object is valid

STATUS_INVALID_HANDLE – the PseudoVolumeHandle provided is not a valid FSDK pseudo volume handle.

OwGetTopLevelIrp

```
PIRP
OwGetTopLevelIrp
 (void);
```

Description:

This function may be used by the FSD to retrieve the IRP that is currently being processed. The FSD is responsible for interpreting the contents of this value and handling it as appropriate.

Returns:

NULL – an IRP is not available

Any other value represents the IRP currently being processed

OwIoControl

```
NTSTATUS
OwIoControl(
    IN PVOID MediaHandle,
    IN ULONG IoctlCode,
    IN PVOID InputBuffer,
    IN ULONG InputBufferSize,
    OUT PVOID OutputBuffer,
    OUT ULONG OutputBufferSize);
```

Parameters:

MediaHandle — This is the media-based handle provided to the FSD by the Wrapper for use when calling into the lower-edge interfaces.

IoctlCode — The identifier for the FSD-specific control operation.

InputBuffer — The input information (if any) for this control operation.

InputBufferSize — The size of the information contained within the input buffer.

OutputBuffer — The location where *output* information for this control operation should be written (if any).

OutputBufferSize — The amount of data contained within the output buffer.

Description:

This routine can be used by an FSD to send IOCTL codes to the underlying media device itself. The Wrapper returns the results of those calls to the FSD. The Wrapper does not interpret the calls and the IOCTL values supported by the underlying media device are entirely dependent upon the implementation of the media device's driver.

Normally this is used by an FSD to interact with its underlying media using standard disk storage IOCTLs, including but not limited to:

IOCTL_GET_DRIVE_GEOMETRY

IOCTL_IS_WRITEABLE

IOCTL_CDROM_READ_TOC

IOCTL_STORAGE_GET_MEDIA_TYPES

FT_QUERY_SET_STATE

The FSD is responsible for allocating and freeing storage for both the *input* and *output* buffer.

Returns:

The status code returned by the underlying media device.

OwMediaRead

```
NTSTATUS
OwMediaRead(
    IN PVOID MediaHandle,
    IN LARGE_INTEGER Offset,
    IN ULONG Length,
    IN PMDL MdlChain);
```

Parameters:

MediaHandle — This is the media-based handle provided to the FSD by the Wrapper for use when calling into the lower-edge interfaces.

Offset — This is the disk-based (volume) offset to be used when reading data from the media device.

Length — The amount of data to be read from the underlying volume.

MdlChain — This describes the MDL chain where the data read from disk should be placed.

Description:

The OwMediaRead routine is used by the FSD to transfer data from the disk into the specified MDL. The *Offset* must be on a sector boundary, and the *Length* must be a multiple of the sector size.

Note that an *MdlChain* may be passed to this routine. Should the underlying device be unable to handle an MDL chain, the I/O operations will be structured as a series of related I/O operations.

Returns:

STATUS_SUCCESS – The read was successful.

Other I/O errors as returned by the underlying subsystems.

> *OwMediaRead must not be used when accessing a paging file. This is because of the special restrictions the operating system imposes when accessing a paging file.*

OwMediaWrite

```
NTSTATUS
OwMediaWrite(
    IN PVOID MediaHandle,
    IN LARGE_INTEGER Offset,
    IN ULONG Length,
    IN PMDL MdlChain);
```

Parameters:

MediaHandle — This is the media-based handle provided to the FSD by the Wrapper for use when calling into the lower-edge interfaces.

Offset — This is the disk-based (volume) offset to be used when writing data to the media device.

Length — The amount of data to be written to the underlying volume.

MdlChain — This describes the MDL chain where the data to be written to disk is located.

Description:

The OwMediaWrite routine is used by the FSD to transfer data to the disk from the specified MDL. The *Offset* must be on a sector boundary, and the *Length* must be a multiple of the sector size.

As with the *read* case, the Wrapper is responsible for handling the MDL chain and ensuring that the complete I/O operation has been satisfied.

Returns:

STATUS_SUCCESS – The write was successful.

Other I/O errors as returned by the underlying subsystems.

> *OwMediaWrite must not be used when accessing a paging file. This is because of the special restrictions the operating system imposes for accessing paging files.*

OwNotifyDirectoryChange

```
NTSTATUS
OwNotifyDirectoryChange (
    IN FS_FILE_HANDLE FileHandle,
    IN PUNICODE_STRING FullTargetName,
    IN USHORT TargetNameOffset,
    IN PUNICODE_STRING StreamName,
    IN PUNICODE_STRING ParentName,
    IN ULONG FilterMatch);
```

Parameters:

FileHandle —The file handle, upon which we are operating. This file handle uniquely identifies the directory whose contents have changed.

FullTargetName —The path name of the file within the directory that changed.

TargetNameOffset —The offset in the FullTargetName to the file name component of the name.

StreamName —This is optional and indicates the stream name if it was a stream of the given file that changed.

ParentName —This is optional and indicates the name of the DIRECTORY that changed (for multi-linked directories)

FilterMatch —This indicates what changed. The FSD passes one or more of the values listed below into OwNotifyDirectoryChange so the FSDK can tell the operating system what has happened to the specified file.

FILE_NOTIFY_CHANGE_FILE_NAME

FILE_NOTIFY_CHANGE_DIR_NAME

FILE_NOTIFY_CHANGE_NAME

FILE_NOTIFY_CHANGE_ATTRIBUTES

FILE_NOTIFY_CHANGE_SIZE

FILE_NOTIFY_CHANGE_LAST_WRITE FILE_NOTIFY_CHANGE_LAST_ACCESS

FILE_NOTIFY_CHANGE_CREATION

FILE_NOTIFY_CHANGE_EA

FILE_NOTIFY_CHANGE_SECURITY

FILE_NOTIFY_CHANGE_STREAM_NAME

FILE_NOTIFY_CHANGE_STREAM_SIZE

FILE_NOTIFY_CHANGE_STREAM_WRITE

Description:

This call, provided for multi-initiator file system implementations, provides a mechanism whereby the FSD can notify the wrapper of an external event which triggers an FSD change. In other words, when a change to the underlying file system occurs outside the scope of the FSDK this call may be used to indicate a change has occurred in the specified directory.

As a side effect of this call, any directory information that is cached is also discarded.

Note that if the file system in question supports hard links, each directory in use to access this file will receive a separate directory change notification.

Returns:

STATUS_SUCCESS – the directory change notification has been successfully registered.

STATUS_INVALID_DEVICE_STATE – this call was made at IRQL > PASSIVE_LEVEL

STATUS_INVALID_HANDLE - the file handle provided was not valid

STATUS_NOT_A_DIRECTORY - not a directory

STATUS_INSUFFICIENT_RESOURCES – not enough memory

OwPostWork

```
VOID
OwPostWork(
    IN POW_WORK WorkToDo);
```

Parameters:

WorkToDo — The work to perform, specified by calling the desired function within this parameter.

Description:

The OwPostWork routine may be used by an FSD to request that an FSD-provided function be called in a worker thread to perform a background operation. This will be accomplished by using one of the FSDK internal work queues. The WorkToDo will be described using the OW_WORK structure.

This interface provides an alternative to system queues.

Returns:

None.

OwPurgeCache

```
NTSTATUS
OwPurgeCache (
    IN FS_FILE_HANDLE FileHandle);
```

Parameters:

FileHandle —The handle provided by the FSD to identify a given file or directory instance.

Description:

This routine is called by the FSD to indicate that any cached data for this file (or directory) should be purged.

This call may be made by any FSD. Typically it is used for:

- Implementing a distributed cache coherency protocol, such as in the case of a network file system or shared media (SAN) file system.
- To ensure that a file is released because as long as there are VM references to the file, the FSDK cannot release the file handle.
- To implement other policy of the file system that requires the data for this file is discarded. For example, a file system that supports encryption or compression might allow modification of both forms of the file. Thus, a change to one form of the data would require that the other form of the data be discarded.

Please note that the implementation of this functionality by the Wrapper is subject to restrictions placed on purging user-mapped files by the Windows VM system implementation. Thus, a request to purge a file may *fail* because it is not possible to discard the VM references to the file.

Note that the FSDK does not implement a distributed cache coherency protocol, although it does implement a cache coherency protocol (consistent with existing Windows file systems) on the same system. Thus, it is the responsibility of the FSD to implement any necessary cache consistency protocol.

It is important to note that Windows does *not* guarantee cache coherency between non-cached I/O and memory mapped I/O. Thus, it is possible to observe certain race conditions in updating the data. In this case, the behavior of the system is “last writer wins”. This is **not** a function of the FSDK. Instead, it is an operational restriction of the Windows virtual memory system.

Calling this routine requires the Wrapper to discard any cached attribute information it has stored regarding this file. Typically, this will result in a subsequent release of the file handle, although that is not guaranteed.

Note that for a *directory* this will purge any cached directory information (such as the directory enumeration.)

Returns:

STATUS_SUCCESS – the cache has been successfully purged.

STATUS_INSUFFICIENT_RESOURCES – a work item could not be allocated.

STATUS_INVALID_HANDLE – the handle passed in by the FSD is not valid.

STATUS_USER_MAPPED_FILE – the file in question is mapped by a user application and the cache for this file cannot be purged.

STATUS_OPLOCK_BREAK_IN_PROGRESS – the file in question may have had an outstanding oplock. A break of that oplock is in progress.

STATUS_RETRY – due to lock state within the FSDK, the operation could not be completed immediately. The request has been posted and the FSD should retry this at a later time.

OwRegister

```
NTSTATUS
OwRegister(
    IN PDRIVER_OBJECT FsdDriverObject,
    IN PUNICODE_STRING RegistryPath,
    IN ULONG VersionNumber,
    IN FS_TYPE FsdType,
    IN PFS_OPERATIONS FsdOperations,
    OUT PVOID *RegistrationHandle);
```

Parameters:

FsdDriverObject — This is the driver object passed to the FSD in its *DriverEntry* routine.

RegistryPath — This is the registry path provided to the FSD in its *DriverEntry* routine. Parameters associated with this FSD will be stored under this key and will be read by the Wrapper.

VersionNumber — The wrapper version number that the FSD expects to be supported.

FsdType — Indicates the *type* of file system being registered with the Wrapper.

FsdOperations — These are file system operations that are supported by *this* FSD. Entries not supported are set to zero within this table.

RegistrationHandle — A handle to be used by the FSD when deregistering with the Wrapper library.

Description:

This entry point must be used by the FSD to register with the Wrapper library itself. The FSD provides versioning information and if the Wrapper is incompatible with the version specified, the registration request will be rejected.

The *RegistrationHandle* returned by the Wrapper may be used to *deregister* this FSD with the Wrapper.

The *FsdType* is structured to consist of a *major* file system type and then a set of characteristics for each file system. The major types, along with their characteristic are:

1. Media file system (OW_FS_TYPE_MEDIA)

   Media file systems minor types, representing the characteristics, are:

   Disk file system (OW_FS_TYPE_MEDIA_DISK)

   Tape file system (OW_FS_TYPE_MEDIA_TAPE)

   CD ROM file system (OW_FS_TYPE_MEDIA_CDROM)

   At least one of these minor types must be specified in addition to OW_FS_TYPE_MEDIA for media file systems. Multiple options may also be specified by an FSD. These values are specific to the type of

media on which the file system might be stored. For any media type specified as part of the registration operation, the FSD may be asked to mount the given media.

2. Network file system (OW_FS_TYPE_NETWORK)

Network file systems minor types, representing the characteristics, are:

- Named pipe supporting file system (OW_FS_TYPE_NETWORK_NAMED_PIPE)
- Mail slot supporting file systems (OW_FS_TYPE_NETWORK_MAILSLOT)

The named pipe information is not used in this release and is reserved for future use.

Any combination of options is supported. Minor types are not required for network file systems. At present, the Wrapper does not implement named pipe support.

3. Pseudo file system (OW_FS_TYPE_PSEUDO)

The *FsdOperations* vector contains key information about the entry points implemented by the FSD that in turn may be called by the Wrapper. Any unused entry must be set to zero.

Note that the Wrapper will initialize the *FsdDriverObject* as a result of this call. An FSD that modifies the FsdDriverObject's dispatch entry table or Fast I/O table can lead to unpredictable results when using the file system.

The RegistrationHandle should be used by the FSD in any subsequent call to OwDeregister.

The Wrapper will create a named device object and symbolic link in the Win32 name space using the name provided by the registering FSD. If this name conflicts with an existing device object or symbolic link the registration call will fail. These named device objects may be subsequently used by custom utilities attempting to interact with the FSD.

Thus, for a Network file system, if the name specified in the *FsdOperations* vector were "MyFsd", the wrapper would create a device object called "\Device\MyFsd" and a symbolic link called "\DosDevices\MyFsd". For a Media file system, we create the name at the top level of the tree.

Returns:

STATUS_SUCCESS - the FSD has been successfully registered with the Wrapper.

STATUS_INVALID_DEVICE_REQUEST - indicates that the Wrapper was loaded incorrectly.

STATUS_REVISION_MISMATCH - indicates that the FSD being registered is not compatible with the Wrapper.

STATUS_INVALID_PARAMETER_4 - indicates the *FsdType* parameter is not one of the valid file system types

STATUS_INSUFFICIENT_RESOURCES - memory allocation failed within the Wrapper

STATUS_ACCESS_VIOLATION - The *FsdOperations* vector passed to the Wrapper pointed to invalid memory.

STATUS_OBJECT_NAME_COLLISION – The FSD name specified in the *FsdOperations* vector conflicted with the name of an existing device object or symbolic link

OwSetReadAhead

```
VOID
OwSetReadAhead (
    IN FS_FILE_HANDLE FileHandle,
    IN UCHAR NumberOfPages);
```

Parameters:

FileHandle —The handle provided by the FSD to identify a given file instance.

NumberOfPages —Indicates the number of PAGE_SIZE pages to use as the read-ahead value.

Description:

This routine may be called by the FSD to set the *read ahead* size used by the Cache Manager when reading in pieces of the file. Because this size may have profound effect on the overall performance of the file system, this interface may be used by the FSD to control this characteristic.

The Wrapper will default to 64k. Note that even if a larger read-ahead value is set for a file, the VM system in Windows restricts the paging operations to a maximum size of 64k.

This call may only be made for files and is ignored if called for directories.

This call is considered advisory for the Wrapper. That is, the Wrapper will use the indicated read-ahead size as an indication (or hint) regarding read ahead behavior for the file system, but the actual read-ahead used may vary from that amount to fit the current running state of the system.

Returns:

None.

OwSetWriteBehind

```
VOID
OwSetWriteBehind (
    IN FS_FILE_HANDLE FileHandle,
    IN UCHAR NumberOfPages);
```

Parameters:

FileHandle —The handle provided by the FSD to identify a given file instance.

NumberOfPages —Indicates the maximum number of PAGE_SIZE pages which can be kept in memory and dirty.

Description:

This routine may be called by the FSD to set the *write behind* size used by the Cache Manager when doing asynchronous writeback of the file. This size has profound effect on the caching policy of the FSD and this interface allows the FSD to implement a very *restrictive* caching policy (forcing data back to disk whenever it becomes dirty by setting the *NumberOfPages* to zero) or a more permissive strategy.

By default, the strategy implemented by the Wrapper will be very permissive. Thus, the threshold will be set above the size of the cache – and the entire file may be cached in memory.

While this might sound like an unwise policy, it is important to remember that theWindows Cache Manager implements a dirty data writeback policy *anyway* by writing 25% of the dirty data in the system back every second. Thus, it should only take a few seconds for dirty file data to be written back.

Disabling this write behind feature will severely impair the performance of the system.

This call may only be made for *files* and is ignored if called for directories.

This call is considered advisory for the Wrapper. That is, the Wrapper will use the indicated write-behind size as an indication (or hint) regarding write behind behavior for the file system, but the actual write-behind used may vary from that amount to fit the current running state of the system.

Returns:

None.

FSDK FILE SYSTEM DRIVER INTERFACE

Overview

This section sets forth, in alphabetical order, the interface functions exported by a file system driver that will be called from the FSDK in order to implement the functionality of a file system.

Interface Routines

The routines described in this section are exported by a file system driver that will be called from the FSDK in order to implement the functionality of a file system.

Note that not all functions are required for all file systems. Please refer to the individual function descriptions for additional information on whether or not the given function is appropriate or necessary for your file system driver.

© | OSR Open Systems Resources, Inc.

FS_ACCESS

```
NTSTATUS
(*FS_ACCESS)(
    IN FS_FILE_HANDLE FileHandle,
    IN OUT PACCESS_STATE AccessState);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file or directory handle on which the access check is to be made.

AccessState — This contains all security information about the original caller of this request. The FSD is responsible for updating this data structure.

Description:

If the FSD supports the FS_GET_SECURITY/FS_SET_SECURITY entry points, then standard Windows security policies will be enforced for access to local file objects.

This entry point is intended to be used by FSDs that implement their own additional security policy. It will be called when a file is opened for access.

Note that the Wrapper does not implement *any additional* security policy, other than the above-mentioned optional support for standard Windows local file object security. Specifically, the Wrapper does no enforcement of the "read-only" attribute which may be maintained on a file as it is the responsibility of the FSD to define what portions of the file are "read-only" based upon the state of this bit, as well as to merge the implementation of this very simple security with any other security policy implemented by the FSD.

The ACCESS_STATE data structure is defined in *ntddk.h* as part of the normal DDK environment. However, for the purposes of discussion, we describe this entire structure:

```
typedef struct _ACCESS_STATE {
   LUID OperationID;
   BOOLEAN SecurityEvaluated;
   BOOLEAN GenerateAudit;
   BOOLEAN GenerateOnClose;
   BOOLEAN PrivilegesAllocated;
   ULONG Flags;
   ACCESS_MASK RemainingDesiredAccess;
   ACCESS_MASK PreviouslyGrantedAccess;
   ACCESS_MASK OriginalDesiredAccess;
   SECURITY_SUBJECT_CONTEXT SubjectSecurityContext;
   PSECURITY_DESCRIPTOR SecurityDescriptor;
   PVOID AuxData;
   union {
      INITIAL_PRIVILEGE_SET InitialPrivilegeSet;
      PRIVILEGE_SET PrivilegeSet;
      } Privileges;

   BOOLEAN AuditPrivileges;
   UNICODE_STRING ObjectName;
   UNICODE_STRING ObjectTypeName;

   } ACCESS_STATE, *PACCESS_STATE;
```

Typically, the FSD will only be interested in a subset of these fields:

PreviouslyGrantedAccess – the information about access that has ALREADY been granted to the caller of this routine. The Windows Security system grants certain rights based upon the privileges of the caller, such as traverse right (the ability to traverse through a directory as part of opening a subdirectory or file.)

The *OriginalDesiredAccess* field contains the original access rights requested by the caller.

The *RemainingDesiredAccess* field describes the access rights that have not yet been granted to the caller. The FSD uses this field to determine if access can be granted and if it can, the PreviouslyGrantedAccess and RemainingDesiredAccess fields are updated accordingly.

An ACCESS_MASK is divided into three distinct sections:

- GENERIC rights
- STANDARD rights
- SPECIFIC rights

There are four types of generic rights:

- GENERIC_READ – the ability to read data/attributes of the object
- GENERIC_WRITE – the ability to modify data/attributes of the object
- GENERIC_EXECUTE – the ability to “execute” the object
- GENERIC_ALL – all rights available for the given object

In addition, all Windows objects have a set of standard rights:

- DELETE – the ability to delete the object or something contained within the object
- READ_CONTROL – the ability to read control information about the object

- WRITE_DAC – the ability to modify the discretionary access controls on the object
- WRITE_OWNER – the ability to modify the "owner" field of the object
- SYNCHRONIZE – the ability to use the object for synchronization

In addition, any Windows object may have object-specific rights. For FILE_OBJECTs the specific rights are:

- FILE_READ_DATA - read data from the file
- FILE_LIST_DIRECTORY - list the contents of the directory
- FILE_WRITE_DATA - write data anywhere within the file
- FILE_ADD_FILE - add a new file to a directory
- FILE_APPEND_DATA - write data to the END of the file
- FILE_ADD_SUBDIRECTORY - create a subdirectory in the current directory
- FILE_READ_EA - read an EA from a file or directory
- FILE_WRITE_EA - change the EA for a file or directory
- FILE_EXECUTE - execute the file
- FILE_TRAVERSE - "cd" through the directory
- FILE_DELETE_CHILD - delete a subdirectory of a directory
- FILE_READ_ATTRIBUTES - read the attribute information for the file
- FILE_WRITE_ATTRIBUTES - modify the attribute information for the file

Windows relies upon mappings to convert the generic object rights into a set of standard and specific rights. The creator of the object defines this mapping. The I/O Manager does this for FILE_OBJECTs. There is an exposed interface for retrieving these mappings from the I/O Manager (IoGetFileObjectGenericMapping).

Note that these rights are dependent upon the type of "file" represented by the FILE_OBJECT.

> *Keep in mind that a FILE_OBJECT on Windows can represent any one of three things: a "volume" of a file system, a directory, or a file.*

Returns:

STATUS_SUCCESS – access is granted

STATUS_ACCESS_DENIED – access is refused

STATUS_NOT_IMPLEMENTED – use to indicate success in the case where both FS_ACCESS and FS_GET/SET_SECURITY are supported by the FSD

FS_CHECK_LOCK

```
NTSTATUS
(*FS_CHECK_LOCK)(
    IN FS_FILE_HANDLE FileHandle,
    IN LARGE_INTEGER FileOffset,
    IN ULONG BufferSize,
    IN BOOLEAN Read);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided handle identifying the given file.

FileOffset — The first byte in the range to check.

BufferSize — The number of bytes to lock beginning at *FileOffset*.

Read — Indicates if the requested check should be done for a read operation (would conflict with exclusive locks) or a write operation (would conflict with any locks.)

Description:

This routine is used by the FSDK to verify that a particular I/O operation is consistent with the locking state of the given file. If this routine is not implemented, only locally maintained lock state is enforced.

Returns:

STATUS_SUCCESS – the operation is compatible with existing lock state

STATUS_FILE_LOCK_CONFLICT – the operation conflicts with existing lock state

FS_CLEAR

```
NTSTATUS
(*FS_CLEAR)(
   IN FS_FILE_HANDLE FileHandle,
    IN PLARGE_INTEGER FileOffset,
    IN PLARGE_INTEGER Length);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file handle against which this clear operation is being performed.

FileOffset — The first byte to be cleared.

Length — The number of bytes that should be zeroed.

Description:

This API is used to support sparse files. File systems that support sparse files may be able to eliminate some (or all) of the allocated range (the amount that may be freed is a function of the allocation size, etc.)

Returns:

STATUS_SUCCESS – the file has been successfully cleared.

FS_CONNECT

```
NTSTATUS
(*FS_CONNECT)(
    IN PWSTR RemoteName,
    IN UCHAR ConnectionType,
    OUT PFS_VOL_HANDLE FsdVolumeHandle,
    OUT PFS_FILE_HANDLE RootDirectoryHandle);
```

Status:		Required	Used if Present
	Media File Systems	No	No
	Network File Systems	Yes	Yes
	Pseudo File Systems	No	No

Parameters:

RemoteName — The UNC name to be used for the connection by the FSD. This is a null-terminated wide-character string.

ConnectionType — Indicates the type of connection this connect request represents.

FsdVolumeHandle — The handle to be used by the Wrapper to identify *this* network share when communicating with the FSD.

RootDirectoryHandle — The handle used to identify the *root* directory of this network share.

Description:

Logically, this routine performs the same functions for a network-based file system that the *Mount* operation does for a media-based file system.

Functionally, this call should operate much like FS_MOUNT operates for media file systems.

One issue for a network file system, which is not an issue for a physical-media file system, is how to support "multiple mounts" of the same remote drive. In this case, the Wrapper will handle this based upon the behavior of the underlying FSD. If, as a result of a mount request, the underlying FSD returns an FsdVolume handle which has already been returned the Wrapper will fail the attempt to re-mount that drive.

This allows an FSD to decide if it wishes to support this functionality or not: if it returns a new unique value for *FsdVolumeHandle* and *RootDirectoryHandle* then the mount will be allowed. If it does not return a new unique value for *FsdVolumeHandle* and *RootDirectoryHandle* then the FSDK will indicate to the caller that the mount has failed. It will indicate that the drive is already in use (STATUS_DEVICE_ALREADY_ATTACHED.) This allows a network provider to detect and handle this condition.

If an FSD returns a unique value for *FsdVolumeHandle* the value for *RootDirectoryHandle* should similarly be unique. If they are not, the results of this operation are unpredictable.

> *Note: The Wrapper does not reference count the root directory. The root directory is only released as part of a disconnect or dismount. Thus, whenever the disconnect is handled, the root directory handle will be released regardless of the actual reference count on the root directory handle.*

Also, note that if your FSD returns the same *FsdVolumeHandle* to a connect request only a single *disconnect* request will be received for that operation subsequently.

The *ConnectionType* is one of:

OW_CONNECTION_PRINTER

OW_CONNECTION_DISK

Future values may be defined to support additional types of connections.

The FSD is responsible for validating that the requested connection is acceptable for the given connection type.

Returns:

STATUS_SUCCESS – the connect operation was successful

STATUS_INVALID_DEVICE_REQUEST – the requested connection is incompatible with the remote device

STATUS_OBJECT_NAME_EXISTS – the requested connection utilizes a drive letter that is already in use.

FS_CREATE

```
NTSTATUS
(*FS_CREATE)(
    IN FS_FILE_HANDLE DirectoryHandle,
    IN PWSTR FileName,
    IN PWSTR ShortFileName,
    IN ULONG FileType,
    OUT PFS_FILE_HANDLE NewFileHandle);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

DirectoryHandle — The FSD-provided directory handle against which the object is to be created.

FileName — The name of the new object to be created. This name is *not* permitted to contain an embedded path and must be a new entry in the directory represented by *DirectoryHandle.*

ShortFileName – The alternate (8.3) file name for the file. If null, there is no alternate file name.

FileType — The type of object to create.

NewFileHandle — The FSD-provided file handle for the newly created file or directory.

Description:

This routine may be used to create a new file or directory within the existing directory specified by *DirectoryHandle*. If a file or directory with the specified name already exists, the FSD should return an error.

The File*Type* field indicates what type of object to create. It is one of the following values:

OW_HANDLE_TYPE_FILE

OW_HANDLE_TYPE_DIRECTORY

OW_HANDLE_TYPE_SYMBOLIC_LINK

OW_HANDLE_TYPE_OTHER

If the create operation is successful, the new file or directory handle is returned in *NewFileHandle.* This file handle must be valid for use in subsequent calls to the FSD until the Wrapper calls FS_RELEASE with that handle.

If the FSD has set OW_FS_ATTR_SHORT_NAMES, the Wrapper generates the short file name by utilizing the same name generation algorithm as NTFS. To validate that a name is unique, a series of lookup operations are performed attempting to locate a unique file name. When a new unique file name is found, it will be passed as part of this call. The FSDK serializes during the creation to ensure the short name does not collide with any other short name. The wrapper will not generate a short file name, and the *ShortFileName* parameter will be a null pointer, if the OW_FS_ATTR_SHORT_NAMES attribute is not set.

If the FSD has set OW_FS_ATTR_FSD_SHORT_NAMES, the wrapper will not pass in a short name, but will expect the FSD to generate a valid 8.3 short name. The FSDK must be able to retrieve the valid short name via the FS_GET_NAMES call.

Returns:

STATUS_SUCCESS – the create operation was successful

STATUS_OBJECT_NAME_EXISTS – the specified object already exists and cannot be created.

FS_DELETE

```
NTSTATUS
(*FS_DELETE)(
    IN FS_FILE_HANDLE FileHandle);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file handle that represents the file or directory being deleted.

Description:

This routine may be called to delete an existing file or directory. Note that unlike FS_DELETE2 and FS_DELETE3, this entry point does not require a file or directory handle.

Upon a successful completion of this call, the specified *FileHandle* is no longer valid. That is, the Wrapper will treat the handle as if an implicit call to FS_RELEASE has been made. If the underlying FSD fails a call to delete, the FSDK will subsequently call FS_RELEASE on that file handle. Note that Windows applications will not learn of the rejected delete as a result of failing this call because this operation is done when the file is closed by the application and thus there is no mechanism for reporting this error back to the application.

This routine will not be supported in a future release of the FSDK. New file systems should use **FS_DELETE3** and existing file systems should convert to **FS_DELETE3** as soon as possible. An FSD should only provide FS_DELETE or FS_DELETE2 or FS_DELETE3. If more than one delete routine is provided, the results are undefined.

For more information, see FS_DELETE3.

Returns:

STATUS_SUCCESS – the delete operation was successful

FS_DELETE2

```
NTSTATUS
(*FS_DELETE2)(
    IN FS_FILE_HANDLE FileHandle,
    IN FS_FILE_HANDLE ParentDirectoryHandle,
    IN PWSTR FileName);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file handle that represents the file or directory being deleted.

ParentDirectoryHandle — The FSD-provided file handle that represents the directory containing the file or directory being deleted.

FileName — The name of the instance of the file to be deleted in the specified directory.

Description:

This routine may be called to delete an existing file or directory. Note that unlike FS_DELETE this call also specifies the containing directory and instance name that is being deleted. For file systems supporting hard linked files this information is necessary to allow the FSD to identify the correct *link* to be deleted, not just the correct *file*.

Note that under certain circumstances, the FSDK Wrapper may not be able to provide the name. In such a case, only the *FileHandle* and *ParentDirectoryHandle* are provided, and the *FileName* is **NULL.** In such a case, the *FileHandle* represents a directory.

> ***It is a restriction of the FSDK that a directory cannot support more than a single link to it****. Because of this, the FileHandle and ParentDirectoryHandle are sufficient to provide an unambiguous description of a directory being deleted.*

Upon a successful completion of this call, the specified *FileHandle* is no longer valid. That is, the Wrapper will treat the handle as if an implicit call to FS_RELEASE has been made. If the underlying FSD fails a call to delete, the FSDK will subsequently call FS_RELEASE on that file handle. Note that Windows applications will not learn of the rejected delete as a result of failing this call because this operation is done when the file is closed by the application and thus there is no mechanism for reporting this error back to the application.

Returns:

STATUS_SUCCESS – the delete operation was successful

FS_DELETE3

```
NTSTATUS
(*FS_DELETE3)(
  IN FS_FILE_HANDLE FileHandle,
  IN OUT PFS_DELETE3_EXTENDED_INFO ExtendedInformation);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file handle that represents the file or directory being deleted.

ExtendedInformation — Pointer to FS_DELETE3_EXTENDED_INFO structure containing information on file to be deleted.

Description:

This routine may be called to delete an existing file, link, or directory. Note that unlike FS_DELETE this call also specifies the containing directory, instance name and whether or not to release the handle of the file that is being deleted. For file systems supporting hard linked files this information is necessary to allow the FSD to identify the correct *link* to be deleted, not just the correct *file*.

Note that under certain circumstances, the FSDK Wrapper may not be able to provide the name. In such a case, only the *FileHandle* and *ParentDirectoryHandle* are provided, and the *FileName* is **NULL.** In such a case, the *FileHandle* represents a directory.

> ***It is a restriction of the FSDK that a directory cannot support more than a single link to it****. Because of this, the FileHandle and ParentDirectoryHandle are sufficient to provide an unambiguous description of a directory being deleted.*

Upon a successful completion of this call, the specified *FileHandle* is no longer valid. That is, the Wrapper will treat the handle as if an implicit call to FS_RELEASE has been made. If the underlying FSD fails a call to delete, the FSDK will subsequently call FS_RELEASE on that file handle. Note that Windows applications will not learn of the rejected delete as a result of failing this call because this operation is done when the file is closed by the application and thus there is no mechanism for reporting this error back to the application.

Returns:

STATUS_SUCCESS – the delete operation was successful

FS_DISCONNECT

```
NTSTATUS
(*FS_DISCONNECT)(
    IN FS_VOL_HANDLE FsdVolumeHandle);
```

Status:		Required	Used if Present
	Media File Systems	No	No
	Network File Systems	No	Yes
	Pseudo File Systems	No	No

Parameters:

FsdVolumeHandle — The handle to be used by the Wrapper to identify *this* network share when communicating with the FSD.

Description:

This routine is used to tear down an existing connection. It is the equivalent of an *unmount* in the media file system case.

To disconnect a drive, the Wrapper relies upon the *force* level that was specified by the Network Provider (see OW_FSCTL_DISCONNECT). When the disconnection is not forced, the Wrapper will attempt to flush all outstanding dirty data back to the underlying file system. It will then purge all VM references to those files.

Operational experience indicates that this initial purge does not always succeed but that subsequent VM activity will release all page references and the volume can be disconnected. During disconnection, all remaining file handles will be released back to the underlying FSD. Finally, the root directory handle of the FSD will be released and FS_DISCONNECT will be called to complete the operation.

Once this routine has returned successfully, the Wrapper will no longer use the *FsdVolumeHandle* for access to the underlying file system.

Returns:

STATUS_SUCCESS – the disconnect operation was successful

FS_FLUSH

```
NTSTATUS
(*FS_FLUSH)(
    IN FS_FILE_HANDLE FileHandle);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file or directory handle.

Description:

This routine is used to ensure that any FSD-cached information regarding the given file or directory has been committed to disk.

Note that the Wrapper assumes *no* user-data was cached regarding the file or directory. Rather this is used to ensure that any *metadata* associated with the file has been successfully written.

Note that if an FSD does *no* write-behind caching of FSD metadata, this entry point need not be supported.

Typically, there is little the Wrapper can do to recover from an error returned from this function. Thus, the FSD should only return an error for an unrecoverable failure, such as an I/O error.

Returns:

STATUS_SUCCESS – the operation was successful

FS_FSCTRL

```
NTSTATUS
(*FS_FSCTRL)(
    IN FS_VOL_HANDLE FsdVolumeHandle,
    IN ULONG ControlCode,
    IN PVOID InputBuffer,
    IN ULONG InputBufferSize,
    IN PVOID OutputBuffer,
    IN ULONG OutputBufferSize,
    OUT PULONG BytesWritten,
    OUT PULONG BytesNeeded);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FsdVolumeHandle — The FSD-provided handle representing the volume to be used for this operation.

ControlCode — The identifier for the FSD-specific control operation.

InputBuffer — The input information (if any) for this control operation.

InputBufferSize — The size of the information contained within the input buffer.

OutputBuffer — The location where *output* information for this control operation should be written (if any).

OutputBufferSize — The size of the data contained within the output buffer. This field *can* be zero.

BytesWritten — The number of bytes written to the output buffer.

BytesNeeded — The number of bytes actually required in the output buffer to complete the operation.

Description:

This routine allows FSD-defined control information to be sent between external (application) programs and the FSD. This can be used to implement statistics gathering, modify file system behavior, etc.

The Wrapper provides both the *InputBuffer* and *OutputBuffer* to the FSD. These buffers have been probed for access and should be considered safe to access by the FSD without additional exception handlers or probing required.

If the *InputBuffer* is too small to contain the necessary information, the FSD should return STATUS_INVALID_PARAMETER. If the *OutputBuffer* is too small the FSD should return STATUS_BUFFER_TOO_SMALL and set *BytesNeeded* to indicate the minimum sized buffer required.

Additionally, certain IOCTL values have been defined for use with the Wrapper. They are described in more detail in the Wrapper IOCTL Interface section of this document. The FSD is free to implement and define additional IOCTL values.

Returns:

STATUS_SUCCESS - the I/O Control operation completed successfully

STATUS_INVALID_PARAMETER - the input buffer was too small

STATUS_BUFFER_TOO_SMALL - the output buffer was too small

FS_GET_ATTRIBUTES

```
NTSTATUS
(*FS_GET_ATTRIBUTES)(
    IN FS_FILE_HANDLE FileHandle,
    IN OUT PFS_FILE_ATTRIBUTES FileAtttributes);
```

Status:		Required	Used if Present
	Media File Systems	Yes	Yes
	Network File Systems	Yes	Yes
	Pseudo File Systems	Yes	Yes

Parameters:

FileHandle — The FSD-provided file handle for which the attributes are being retrieved.

FileAttributes — The Wrapper-allocated buffer containing the general file attributes.

Description:

This routine is used to retrieve basic attributes associated with *any* file or directory stored by the FSD:

CreationTime - Indicates when the file was first created

LastAccessTime - Indicates the last time the file was accessed

LastModifiedTime - Indicates the last time the contents of the file were modified

Attributes – Standard Windows file attributes (read-only, hidden, system, archive, etc.)

LastByteWritten - Last byte in the file modified by the application program. It is the ValidDataLength according to Windows and is <= ValidDataLength.

ValidDataLength - Last byte in the file that may be read. It is the "end of file" marker for the file and <= DiskSize

DiskSize - Number of bytes actually allocated, independent of data size

LinkCount - Number of hard links associated with this file

FileID - Unique 64-bit file descriptor

The *FileAttributes* structure (FS_FILE_ATTRIBUTES) will be allocated by the Wrapper and need only be filled in by the FSD. Time values are expressed as the count of 100-nanosecond intervals since January 1, 1601.

The FSD is free to define any or all of these fields to fit its own implementation requirements. Fields that are not supported by the FSD should be set to zero.

The *FileID* field is defined by the FSD and is used by the Wrapper to support "open by file ID" by Windows applications. This ID value must uniquely identify the file, although this value may be reused after the system is rebooted. If this value is zero, the Wrapper will not support opening a file by ID.

Returns:

STATUS_SUCCESS – the operation was successful

FS_GET_EA

```
NTSTATUS
(*FS_GET_EA)(
    IN FS_FILE_HANDLE FileHandle,
    IN OUT PVOID EABuffer,
    IN OUT PULONG EABufferSize);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file or directory handle against which the EA is stored.

EABuffer — This buffer contains the extended attribute set for this file (if any).

EABufferSize — On input, this value is the size of the *EABuffer* allocated by the wrapper. On output, this value indicates the size of the *EABuffer* returned to the Wrapper by the FSD.

Description:

This routine is used to retrieve the extended attribute information associated with the specified file or directory. When the FSDK first calls the FSD with this routine, the *EABuffer* is passed in as a null pointer and the *EABufferSize* is set to zero. If there is extended attribute information associated with the file or directory, then the FSD will return return STATUS_BUFFER_OVERFLOW and the true size of the buffer needed in *EABufferSize*. On the next call, the *EABuffer* will point to the buffer containing a FILE_FULL_EA_INFORMATION structure and *EABufferSize* will indicate the required size, as returned by the first call. The FILE_FULL_EA_INFORMATION structure (from *ntddk.h*) is:

```
typedef struct _FILE_FULL_EA_INFORMATION {
    ULONG NextEntryOffset;
    UCHAR Flags;
    UCHAR EaNameLength;
    USHORT EaValueLength;
    CHAR EaName[1];
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

Because the buffer used to store *EABuffer* is allocated by the Wrapper prior to calling the FSD, it is possible the allocated buffer will be too small. In this case, the FSD should set *EABufferSize* to indicate the minimum acceptable size of the buffer and return STATUS_BUFFER_OVERFLOW.

Returns:

STATUS_SUCCESS – the operation was successful

STATUS_BUFFER_OVERFLOW – the buffer provided was too small. The minimum sized buffer required is specified in *EaBufferSize* upon return from the FSD.

STATUS_NO_EAS_ON_FILE – there are no EAs on the file or directory

FS_GET_NAMES

```
NTSTATUS
(*FS_GET_NAMES)(
    IN FS_FILE_HANDLE FileHandle,
    IN OUT PUNICODE_STRING LongName,
    IN OUT PUNICODE_STRING ShortName);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file handle against which this operation is being performed.

LongName — On input, the long name for this file (if known). On output, a long name for this file.

ShortName — On input, the short name for this file (if known). On output, a short name for this file.

Description:

In order to allow the FSDK to handle long/short name issues more effectively (in earlier versions it accomplishes this by enumerating the directory and matching the names in that fashion) this API allows the FSDK to specify one of the known names for a file and retrieve the corresponding long or short name for the file.

If this API is not supported *and* the FSD has set OW_FS_ATTR_SHORT_NAMES, the FSDK will query the directory.

If the FSD has set OW_FS_ATTR_FSD_SHORT_NAMES, this API *must* be supported.

Returns:

STATUS_SUCCESS – the operation was successful.

STATUS_BUFFER_OVERFLOW – one (or both) of the buffers was too small.

FS_GET_NAMES2

```
NTSTATUS
(*FS_GET_NAMES2)(
    IN FS_FILE_HANDLE FileHandle,
    IN OUT PUNICODE_STRING LongName,
    IN OUT PUNICODE_STRING ShortName,
    IN OUT PFS_GET_NAMES2_EXTENDED_INFO ExtendedInfo);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file handle against which this operation is being performed.

LongName — On input, the long name for this file (if known). On output, a long name for this file.

ShortName — On input, the short name for this file (if known). On output, a short name for this file.

ExtendedInfo — This contains specific information about the type of name being sought in this case.

Description:

In order to allow the FSDK to handle long/short name issues more effectively (in earlier versions it accomplishes this by enumerating the directory and matching the names in that fashion) this API allows the FSDK to specify one of the known names for a file and retrieve the corresponding long or short name for the file.

If this API is not supported *and* the FSD has set OW_FS_ATTR_SHORT_NAMES, the FSDK will query the directory.

If the FSD has set OW_FS_ATTR_FSD_SHORT_NAMES, this API *must* be supported. This API must also be provided if LookupById or LookupByObjectId are supported.

Returns:

STATUS_SUCCESS – the operation was successful.

STATUS_BUFFER_OVERFLOW – one (or both) of the buffers was too small.

FS_GET_SECURITY

```
NTSTATUS
(*FS_GET_SECURITY)(
    IN FS_FILE_HANDLE FileHandle,
    IN OUT PVOID *SecurityDescriptor,
    IN OUT PULONG SecurityDescriptorSize);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file or directory handle on which the security descriptor is to be retrieved.

SecurityDescriptor — This buffer contains the self-relative format security descriptor stored with this file (if any).

SecurityDescriptorSize — This value indicates the size of the *SecurityDescriptor* returned to the Wrapper by the FSD.

Description:

This routine is used to retrieve the Windows-format security information for the given file if one exists. If not, the *SecurityDescriptor* value should be set to **NULL** which indicates that there is no security descriptor associated with the file. Otherwise, the returned *SecurityDescriptor* should point to a buffer containing a valid security descriptor (use **RtlValidateSecurityDescriptor** to validate security descriptors). If the returned *SecurityDescriptor* has no associated DACL then no access will be granted to the file, except for specific operations:

- Any process with **SeTakeOwnershipPrivilege** may take ownership of the file, which will create a new security descriptor encoding the new owner of the file.
- Any process associated with the **SID** that owns the existing file (if any) may change the security descriptor for the file.

Support for FS_GET_SECURITY is optional. However, the FSD must support both FS_GET_SECURITY and FS_SET_SECURITY if it supports either interface.

This operation requests that the FSD return a self-relative formatted security descriptor for the file or directory if one exists. If it does exist, the FSD should allocate a buffer sufficiently large to contain the security descriptor by calling **OwAllocateSDBuffer** and using that buffer to store the security descriptor.

To summarize:

- Upon entry to the FSD, the value contained in *SecurityDescriptor* will be **NULL**.
- If the file or directory does not have a security descriptor, the FSD should return an error status (all error status are treated as indications there is no security descriptor). The FSDK will not track a security descriptor for this file (although an external application can set one, if supported by the FSD).
- If the file or directory has an empty security descriptor, the FSD should return a valid security descriptor that does not contain any of the four fields (owner SID, group SID, DACL, or SACL). The buffer should be allocated using **OwAllocateSDBuffer** and the size returned should be the value returned by **RtlLengthSecurityDescriptor** (or equivalent calculation).
- If the file or directory has a security descriptor, the FSD should allocate a buffer of sufficient size using **OwAllocateSDBuffer** and set the value of *SecurityDescriptor* to the address of the buffer. The contents of the security descriptor buffer must be a valid security descriptor.

Each file or directory can have its own distinct security descriptor.

The security information contained in the buffer is generally not useful to the FSD. This interface provides a simple mechanism that allows a file system to support standard Windows security policies by simply storing and retrieving security information on a per file (or directory) basis. The wrapper performs all security operations for the FSD if the FS_GET_SECURITY and FS_SET_SECURITY interfaces are implemented by the FSD.

Additional security policies may be implemented using the FS_ACCESS interface.

The FSDK will deallocate the buffer returned by the FSD for the security descriptor.

Returns:

STATUS_SUCCESS – the operation was successful.

STATUS_BUFFER_OVERFLOW – the *SecurityDescriptor* buffer was too small. *SecurityDescriptorSize* is set to the minimum size.

FS_GET_UNC_VOLUME_ATTRIBUTES

```
NTSTATUS
(*FS_GET_UNC_VOLUME_ATTRIBUTES)(
    IN FS_FILE_HANDLE FileHandle,
    IN PFS_VOL_ATTRIBUTES VolumeAtttributes,
    IN OUT PULONG VolumeAttributesBufferSize);
```

Status:		Required	Used if Present
	Media File Systems	No	No
	Network File Systems	No	Yes
	Pseudo File Systems	No	No

Parameters:

FileHandle — An FSD-provided handle representing a file or directory on the UNC volume of interest.

VolumeAttributes – The Wrapper-allocated structure being filled in by the FSD.

VolumeAttributesBufferSize – On input, the size of the *VolumeAttributes* buffer. On output, the number of bytes of data in the buffer.

Description:

This routine is used by the FSDK to retrieve volume attributes for a given UNC volume. Because such a volume does not have *connection state* associated with it, there is no corresponding volume information to provide. If the FSD does not provide this information, the FSDK will return synthetically generated information to the application program.

Returns:

STATUS_SUCCESS – the buffer contains the complete information

STATUS_BUFFER_OVERFLOW – the buffer was not large enough for the information. The *VolumeAttributesBufferSize* contains the minimum buffer size required.

FS_GET_VOLUME_ATTRIBUTES

```
NTSTATUS
(*FS_GET_VOLUME_ATTRIBUTES)(
    IN FS_VOL_HANDLE FsdVolumeHandle,
    IN OUT PFS_VOL_ATTRIBUTES VolumeAtttributesBuffer,
    IN OUT PULONG VolumeAttributesBufferSize);
```

Status:		Required	Used if Present
	Media File Systems	Yes	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	Yes	Yes

Parameters:

FsdVolumeHandle — The FSD-provided handle representing the volume to be used for this operation.

VolumeAttributesBuffer — The Wrapper-allocated buffer where the FSD will store the volume general attributes.

VolumeAttributesBufferSize — On input, the size of the *VolumeAttributesBuffer*. On output, the number of bytes of data in the buffer

Description:

This routine may be used to retrieve one of two possible per-volume attribute structures: Volume Attributes (FS_VOL_ATTRIBUTES) or Extended Volume Attributes (FS_EXTENDED_VOL_ATTRIBUTES).

Volume Attributes

This first structure is compatible with the FSDK V1.0 structure (see the complete definition of the structure FS_VOL_ATTRIBUTES later in this document.) The critical fields are:

CreationTime - when the volume was first initialized (this is the time when the volume was formatted)

SerialNumber - a unique number identifying this file system instance

LabelLength - the volume label length in bytes

Unused – this field is not used and must be zero for compatibility

VolumeSize – the number of allocation units available on the volume

FreeSpace - the number of unallocated allocation units on the volume

BytesPerAllocationUnit - the number of bytes per allocation unit

DeviceType – standard Windows device type information.

DeviceCharacteristics – standard Windows device characteristics

VolumeLabel - the volume label

The Wrapper allocates the *VolumeAttributesBuffer*. The size of the passed-in buffer is *VolumeAttributesBufferSize.* Should it not be large enough to contain the volume attributes being returned by the FSD, the *VolumeAttributesBufferSize* parameter should be set to the minimum size required and STATUS_BUFFER_OVERFLOW should be returned to the Wrapper.

All times must be returned in Windows standard format – the number of 100 nanosecond intervals since January 1, 1601.

The Wrapper considers the values provided in this data structure informational in nature and hence it is the responsibility of the *FSD* to define the precise semantics of these calls. The one exception to this is the *DeviceCharacteristics* field that must conform to the Windows semantics for Device Characteristics, namely one or more of the following.

FILE_CASE_SENSITIVE_SEARCH

FILE_CASE_PRESERVED_NAMES

FILE_UNICODE_ON_DISK

FILE_PERSISTENT_ACLS

FILE_FILE_COMPRESSION

FILE_VOLUME_IS_COMPRESSED

These values are from winnt.h and describe basic characteristics of a given file system device.

If any particular field is not supported by the FSD, it should be set to zero (or its equivalent for the data type, such as the “null string” for the VolumeLabel).

Extended Volume Attributes

In order to allow for additional information to be passed as a result of this call, a newer structure, the *Extended Volume Attributes* structure (see the complete definition of the structure FS_EXTENDED_VOL_ATTRIBUTES later in this document) has been defined in a fashion that is upwards compatible with the previous implementation of volume attributes. An FSD may return this extended volume attributes structure; that this extended volume attributes information is being returned is indicated using a new *flags* field that overlaps with the previous *Unused* field. This *Flags* field must have the OW_VOL_ATTR_FLAG_EXTENDED bit set. The FSDK uses this bit to determine if the new version (extended volume attributes) or the old version is present in the return buffer.

Returns:

STATUS_SUCCESS – the operation was successful.

STATUS_BUFFER_OVERFLOW – the *VolumeAttributesBuffer* is too small. The minimum sized buffer required is returned in *VolumeAttributesBufferSize*.

FS_IOCTL

```
NTSTATUS
(*FS_IOCTL)(
    IN FS_FILE_HANDLE FileHandle,
    IN ULONG ControlCode,
    IN PVOID InputBuffer,
    IN ULONG InputBufferSize,
    IN PVOID OutputBuffer,
    IN ULONG OutputBufferSize,
    OUT PULONG BytesWritten,
    OUT PULONG BytesNeeded);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided handle representing the file to be used for this operation.

ControlCode — The identifier for the FSD-specific control operation.

InputBuffer — The input information (if any) for this control operation.

InputBufferSize — The size of the information contained within the input buffer.

OutputBuffer — The location where *output* information for this control operation should be written (if any). This field may be NULL if the *OutputBufferSize* parameter is zero.

OutputBufferSize — The size of the data contained within the output buffer. This field may be zero.

BytesWritten — The number of bytes written to the output buffer.

BytesNeeded — The number of bytes actually required in the output buffer to complete the operation.

Description:

This routine allows FSD-defined control information to be sent between external (application) programs and the FSD. This can be used to implement statistics gathering, modify file system behavior, etc. The key difference between this routine and the FS_FSCTRL entry point is that this routine is targeted towards a particular *file* while the other entry point is targeted towards the entire file system.

The Wrapper will provide both the *InputBuffer* and *OutputBuffer* to the FSD, indicating the size of each. If the input buffer size is incorrect, the FSD should return STATUS_INVALID_PARAMETER. If the *output* buffer size is incorrect, the FSD should return STATUS_BUFFER_TOO_SMALL, *BytesWritten* should be zero, and *BytesNeeded* should indicate the minimum size of the output buffer to complete the operation.

© | OSR Open Systems Resources, Inc.

Note that this call is not identical to the FS_FSCTL entry point in that the target of the call is different. For this call, the target is the specified file, while in the previous case, the target is the FSD itself. Thus, this may be used to implement per-file mechanisms for use with a particular FSD.

For media file systems, the Control Code must use a device type of FILE_DEVICE_FILE_SYSTEM. All other control code values will be passed to the underlying physical media device object bu the wrapper. This behavior is compatible with the existing implementation of the Windows physical file system.

Returns:

STATUS_SUCCESS – the operation was successful.

STATUS_INVALID_DEVICE_REQUEST – this device does not support the requested operation.

STATUS_INVALID_PARAMETER – the input buffer was not the correct size.

STATUS_BUFFER_TOO_SMALL – the output buffer was too small for the given operation.

FS_LINK

```
NTSTATUS
(*FS_LINK)(
    IN FS_FILE_HANDLE DirectoryHandle,
    IN FS_FILE_HANDLE FileHandle,
    IN PWSTR LinkName,
    IN PWSTR ShortLinkName);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

DirectoryHandle — The FSD-provided directory handle that contains the directory where the new link is to be created.

FileHandle — The FSD-provided file handle representing an existing file.

LinkName — The name of the new link to be created. This name is *not* permitted to contain an embedded path and must *not* be an existing entry in the directory represented by *DirectoryHandle.*

ShortLinkName — The alternate (8.3) name of the new link to be created. If null, there is no alternate name.

Description:

This routine is used to create a *hard link* to an existing file. It is the responsibility of the FSD to manage the link count information associated with the given file and to ensure that the actual file is not deleted until the last link to that file is deleted.

It is an error to create a hard link to a directory.

Note that the Win 32 API CreateHardLink allows for the creation of hard links.

Returns:

STATUS_SUCCESS – the create operation was successful

FS_LOCK

```
NTSTATUS
(*FS_LOCK)(
    IN FS_FILE_HANDLE FileHandle,
    IN LARGE_INTEGER Offset,
    IN LARGE_INTEGER Length,
    IN BOOLEAN Exclusive,
    IN BOOLEAN Wait,
    OUT PFS_LOCK_HANDLE LockHandle);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided handle identifying the given file.

Offset — The first byte in the locked range.

Length — The number of bytes to lock beginning at *Offset*.

Exclusive — Indicates if the requested lock is for *exclusive* (write) or *shared* (read) access to the file.

Wait — Indicates if the call should *block* waiting for the lock to be acquired.

LockHandle — The FSD-defined identifier for this particular lock.

Description:

The Wrapper always provides the standard Windows byte range locking package. However, some file systems may require notification of such lock requests in order to coordinate with access external to the local host computer and of which the Wrapper would otherwise not be aware.

Each lock request passed to the FSD is distinct. The Wrapper will not request that the FSD acquire locks incompatible with previously granted locks, but may request that the FSD grant locks compatible with previously granted locks. Each such locking request should either be denied, returning an appropriate error status, or granted. For each such unique lock handle that is returned, a subsequent unlock call (see FS_UNLOCK) will be made by the Wrapper to the FSD.

If this routine is not implemented by the FSD, all byte range locking will be done without support from the FSD.

If the *Wait* parameter is TRUE, the calling thread should block until the lock can be granted. If the *Wait* parameter is FALSE and it is possible for the FSD to grant the lock without waiting, it should do so. Otherwise, the FSD should return STATUS_LOCK_NOT_GRANTED.

The *LockHandle* parameter is created by the FSD to identify this particular lock. The Wrapper uses it in a subsequent call to release the lock. Please note that the FSD must return an identical *LockHandle* when attempting to lock the same *Offset* and *Length* of a specific file identified by an identical *FileHandle.*

Returns:

STATUS_SUCCESS – the lock was granted

STATUS_FILE_LOCK_CONFLICT – the lock could not be granted due to a lock conflict

STATUS_LOCK_NOT_GRANTED – the lock could not be granted for unspecified reasons.

© | OSR Open Systems Resources, Inc.

FS_LOOKUP

```
NTSTATUS
(*FS_LOOKUP)(
    IN FS_FILE_HANDLE DirectoryHandle,
    IN PWSTR FileName,
    IN BOOLEAN CaseInsensitive,
    OUT PFS_FILE_HANDLE FileHandle,
    OUT PULONG Type);
```

Status:		Required	Used if Present
	Media File Systems	Yes	Yes
	Network File Systems	Yes	Yes
	Pseudo File Systems	Yes	Yes

Parameters:

DirectoryHandle — The FSD-provided directory handle representing the base directory from which this file itself is being opened.

FileName — A null-terminated wide character string describing the file or directory to be located. The file name will contain neither wildcards nor a partial pathname.

CaseInsensitive — Indicates if the *case* of the name should be ignored when looking for the file name within the directory.

FileHandle — The FSD-provided handle to the returned file or directory.

Type — The type of file that has been opened.

Description:

The Wrapper calls this entry to obtain a handle for subsequent reference to a file or directory. This file handle will be used in subsequent operations from the Wrapper to the FSD.

A lookup is performed for a particular entry relative to a given directory. The *DirectoryHandle* indicates the directory to be searched and the *FileName* indicates the name to be located.

The *FileName* parameter may not include a path separator character ('\') and hence may not be a "path name". The *CaseInsensitive* parameter is used to indicate if name comparisons should depend upon the case of the names being identical. The Wrapper rejects all names containing Win32 wildcard characters prior to passing such names to the FSD, so the FSD need not check for wildcard characters (such as "*").

If the given *FileName* exists within the directory represented by *DirectoryHandle,* the FSD will create a *FileHandle* representing that file or directory for use in subsequent calls to and from the file system.

If the given *FileName* does not exist within the directory represented by *DirectoryHandle,* the FSD will return STATUS_OBJECT_NAME_NOT_FOUND to the Wrapper.

If more than one file or directory with the given *FileName* exists within the directory represented by *DirectoryHandle*, the FSD may return *any* one of the files or directories within the directory. For example, this would be the case for a file created from a case sensitive system such as POSIX and then accessed from a case insensitive system such as Win32.

If the FSD returns STATUS_SUCCESS (indicating the operation was successful) the *FileHandle* returned must be valid. The Wrapper will then use it for subsequent calls to the FSD. The *FileHandle* remains valid until a subsequent call to the FS_RELEASE entry point for the file system.

The *Type* field indicates what type of file or directory has been found. It is one of the following values:

OW_HANDLE_TYPE_FILE

OW_HANDLE_TYPE_DIRECTORY

OW_HANDLE_TYPE_SYMBOLIC_LINK

OW_HANDLE_TYPE_OTHER

This allows the Wrapper to restrict operations (such as enumerating the contents of a directory) to the appropriate type of object (a directory).

Other types may be defined as necessary in the future.

This interface allows the supporting of streams using the standard NTFS model. A *stream* is uniquely identified by the suffix "::" followed by a name (e.g., $DATA) representing the specific stream. The default stream is $DATA and hence a file without an explicit stream name will be considered to be equivalent to a stream with the name "::$DATA" appended to it.

What constitutes a valid file name may be arbitrarily restricted by the FSD. The Wrapper does not explicitly enforce any particular name requirements.

The Wrapper implements an internal reference counting mechanism so that it is not necessary for the FSD to do so. Thus, the Wrapper may call the FS_LOOKUP interface many times but will only call the FS_RELEASE interface once, when the last reference to the file or directory is released by the Wrapper.

Returns:

STATUS_SUCCESS – the lookup was successful

STATUS_OBJECT_NAME_NOT_FOUND – the named file or directory could not be located and the operation failed.

FS_LOOKUP_BY_ID

```
NTSTATUS
(*FS_LOOKUP_BY_ID)(
    IN FS_FILE_HANDLE DirectoryHandle,
    IN LARGE_INTEGER FileId,
    OUT PFS_FILE_HANDLE FileHandle,
    OUT PULONG Type);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

DirectoryHandle — The FSD-provided directory handle representing the base directory from which this file itself is being opened.

FileId — A large integer FSD-specific ID used to identify the given file.

FileHandle — The FSD-provided handle to the returned file or directory.

Type — The type of file that has been opened.

Description:

The Wrapper will call this entry point in order to resolve an “open by id” call from the FSD.

Operationally, this call is identical to FS_LOOKUP except that a File ID is used in place of a file name.

The construction of a file ID is FSD-specific. The actual file ID to use is retrieved from a previously opened file object (q.v., FS_GET_ATTRIBUTES). Typically, this operation is used by file servers that implement stateless file system protocols and hence embed the file id in an opaque network file handle.

Note that support for this interface is currently not required for Windows. This functionality is currently used by the “Services For Macintosh” and “Services For Unix” support. However, this product cannot be used with any file systems other than NTFS and CDFS.

If this entry point is supported, then the FSD should also support FS_GET_NAMES2, FS_DELETE3, and FS_RENAME2.

Returns:

STATUS_SUCCESS – the lookup was successful

STATUS_OBJECT_NAME_NOT_FOUND – the lookup failed

FS_LOOKUP_BY_OBJECT_ID

```
NTSTATUS
(*FS_LOOKUP_BY_OBJECT_ID)(
    IN FS_FILE_HANDLE DirectoryHandle,
    IN GUID *Guid,
    OUT PFS_FILE_HANDLE FileHandle,
    OUT PULONG Type);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

DirectoryHandle — The FSD-provided directory handle representing the base directory from which this file itself is being opened.

Guid — A 16 byte data element used within the Windows system as a "globally unique identifier".

FileHandle — The FSD-provided handle to the returned file or directory.

Type — The type of file that has been opened.

Description:

The Wrapper will call this entry point in order to resolve an “open by object id” call from the FSD.

Operationally, this call is identical to FS_LOOKUP except that a Guid is used in place of a file name.

A globally unique identifier may be assigned by the file system. If the file system does not assign a GUID but wishes to support per-file GUIDs, it must allow applications (via the appropriate FSCTL operations) to set the GUID value for the given file. The GUID of the given file is retrieved using the **FSCTL_GET_OBJECT_ID** operation.

Note that support for this interface is currently not required for Windows. This functionality is currently used by several OS components, such as the Single Instance Store (SIS).

If this entry point is supported, then the FSD should also support FS_GET_NAMES2, FS_DELETE3, and FS_RENAME2.

Returns:

STATUS_SUCCESS – the lookup was successful

STATUS_OBJECT_NAME_NOT_FOUND – the lookup failed

FS_LOOKUP_PATH

```
NTSTATUS
(*FS_LOOKUP_PATH)(
    IN FS_FILE_HANDLE DirectoryHandle,
    IN PWSTR PathName,
    IN BOOLEAN CaseInsensitive,
    IN PIO_SECURITY_CONTEXT CallerSecurityContext,
    OUT PFS_FILE_HANDLE ParentDirectoryHandle,
    OUT PFS_FILE_HANDLE FileHandle,
    OUT PULONG Type);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

DirectoryHandle — The FSD-provided directory handle representing the base directory from which this file itself is being opened.

PathName — A null-terminated wide character string describing the partially qualified path of the file or directory to be looked up.

CaseInsensitive — Indicates if the *case* of the name should be ignored when looking for the file name within the directory.

CallerSecurityContext — The security context of the caller attempting to perform the lookup operation.

ParentDirectoryHandle — The FSD-provided handle to the directory in which the file or directory represented by *FileHandle* is located.

FileHandle — The FSD-provided handle to the returned file or directory.

Type — The type of file that has been opened.

Description:

The Wrapper calls this entry to obtain a handle for subsequent reference to a file or directory. This file handle will be used in subsequent operations from the Wrapper to the FSD.

Returns:

STATUS_SUCCESS – the lookup was successful

STATUS_ACCESS_DENIED – the caller did not have sufficient privilege level to access the file

STATUS_OBJECT_PATH_NOT_FOUND – a component of the path name was not valid

STATUS_OBJECT_NAME_NOT_FOUND – the last component of the path name was not valid

If the FSD returns STATUS_OBJECT_NAME_NOT_FOUND the ParentDirectoryHandle must be a valid handle. This allows for the creation of new files, for example, where the directory does exist, but the new file does not.

FS_MOUNT

```
NTSTATUS
(*FS_MOUNT)(
    IN PVOID MediaHandle,
    IN OUT PFS_VOL_HANDLE FsdVolumeHandle,
    OUT PFS_FILE_HANDLE RootDirectoryHandle);
```

Status:		Required	Used if Present
	Media File Systems	Yes	Yes
	Network File Systems	No	No
	Pseudo File Systems	Yes	Yes

Parameters:

MediaHandle — An opaque handle provided by the Wrapper to the FSD to indicate what media is being mounted.

FsdVolumeHandle — The handle to be used by the Wrapper when sending messages to the FSD. This context handle is provided *to* the Wrapper *by* the FSD.

RootDirectoryHandle — The file handle representing the *root directory* of the given file system media device.

Description:

The *mount* entry point is used by the Wrapper to provide the FSD with an opportunity to examine the proffered volume and determine if the disk signature indicates the given volume belongs to this FSD.

For physical media file systems, the *MediaHandle* is used in the lower-edge routines to indicate the particular media volume being managed. For a pseudo file system, the *MediaHandle* represents the opaque handle value provided as part of a "pseudo mount" operation.

An FSD is responsible for examining the volume specified by the *MediaHandle* and determining if the volume should be mounted. The exact mechanism for doing this is the responsibility of the FSD, but is normally done via some "disk signature" examination method.

Returns:

STATUS_UNRECOGNIZED_VOLUME – the volume specified is not mountable by this file system.

STATUS_SUCCESS – the volume has been successfully mounted

FS_QUERY_PATH

```
NTSTATUS
(*FS_QUERY_PATH)(
    IN PWSTR PathName,
    OUT PULONG PathNameAccepted,
    OUT PFS_FILE_HANDLE DirectoryHandle);
```

Status:		Required	Used if Present
	Media File Systems	No	No
	Network File Systems	Yes	Yes
	Pseudo File Systems	No	No

Parameters:

PathName — The null-terminated UNC name to be analyzed by the FSD.

PathNameAccepted — The number of bytes (*not* wide-characters) recognized by the FSD as being the "prefix" of a network share.

DirectoryHandle — The FSD provided handle used to represent the directory recognized by the FSD.

Description:

This routine is called by the Wrapper to determine if the network FSD recognizes a given UNC name. If it is recognized, the FSD should return STATUS_SUCCESS and indicate the portion of the name that represents a valid network share, *not* including the final separator character ('\') of the recognized portion of the name. If the UNC name is not recognized, the FSD should return STATUS_OBJECT_NAME_NOT_FOUND.

The *DirectoryHandle* will be used in subsequent calls to access the network share via its UNC name. This *DirectoryHandle* must remain valid until a subsequent call to FS_RELEASE from the Wrapper.

The purpose of this call is to determine the "ownership" of a particular network name. Thus, various Windows OS components query each registered UNC provider in order to determine where to send subsequent requests for that particular name. However, rather than restrict it only to a given file, the interface is constructed so that the FSD indicates the "prefix" portion of the name. Any future names that match the same prefix can be routed immediately to the correct file system, thus enhancing the overall performance and responsiveness of the system.

Returns:

STATUS_SUCCESS – the prefix was recognized

STATUS_OBJECT_NAME_NOT_FOUND – the prefix was not recognized

FS_READ

```
NTSTATUS
(*FS_READ)(
    IN FS_FILE_HANDLE FileHandle,
    IN LARGE_INTEGER FileOffset,
    IN PMDL Buffer,
    IN ULONG BufferSize,
    OUT PULONG BytesRead);
```

Status:		Required	Used if Present
	Media File Systems	Yes	Yes
	Network File Systems	Yes	Yes
	Pseudo File Systems	Yes	Yes

Parameters:

FileHandle — The FSD-provided file handle against which this read operation is being performed.

FileOffset — The byte-range offset into the file where the I/O begins.

Buffer — A pointer to the MDL representing the pages where data should be read from the disk into memory.

BufferSize — The number of bytes to be read into the region described by *Buffer*.

BytesRead – This indicates the number of bytes read by the FSD.

Description:

This entry point is used by the Wrapper to perform actual I/O. Because the *normal* model for the Wrapper is to cache data, this entry point will typically be called to handle a non-cached I/O request. Typically this is a request to satisfy a *page fault* while accessing the file cache.

To facilitate the development of an FSD, the FSD may assume that the I/O operations adhere to the following guidelines:

For physical media file systems, the *FileOffset* will be aligned on a sector-sized boundary. For all other file systems, the alignment will be 512 bytes. In fact, we expect that this will be page aligned, but we use the weaker requirement to support non-cached I/O directly from utilities and user applications, although such are rare.

If this entry point returns STATUS_SUCCESS, *BytesRead* will indicate the actual number of bytes read by the FSD.

Upon completion of the Read operation by the FSD, *BytesRead* must be less than or equal to *BufferSize*.

The Wrapper is responsible for ensuring that any data beyond the current end-of-file is eliminated prior to presentation to a user application. Thus, *BytesRead* does not indicate that the size of the valid data portion of the file has been extended.

The principal goal of this design is to ensure that I/O to the FSD falls on naturally aligned boundaries. Note that typically, I/O is done in PAGE_SIZE units. Odd sizes are typically only used for "boundary conditions" such as the last component (less than PAGE_SIZE) of a file.

For file systems that implement sparse storage, this operation expects that any unallocated space will be returned as zero-filled sections of the file and that the *BytesRead* parameter reports the actual number of bytes that were transferred into the output buffer.

Returns:

STATUS_SUCCESS – the operation was successful.

STATUS_WRONG_VOLUME – the volume has changed and the read cannot be completed with the currently available volume (verify is required.)

STATUS_NO_MEDIA_IN_DEVICE – the volume is not available because the media has been removed.

STATUS_INSUFFICIENT_RESOURCES – the FSD was unable to complete the I/O operation due to insufficient memory resources

STATUS_DISK_CORRUPT – the file system structure on the disk is damaged.

FS_READ_DIRECTORY

```
NTSTATUS
(*FS_READ_DIRECTORY)(
    IN FS_FILE_HANDLE DirectoryHandle,
    IN OUT PULONG DirectoryOffset,
    IN PFS_DIRECTORY_ENTRY OutputBuffer,
    IN ULONG OutputBufferSize);
```

Status:		Required	Used if Present
	Media File Systems	Yes	Yes
	Network File Systems	Yes	Yes
	Pseudo File Systems	Yes	Yes

Parameters:

DirectoryHandle —The FSD-provided directory handle for the directory being read.

DirectoryOffset —This provides the *offset* context being used to enumerate the directory contents.

OutputBuffer —This buffer is provided by the Wrapper and may contain one or more directory entries.

OutputBufferSize —This indicates the size of the buffer provided by the Wrapper.

Description:

This function is obsolete and has been replaced with the FS_READ_DIRECTORY3 interface. It is documented here only for compatibility with existing file system drivers.

The FS_READ_DIRECTORY and FS_READ_DIRECTORY3 functions work identically except that the format of the structure they use is slightly different. For FS_READ_DIRECTORY the structure is:

```
typedef struct {
    ULONG EntrySize;
    ULONG LastDirectoryEntry;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastModifiedTime;
    LARGE_INTEGER Unused;
    LARGE_INTEGER ValidDataLength;
    LARGE_INTEGER DiskLength;
    ULONG Attributes;
    ULONG NameSize;
    UCHAR ShortNameSize;
    WCHAR ShortName[12];
    WCHAR Name[1]; // and for the balance of this entry
} FS_DIRECTORY_ENTRY, *PFS_DIRECTORY_ENTRY;
```

Please refer to FS_READ_DIRECTORY3 for details about the manner in which this call is implemented.

Returns:

STATUS_SUCCESS – the operation was successful and the buffer contents are valid

STATUS_NO_SUCH_FILE – the directory is empty

STATUS_NO_MORE_FILES – there are no additional entries to enumerate, the buffer contents are not valid

© | OSR Open Systems Resources, Inc.

FS_READ_DIRECTORY2

```
NTSTATUS
(*FS_READ_DIRECTORY2)(
    IN FS_FILE_HANDLE DirectoryHandle,
    IN OUT PULONG DirectoryOffset,
    IN PFS_DIRECTORY_ENTRY2 OutputBuffer,
    IN ULONG OutputBufferSize);
```

Status:		Required	Used if Present
	Media File Systems	Yes	Yes
	Network File Systems	Yes	Yes
	Pseudo File Systems	Yes	Yes

Parameters:

DirectoryHandle —The FSD-provided directory handle for the directory being read.

DirectoryOffset —This provides the *offset* context being used to enumerate the directory contents.

OutputBuffer —This buffer is provided by the Wrapper and may contain one or more directory entries.

OutputBufferSize —This indicates the size of the buffer provided by the Wrapper.

Description:

Note that this call has superseded FS_READ_DIRECTORY. The two functions work identically (and are presently implemented using the same functional logic) but utilize slightly different structures. FS_READ_DIRECTORY2 uses the FS_DIRECTORY_ENTRY2 structure, which is compatible with the Windows directory structure data and thus simplifies the implementation of certain types of “pseudo” file systems.

The Wrapper manages the interface with the Windows directory enumeration mechanism, including packing user buffers, etc. The interface between the Wrapper and the FSD is a simpler iterative interface.

The Wrapper does this by making a series of calls into the FSD to retrieve file system information. The *DirectoryOffset* value is used to communicate information between successive calls to this interface routine.

That is, the first time this call will be made, the Wrapper will set this value to zero. Thus, the FSD should treat a *DirectoryOffset* value of zero as indicating an enumeration of the directory beginning with the first entry in the directory.

The FSD then uses the *OutputBuffer* as storage for its response to this request. The FSD must pack at least one entry into this buffer, if one is available, but may pack as many entries as can fit into the buffer and then returns STATUS_SUCCESS to the Wrapper to indicate there is some valid data within the buffer.

A directory entry has a fixed format:

```
typedef struct {
    ULONG EntrySize;
    ULONG LastDirectoryEntry;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastModifiedTime;
    LARGE_INTEGER Unused;
    LARGE_INTEGER ValidDataLength;
    LARGE_INTEGER DiskLength;
    ULONG Attributes;
    ULONG NameSize;
    ULONG EaSize;
    UCHAR ShortNameSize;
    WCHAR ShortName[12];
    WCHAR Name[1]; // and for the balance of this entry
} FS_DIRECTORY_ENTRY2, *PFS_DIRECTORY_ENTRY2;
```

Note that this structure declaration includes the EaSize field which makes this compatible with the standard Windows directory entry.

The Wrapper parses the buffer by examining each entry in turn, using a simple algorithm:

1. If *NameSize* is zero, the entry is skipped
2. Otherwise, the entry is processed as a valid directory entry
3. If *LastDirectoryEntry* is non-zero, the next entry in the buffer exists and will be processed

Thus, the buffer is parsed until the Wrapper has properly processed all valid entries within the buffer.

An FSD need not return the entire directory contents in a single buffer. To handle this case, the FSD returns the *DirectoryOffset* value. If, after processing the current buffer, this value is non-zero, the Wrapper will call the FSD again, asking for additional directory entries to process. This process continues until one of three things occurs:

1. The FSD returns a *DirectoryOffset* value of zero and a return value of STATUS_SUCCESS. This indicates that the current buffer has valid contents, but that there is no additional information available regarding this particular directory;
2. The FSD returns a status code of STATUS_NO_MORE_FILES or STATUS_NO_SUCH_FILE, in which case enumeration terminates immediately and the returned buffer is assumed to contain no valid data;
3. The FSD returns any error status, in which case it is treated as an I/O error and the user request to enumerate the directory will fail with the given error code. This should be reserved for serious FSD errors, such as a media or network failure.

If a directory is empty of all contents, the FSD should return STATUS_NO_SUCH_FILE.

Windows assumes that every directory contains two entries: “.” and “..”. An FSD need not return these entries to the Wrapper. If an FSD does return these entries to the wrapper the first entry returned must be “.” and the second entry “..”. Otherwise, if the first entry is not “.” and the second entry “..” the Wrapper will create a “.” and “..” entry for the directory.

Note: *At the risk of confusing things further, there is one case when the Wrapper does not create these entries for an empty directory. If the root directory of a volume is totally empty, in that particular case the semantics of the existing file systems (NTFS and FAT) is to return STATUS_NO_SUCH_FILE to the caller. These semantics are duplicated by the Wrapper.*

Returns:

STATUS_SUCCESS – the operation was successful and the buffer contents are valid

STATUS_NO_SUCH_FILE – the directory is empty

STATUS_NO_MORE_FILES – there are no additional entries to enumerate, the buffer contents are not valid

FS_READ_DIRECTORY3

```
NTSTATUS
(*FS_READ_DIRECTORY3)(
    IN FS_FILE_HANDLE DirectoryHandle,
    IN OUT PULONG DirectoryOffset,
    IN PFS_DIRECTORY_ENTRY3 OutputBuffer,
    IN ULONG OutputBufferSize);
```

Status:		Required	Used if Present
	Media File Systems	Yes	Yes
	Network File Systems	Yes	Yes
	Pseudo File Systems	Yes	Yes

Parameters:

DirectoryHandle —The FSD-provided directory handle for the directory being read.

DirectoryOffset —This provides the *offset* context being used to enumerate the directory contents.

OutputBuffer —This buffer is provided by the Wrapper and may contain one or more directory entries.

OutputBufferSize —This indicates the size of the buffer provided by the Wrapper.

Description:

Note that this call has superseded FS_READ_DIRECTORY2. The two functions work identically (and are presently implemented using the same functional logic) but utilize slightly different structures. FS_READ_DIRECTORY3 uses the FS_DIRECTORY_ENTRY3 structure, which is compatible with the Windows directory structure data and thus simplifies the implementation of certain types of “pseudo” file systems.

The Wrapper manages the interface with the Windows directory enumeration mechanism, including packing user buffers, etc. The interface between the Wrapper and the FSD is a simpler iterative interface.

The Wrapper does this by making a series of calls into the FSD to retrieve file system information. The *DirectoryOffset* value is used to communicate information between successive calls to this interface routine.

That is, the first time this call will be made, the Wrapper will set this value to zero. Thus, the FSD should treat a *DirectoryOffset* value of zero as indicating an enumeration of the directory beginning with the first entry in the directory.

The FSD then uses the *OutputBuffer* as storage for its response to this request. The FSD must pack at least one entry into this buffer, if one is available, but may pack as many entries as can fit into the buffer and then returns STATUS_SUCCESS to the Wrapper to indicate there is some valid data within the buffer.

A directory entry has a fixed format:

```
typedef struct {ULONG EntrySize;
    ULONG LastDirectoryEntry;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastModifiedTime;
    LARGE_INTEGER Unused;
    LARGE_INTEGER ValidDataLength;
    LARGE_INTEGER DiskLength;
    ULONG Attributes;
    ULONG NameSize;
    ULONG EaSize;
    UCHAR ShortNameSize;
    WCHAR ShortName[12];
     LARGE_INTEGER FileID;
    WCHAR Name[1]; // and for the balance of this entry
} FS_DIRECTORY_ENTRY3, *PFS_DIRECTORY_ENTRY3;
```

*Note that this structure declaration includes the **EaSize** field which makes this compatible with the standard Windows directory entry.*

The Wrapper parses the buffer by examining each entry in turn, using a simple algorithm:

1. If *NameSize* is zero, the entry is skipped
2. Otherwise, the entry is processed as a valid directory entry
3. If *LastDirectoryEntry* is non-zero, the next entry in the buffer exists and will be processed

Thus, the buffer is parsed until the Wrapper has properly processed all valid entries within the buffer.

An FSD need not return the entire directory contents in a single buffer. To handle this case, the FSD returns the *DirectoryOffset* value. If, after processing the current buffer, this value is non-zero, the Wrapper will call the FSD again, asking for additional directory entries to process. This process continues until one of three things occurs:

1. The FSD returns a *DirectoryOffset* value of zero and a return value of STATUS_SUCCESS. This indicates that the current buffer has valid contents, but that there is no additional information available regarding this particular directory;
2. The FSD returns a status code of STATUS_NO_MORE_FILES or STATUS_NO_SUCH_FILE, in which case enumeration terminates immediately and the returned buffer is assumed to contain no valid data;
3. The FSD returns any error status, in which case it is treated as an I/O error and the user request to enumerate the directory will fail with the given error code. This should be reserved for serious FSD errors, such as a media or network failure.

If a directory is empty of all contents, the FSD should return STATUS_NO_SUCH_FILE.

Windows assumes that every directory contains two entries: "." and "..". An FSD need not return these entries to the Wrapper. If an FSD does return these entries to the wrapper the first entry returned must be

"." and the second entry "..". Otherwise, if the first entry is not "." and the second entry ".." the Wrapper will create a "." and ".." entry for the directory.

> ***Note:*** *At the risk of confusing things further, there is one case when the Wrapper does not create these entries for an empty directory. If the root directory of a volume is totally empty, in that particular case the semantics of the existing file systems (NTFS and FAT) is to return STATUS_NO_SUCH_FILE to the caller. These semantics are duplicated by the Wrapper.*

Returns:

STATUS_SUCCESS – the operation was successful and the buffer contents are valid

STATUS_NO_SUCH_FILE – the directory is empty

STATUS_NO_MORE_FILES – there are no additional entries to enumerate, the buffer contents are not valid

FS_READ_STREAM_INFORMATION

```
NTSTATUS
(*FS_READ_STREAM_INFORMATION)(
    IN FS_FILE_HANDLE FileHandle,
    IN OUT PFS_STREAM_ENTRY OutputBuffer,
    IN OUT PULONG OutputBufferSize);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file that is being queried with respect to its *stream* state.

OutputBuffer — This buffer contains one or more stream information structures within the given file.

OutputBufferSize — This value indicates the size of the *OutputBuffer* being returned by the FSD to the Wrapper.

Description:

This entry point is used by the Wrapper to enumerate the streams associated with a given file being stored by the FSD.

The *OutputBuffer* is allocated by the Wrapper and its size is indicated in *OutputBufferSize.* If the given buffer is not large enough to contain the requested information, the FSD should fail the request and indicate the minimum required size via *OutputBufferSize*.

Returns:

STATUS_SUCCESS – the operation was successful

STATUS_BUFFER_OVERFLOW – the stream information buffer *OutputBuffer* was too small to contain the information. The minimum size is indicated in *OutputBufferSize*.

FS_RELEASE

```
NTSTATUS
(*FS_RELEASE)(
    IN FS_FILE_HANDLE FileHandle);
```

Status:		Required	Used if Present
	Media File Systems	Yes	Yes
	Network File Systems	Yes	Yes
	Pseudo File Systems	Yes	Yes

Parameters:

FileHandle — The FSD-provided file handle to be released.

Description:

This routine is called by the Wrapper to release the given file handle from further use. The *FileHandle* is no longer considered valid after a call to Release, although the FSD is free to *cache* file handles for use with subsequent calls to *Lookup.*

Note that because the Wrapper is implementing an internal reference counting scheme, regardless of the number of calls the Wrapper makes to FS_LOOKUP there will be only a single call to FS_RELEASE, when the Wrapper is releasing the file handle. It is not necessary for the FSD to implement its own internal reference counting scheme.

The Wrapper will treat a release failure as catastrophic and halt system operation.

Note that it is considered normal system behavior that the call to FS_RELEASE is not coordinated with user applications closing the file. This is due to integration with the Virtual Memory system that caches file system data until the memory is needed for newer, more urgent operations. Thus, typically it is other I/O to the file systems that trigger the FS_RELEASE call.

Returns:

STATUS_SUCCESS – the release was successful

FS_REMOVE_SHARE_ACCESS

```
NTSTATUS
(*FS_REMOVE_SHARE_ACCESS)(
    IN FS_FILE_HANDLE FileHandle,
    IN ULONG ShareAccess);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file or directory handle on which the share check is to be made.

ShareAccess – a combination of FILE_SHARE_READ, FILE_SHARE_WRITE, and FILE_SHARE_DELETE options

Description:

This routine is used by the FSDK to indicate the release of shared access rights to the underlying file system, which were previously obtained via a successful call to FS_UPDATE_SHARE_ACCESS.

Returns:

STATUS_SUCCESS – the share access was successfully removed

STATUS_NOT_IMPLEMENTED – the only allowable error status

FS_RENAME

```
NTSTATUS
(*FS_RENAME)(
    IN FS_FILE_HANDLE SourceDirectoryHandle,
    IN FS_FILE_HANDLE TargetDirectoryHandle,
    IN FS_FILE_HANDLE SourceFileHandle,
    IN PWSTR TargetFileName,
    IN PWSTR TargetFileAlternateName,
    OUT PFS_FILE_HANDLE TargetFileHandle);
```

Status:		Required	Used if Present
	Media File Systems	No (Yes, if FS-CREATE is supported)	Yes
	Network File Systems	No (Yes, if FS-CREATE is supported)	Yes
	Pseudo File Systems	No	Yes

Parameters:

SourceDirectoryHandle — The FSD-provided directory handle of the directory that contains the currently existing file.

TargetDirectoryHandle — The FSD-provided directory handle of the directory that *will* contain the file upon completion of the operation.

SourceFileHandle — The FSD-provided file handle for the target of the rename operation.

TargetFileName — The name of the file in the target directory upon completion of this operation.

TargetFileAlternateName — The alternate (8.3) name of the file in the target directory upon completion of this operation. If this value is null, there is no alternative name.

TargetFileHandle — The handle of the newly renamed file upon completion of this operation.

Description:

The FS_RENAME routine is used to move a file from one directory to another directory. The *SourceDirectoryHandle* represents the directory where the file is currently located. The *TargetDirectoryHandle* represents the directory where the file will be located upon completion of this call. *SourceDirectoryHandle* and *TargetDirectoryHandle* may be identical.

If *SourceFileHandle* and *TargetFileHandle* differ, the Wrapper is responsible for updating its own internal state so that only *TargetFileHandle* is used in the future to access this file. The Wrapper will no longer use *SourceFileHandle* as if an implicit call to FS_RELEASE had been made with respect to the *SourceFileHandle*.

Returns:

STATUS_SUCCESS – the operation was successful

*This function is suitable only for use by file systems that **do not** support hard links. File systems that support hard links should use FS_RENAME2*

FS_RENAME2

```
NTSTATUS
(*FS_RENAME2)(
    IN FS_FILE_HANDLE SourceDirectoryHandle,
    IN FS_FILE_HANDLE TargetDirectoryHandle,
    IN FS_FILE_HANDLE SourceFileHandle,
    IN PWSTR SourceFileName,
    IN PWSTR TargetFileName,
    IN PWSTR TargetFileAlternateName,
    OUT PFS_FILE_HANDLE TargetFileHandle);
```

Status:		Required	Used if Present
	Media File Systems	No (Yes, if FS-CREATE is supported)	Yes
	Network File Systems	No (Yes, if FS-CREATE is supported)	Yes
	Pseudo File Systems	No	Yes

Parameters:

SourceDirectoryHandle — The FSD-provided directory handle of the directory that contains the currently existing file.

TargetDirectoryHandle — The FSD-provided directory handle of the directory that *will* contain the file upon completion of the operation.

SourceFileHandle — The FSD-provided file handle for the target of the rename operation.

SourceFileName — The name of the file in the source directory.

TargetFileName — The name of the file in the target directory upon completion of this operation.

TargetFileAlternateName — The alternate (8.3) name of the file in the target directory upon completion of this operation. If this value is null, there is no alternative name.

TargetFileHandle — The handle of the newly renamed file upon completion of this operation.

Description:

This routine is used to rename an existing file. This may include:

- Changing the name of a file within its current directory
- Moving a file from one directory to another directory
- Changing the *case* of the name of a file

Note that rename operations are not supported across volumes. However, if your file system does not expose the “normal” volume model you will be responsible for rejecting any cross-volume mount requests.

The *SourceDirectoryHandle* represents the directory where the file is currently located. The *TargetDirectoryHandle* represents the directory where the file will be located upon completion of this call. *SourceDirectoryHandle* and *TargetDirectoryHandle* may be identical.

If *SourceFileHandle* and *TargetFileHandle* differ, the Wrapper is responsible for updating its own internal state so that only *TargetFileHandle* is used in the future to access this file. The Wrapper will no longer use *SourceFileHandle* as if an implicit call to FS_RELEASE had been made with respect to the *SourceFileHandle*.

The *SourceFileName* is provided to allow file systems that support hard links to distinguish which particular instance of a directory entry is being renamed as a result of this operation. Otherwise, this entry point functions identically to the FS_RENAME entry point.

Returns:

STATUS_SUCCESS – the operation was successful

FS_SET_ACCESS_MODE

```
NTSTATUS
(*FS_SET_ACCESS_MODE)(
    IN FS_FILE_HANDLE FileHandle,
    IN UCHAR AccessMode);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file handle representing the instance of the file for which the access mode is being set.

AccessMode — A value indicating the type of access requested for this file.

Description:

The Wrapper calls this routine prior to any I/O calls on the given file to indicate the type of I/O access the FSD can expect for the given file.

The Wrapper will not call FS_READ, FS_WRITE, or FS_SET_LENGTH until this entry point has been called, but may use other entry points to obtain attribute information about the file. Once set, the nature of the file access will not be modified – that is, a second call will not be made to the FSD using the same file handle. Hence, subsequent accesses to the file must be consistent with previous accesses to the file.

The *AccessMode* may be one of the following values:

FS_ACCESS_NONE – no I/O operations will be performed for this file

FS_ACCESS_READ – only read I/O operations will be performed for this file

FS_ACCESS_WRITE – only write I/O operations will be performed for this file

FS_ACCESS_ALL – both read and write I/O operations will be performed for this file

FS_ACCESS_SEQUENTIAL – I/O access to this file is expected to be sequential

FS_ACCESS_RANDOM – I/O access to this file is expected to be random

FS_ACCESS_TEMPORARY – this file's existence is temporary in nature and is expected to be deleted or truncated.

© | OSR Open Systems Resources, Inc.

These access modes are “hints” into the underlying FSD based upon expected access patterns to the file. Access patterns not conforming to these hints should still work correctly, but might entail lower performance.

Ideally, the FSD may use this to prepare for I/O access. This allows an FSD to allocate any FSD-internal data structures or buffers necessary to provide FSD-specific service guarantees as well as optimizing use of FSD resources. For instance, certain file systems may need to allocate buffers differently depending upon this access mode.

An FSD need not provide this entry point. If this entry point is not provided, the Wrapper will treat the file as if FS_ACCESS_ALL had been granted for the given file.

Returns:

STATUS_SUCCESS – the operation was successful

Since the intent of this routine is to inform the FSD of the expected file access pattern, STATUS_SUCCESS is the only valid return value. All other return values will cause an assertion in the debug version of the wrapper.

FS_SET_ATTRIBUTES

```
NTSTATUS
(*FS_SET_ATTRIBUTES)(
    IN FS_FILE_HANDLE FileHandle,
    IN PFS_FILE_ATTRIBUTES FileAtttributes);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file handle for which the attribute information is being set.

FileAttributes — The Wrapper-allocated buffer containing the general file attributes.

Description:

This routine is used to modify the basic attributes associated with any file or directory stored by the FSD. Note that the FSD is free to specify that *any* of these attributes are "read-only" and that they cannot be modified via this call.

When the Wrapper wants to change a subset of parameters in a FS_FILE_ATTRIBUTES structure, it precedes FS_SET_ATTRIBUTES with a call to FS_GET_ATTRIBUTES. The Wrapper then changes the desired attribute, and calls FS_SET_ATTRIBUTES with the complete structure.

The Wrapped *does not* attempt to change file length (either used or allocated) using this function. *All* file length changes are done using FS_SET_LENGTH. Therefore, the FSD may ignore the AllocationSize and ValidDataLength fields passed during calls to the FS_SET_ATTRIBUTES function.

The Wrapper will strip the FILE_ATTRIBUTE_DIRECTORY from any incoming request to modify the attributes of the file. Thus, it is not possible for a user, via this interface, to modify a file into becoming a directory (or vice-versa.)

Further, the FILE_ATTRIBUTE_NORMAL bit will be sent to the FSD for otherwise "empty" requests in order to ensure that the FSD can ascertain that a change in attributes is being attempted. This bit is not part of the on-disk structure but is instead used to differentiate between "ignore this attribute field" and "clear all attributes".

> ***Note:*** *The FSDK does not modify or change any of these bits. Thus, it is the responsibility of the FSD to set bits as the implementers see fit. For example, the ARCHIVE bit may be set by the FSD as the result of a modification of the file. To the*

extent possible, the FSDK attempts to avoid implementing file system level policy and leaves this to the FSD.

Applications can, of course, request that these bits be modified. Calls to this interface that indicate such changes are as a result of such application-level change requests.

Returns:

STATUS_SUCCESS – the operation was successful

FS_SET_EA

```
NTSTATUS
(*FS_SET_EA)(
    IN FS_FILE_HANDLE FileHandle,
    IN PVOID EABuffer,
    IN ULONG EABufferSize);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file or directory handle against which the EA is being stored.

EABuffer — This buffer contains the extended attribute set for this file (if any).

EABufferSize — This value indicates the size of the *EABuffer* being passed to the FSD by the Wrapper.

Description:

This routine is used to set the extended attribute information associated with the specified file or directory. If no EA is available, the *EABuffer* will be a null pointer and the *EABufferSize* will be set to zero. Otherwise, the *EABuffer* will point to the buffer containing a FILE_FULL_EA_INFORMATION structure and *EABufferSize* will indicate its size.

The FILE_FULL_EA_INFORMATION structure (from *ntddk.h*) is:

```
typedef struct _FILE_FULL_EA_INFORMATION {
    ULONG NextEntryOffset;
    UCHAR Flags;
    UCHAR EaNameLength;
    USHORT EaValueLength;
    CHAR EaName[1];
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

Returns:

STATUS_SUCCESS – the operation was successful

FS_SET_LENGTH

```
NTSTATUS
(*FS_SET_LENGTH)(
    IN FS_FILE_HANDLE FileHandle,
    IN LARGE_INTEGER AllocationSize,
    IN LARGE_INTEGER DataSize);
```

Status:		Required	Used if Present
	Media File Systems	No (Yes, if FS-CREATE is supported)	Yes
	Network File Systems	No (Yes, if FS-CREATE is supported)	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file handle for which the size of the file is being set.

AllocationSize — The minimum amount of disk allocation to be provided for this file, in bytes.

DataSize — The size of the valid data section of the file.

Description:

This routine is used to control both the allocation and valid data portion information of a file. The allocation of a file must always be at least as large as the valid data portion of the file.

The *AllocationSize* does not represent the actual amount of physical disk space allocated to the file. Instead, it represents the amount of *data* that may fit in the space that has previously been allocated to this file. Thus, a file system that implements any compression scheme (including "sparse storage") must represent the AllocationSize to be sufficiently large that the data of the file can fit into that space.

The *DataSize* represents the end of the file with respect to the amount of data that might fit into the file.

An FSD is free to implement the data storage of the file in whatever manner the implementers see fit. Thus, even when allocation size information of the file is extended, the FSD may ignore such a request until a subsequent I/O operation actually uses this information. Alternatively, an FSD may allocate even more space than is requested in order to match the specific operational requirements of the underlying hardware.

Returns:

STATUS_SUCCESS – the operation was successful

STATUS_DISK_FULL – the operation failed because there was no disk space available to satisfy the request.

FS_SET_SECURITY

```
NTSTATUS
(*FS_SET_SECURITY)(
    IN FS_FILE_HANDLE FileHandle,
    IN PVOID SecurityDescriptor,
    IN ULONG SecurityDescriptorSize);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file or directory handle on which the security descriptor is to be applied.

SecurityDescriptor — This buffer contains the self-relative format security descriptor to be stored with this file (if any).

SecurityDescriptorSize — This value indicates the size of the *SecurityDescriptor* passed to the FSD from the Wrapper.

Description:

This routine is used to save the Windows-format security information for the given file.

Support for FS_SET_SECURITY is optional. However, the FSD must support both FS_GET_SECURITY and FS_SET_SECURITY if it supports either interface.

If the file should have no security descriptor (or the current security descriptor should be deleted) the *SecurityDescriptor* will be a null pointer and the *SecurityDescriptorSize* will be set to zero.

Otherwise, the *SecurityDescriptor* will point to the buffer containing the Windows security information that should be stored with the given file. The buffer pointed to by *SecurityDescriptor* is owned by the wrapper and should not be modified by the FSD.

The security information contained in the buffer is generally not useful to the FSD. This interface provides a simple mechanism that allows a file system to support standard Windows security policies by simply storing and retrieving security information on a per file (or directory) basis. The wrapper performs all security operations for the FSD if the FS_GET_SECURITY and FS_SET_SECURITY interfaces are implemented by the FSD.

Returns:

STATUS_SUCCESS – the operation was successful

© | OSR Open Systems Resources, Inc.

FS_SET_VOLUME_ATTRIBUTES

```
NTSTATUS
(*FS_SET_VOLUME_ATTRIBUTES)(
    IN FS_VOL_HANDLE FsdVolumeHandle,
    IN PFS_VOL_ATTRIBUTES VolumeAtttributes);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FsdVolumeHandle — The FSD-provided handle representing the volume to be used for this operation.

VolumeAttributes — The Wrapper-allocated buffer containing the general volume attributes.

Description:

This routine is used to modify the basic attributes associated with the given volume. Note that the FSD is free to specify that *any* of these attributes are “read-only” and that they cannot be modified via this call.

Typically, this call is only used to modify the volume label.

Returns:

STATUS_SUCCESS – the volume label has been changed

FS_SHUTDOWN

```
NTSTATUS
(*FS_SHUTDOWN)(
    IN ULONG Stage);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

Stage — One of the following two values:

OW_SHUTDOWN INITIATE – indicates start of shutdown

OW_SHUTDOWN_COMPLETE – indicates termination of shutdown

Description:

This routine is used to advise the underlying FSD that a shutdown is in progress. It calls initially to notify the FSD that shutdown has started. The FSDK will then dismount all volumes and, finally, notify the FSD that shutdown processing has completed.

Returns:

Ignored.

FS_UNC_ROOT

```
VOID
(*FS_UNC_ROOT)(
    OUT PFS_FILE_HANDLE DirectoryHandle);
```

Status:		Required	Used if Present
	Media File Systems	No	No
	Network File Systems	No	Yes
	Pseudo File Systems	No	No

Parameters:

DirectoryHandle — An FSD-provided handle representing the root directory of the given FSD. This directory handle can then be used in subsequent FSD operations.

Description:

This routine is used by the Wrapper to retrieve a valid handle to the root directory of the "UNC" form for the FSD. This handle will be used for all future FSD operations when a UNC-style name is used.

Because these handles are opaque values to the Wrapper, one possible implementation of this would be to use a "magic number" which indicated (in a subsequent call to FS_CREATE) that the directory being used was the special UNC name directory.

The value zero is not allowed for use as a file handle.

Returns:

None.

FS_UNLOCK

```
NTSTATUS
(*FS_UNLOCK)(
    IN FS_FILE_HANDLE FileHandle,
    IN FS_LOCK_HANDLE LockHandle);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided handle identifying the given file.

LockHandle — The FSD-defined identifier for this particular lock.

Description:

This routine is used by the Wrapper to allow a network file system to implement a distributed byte-range-locking scheme. If present, the Wrapper will call into the FSD to release a previously granted lock.

The *LockHandle* parameter was created by the FSD in a prior call to the FS_LOCK entry point and is used by the FSD to identify this particular lock.

It is considered an error for an FSD to implement the *FS_UNLOCK* entry point unless the *FS_LOCK* entry point is also defined.

Returns:

STATUS_SUCCESS – the lock was released

STATUS_RANGE_NOT_LOCKED – the memory range specified could not be unlocked because it was not previously locked.

FS_UNMOUNT

```
NTSTATUS
(*FS_UNMOUNT)(
    IN FS_VOL_HANDLE FsdVolumeHandle);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	No
	Pseudo File Systems	No	Yes

Parameters:

FsdVolumeHandle — The FSD-provided handle representing the volume to be dismounted.

Description:

This routine is used by the Wrapper to *dismount* the given file system volume. It is an error for this to be called if there are *any* outstanding references to any FSD objects provided to the Wrapper.

Upon completion of this routine, the FsdVolumeHandle is assumed to be invalid and no further references to this file system can be made.

Note that if this interface is not supported, use of removable media is not allowed. Further, the file system will *not* be called during shutdown processing.

Returns:

STATUS_SUCCESS – the volume has been successfully dismounted

© | OSR Open Systems Resources, Inc.

FS_UPDATE_SHARE_ACCESS

```
NTSTATUS
(*FS_UPDATE_SHARE_ACCESS)(
    IN FS_FILE_HANDLE FileHandle,
    IN ULONG ShareAccess);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file or directory handle on which the share check is to be made.

ShareAccess – a combination of FILE_SHARE_READ,FILE_ SHARE_WRITE, and FILE_SHARE_DELETE options.

Descriptions:

This routine may be provided by an FSD to participate in arbitrating share access to a given file. When a file is opened for access, the open operation indicates if the caller wishes to share access with anyone else accessing the given file or directory.

When a subsequent attempt to open the file is made, its share access must be compatible with the share access of any other process that currently has the file open.

Normally, the FSDK handles this share access internally. However, for multi-initiator file systems (e.g., network) it is essential that they arbitrate between all users of the file. When an FSD implements share access checking, this will replace, not augment, the FSDK's default handling of share access.

The FSD must return either SUCCESS, in which case the access is granted, or SHARING_VIOLATION, in which case the access is denied.

If sharing is granted, the FSD is responsible for tracking this information until the file is either released (which implicitly releases all sharing controls on the file) or a call to FS_REMOVE_SHARE_ACCESS is made.

See the Windows DDK Reference Manual on IoSetShareAccess, IoCheckShareAccess, IoUpdateShareAccess, and IoRemoveShareAccess for further information.

Returns:

STATUS_SUCCESS – the share access is compatible.

STATUS_ SHARING_VIOLATION – the share access is not compatible.

FS_VERIFY

```
NTSTATUS
(*FS_VERIFY)(
    IN FS_VOL_HANDLE FsdVolumeHandle);
```

Status:		Required	Used if Present
	Media File Systems	Yes	Yes
	Network File Systems	No	No
	Pseudo File Systems	No	Yes

Parameters:

FsdVolumeHandle — The FSD-provided handle representing the volume to be verified.

Description:

The *verify* entry point is used by the Wrapper to request that the FSD verify the given volume is the same volume as was previously present. A media file system FSD is responsible for reading the signature information from the disk and verifying that the media has not changed.

This entry point is used to determine if the media in the device has changed. Thus, unlike mount, the OS uses this entry point to validate that the media, which was previously mounted by this FSD, is still the same volume.

Note that if an FSD fails a verification request, the Wrapper will treat the volume as dismounted. That is, a separate call to FS_UNMOUNT will not be made to the FSD.

Subsequent user-level I/O operations that access the volume will fail, as the media is no longer mounted.

For pseudo file systems, it is the responsibility of the FSD to do verification of the pseudo-volume. What this entails is up each individual FSD. It is the Pseudo FSD's responsibility to do whatever additional processing is necessary (including unmounting the pseudo volume) if the verify fails.

Returns:

STATUS_SUCCESS - the volume has not changed

STATUS_REPARSE - the volume has changed, the request should be re-issued against the volume (this is actually for pseudo file systems so that they can handle verify failures of their underlying volume)

STATUS_WRONG_VOLUME - the volume has changed and no longer contains the same media

FS_WRITE

```
NTSTATUS
(*FS_WRITE)(
    IN FS_FILE_HANDLE FileHandle,
    IN LARGE_INTEGER FileOffset,
    IN PMDL Buffer,
    IN ULONG BufferSize,
    OUT PULONG BytesWritten);
```

Status:		Required	Used if Present
	Media File Systems	No	Yes
	Network File Systems	No	Yes
	Pseudo File Systems	No	Yes

Parameters:

FileHandle — The FSD-provided file handle against which this write operation is being performed.

FileOffset — The byte-range offset into the file where the I/O begins.

Buffer — A pointer to the MDL representing the pages where data should be read from the disk into memory.

BufferSize — The number of bytes to be written from the region described by *Buffer*.

BytesWritten — The total number of bytes successfully written by the FSD to the backing store.

Description:

This entry point is used by the Wrapper to perform actual I/O. Because the *normal* model for the Wrapper is to cache writes, this entry point will typically be called to handle a non-cached I/O request. Typically this is to handle a *write* operation from the modified page daemon, or the lazy writer, both of which are dedicated threads in the Windows system implementing slightly different write-back policies for cached data.

To facilitate the development of an FSD, the FSD may assume that the I/O operations adhere to the following guidelines:

For physical media file systems, the *FileOffset* will be aligned on a sector-sized boundary. For all other file systems, the alignment will be 512 bytes. In fact, we expect that this will be page aligned, but we use the weaker requirement to support non-cached I/O directly from utilities and user applications, although such are rare.

If this entry point returns STATUS_SUCCESS, *BytesWritten* will indicate the actual number of bytes written by the FSD.

Upon completion of the Write operation by the FSD, *BytesWritten* must be less than or equal to *BufferSize*.

The Wrapper is responsible for managing the correct length of valid data in the file (although the FSD is responsible for *storage* of that information). Thus, *BytesWritten* does not indicate that the size of the valid data portion of the file has been extended.

The principal goal of this design is to ensure that I/O to the FSD falls on naturally aligned boundaries. Note that typically, I/O is done in PAGE_SIZE units. Odd sizes are typically only used for "boundary conditions" such as the last component (less than PAGE_SIZE) of a file.

Returns:

STATUS_SUCCESS – the operation was successful

STATUS_WRONG_VOLUME – the volume has changed and the read cannot be completed with the currently available volume (verify is required.)

STATUS_NO_MEDIA_IN_DEVICE – the volume is not available because the media has been removed.

STATUS_INSUFFICIENT_RESOURCES – the FSD was unable to complete the I/O operation due to insufficient memory resources

STATUS_DISK_CORRUPT – the file system structure on the disk is damaged.

STATUS_DISK_FULL – the volume is full and no additional data may be written to it.

OPERATIONS REQUIREMENTS

Overview

In this section we describe the requirements for the various operations. The following table sets forth the operation and indicates for media, network, and pseudo file systems if the operation is *Required, Optional*, required *Conditionally* on support of FS_CREATE, or *Unused.*

A *required* operation *must* be present for the FSD or the FSD will not function properly.

An *optional* operation is one that will be used by the Wrapper if it is present. Otherwise, the Wrapper will handle the error conditions.

An *unused* operation is one that cannot be used for the given type of FSD.

Operation	**Media File Systems**	**Network File Systems**	**Pseudo File Systems**
FS_ACCESS	O	O	O
FS_CHECK_LOCK	O	O	O
FS_CLEAR	O	O	O
FS_CONNECT	U	R	U
FS_CREATE	O	O	O
FS_DELETE	O	O	O
FS_DELETE2	O	O	O
FS_DELETE3	O	O	O
FS_DISCONNECT	U	O	U
FS_FLUSH	O	O	O
FS_FSCTRL	O	O	O
FS_GET_ATTRIBUTES	R	R	R
FS_GET_EA	O	O	O
FS_GET_NAMES	O	O	O

© | OSR Open Systems Resources, Inc.

FS_GET_NAME2	O	O	O
FS_GET_SECURITY	O	O	O
FS_GET_UNC_VOLUME_ATTRIBUTES	U	O	U
FS_GET_VOLUME_ATTRIBUTES	R	O	R
FS_IOCTL	O	O	O
FS_LINK	O	O	O
FS_LOCK	O	O	O
FS_LOOKUP	R	R	R
FS_LOOKUP_BY_ID	O	O	O
FS_LOOKUP_BY_OBJECT_ID			
FS_LOOKUP_PATH	O	O	O
FS_MOUNT	R	U	R
FS_QUERY_PATH	U	R	U
FS_READ	R	R	R
FS_READ_DIRECTORY	R	R	R
FS_READ_DIRECTORY2	R	R	R
FS_READ_DIRECTORY3	R	R	R
FS_READ_STREAM_INFORMATION	O	O	O
FS_RELEASE	R	R	R
FS_REMOVE_SHARE_ACCESS	O	O	O
FS_RENAME	C	C	O
FS_RENAME2	C	C	O
FS_SET_ACCESS_MODE	O	O	O
FS_SET_ATTRIBUTES	O	O	O

FS_SET_EA	O	O	O
FS_SET_LENGTH	C	C	C
FS_SET_SECURITY	O	O	O
FS_SET_VOLUME_ATTRIBUTES	O	O	O
FS_UNC_ROOT	U	O	U
FS_UNLOCK	O	O	O
FS_UNMOUNT	O	U	O
FS_UPDATE_SHARE_ACCESS	O	O	O
FS_VERIFY	R	U	R
FS_WRITE	O	O	O

Important Notes:

- FS_GET_NAME2, FS_DELETE3, and FS_RENAME2 must be supported if FS_LOOKUP_BY_ID or FS_LOOKUP_BY_OBJECT_ID is supported.
- FS_READ_DIRECTORY or FS_READ_DIRECTORY2 or FS_READ_DIRECTORY3 must be implemented but it is not necessary to implement all three.
- FS_RENAME or FS_RENAME2 must be implemented (conditionally), but it is not necessary to implement both.

DATA STRUCTURES

Overview

In this section we describe the data structures used by this interface.

FS_VOL_HANDLE

```
typedef PVOID FS_VOL_HANDLE;
```

The FS_VOL_HANDLE is used to identify a file system volume. It is created by the FSD and used by the Wrapper to identify the specific volume instance to the FSD.

FS_FILE_HANDLE

```
typedef PVOID FS_FILE_HANDLE;
```

The FS_FILE_HANDLE is used to identify a *file* or *directory*.

FS_LOCK_HANDLE

```
typedef PVOID FS_FILE_HANDLE;
```

The FS_LOCK_HANDLE is used to identify a *lock* created by an FSD. It is created by the FSD and used by the Wrapper to identify the specific lock instance to the FSD.

FS_DELETE3_EXTENDED_INFO

```
typedef struct {
    USHORT Size;
    USHORT Version;
    FS_FILE_HANDLE ParentDirectoryHandle;
    PWSTR FileName;
    BOOLEAN ReleaseFileHandle;
} FS_DELETE3_EXTENDED_INFO, *PFS_DELETE3_EXTENDED_INFO;
```

This structure is used with the FS_DELETE3 entry point to indicate to the FSD more information about the item being deleted. Size will be set to the size of the FS_DELETE3_EXTENDED_INFO structure and version will be set to FS_DELETE3_EXTENDED_INFO_V1. This structure is initialized by the FSDK.

FS_DIRECTORY_ENTRY

```
typedef struct {
    ULONG EntrySize;
    ULONG LastDirectoryEntry;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastModifiedTime;
    LARGE_INTEGER Unused;
    LARGE_INTEGER ValidDataLength;
    LARGE_INTEGER DiskLength;  // storage-based
    ULONG Attributes;
    ULONG NameSize;            // in bytes
    UCHAR ShortNameSize;       // in bytes
    WCHAR ShortName[12];
    WCHAR Name[1];  // and for the balance of this entry
} FS_DIRECTORY_ENTRY, *PFS_DIRECTORY_ENTRY;
```

The FS_DIRECTORY_ENTRY is used by the FS_READ_DIRECTORY entry point to allow the Wrapper to enumerate the contents of a directory.

Time stamps are to be in Windows standard format that is the number of 100 nanosecond intervals since January 1, 1601.

FS_DIRECTORY_ENTRY2

```
typedef struct {
    ULONG EntrySize;
    ULONG LastDirectoryEntry;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastModifiedTime;
    LARGE_INTEGER Unused;
    LARGE_INTEGER ValidDataLength;
    LARGE_INTEGER DiskLength;     // storage-based
    ULONG Attributes;
    ULONG NameSize;               // in bytes
    ULONG EaSize;                 // in bytes
    UCHAR ShortNameSize;          // in bytes
    WCHAR ShortName[12];
    WCHAR Name[1]; // and for the balance of this entry
} FS_DIRECTORY_ENTRY2, *PFS_DIRECTORY_ENTRY2;
```

The FS_DIRECTORY_ENTRY2 is used by the FS_READ_DIRECTORY2 entry point to allow the Wrapper to enumerate the contents of a directory.

Time stamps are to be in Windows standard format, that is the number of 100 nanosecond intervals since January 1, 1601.

FS_DIRECTORY_ENTRY3

```
typedef struct {
    ULONG EntrySize;
    ULONG LastDirectoryEntry;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastModifiedTime;
    LARGE_INTEGER Unused;
    LARGE_INTEGER ValidDataLength;
    LARGE_INTEGER DiskLength;  // storage-based
    ULONG Attributes;
    ULONG NameSize;            // in bytes
     ULONG EaSize;                 // in bytes
    UCHAR ShortNameSize;       // in bytes
    WCHAR ShortName[12];
    LARGE_INTEGER FileID;
    WCHAR Name[1];  // and for the balance of this entry
} FS_DIRECTORY_ENTRY3, *PFS_DIRECTORY_ENTRY3;
```

The FS_DIRECTORY_ENTRY3 is used by the FS_READ_DIRECTORY3 entry point to allow the Wrapper to enumerate the contents of a directory.

Time stamps are to be in Windows standard format that is the number of 100 nanosecond intervals since January 1, 1601.

FS_FILE_ATTRIBUTES

```
typedef struct {
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastModifiedTime;
    ULONG Attributes;
    LARGE_INTEGER LastByteWritten;
    LARGE_INTEGER ValidDataLength;
    LARGE_INTEGER DiskLength;
    ULONG LinkCount;
    LARGE_INTEGER FileID;
} FS_FILE_ATTRIBUTES, *PFS_FILE_ATTRIBUTES;
```

The *FS_FILE_ATTRIBUTES* are used to describe the “generic” attributes associated with a given file or directory. They correspond to the standard Windows file attributes, including the *Attributes* field which indicates information including *read-only, system, hidden,* and *archive.*

FS_GET_NAMES2_EXTENDED_INFO

```
typedef enum {
    FsFileOpenByName = 1000,
    FsFileOpenById,
    FsFileOpenByObjectId,
} FS_FILE_OPEN_TYPE, *PFS_FILE_OPEN_TYPE;


typedef enum {
    FsCaseUnknown,
    FsCaseSensitive=2000,
    FsCaseInsensitive,
} FS_CASE_SENSITIVITY_TYPE, *PFS_FILE_OPEN_TYPE;


typedef struct {
    USHORT Size;
    USHORT Version;
    FS_FILE_OPEN_TYPE  FileOpenType;
    FS_CASE_SENSITIVITY_TYPE FileCaseSensitivityType;
    UNICODE_STRING FileName;  // READ_ONLY
    IN OUT OPTIONAL PFS_FILE_HANDLE ParentDirectory;
    IN OUT BOOLEAN FullPathNameReturned;
} FS_GET_NAMES2_EXTENDED_INFO, *PFS_GET_NAME2_EXTENDED_INFO;
```

This structure is used with the FS_GET_NAMES2 Entry point to indicate to the FSD more information about the names requested. Size will be set to the size of the FS_GET_NAMES2_EXTENDED_INFO structure and version will be set to FS_GET_NAMES2_EXTENDED_INFO_V1. This structure is initialized by the FSDK.

FS_STREAM_ENTRY

```
typedef struct {
    ULONG EntrySize;
    ULONG NameSize;
    LARGE_INTEGER ValidDataLength;
    LARGE_INTEGER DiskLength;
    WCHAR Name[1]; // balance of this entry; contains
                       // portion of file name after the
                       // colon (:)
} FS_STREAM_ENTRY, *PFS_STREAM_ENTRY;
```

The *FS_STREAM_ENTRY* is used to enumerate streams associated with a given file.

FS_VOL_ATTRIBUTES

```
typedef struct {
    LARGE_INTEGER CreationTime;
    ULONG SerialNumber;
    ULONG LabelLength;
    BOOLEAN Unused; // must be zero
    LARGE_INTEGER VolumeSize; // in allocation units
    LARGE_INTEGER FreeSpace; // in allocation units
    ULONG BytesPerAllocationUnit;
    ULONG DeviceType; // FS type, one of FILE_DEVICE_DISK_FILE_SYSTEM,
FILE_DEVICE_NETWORK_FILE_SYSTEM
    ULONG DeviceCharacteristics;
    WCHAR VolumeLabel[1]; // and for balance of entry
} FS_VOL_ATTRIBUTES, *PFS_VOL_ATTRIBUTES;
```

The *volume attributes* describe basic information about the underlying disk volume.

FS_EXTENDED_VOL_ATTRIBUTES

The following structure may optionally be used instead of the FS_VOL_ATTRIBUTES structure:

```
typedef struct {
    LARGE_INTEGER CreationTime;
    ULONG SerialNumber;
    ULONG LabelLength;
    BOOLEAN Flags;
    BOOLEAN Unused[3];
    LARGE_INTEGER VolumeSize; // in allocation units
    LARGE_INTEGER FreeSpace; // in allocation units
    ULONG BytesPerAllocationUnit;
    ULONG DeviceType; // FS type, one of DISK_FILE_SYSTEM,
NETWORK_FILE_SYSTEM
    ULONG DeviceCharacteristics;
    USHORT MaximumComponentNameLength;
    USHORT FileSystemNameLength;
    WCHAR VolumeLabel[1]; // and for balance of entry
    // file system name immediately follows the volume label
} FS_EXTENDED_VOL_ATTRIBUTES, *PFS_EXTENDED_VOL_ATTRIBUTES;

#define OW_VOL_ATTR_FLAG_EXTENDED (0x01)
```

In order to allow the FSDK to distinguish between the two, OW_VOL_ATTR_FLAG_EXTENDED must be set in the *Flags* field, which corresponds with a previously unused field in the volume attributes structure.

File Types

The *file types* are used to distinguish the various potential types of objects represented by an FS_FILE_HANDLE.

```
#define OSRFS_TYPE_FILE          0x1
#define OSRFS_TYPE_DIRECTORY     0x2
#define OSRFS_TYPE_SYMBOLIC_LINK 0x4
#define OSRFS_TYPE_OTHER         0xFF
```

File Attributes

File attributes passed by the Wrapper are those used and defined by Windows:

```
#define FILE_ATTRIBUTE_READONLY         0x00000001
#define FILE_ATTRIBUTE_HIDDEN           0x00000002
#define FILE_ATTRIBUTE_SYSTEM           0x00000004
#define FILE_ATTRIBUTE_DIRECTORY        0x00000010
#define FILE_ATTRIBUTE_ARCHIVE          0x00000020
#define FILE_ATTRIBUTE_NORMAL           0x00000080
#define FILE_ATTRIBUTE_TEMPORARY        0x00000100
#define FILE_ATTRIBUTE_RESERVED0        0x00000200
#define FILE_ATTRIBUTE_RESERVED1        0x00000400
#define FILE_ATTRIBUTE_COMPRESSED       0x00000800
#define FILE_ATTRIBUTE_OFFLINE          0x00001000
#define FILE_ATTRIBUTE_PROPERTY_SET     0x00002000
#define FILE_ATTRIBUTE_VALID_FLAGS      0x00003fb7
#define FILE_ATTRIBUTE_VALID_SET_FLAGS  0x00003fa7
```

Volume Attributes

Volume attributes used by the Wrapper have a one-to-one correspondence with the following Windows volume attributes:

FILE_CASE_SENSITIVE_SEARCH

FILE_CASE_PRESERVED_NAMES

FILE_UNICODE_ON_DISK

FILE_PERSISTENT_ACLS

FILE_FILE_COMPRESSION

FILE_VOLUME_IS_COMPRESSED

FS_OPERATIONS

The following describes the total set of operations that are provided by the file system driver (FSD) and the data structure used to represent those entries:

```
typedef struct _FS_OPERATIONS {
    UNICODE_STRING    FsName;               // name of file system driver
    ULONG             Flags;                // flags
    ULONG             MaximumIoSize;        // maximum I/O transfer allowed, 0 = default
    ULONG             Attributes;           // FSD Attributes - OW_FS_ATTR_* listed below
    ULONG             ReadAheadSize;        // Maximum read-ahead size
    ULONG             WriteBehindSize;      // Maximum write-behind size
    ULONG             DirReadSize;          // Maximum read size for directory enum, 0 = default
    PVOID             UpcaseTable;          // FSD's upper case table, 0 = default
    ULONG             AlignmentRequirement; // e.g. sector size -1 (511 for a 512 byte sector.)
    ULONG             Reserved[3];          // Reserved for future use
    PFS_OPERATIONS_FILTER      FilterTable;
    FS_MOUNT                   Mount;
    FS_VERIFY                  Verify;
    FS_UNMOUNT                 Unmount;
    FS_FSCTRL                  FsCtrl;
    FS_GET_QUOTA               GetQuota;
    FS_SET_QUOTA               SetQuota;
    FS_GET_VOLUME_ATTRIBUTES   GetVolAttributes;
    FS_SET_VOLUME_ATTRIBUTES   SetVolAttributes;
    FS_LOOKUP                  Lookup;
    FS_LOOKUP_BY_ID            LookupById;
    FS_LOOKUP_PATH             LookupPath;
    FS_SET_ACCESS_MODE         SetAccessMode;
    FS_RELEASE                 Release;
    FS_CREATE                  Create;
    FS_DELETE                  Delete;
    FS_RENAME                  Rename;
    FS_ACCESS                  Access;
    FS_GET_SECURITY            GetSecurity;
    FS_SET_SECURITY            SetSecurity;
    FS_LINK                    Link;
    FS_READ_SYMBOLIC_LINK      ReadSymbolicLink;
    FS_WRITE_SYMBOLIC_LINK     WriteSymbolicLink;
    FS_GET_ATTRIBUTES          GetAttributes;
    FS_SET_ATTRIBUTES          SetAttributes;
    FS_FLUSH                   Flush;
    FS_SET_LENGTH              SetLength;
    FS_GET_EA                  GetEA;
    FS_SET_EA                  SetEA;
    FS_READ                    Read;
    FS_WRITE                   Write;
    FS_IOCTL                   Ioctl;
    FS_READ_DIRECTORY          ReadDirectory;
    FS_READ_STREAM_INFORMATION ReadStreamInformation;
    FS_QUERY_PATH              QueryPath;
    FS_CONNECT                 Connect;
    FS_DISCONNECT              Disconnect;
```

```
    FS_UNC_ROOT                 UncRoot;
    FS_LOCK                     Lock;
    FS_UNLOCK                   Unlock;
    FS_GET_UNC_VOLUME_ATTRIBUTES GetUncVolAttributes;
    FS_CHECK_LOCK                 CheckLock;
    FS_UPDATE_SHARE_ACCESS      UpdateShareAccess;
    FS_REMOVE_SHARE_ACCESS      RemoveShareAccess;
    FS_DELETE2                  Delete2;
    FS_READ_DIRECTORY2          ReadDirectory2;
    FS_RENAME2                  Rename2;
    FS_OPEN                     Open;
    FS_CLOSE                    Close;
    FS_READ_COMPRESSED          ReadCompressed;
    FS_WRITE_COMPRESSED         WriteCompressed;

    FS_READ                     PagefileRead;
    FS_WRITE                    PagefileWrite;
    FS_CLEAR                    Clear;
    FS_GET_NAMES                GetNames;
    FS_READ_DIRECTORY3          ReadDirectory3;
    FS_LOOKUP_BY_OBJECT_ID      LookupByObjectId;
    FS_SHUTDOWN                 Shutdown;
    FS_DELETE3                  Delete3;
    FS_GET_NAMES2               GetNames2;
    FS_RESERVED_FUNCTION        ReservedFunctions[6];
} FS_OPERATIONS, *PFS_OPERATIONS;
```

This structure is used by the FSD when registering these operations with the Wrapper.

Most entries in this structure have previously been described (they are the entry points declared by the FSD and used by the Wrapper.) The initial fields within this structure, however are used for the following:

FsName – this field is used to create the named file system device object. For media file systems, this is in the root directory of the object manager name space. For network file systems, this is in the Device portion of the object manager name space. For network file systems, this name is also used to create a symbolic link to the file system. All other file system types are responsible for creating such a link.

> *Existing Windows media file systems do not create such a Win32-visible symbolic link name.*

Flags – this field indicates specific behavior of the underlying FSD. These flags correspond to the volume attributes described in the previous section, Volume Attributes.

MaximumIoSize – this field is used by the Wrapper to determine the maximum single I/O transfer allowed as part of a read or write I/O operation. The default value (64KB) corresponds with the maximum size used by the Windows VM system. This value may be increased to allow applications doing non-cached I/O to perform large I/O operations directly to disk. The wrapper will break oversized operations into a series of smaller sub-operations. This value must be at least 64KB.

Attributes – this field identifies unique attributes of the file system. Currently the defined attributes and their meaning:

- OW_FS_ATTR_NO_DIR_CACHE – indicates the file system in question does not wish to have the FSDK Wrapper provide any *directory caching* support. Each directory enumeration request is passed to the underlying file system.
- OW_FS_ATTR_NO_FILE_CACHE – indicates the file system in question does not wish to have the FSDK Wrapper provide any *data caching* support. Each file system read request is passed to the underlying file system.
- OW_FS_ATTR_NO_OPLOCKS – indicates the file system in question does not support oplocks. Network file systems do not support oplocks regardless of this value.
- OW_FS_ATTR_COMPRESSION – indicates the file system in question wishes to support *data compression.*

  This option is reserved for future use.

- OW_FS_ATTR_SHORT_NAMES – indicates the file system in question supports *short file names*. The FSDK Wrapper will provide 8.3 (DOS-compatible) names to the underlying file system. The FSDK will look up the short name of a file in one of two ways: 1) If the FSD does not support the FS_GET_NAMES entry point, then the FSDK will enumerate the target directory looking for the file in order to get its short name. 2) If the FSD supports the FS_GET_NAMES entry point, then this routine will be called to retrieve the short name of a file.
- OW_FS_ATTR_FSD_SHORT_NAMES – indicates that the FSD (not the FSDK) generates and supports short names (it is assumed that the generated short names are 8.3 compliant). Retrieval of these generated names is via the new FS_GET_NAMES function that was introduced in FSDK V2.0. Support for this function is mandatory if the OW_FS_ATTR_FSD_SHORT_NAMES flag is set. This bit would typically be set by a Pseudo File System that was layered upon an existing File System such as NTFS or FAT, which generated their own short names.
- OW_FS_ATTR_WILDCARDS_LEGAL – indicates the file system in question supports Windows wildcards as valid file name characters. If this option is not set, the FSDK rejects names with wildcards contained within them.

ReadAheadSize – this is the default amount of "read ahead" support requested by the underlying FSD. The default value of 64KB is used if this field is zero.

WriteBehindSize – this is the maximum amount of dirty data that may be cached by the FSDK Wrapper. Once this limit is reached, subsequent writes are blocked until dirty, cached data can be written to disk. A zero value indicates there is no maximum write behind size for the file system.

DirReadSize – this indicates the preferred buffer size for directory enumeration. A zero value indicates the FSDK default (currently 2 * PAGE_SIZE) should be used.

UpcaseTable – this indicates the upper case table the FSD would like the FSDK to use when performing case insensitive comparison operations. If no table is provided, the operating system default upcase table is used.

Reserved – these values are reserved for future use.

This structure is initially set up by the FSD and is passed to the Wrapper as part of its normal registration callOwRegister.

OW_WORK

This structure is used to describe a work function. See OwPostWork for information about utilizing this structure within an FSD.

The structure is:

```
typedef struct _OW_WORK {
    VOID     (*WorkFunction)(PVOID ContextValue);
    PVOID    ContextValue;
} OW_WORK, *POW_WORK
```

WRAPPER IOCTL INTERFACE

Overview

A number of operations inherent in implementing particular types of file systems require processing or coordination by both the FSD and the Wrapper. Where these operations are implemented from user-mode applications, I/O Control Codes (IOCTLs) are used to pass these requests to the FSDK. Some requests cause the Wrapper to take a particular action. Other pre-defined IOCTLs are simply passed along to the FSD. The individual codes are described below.

OW_FSCTL_MOUNT_PSEUDO

This IOCTL is used to create a new "device" for the pseudo file system volume. The data structure associated with this request is:

```
typedef struct {
    HANDLE PseudoVolumeHandle;
    USHORT PseudoDeviceNameLength;
    PWCHAR PseudoDeviceName;
    USHORT PseudoLinkNameLength;
    PWCHAR PseudoLinkName;
}OW_PSEUDO_MOUNT_INFO;
```

The PseudoVolumeHandle will be passed from the Wrapper to the FSD as part of mount processing, to allow the pseudo volume handle to "mount" the new pseudo volume. The PseudoDeviceName indicates the name that should be used when creating the new mount point in the "\Device" directory. The PseudoLinkName is the name that should be used when creating a symbolic link in the "\DosDevices" directory to the newly created device. If either name exists, the pseudo mount aborts and the entire operation fail.

OW_FSCTL_MOUNT_PSEUDO2

This IOCTL is a superset of OW_FSCTL_MOUNT_PSEUDO. The only difference between the two IOCTLs is that OW_FSCTL_MOUNT_PSEUDO2 allows passing the DeviceType and Characteristics. The data structure associated with this request is:

```
typedef struct {
    HANDLE PseudoVolumeHandle;
    USHORT PseudoDeviceNameLength;
    PWCHAR PseudoDeviceName;
    USHORT PseudoLinkNameLength;
    PWCHAR PseudoLinkName;
    DEVICE_TYPE PseudoDeviceType;
    ULONG  PseudoCharacteristics;
}OW_PSEUDO_MOUNT_INFO2;
```

PseudoDeviceType and PseudoCharacteristics will be copies into the newly created device object in the DeviceObject->DeviceType and DeviceObject->Characteristics fields respectively.

OW_FSCTL_DISMOUNT_PSEUDO

This IOCTL is used to dismount a previously mounted pseudo file system volume. The data structure associated with this request is:

```
typedef struct {
    HANDLE VolumeHandle;
    BOOLEAN Force;
} OW_PSEUDO_DISMOUNT_INFO;
```

The VolumeHandle is the handle of a FILE_OBJECT representing an open instance of the volume. Thus, this parallels the standard Windows mechanism for dismounting a physical media volume: open the volume (which yields a handle), dismount the volume, and close the volume.

Upon dismounting the volume all subsequent operations (except that final closure of the volume handle) are invalid. The device object and symbolic link are deleted as part of the pseudo dismount process.

OW_FSCTL_PSEUDO_VOLUME_READ

This IOCTL is used to call the FSD with a pseudo volume read. Since the read is to the volume, FS_FSCTL will be called by the FSDK with the following structure passed in the InputBuffer:

```
typedef struct {
    LARGE_INTEGER VolumeOffset;
    PMDL Buffer;
    ULONG BufferSize;
} OW_PSEUDO_VOLUME_IO_INFO;
```

The FSD must fill in the following structure in the OutputBuffer before returning to the FSDK.

```
typedef struct {
    ULONG Bytes;
} OW_PSEUDO_VOLUME_IO_COMPLETE;
```

The FSDK will deallocate both the InputBuffer and the OutputBuffer.

OW_FSCTL_PSEUDO_VOLUME_WRITE

This IOCTL is used to call the FSD with a pseudo volume write. Since the write is to the volume, FS_FSCTL will be called by the FSDK with the following structure passed in the InputBuffer:

```
typedef struct {
    LARGE_INTEGER VolumeOffset;
    PMDL Buffer;
    ULONG BufferSize;
} OW_PSEUDO_VOLUME_IO_INFO;
```

The FSD must fill in the following structure in the OutputBuffer before returning to the FSDK.

```
typedef struct {
    ULONG Bytes;
} OW_PSEUDO_VOLUME_IO_COMPLETE;
```

The FSDK will deallocate both the InputBuffer and the OutputBuffer.

OW_FSCTL_INIT

This value is provided to allow a network provider (such as the sample network provider in the FSDK) to initialize its underlying file system. The actual data transferred via this call is not defined by the FSDK and is private to the Network Provider and FSD.

The Wrapper does not implement this operation. Rather, it is passed along to the underlying FSD.

OW_FSCTL_ENUM

This value is provided to allow a network provider (such as the sample network provider in the FSDK) to enumerate potential remote network resources. The actual data transferred via this call is not defined by the FSDK and is private to the Network Provider and FSD.

The Wrapper does not implement this operation. Rather, it is passed along to the underlying FSD.

OW_FSCTL_CONNECT

The Network Provider uses this value when it wishes to create a new network connection. The Wrapper will create any necessary symbolic link, as well as initialize internal data structures so that calls to the specified drive letter will be handled correctly.

The connection to be made is described by the following data structure:

```
typedef struct {
    WCHAR   ConnectionIdentifier;
    UCHAR   ConnectionType;
    WCHAR   ConnectionName;
} OW_FSCTL_CONNECT_INFO, *POW_FSCTL_CONNECT_INFO;
```

The *ConnectionIdentifier* is used to specify information about this particular connection. For a disk drive, this connection identifier may be either null, indicating that no drive letter is associated with this connection, or it may be a character between 'C' and 'Z' indicating that there is an association between the Win32 drive letter and this connection. For a printer, this connection identifier must be a character between '1' and '9' indicating that there is an association between the Win32 LPT printer port and this connection.

The *ConnectionType* field indicates the specific type of connection being created. The values defined are:

OW_CONNECTION_DISK

OW_CONNECTION_PRINTER

The *ConnectionName* is a null-terminated wide character string that will be passed by the Wrapper to the FSD at its FS_CONNECT entry point. This information is used by the Wrapper to create a symbolic name in the Win32 address space, but the specific structure of this information is passed directly to the underlying FSD; hence it may contain any information necessary to establish the connection.

It is possible for the FSD's control application to pass additional information in the connect call. This is done by adding such information beyond the terminating NULL character for the *ConnectionName* field.

This works because the FSCTL operation indicates the size of the buffer. The full buffer is captured and passed from the FSDK to the FSD, although the FSDK only uses the null-terminated portion of the *ConnectionName* field. Data beyond the NULL terminator is not interpreted, but it is provided to the underlying FSD.

OW_FSCTL_DISCONNECT

This call is used by the Network Provider to delete an existing connection of any type. The Wrapper will clean up internal state and delete any symbolic links that exit to support this connection. The FSD will be called at its FS_DISCONNECT entry point as part of processing this request.

The following data structure is used to identify the volume to be disconnected:

```
typedef struct {
    WCHAR   ConnectionIdentifier;
    UCHAR   Force;
    WCHAR   ConnectionName;
} OW_FSCTL_DISCONNECT_INFO, *POW_FSCTL_DISCONNECT_INFO;
```

The *ConnectionIdentifier* is used to identify this particular connection. The specific interpretation and restrictions on this value are described in the OW_FSCTL_CONNECT section.

The *Force* parameter indicates if the connection should be deleted even if there are open, active references to the file in question. An "active reference" refers to a file that is opened by a user-level application. The Wrapper, as part of the disconnection sequence, closes files that remain open due to caching behavior of the virtual memory system. The following table summarizes the behavior of the Wrapper depending upon the *Force* parameter and whether or not there are any active references:

Force Value	Open Files	Result
FALSE	NO	Disconnect Successful
FALSE	YES	Disconnect Unsuccessful
TRUE	NO	Disconnect Successful
TRUE	YES	Disconnect Successful

The *ConnectionName* parameter is used if *ConnectionIdentifier* value is null in order to identify an unnamed connection.

OW_FSCTL_GETCONNECTIONS

This IOCTL is used by a network provider to enumerate the list of all network connections currently known to the Wrapper for the given file system. The returned information is described in a buffer encoded so that the first four bytes represent the number of entries to follow. Then each entry consists of two bytes representing the drive letter and an arbitrary number of bytes representing the name of the connection, with a wide character NULL terminator. The termination of the list is indicated with two contiguous wide character NULL terminators.

Note that as an alternative, the OW_FSCTL_GET_FULL_CONNECTIONS can be used to retrieve the identical information in a slightly different format.

OW_FSCTL_GET_FULL_CONNECTIONS

This IOCTL is used by the Network Provider to enumerate the list of all network connections that are currently known to the Wrapper for the given file system. The returned information is described by the following data structure:

```
typedef struct {
    ULONG NumberOfEntries;
    OW_FSCTL_CONNECT_INFO Connections[1];
} OW_FSCTL_GETCONNECTION_INFO, *POW_FSCTL_GETCONNECTION_INFO;
```

The variable length array of Connections is terminated when both ConnectionIdentifier and ConnectionName are the null character.

This implementation allows the caller to also determine the type of the connection in addition to the drive letter and name of the connection.

INDEX

L

M

N

P

R

S

U

V

W