FILE SYSTEM FILTER MANAGER

Filter Driver Development Guide

Version 1.0a

Summary

This specification includes, but is not limited to, the following topic areas:

- Description of the new 'File System Filter Manager' architecture and interfaces
- Aspects of the Windows 'Memory Manager', 'I/O Manager', and 'Cache Manager' interfaces that affect the development of file system filters using the new File System Filter Manager architecture.

Table Of Contents

1. Overview

This document pertains to filter drivers between the I/O manager and the base filesystem, which can be local or remote. It does *not* pertain to filter drivers that sit between the filesystem and the storage driver(s), such as FtDisk or DMIO.

We will refer to a file system filter driver written using the new model as a mini file system filter driver/minifilter.

The existing file system filters based on the sfilter sample – using IRP and device-object based filtering will be referred to as 'legacy filters'.

One of the key components of the new architecture is a legacy file system filter which is called 'Filter Manager'. In future Windows operating system releases, this driver will be installed by default on the system. This driver manages the minifilters by providing export libraries to which the minifilters link. The necessary header files, libraries and binaries are available in the Minifilter IFSKit.

Why write a mini file system filter driver?

- Simpler, get a more reliable filter driver with less development effort.
- Dynamic load and unload, attach and detach.
- Attach at a well-defined location in the filter stack.
- Context management: fast, clean, reliable contexts for file objects, streams, files, instances, and volumes
- A host of utility routines including support for looking up names and caching them for efficient access, communication between minifilters and user mode services, and IO queuing.
- Support non-recursive I/O so minifilter generated I/O can easily be seen only by lower minifilters and the file system.
- Filter only operations of interest to minifilter – unlike the legacy model where a filter has to hook every entry point to pass through operation.

2. Terms

In the Filter Manager architecture, some new objects are defined. For clarity these objects will be defined up front.

Filter: A driver that performs some type of filtering operation on the file system.

Volume: For local file systems, the object that represents logical volume the file system manages. For network redirectors, this represents the object to which all network requests are directed. The volume maps directly to the file system (local or remote) DEVICE_OBJECT in the legacy filter model.

Instance: An instantiation of a filter on a volume at a unique altitude. The Filter Manager manages presenting IOs to each instance in the stack of instances attached to a volume. There may be more than one instance of a given minifilter for a given volume. The canonical example is a filter like FileSpy. It would be helpful to attach FileSpy both above and below another filter on the same volume with each instance maintaining a private context. This context could contain the log of IOs seen by the instance for later comparison of the IO as seen above and below the filter.

File: A named object of data that could be composed of multiple streams that is stored on a disk by the file system.

Stream: Represents a physical stream of data in a file.

FileObject: Represents a user's open of a physical stream of data in a file.

CallbackData: The Filter Manager structure that encapsulates all the information about an operation. This is equivalent to an IRP in the legacy filter model.

3. Minifilter Installation

Minifilters will be installed via an INF file. The INF file indicates what instances the minifilter supports. Instances are defined in Section 5. Each instance is also accompanied by an altitude value which indicates its absolute position in a minifilter stack and a set of flags.

The list of instances with their altitudes in the INF is the preferred means for file system filter vendors to ship their minifilters. The flags indicate if the minifilter needs 'automatic attachment'. If this is the case, for every volume in the system (and for new ones as they arrive), the minifilter gets a notification that it can respond to and attach. The altitude in the INF file determines the altitude to which the minifilter attaches if it chooses to do so.

A file system filter vendor may also create an instance dynamically at a specified altitude at any time when the minifilter is running via the `FilterAttachAtAltitude()` function. This is provided as an override and a potential debugging and testing aid.

4. Minifilter Registration

Mini file system filter drivers are kernel-mode drivers. As such, they must export a function called `DriverEntry()`, which will be the first function invoked when the driver is loaded. Most minifilters will call `FltRegisterFilter()` in their `DriverEntry()`.

`FltRegisterFilter()` takes as a parameter a `FLT_REGISTRATION` structure, which contains an unload routine, instance notification callbacks, a list of context callback function pointers and a list of file system operation callback pointers. In the most common case, where the minifilter hooks just a few operations, this list can be very short.

The minifilter also can specify additional flags for each operation that controls whether it receives the callbacks in all cases. For instance, if `FLTFL_OPERATION_REGISTRATION_SKIP_PAGING_IO` is supplied, the minifilter will not receive any paging I/O operations for that particular IRP major code.

Similarly, if `FLTFL_OPERATION_REGISTRATION_SKIP_CACHED_IO` is specified, only the non-cached I/O path will be seen for that operation (i.e., if it is supplied for `IRP_MJ_READ`, all cached read paths will be skipped by that minifilter).

5. Initiating filtering

Once a minifilter registers itself, it should initiate filtering at some point by calling the function `FltStartFiltering()`. It is not necessary to call it from `DriverEntry()`, though most filters will probably prefer to.

This function will kick-off the necessary notifications that will result in the minifilter attaching to the appropriate volumes and start filtering I/O.

For this the Filter Manager walks through the list of instances that are registered by the minifilter via its INF.

Each instance specifies an "altitude". An altitude is an infinite precision string (e.g., "100.123456") that defines where it will be attached in a stack of minifilter instances. Altitudes for commercially released minifilter drivers will be assigned by Microsoft.

The larger the altitude in numeric value, the higher the instance is attached in the minifilter stack for a volume. Several sample altitudes are provided for developers to use while implementing minifilters and these are the only altitudes an unsigned driver may use. There are two purposes for altitudes. The first is to enforce relative minifilter ordering when it is necessary for proper functioning regardless of when the individual drivers load. For instance, an encryption filter and an antivirus filter must attach with the antivirus filter on top, otherwise the antivirus filter could not scan the clear text for viruses. The second purpose is to provide a smaller test matrix for minifilter interoperability testing. Each minifilter instance will have a well-specified altitude, therefore, when testing the interoperability between one or more minifilters, there is exactly one well-defined configuration to test.

An INF instance specification also has associated flags. If bit 1 is set, the minifilter will not be automatically notified when volumes appear in the system so that it can attach its instance. An instance with this flag set in

its specification would be manually attached via a Filter Manager API. If bit 2 is set, the minifilter will not be asked to attach this instance to a volume when a manual attachment request is made.

6. Instance Notification Support

A set of callbacks are provided to notify a minifilter when an instance is being created or torn down. Through these callbacks, the minifilter can control when instances are attached and detached from volumes.

6.1. Setting up an Instance

The `InstanceSetupCallback()` routine gets called for the following reasons:

- When a minifilter is loaded it is called for each volume that currently exists in the system.
- When a new volume is mounted.
- When `FltAttachVolume()` is called (kernel mode).
- When `FltAttachVolumeAtAltitude()` is called (kernel mode).
- When `FilterAttach()` is called (user mode).
- When `FilterAttachAtAltitude()` is called (user mode).

It is during the processing of the InstanceSetupCallback() that the minifilter decides whether or not the instance should be created on the volume specified. This callback has the following signature:

```
typedef NTSTATUS
(*PFLT_INSTANCE_SETUP_CALLBACK) (
    IN PCFLT_RELATED_OBJECTS FltObjects,
    IN FLT_INSTANCE_SETUP_FLAGS Flags,
    IN DEVICE_TYPE VolumeDeviceType,
    IN FLT_FILESYSTEM_TYPE VolumeFilesystemType
    );
```

The `FltObjects` structure contains pointers to the minifilter, volume and instance that define the instance being created by this `InstanceSetupCallback()`. The Flags explain what operation triggered this `InstanceSetupCallback()`:

- `FLTFL_INSTANCE_SETUP_AUTOMATIC_ATTACHMENT`: This is an automatic attachment that is occurring because the minifilter has just registered; therefore the Filter Manager is enumerating all existing volumes in the systems for the newly loaded minifilter. This flag is not set when a user explicitly request an instance to attach to a volume.
- `FLTFL_INSTANCE_SETUP_MANUAL_ATTACHMENT`: A manual attachment request has been issued via the `FilterAttach()` (user-mode), `FilterAttachVolumeAtAltitude()` (user-mode), `FltAttachVolume()` (kernel-mode), or `FltAttachVolumeAtAltitude()` (kernel-mode) call.
- `FLTFL_INSTANCE_SETUP_NEWLY_MOUNTED_VOLUME`: The file system has just mounted this volume, therefore it is calling the `InstanceSetupCallback()` for the minifilter to create an instance on the volume if desired.

During the InstanceSetupCallback(), the minifilter also receives the VolumeDeviceType and the VolumeFilesystemType to help the minifilter decide whether or not it is interested in attaching to this volume. From the InstanceSetupCallback(), a minifilter can query the volume properties by calling `FltGetVolumeProperties()` and set a context using `FltSetInstanceContext()` on this instance if it plans to attach this instance to the volume. It can also open and close files on the volume.

If the minifilter returns a success code from the instance setup callback, the instance will be attached to the given volume. If the minifilter returns a warning or an error code, the instance will **not** be attached to the volume.

If a minifilter does not register an `InstanceSetup()` routine, the system will treat this as if the user returned `STATUS_SUCCESS` and allow the instance to be created on the volume.

6.2. Controlling Instance Teardown

The `InstanceQueryTeardown()` callback is called only when a manual detach request is made. The following operations cause manual detach requests:

- `FltDetachVolume()` (kernel-mode)
- `FilterDetach()` (user-model)

If a minifilter does not provide this callback than manual detach is not supported. However, volume dismounts and minifilter unloads will still be allowed.

If this callback is defined and a success code is returned, Filter Manager will start tearing down the given instance. Eventually the instance's `InstanceTeardownStart()` and `InstanceTeardownComplete()` routines will be called. If the minifilter returns a warning/error status, the manual detach will fail. The recommended error code is `STATUS_FLT_DO_NOT_DETACH but any failure status may be returned.`

The signature for the `InstanceQueryTeardown()` callback is as follows:

```
typedef NTSTATUS
(*PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK) (
    IN PCFLT_RELATED_OBJECTS FltObjects,
    IN FLT_INSTANCE_QUERY_TEARDOWN_FLAGS Flags
    );
```

As in the `InstanceSetupCallback()`, the `FltObjects` specify the minifilter, volume and instance to which this `InstanceQueryTeardown()` operation is related.

6.3. Synchronizing Instance Detach

Once the decision to detach has been made the `InstanceTeardownStart()` callback is called. This routine must do the following things:

- Restart any pended I/O operations (both pre & post operations)
- Guarantee no new I/O operations will be pended
- Start doing minimal work on new operations as they arrive.

This routine may also do the following work:

- Close opened files
- Cancel filter initiated I/O's
- Stop queuing new work items

The minifilter then returns control back to the Filter Manager to continue its teardown processing.

After all operations in which this instance participated have drained or completed, the `InstanceTeardownComplete()` callback is called to allow the minifilter to do the final cleanup for the teardown of this instance. When `InstanceTeardownComplete()` is called, the Filter Manager guarantees that all existing operation callbacks have been completed for this instance. At this time, the minifilter must close any files it still has open that are associated with this instance.

The `InstanceTeardownStart()` and `InstanceTeardownComplete()` callbacks have the same signature:

```
typedef VOID
(*PFLT_INSTANCE_TEARDOWN_CALLBACK) (
    IN PCFLT_RELATED_OBJECTS FltObjects,
    IN FLT_INSTANCE_TEARDOWN_FLAGS Reason
    );
```

The `FltObjects` parameter describes the minifilter, volume and instance of interest for the callback. The Reason parameter describes the reason for this teardown callback and may be a combination of the following flag values:

- FLTFL_INSTANCE_TEARDOWN_MANUAL: This teardown operation is the result of a manual request to detach this instance (FilterDetach() or FltDetachVolume()).
- FLTFL_INSTANCE_TEARDOWN_FILTER_UNLOAD: This teardown operation is the result of the minifilter being unloaded and the minifilter could have chosen to fail the unload request.
- FLTFL_INSTANCE_TEARDOWN_MANDATORY_FILTER_UNLOAD: This teardown operation is the result of the minifilter being unloaded and the minifilter could **not** have chosen to fail the unload request.
- FLTFL_INSTANCE_TEARDOWN_VOLUME_DISMOUNT: This teardown request is the result of the volume being dismounted.
- FLTFL_INSTANCE_TEARDOWN_INTERNAL_ERROR: This teardown request is the result of some error that occurred while trying to setup the instance, like insufficient memory.

Notice that there is no return value — the InstanceTeardownStart() and InstanceTeardownComplete() cannot be failed. The Filter Manager guarantees that these callbacks will be called at passive IRQL.

7. Callback Support

7.1. Callback data

The callback data structure is the new I/O packet descriptor for the Filter Manager, and it is analogous to the IRP in the legacy model. Minifilters talk to Filter Manager and each other via this structure. Unlike an IRP however, minifilters do not manage stack locations but instead indicate how the callback data should be managed via well-defined Filter Manager interfaces and return status values to the Filter Manager.

The FLT_CALLBACK_DATA type describes all the information provided to a minifilter to describe an I/O. The following description goes through each field in this structure to describe the information it contains.

Flags: Provides information about this operation. One or more of the following flags could be set in a callback data structure:

- FLTFL_CALLBACK_DATA_IRP_OPERATION: This callback data structure represents an IRP operation.
- FLTFL_CALLBACK_DATA_FAST_IO_OPERATION: This callback data structure represents a FAST IO operation.
- FLTFL_CALLBACK_DATA_FS_FILTER_OPERATION: This callback data structure represents an FsFilter operation.
- FLTFL_CALLBACK_DATA_SYSTEM_BUFFER: The buffer for the operation described by this callback data is a system allocated buffer.
- FLTFL_CALLBACK_DATA_GENERATED_IO: This callback data represents an operation that was generated by another minifilter.
- FLTFL_CALLBACK_DATA_REISSUED_IO: This callback data represents an operation that has been sent back down to the file system by a minifilter instance above the current instance in the stack.
- FLTFL_CALLBACK_DATA_DRAINING_IO: Only set for a post-operation callback, this callback data represents an IO operation that is being drained so that the minifilter can unload.
- FLTFL_CALLBACK_DATA_POST_OPERATION: Only set for a post-operation callback, this callback data is currently in the post-operation processing for this I/O.
- FLTFL_CALLABACK_DATA_DIRTY: This is set after a minifilter has changed one or more of the changeable parameters for this operation. This flag is only set during pre-operation processing. Minifilters should use FLT_SET_CALLBACK_DATA_DIRTY() and FLT_CLEAR_CALLBACK_DATA_DIRTY() to manipulate this flag.

Thread: The address of the thread which initiated this operation.

Iopb: Pointer to the changeable parameters for this operation. This structure is described in more detail later in this document.

IoStatus: The `IO_STATUS_BLOCK` which will receive the final status for this operation. If a minifilter wants to complete an operation, the status should be set in this field before completing the operation. For operations that are passed through to the file system, this field contains the final status of the operation during post-operation processing.

TagData: This field is only valid in post-callback processing for CREATE operations when the target file of the operation has a reparse point set on it.

QueueLinks: The list entry structure used to add this callback data structure to a work queue if needed.

QueueContext[2]: An array of `PVOID` structures to allow additional context to be passed to the work queue processing routine.

FilterContext[4]: An array of PVOID structures that can be used as desired by a minifilter if this callback data has been queued in a manner that does not rely on the queuing infrastructure provided by the Filter Manager.

RequestorMode: The requestor mode of the caller who initiated this IO operation.

The `FLT_IO_PARAMETER_BLOCK` is the structure pointed to by the **Iopb** field of the callback data structure and contains the changeable portion of the callback data structure. To extend the IRP analogy, this is like the "current stack location" of the IRP. Minifilters should access this structure to retrieve I/O parameters in both pre- and post-callbacks. The following is a more detailed description of the fields this structure contains.

IrpFlags: The flags from the IRP which describe this operation. These flags begin with the IRP_ prefix.

MajorFunction: The IRP_MJ function which describes this operation.

MinorFunction: The IRP_MN function which describes this operation.

OperationFlags: The flag values in the IO_STACK_LOCATION.Flags field in the legacy filter model. These flags being with the SL_ prefix.

TargetFileObject: The file object which represents the file stream which this operation affects.

TargetInstance: The instance to which this IO is directed.

Parameters: The FLT_PARAMETERS union describes parameters specific to the operation specified in the MajorFunction and MinorFunction fields.

Minifilters may change any of the parameters in the `FLT_IO_PARAMETER_BLOCK` except the `MajorFunction` in the pre-callbacks. If parameters have been changed, the minifilter should call `FLT_SET_CALLBACK_DIRTY()` to indicate this change. More information regarding changing the parameters of an operation are in Section 8.

Minifilters will see the same parameters in both their pre- and post-callback routines for the same I/O — even though they or other minifilters below may have changed the parameters. This is guaranteed by the Filter Manager. Although the contents of the `FLT_IO_PARAMETER_BLOCK` will be the same in the pre- and post-callback routines for the same I/O, the address to the `FLT_IO_PARAMETER_BLOCK` may not be the same, therefore minifilters should not rely on this.

The callback data structure also contains the IO_STATUS_BLOCK that describes the status of the operation. A minifilter can change this value and it will be honored by the Filter Manager without marking the callback data dirty. The minifilter should set the status for the operation if it is completing the operation in its pre-operation callback or undoing the operation in the post-operation callback.

7.2. Pre-Operation Callbacks

All pre-operation callbacks have the same signature:

```
typedef FLT_PREOP_CALLBACK_STATUS
(*PFLT_PRE_OPERATION_CALLBACK) (
    IN OUT PFLT_CALLBACK_DATA Data,
    IN PCFLT_RELATED_OBJECTS FltObjects,
```

```
    OUT PVOID *CompletionContext
    );
```

All pre-operation callbacks return an FLT_PRE_OPERATION_CALLBACK_STATUS. The values are defined as follows:

- FLT_PREOP_SUCCESS_WITH_CALLBACK: The operation succeeded and the minifilter wants to have its post-operation callback called.
- FLT_PREOP_SUCCESS_NO_CALLBACK: The operation succeeded, but the minifilter does not want to have its post-operation callback called.
- FLT_PREOP_PENDING: The minifilter driver will complete the operation (by calling FltCompletePendedPreOperation()) sometime in the future. There is no need for the minifilter to call any special preparation routine before returning this status, like IoMarkIrpPending() in the legacy filter model. If this status is returned, all further processing on the I/O is suspended by the Filter Manager (i.e. no pre-callbacks of lower drivers are called etc.) until FltCompletePendedPreOperation() is called.
- FLT_PREOP_COMPLETE: The minifilter completed the operation. The minifilter sets the I/O status of the operation in Data->IoStatus.Status. Minifilters attached below this minifilter, legacy filters, and the base file system will not see this I/O. Minifilters attached above this minifilter will see the operation complete with the appropriate status. For CLEANUP and CLOSE operations, it is incorrect for a minifilter to complete these operations with a failure status since these operations cannot fail.
- FLT_PREOP_SYNCHRONIZE: This status is valid only for non-create operations (creates are automatically synchronized). A minifilter MUST have a post-callback for the operation when this status is returned. This indicates that the minifilter wants this I/O to be completed in the same thread context as the pre-operation process. The post-callback will be called after the I/O is completed in this thread context. Filter Manager guarantees this — regardless of whether a lower filter/file system may pend the I/O or abort completion processing etc. This status should be used with care because it requires Filter Manager tosynchronizes the entire I/O and this has negative performance impacts on the systems overall throughput.
- FLT_PREOP_DISALLOW_FAST_IO: This status is valid only for fast I/O operations that return BOOLEAN in the legacy model. This status indicates to I/O to retry the operation using the IRP path instead.

Filters may change the parameters for the I/O before returning from the pre-operation callback. However the only parameters that *may* be changed are in the Data->Iopb structure (FLT_IO_PARAMETER_BLOCK). When a minifilter changes any of the parameters in this structure, it needs to issue FLT_SET_CALLBACK_DATA_DIRTY() on the passed in CallbackData, or the changes will not be honored, and unpredictable failures may occur.

There are a couple exceptions to this. If a minifilter is completing the operation or changing the operations status, it may set the IoStatus appropriately in the callback data structure and it is not necessary to mark the callback data dirty for this change to be honored by the Filter Manager.

Another exception is during the post-operation processing of IRP_MJ_CREATE operations. When a reparse point is encountered, the Data->TagData will point to the reparse data buffer. If a minifilter wishes to change this buffer in any way, it can free the existing buffer and replace Data->TagData with a new buffer (or NULL) without calling FLT_SET_CALLBACK_DATA_DIRTY().

7.3. Post-Operation Callbacks

All post-operation callbacks have the same signature:

```
typedef FLT_POSTOP_CALLBACK_STATUS
(*PFLT_POST_OPERATION_CALLBACK) (
    IN OUT PFLT_CALLBACK_DATA Data,
    IN PCFLT_RELATED_OBJECTS FltObjects,
    IN PVOID CompletionContext,
    IN FLT_POST_OPERATION_FLAGS Flags
    );
```

The flag FLTFL_CALLBACK_DATA_POST_OPERATION is set in the Data->Flags field for post-operation callbacks.

If a minifilter returns FLT_PREOP_SUCCESS_WITH_CALLBACK from its pre-operation callback, it is guaranteed to receive exactly one completion callback per pre-operation callback. Post operations can return a status of type FLT_POSTOP_CALLBACK_STATUS. The values are:

- FLT_POSTOP_FINISHED_PROCESSING —The minifilter has completed its processing of the request and control should be returned to whoever initiated the I/O.
- FLT_POSTOP_STATUS_MORE_PROCESSING_REQUIRED — The minifilter has *not* completed its processing of the request, and will later complete the request using FltCompletePendedPostOperation().

Like completion routines in the legacy filter model, post-operation callback routines can be called at DPC. If a minifilter needs to do work once the operation is completed but this work cannot be performed at DPC, the minifilter can call FltDoCompletionProcessingWhenSafe(). This routine will queue this work to a worker thread only if it is necessary (i.e., we are at DPC_LEVEL) and it is safe to do so. This call can fail if the completion processing must be queued to a worker thread, but this is an I/O that cannot be queued (e.g., paging I/Os).

For filters who want to cancel an opened file in the post-operation callback, FltCancelFileOpen() is provided which does a cleanup and close on the specified FileObject on behalf of the minifilter. The minifilter should fill in an appropriate error status code and return FLT_POSTOP_FINISHED_PROCESSING from its post-operation callback.

When an instance is being torn down, the Filter Manager may call the post-operation callback before it is actually completed by the lower filters or file system. In this case, the flag FLTFL_POST_OPERATION_DRAINING will be set when the post-operation is called. Minimal information about the operation will be provided, so at this time, the minifilter should cleanup any operation context passed in from its pre-operation and return FLT_POSTOP_FINISHED_PROCESSING.

The operations defined are based on the IRP_MJ codes with the addition of some new codes to represent Fast I/O operations that have no IRP equivalents. The goal is to present IRP-based and Fast I/O operations in a consistent manner to the minifilter. There will be a flag in the callback data so that it can determine the difference between an Irp-based, FsFilter based, or Fast I/O based operation, but it will not have to register separate callbacks for operations that are similar (i.e., Reads, Writes, and Locking operations).

The operations for which pre- and post-operation callbacks can be provided are:

- All the existing IRP_MJ codes from IRP_MJ_CREATE to IRP_MJ_PNP
- IRP_MJ_FAST_IO_CHECK_IF_POSSIBLE
- IRP_MJ_NETWORK_QUERY_OPEN
- IRP_MJ_MDL_READ
- IRP_MJ_MDL_READ_COMPLETE
- IRP_MJ_PREPARE_MDL_WRITE
- IRP_MJ_MDL_WRITE_COMPLETE
- IRP_MJ_VOLUME_MOUNT
- IRP_MJ_VOLUME_DISMOUNT
- IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION
- IRP_MJ_RELEASE_FOR_SECTION_SYNCHRONIZATION
- IRP_MJ_ACQUIRE_FOR_MOD_WRITE
- IRP_MJ_RELEASE_FOR_MOD_WRITE
- IRP_MJ_ACQUIRE_FOR_CC_FLUSH
- IRP_MJ_RELEASE_FOR_CC_FLUSH

A few special notes for some of the IRP_MJ codes:

- The pre-operation callback for `IRP_MJ_CREATE` can not set or get file, stream, or streamHandle contexts as it is not yet determined at pre-create time what file or stream (if any) is going to be created.
- The post-operation callback for `IRP_MJ_CLOSE` will not be able to set or get contexts for file, stream, or stream handle, as the system-internal structures with which they are associated are freed before the post-close routine is called.
- `IRP_MJ_CLEANUP` and `IRP_MJ_CLOSE` must never be failed by the minifilter. They can be pended, passed on, or completed with success. It is illegal to return FLT_PREOP_COMPLETE and fill in a failure status in the IoStatus block.
- Post callbacks cannot fail, as the operation has already taken place. If the minifilter wishes to fail an operation from the post-operation callback, it must take the necessary action to undo the operation that has succeeded. For failing IRP_MJ_CREATE operations, the Filter Manager provides some help with the FltCancelFileOpen() routine to teardown the opened file object, but the filter is still responsible to restore any file contents that were lost by CREATE operation which overwrote the original file.

Any changes made to the operation parameters in the post-operation callbacks will not be honored by the Filter Manager.

8. Manipulating Callback Data Parameters

8.1. I/O Parameter Block

As mentioned earlier, the callback data encapsulates two important data structures for the I/O.

1.) The I/O Status Block: Data->IoStatus is used to indicate the status of the operation when the minifilter is completing the I/O by itself (or contains the status it can read in its post-callback if a filter/filesystem below completed the I/O)

2.) The I/O Parameter Block: Data->Iopb points to the parameters that are specific to that minifilter for the I/O.

This section will discuss accessing/changing the parameters in more detail.

The `MajorFunction` and `MinorFunction` in the `Iopb` indicate the IRP/FastIo/FsFilter major/minor function for the operation. The `MajorFunction` may not be changed by the filters – Filter Manager does not support it.

The `TargetFileObject` indicates the stream's file object that the I/O is targeted to. This can be changed by the minifilter (the minifilter must call FLT_SET_CALLBACK_DATA_DIRTY()) and will be honored.

The `TargetInstance` parameter is different from instance to instance as the I/O is passed down, and indicates the current instance. However, filters can change this to point to another instance that they own, at the same altitude on another volume. Again, FLT_SET_CALLBACK_DATA_DIRTY() must be called to have this change honored.

Minifilters are NOT allowed to change the `TargetInstance` to an instance on the same volume. For example consider a minifilter that has 2 instances on volume C:, one at altitude 200 called Sample_Instance_C_200, and another at altitude 100, called Sample_Instance_C_100

Assume it also has another instance, called Sample_Instance_D_200 at altitude 200 on volume D:

Now suppose the `IRP_MJ_READ` pre-callback for the instance at altitude 200 on C: is called (i.e., `TargetInstance` points to Sample_Instance_C_200.) The minifilter can change the TargetInstance parameter to point to Sample_Instance_D_200, in which case it is re-directing I/O to the D: volume. However it should not change it to Sample_Instance_C_100.

This is to prevent filters from illegally bypassing other filters sitting below them on the same volume.

The parameters structure in the I/O parameter block contains operation-specific parameters. Minifilters must use the operation-appropriate union to access them. For IOCTLs and FSCTLs, there are method-specific

unions based on the method of the control code. Minifilters should test for the method explicitly and use the appropriate union.

8.2. Accessing buffer/MDL

In the IRP world, buffers are passed to the drivers through a number of mechanisms. For device objects which supported 'neither' I/O – which is the most common for file systems – the buffer was available via `Irp->UserBuffer`. For device objects which only supported buffered I/O, the buffer was buffered by the IO Manager and supplied via `Irp->AssociatedIrp.SystemBuffer`. For device objects that supported direct I/O, the buffer was supplied via `Irp->MdlAddress`, a locked down MDL describing the buffer.

However, there are exceptions and certain IRPs passed the buffer directly through one of the stack location parameters. In that case it could come from either kernel memory or raw user memory, independent of the device object's I/O requirements. These buffers are not passed to the hardware; hence, they ignored the device object's I/O support.

There are also certain IRPs which are always buffered – such as IRP_MJ_QUERY/SET_INFORMATION.

For minifilters, the buffers are always passed through the appropriate operation-specific union. There is no 'common' place where they can grab the buffer/MDL address from. This was designed to eliminate the confusion with the location of the buffers.

The flag `FLTFL_CALLBACK_DATA_SYSTEM_BUFFER` is set in the callback data's flag field if the operation is buffered. If this flag is set, it indicates that the buffer is from non-paged pool.

If the flag is not set, it means the IO Manager is not buffering the operation. However it is possible that a minifilter switched the buffers (see next section), so the buffer could still be from one of the system pools. However if the flag is not set, filters should always assume that the buffer is a raw user-buffer. We will discuss a rule in the next section that requires that a MDL describing the locked down pages will always be present if the buffer is not a user-buffer in such a case, and that should be used by the caller to obtain a system address to the buffer.

Finally, for those filters which wish to locate the buffer/length/MDL for the most common operations, `FltDecodeParameters()` is provided which does a fast lookup and returns the pointer to the buffer /MDL/length fields in the Parameter structures. For operations for which buffers are not applicable, this routine returns `STATUS_INVALID_PARAMETER`.

```
NTSTATUS
FLTAPI
FltDecodeParameters(
    IN PFLT_CALLBACK_DATA CallbackData,
    OUT PMDL **MdlAddressPointer OPTIONAL,
    OUT PVOID  **Buffer OPTIONAL,
    OUT PULONG *Length OPTIONAL,
    OUT LOCK_OPERATION *DesiredAcces OPTIONAL
    );
```

The `DesiredAccess` field indicates the kind of access to the buffer the minifilter can assume. For instance, for IRP_MJ_READ, the `DesiredAccess` can indicate IoWriteAccess, implying that the buffer can be written to. For IRP_MJ_WRITE, it's usually IoReadAccess indicating the minifilter can potentially only read the contents of the buffer but not modify it, since it is legal for an application to issue a write with a buffer that has read/only access page-level protection.

Filters that wish to ensure the buffer is locked down, should follow these guidelines: if the `FLTFL_CALLBACK_DATA_SYSTEM_BUFFER` flag is set, they may assume that the buffer is already locked down and access it safely.

If it is not set they should call `FltLockUserBuffer()` to lock the pages down. This API will ensure that the pages are locked down with the right access based on the operation, and if successful, it will set the MdlAddress field in the operation-specific parameter portion, to the MDL describing the pages.

8.3. Swapping buffers

Certain minifilters need to swap the supplied buffer for certain operations. Consider a minifilter that implements custom encryption. On a non-cached IRP_MJ_READ, it normally wishes to decrypt the contents of the buffer that was read from the filesystem. Similarly on a write, it wishes to encrypt the contents. Take the latter case: the contents cannot be encrypted in place, because for IRP_MJ_WRITE, the maximal access to the buffer that the minifilter can assume is IoReadAccess.

Hence the minifilter needs to supplant its own buffer which has read/write access, encrypt the contents of the original buffer into the new one, and send the I/O down.

For this scenario, the Filter Manager supports buffer-switching. However there are a few rules to which the minifilter must adhere:

1. Minifilters that switch a buffer must supply a post-callback. This is so that the buffer can be switched back by Filter Manager automatically.
2. If the minifilter is switching the buffer for an operation for which `FLTFL_CALLBACK_DATA_SYSTEM_BUFFER` flag was set, it MUST ensure that the new buffer is from non-paged memory (i.e. either from non-paged pool or from locked down memory).
3. If the above flag was not set, minifilter must adhere to the requirements of the device (by looking at the DeviceObjectFlags etc. in the volume properties) when supplanting the buffer, based on the operation. For instance if it supports direct I/O, it must supply a MDL etc.
4. If the minifilter supplants a non-paged pool buffer for an operation for which the `FLTFL_CALLBACK_DATA_SYSTEM_BUFFER` flag is not set, it must also build a MDL via `MmBuildMdlForNonPagedPool()` and supply it in the MdlAddress field. This is so that a filter/filesystem below will not attempt to lock non-paged pool (which can assert on checked Windows builds, and is also not good from a performance perspective). If a MDL is supplied, filters/file systems will always access the buffer through the MDL (by obtaining a system address for it).
5. When switching a buffer, the minifilter should also switch the MDL (i.e. the buffer and the MDL must be in sync). It is fine to leave the MDL NULL subject to the usual direct I/O exceptions.
6. Minifilter should NOT free the old buffer/old MDL.
7. Minifilter should NOT attempt to switch back the old buffer /MDL in its post-callback. Filter Manager does this automatically. In fact the buffer/MDL the minifilter sees in the Iopb in its post-callback are the original ones. Minifilters can remember the switched buffer by passing it in via their completion context.
8. Minifilters are expected to free the buffer they allocated (and supplanted) in the post-callback. Filter Manager however, will automatically free the MDL for the swapped buffer if any.
9. Minifilters that do not want Filter Manager to automatically free the MDL for the swapped buffer can call `FltRetainSwappedBufferMdl()`.
10. Minifilters that wish to access the swapped buffer's MDL can use `FltGetSwappedBufferMdl()` in the post-callback. Since a filter/filesystem below the minifilter that swapped the new buffer in, may potentially create a MDL for it, Filter Manager saves any such MDL for the swapped buffer before calling the post-callback for the minifilter that swapped the buffers. This API can be used to access the MDL in that case.

9. Context Support

All filters need to track their own state as different objects in the system are being operated on. One of the features of the Filter Manager is the ability for it to track these contexts on behalf of a minifilter.

A context is a piece of dynamically allocated memory that is created by calling `FltAllocateContext()`. The caller specifies the amount of memory to allocate and is returned a pointer to this memory. The system attaches an internal header used for tracking the context in front of the returned pointer.

Contexts can be created for the following types of objects:

- **Volume** — This represents a mounted device in the system.
- **Instance** — This is a minifilter's attachment to a given volume. A minifilter may attach more then once to a given volume. If a minifilter supports only once instance per volume, it is recommended that instance contexts be used instead of volume contexts because they are more efficient.
- **File** — This represents all opens across all data streams of a file. Currently these contexts are not supported.
- **Stream** — This represents all opens across a single data stream of a file.
- **StreamHandle** — This represents a single open of a file, i.e., a file object.

9.1. Context Registration

At registration time, a minifilter defines the types of contexts it will use, the size of the context and the routine used to cleanup that context. The minifilter uses an array of `FLT_CONTEXT_REGISTRATION` structures to define these parameters.

The `FLT_CONTEXT_REGISTRATION` structure is explained in detail below:

ContextType: The type of context this context registration describes.

Flags: The flags to denote special handling for this context. The flags currently defined are:

- `FLTFL_CONTEXT_REGISTRATION_NO_EXACT_SIZE_MATCH`: By default, the Filter Manager matches exactly a given context allocation request with a size specified at context registration time. If this flag is specified, Filter Manager will use the allocation routine specified if the requested allocation size is less than or equal to the size specified in the registration entry. This flags is ignored for entries where the size is specified as `FLT_VARIABLE_SIZED_CONTEXTS` or allocation and free routines are specified.

CleanupContext: The routine to be called when the Filter Manager determines that it is time to cleanup this context. This may be NULL if no cleanup is required before this context is freed.

Size: The size in bytes for this context. This is used to allow the Filter Manager to use pooling techniques (such as look aside lists) to make the allocation and freeing of these structures more efficient. This field is ignored if Allocate and Free routines are specified.

PoolTag: The pool tag the minifilter would like to associate with context allocations of this type. This field is ignored if Allocate and Free routines are specified.

Allocate: This should be set to a non-NULL value if the minifilter would like to specify its own allocation routine. If the minifilter would like to rely on the Filter Manager's pooling techniques for allocation, this should be set to NULL.

Free: This should be set to a non-NULL value if the minifilter would like to specify its own free routine.

If a minifilter has contexts of the same type that may be variable sizes, it can register up to 3 different `FLT_CONTEXT_REGISTRATION` structures for the same context type to make use of the Filter Manager supported memory pooling support.

9.2. Context Creation APIs

Following is the prototype definition for `FltAllocateContext()`:

```
NTSTATUS
FLTAPI
FltAllocateContext (
    IN PFLT_FILTER Filter,
    IN FLT_CONTEXT_TYPE ContextType,
    IN SIZE_T ContextSize,
    IN POOL_TYPE PoolType,
    OUT PFLT_CONTEXT *ReturnedContext
    );
```

The ContextType can be one of the following for a context on the associated object: FLT_VOLUME_CONTEXT, FLT_INSTANCE_CONTEXT, FLT_FILE_CONTEXT, FLT_STREAM_CONTEXT, or FLT_STREAMHANDLE_CONTEXT.

The following set context routines are used to attach a context to an object. The context being attached must match the type of the object.

```
NTSTATUS
FLTAPI
FltSetVolumeContext (
    IN PFLT_VOLUME Volume,
    IN FLT_SET_CONTEXT_OPERATION Operation,
    IN PFLT_CONTEXT NewContext,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltSetInstanceContext (
    IN PFLT_INSTANCE Instance,
    IN FLT_SET_CONTEXT_OPERATION Operation,
    IN PFLT_CONTEXT NewContext,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltSetFileContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    IN FLT_SET_CONTEXT_OPERATION Operation,
    IN PFLT_CONTEXT NewContext,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltSetStreamContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    IN FLT_SET_CONTEXT_OPERATION Operation,
    IN PFLT_CONTEXT NewContext,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltSetStreamHandleContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    IN FLT_SET_CONTEXT_OPERATION Operation,
    IN PFLT_CONTEXT NewContext,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );
```

When a context is being set there are 2 types of operations that can occur. They are:

- `FLT_SET_CONTEXT_KEEP_IF_EXISTS`: If there is no existing context, the new context will be set. If there is an existing context the new context will **not** be set and an error will be returned. If the OldContext parameter is defined the existing context will be returned. The caller must release the returned context if any. If the new context was not set the caller must also release that context.
- `FLT_SET_CONTEXT_REPLACE_IF_EXISTS`: This will set a new context even if a context already exists. If the OldContext parameter is defined the context being replaced will be returned. The caller must release the returned context. If the OldContext parameter is not defined then the old context will be released by the filter manager.

While the Filter Manager will call the minifilter at the appropriate time to cleanup a context because the object to which it is associated has been freed by the operating system, the minifilter may want to delete a context on an object at some other time. To do this, the minifilter should call one of the following routines to delete the appropriate context if they already have the pointer to the context:

```
VOID
FLTAPI
FltDeleteContext (
    IN PFLT_CONTEXT Context
    );
```

If the minifilter does not have the context, it can also delete the context by specifying the object to which the context is attached using the appropriate API below:

```
NTSTATUS
FLTAPI
FltDeleteVolumeContext (
    IN PFLT_FILTER Filter,
    IN PFLT_VOLUME Volume,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltDeleteInstanceContext (
    IN PFLT_INSTANCE Instance,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltDeleteFileContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltDeleteStreamContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );

NTSTATUS
```

```
FLTAPI
FltDeleteStreamHandleContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *OldContext OPTIONAL
    );
```

If the option `OldContext` parameter is NULL, the Filter Manager will release the residual reference on the context. Otherwise, the context will be returned via the `OldContext` parameter and the minifilter is responsible for calling `FltReleaseContext()` on this context when it has finished using it.

9.3. Context Retrieval APIs

The following routines can be used to retrieve a context for an individual object. The caller must release the returned context by calling `FltReleaseContext()` when it is done using it. Contexts can not be retrieved at DPC level so if a context is needed in a post-operation callback it may need to be passed from the pre-operation callback.

```
NTSTATUS
FLTAPI
FltGetVolumeContext (
    IN PFLT_FILTER Filter,
    IN PFLT_VOLUME Volume,
    OUT PFLT_CONTEXT *Context
    );

NTSTATUS
FLTAPI
FltGetInstanceContext (
    IN PFLT_INSTANCE Instance,
    OUT PFLT_CONTEXT *Context
    );

NTSTATUS
FLTAPI
FltGetFileContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *Context
    );

NTSTATUS
FLTAPI
FltGetStreamContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *Context
    );

NTSTATUS
FLTAPI
FltGetStreamHandleContext (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    OUT PFLT_CONTEXT *Context
    );
```

The following routine is used to release a context when a minifilter is done using it. It is advised that a minifilter not hold on to contexts across operations. Context tracking is very efficient and it is intended that a minifilter query for desired contexts on each operation.

```
VOID
FLTAPI
FltReleaseContext (
    IN PFLT_CONTEXT Context
    );
```

As with the Instance notification routines, each operation callback routine receives a `FLT_RELATED_OBJECTS` structure. This structure contains all known objects for the given operation. To simplify context retrieval there is an equivalent `FLT_RELATED_CONTEXTS` which can be used to retrieve more then one context at a time. This structure has the following definition:

```
typedef struct _FLT_RELATED_CONTEXTS {

    PFLT_CONTEXT VolumeContext;
    PFLT_CONTEXT InstanceContext;
    PFLT_CONTEXT FileContext;
    PFLT_CONTEXT StreamContext;
    PFLT_CONTEXT StreamHandleContext;

} FLT_RELATED_CONTEXTS, *PFLT_RELATED_CONTEXTS;
```

The following two routine are used to get multiple contexts at once as well as release multiple contexts at once. For `FltGetContexts()` the caller specifies (with the `DesiredContexts` parameter) which contexts are wanted. Internally, there are performance advantages to getting multiple contexts at once versus getting each context individually. Of course, if the minifilter does not need a context it is best not to request it. The `FLT_ALL_CONTEXTS` definition can be used to return all available contexts.

```
VOID
FltGetContexts (
    IN PFLT_RELATED_OBJECTS FltObjects,
    IN FLT_CONTEXT_TYPE DesiredContexts,
    OUT PFLT_RELATED_CONTEXTS Contexts
    );

VOID
FltReleaseContexts (
    IN OUT PFLT_RELATED_CONTEXTS Contexts
    );
```

9.4. Context Freeing APIs

When the Filter Manager determines it is time for a context to be freed a minifilter supplied callback routine is called. This routine should do any necessary cleanup on the contents of the given context structure (cleanup other allocated memory, free resource, etc.). Upon returning from this routine the Filter Manager will free the passed in context structure.

A separate cleanup routine is defined for each type of context. These routines have the following definition:

```
typedef VOID
(*PFLT_CONTEXT_CLEANUP_CALLBACK) (
    IN PFLT_CONTEXT Context,
    IN FLT_CONTEXT_TYPE ContextType
    );
```

The same context free routine may be registered for multiple types of contexts.

10. User Mode Communication

10.1. Filter Communication Port Object

To implement security and enable multiple communication channels, a new object has been introduced called a minifilter communication port. It is intended to be used for kernel-user communication and vice versa. Kernel-kernel communication is not presently supported. A port is a named NT object, with a security descriptor.

Filter Manager creates a new object type, FilterConnectionPort, to facilitate this. Filter Manager creates this new object type during its driver entry initialization before any minifilters are loaded.

Only kernel-mode drivers can create a minifilter communication port, with the following API:

```
NTSTATUS
FltCreateCommunicationPort(
    IN PFLT_FILTER Filter,
    OUT PHANDLE PortHandle,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN PVOID ServerPortCookie OPTIONAL,
    IN PFLT_CONNECT_NOTIFY ConnectNotifyCallback,
    IN PFLT_DISCONNECT_NOTIFY DisconnectNotifyCallback,
    IN PFLT_MESSAGE_NOTIFY MessageNotifyCallback,
    IN ULONG MaxConnections
    );
```

The `Filter` is the handle to the filter object of the minifilter that is creating this communication port object. The handle created by a successful call to `FltCreateCommunicationPort()` is returned in the `PortHandle` parameter.

As with other NT objects, the `ObjectAttributes` parameter defines the `OBJECT_ATTRIBUTES` structure to initialize the name, object attribute flags and security descriptor for the new communication port object being created. Note that for the communication port objects, the object attribute `OBJ_KERNEL_HANDLE` must be specified since this object can only be created as kernel objects.

The `ServerPortCookie` is a context that minifilters can associate with the port and this context is opaque to the Filter Manager. All connect/disconnect notifications to the minifilter for this port will be passed this cookie. This is useful for minifilters which create an array of ports since the cookie can be used to identify which port in the array is being accessed without defining unique port notification routines for each port in the array.

The caller also registers three callbacks when a communication port is created:

- `ConnectNotifyCallback()`: This will be called whenever a user mode process tries to open the port. The minifilter will have the option to fail the connection request. The notify routine will receive a handle that the minifilter must use for communicating on this connection. Each connection has a unique handle. The `ServerPortCookie` associated with the server port will also be passed in. The minifilter may fill in the `ConnectionCookie` argument with a context that will be passed in to all further user-kernel messages and the disconnect routine for this connection.
- `DisconnectNotifyCallback()`: This will be called when the port is closed by user mode (i.e. the handle count goes to zero).
- `MessageNotifyCallback()`: This is called whenever a message is received.

The `MaxConnections` specifies the maximum number of outstanding connections that will be allowed on this communication port. There is no default value, but this value must be greater than 0.

There is no guarantee that the object name will be created in the root of the name space – it's possible that the Filter Manager may map it under the `\FileSystem\Filters` object directory internally. If such a mapping occurs, it will be transparent to the minifilter and the user-mode application. When referencing the communication port by name, both components should use the same name. The Scanner minifilter example show how this is done.

In association with the new object type, new access types will be introduced for objects of this type. The minimal new access types are:

- `FLT_PORT_CONNECT`
- `FLT_PORT_ALL_ACCESS`

These are the access types that callers of this API can grant to users when constructing the security descriptor for the object via `InitializeObjectAttributes()`.

At the moment `FLT_PORT_CONNECT` will be sufficient to let user mode connect to the port and both send and receive messages.

When a minifilter creates the port, it implicitly starts listening on the port for incoming connections. It continues to listen until it closes the handle to the port by calling `ZwClose()`.

10.2. Connecting to Communication Port from User Mode

The minifilter communication port model – like the legacy model – is asymmetric. The kernel-mode end creates the communication port. The user-mode end connects to it. There is an API to open the port for the user-mode application. When a connection is established, the `ConnectNotify()` routine is called notifying the minifilter of the connection creation request as well the minifilter end of the port handle that should be used to send messages across this connection.

The prototype of the user-mode API to connect to the port is:

```
HRESULT
FilterConnectCommunicationPort(
    IN LPWSTR lpPortName,
    IN DWORD dwOptions,
    IN LPVOID lpContext,
    IN WORD wSizeOfContext,
    IN LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    OUT HANDLE *hPort
);
```

The `lpPortName` is a pointer to a wide-character string that specifies the name of the port to which to connect. This name should be the same name used by the minifilter which created the port. The port name may start with a '\' – indicating it is in the root directory. Filter Manager may however open it relatively in the actual appropriate directory that hosts minifilter communication ports.

The `dwOptions` parameter is currently unused.

The `lpContext` is a pointer to an opaque parameter that will be passed to the minifilter's `ConnectNotify()` routine. This can be used to authenticate that the application requesting to create the communication channel is the expected application of the appropriate version. The `wSizeOfContext` specifies the size of `lpContext` in bytes.

The `lpSecurityAttributes` specifies the security attributes for the user end of the connection, and if the handle will be inherited.

If the call is not successful, the appropriate HRESULT will be returned.

The returned handle may be closed/duplicated etc via the usual APIs. It can also be associated with an I/O completion port.

When this API is called, a new kernel-mode only unnamed port object is created (of type `FilterCommunicationPort`) that is used to represent the connection. The minifilter will be notified by calling the `ConnectNotify()` routine it supplied in the create operation for the server port, and given the handle to the connection port which should be used for sending messages from the kernel-side.

Of course if the caller does not have sufficient access to the server port, or the maximum number of connections are exceeded this will fail appropriately.

10.3. Disconnecting from the Communication Port

When the user-mode calls `CloseHandle()` on the handle obtained for the connection or the kernel-mode side calls `ZwClose()` on the connection port handle, the connection is broken.

The minifilter's `DisconnectNotify()` routine will be called only when the user mode closes its handle.

Ideally the minifilter should always close its end of the connection in the `DisconnectNotify()` routine. If a minifilter closes the handle at other times, it should make sure the handle will not be double-closed in its `DisconnectNotify()` by using some sort of synchronization.

When a communication port is disconnected on either the kernel or user-mode end:

1. Any user-mode waiters (via `FilterGetMessage()`) are flushed out and completed with `STATUS_FLT_PORT_DISCONNECTED` (which translates to win32 error `ERROR_DISCONNECTED`)
2. Any kernel mode senders that are blocked on waiters are woken up and completed with `STATUS_FLT_PORT_DISCONNECTED`
3. The port is 'invalidated' so no more futures waiters/senders will be allowed.

The minifilter can always close the server port by calling `ZwClose()` on the server port handle. This does not force existing connections to break, but simply stops the minifilter from accepting any more connections.

10.4. Unload issues

Minifilters must **always** close the server side port handle in their `FilterUnload()` routines (or earlier) before they call `FltUnregisterFilter()` to avoid system hangs during the unload process.

Minifilters will be permitted to unload even though there are connections open (i.e. the user-mode side has handles open to the connections). If this is the case, the connections will be attempted to be forcibly terminated by Filter Manager. The Filter Manager will call the `DisconnectNotify()` routine. The minifilter is expected to close the kernel side handle to the connection in the `DisconnectNotify()` thereby preventing a handle leak when the minifilter unloads.

11. File Name Support

The Filter Manager provides library routines that retrieve the name of the object in the current operation through looking at the operation parameters or querying the file system. For improved efficiency, the Filter Manager also caches the names as they are retrieved so that the cost of generating a file name can be amortized among all minifilters interested in that name.

The Filter Manager name APIs return names in `FLT_FILE_NAME_INFORMATION` structures so as to avoid data copies when a minifilter requests a name. These structures are reference counted and possibly shared by multiple minifilters requesting names. Only the Filter Manager APIs should ever change the data in these structures. There is more information about these structures in the following sections.

11.1. Retrieving a File Name during an Operation

To retrieve a file name for the `CallbackData->Iopb->TargetFileObject` for the current operation, the minifilter should call the following routine:

```
NTSTATUS
FLTAPI
FltGetFileNameInformation (
    IN PFLT_CALLBACK_DATA CallbackData,
    IN FLT_FILE_NAME_FORMAT NameFormat,
    IN FLT_FILE_NAME_QUERY_METHOD QueryMethod,
    OUT PFLT_FILE_NAME_INFORMATION *FileNameInformation
    );
```

The `CallbackData` is the `FLT_CALLBACK_DATA` structure for the operation on the FileObject for which the minifilter wants to query the name.

The NameFormat is one of the following three formats:

- FLT_FILE_NAME_NORMALIZED_FORMAT: A name requested in this format contains the full path for the name, including the volume name. All short names in the path have been expanded to their long name. Any stream name component will have any trailing ":$DATA" removed. If this is the name for a directory other than the root directory, the final '\' in the path will be removed.
- FLT_FILE_NAME_OPENED_FORMAT: A name requested in this format contains a full path for the name, including the volume name, but the name is in the same format that the caller used to open this object. Therefore, it could include short names for any of the components in the path.
- FLT_FILE_NAME_SHORT_FORMAT: A name requested in this format contains the short name (DOS name) for only the final component of the name. The full path for this object is **not** returned.

The QueryMethod is one of the following:

- FLT_FILE_NAME_QUERY_DEFAULT: When looking for a name, the Filter Manager will look in the cache first to find the name, then, if possible, query the file system to retrieve the name requested.
- FLT_FILE_NAME_QUERY_CACHE_ONLY: When looking to fulfill a name request, the Filter Manager will only look in the name cache to find the name. If the name is not found, STATUS_FLT_NAME_CACHE_MISS will be returned.
- FLT_FILE_NAME_QUERY_FILE_SYSTEM_ONLY: When looking to fulfill a name request, the Filter Manager will not look in the name cache and query the file system for the name if possible.

The name is returned in the final parameter, FileNameInformation. This structure is a set of Unicode strings that share the same buffer. The various Unicode strings denote varying sections of the name.

```
typedef struct _FLT_FILE_NAME_INFORMATION {
    USHORT Size;
    FLT_FILE_NAME_FORMAT Format;
    FLT_FILE_NAME_PARSED_FLAGS NamesParsed;
    UNICODE_STRING Name;
    UNICODE_STRING Volume;
    UNICODE_STRING Share;
    UNICODE_STRING Extension;
    UNICODE_STRING Stream;
    UNICODE_STRING FinalComponent;
    UNICODE_STRING ParentDir;
} FLT_FILE_NAME_INFORMATION, *PFLT_FILE_NAME_INFORMATION;
```

When a file name information structure is returned from FltGetFileNameInformation(), the Name, Volume, and Share (for remote file names) will be parsed. If a minifilter needs the other names parsed, it should call FltParseFileNameInformation().

A minifilter can call FltGetFileNameInformation() at any point during its IO processing when it is executing an IRQL less than DPC. If the minifilter is requesting to query the name at a time when it is possible for the name query to cause the system to deadlock (e.g., while processing paging IO), the call will fail if the name is not found in the cache or the caller requested to only query the file system.

A minifilter can use FltGetFileNameInformationUnsafe() to query a name for a file object if it does not have a callback data to describe the current operation targeting this file object and the filter knows that this is a safe time to potentially query the file system to get a name. This routine cannot detect that a file system query could potentially deadlock the system and return a failure status as FltGetFileNameInformation() can.

When a minifilter is finished using the name, it should be released by calling:

```
FLTAPI
FltReleaseFileNameInformation (
    IN PFLT_FILE_NAME_INFORMATION FileNameInformation
    );
```

Filter Manager's name cache is designed to be efficient enough for minifilters to query the name during the operations it needs to process. The name cache manages the invalidation of names due to file or directory renames. Minifilters are isolated from the complex logic necessary to maintain a name cache, but minifilters still need to be aware that names can be invalidated by renames in the system. When a rename occurs, the Filter Manager purges any affected cached entries, but minifilters may have outstanding references to the now stale file name information structure. When all the references are released the stale file name information structure will be freed. If a minifilter queries a name on an object that has been renamed, the new name will be returned if possible.

11.2. Additional Support for File Names

The Filter Manager also provides an API to help minifilters retrieve the destination name for rename and hardlink creation operations:

```
NTSTATUS
FltGetDestinationFileNameInformation (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    IN HANDLE RootDirectory OPTIONAL,
    IN PWSTR FileName,
    IN ULONG FileNameLength,
    IN FLT_FILE_NAME_FORMAT NameFormat,
    IN FLT_FILE_NAME_QUERY_METHOD QueryMethod,
    OUT PFLT_FILE_NAME_INFORMATION *RetFileNameInformation
    );
```

This API should be used during a minifilters pre-operation callback for IRP_MJ_SET_INFORMATION, FileSetNameInformation or FileSetLinkInformation. The caller specifies the parameters from the operation, and the Filter Manager will return a `FLT_FILE_NAME_INFORMATION` structure that contains the destination name for the operation in the format requested. Just as with `FltGetFileNameInformation()`, the minifilter must call `FltReleaseFileNameInformation()` when it is finished using the name returned.

Name tunneling is another aspect of file names where filter commonly make mistakes. The Filter Manager provides the following API to detect and retrieve a new name when needed due to name tunneling:

```
NTSTATUS
FltGetTunneledName (
    IN PFLT_CALLBACK_DATA CallbackData,
    IN PFLT_FILE_NAME_INFORMATION FileNameInformation,
    OUT PFLT_FILE_NAME_INFORMATION *RetTunneledFileNameInformation
    );
```

Name tunneling will only affect a minifilter that is working with names in normalized format. If a minifilter needs a normalized name in its pre-operation callback for CREATE, rename or hardlink creation operations, it should call `FltGetTunneledName()` in its post-operation callback to validate that the name provided in the pre-operation callback was not affected by name tunneling in the file system. If name tunneling did occur, a new file name information structure is returned in `RetTunneledFileNameInformation`. The minifilter should use this name for its name processing and call `FltReleaseFileName()` on this structure when complete.

11.3. Interfaces for Name Providing Filters

If a filter is interested in providing a way to change the name space, it will need to register three name additional callbacks with the Filter Manager. These callbacks allow the name providing filter to be responsible for providing the name content for the `FLT_FILE_NAME_INFORMATION` structure that is returned when a higher filter calls `FltGetFileNameInformation` or `FltGetDestinationFileName`. The name providing filter will also be able to tell the Filter Manager whether or not the name being returned should be cached.

As a name provider, the filter needs to be able to return a name for a given file object in the opened name format. The filter manager will do the work of iterating through the components in the name and then call the name provider back to expand components as necessary to generate a normalized name if that format was requested. In the process of expanding all the components for a given path, the filter's name component normalization routine may be called more than once. The filter is allowed to provide a context that will be

passed to future calls of the name component normalization routine while normalizing the current file name path. At the end of all the processing, the filter will be asked to cleanup this context if it was returned.

When the name providing filter must provide a name, its PFLT_GENERATE_FILE_NAME routine will be called. The filter is given the parameters for this name query, such as the file object for this query, the filter's instance which is receiving this query, the callback data describing the operation if one is present, and the name query options which the caller requested be used. The filter then must generate the name

```
typedef NTSTATUS
(*PFLT_GENERATE_FILE_NAME) (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject,
    IN PFLT_CALLBACK_DATA CallbackData OPTIONAL,
    IN ULONG NameOptions,
    OUT PBOOLEAN CacheFileNameInformation,
    OUT PFLT_NAME_CONTROL FileName
    );

typedef NTSTATUS
(*PFLT_NORMALIZE_NAME_COMPONENT) (
    IN PFLT_INSTANCE Instance,
    IN CONST PUNICODE_STRING ParentDirectory,
    IN USHORT VolumeNameLength,
    IN CONST PUNICODE_STRING Component,
    IN OUT PFILE_NAMES_INFORMATION ExpandComponentName,
    IN ULONG ExpandComponentNameLength,
    IN OUT PVOID *NormalizationContext
    );

typedef VOID
(*PFLT_NORMALIZE_CONTEXT_CLEANUP) (
    IN PVOID *NormalizationContext
    );
```

If a filter which provides names needs to invalidate all cached entries that it provided, it can do so through the following API. If other filters are still using a `FLT_FILE_NAME_INFORMATION` structure provided by the name providing filter after the name provider has requested for the name to be invalidated, the memory will be freed once the `FLT_FILE_NAME_INFORMATION` structure's reference count goes to 0.,

```
NTSTATUS
FltPurgeFileNameInformationCache (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject OPTIONAL
    );
```

To ensure that a name providing filter is able to unload, the Filter Manager will be responsible for managing the caching and freeing of names that are generated by name providing filters.

The Filter Manager will do the work of initializing the `FLT_FILE_NAME_INFORMATION` structure returned to the caller of `FltGetFileNameInformation` or `FltGetDestinationFileName` based on the name returned by the name providing filter.

For efficiency reasons, the Filter Manager passes in a buffer in which the name provider is to put the name when its `PFLT_GENERATE_FILE_NAME` callback is called. This buffer is a `UNICODE_STRING` wrapped in a `FLT_NAME_CONTROL` structure which contains both public and private information. Before a filter tries to fill this buffer with data, it must check to see if the buffer is large enough for the data by calling:

```
NTSTATUS
FltCheckAndGrowNameControl (
    IN OUT PFLT_NAME_CONTROL NameCtrl,
    IN USHORT NewSize
    );
```

If this routine returns STATUS_SUCCESS, the buffer in the FLT_NAME_CONTROL structure is large enough to hold the name from the name provider.

12. Filter Initiated I/O

Certain minifilters need to perform I/O of their own. This I/O is only seen by minifilters below the current minifilter in the minifilter stack of the Volume. For instance, an anti-virus minifilter may wish to read a file before an open has completed. In the new minifilter model, a minifilter will be able to generate I/O in one of two ways: using general routines similar to today's routines, and using simplified routines that provide an interface similar to the ZwXxx routines.

The general routines that the system provides to minifilters for I/O generation are:

```
NTSTATUS
FLTAPI
FltAllocateCallbackData (
    IN PFLT_INSTANCE Instance,
    IN PFILE_OBJECT FileObject OPTIONAL,
    OUT PFLT_CALLBACK_DATA *RetNewCallbackData
    );

VOID
FLTAPI
FltPerformSynchronousIo (
    IN PFLT_CALLBACK_DATA CallbackData
    );

NTSTATUS
FLTAPI
FltPerformAsynchronousIo (
    IN PFLT_CALLBACK_DATA CallbackData,
    IN PFLT_COMPLETED_ASYNC_IO_CALLBACK CallbackRoutine,
    IN PVOID CallbackContext
    );
```

Using the general routines, a minifilter can call FltAllocateCallbackData() to allocate a CallbackData for the operation, and then fill in the appropriate parameters in the CallbackData for the desired operation. The minifilter then calls FltPerformSynchronousIo() or FltPerformAsynchronousIo() to actually initiate the I/O. The instance parameter should always be for the current instance doing the I/O operation.

In addition Filter Manager exports some common utility routines. Examples:

```
NTSTATUS
FLTAPI
FltCreateFile (
    IN PFLT_FILTER Filter,
    IN PFLT_INSTANCE Instance OPTIONAL,
    OUT PHANDLE   FileHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PLARGE_INTEGER AllocationSize OPTIONAL,
    IN ULONG FileAttributes,
    IN ULONG ShareAccess,
    IN ULONG CreateDisposition,
    IN ULONG CreateOptions,
    IN PVOID EaBuffer OPTIONAL,
```

```
    IN ULONG EaLength,
    IN ULONG Flags
    );
```

If `InstanceHandle` is omitted, the CREATE will be sent to the top of the stack (so the minifilter initiating the operation will itself see the I/O recursively.) This is discouraged unless absolutely necessary as it can cause deadlocks and stack overflows if minifilters misuse it.

If `InstanceHandle` is provided (this should always be your own instance), then the create is initiated with the minifilter just below the caller of this API, by passing all legacy minifilters above the Filter Manager and minifilters above the caller.

The FileHandle returned by this API can be used in all Zw* calls that accept a file handle as a parameter. If the Instance parameter is non-NULL in a call to `FltCreateFile()`, it is guaranteed that all future I/O (via the Zw APIs, `FltClose()` etc.), on this handle will only be seen by the instances **below** the `InitiatingInstance`.

`FltReadFile()` and `FltWriteFile()` are support routines to allow minifilters to generate IOs that are only seen by instances below them when they have only the FileObject to represent the file. These routines are analogous to "rolling an IRP" in the legacy filter model.

IMPORTANT NOTE: Filters need NOT use `FltReadFile()`/`FltWriteFile()` to initiate I/O on the handle returned by `FltCreateFile()`. For handles created via `FltCreateFile()`, the normal Zw*() APIs will be targeted to the correct instance relative to the `InitiatingInstance` specified.

```
NTSTATUS
FLTAPI
FltReadFile (
    IN PFLT_INSTANCE InitiatingInstance,
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER ByteOffset OPTIONAL,
    IN ULONG Length,
    OUT PVOID Buffer,
    IN FLT_IO_OPERATION_FLAGS Flags,
    OUT PULONG BytesRead OPTIONAL,
    IN PFLT_COMPLETED_ASYNC_IO_CALLBACK CallbackRoutine OPTIONAL,
    IN PVOID CallbackContext OPTIONAL
    );

NTSTATUS
FLTAPI
FltWriteFile (
    IN PFLT_INSTANCE InitiatingInstance,
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER ByteOffset OPTIONAL,
    IN ULONG Length,
    IN PVOID Buffer,
    IN FLT_IO_OPERATION_FLAGS Flags,
    OUT PULONG BytesWritten OPTIONAL,
    IN PFLT_COMPLETED_ASYNC_IO_CALLBACK CallbackRoutine OPTIONAL,
    IN PVOID CallbackContext OPTIONAL
    );
```

By default, all minifilter-initiated I/O is sent to the next attached instance for the given volume, bypassing any instances attached above the minifilter initiating the I/O.

Minifilter initiated I/O can be synchronous or asynchronous. When the I/O is asynchronous, the minifilter provides a callback routine that the system will call when the I/O is completed.

13. Rules for Unload/Unregister/Detach

Detaching means a minifilter instance is going to be destroyed. That minifilter will no longer be called for any operations on that volume (unless, of course, there's still another instance of that minifilter attached to that volume).

Unloading a minifilter means its code is no longer in memory. This will most often be done at system shutdown time and when a new version of a minifilter is being installed without shutting the system down.

A minifilter instance can be detached from a volume from within the minifilter (by calling `FltDetachVolume()`) but the more common method will be via the UI. A minifilter instance *can* be detached even when there is outstanding I/O. In that case, the minifilter's completion routine(s) will be called for any outstanding I/O operations with the flag `FLTFL_POST_OPERATION_DRAINING` set. The minifilter will *not* receive completion callbacks when those I/O operations actually complete.

When a minifilter instance is detached, the system will call the minifilter's context free routines for all outstanding contexts for files, streams, and stream file objects associated with that instance.

14. Support Routines

In addition to the interfaces already discussed, the Filter Manager provides a number of support routines to help minifilters perform their required tasks. Here's a list of some of the additional routines. Please look to the Filter Manager IFS Kit documentation for more detailed information on these routines:

Routines for name / object translation:

- `FltGetFilterFromName()`
- `FltGetVolumeFromName()`
- `FltGetVolumeInstanceFromName()`

Routines for volume, instance, device object translation:

- `FltGetVolumeFromInstance(), FltGetFilterFromInstance()`
- `FltGetVolumeFromDeviceObject()`
- `FltGetDeviceObject()`
- `FltGetDiskDeviceObject()`

Routines for accessing information on objects:

- `FltGetVolumeProperties()`
- `FltIsVolumeWriteable`
- `FltQueryVolumeInformation(), FltSetVolumeInformation()`
- `FltGetInstanceInformation()`
- `FltGetFilterInformation()`

Enumeration routines:

- `FltEnumerateFilters()`
- `FltEnumerateVolumes()`
- `FltEnumerateInstances()`
- `FltEnumerateFilterInformation()`
- `FltEnumerateInstanceInformationByFilter()`
- `FltEnumerateInstanceInformationByVolume()`
- `FltEnumerateVolumeInformation()`

Oplock routines:

- `FltInitializeOplock()`
- `FltUninitializeOplock()`
- `FltOplockFsctrl()`
- `FltCheckOplock()`
- `FltOplockIsFastIoPossible()`
- `FltCurrentBatchOplock()`

Directory Change Notification routines:

- `FltNotifyFilterChangeDirectory()`

Other:

- `FltGetRequestorProcess(), FltGetRequestorProcessId()`

15. Building a Minifilter

Everything needed to build an application that uses the minifilter architecture can be found in the Filter Manager IFS Kit. This includes:

- The full build environment needed to build a minifilter and a user-mode application that uses the user-mode Filter Manager interfaces
- An install package to install the Filter Manager components on a machine for development until the version of the OS which contains the Filter Manager is released by Microsoft
- Minifilter source code samples

All minifilters include the header FltKernel.h and link with FltMgr.lib. User mode filter components will include the header FltUser.h and link with FltLib.lib.