Table of Contents

Installable file systems driver design guide

4/26/2017 • 2 min to read • Edit Online

The file systems in Windows are implemented as file system drivers working above the storage system. Each of the file systems in Windows are designed to provide reliable data storage with varying features to meet the user's requirements. A comparison of features for each of the standard file systems in Windows is shown in File System Functionality Comparison. New for Windows Server 2012 is ReFS. ReFS is a file system with scalable large volume support and the ability detect and correct data corruption on disk.

Creating a new file system driver in addition to those supplied in Windows is likely unnecessary. File Systems and File System Filter Drivers can provide any customized behavior required to modify the operation of existing file systems.

File System Filter Driver Development

A file system filter driver intercepts requests targeted at a file system or another file system filter driver. By intercepting the request before it reaches its intended target, the filter driver can extend or replace functionality provided by the original target of the request. Examples of File Systems and File System Filter Drivers include anti-virus filters, backup agents, and encryption products.

File system filtering services are available through the Filter Manager in Windows. The Filter Manager provides a framework for developing File Systems and File System Filter Drivers without having to manage all the complexities of file I/O. The Filter Manager simplifies the development of third-party filter drivers and solves many of the problems with the existing legacy filter driver model, such as the ability to control load order through an assigned altitude. A filter driver developed to the Filter Manager model is called a minifilter. Every minifilter driver has an assigned altitude, which is a unique identifier that determines where the minifilter is loaded relative to other minifilters in the I/O stack. Altitudes are allocated and managed by Microsoft.

File System Filter Driver Certification

Certification information for File Systems and File System Filter Drivers is found in the Windows Hardware Certification Kit (HCK). Tests for File Systems and File System Filter Drivers are found in the Filter.Driver category of the HCK.

File System Filter Driver Developer Resources

To request an altitude allocation from Microsoft, send an e-mail asking for an altitude assignment for your minifilter. Follow the instructions in Minifilter Altitude Request to submit a request.

To obtain an ID for a filter driver that uses reparse points follow the steps in Reparse Point Request.

You can subscribe to the NTFSD newsgroup for details about developing file systems and filter drivers. The group is found at NT File System Drivers Newsgroup.

OSR's "Developing File Systems for Windows" seminar explores developing file systems and File Systems and File System Filter Drivers. See Training for IFS Developers.

In this section

This design guide includes the following sections:

- File System Drivers

- File System Filter Drivers
- File System Minifilter Drivers
- Network Redirector Drivers
- Security Considerations for File Systems
- Miscellaneous Information

File System Drivers

4/26/2017 • 1 min to read • Edit Online

In most situations, developing a full file system driver is not necessary. First consider developing a file system filter driver or a file system minifilter driver.

This section includes the following topics.

Installing a File System Driver

Using Extra Create Parameters with an IRP_MJ_CREATE Operation

Installing a File System Driver

4/26/2017 • 1 min to read • Edit Online

For Microsoft Windows XP and later operating systems, you should install your file system drivers by using an INF file and an installation application. (On Microsoft Windows 2000 and earlier operating systems, file system drivers were commonly installed by the Service Control Manager.)

In the future, INF-based installation is expected to meet Windows Hardware Certification Kit requirements for file system drivers. Note that "INF-based installation" means only that you will need to use an INF file to copy files and to store information in the registry. You will not be required to install your entire product by using only an INF file, and you will not be required to provide a "right-click install" option for your driver.

This section includes:

Creating an INF File for a File System Driver

The following sections in the File System Filter Drivers section regarding installing and uninstalling file system filter drivers is also applicable to file system drivers.

Using an INF File to Install a File System Filter Driver

Using an INF File to Uninstall a File System Filter Driver

Creating an INF File for a File System Driver

4/26/2017 • 9 min to read • Edit Online

The Windows Setup and Device Installer Services, known collectively as SetupAPI, provide the functions that control Windows setup and driver installation. The installation process is controlled by INF files.

A file system driver's INF file provides instructions that SetupAPI uses to install the driver. The INF file is a text file that specifies the files that must be present for your driver to run and the source and destination directories for the driver files. An INF file also contains driver configuration information that SetupAPI stores in the registry, such as the driver's start type and load order group.

For more information about INF files and how they are created, see Creating an INF File and INF File Sections and Directives. For general information about signing drivers, see Driver Signing.

You can create a single INF file to install your driver on multiple versions of the Windows operating system. For more information about creating such an INF file, see Creating INF Files for Multiple Platforms and Operating Systems and Creating International INF Files.

Starting with 64-bit versions of Windows Vista, all kernel-mode components, including non-PnP (Plug and Play) drivers such as file system drivers (file system, legacy filter, and minifilter drivers), must be signed in order to load and execute. For these versions of the Windows operating system, the following list contains information that is relevant to file system drivers.

- INF files for non-PnP drivers, including file system drivers, are not required to contain [Manufacturer] or [Models] sections.
- The **SignTool** command-line tool, located in the \bin\SelfSign directory of the WDK installation directory, can be used to directly "embed sign" a driver SYS executable file. For performance reasons, boot-start drivers must contain an embedded signature.
- Given an INF file, the **Inf2Cat** command-line tool can be used to create a catalog (.cat) file for a driver package. Only catalog files can receive WHQL logo signatures.
- With Administrator privileges, an unsigned driver can still be installed on x64-based systems starting with Windows Vista. However, the driver will fail to load (and thus execute) because it is unsigned.
- For detailed information about the driving signing process, including the driving signing process for 64-bit versions of Windows Vista, see Kernel-Mode Code Signing Walkthrough.
- All kernel-mode components, including custom kernel-mode development tools, must be signed. For more information, see Signing Drivers during Development and Test (Windows Vista and Later).

INF files cannot be used to read information from the registry or to launch a user-mode application.

After creating an INF file, you will typically write the source code for your setup application. The setup application calls user-mode setup functions to access the information in the INF file and perform installation operations.

To construct your own file system driver INF file, use the following information as a guide. You can use the ChkINF tool to check the syntax of your INF file.

An INF file for a file system driver generally contains the following sections.

- Version (required)
- DestinationDirs (optional but recommended)

- SourceDisksNames (required)
- SourceDisksFiles (required)
- DefaultInstall (required)
- DefaultInstall.Services (required)
- ServiceInstall (required)
- DefaultUninstall (optional)
- DefaultUninstall.Services (optional)
- Strings (required)

Version Section (required)

The **Version** section specifies the driver version information, as shown in the following code example.

```
[Version]
Signature   = "$WINDOWS NT$"
Provider    = %Msft%
DriverVer   = 08/28/2000,1.0.0.1
CatalogFile =
```

The following table shows the values that file system filter drivers should specify in the **Version** section.

ENTRY	VALUE
Signature	"$WINDOWS NT$"
Provider	In your own INF file, you should specify a provider other than Microsoft.
DriverVer	See **INF DriverVer directive**.
CatalogFile	Leave this entry blank. In the future, it will contain the name of a WHQL-supplied catalog file for signed drivers.

DestinationDirs Section (optional but recommended)

The **DestinationDirs** section specifies the directories where the file system driver files will be copied.

In this section and in the **ServiceInstall** section, you can specify well-known system directories by using system-defined numeric values. For a list of these values, see **INF DestinationDirs Section**. In the following code example, the value "12" refers to the Drivers directory (%windir%\system32\drivers).

```
[DestinationDirs]
DefaultDestDir = 12
ExampleFileSystem.DriverFiles = 12
```

SourceDisksNames Section (required)

The **SourceDisksNames** section specifies the distribution media to be used.

In the following code example, the **SourceDisksNames** section lists a single distribution media for the file system driver. The unique identifier for the media is 1. The name of the media is specified by the %Disk1% token, which is defined in the **Strings** section of the INF file.

```
[SourceDisksNames]
1 = %Disk1%
```

SourceDisksFiles Section (required)

The **SourceDisksFiles** section specifies the location and names of the files to be copied.

In the following code example, the **SourceDisksFiles** section lists the file to be copied for the file system driver and specifies that the files can be found on the media whose unique identifier is 1 (This identifier is defined in the **SourceDisksNames** section of the INF file.)

```
[SourceDisksFiles]
examplefilesystem.sys = 1
```

DefaultInstall Section (required)

In the **DefaultInstall** section, a **CopyFiles** directive copies the file system driver's driver files to the destination that is specified in the **DestinationDirs** section.

Note The **CopyFiles** directive should not refer to the catalog file or the INF file itself; SetupAPI copies these files automatically.

You can create a single INF file to install your driver on multiple versions of the Windows operating system. This type of INF file is created by creating additional **DefaultInstall**, **DefaultInstall.Services**, **DefaultUninstall**, and **DefaultUninstall.Services** sections for each operating system version. Each section is labeled with a *decoration* (for example, .ntx86, .ntia64, or .nt) that specifies the operating system version to which it applies. For more information about creating this type of INF file, see Creating INF Files for Multiple Platforms and Operating Systems.

In the following code example, the **CopyFiles** directive copies the files that are listed in the ExampleFileSystem.DriverFiles section of the INF file.

```
[DefaultInstall]
OptionDesc = %ServiceDesc%
CopyFiles = ExampleFileSystem.DriverFiles

[ExampleFileSystem.DriverFiles]
examplefilesystem.sys
```

DefaultInstall.Services Section (required)

The **DefaultInstall.Services** section contains an **AddService** directive that controls how and when the services of a particular driver are loaded.

In the following code example, the **AddService** directive adds the file system service to the operating system. The %ServiceName% token contains the service name string, which is defined in the **Strings** section of the INF file. ExampleFileSystem.Service is the name of the file system driver's **ServiceInstall** section.

```
[DefaultInstall.Services]
AddService = %ServiceName%,,ExampleFileSystem.Service
```

ServiceInstall Section (required)

The **ServiceInstall** section adds subkeys or value names to the registry and sets values. The name of the

ServiceInstall section must appear in an **AddService directive** in the **DefaultInstall.Services section**.

The following code example shows the **ServiceInstall** section for the file system driver.

```
[ExampleFileSystem.Service]
DisplayName    = %ServiceName%
Description    = %ServiceDesc%
ServiceBinary  = %12%\examplefilesystem.sys
ServiceType    = 2 ;    SERVICE_FILE_SYSTEM_DRIVER
StartType      = 1 ;    SERVICE_SYSTEM_START
ErrorControl   = 1 ;    SERVICE_ERROR_NORMAL
LoadOrderGroup = "File System"
AddReg         = ExampleFileSystem.AddRegistry
```

The **DisplayName** entry specifies the name for the service. In the preceding example, the service name string is specified by the %ServiceName% token, which is defined in the **Strings** section of the INF file.

The **Description** entry specifies a string that describes the service. In the preceding example, this string is specified by the %ServiceDesc% token, which is defined in the **Strings** section of the INF file.

The **ServiceBinary** entry specifies the path to the executable file for the service. In the preceding example, the value 12 refers to the Drivers directory (%windir%\system32\drivers).

The **ServiceType** entry specifies the type of service. The following table lists the possible values for **ServiceType** and their corresponding service types.

VALUE	DESCRIPTION
0x00000001	SERVICE_KERNEL_DRIVER (Device driver service)
0x00000002	SERVICE_FILE_SYSTEM_DRIVER (File system or file system filter driver service)
0x00000010	SERVICE_WIN32_OWN_PROCESS (Microsoft Win32 service that runs in its own process)
0x00000020	SERVICE_WIN32_SHARE_PROCESS (Win32 service that shares a process)

The **ServiceType** entry should always be set to SERVICE_FILE_SYSTEM_DRIVER for a file system driver.

The **StartType** entry specifies when to start the service. The following table lists the possible values for **StartType** and their corresponding start types.

VALUE	DESCRIPTION
0x00000000	SERVICE_BOOT_START
0x00000001	SERVICE_SYSTEM_START
0x00000002	SERVICE_AUTO_START

VALUE	DESCRIPTION
0x00000003	SERVICE_DEMAND_START
0x00000004	SERVICE_DISABLED

For detailed descriptions of these start types to determine which one is appropriate for your file system driver, see What Determines When a Driver Is Loaded.

Starting with x64-based Windows Vista systems, the binary image file of a boot-start driver (a driver that has a start type of SERVICE_BOOT_START) must contain an embedded signature. This requirement ensures optimal system boot performance. For more information, see Kernel-Mode Code Signing Walkthrough.

For information about how the **StartType** and **LoadOrderGroup** entries determine when the driver is loaded, see What Determines When a Driver Is Loaded.

The **ErrorControl** entry specifies the action to be taken if the service fails to start during system startup. The following table lists the possible values for **ErrorControl** and their corresponding error control values.

VALUE	ACTION
0x00000000	SERVICE_ERROR_IGNORE (Log the error and continue system startup.)
0x00000001	SERVICE_ERROR_NORMAL (Log the error, display a message to the user, and continue system startup.)
0x00000002	SERVICE_ERROR_SEVERE (Switch to the registry's LastKnownGood control set and continue system startup.)
0x00000003	SERVICE_ERROR_CRITICAL (If system startup is not using the registry's LastKnownGood control set, switch to LastKnownGood and try again. If startup still fails, run a bug-check routine. Only the drivers that are needed for the system to startup should specify this value in their INF files.)

The **LoadOrderGroup** entry must always be set to "File System" for a file system driver. This is different from what is specified for a file system filter driver or file system minifilter driver where the **LoadOrderGroup** entry is set to one of the file system filter load order groups. For more information about the load order groups that are used for file system filter drivers and file system minifilter drivers, see Load Order Groups for File System Filter Drivers and Load Order Groups and Altitudes for Minifilter Drivers.

The **AddReg directive** refers to one or more INF writer-defined **AddRegistry** sections that contain any information to be stored in the registry for the newly installed service.

Note If the INF file will also be used for upgrading the driver after the initial install, the entries that are contained in the **AddRegistry** section should specify the 0x00000002 (FLG_ADDREG_NOCLOBBER) flag. Specifying this flag preserves the registry entries in HKLM\CurrentControlSet\Services when subsequent files are installed. For example:

```
[ExampleFileSystem.AddRegistry]
HKR,Parameters,ExampleParameter,0x00010003,1
```

DefaultUninstall Section (optional)

The **DefaultUninstall** section is optional but recommended if your driver can be uninstalled. It contains **DelFiles** and **DelReg** directives to remove files and registry entries.

In the following code example, the **DelFiles** directive removes the files that are listed in the ExampleFileSystem.DriverFiles section of the INF file.

```
[DefaultUninstall]
DelFiles    = ExampleFileSystem.DriverFiles
DelReg      = ExampleFileSystem.DelRegistry
```

The **DelReg** directive refers to one or more INF writer-defined **DelRegistry** sections that contain any information to be removed from the registry for the service that is being uninstalled.

DefaultUninstall.Services Section (optional)

The **DefaultUninstall.Services** section is optional but recommended if your driver can be uninstalled. It contains **DelService** directives to remove the file system driver's services.

In the following code example, the **DelService** directive removes the file system driver's service from the operating system.

```
[DefaultUninstall.Services]
DelService = %ServiceName%,0x200
```

Note The **DelService** directive should always specify the 0x200 (SPSVCINST_STOPSERVICE) flag to stop the service before it is deleted.

Note There are certain classes of file system products that cannot be completely uninstalled. In this situation, it is acceptable to just uninstall the components of the product that can be uninstalled and leave installed the components of the product that cannot be uninstalled. An example of such a product is the Microsoft Single Instance Store (SIS) feature.

Strings Section (required)

The **Strings** section defines each %strkey% token that is used in the INF file.

For example, the file system driver defines the following strings in its INF file.

```
[Strings]
Msft        = "Microsoft Corporation"
ServiceDesc = "Example File System Driver"
ServiceName = "ExampleFileSystem"
ParameterPath = "SYSTEM\CurrentControlSet\Services\ExampleFileSystem\Parameters"
Disk1       = "Example File System Driver CD"
```

You can create a single international INF file by creating additional locale-specific **Strings.***LanguageID* sections in the INF file. For more information about international INF files, see Creating International INF Files.

Using Extra Create Parameters with an IRP_MJ_CREATE Operation

7/21/2017 • 1 min to read • Edit Online

Components of the operating system use extra create parameters (ECPs) to associate additional information with the **IRP_MJ_CREATE** operation on a file. Drivers can also use ECPs to process or associate additional information with the IRP_MJ_CREATE operation on a file in the following situations:

- When a kernel-mode driver calls the **FltCreateFileEx2** or **IoCreateFileEx** routine to create or open the file
- When a file system filter driver processes the **IRP_MJ_CREATE** operation for the file

The following sections describe how to define, attach, and use ECPs. The following sections also describe operating system-defined ECPs.

Attaching ECPs to IRP_MJ_CREATE Operations that a Kernel-Mode Driver Originated

7/21/2017 • 1 min to read • Edit Online

You must follow these steps to set up ECPs and attach the ECPs to an **IRP_MJ_CREATE** operation on a file:

1. Call the **FltAllocateExtraCreateParameterList** or **FsRtlAllocateExtraCreateParameterList** routine to allocate memory for an ECP_LIST structure. The operating system does not automatically free ECP_LIST structures. Instead, after the ECP_LIST structure is allocated, the minifilter driver must eventually call the **FltFreeExtraCreateParameterList** or **FsRtlFreeExtraCreateParameterList** routine to free ECP_LIST.
2. Call the **FltAllocateExtraCreateParameter** or **FsRtlAllocateExtraCreateParameter** routine to allocate paged memory pool for an ECP context structure and to generate a pointer to that structure.
3. Call the **FltInsertExtraCreateParameter** or **FsRtlInsertExtraCreateParameter** routine to insert ECP context structures into the ECP_LIST structure.
4. Call the **IoInitializeDriverCreateContext** routine to initialize an **IO_DRIVER_CREATE_CONTEXT** structure.
5. Define the **IO_DRIVER_CREATE_CONTEXT** structure. In this definition, point the **ExtraCreateParameter** member of **IO_DRIVER_CREATE_CONTEXT** to the ECP_LIST structure.
6. Call the **FltCreateFileEx2** or **IoCreateFileEx** routine to attach the ECPs to the **IRP_MJ_CREATE** operation on the file. In this call, pass a pointer to the **IO_DRIVER_CREATE_CONTEXT** structure to the *DriverContext* parameter.
7. Call the **FltFreeExtraCreateParameterList** or **FsRtlFreeExtraCreateParameterList** routine to free the ECP_LIST structure.

Using ECPs to Process IRP_MJ_CREATE Operations in a File System Filter Driver

7/21/2017 • 3 min to read • Edit Online

You can use ECPs in your file system filter driver to process **IRP_MJ_CREATE** operations. Your file system filter driver can call the routines in the following sections to retrieve, acknowledge, add, and remove ECPs for the **IRP_MJ_CREATE** operation. You can also determine the operating-system space from which the ECPs originated.

Retrieving ECPs

Your file system filter driver can follow these steps to retrieve ECPs for the **IRP_MJ_CREATE** operation:

1. Call the **FltGetEcpListFromCallbackData** or **FsRtlGetEcpListFromIrp** routine to retrieve a pointer to an ECP context structure list (ECP_LIST) that is associated with the create operation.

2. Perform either of the following operations:

   - Call the **FltGetNextExtraCreateParameter** or **FsRtlGetNextExtraCreateParameter** routine to retrieve a pointer to the next (or first) ECP context structure in the ECP list.
   - Call the **FltFindExtraCreateParameter** or **FsRtlFindExtraCreateParameter** routine to search the ECP list for an ECP context structure of a given type. Either routine returns a pointer to the ECP context structure if the structure is found.

Setting ECPs

To set ECPs for the **IRP_MJ_CREATE** operation, your file system filter driver can first either retrieve an existing ECP context structure list (ECP_LIST) that is associated with the create operation, or allocate ECP_LIST and an ECP context structure and insert the ECP context structure in the ECP_LIST.

Your file system filter driver can follow these steps to set ECPs in an existing ECP_LIST that is associated with the create operation:

1. Call the **FltGetEcpListFromCallbackData** or **FsRtlGetEcpListFromIrp** routine to retrieve a pointer to an ECP context structure list (ECP_LIST) that is associated with the create operation.

2. Call the **FltAllocateExtraCreateParameter** or **FsRtlAllocateExtraCreateParameter** routine to allocate paged memory pool for an ECP context structure and to generate a pointer to that structure.

3. Call the **FltInsertExtraCreateParameter** or **FsRtlInsertExtraCreateParameter** routine to insert ECP context structures into the ECP_LIST structure.

Your file system filter driver can follow these steps to set ECPs in a newly created ECP_LIST that is associated with the create operation:

1. Call the **FltAllocateExtraCreateParameterList** or **FsRtlAllocateExtraCreateParameterList** routine to allocate memory for an ECP_LIST structure.

2. Call the **FltAllocateExtraCreateParameter** or **FsRtlAllocateExtraCreateParameter** routine to allocate paged memory pool for an ECP context structure and to generate a pointer to that structure.

3. Call the **FltInsertExtraCreateParameter** or **FsRtlInsertExtraCreateParameter** routine to insert ECP context structures into the ECP_LIST structure.

4. Call the **FltSetEcpListIntoCallbackData** or **FsRtlSetEcpListIntoIrp** routine to attach an ECP list to the create operation.

Removing ECPs

Your file system filter driver can follow these steps to remove ECPs for the **IRP_MJ_CREATE** operation:

1. Call the **FltRemoveExtraCreateParameter** or **FsRtlRemoveExtraCreateParameter** routine to search an ECP list for an ECP context structure. If the ECP context structure is found, the routine detaches the ECP context structure from the ECP list.

2. To free the memory for the detached ECP context structure, call the **FltFreeExtraCreateParameter** or **FsRtlFreeExtraCreateParameter** routine. You can call these routines to free memory for the ECP context structure if you have allocated the memory in one of the following ways:

   - You called the **FltAllocateExtraCreateParameter** or **FsRtlAllocateExtraCreateParameter** routine to allocate paged memory pool
   - You called the **FltAllocateExtraCreateParameterFromLookasideList** or **FsRtlAllocateExtraCreateParameterFromLookasideList** routine to allocate memory pool from a lookaside list

Marking ECPs as Acknowledged, or Determining Acknowledge Status

Your file system filter driver can call the following routines to either mark ECPs as acknowledged or determine whether the ECPs are marked as acknowledged:

- Call the **FltAcknowledgeEcp** or **FsRtlAcknowledgeEcp** routine to mark an ECP context structure as acknowledged. The ECP can be marked as looked at, used, processed, or any other condition of the ECP.

- Call the **FltIsEcpAcknowledged** or **FsRtlIsEcpAcknowledged** routine to determine whether an ECP context structure is marked as acknowledged.

Determining Origination Mode

Your file system filter driver can call the **FltIsEcpFromUserMode** or **FsRtlIsEcpFromUserMode** routine to determine whether an ECP context structure originated from user mode. A file system filter driver can refuse to accept an ECP context structure that originated from user mode.

Using Lookaside Lists to Allocate ECPs

Your file system filter driver can call the following routines to allocate ECPs from lookaside lists and to manage the lookaside lists and ECPs:

- Call the **FltInitExtraCreateParameterLookasideList** or **FsRtlInitExtraCreateParameterLookasideList** routine to initialize a paged or nonpaged pool lookaside list that is used for the allocation of one or more ECP context structures of fixed size.

- Call the **FltDeleteExtraCreateParameterLookasideList** or **FsRtlDeleteExtraCreateParameterLookasideList** routine to free the lookaside list.

- Call the **FltAllocateExtraCreateParameterFromLookasideList** or **FsRtlAllocateExtraCreateParameterFromLookasideList** routine to allocate memory pool from the lookaside list for an ECP context structure and to generate a pointer to that structure.

- Call the **FltFreeExtraCreateParameter** or **FsRtlFreeExtraCreateParameter** routine to free the memory for the ECP context structures.

User-Defined ECPs

4/26/2017 • 1 min to read • Edit Online

To define your own ECP, define a GUID that identifies your ECP and a context structure that describes your ECP. For information about how to define GUIDs and using them in driver code, see Using GUIDs in Drivers.

System-Defined ECPs

4/26/2017 • 3 min to read • Edit Online

The operating system defines the following ECPs in the *Ntifs.h* header file. These system-defined ECPs attach the specified extra information to the **IRP_MJ_CREATE** operation on a file. Elements of the file-system stack can query the ECPs for the extra information.

Typically, a filter that processes the **IRP_MJ_CREATE** operation on a file and then passes the file down to filters below it must not attach and spoof any of the following system-defined ECPs to the **IRP_MJ_CREATE** operation on the file. Similarly, a kernel-mode driver that processes and issues IRP_MJ_CREATE operations on files must not attach and spoof any of the following system-defined ECPs to the IRP_MJ_CREATE operations on the files. The following system-defined ECPs are read-only. You should use them to retrieve information only.

One exception to restricting a filter driver from attaching any of the following system-defined ECPs is when the filter driver implements a layered file system. It does this by owning file objects and by issuing its own **IRP_MJ_CREATE** operations on files below its filter, in response to the IRP_MJ_CREATE operation on a file that the filter driver services on its own file objects. Such a filter driver should propagate any ECP context structure lists (ECP_LIST) from the original IRP_MJ_CREATE operation on a file to the IRP_MJ_CREATE operations that the filter driver issues below it. By propagating these ECP lists, the filter driver ensures that any filters below the filter that issues the IRP_MJ_CREATE operations are aware of the context of the original IRP_MJ_CREATE operation.

GUID_ECP_OPLOCK_KEY
A GUID that identifies the **OPLOCK_KEY_ECP_CONTEXT** structure and is used to attach an oplock key to the open file request. The oplock key lets an application open multiple handles to the same stream without breaking the application's own oplock.

For more information about oplocks and oplock keys, see Oplock Semantics Overview.

GUID_ECP_NETWORK_OPEN_CONTEXT
A GUID that identifies the **NETWORK_OPEN_ECP_CONTEXT** structure and is used to attach extra information for network redirectors. This GUID also identifies the **NETWORK_OPEN_ECP_CONTEXT_V0** structure for drivers that run on Windows 7 and later versions of Windows and that must interpret network ECP contexts on files that reside on Windows Vista.

GUID_ECP_PREFETCH_OPEN
A GUID that identifies the **PREFETCH_OPEN_ECP_CONTEXT** structure.

The prefetcher is a component of the operating system that is tightly integrated with the cache manager and the memory manager to make disk accesses more efficient and therefore improve performance. If other components interfere with the prefetcher, system performance decreases and might deadlock. Therefore, the prefetcher attaches the PREFETCH_OPEN_ECP_CONTEXT structure to a file to communicate that the prefetcher performs an open request on the file. This open request is specified by the **Context** member of PREFETCH_OPEN_ECP_CONTEXT. Other components, such as, file system filter drivers, can determine whether PREFETCH_OPEN_ECP_CONTEXT is attached to the file and then take appropriate action.

GUID_ECP_NFS_OPEN
A GUID that identifies the **NFS_OPEN_ECP_CONTEXT** structure. The Network File System (NFS) server attaches the NFS_OPEN_ECP_CONTEXT structure to an open file request. The NFS server uses this GUID on any open file request that the NFS server makes to satisfy a client request. The file-system stack can then determine whether NFS_OPEN_ECP_CONTEXT is attached to the open file request. Based on the information in NFS_OPEN_ECP_CONTEXT the file-system stack can determine the client that requested that the file be opened and why.

GUID_ECP_SRV_OPEN
A GUID that identifies the **SRV_OPEN_ECP_CONTEXT** structure. A server attaches the SRV_OPEN_ECP_CONTEXT structure to an open file request. The server uses this GUID on any open file request that the server makes to satisfy a conditional client request. The file-system stack can then determine whether SRV_OPEN_ECP_CONTEXT is attached to the open file request. Based on the information in SRV_OPEN_ECP_CONTEXT the file-system stack can determine the client that requested that the file be opened and why.

GUID_ECP_DUAL_OPLOCK_KEY
A GUID that identifies the **DUAL OPLOCK_KEY_ECP_CONTEXT** structure. Like the **OPLOCK_KEY_ECP_CONTEXT** structure, **DUAL OPLOCK_KEY_ECP_CONTEXT** is used to attach an oplock key to the open file request. With **DUAL OPLOCK_KEY_ECP_CONTEXT**, however, a parent key can also be set to provide oplock for a target file's directory.

GUID_ECP_IO_DEVICE_HINT
A GUID that identifies the **IO_DEVICE_HINT_ECP_CONTEXT** structure. Device hints are used to assist name provider minifilter drivers in tracking a reparse target to new device.

GUID_ECP_NETWORK_APP_INSTANCE
A GUID that identifies the **NETWORK_APP_INSTANCE_ECP_CONTEXT** structure. A client application in a failover cluster may have a set of files opened on a node in the cluster. The file objects are tagged to an application by an instance identifier in the **NETWORK_APP_INSTANCE_ECP_CONTEXT** structure. On failover, a secondary node can validate a client application's access to the opened files with the previously cached application instance identifier.

File System Filter Drivers

4/26/2017 • 1 min to read • Edit Online

Note For optimal reliability and performance, we recommend using file system minifilter drivers instead of legacy file system filter drivers. Also, legacy file system filter drivers can't attach to direct access (DAX) volumes. For more about file system minifilter drivers, see Advantages of the Filter Manager Model. To port your legacy driver to a minifilter driver, see Guidelines for Porting Legacy Filter Drivers.

This section includes the following topics, which describe file system filter drivers:

- File System Fundamentals
- Introduction to File System Filter Drivers
- Filtering IRPs and Fast I/O
- Writing IRP Dispatch Routines
- Using IRP Completion Routines
- Tracking Per-Stream Context in a Legacy File System Filter Driver
- Tracking Per-File Context in a Legacy File System Filter Driver
- Blocking legacy file system filter drivers

File System Fundamentals

4/26/2017 • 1 min to read • Edit Online

Note For optimal reliability and performance, we recommend using file system minifilter drivers instead of legacy file system filter drivers. Also, legacy file system filter drivers can't attach to direct access (DAX) volumes. For more about file system minifilter drivers, see Advantages of the Filter Manager Model. To port your legacy driver to a minifilter driver, see Guidelines for Porting Legacy Filter Drivers.

This section introduces fundamental file system driver concepts that are important to developers of file system filter drivers. The following topics are discussed:

What Determines When a Driver Is Loaded

What Happens to File Systems During System Boot

Storage Volumes, Storage Device Stacks, and File System Stacks

Mounting a Volume

What Determines When a Driver Is Loaded

4/26/2017 • 3 min to read • Edit Online

Before exploring when and how file system drivers are loaded during the system boot sequence, it is necessary to understand driver start types and load order groups.

Driver Start Types

A kernel-mode driver's *start type* specifies whether the driver is to be loaded during or after system startup. There are five possible start types:

SERVICE_BOOT_START (0x00000000)
Indicates a driver started by the operating system (OS) loader. File system filter drivers commonly use this start type or SERVICE_DEMAND_START. On Microsoft Windows XP and later systems, filters must use this start type in order to take advantage of the new file system filter load order groups.

SERVICE_SYSTEM_START (0x00000001)
Indicates a driver started during OS initialization. This start type is used by the file system recognizer. Except for the file systems listed below under "SERVICE_DISABLED," file systems (including network file system components) commonly use this start type or SERVICE_DEMAND_START. This start type is also used by device drivers for PnP devices that are enumerated during system initialization but not required to load the system.

SERVICE_AUTO_START (0x00000002)
Indicates a driver started by the Service Control Manager during system startup. Rarely used.

SERVICE_DEMAND_START (0x00000003)
Indicates a driver started on demand, either by the PnP Manager (for device drivers) or by the Service Control Manager (for file systems and file system filter drivers).

SERVICE_DISABLED (0x00000004)
Indicates a driver that is not started by the OS loader, Service Control Manager, or PnP Manager. Used by file systems that are loaded by a file system recognizer (except when they are the boot file system) or (in the case of EFS) by another file system. Such file systems include CDFS, EFS, FastFat, NTFS, and UDFS. Also used to temporarily disable a driver during debugging.

Specifying Start Type

A driver writer can specify the start type for a driver at installation time in either of the following ways:

- By specifying the desired start type for the **StartType** entry in the *service-install-section* referred to by an **AddService** directive in the driver's INF file. This method is described in ServiceInstall Section.
- By passing the desired start type for the *dwStartType* parameter when calling **CreateService** or **ChangeServiceConfig** from a user-mode installation program. This method is described in the reference entries for **CreateService** and **ChangeServiceConfig** in the Microsoft Windows SDK documentation.

Driver Load Order Groups

Within the SERVICE_BOOT_START and SERVICE_SYSTEM_START start types, the relative order in which drivers are loaded is specified by each driver's *load order group*.

Drivers whose start type is SERVICE_BOOT_START are called *boot (or boot-start) drivers*. On Microsoft Windows 2000 and earlier systems, most filters that are boot drivers belong to the "filter" group. On Microsoft Windows XP and later systems, filters that are boot drivers generally belong to one of the new FSFilter load order groups. These load order groups are described in detail in Load Order Groups for File System Filter Drivers.

Driver whose start type is SERVICE_SYSTEM_START are also loaded in the order of the load order groups to which they belong. However, no system-start driver is loaded until after all boot drivers have been loaded.

Note Load order groups are ignored for drivers whose start type is SERVICE_AUTO_START, SERVICE_DEMAND_START, or SERVICE_DISABLED.

A complete, ordered list of load order groups can be found under the **ServiceGroupOrder** subkey of the following registry key:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control

The same load group ordering is used for SERVICE_BOOT_START and SERVICE_SYSTEM_START drivers. However, all SERVICE_BOOT_START drivers are loaded and started before any SERVICE_SYSTEM_START drivers are loaded.

Specifying Load Order Group

A driver writer can specify the load order group for a driver at installation time in either of the following ways:

- By specifying the desired load order group for the **LoadOrderGroup** entry in the *service-install-section* referred to by an **AddService** directive in the driver's INF file. This method is described in ServiceInstall Section.

- By passing the desired start type for the *lpLoadOrderGroup* parameter when calling **CreateService** or **ChangeServiceConfig** from a user-mode installation program. This method is described in the reference entries for **CreateService** and **ChangeServiceConfig** in the Microsoft Windows SDK documentation.

For more general information about driver load order and load order groups, see Specifying Driver Load Order.

What Happens to File Systems During System Boot

4/26/2017 • 4 min to read • Edit Online

File systems are initialized during the system boot process; specifically, during I/O system initialization. The I/O Manager creates the global file system queue and initializes the file system and filter drivers that were loaded by the operating system (OS) loader and the PnP Manager.

Following is a summary of selected portions of the system boot process that are of interest to file system and filter driver developers.

1. During system boot, the OS loader loads the boot file system, the RAW file system, and all drivers of type SERVICE_BOOT_START before the loader transfers control to the kernel. These drivers are in memory when the kernel gets control.

   Drivers are loaded in order of the load order groups to which they are assigned. Among file system filters, those that are assigned to one of the new file system filter driver load order groups are loaded before all other filter drivers. These load order groups are described in detail in Load Order Groups for File System Filter Drivers.

   Then all drivers in the "filter" load order group are loaded. Note that the "filter" group includes storage filter drivers as well as file system filter drivers, and it includes third-party as well as built-in filter drivers.

2. The I/O Manager creates a global file system queue with four segments: one each for CD-ROM, disk, tape devices, and network file systems. Later, when each file system is registered, its control device objects are added to the appropriate segments of this queue. At this point, however, no file systems have yet been registered, so the queue is empty.

3. The PnP Manager calls the **DriverEntry** routines of the RAW file system and all SERVICE_BOOT_START drivers.

   If a SERVICE_BOOT_START driver is dependent on other drivers, those drivers are loaded and started as well.

   The PnP Manager starts the boot devices by calling the **AddDevice** routines of the boot device drivers. If a boot device has child devices, those devices are enumerated. The child devices are also configured and started if their drivers are boot-start drivers. If a device's drivers are not all boot-start drivers, the PnP Manager creates a devnode for the device but does not start the device.

   At this point, all boot drivers are loaded and the boot devices are started.

4. The PnP Manager traverses the PnP device tree, locating and loading the drivers that are associated with each devnode but not already running.

   (For more information about the PnP device tree, see Device Tree.) When each PnP device starts, the PnP Manager enumerates the children of the device, if any. The PnP Manager configures the child devices, loads their device drivers, and starts the devices.

   The PnP Manager loads each device's drivers *regardless* of the drivers' **StartType**, **LoadOrderGroup**, or **Dependencies** values.

   In this step, the PnP Manager only configures and starts devices that are PnP-enumerable. If a device is not PnP-enumerable, the PnP Manager ignores the device and does not enumerate its children, even if the child devices are PnP-enumerable.

5. The PnP Manager loads and initializes drivers of type SERVICE_SYSTEM_START that are not yet loaded.

The file system recognizer (FsRec) is loaded at this time. Note that, although it is in the "Boot File System" load order group, FsRec is not the boot file system. The actual boot file system – that is, the file system that mounted the boot volume – is loaded at the start of the boot process.

Later in the SERVICE_SYSTEM_START phase, file systems in the "File System" load order group are loaded. This includes the Named Pipe File System (NPFS) and Mailslot File System (MSFS). It does not include media-based file systems, such as NTFS, FAT, CDFS, or UDFS.

Network file systems, which are in the "Network" load order group, are also loaded during this phase.

6. After all drivers that load at boot time have been initialized, the I/O Manager calls the reinitialization routines of any drivers that have them. A *reinitialization routine* is a callback routine that is registered by a boot driver that needs to be given additional processing time at this point in the boot process. Reinitialization routines are registered by calling **IoRegisterBootDriverReinitialization** or **IoRegisterDriverReinitialization**.

7. The Service Control Manager loads drivers of type SERVICE_AUTO_START that are not already loaded.

File System Recognizer

After system boot, the storage device drivers for all volumes attached to the system are loaded and started. However, not all built-in file systems are loaded, and not all file system volumes are mounted. The File System Recognizer (FsRec) performs these tasks as needed to process **IRP_MJ_CREATE** requests.

FsRec is loaded in the SERVICE_SYSTEM_START phase of system startup. Note that, although it is in the "Boot File System" load order group, FsRec is not the boot file system. The actual boot file system – that is, the file system that mounted the boot volume – is loaded at the start of the boot process.

Storage Device Stacks, Storage Volumes, and File System Stacks

4/26/2017 • 1 min to read • Edit Online

Before exploring how file system filter drivers attach to file systems and volumes, it is necessary to understand the relationship between storage device stacks, storage volumes, and file system stacks. This relationship is discussed in the following sections:

Storage Device Stacks

Storage Volumes

File System Stacks

Storage Device Stacks

4/26/2017 • 1 min to read • Edit Online

Most storage drivers are PnP device drivers, which are loaded and managed by the PnP Manager. Storage devices are represented in the PnP *device tree*, which contains a device node, or *devnode*, for every physical or logical device on the machine. It is important to note that file systems and file system filter drivers are not PnP device drivers; thus the PnP device tree contains no devnodes for them. For more information about the PnP device tree, see Device Tree.

The devnode for a particular storage device contains the *storage device stack* for the device; this is the chain of attached device objects that represent the device's storage device drivers. Because a storage device, such as a disk, might contain one or more logical volumes (partitions or dynamic volumes), the storage device stack itself often looks more like a tree than a stack. The root of this tree is a functional device object (FDO) for a storage adapter or for another device stack that is integrated with the storage stack. The leaves of this tree are the physical device objects (PDO) for the logical volumes, also called storage volumes, on which file system volumes can be mounted.

For diagrams and descriptions of some typical storage device stacks, see the following sections of the Storage Devices Design Guide:

Device Object Example for a SCSI HBA

Device Object Example for an IEEE 1394 Controller

Storage Volumes

4/26/2017 • 1 min to read • Edit Online

A *volume* is a storage device, such as a fixed disk, floppy disk, or CD-ROM, that is formatted to store directories and files. A large volume can be divided into more than one *logical volume*, also called a *partition*. Each logical volume is formatted for use by a particular media-based file system, such as NTFS, FAT, or CDFS.

A *storage volume*, or *storage device object*, is a device object – usually a physical device object (PDO) – that represents a logical volume to the system. The storage device object resides in the storage device stack, but it is not necessarily the topmost device object in the stack.

When a file system is mounted on a storage volume, it creates a file system volume device object (VDO) to represent the volume to the file system. The file system VDO is mounted on the storage device object by means of a shared object called a *volume parameter block* (VPB).

Mount Manager

The *Mount Manager* is the part of the I/O system that is responsible for managing storage volume information such as volume names, drive letters, and volume mount points. When a new storage volume is added to the system, the Mount Manager is notified of its arrival in either of the following ways:

- The class driver that created the storage volume calls **IoRegisterDeviceInterface** to register a new interface in the MOUNTDEV_MOUNTED_DEVICE_GUID interface class. When this happens, the Plug and Play device interface notification mechanism alerts the Mount Manager of the volume's arrival in the system.
- The driver for the storage volume sends the Mount Manager an IRP_MJ_DEVICE_CONTROL request, specifying **IOCTL_MOUNTMGR_VOLUME_ARRIVAL_NOTIFICATION** for the I/O control code. This request can be created by calling **IoBuildDeviceIoControlRequest**.

Unique Volume Name

The Mount Manager responds to the arrival of a new storage volume by querying the volume driver for the following information:

- The volume's nonpersistent device object name (or target name), located in the **Device** directory of the system object tree (for example: "\Device\HarddiskVolume1")
- The volume's globally unique identifier (GUID), also called the *unique volume name*
- A suggested persistent symbolic link name for the volume, such as a drive letter (for example, "\DosDevices\D:")

For more information about the interaction between storage drivers and the Mount Manager, see Supporting Mount Manager Requests in a Storage Class Driver.

File System Stacks

4/26/2017 • 1 min to read • Edit Online

File system drivers create two different types of device objects: control device objects (CDO) and volume device objects (VDO). A *file system stack* consists of one of these device object, together with any filter device objects for file system filter drivers that are attached to it. The file system's device object always forms the bottom of the stack.

File System Control Device Objects

File system control device objects represent entire file systems, rather than individual volumes, and are stored in the global file system queue. A file system creates one or more named control device objects in its **DriverEntry** routine. For example, FastFat creates two CDOs: one for fixed media and one for removable media. CDFS creates only one CDO, because it has only removable media.

File system control device objects are required to be named. This is because file system filter drivers, as well as many kernel-mode support routines, rely on this difference between volume device objects and control device objects as a way of telling them apart.

File System Volume Device Objects

File system volume device objects represent volumes mounted by file systems. A file system creates a volume device object when it mounts a volume, usually in response to a volume mount request. Unlike a control device object, a volume device object is always associated with a specific logical or physical storage device.

Note Unlike control device objects, volume device objects must never be named, because naming a volume device object would create a security hole.

Mounting a Volume

4/26/2017 • 1 min to read • Edit Online

The volume mount process is typically triggered by a request to open a file on a logical volume (that is, a partition or dynamic volume) as follows:

1. A user application calls **CreateFile** to open a file. Or a kernel-mode driver calls **ZwCreateFile** or **IoCreateFileSpecifyDeviceObjectHint**.
2. The I/O Manager determines which logical volume is the target of the request and checks its device object to see whether it is mounted. If the VPB_MOUNTED flag is set, the volume has been mounted by a file system.
3. If the volume has not been mounted by a file system since system boot (that is, the VPB_MOUNTED flag is not set), the I/O Manager sends a volume mount (**IRP_MJ_FILE_SYSTEM_CONTROL**, IRP_MN_MOUNT_VOLUME) request to each file system that might claim the volume.

   Not all built-in file systems are necessarily loaded – even well after system boot. (See What Happens to File Systems During System Boot.) For built-in file systems that are not yet loaded, the I/O Manager sends the volume mount request to the file system recognizer (FsRec), which checks the volume boot sector on behalf of these file systems.

   If FsRec determines that the volume was formatted by a not-yet-loaded file system, the I/O Manager responds by sending a load file system (**IRP_MJ_FILE_SYSTEM_CONTROL**, IRP_MN_LOAD_FILE_SYSTEM) request to FsRec, which loads the file system. The I/O Manager then sends the original volume mount request to the file system.
4. Each file system that receives the mount volume request examines the volume's boot sector to determine whether the volume's format and other information indicate that the volume was formatted by that particular file system. If the format matches, the file system mounts the volume.

The following sections discuss how the file system mounts the volume after recognizing it:

How the Volume Is Mounted

Volume Mount Example

How the Volume Is Mounted

4/26/2017 • 1 min to read • Edit Online

How the volume is mounted depends on the file system and whether it has previously mounted the volume.

When a file system receives the volume mount request for a new volume, it creates a volume device object (VDO) for the volume. The VDO consists of a DEVICE_OBJECT plus an optional file-system-defined device extension. The newly created VDO forms the base of the file system volume stack for the new (or remounted) volume.

The file system mounts the volume by associating the VDO with the volume parameter block (VPB) for the corresponding storage device object and sets the VPB_MOUNTED flag on the VPB.

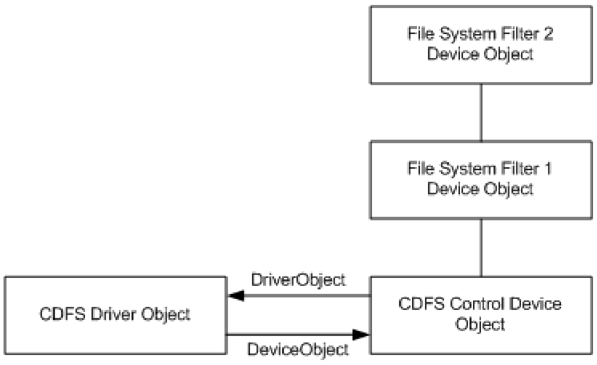
After the volume is mounted by the file system, file system filter drivers can attach to the top of the new file system volume stack. Any I/O requests sent to the file system are automatically sent first to the file system filter device object at the top of the volume stack. However, file system filters should only detach from the volume stack when the I/O Manager sends a fast I/O detach request to notify drivers on the volume stack that the volume is about to be removed.

Note The storage device object for the volume resides in the storage device stack, but it is not necessarily the topmost device object in the stack. Moreover, even after the volume is mounted, storage filter drivers can still attach to the top of the storage stack. It is important for driver writers to keep in mind that, when the file system sends an IRP from the VDO to the storage device stack, it sends it to the storage device object for the volume, not the topmost device object in the stack. (However, when the I/O Manager sends an IRP directly to the storage stack, bypassing the file system, that IRP is sent to the topmost device object in the stack.)

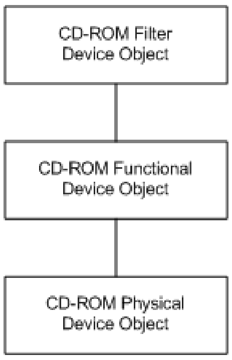
Volume Mount Example

4/26/2017 • 1 min to read • Edit Online

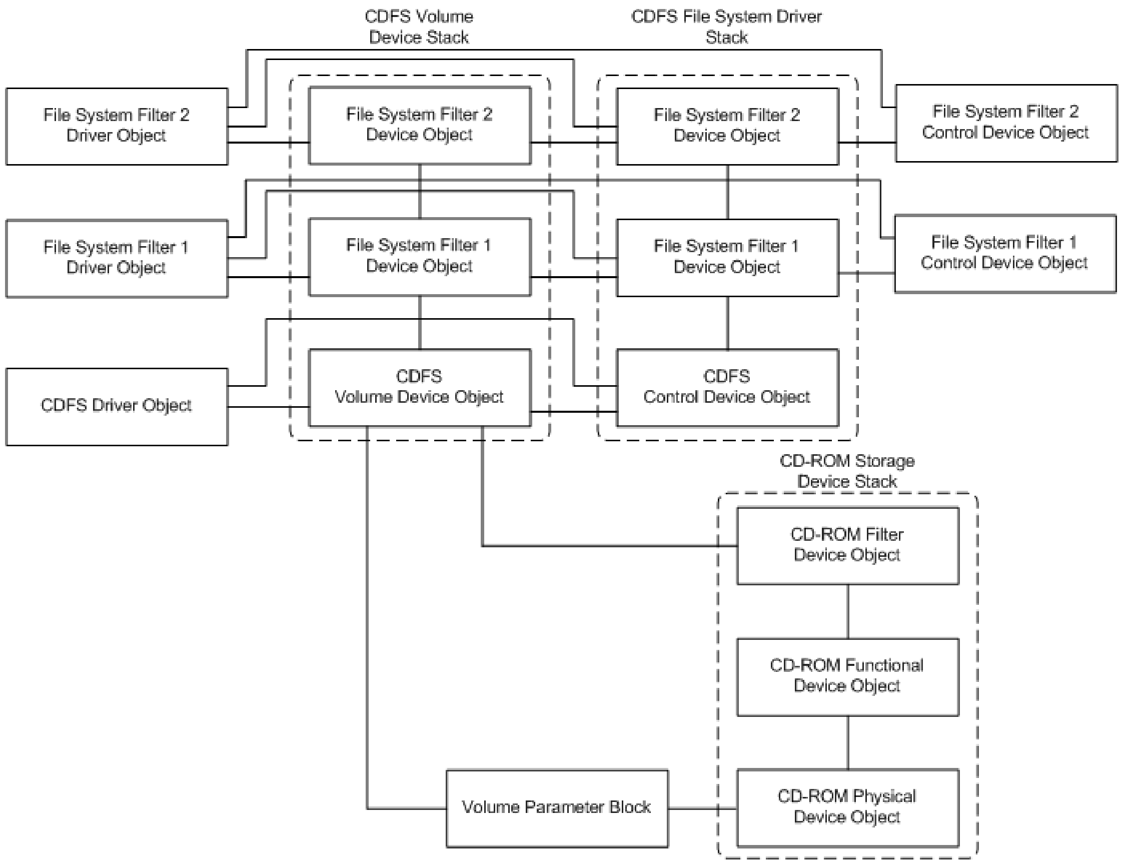
The following figure shows what CDFS might look like before it has mounted any volumes. In this example, two filters have attached themselves to the CDFS control device object. (Note: The global file system queue that contains the CDFS control device object is not shown.)


File System Filter 2 Device Object
File System Filter 1 Device Object
CDFS Control Device Object
CDFS Driver Object
DriverObject
DeviceObject

The following figure shows a typical driver stack for a CD-ROM storage device that has not yet been mounted as a CDFS volume.


CD-ROM Filter Device Object
CD-ROM Functional Device Object
CD-ROM Physical Device Object

The following figure shows what the file system driver stack, volume stack, and CD-ROM storage device stack look like after the CDFS file system has mounted a volume on a CD-ROM device.



CDFS Volume Device Stack
CDFS File System Driver Stack
File System Filter 2 Driver Object
File System Filter 2 Device Object
File System Filter 2 Device Object
File System Filter 2 Control Device Object
File System Filter 1 Driver Object
File System Filter 1 Device Object
File System Filter 1 Device Object
File System Filter 1 Control Device Object
CDFS Driver Object
CDFS Volume Device Object
CDFS Control Device Object
CD-ROM Storage Device Stack
CD-ROM Filter Device Object
CD-ROM Functional Device Object
Volume Parameter Block
CD-ROM Physical Device Object

Some notes about the preceding figure:

- The CDFS control device object forms the base of a file system driver stack. This stack, which is not mounted on a storage device, can receive IRPs directly, and can also contain file system filter device objects. Filters attach to file system control device objects to watch for volume mount (**IRP_MJ_FILE_SYSTEM_CONTROL**, IRP_MN_MOUNT_VOLUME) requests. File system control device objects are required to be named. This distinguishes them from file system volume device objects, which are never named.

- As the diagram shows, although it would be possible to attach a second storage filter to the top of the CD-ROM storage device stack after the CDFS volume has been mounted, this filter would not receive any IRPs that are passed down from the file system stack to the storage device stack. However, it would receive any IRPs that are sent directly to the storage device stack.

- It is important to note that, after the file system has mounted the volume, the storage device stack can still receive IRPs directly. Specifically, power IRPs (IRP_MJ_POWER) are always sent directly to the storage device stack, never to the file system stack. (Thus, for example, file system filter drivers should never register a dispatch routine for IRP_MJ_POWER in their **DriverEntry** routines.)

  However, PnP IRPs (**IRP_MJ_PNP**) can be sent to either stack. Filter drivers chained above a file system volume should always pass these IRPs down to the next lower driver by default so that the file system's volume device can pass the IRPs down to the storage device stack.

Introduction to File System Filter Drivers

4/26/2017 • 1 min to read • Edit Online

Note For optimal reliability and performance, we recommend using file system minifilter drivers instead of legacy file system filter drivers. Also, legacy file system filter drivers can't attach to direct access (DAX) volumes. For more about file system minifilter drivers, see Advantages of the Filter Manager Model. To port your legacy driver to a minifilter driver, see Guidelines for Porting Legacy Filter Drivers.

This section introduces file system filter drivers. It includes the following topics:

What Is a File System Filter Driver?

File System Filter Drivers Are Not Device Drivers

Installing a File System Filter Driver

Initializing a File System Filter Driver

Attaching a Filter to a File System or Volume

What Is a File System Filter Driver?

4/26/2017 • 1 min to read • Edit Online

A *file system filter driver* is an optional driver that adds value to or modifies the behavior of a file system. A file system filter driver is a kernel-mode component that runs as part of the Windows executive.

A file system filter driver can filter I/O operations for one or more file systems or file system volumes. Depending on the nature of the driver, *filter* can mean *log*, *observe*, *modify*, or even *prevent*. Typical applications for file system filter drivers include antivirus utilities, encryption programs, and hierarchical storage management systems.

File System Filter Drivers Are Not Device Drivers

4/26/2017 • 1 min to read • Edit Online

A *device driver* is a software component that controls a particular hardware I/O device. For example, a DVD storage driver controls a DVD drive.

In contrast, a file system filter driver works in conjunction with one or more file systems to manage file I/O operations. These operations include creating, opening, closing, and enumerating files and directories; getting and setting file, directory, and volume information; and reading and writing file data. In addition, file system filter drivers must support file system-specific features such as caching, locking, sparse files, disk quotas, compression, security, recoverability, reparse points, and volume mount points.

For more details on the similarities and differences between file system filter drivers and device drivers, see the following:

How File System Filter Drivers Are Similar to Device Drivers

How File System Filter Drivers Are Different from Device Drivers

How File System Filter Drivers Are Similar to Device Drivers

4/26/2017 • 1 min to read • Edit Online

The following subsections describe some of the similarities between file system filter drivers and device drivers in the Microsoft Windows operating system.

Similar Structure

Like device drivers, file system filter drivers have **DriverEntry**, dispatch, and I/O completion routines. They call many of the same kernel-mode routines that device drivers call, and they filter I/O requests for devices (that is, file system volumes) with which they are associated.

Similar Functionality

Because file system filter drivers and device drivers are part of the I/O system, they both receive I/O request packets (IRPs) and act on them.

Like device drivers, file system filter drivers can also create their own IRPs and send them to lower-level drivers.

Both kinds of drivers can register for notification (by using callback functions) of various system events.

Other Similarities

Like device drivers, file system filter drivers can receive Introduction to I/O Control Codes (IOCTLs). However, file system filter drivers can also receive--and define--file system control codes (FSCTLs).

Like device drivers, file system filter drivers can be configured to be loaded at system startup time or to be loaded later, after the system startup process is complete.

How File System Filter Drivers Are Different from Device Drivers

4/26/2017 • 1 min to read • Edit Online

The following subsections describe some of the differences between file system filter drivers and device drivers.

No Power Management

Because file system filter drivers are not device drivers and thus do not control hardware devices directly, they do not receive **IRP_MJ_POWER** requests. Instead, power IRPs are sent directly to the storage device stack. In rare circumstances, however, file system filter drivers might interfere with power management. For this reason, file system filter drivers should not register dispatch routines for IRP_MJ_POWER in the **DriverEntry** routine, and they should not call PoXxx routines.

No WDM

File system filter drivers cannot be Windows Driver Model (WDM) drivers. The Microsoft Windows Driver Model is only for device drivers. For more information about file system driver development in Windows Me, Windows 98, and Windows 95, see the Windows Me Driver Development Kit (DDK).

No AddDevice or StartIo

Because file system filter drivers are not device drivers and thus do not control hardware devices directly, they should not have **AddDevice** or **StartIo** routines.

Different Device Objects Created

Although file system filter drivers and device drivers both create device objects, they differ in the number and kinds of device objects that they create.

Device drivers create physical and functional device objects to represent devices. The Plug and Play (PnP) Manager builds and maintains a global device tree that contains all device objects that are created by device drivers. The device objects that file system filter drivers create are not contained in this device tree.

File system filter drivers do not create physical or functional device objects. Instead, they create control device objects and filter device objects. The *control device object* represents the filter driver to the system and to user-mode applications. The *filter device object* performs the actual work of filtering a specific file system or volume. A file system filter driver normally creates one control device object and one or more filter device objects.

Other Differences

Because file system filter drivers are not device drivers, they do not perform direct memory access (DMA).

Unlike device filter drivers, which can attach above or below a target device's function driver, file system filter drivers can attach only above a target file system driver. Thus, in device-driver terms, a file system filter driver can be only an upper filter, never a lower filter.

Installing a File System Filter Driver

4/26/2017 • 1 min to read • Edit Online

For Microsoft Windows XP and later operating systems, you should install your file system filter drivers by using an INF file and an installation application. (On Windows 2000 and earlier operating systems, filter drivers were commonly installed by the Service Control Manager.)

In the future, INF-based installation is expected to meet Windows Hardware Certification Kit requirements for file system filter drivers. Note that "INF-based installation" means only that you will need to use an INF file to copy files and to store information in the registry. You will not be required to install your entire product by using only an INF file, and you will not be required to provide a "right-click install" option for your driver.

This section includes:

Creating an INF File for a File System Filter Driver

4/26/2017 • 9 min to read • Edit Online

The Windows Setup and Device Installer Services, known collectively as SetupAPI, provide the functions that control Windows setup and driver installation. The installation process is controlled by INF files.

A file system filter driver's INF file provides instructions that SetupAPI uses to install the driver. The INF file is a text file that specifies the files that must be present for your driver to run and the source and destination directories for the driver files. An INF file also contains driver configuration information that SetupAPI stores in the registry, such as the driver's start type and load order group.

For more information about INF files and how they are created, see Creating an INF File and INF File Sections and Directives. For general information about signing drivers, see Driver Signing.

You can create a single INF file to install your driver on multiple versions of the Windows operating system. For more information about creating such an INF file, see Creating INF Files for Multiple Platforms and Operating Systems and Creating International INF Files.

Starting with 64-bit versions of Windows Vista, all kernel-mode components, including non-PnP (Plug and Play) drivers such as file system drivers (file system, legacy filter, and minifilter drivers), must be signed in order to load and execute. For these versions of the Windows operating system, the following list contains information that is relevant to file system filter drivers.

- INF files for non-PnP drivers, including file system drivers, are not required to contain [Manufacturer] or [Models] sections.
- The **SignTool** command-line tool, located in the \bin\SelfSign directory of the WDK installation directory, can be used to directly "embed sign" a driver SYS executable file. For performance reasons, boot-start drivers must contain an embedded signature.
- Given an INF file, the **Inf2Cat** command-line tool can be used to create a catalog (.cat) file for a driver package. Only catalog files can receive WHQL logo signatures.
- With Administrator privileges, an unsigned driver can still be installed on x64-based systems starting with Windows Vista. However, the driver will fail to load (and thus execute) because it is unsigned.
- For detailed information about the driving signing process, including the driving signing process for 64-bit versions of Windows Vista, see Kernel-Mode Code Signing Walkthrough.
- All kernel-mode components, including custom kernel-mode development tools, must be signed. For more information, see Signing Drivers during Development and Test (Windows Vista and Later).

INF files cannot be used to read information from the registry or to launch a user-mode application.

After creating an INF file, you will typically write the source code for your setup application. The setup application calls user-mode setup functions to access the information in the INF file and perform installation operations.

To construct your own filter driver INF file, use the INF files for the sample file system filter drivers as a template. You can use the ChkINF tool to check the syntax of your INF file.

An INF file for a file system filter driver generally contains the following sections.

- Version (required)
- DestinationDirs (optional but recommended)

- SourceDisksNames (required)
- SourceDisksFiles (required)
- DefaultInstall (required)
- DefaultInstall.Services (required)
- ServiceInstall (required)
- DefaultUninstall (optional)
- DefaultUninstall.Services (optional)
- Strings (required)

Version Section (required)

The **Version** section specifies a class and GUID that are determined by the type of filter, as shown in the following code example.

```
[Version]
Signature   = "$WINDOWS NT$"
Class       = "ActivityMonitor"
ClassGuid   = {b86dff51-a31e-4bac-b3cf-e8cfe75c9fc2}
Provider    = %Msft%
DriverVer   = 08/28/2000,1.0.0.1
CatalogFile =
```

The following table shows the values that file system filter drivers should specify in the **Version** section.

ENTRY	VALUE
Signature	"$WINDOWS NT$"
Class	See File System Filter Driver Classes and Class GUIDs.
ClassGuid	See File System Filter Driver Classes and Class GUIDs.
Provider	In your own INF file, you should specify a provider other than Microsoft.
DriverVer	See **INF DriverVer directive**.
CatalogFile	Leave this entry blank. In the future, it will contain the name of a WHQL-supplied catalog file for signed drivers.

DestinationDirs Section (optional but recommended)

The **DestinationDirs** section specifies the directories where filter driver and application files will be copied.

In this section and in the **ServiceInstall** section, you can specify well-known system directories by using system-defined numeric values. For a list of these values, see **INF DestinationDirs Section**. In the following code example, the value "12" refers to the Drivers directory (%windir%\system32\drivers), and the value "10" refers to the Windows directory (%windir%).

```
[DestinationDirs]
DefaultDestDir             = 12
MyLegacyFilter.DriverFiles = 12
MyLegacyFilter.UserFiles   = 10,MyLegacyFilter
```

SourceDisksNames Section (required)

The **SourceDisksNames** section specifies the distribution media to be used.

In the following code example, the **SourceDisksNames** section lists a single distribution media. The unique identifier for the media is 1. The name of the media is specified by the %Disk1% token, which is defined in the **Strings** section of the INF file.

```
[SourceDisksNames]
1 = %Disk1%
```

SourceDisksFiles Section (required)

The **SourceDisksFiles** section specifies the location and names of the files to be copied.

In the following code example, the **SourceDisksFiles** section lists the files to be copied for the driver and specifies that the files can be found on the media whose unique identifier is 1 (This identifier is defined in the **SourceDisksNames** section of the INF file.)

```
[SourceDisksFiles]
myLegacyFilter.exe = 1
myLegacyFilter.sys = 1
```

DefaultInstall Section (required)

In the **DefaultInstall** section, a **CopyFiles** directive copies the file system filter driver's driver files and user-application files to the destinations that are specified in the **DestinationDirs** section.

Note The **CopyFiles** directive should not refer to the catalog file or the INF file itself; SetupAPI copies these files automatically.

You can create a single INF file to install your driver on multiple versions of the Windows operating system. This type of INF file is created by creating additional **DefaultInstall**, **DefaultInstall.Services**, **DefaultUninstall**, and **DefaultUninstall.Services** sections for each operating system version. Each section is labeled with a *decoration* (for example, .ntx86, .ntia64, or .nt) that specifies the operating system version to which it applies. For more information about creating this type of INF file, see Creating INF Files for Multiple Platforms and Operating Systems.

In the following code example, the **CopyFiles** directive copies the files that are listed in the MyLegacyFilter.DriverFiles and MyLegacyFilter.UserFiles sections of the INF file.

```
[DefaultInstall]
OptionDesc = %MyLegacyFilterServiceDesc%
CopyFiles = MyLegacyFilter.DriverFiles, MyLegacyFilter.UserFiles
```

DefaultInstall.Services Section (required)

The **DefaultInstall.Services** section contains an **AddService** directive that controls how and when the services of a particular driver are loaded.

In the following code example, the **AddService** directive adds the MyLegacyFilter service to the operating system. The %MyLegacyFilterServiceName% token contains the service name string, which is defined in the **Strings** section of the INF file. MyLegacyFilter.Service is the name of the example driver's **ServiceInstall** section.

```
[DefaultInstall.Services]
AddService = %MyLegacyFilterServiceName%,,MyLegacyFilter.Service
```

ServiceInstall Section (required)

The **ServiceInstall** section adds subkeys or value names to the registry and sets values. The name of the **ServiceInstall** section must appear in an **AddService** directive in the **DefaultInstall.Services** section.

The following code example shows the **ServiceInstall** section for the MyLegacyFilter example driver.

```
[MyLegacyFilter.Service]
DisplayName     = %MyLegacyFilterServiceName%
Description     = %MyLegacyFilterServiceDesc%
ServiceBinary   = %12%\myLegacyFilter.sys
ServiceType     = 2 ;    SERVICE_FILE_SYSTEM_DRIVER
StartType       = 3 ;    SERVICE_DEMAND_START
ErrorControl    = 1 ;    SERVICE_ERROR_NORMAL
LoadOrderGroup = "FSFilter Activity Monitor"
AddReg          = MyLegacyFilter.AddRegistry
```

The **DisplayName** entry specifies the name for the service. In the preceding example, the service name string is specified by the %MyLegacyFilterServiceName% token, which is defined in the **Strings** section of the INF file.

The **Description** entry specifies a string that describes the service. In the preceding example, this string is specified by the %MyLegacyFilterServiceDesc% token, which is defined in the **Strings** section of the INF file.

The **ServiceBinary** entry specifies the path to the executable file for the service. In the preceding example, the value 12 refers to the Drivers directory (%windir%\system32\drivers).

The **ServiceType** entry specifies the type of service. The following table lists the possible values for **ServiceType** and their corresponding service types.

VALUE	DESCRIPTION
0x00000001	SERVICE_KERNEL_DRIVER (Device driver service)
0x00000002	SERVICE_FILE_SYSTEM_DRIVER (File system or file system filter driver service)
0x00000010	SERVICE_WIN32_OWN_PROCESS (Microsoft Win32 service that runs in its own process)
0x00000020	SERVICE_WIN32_SHARE_PROCESS (Win32 service that shares a process)

The **ServiceType** entry should always be set to SERVICE_FILE_SYSTEM_DRIVER for a file system filter driver.

The **StartType** entry specifies when to start the service. The following table lists the possible values for **StartType** and their corresponding start types.

VALUE	DESCRIPTION
0x00000000	SERVICE_BOOT_START
0x00000001	SERVICE_SYSTEM_START
0x00000002	SERVICE_AUTO_START
0x00000003	SERVICE_DEMAND_START
0x00000004	SERVICE_DISABLED

For detailed descriptions of these start types, see What Determines When a Driver Is Loaded.

If your driver's start type is SERVICE_BOOT_START (that is, the driver is a boot-start driver), you should also ensure that the **LoadOrderGroup** entry is appropriate for the type of filter you are developing. To choose a load order group, see Load Order Groups for File System Filter Drivers. Additionally, starting with x64-based Windows Vista systems, the binary image file of a boot-start driver must contain an embedded signature. This requirement ensures optimal system boot performance. For more information, see Kernel-Mode Code Signing Walkthrough.

For information about how the **StartType** and **LoadOrderGroup** entries determine when the driver is loaded, see What Determines When a Driver Is Loaded.

The **ErrorControl** entry specifies the action to be taken if the service fails to start during system startup. The following table lists the possible values for **ErrorControl** and their corresponding error control values.

VALUE	ACTION
0x00000000	SERVICE_ERROR_IGNORE (Log the error and continue system startup.)
0x00000001	SERVICE_ERROR_NORMAL (Log the error, display a message to the user, and continue system startup.)
0x00000002	SERVICE_ERROR_SEVERE (Switch to the registry's LastKnownGood control set and continue system startup.)
0x00000003	SERVICE_ERROR_CRITICAL (If system startup is not using the registry's LastKnownGood control set, switch to LastKnownGood and try again. If startup still fails, run a bug-check routine. Only the drivers that are needed for the system to startup should specify this value in their INF files.)

The **LoadOrderGroup** entry should be set to a load order group that is appropriate for the type of file system filter driver that you are developing. To choose a load order group, see Load Order Groups for File System Filter Drivers.

The **AddReg directive** refers to one or more INF writer-defined **AddRegistry** sections that contain any information to be stored in the registry for the newly installed service.

Note If the INF file will also be used for upgrading the driver after the initial install, the entries that are contained in the **AddRegistry** section should specify the 0x00000002 (FLG_ADDREG_NOCLOBBER) flag. Specifying this flag preserves the registry entries in HKLM\CurrentControlSet\Services when subsequent files are installed. For example:

```
[ExampleFileSystem.AddRegistry]
HKR,Parameters,ExampleParameter,0x00010003,1
```

DefaultUninstall Section (optional)

The **DefaultUninstall** section is optional but recommended if your driver can be uninstalled. It contains **DelFiles** and **DelReg** directives to remove files and registry entries.

In the following code example, the **DelFiles** directive removes the files that are listed in the MyLegacyFilter.DriverFiles and MyLegacyFilter.UserFiles sections of the driver's INF file:

```
[DefaultUninstall]
DelFiles   = MyLegacyFilter.DriverFiles, MyLegacyFilter.UserFiles
DelReg     = MyLegacyFilter.DelRegistry
```

The **DelReg** directive refers to one or more INF writer-defined **DelRegistry** sections that contain any information to be removed from the registry for the service that is being uninstalled.

DefaultUninstall.Services Section (optional)

The **DefaultUninstall.Services** section is optional but recommended if your driver can be uninstalled. It contains **DelService** directives to remove the file system filter driver's services.

In the following code example, the **DelService** directive removes the MyLegacyFilter service from the operating system.

```
[DefaultUninstall.Services]
DelService = MyLegacyFilter,0x200
```

Note The **DelService** directive should always specify the 0x200 (SPSVCINST_STOPSERVICE) flag to stop the service before it is deleted.

Strings Section (required)

The **Strings** section defines each %strkey% token that is used in the INF file, as shown in the following example.

```
[Strings]
Msft                       = "Microsoft Corporation"
MyLegacyFilterServiceDesc = "MyLegacyFilterFilter Driver"
MyLegacyFilterServiceName = "MyLegacyFilter"
MyLegacyFilterRegistry     = "system\currentcontrolset\services\MyLegacyFilter"
MyLegacyFilterMaxRecords  = "MaxRecords"
MyLegacyFilterMaxNames     = "MaxNames"
MyLegacyFilterDebugFlags  = "DebugFlags"
Disk1                      = "MyLegacyFilter Source Media"
```

You can create a single international INF file by creating additional locale-specific **Strings.***LanguageID* sections in the INF file. For more information about international INF files, see Creating International INF Files.

Load Order Groups for File System Filter Drivers

4/26/2017 • 3 min to read • Edit Online

Microsoft Windows XP and later operating systems provide a dedicated set of load order groups for file system filter drivers that are loaded at system startup time. On operating systems before Windows XP, filter drivers could use only the "filter" and "file system" load order groups.

A filter can attach only to the top of an existing file system driver stack and cannot attach in the middle of a stack. As a result, load order groups are important to file system filter drivers, because the earlier a filter driver loads, the lower it can attach on the file system driver stack.

The following rules about load order groups determine when a file system filter driver will be loaded:

- A file system filter driver that specifies a particular load order group is loaded at the same time as other filter drivers in that group.
- Within each load order group, filter drivers are loaded in random order.
- If a file system filter driver does not specify a load order group, it is loaded after all of the other drivers of the same start type that do specify a load order group.

The following table lists the system-defined load order groups for file system filter drivers. For each load order group, the Load Order Group column contains the value that should be specified for that group in the **LoadOrderGroup** entry in the **Version section** of a filter's INF file.

Note that the load order groups are listed as they appear on the stack, which is the reverse of the order in which they are loaded.

LOAD ORDER GROUP	DESCRIPTION
Filter	This group is the same as the "filter" load order group that was available on Windows 2000 and earlier. This group loads last and thus attaches furthest from the file system.
FSFilter Top	This group is provided for filter drivers that must attach above all other FSFilter types.
FSFilter Activity Monitor	This group includes filter drivers that observe and report on file I/O.
FSFilter Undelete	This group includes filter drivers that recover deleted files.
FSFilter Anti-Virus	This group includes filters that detect and disinfect viruses during file I/O.
FSFilter Replication	This group includes filter drivers that replicate file data to remote servers.

LOAD ORDER GROUP	DESCRIPTION
FSFilter Continuous Backup	This group includes filter drivers that replicate file data to backup media.
FSFilter Content Screener	This group includes filter drivers that prevent the creation of specific files or file content.
FSFilter Quota Management	This group includes filter drivers that provide enhanced file system quotas.
FSFilter System Recovery	This group includes filter drivers that perform operations to maintain operating system integrity, such as the System Restore (SR) filter.
FSFilter Cluster File System	This group includes filter drivers that are used in products that provide file server metadata across a network.
FSFilter HSM	This group includes filter drivers that perform hierarchical storage management.
FSFilter Imaging	This group includes ZIP-like filter drivers that provide a virtual namespace. This load group is available on Windows Vista and later versions of the operating system.
FSFilter Compression	This group includes filter drivers that perform file data compression.
FSFilter Encryption	This group includes filter drivers that encrypt and decrypt data during file I/O.
FSFilter Virtualization	This group includes filter drivers that virtualize the file path, such as the Least Authorized User (LUA) filter driver added in Windows Vista. This load group is available on Windows Vista and later versions of the operating system.
FSFilter Physical Quota Management	This group includes filter drivers that manage quotas by using physical block counts.
FSFilter Open File	This group includes filter drivers that provide snapshots of already open files.
FSFilter Security Enhancer	This group includes filter drivers that apply lockdown and enhanced access control lists (ACLs).

LOAD ORDER GROUP	DESCRIPTION
FSFilter Copy Protection	This group includes filter drivers that check for out-of-band data on media.
FSFilter Bottom	This group is provided for filter drivers that must attach below all other FSFilter types.
FSFilter System	Reserved for internal use. This group includes the HSM and SIS filter drivers.
FSFilter Infrastructure	Reserved for internal use. This group loads first and thus attaches closest to the file system.

File System Filter Driver Classes and Class GUIDs

4/26/2017 • 1 min to read • Edit Online

Microsoft Windows XP and later operating systems provide setup classes for file system filter drivers. These classes provide a subset of the functionality that system-supplied device setup classes provide for hardware devices. (For more information about setup classes for hardware devices, see Device Setup Classes.)

Each setup class is associated with a class GUID. The system-defined class GUIDs are defined in devguid.h.

This topic lists the setup classes for file system filter drivers. In the definition for each class, the **Class** and **ClassGuid** entries contain the values that you should specify in the **INF Version section** of a filter's INF file. Your filter driver should use the class and GUID that match the load order group that is specified in your driver's INF file.

Supplying the appropriate class GUID value in the INF file for a device, rather than, or in addition to, the **Class** = *class-name* entry, significantly improves the performance of searching system INF files.

The following list includes system-defined classes and class GUIDs for file system filter drivers. The entries in this list correspond to the load order groups that were created for file system filter drivers in Windows XP and later operating systems.

Note Three load order groups do not appear in the list, because they are not considered setup classes and thus do not have class GUIDs assigned to them: Filter, FSFilter Top, and FSFilter Bottom.

FSFilter Activity Monitor
Class = ActivityMonitor
ClassGuid = {b86dff51-a31e-4bac-b3cf-e8cfe75c9fc2}

FSFilter Undelete
Class = Undelete
ClassGuid = {fe8f1572-c67a-48c0-bbac-0b5c6d66cafb}

FSFilter Anti-Virus
Class = AntiVirus
ClassGuid = {b1d1a169-c54f-4379-81db-bee7d88d7454}

FSFilter Replication
Class = Replication
ClassGuid = {48d3ebc4-4cf8-48ff-b869-9c68ad42eb9f}

FSFilter Continuous Backup
Class = ContinuousBackup
ClassGuid = {71aa14f8-6fad-4622-ad77-92bb9d7e6947}

FSFilter Content Screener
Class = ContentScreener
ClassGuid = {3e3f0674-c83c-4558-bb26-9820e1eba5c5}

FSFilter Quota Management
Class = QuotaManagement
ClassGuid = {8503c911-a6c7-4919-8f79-5028f5866b0c}

FSFilter Cluster File System
Class = CFSMetaDataServer

ClassGuid = {cdcf0939-b75b-4630-bf76-80f7ba655884}

FSFilter HSM
Class = HSM
ClassGuid = {d546500a-2aeb-45f6-9482-f4b1799c3177}

FSFilter Compression
Class = Compression
ClassGuid = {f3586baf-b5aa-49b5-8d6c-0569284c639f}

FSFilter Encryption
Class = Encryption
ClassGuid = {a0a701c0-a511-42ff-aa6c-06dc0395576f}

FSFilter Physical Quota Management
Class = PhysicalQuotaManagement
ClassGuid = {6a0a8e78-bba6-4fc4-a709-1e33cd09d67e}

FSFilter Open File
Class = OpenFileBackup
ClassGuid = {f8ecafa6-66d1-41a5-899b-66585d7216b7}

FSFilter Security Enhancer
Class = SecurityEnhancer
ClassGuid = {d02bc3da-0c8e-4945-9bd5-f1883c226c8c}

FSFilter Copy Protection
Class = CopyProtection
ClassGuid = {89786ff1-9c12-402f-9c9e-17753c7f4375}

FSFilter System
Class = FSFilterSystem
ClassGuid = {5d1b9aaa-01e2-46af-849f-272b3f324c46}

FSFilter Infrastructure
Class = Infrastructure
ClassGuid = {e55fa6f9-128c-4d04-abab-630c74b1453a}

Using an INF File to Install a File System Filter Driver

4/26/2017 • 1 min to read • Edit Online

After you have created an INF file, you can use it to install, upgrade, and uninstall your file system filter driver. You can use the INF file alone or together with a batch file or a user-mode setup application.

Right-Click Install

To execute the **DefaultInstall** and **DefaultInstall.Services** sections of your INF file, you should do the following:

1. In Windows Explorer, right-click the INF file name. A shortcut menu will appear.
2. Click **Install**.

Note The shortcut menu appears only if the INF file contains a **DefaultInstall** section.

Command-Line or Batch File Install

To execute the **DefaultInstall** and **DefaultInstall.Services** sections of your INF file on the command line or by using a batch file installation, type the following command at the command prompt, or create and run a batch file that contains this command:

```
RUNDLL32.EXE SETUPAPI.DLL,InstallHinfSection DefaultInstall 132 path-to-inf\infname.inf
```

"Rundll32" and "InstallHinfSection" are described in the Tools and Setup and System Administration sections, respectively, of the Microsoft Windows SDK documentation.

Setup Application

InstallHinfSection can also be called from a setup application, as shown in the following code example:

```
InstallHinfSection(NULL,NULL,TEXT("DefaultInstall 132 path-to-inf\infname.inf"),0);
```

If you use a setup application to install your driver, observe the following guidelines:

- To prepare for eventual uninstall, the setup application should copy the driver INF file to an uninstall directory.
- If the setup application installs a user-mode application with the driver, this application should be listed in Add or Remove Programs in Control Panel so that the user can uninstall it if desired. Only one item should be listed, representing both the application and the driver.

  For more information about how to list your application in Add or Remove Programs, see "Removing an Application" in the Setup and System Administration section of the Windows SDK documentation.
- Setup applications should never copy driver INF files to the Windows INF file directory (*%windir%\INF*). SetupAPI copies the files there automatically as part of the **InstallHinfSection** call.

For more information about setup applications, see Writing a Device Installation Application.

Using an INF File to Uninstall a File System Filter Driver

4/26/2017 • 1 min to read • Edit Online

You can uninstall your driver by using an INF file together with a batch file or a user-mode uninstall application.

There is no "right-click uninstall" option.

Command-Line or Batch File Uninstall

To execute the **DefaultUninstall** and **DefaultUninstall.Services** sections of your INF file on the command line, type the following command at the command prompt, or create and run a batch file that contains this command:

```
RUNDLL32.EXE SETUPAPI.DLL,InstallHinfSection DefaultUninstall 132 path-to-uninstall-dir\infname.inf
```

Rundll32 and **InstallHinfSection** are described in the Tools and Setup and System Administration sections, respectively, of the Microsoft Windows SDK documentation.

Uninstall Application

You can also execute the **DefaultUninstall** and **DefaultUninstall.Services** sections of your INF file from an uninstall application, as shown in the following code example:

```
InstallHinfSection(NULL,NULL,TEXT("DefaultUninstall 132 path-to-uninstall-dir\infname.inf"),0);
```

If you use an application to uninstall your driver, observe the following guidelines:

- To prepare for eventual uninstall, a setup application should copy the driver INF file to an uninstall directory.
- In the **DefaultUninstall.Services** section of the INF file, the **DelService** directive should always specify the 0x200 (SPSVCINST_STOPSERVICE) flag to stop the service before it is deleted.
- If a user-mode application was installed with the driver, this application should be listed in Add or Remove Programs in Control Panel so that the user can uninstall it if desired. Only one item should be listed, representing both the application and the driver.

  For more information about how to list your application in Add or Remove Programs, see "Removing an Application" in the Setup and System Administration section of the Microsoft Windows SDK documentation.
- An uninstall application should not delete the INF file (or its associated PNF file) from the Windows INF file directory (*%windir%\INF*).
- Some filter driver files cannot safely be removed when the application is uninstalled. These files should not be listed in the **DefaultUninstall.Services** section of the INF file.

For more information about uninstall applications, see Writing a Device Installation Application.

Initializing a File System Filter Driver

4/26/2017 • 1 min to read • Edit Online

The **DriverEntry** routine for initializing a file system filter driver is very similar to the **DriverEntry** routine for initializing a device driver. After a driver is loaded, the same component that loaded the driver also initializes the driver by calling the driver's **DriverEntry** routine. For file system filter drivers, the component that loads the driver is either the I/O Manager (for filters whose start type is SERVICE_BOOT_START) or the Service Control Manager (for other start types).

The **DriverEntry** routine runs in a system thread context at IRQL = PASSIVE_LEVEL. This routine can be pageable and should be in an INIT segment so that it will be discarded. For more information about how to make your driver code pageable, see the Remarks section of **MmLockPagableCodeSection**.

The **DriverEntry** routine is defined as follows:

```
NTSTATUS
(*PDRIVER_INITIALIZE) (
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    );
```

This routine has two input parameters. The first, *DriverObject*, is the driver object that was created when the file system filter driver was loaded. The second, *RegistryPath*, is a pointer to a counted Unicode string that contains a path to the driver's registry key.

The **DriverEntry** routine for a file system filter driver performs the following steps:

Creating the Control Device Object

Registering IRP Dispatch Routines

Registering Fast I/O Dispatch Routines

Registering FsFilter Callback Routines

Performing Any Other Needed Initialization

[Optional] Registering Callback Routines

[Optional] Saving a Copy of the Registry Path String

Returning Status

Creating the Control Device Object

4/26/2017 • 2 min to read • Edit Online

A file system filter driver's **DriverEntry** routine usually begins by creating a *control device object*. The purpose of the control device object is to allow applications to communicate with the filter driver directly, even before the filter is attached to a file system or volume device object.

Note that file systems also create control device objects. When a file system filter driver attaches itself to a file system, rather than an individual file system volume, it does so by attaching itself to the file system's control device object.

The following example creates a control device object:

```
RtlInitUnicodeString(&nameString, MYLEGACYFILTER_FULLDEVICE_NAME);
status = IoCreateDevice(
        DriverObject,                  //DriverObject
        0,                             //DeviceExtensionSize
        &nameString,                   //DeviceName
        FILE_DEVICE_DISK_FILE_SYSTEM,  //DeviceType
        FILE_DEVICE_SECURE_OPEN,       //DeviceCharacteristics
        FALSE,                         //Exclusive
        &gControlDeviceObject);        //DeviceObject

RtlInitUnicodeString(&linkString, MYLEGACYFILTER_DOSDEVICE_NAME);
status = IoCreateSymbolicLink(&linkString, &nameString);
```

Unlike file systems, file system filter drivers are not required to name their control device objects. A user mode application can not access the filter driver with out a device name since a call to **IoRegisterDeviceInterface** is not valid for control device objects. If a non-**NULL** value is passed in the *DeviceName* parameter, this value becomes the name of the control device object. The **DriverEntry** routine can then call the **IoCreateSymbolicLink** routine, as shown in the preceding code example, to link the object's kernel-mode name to a user-mode name that is visible to applications.

Note The control device object is the only type of device object that can safely be named, because it is the only device object that is not attached to a driver stack. Thus, control device objects for file system filter drivers can optionally be named. Note that control device objects for file systems must be named. Filter device objects should never be named.

The value that is assigned to the *DeviceType* parameter should be one of the device types that are defined in ntifs.h, such as FILE_DEVICE_DISK_FILE_SYSTEM.

If a non-**NULL** value is passed in the *DeviceName* parameter, the *DeviceCharacteristics* flags must include FILE_DEVICE_SECURE_OPEN. This flag directs the I/O Manager to perform security checks on all open requests that are sent to the control device object. These security checks are made against the ACL for the named device object.

An effective way for a file system filter driver to identify its own control device object in dispatch routines is to compare the device pointer and a previously stored global pointer to the control device object. Thus, the preceding sample stores the *DeviceObject* pointer that was returned by **IoCreateDevice** into `gControlDeviceObject`, a globally defined pointer variable.

Registering IRP Dispatch Routines

4/26/2017 • 1 min to read • Edit Online

The *DriverObject* parameter of the filter driver's **DriverEntry** routine supplies a pointer to the filter driver's **driver object**. To register I/O request packet (IRP) dispatch routines, you must store the entry points of these routines into the **MajorFunction** member of the driver object. For example, a hypothetical "MyLegacyFilter" driver can set the entry points for its dispatch routine as follows:

```
for (i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++) {
    DriverObject->MajorFunction[i] = MyLegacyFilterDispatch;
}
DriverObject->MajorFunction[IRP_MJ_CREATE] = MyLegacyFilterCreate;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = MyLegacyFilterClose;
DriverObject->MajorFunction[IRP_MJ_FILE_SYSTEM_CONTROL] = MyLegacyFilterFsControl;
```

Note that the above **FOR** loop assigns a default dispatch routine for all IRP major function codes. This assignment is good practice, because otherwise the I/O Manager completes any unrecognized IRP with STATUS_INVALID_DEVICE_REQUEST by default. File system filter drivers should not reject unfamiliar IRPs in this way, because such requests are usually intended for another driver that is lower in the driver stack. For this reason, the default dispatch routine normally just passes the IRP down to the next-lower-level driver.

Registering Fast I/O Dispatch Routines

4/26/2017 • 1 min to read • Edit Online

The *DriverObject* parameter of the filter driver's **DriverEntry** routine supplies a pointer to the filter driver's **driver object**.

To register the file system filter driver's fast I/O dispatch routines, you must allocate and initialize a fast I/O dispatch table, store the entry points of the fast I/O dispatch routines into the table, and store the address of the table in the **FastIoDispatch** member of the driver object.

For example, a hypothetical "MyLegacyFilter" driver can set the entry points for its fast I/O dispatch routines as follows:

```
RtlZeroMemory(fastIoDispatch, sizeof(FAST_IO_DISPATCH));
fastIoDispatch->SizeOfFastIoDispatch = sizeof(FAST_IO_DISPATCH);
fastIoDispatch->FastIoCheckIfPossible = MyLegacyFilterIoCheckIfPossible;
fastIoDispatch->FastIoRead = MyLegacyFilterIoRead;
fastIoDispatch->FastIoWrite = MyLegacyFilterIoWrite;
fastIoDispatch->FastIoQueryBasicInfo = MyLegacyFilterIoQueryBasicInfo;
fastIoDispatch->FastIoQueryStandardInfo = MyLegacyFilterIoQueryStandardInfo;
fastIoDispatch->FastIoLock = MyLegacyFilterIoLock;
fastIoDispatch->FastIoUnlockSingle = MyLegacyFilterIoUnlockSingle;
fastIoDispatch->FastIoUnlockAll = MyLegacyFilterIoUnlockAll;
fastIoDispatch->FastIoUnlockAllByKey = MyLegacyFilterIoUnlockAllByKey;
fastIoDispatch->FastIoDeviceControl = MyLegacyFilterIoDeviceControl;
fastIoDispatch->FastIoDetachDevice = MyLegacyFilterIoDetachDevice;
fastIoDispatch->FastIoQueryNetworkOpenInfo = MyLegacyFilterIoQueryNetworkOpenInfo;
fastIoDispatch->MdlRead = MyLegacyFilterIoMdlRead;
fastIoDispatch->MdlReadComplete = MyLegacyFilterIoMdlReadComplete;
fastIoDispatch->PrepareMdlWrite = MyLegacyFilterIoPrepareMdlWrite;
fastIoDispatch->MdlWriteComplete = MyLegacyFilterIoMdlWriteComplete;
fastIoDispatch->FastIoReadCompressed = MyLegacyFilterIoReadCompressed;
fastIoDispatch->FastIoWriteCompressed = MyLegacyFilterIoWriteCompressed;
fastIoDispatch->MdlReadCompleteCompressed = MyLegacyFilterIoMdlReadCompleteCompressed;
fastIoDispatch->MdlWriteCompleteCompressed = MyLegacyFilterIoMdlWriteCompleteCompressed;
fastIoDispatch->FastIoQueryOpen = MyLegacyFilterIoQueryOpen;

DriverObject->FastIoDispatch = fastIoDispatch;
```

Registering FsFilter Callback Routines

4/26/2017 • 1 min to read • Edit Online

FsFilter notification callback routines are called before and after the underlying file system performs certain operations. For more information about FsFilter callback routines, see **FsRtlRegisterFileSystemFilterCallbacks**.

To register the FsFilter notification callback routines, you must allocate and initialize an FS_FILTER_CALLBACKS structure, store the entry points of the FsFilter callback routines in the structure, and pass the address of the structure in the *Callbacks* parameter to **FsRtlRegisterFileSystemFilterCallbacks**.

For example, a hypothetical "MyLegacyFilter" driver can register its FsFilter callback routines as follows:

```
fsFilterCallbacks.SizeOfFsFilterCallbacks = sizeof(FS_FILTER_CALLBACKS);
fsFilterCallbacks.PreAcquireForSectionSynchronization = MyLegacyFilterPreFsFilterOperation;
fsFilterCallbacks.PostAcquireForSectionSynchronization = MyLegacyFilterPostFsFilterOperation;
fsFilterCallbacks.PreReleaseForSectionSynchronization = MyLegacyFilterPreFsFilterOperation;
fsFilterCallbacks.PostReleaseForSectionSynchronization = MyLegacyFilterPostFsFilterOperation;
fsFilterCallbacks.PreAcquireForCcFlush = MyLegacyFilterPreFsFilterOperation;
fsFilterCallbacks.PostAcquireForCcFlush = MyLegacyFilterPostFsFilterOperation;
fsFilterCallbacks.PreReleaseForCcFlush = MyLegacyFilterPreFsFilterOperation;
fsFilterCallbacks.PostReleaseForCcFlush = MyLegacyFilterPostFsFilterOperation;
fsFilterCallbacks.PreAcquireForModifiedPageWriter = MyLegacyFilterPreFsFilterOperation;
fsFilterCallbacks.PostAcquireForModifiedPageWriter = MyLegacyFilterPostFsFilterOperation;
fsFilterCallbacks.PreReleaseForModifiedPageWriter = MyLegacyFilterPreFsFilterOperation;
fsFilterCallbacks.PostReleaseForModifiedPageWriter = MyLegacyFilterPostFsFilterOperation;

status = FsRtlRegisterFileSystemFilterCallbacks(DriverObject, &fsFilterCallbacks);
```

Performing Any Other Needed Initialization

4/26/2017 • 1 min to read • Edit Online

After registering IRP and fast I/O dispatch routines, your file system filter driver's **DriverEntry** routine can initialize additional global driver variables and data structures as needed.

[Optional] Registering Callback Routines

7/27/2017 • 1 min to read • Edit Online

Filter drivers can call **IoRegisterFsRegistrationChange** to register a callback routine to be called whenever a file system driver calls **IoRegisterFileSystem** or **IoUnregisterFileSystem** to register or unregister itself. Filter drivers register this callback routine so they can see new file systems enter the system and choose whether to attach to them.

Note File system filter drivers must never call **IoRegisterFileSystem** or **IoUnregisterFileSystem**. These routines are only for file systems.

Filter drivers that attach to volumes only when explicitly directed (for example, by a user-mode application) should not call **IoRegisterFsRegistrationChange**. Note, however, that a filter that uses this routine has the ability to attach to any given volume immediately after that volume is mounted. Using this routine does not guarantee that the filter will attach directly to the volume device object. But it does ensure that such a filter attaches before (and thus below) any filter that instead waits for a command from a user-mode application, because filters can attach only at the top of the current file system volume device stack.

[Optional] Saving a Copy of the Registry Path String

7/27/2017 • 1 min to read • Edit Online

Note This step is necessary only if the filter driver needs to use the registry path after the **DriverEntry** routine returns.

Save a copy of the *RegistryPath* string that was passed as input to **DriverEntry**. This parameter points to a counted Unicode string that specifies a path to the driver's registry key, **\Registry\Machine\System\CurrentControlSet\Services\***DriverName*, where *DriverName* is the name of the driver. If the *RegistryPath* string will be needed later, **DriverEntry** must save a copy of it, not just a pointer to it, because the pointer is no longer valid after the **DriverEntry** routine returns. You can use the **RtlCopyUnicodeString** routine to copy the *RegistryPath* source string to a destination string.

Returning Status

4/26/2017 • 1 min to read • Edit Online

A file system filter driver's **DriverEntry** routine normally returns STATUS_SUCCESS. However, if driver initialization fails, the **DriverEntry** routine should return an appropriate error status value.

If the **DriverEntry** routine returns a status value that is not a success status value, the system responds by unloading the driver. For this reason, the **DriverEntry** routine must always free any memory that was allocated for system resources, such as device objects, before returning a status value that is not a success status value.

Attaching a Filter to a File System or Volume

4/26/2017 • 1 min to read • Edit Online

A file system filter driver attaches itself to one or more mounted volumes and filters all I/O operations on them. But how does it determine which volumes to attach itself to? The sample filter drivers in the Windows Driver Kit (WDK) illustrate the two most common ways in which this is done:

- The end user can specify the volumes to filter by, for example, typing in the drive letters for the volumes. The end user's commands are relayed to the filter driver as a private **IRP_MJ_DEVICE_CONTROL** request.
- The file system filter driver can attach to one or more file system drivers, listen for **IRP_MJ_FILE_SYSTEM_CONTROL**, IRP_MN_MOUNT_VOLUME requests, and attach to volumes as they are mounted.

Note You should generally assume that the mapping of volumes to drive letters is one-to-many, not one-to-one. This is because of advanced storage features, such as dynamic volumes and volume mount points.

Note You should not assume that IRP_MN_MOUNT_VOLUME requests are always handled synchronously by the file system. For example, a floppy drive may be mounted asynchronously if there is no floppy disk in the drive. Thus your filter driver should be prepared to propagate the **PendingReturned** flag in its mount completion routine. For more information, see "Checking the PendingReturned Flag."

File system filter drivers can attach to, and filter I/O for, any file system volume. They cannot attach directly to storage devices, such as disk drives or partitions. Also, they cannot attach to individual directories or files.

For more information, see the following topics:

Creating the Filter Device Object

Attaching the Filter Device Object to the Target Device Object

Propagating the DO_BUFFERED_IO and DO_DIRECT_IO Flags

Propagating the FILE_DEVICE_SECURE_OPEN Flag

Clearing the DO_DEVICE_INITIALIZING Flag

Creating the Filter Device Object

4/26/2017 • 2 min to read • Edit Online

Call **IoCreateDevice** to create a filter device object to attach to a volume or file system stack, as in the following example:

```
status = IoCreateDevice(
          gFileSpyDriverObject,                     //DriverObject
          sizeof(MYLEGACYFILTER_DEVICE_EXTENSION),  //DeviceExtensionSize
          NULL,                                     //DeviceName
          DeviceObject->DeviceType,                 //DeviceType
          0,                                        //DeviceCharacteristics
          FALSE,                                    //Exclusive
          &newDeviceObject);                        //DeviceObject
```

In the above code snippet, *DeviceObject* is a pointer to the target device object to which the filter device object will be attached; *newDeviceObject* is a pointer to the filter device object itself.

Setting the *DeviceExtensionSize* parameter to **sizeof**(MYLEGACYFILTER_DEVICE_EXTENSION) causes a MYLEGACYFILTER_DEVICE_EXTENSION structure to be allocated for the filter device object. The newly created filter device object's **DeviceExtension** member is set to point to this structure. File system filter drivers usually define and allocate memory for a device extension for each filter device object. The structure and contents of the device extension are driver-specific. However, on Microsoft Windows XP and later, filter drivers should define a DEVICE_EXTENSION structure for filter driver objects that contains at least the following member:

```
PDEVICE_OBJECT AttachedToDeviceObject;
```

In the above call to **IoCreateDevice**, setting the *DeviceName* parameter to **NULL** specifies that the filter device object will not be named. Filter device objects are never named. Because a filter device object is attached to a file system or volume driver stack, assigning a name to the filter device object would create a system security hole.

The *DeviceType* parameter must always be set to the same device type as that of the target (file system or filter) device object to which the filter device object is being attached. It is important to propagate the device type in this way, because it is used by the I/O Manager and can be reported back to applications.

Note File systems and file system filter drivers should never set the *DeviceType* parameter to FILE_DEVICE_FILE_SYSTEM. This is not a valid value for this parameter. (The FILE_DEVICE_FILE_SYSTEM constant is intended only for use in defining FSCTL codes.)

Another reason why the *DeviceType* parameter is important is that many filters attach only to certain types of file systems. For example, a particular filter may attach to all local disk file systems, but not to CD-ROM file systems or remote file systems. Such filters determine the type of file system by examining the device type of the topmost device object in the file system or volume driver stack. In most cases, the topmost device object in the stack is a filter device object. Thus it is essential that all attached filter device objects have the same device type as that of the underlying file system or volume device object.

Attaching the Filter Device Object to the Target Device Object

4/26/2017 • 1 min to read • Edit Online

Call **IoAttachDeviceToDeviceStackSafe** to attach the filter device object to the filter driver stack for the target file system or volume.

```
devExt = myLegacyFilterDeviceObject->DeviceExtension;

status = IoAttachDeviceToDeviceStackSafe(
           myLegacyFilterDeviceObject,        //SourceDevice
           DeviceObject,                      //TargetDevice
           &devext->AttachedToDeviceObject);  //AttachedToDeviceObject
```

Note that the device object pointer received by the *AttachedToDeviceObject* output parameter can differ from *TargetDevice* if any other filters were already chained above the device object that is pointed to by *TargetDevice*.

Attaching to a File System by Name

Every file system is required to create one or more named control device objects. To attach to a particular file system directly, a file system filter driver passes the name of the appropriate file system control device object to **IoGetDeviceObjectPointer** to get a device object pointer. The following code snippet shows how to get such a pointer to one of the two control device objects for the RAW file system:

```
RtlInitUnicodeString(&nameString, L"\\Device\\RawDisk");

status = IoGetDeviceObjectPointer(
            &nameString,                    //ObjectName
            FILE_READ_ATTRIBUTES,           //DesiredAccess
            &fileObject,                    //FileObject
            &rawDeviceObject);              //DeviceObject

if (NT_SUCCESS(status)) {
            ObDereferenceObject(fileObject);
}
```

If the call to **IoGetDeviceObjectPointer** succeeds, the file system filter driver can then call **IoAttachDeviceToDeviceStackSafe** to attach to the returned control device object.

Note In addition to the control device object pointer (*rawDeviceObject*), **IoGetDeviceObjectPointer** returns a pointer to a file object (*fileObject*) that represents the device object in user mode. In the code snippet above, the file object is not needed, so it is closed by calling **ObDereferenceObject**. It is important to note that decrementing the reference count on the file object returned by **IoGetDeviceObjectPointer** causes the reference count on the device object to be decremented as well. Thus the *fileObject* and *rawDeviceObject* pointers should both be considered invalid after the above call to **ObDereferenceObject**, unless the reference count on the device object is incremented by an additional call to **ObReferenceObject** before **ObDereferenceObject** is called for the file object.

Propagating the DO_BUFFERED_IO and DO_DIRECT_IO Flags

7/21/2017 • 1 min to read • Edit Online

After attaching a filter device object to a file system or volume, always be sure to set or clear the DO_BUFFERED_IO and DO_DIRECT_IO flags as needed so that they match the values of the next-lower device object on the driver stack. (For more information about these flags, see Methods for Accessing Data Buffers.) An example of this follows:

```
if (FlagOn( DeviceObject->Flags, DO_BUFFERED_IO )) {
    SetFlag( myLegacyFilterDeviceObject->Flags, DO_BUFFERED_IO );
}
if (FlagOn( DeviceObject->Flags, DO_DIRECT_IO )) {
    SetFlag(myLegacyFilterDeviceObject->Flags, DO_DIRECT_IO );
}
```

In the above code snippet, *DeviceObject* is a pointer to the device object to which the filter device object has just been attached; myLegacyFilter *DeviceObject* is a pointer to the filter device object itself.

Propagating the FILE_DEVICE_SECURE_OPEN Flag

7/21/2017 • 1 min to read • Edit Online

After attaching a filter device object to a file system (but not to a volume), always be sure to set the FILE_DEVICE_SECURE_OPEN flag on the filter device object as needed to so that it matches the value of the next-lower device object on the driver stack. (For more information about this flag, see Specifying Device Characteristics in the Kernel Architecture Design Guide and **DEVICE_OBJECT** in the Kernel Reference.) An example of this follows:

```
if (FlagOn( DeviceObject->Characteristics, FILE_DEVICE_SECURE_OPEN )) {
    SetFlag(myLegacyFilterDeviceObject->Characteristics, FILE_DEVICE_SECURE_OPEN );
}
```

In the above code snippet, *DeviceObject* is a pointer to the device object to which the filter device object has just been attached; myLegacyFilter *DeviceObject* is a pointer to the filter device object itself.

Clearing the DO_DEVICE_INITIALIZING Flag

7/21/2017 • 1 min to read • Edit Online

After attaching a filter device object to a file system or volume, always be sure to clear the DO_DEVICE_INITIALIZING flag on the filter device object. (For more information about this flag, see **DEVICE_OBJECT** in the Kernel Reference.) This can be done as follows using the **ClearFlag** macro defined in *ntifs.h*:

```
ClearFlag(NewDeviceObject->Flags, DO_DEVICE_INITIALIZING);
```

When the filter device object is created, **IoCreateDevice** sets the DO_DEVICE_INITIALIZING flag on the device object. After the filter is successfully attached, this flag must be cleared. Note that if this flag is not cleared, no more filter drivers can attach to the filter chain because the call to **IoAttachDeviceToDeviceStackSafe** will fail.

Note It is not necessary to clear the DO_DEVICE_INITIALIZING flag on device objects that are created in DriverEntry, because this is done automatically by the I/O Manager. However, your driver should clear this flag on all other device objects that it creates.

Filtering IRPs and Fast I/O

4/26/2017 • 1 min to read • Edit Online

Note For optimal reliability and performance, we recommend using file system minifilter drivers instead of legacy file system filter drivers. Also, legacy file system filter drivers can't attach to direct access (DAX) volumes. For more about file system minifilter drivers, see Advantages of the Filter Manager Model. To port your legacy driver to a minifilter driver, see Guidelines for Porting Legacy Filter Drivers.

A file system filter driver filters I/O requests for one or more file systems or file system volumes. Each I/O request appears as an I/O request packet (IRP) or fast I/O request. IRPs are I/O system structures that are handled by a driver's IRP dispatch routines. Fast I/O requests are handled by the driver's fast I/O callback routines.

When a filter driver is initialized, its **DriverEntry** routine registers the filter driver's IRP dispatch routines and fast I/O callback routines. Only one set of these routines can be registered for each filter driver.

Some types of IRPs have fast I/O equivalents, and some fast I/O requests have IRP equivalents. However, IRPs handle many types of I/O that fast I/O cannot. Also, certain specialized fast I/O routines are used to preacquire file system resources for the Cache Manager or Memory Manager without creating an IRP. Thus, for the most part, IRPs and fast I/O requests perform separate roles in I/O operations.

This section covers the following topics:

IRPs Are Different From Fast I/O

Types of File System Filter Driver Device Objects

IRPs Are Different From Fast I/O

4/26/2017 • 1 min to read • Edit Online

IRPs are the default mechanism for requesting I/O operations. IRPs can be used for synchronous or asynchronous I/O, and for cached or noncached I/O. IRPs are also used for paging I/O. The Memory Manager processes page faults by sending appropriate IRPs to the file system.

Fast I/O is specifically designed for rapid synchronous I/O on cached files. In fast I/O operations, data is transferred directly between user buffers and the system cache, bypassing the file system and the storage driver stack. (Storage drivers do not use fast I/O.) If all of the data to be read from a file is resident in the system cache when a fast I/O read or write request is received, the request is satisfied immediately. Otherwise, a page fault can occur, causing one or more IRPs to be generated. When this happens, the fast I/O routine either returns **FALSE**, or puts the caller into a wait state until the page fault is processed. If the fast I/O routine returns **FALSE**, the requested operation failed and the caller must create an IRP.

File systems and file system filters are required to support IRPs, but they are not required to support fast I/O. However, file systems and file system filters must implement fast I/O routines. Even if file systems and file system filters do not support fast I/O, they must define a fast I/O routine that returns **FALSE** (that is, the fast I/O routine does not implement any functionality). When the I/O Manager receives a request for synchronous file I/O (other than paging I/O), it invokes the fast I/O routine first. If the fast I/O routine returns **TRUE**, the operation was serviced by the fast I/O routine. If the fast I/O routine returns **FALSE**, the I/O Manager creates and sends an IRP instead.

File system filter drivers are not required to support I/O on control device objects. However, filter device objects that are attached to file systems or volumes are required to pass all unrecognized or unwanted IRPs to the next-lower driver on the driver stack. In addition, filter device objects that are attached to volumes must implement FastIoDetachDevice.

Types of Device Objects Used by File System Filter Drivers

4/26/2017 • 1 min to read • Edit Online

To write effective dispatch routines for IRPs and fast I/O requests, it is important to understand the various types of target device objects through which your filter driver can receive these requests.

Unlike drivers for Plug and Play (PnP) devices, such as disk drives, file system filter drivers do not create functional or physical device objects. Instead, they create control device objects and filter device objects. The *control device object* (CDO) represents the filter driver to the system and to user-mode applications. The *filter device object* (filter DO) performs the actual work of filtering a specific file system or volume. A file system filter driver normally creates one CDO and one or more filter DOs.

For each type of I/O request received by your filter driver, the target device object can be one or more of the following:

- The filter driver's CDO, which is not attached to a driver stack
- A filter device object that is chained above a file system CDO in the global file system queue
- A filter device object that is chained above a mounted file system volume device object (VDO)

A filter driver can register only one set of dispatch routines for IRPs and one for fast I/O requests. Thus, each type of request must be handled by a single routine. This routine must ascertain which of the target device objects listed above received the request, so that it can handle the request appropriately.

This section includes:

The Filter Driver's Control Device Object

Filter Device Object Attached to a File System

Filter Device Object Attached to a Volume

The Filter Driver's Control Device Object

4/26/2017 • 1 min to read • Edit Online

Unlike a file system, which is required to create and use a named control device object (CDO), a file system filter driver is not required to have a CDO. If it does, this CDO, which can optionally be named, represents the filter driver to the system. Its role is to receive I/O requests from a user-mode application (or, less commonly, another kernel-mode driver), and to act on them appropriately.

Most file system filter drivers create and use a CDO. However, support for I/O requests on the CDO is optional. To provide this support, when the filter driver calls **IoCreateDevice** to create the CDO, it must supply a device name for the object. The user-mode application can then obtain a handle to the named CDO by calling **CreateFile**, supplying the user-mode version of the device name.

For example, consider a hypothetical "MyLegacyFilter" kernel-mode driver. This driver can create a CDO with the name:

```
\Device\MyLegacyFilter
```

and calls **IoCreateSymbolicLink** to link this name to an equivalent user-mode-visible name. This is done so that MyLegacyFilter's user-mode application can open a handle to the kernel-mode driver's CDO by supplying the name:

```
\\.\MyLegacyFilter
```

when it calls **CreateFile**.

Types of I/O Requests That Are Sent to the Filter Driver's Control Device Object

File system filter drivers are not required to support any I/O operations on the control device object (CDO). However, most filters permit the following types of I/O requests to be sent to the filter's CDO:

- **IRP_MJ_CREATE** (to open a handle to the target device object, and give that handle to a user application)
- **IRP_MJ_CLEANUP** (to close a user-mode application's handle to the target device object)
- **IRP_MJ_CLOSE** (to close the last remaining open handle to the target device object)
- **IRP_MJ_DEVICE_CONTROL**, **IRP_MJ_FILE_SYSTEM_CONTROL**, or **FastIoDeviceControl** (to send private IOCTLs or FSCTLs to the filter driver)

Note that, unlike all other device objects that a file system filter driver creates, the CDO is not attached to a driver stack. No device objects are attached above or below the filter driver's CDO. Thus, for any I/O request it receives, the CDO can safely assume that it is the sole intended recipient. This is not true for filter device objects or file system CDOs. Accordingly, the CDO must eventually complete every IRP it receives. For fast I/O requests, it must return **TRUE** or **FALSE**.

Filter Device Object Attached to a File System

4/26/2017 • 1 min to read • Edit Online

To filter an entire file system, a file system filter driver creates a filter device object and attaches it above a file system driver's CDO in the global file system queue.

Types of I/O Requests That Are Sent to a File System

A filter device object that is attached above a file system can generally expect to receive the following types of I/O requests:

IRP_MJ_DEVICE_CONTROL

IRP_MJ_FILE_SYSTEM_CONTROL

IRP_MJ_SHUTDOWN

If the file system supports opening handles to its control device object, filters can expect to see the following types of I/O requests as well:

IRP_MJ_CLEANUP

IRP_MJ_CLOSE

IRP_MJ_CREATE

File system filter device objects attached to file systems are required to pass all unrecognized or unwanted IRPs to the next-lower driver on the driver stack by default.

Filter Device Object Attached to a Volume

4/26/2017 • 1 min to read • Edit Online

To filter a volume, a filter driver creates a filter device object and attaches it above the volume device object for the volume.

Types of I/O Requests That Are Sent to a Volume

A filter device object that is attached above a volume can generally expect to receive the following types of I/O requests:

IRP_MJ_CLEANUP

IRP_MJ_CLOSE

IRP_MJ_CREATE

IRP_MJ_DEVICE_CONTROL

IRP_MJ_DIRECTORY_CONTROL

IRP_MJ_FILE_SYSTEM_CONTROL

IRP_MJ_FLUSH_BUFFERS

IRP_MJ_INTERNAL_DEVICE_CONTROL

IRP_MJ_LOCK_CONTROL

IRP_MJ_PNP

IRP_MJ_QUERY_EA

IRP_MJ_QUERY_INFORMATION

IRP_MJ_QUERY_QUOTA

IRP_MJ_QUERY_SECURITY

IRP_MJ_QUERY_VOLUME_INFORMATION

IRP_MJ_READ

IRP_MJ_SET_EA

IRP_MJ_SET_INFORMATION

IRP_MJ_SET_QUOTA

IRP_MJ_SET_SECURITY

IRP_MJ_SET_VOLUME_INFORMATION

IRP_MJ_SHUTDOWN

IRP_MJ_WRITE

FastIoCheckIfPossible

FastIoDetachDevice

FastIoDeviceControl

FastIoLock

FastIoQueryBasicInfo

FastIoQueryNetworkOpenInfo

FastIoQueryOpen

FastIoQueryStandardInfo

FastIoRead

FastIoReadCompressed

FastIoUnlockAll

FastIoUnlockAllByKey

FastIoUnlockSingle

FastIoWrite

FastIoWriteCompressed

MdlRead

MdlReadComplete

MdlReadCompleteCompressed

MdlWriteComplete

MdlWriteCompleteCompressed

PrepareMdlWrite

File system filter device objects, attached to volumes, are required to pass all unrecognized or unwanted IRPs to the next-lower driver on the driver stack by default. In addition, they must implement **FastIoDetachDevice**.

Note On Microsoft Windows XP and later, the following fast I/O callback routines are obsolete and should not be used by file system filter drivers: **AcquireForCcFlush**

AcquireFileForNtCreateSection

AcquireForModWrite

ReleaseForCcFlush

ReleaseFileForNtCreateSection

ReleaseForModWrite

For more information, see the reference entry for **FsRtlRegisterFileSystemFilterCallbacks**.

Writing IRP Dispatch Routines

4/26/2017 • 2 min to read • Edit Online

Note For optimal reliability and performance, we recommend using file system minifilter drivers instead of legacy file system filter drivers. Also, legacy file system filter drivers can't attach to direct access (DAX) volumes. For more about file system minifilter drivers, see Advantages of the Filter Manager Model. To port your legacy driver to a minifilter driver, see Guidelines for Porting Legacy Filter Drivers.

File system filter drivers use dispatch routines that are similar to those used in device drivers. A *dispatch routine* handles one or more types of IRPs. (The *type* of an IRP is determined by its major function code.) The driver's DriverEntry routine *registers* dispatch routine entry points by storing them in the driver object's dispatch table. When an IRP is sent to the driver, the I/O subsystem calls the appropriate dispatch routine based on the IRP's major function code.

Every IRP dispatch routine is defined as follows:

```
NTSTATUS
(*PDRIVER_DISPATCH) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );
```

File system filter driver dispatch routines are most often called at IRQL PASSIVE_LEVEL, in the context of the thread that originated the I/O request, which is commonly a user-mode application thread. However, there are some exceptions to this rule. For example, page faults cause read and write dispatch routines to be called at IRQL APC_LEVEL. These exceptions are summarized in a table in Dispatch Routine IRQL and Thread Context. Unfortunately, it is not currently possible to prevent drivers in the filter chain from calling **IoCallDriver** at IRQL > PASSIVE_LEVEL (for example, by failing to release a spinlock or fast mutex). Nevertheless, it is strongly recommended that filter dispatch routines always call **IoCallDriver** at the same IRQL at which they were called.

Dispatch routines can be made pageable, provided that they meet the criteria described in the Making Drivers Pageable section of the Kernel-Mode Driver Architecture Design Guide.

If a file system filter driver has a control device object (CDO), its dispatch routines must be able to detect and handle cases where the IRP's target device object is the CDO rather than a volume device object (VDO) for a mounted volume. For more information about the CDO, see The Filter Driver's Control Device Object.

This section discusses the following topics:

Completing the IRP

4/26/2017 • 1 min to read • Edit Online

Every dispatch routine receives a pointer to the IRP's target device object in its *DeviceObject* parameter. If the filter driver has a control device object (CDO), the dispatch routine should check whether the *DeviceObject* pointer points to the filter driver's CDO before performing any processing on the IRP.

File system filter drivers are not required to support any I/O operations that are sent specifically to the CDO. (For more information about commonly supported operations, see The Filter Driver's Control Device Object.) However, the CDO must complete every IRP that is sent to it.

To *complete* an IRP, a dispatch routine must perform all of the following steps:

1. Set **Irp->IoStatus.Status** to an appropriate NTSTATUS value.
2. Call **IoCompleteRequest** to return the IRP to the I/O Manager.
3. Return the same status value as in step 1 to the caller.

Completing an IRP is sometimes referred to as "succeeding" or "failing" the IRP:

- To *succeed* an IRP means to complete it with a success or informational NTSTATUS value such as STATUS_SUCCESS.
- To *fail* an IRP means to complete it with an error or warning NTSTATUS value such as STATUS_INVALID_DEVICE_REQUEST or STATUS_BUFFER_OVERFLOW.

NTSTATUS values are defined in ntstatus.h. These values fall into four categories: success, informational, warning, and error. For more information, see Using NTSTATUS Values.

Note Although STATUS_PENDING is a success NTSTATUS value, it is a programming error to complete an IRP with STATUS_PENDING.

Passing the IRP Down to Lower-Level Drivers

4/26/2017 • 1 min to read • Edit Online

By default every dispatch routine, after checking the IRP's target device object, must pass the IRP down to the next-lower-level device driver by calling **IoCallDriver**. It is especially important that your filter driver pass down any IRPs that it does not recognize instead of simply failing them. Failing unfamiliar IRPs can cause the operating system itself to fail in unexpected ways. For example, failing IRP_MJ_PNP requests in a file system filter driver can interfere with power management by preventing system hibernation. This is true despite the fact that file system filter drivers are not involved in power management and do not receive **IRP_MJ_POWER** requests.

Returning Status from Dispatch Routines

4/26/2017 • 1 min to read • Edit Online

Except when completing an IRP, a dispatch routine that does not set a completion routine should always return the NTSTATUS value returned by **IoCallDriver**. Unless this value is STATUS_PENDING, it must match the value of **Irp->IoStatus.Status** set by the driver that completed the IRP.

A dispatch routine that sets a completion routine that might post the IRP to a work queue should do one of the following:

- Return the NTSTATUS value that was returned by **IoCallDriver**.
- Wait for the completion routine to signal an event and return the value of **Irp->IoStatus.Status**.
- Mark the IRP pending, post it to a work queue, and return STATUS_PENDING.
- If the completion routine might post the IRP to a work queue, the dispatch routine must mark the IRP pending and return STATUS_PENDING.

Which of these behaviors is correct, or even possible, depends on the specific operation. Some operations, such as directory change notification, cannot be made synchronous; and some, such as oplocks, cannot be made asynchronous.

For more information about returning status from a dispatch routine, see Constraints on Dispatch Routines.

Example: Passing the IRP Down Without Setting a Completion Routine

4/26/2017 • 1 min to read • Edit Online

To pass the IRP down to lower-level drivers without setting a completion routine, a dispatch routine must do the following:

- Call **IoSkipCurrentIrpStackLocation** to remove the current IRP stack location, so that the I/O Manager will not look for a completion routine there when it performs completion processing on the IRP.
- Call **IoCallDriver** to pass the IRP down to the next lower-level driver.

This technique is illustrated in the following code examples:

```
IoSkipCurrentIrpStackLocation ( Irp );
return IoCallDriver ( NextLowerDriverDeviceObject, Irp );
```

Or, equivalently:

```
IoSkipCurrentIrpStackLocation ( Irp );
status = IoCallDriver ( NextLowerDriverDeviceObject, Irp );
/* log or debugprint the status value here */
return status;
```

In these examples, the first parameter in the call to **IoCallDriver** is a pointer to the next-lower-level filter driver's device object. The second parameter is a pointer to the IRP.

Advantages of This Approach

The technique shown above (calling **IoSkipCurrentIrpStackLocation**) is simple and efficient and should be used in all cases where the driver passes the IRP down the driver stack without registering a completion routine.

Disadvantages of This Approach

After **IoCallDriver** is called, the IRP pointer that was passed to **IoCallDriver** is no longer valid and cannot safely be dereferenced. If the driver needs to perform further processing or cleanup after the IRP has been processed by lower-level drivers, it must set a completion routine before sending the IRP down the driver stack. For more information about writing and setting completion routines, see Using Completion Routines.

If you call **IoSkipCurrentIrpStackLocation** for an IRP, you cannot set a completion routine for it.

Constraints on Dispatch Routines

4/26/2017 • 3 min to read • Edit Online

The following guidelines briefly discuss how to avoid common programming errors in dispatch routines for file system filter drivers.

IRQL-Related Constraints

Note: For information about which types of IRPs are used in paging I/O, see Dispatch Routine IRQL and Thread Context.

- Dispatch routines in the paging I/O path should never call **IoCallDriver** at any IRQL above APC_LEVEL. If a dispatch routine raises IRQL, it must lower it before calling **IoCallDriver**.
- Dispatch routines in the paging path, such as read and write, cannot safely call any kernel-mode routines that require callers to be running at IRQL PASSIVE_LEVEL.
- Dispatch routines that are in the paging file I/O path cannot safely call any kernel-mode routines that require a caller to be running at IRQL < DISPATCH_LEVEL.
- Dispatch routines that are not in the paging I/O path should never call **IoCallDriver** at any IRQL above PASSIVE_LEVEL. If a dispatch routine raises IRQL, it must lower it before calling **IoCallDriver**.

Constraints on Processing IRPs

- If the IRP parameters include any user-space addresses, these must be validated before they are used. For more information, see Errors in Buffered I/O.
- Additionally, if the IRP contains an IOCTL or FSCTL buffer that was sent from a 32-bit platform to a 64-bit platform, the buffer contents may need to be thunked. For more information, see Supporting 32-Bit I/O in Your 64-Bit Driver.
- Unlike file systems, file system filter drivers should never call **FsRtlEnterFileSystem** or **FsRtlExitFileSystem** except before calling **ExAcquireFastMutexUnsafe** or **ExAcquireResourceExclusiveLite**. **FsRtlEnterFileSystem** and **FsRtlExitFileSystem** disable normal kernel APCs, which are needed by most file systems.

Constraints on Completing IRPs

- When completing IRPs, file system filter drivers should use only success and error status values.
- Although STATUS_PENDING is a success NTSTATUS value, it is a programming error to complete an IRP with STATUS_PENDING.
- After a dispatch routine calls **IoCompleteRequest**, the IRP pointer is no longer valid and cannot safely be dereferenced.

Constraints on Setting a Completion Routine

Note: For information about setting completion routines, see Using Completion Routines.

- When a dispatch routine calls **IoSetCompletionRoutine**, it can optionally pass a *Context* pointer to a structure for the completion routine to use when processing the given IRP. This structure must be allocated from nonpaged pool, because the completion routine can be called IRQL DISPATCH_LEVEL.
- If a dispatch routine sets a completion routine that may return STATUS_MORE_PROCESSING_REQUIRED, it must do one of the following to prevent the I/O Manager from completing the IRP prematurely:

    - Mark the IRP pending, call **IoCallDriver**, and return STATUS_PENDING.
    - Call **KeWaitForSingleObject** to wait for the completion routine to execute, then call **IoCompleteRequest** to complete the IRP.

Constraints on Passing IRPs Down

- After a dispatch routine calls **IoCallDriver**, the IRP pointer is no longer valid and cannot safely be dereferenced, unless the dispatch routine waits for the completion routine to signal that it has been called.
- It is a programming error to call **PoCallDriver** from a file system filter driver. (**PoCallDriver** is used to pass IRP_MJ_POWER requests to lower-level drivers. File system filter drivers never receive IRP_MJ_POWER requests.)

Constraints on Returning Status

- Except when completing an IRP, a dispatch routine that does not set a completion routine should always return the NTSTATUS value returned by **IoCallDriver**. Unless this value is STATUS_PENDING, it must match the value of **Irp->IoStatus.Status** set by the driver that completed the IRP.
- When **IoCallDriver** returns STATUS_PENDING, the dispatch routine should also return STATUS_PENDING, unless it waits for the completion routine to signal an event.
- When posting the IRP to a worker queue for later processing, the dispatch routine should mark the IRP pending and return STATUS_PENDING.
- When setting a completion routine that might post the IRP to a worker queue for later processing, the dispatch routine should mark the IRP pending and return STATUS_PENDING.
- A dispatch routine that marks an IRP pending must return STATUS_PENDING.
- Oplock operations cannot be pended (posted to a worker queue), and dispatch routines cannot return STATUS_PENDING for them.

Constraints on Posting IRPs to a Work Queue

- If a dispatch routine posts IRPs to a work queue, it must call **IoMarkIrpPending** before posting each IRP to the worker queue. Otherwise, the IRP could be dequeued, completed by another driver routine, and freed by the system before the call to **IoMarkIrpPending** occurs, thereby causing a crash.

Dispatch Routine IRQL and Thread Context

4/26/2017 • 1 min to read • Edit Online

The following table summarizes the IRQL and thread context requirements for file system filter driver dispatch routines.

DISPATCH ROUTINE	CALLER'S IRQL:	CALLER'S THREAD CONTEXT:
Cleanup	PASSIVE_LEVEL	Nonarbitrary
Close	APC_LEVEL	Arbitrary
Create	PASSIVE_LEVEL	Nonarbitrary
DeviceControl (except paging I/O)	PASSIVE_LEVEL	Nonarbitrary
DeviceControl (paging I/O path)	APC_LEVEL	Arbitrary
DirectoryControl	APC_LEVEL	Arbitrary
FlushBuffers	PASSIVE_LEVEL	Nonarbitrary
FsControl (except paging I/O)	PASSIVE_LEVEL	Nonarbitrary
FsControl (paging I/O path)	APC_LEVEL	Arbitrary
LockControl	PASSIVE_LEVEL	Nonarbitrary
PnP	PASSIVE_LEVEL	Arbitrary
QueryEa	PASSIVE_LEVEL	Nonarbitrary
QueryInformation	PASSIVE_LEVEL	Nonarbitrary
QueryQuota	PASSIVE_LEVEL	Nonarbitrary
QuerySecurity	PASSIVE_LEVEL	Nonarbitrary

DISPATCH ROUTINE	CALLER'S IRQL:	CALLER'S THREAD CONTEXT:
QueryVolumeInfo	PASSIVE_LEVEL	Nonarbitrary
Read (except paging I/O)	PASSIVE_LEVEL	Nonarbitrary
Read (paging I/O path)	APC_LEVEL	Arbitrary
SetEa	PASSIVE_LEVEL	Nonarbitrary
SetInformation	PASSIVE_LEVEL	Nonarbitrary
SetQuota	PASSIVE_LEVEL	Nonarbitrary
SetSecurity	PASSIVE_LEVEL	Nonarbitrary
SetVolumeInfo	PASSIVE_LEVEL	Nonarbitrary
Shutdown	PASSIVE_LEVEL	Arbitrary
Write (except paging I/O)	PASSIVE_LEVEL	Nonarbitrary
Write (paging I/O path)	APC_LEVEL	Arbitrary

Using IRP Completion Routines

4/26/2017 • 1 min to read • Edit Online

Note For optimal reliability and performance, we recommend using file system minifilter drivers instead of legacy file system filter drivers. Also, legacy file system filter drivers can't attach to direct access (DAX) volumes. For more about file system minifilter drivers, see Advantages of the Filter Manager Model. To port your legacy driver to a minifilter driver, see Guidelines for Porting Legacy Filter Drivers.

File system filter drivers use completion routines that are similar to those used by device drivers. A *completion routine* performs completion processing on an IRP. Any driver routine that passes an IRP down to the next-lower-level driver can optionally register a completion routine for the IRP by calling **IoSetCompletionRoutine** before calling **IoCallDriver**.

Every IRP completion routine is defined as follows:

```
NTSTATUS
(*PIO_COMPLETION_ROUTINE) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    );
```

Completion routines are called at IRQL <= DISPATCH_LEVEL, in an arbitrary thread context.

Because they can be called at IRQL DISPATCH_LEVEL, completion routines cannot call kernel-mode routines that must be called at a lower IRQL, such as **IoDeleteDevice**. For the same reason, any data structures that are used in a completion routine must be allocated from nonpaged pool.

This section discusses the following topics:

How Completion Processing Is Performed

Checking the PendingReturned Flag

Returning Status from Completion Routines

Example: Simple Pass-Through Dispatch and Completion

Constraints on Completion Routines

How Completion Processing Is Performed

4/26/2017 • 1 min to read • Edit Online

Completion processing is performed in two stages. The first stage is performed in an arbitrary thread context, at IRQL <= DISPATCH_LEVEL. In this stage, the following tasks are performed:

- Each completion routine registered for the IRP is called in turn, beginning with the lowest IRP stack location. If a completion routine returns STATUS_MORE_PROCESSING_REQUIRED, completion processing is halted.
- If the IRP contains a memory descriptor list (MDL), any physical pages mapped by the MDL are unlocked.
- The second phase of I/O completion is queued to the target (requesting) thread as a special kernel APC.

The second stage is performed in the context of the thread that originated the I/O request. It is executed as a special kernel APC and therefore runs at IRQL APC_LEVEL. In this stage, the following tasks are performed:

- If the IRP represents a buffered operation, the contents of **Irp->AssociatedIrp.SystemBuffer** are copied to **Irp->UserBuffer**.
- If the IRP contains an MDL, the MDL is freed.
- The contents of **Irp->IoStatus** are copied to **Irp->UserIosb** so that the originator of the I/O request can see the final status of the operation.
- If an event has been supplied in **Irp->UserEvent**, it is signaled. Otherwise, if there is a file object for this IRP, its event is signaled.
- If the IRP was created by calling **IoBuildDeviceIoControlRequest** or **IoBuildSynchronousFsdRequest**, it is dequeued from the thread's pending I/O request list.
- A user APC is queued, if the caller requested one.
- The IRP is freed.

Note If completion processing for an IRP is halted because a completion routine returned STATUS_MORE_PROCESSING_REQUIRED, it can be resumed by calling **IoCompleteRequest** on the same IRP. When this happens, first-stage processing resumes, beginning with the completion routine for the driver immediately above the one whose completion routine returned STATUS_MORE_PROCESSING_REQUIRED.

Checking the PendingReturned Flag

4/26/2017 • 1 min to read • Edit Online

If a completion routine does not signal an event, it must check the **Irp->PendingReturned** flag. If this flag is set, the completion routine must mark the IRP pending by calling **IoMarkIrpPending**.

Returning Status from Completion Routines

4/26/2017 • 1 min to read • Edit Online

A file system filter driver completion routine normally returns one of the following two NTSTATUS values to the caller:

- STATUS_SUCCESS
- STATUS_MORE_PROCESSING_REQUIRED

STATUS_SUCCESS indicates that the driver is finished with the IRP and allows the I/O Manager to continue completion processing on the IRP.

STATUS_MORE_PROCESSING_REQUIRED halts the I/O Manager's completion processing on the IRP.

If any other NTSTATUS value is returned, the I/O Manager resets it to STATUS_SUCCESS.

For more information about returning STATUS_MORE_PROCESSING_REQUIRED, see Constraints on Completion Routines.

Example: Simple Pass-Through Dispatch and Completion

4/26/2017 • 1 min to read • Edit Online

To set a completion routine for an IRP and pass the IRP down, a dispatch routine must do the following:

- Call **IoCopyCurrentIrpStackLocationToNext** to copy the parameters from the current stack location to that of the next-lower-level driver.
- Call **IoSetCompletionRoutine** to designate a completion routine for the IRP.
- Call **IoCallDriver** to pass the IRP down to the next-lower-level driver.

This technique is illustrated in the following code example:

Dispatch Routine

```
IoCopyCurrentIrpStackLocationToNext( Irp );
IoSetCompletionRoutine( Irp,                                  // Irp
                        MyLegacyFilterPassThroughCompletion, // CompletionRoutine
                        (PVOID)recordList,                   // Context
                        TRUE,                                // InvokeOnSuccess
                        TRUE,                                // InvokeOnError
                        TRUE);                               // InvokeOnCancel
return IoCallDriver ( NextLowerDriverDeviceObject, Irp );
```

In this example, the call to **IoSetCompletionRoutine** sets a completion routine for an IRP.

The first two parameters in the call to **IoSetCompletionRoutine** are a pointer to the IRP and the name of the completion routine. The third parameter is a pointer to a driver-defined structure to be passed to the completion routine. This structure contains context information that the completion routine will need when it performs completion processing on the IRP. The context structure must be allocated from nonpaged pool, because the completion routine can be called at IRQL DISPATCH_LEVEL.

The last three parameters passed to **IoSetCompletionRoutine** are flags that specify whether the completion routine is called when the I/O request succeeds, fails, or is canceled.

Completion Routine

If a dispatch routine sets a completion routine and immediately returns after calling **IoCallDriver** (as shown in the above dispatch routine), the corresponding completion routine must check the IRP's PendingReturned flag and, if it is set, call **IoMarkIrpPending**. Then it should return STATUS_SUCCESS, as shown in the following example:

```
if (Irp->PendingReturned) {
    IoMarkIrpPending( Irp );
}
return STATUS_SUCCESS;
```

Advantages of This Approach

Setting a completion routine allows the driver to further process the IRP after it has been processed by lower-level drivers. The completion routine can decide how to process the IRP based on the outcome of the requested I/O operation.

Disadvantages of This Approach

Because it runs in an arbitrary thread context at IRQL <= DISPATCH_LEVEL, a completion routine can perform only limited processing on the IRP.

Constraints on Completion Routines

4/26/2017 • 1 min to read • Edit Online

The following guidelines briefly discuss how to avoid common programming errors in file system filter driver completion routines.

IRQL-Related Constraints

Because completion routines can be called at IRQL DISPATCH_LEVEL, they are subject to the following constraints:

- Completion routines cannot safely call kernel-mode routines that require a lower IRQL, such as **IoDeleteDevice** or **ObQueryNameString**.
- Any data structures used in completion routines must be allocated from nonpaged pool.
- Completion routines cannot be made pageable.
- Completion routines cannot acquire resources, mutexes, or fast mutexes. However, they can acquire spin locks.

Checking the PendingReturned Flag

- Unless the completion routine signals an event, it must check the **Irp->PendingReturned** flag. If this flag is set, the completion routine must call **IoMarkIrpPending** to mark the IRP as pending.
- If a completion routine signals an event, it should not call **IoMarkIrpPending**.

Constraints on Returning Status

- A file system filter driver completion routine must return either STATUS_SUCCESS or STATUS_MORE_PROCESSING_REQUIRED. All other NTSTATUS values are reset to STATUS_SUCCESS by the I/O Manager.

Constraints on Returning STATUS_MORE_PROCESSING_REQUIRED

There are three cases when completion routines should return STATUS_MORE_PROCESSING_REQUIRED:

- When the corresponding dispatch routine is waiting for an event to be signaled by the completion routine. In this case, it is important to return STATUS_MORE_PROCESSING_REQUIRED to prevent the I/O Manager from completing the IRP prematurely after the completion routine returns.
- When the completion routine has posted the IRP to a worker queue and the corresponding dispatch routine has returned STATUS_PENDING. In this case as well, it is important to return STATUS_MORE_PROCESSING_REQUIRED to prevent the I/O Manager from completing the IRP prematurely after the completion routine returns.
- When the same driver created the IRP by calling **IoAllocateIrp** or **IoBuildAsynchronousFsdRequest**. Because the driver did not receive this IRP from a higher-level driver, it can safely free the IRP in the completion routine. After calling **IoFreeIrp**, the completion routine must return STATUS_MORE_PROCESSING_REQUIRED to indicate that no further completion processing is needed.

A completion routine cannot return STATUS_MORE_PROCESSING_REQUIRED for an oplock operation. Oplock operations cannot be pended (posted to a worker queue), and dispatch routines cannot return STATUS_PENDING for them.

Constraints on Posting IRPs to a Work Queue

- If a completion routine posts IRPs to a work queue, it must call **IoMarkIrpPending** before posting each IRP to the worker queue. Otherwise, the IRP could be dequeued, completed by another driver routine, and freed by the system before the call to **IoMarkIrpPending** occurs, thereby causing a crash.

Tracking Per-Stream Context in a Legacy File System Filter Driver

4/26/2017 • 1 min to read • Edit Online

Note For optimal reliability and performance, we recommend using file system minifilter drivers instead of legacy file system filter drivers. Also, legacy file system filter drivers can't attach to direct access (DAX) volumes. For more about file system minifilter drivers, see Advantages of the Filter Manager Model. To port your legacy driver to a minifilter driver, see Guidelines for Porting Legacy Filter Drivers.

This section covers per-stream context tracking in Microsoft Windows XP and later OS versions. The following topics are discussed:

File Streams, Stream Contexts, and Per-Stream Contexts

Creating and Using Per-Stream Context Structures

File Streams, Stream Contexts, and Per-Stream Contexts

4/26/2017 • 2 min to read • Edit Online

A *file stream* is a sequence of bytes used to hold file data. Usually a file has only one file stream, namely the file's default data stream. However, on file systems that support multiple data streams, each file can have multiple file streams. One of these is the default data stream, which is unnamed. The others are named alternate data streams. When you open a file, you are actually opening a stream of the given file.

When a file system opens a file stream for the first time, it creates a file-system-specific *stream context* structure, such as a file control block (FCB) or stream control block (SCB), and stores the address of this structure in the **FsContext** member of the resulting file object.

For local file systems, if the already opened file stream is opened again (for shared read access, for example), the I/O subsystem creates another file object, but the file system does not create a new stream context. Both file objects receive the address of the same stream context structure. Thus, for local file systems, the stream context pointer uniquely identifies a file stream.

For network file systems that support per-stream contexts, if the already opened file stream is opened again using the same network share name or IP address, the behavior is the same as for local file systems. The I/O subsystem creates a new file object, but the file system does not create a new stream context. Instead, it assigns the same **FsContext** pointer value to both file objects. However, if the file stream is opened using a different path (for example, a different share name, or an IP address for a file previously opened using a share name), the file system does create a new stream context. Thus, for network file systems that support per-stream contexts, the **FsContext** pointer does not uniquely identify a file stream.

A *per-stream context* is a filter-defined structure that contains a **FSRTL_PER_STREAM_CONTEXT** structure as one of its members. Filter drivers use this structure to track information about each file stream that is opened by the file system.

File System Support for Per-Stream Contexts

On Microsoft Windows XP and later, file systems that support per-stream contexts must use stream context structures that contain a **FSRTL_ADVANCED_FCB_HEADER** structure.

The global list of per-stream contexts associated with a particular file stream is owned by the file system. When the file system creates a new stream context (FSRTL_ADVANCED_FCB_HEADER object) for a file stream, it calls **FsRtlSetupAdvancedHeader** to initialize this list. When a file system filter driver calls **FsRtlInsertPerStreamContext**, the per-stream context created by the filter is added to the global list.

When the file system deletes its stream context for a file stream, it calls **FsRtlTeardownPerStreamContexts** to free all per-stream contexts that filters have associated with the file stream. This routine calls the **FreeCallback** routine for each per-stream context in the global list. Note that the **FreeCallback** routine must assume that the file object for the file stream has already been freed.

To query whether the file system supports per-stream contexts for the file stream represented by a given file object, call **FsRtlSupportsPerStreamContexts** on the file object. Note that a file system might support per-stream contexts for some types of files but not for others. For example, NTFS and FAT do not currently support per-stream contexts for paging files. Thus if **FsRtlSupportsPerStreamContexts** returns **TRUE** for one file stream, this does not imply that it returns **TRUE** for all file streams.

Creating and Using Per-Stream Context Structures

4/26/2017 • 1 min to read • Edit Online

File system filter drivers that use a per-stream context structure containing a **FSRTL_PER_STREAM_CONTEXT** structure can take advantage of built-in per-stream context support in Microsoft Windows XP and later. This section covers the following topics:

Creating a Per-Stream Context

4/26/2017 • 1 min to read • Edit Online

A file system filter driver typically creates a per-stream context structure for a file stream when the file stream is first opened. However, a per-stream context structure can be created and associated with a file stream during any operation.

Allocating the Per-Stream Context

Per-stream context structures can be allocated from paged or nonpaged pool. To allocate a per-stream context, call **ExAllocatePoolWithTag** as shown in the following example:

```
contextSize = sizeof(SPY_STREAM_CONTEXT) + fileName.Length;
ctx = ExAllocatePoolWithTag(NonPagedPool,
                             contextSize,
                             MYLEGACYFILTER_CONTEXT_TAG);
```

Note If your filter allocates the per-stream context structure from paged pool, it cannot call **ExAllocatePoolWithTag** from its create completion routine. This is because completion routines can be called at IRQL DISPATCH_LEVEL.

Initializing the Per-Stream Context

File system filter drivers call **FsRtlInitPerStreamContext** to initialize a per-stream context structure. This routine initializes the FSRTL_PER_STREAM_CONTEXT portion of the context structure. (The remainder of the structure is filter-driver-specific.)

Note If your filter driver creates only one per-stream context structure per file stream, it should pass **NULL** for the *InstanceId* parameter to **FsRtlInitPerStreamContext**.

A filter driver can initialize a per-stream context at any time. However, it must do so before associating the context with a file stream.

Associating a Per-Stream Context With a File Stream

4/26/2017 • 1 min to read • Edit Online

A per-stream context structure can be associated with a file stream only after the file system has successfully processed the **IRP_MJ_CREATE** request to open the stream. This is because it is only after the file system has processed the create request that the file object's FsContext pointer can be considered valid by a file system filter driver. Because the FsContext pointer uniquely identifies a file stream, it is needed to determine whether the file object represents a file stream that the filter has already seen -- and for which the filter has already created a per-stream context. For this reason, it is not unusual for a filter to create a per-stream context in the create dispatch (or "pre-Create") path, only to delete it in the create completion (or "post-Create") path because it turns out to be a duplicate.

To check whether it has already associated another per-stream context with the same file stream, a file system filter driver calls **FsRtlLookupPerStreamContext**.

If **FsRtlLookupPerStreamContext** finds an existing per-stream context for the same file stream, the filter should delete the newly created per-stream context.

If **FsRtlLookupPerStreamContext** does not find a per-stream context that your filter has already created previously for the file stream, the filter can call **FsRtlInsertPerStreamContext** to associate the newly created stream context with the file stream.

After **FsRtlInsertPerStreamContext** is called for a per-stream context, the file system assumes responsibility for deleting and freeing it. If your filter driver allocates a per-stream context and does not call **FsRtlInsertPerStreamContext** for it, your filter driver is still responsible for freeing it by calling **ExFreePool**.

Deleting a Per-Stream Context

4/26/2017 • 1 min to read • Edit Online

When a per-stream context that is associated with a file stream is no longer needed, it should be deleted. Stream contexts are deleted in one of two ways:

- Manually, when the filter driver calls **FsRtlRemovePerStreamContext**.
- Automatically, when the file system calls **FsRtlTeardownPerStreamContexts**, which in turn calls the stream context's **FreeCallback** routine.

When the Filter Deletes the Per-Stream Context

When a filter driver needs to delete its per-stream context for a file stream while the file stream is still open, it first calls **FsRtlRemovePerStreamContext** to remove the context from the global list of contexts associated with the given file. After calling **FsRtlRemovePerStreamContext**, the filter usually frees the context structure.

Note After your filter driver has called **FsRtlInsertPerStreamContext** to associate a per-stream context structure with a file stream, it must call **FsRtlRemovePerStreamContext** for the context before freeing it. Otherwise, the system will crash when the file stream is closed.

When the Per-Stream Context's FreeCallback Is Called

When the file stream is being closed or deleted, the file system frees its own stream context for the file stream. At this time, the file system also calls **FsRtlTeardownPerStreamContexts**, which in turn calls the **FreeCallback** routines registered for all of the per-stream contexts contained in the global list of contexts for that file stream. (A **FreeCallback** routine is registered when the filter driver calls **FsRtlInitPerStreamContext** to initialize a per-stream context structure. For more information, see **FSRTL_PER_STREAM_CONTEXT**.)

Note After your filter driver has called **FsRtlInsertPerStreamContext** to associate a per-stream context structure with a file stream, the file system is responsible for ensuring that the **FreeCallback** routine for the filter's per-stream context is called when there are no longer any open references to the stream.

Tracking Per-File Context in a Legacy File System Filter Driver

4/26/2017 • 1 min to read • Edit Online

Note For optimal reliability and performance, we recommend using file system minifilter drivers instead of legacy file system filter drivers. Also, legacy file system filter drivers can't attach to direct access (DAX) volumes. For more about file system minifilter drivers, see Advantages of the Filter Manager Model. To port your legacy driver to a minifilter driver, see Guidelines for Porting Legacy Filter Drivers.

A legacy file system filter driver can record context information for a file by associating a **FSRTL_PER_FILE_CONTEXT** object with a user-defined context information structure.

Note Not all file systems support per-file context objects. To find out whether a file is associated with a file system that supports them, use the **FsRtlSupportsPerFileContexts** macro.

Use the **FsRtlInitPerFileContext** macro to initialize the FSRTL_PER_FILE_CONTEXT object. Then use the **FsRtlInsertPerFileContext** routine to associate the file with an arbitrary context object.

Use the **FsRtlGetPerFileContextPointer** macro to get a pointer that is used by the file system runtime library (FSRTL) package to track file contexts.

A filter driver can use the **FsRtlLookupPerFileContext** routine to find a file context object that is associated with a file. The routine can specify the owner of a structure or an instance of a structure to narrow the search.

The filter driver can remove a context object by using **FsRtlRemovePerFileContext**. The routine can specify the owner of a structure or an instance of a structure to narrow the search.

Note Only use the **FsRtlRemovePerFileContext** routine to remove context objects while the file is still open. Do not confuse it with **FsRtlTeardownPerFileContexts**.

File systems call **FsRtlTeardownPerFileContexts** to free any filter contexts that are still associated with a per-file control block structure (FCB) that they are tearing down. The **FsRtlTeardownPerFileContexts** routine calls the **FreeCallback** routine that is specified in the FSRTL_PER_FILE_CONTEXT object for each filter context.

Blocking legacy file system filter drivers

4/26/2017 • 2 min to read • Edit Online

Starting in Windows 10, version 1607, administrators and driver developers can use a registry setting to block legacy file system filter drivers. *Legacy file system filter drivers* are drivers that attach to the file system stack directly and don't use Filter Manager. This topic describes the registry setting for blocking and unblocking legacy file system filter drivers. It also describes the event entered into the System event log when a legacy file system filter is blocked and how to check if the OS has legacy file system drivers running.

Note For optimal reliability and performance, we recommend using file system minifilter drivers instead of legacy file system filter drivers. Also, legacy file system filter drivers can't attach to direct access (DAX) volumes. For more about file system minifilter drivers, see Advantages of the Filter Manager Model. To port your legacy driver to a minifilter driver, see Guidelines for Porting Legacy Filter Drivers.

How to block legacy drivers

Use the **IoBlockLegacyFsFilters** registry key to specify if the system blocks legacy file system filter drivers. When blocked, all legacy file system filter drivers are blocked from loading. For the registry changes to take effect, perform a system restart.

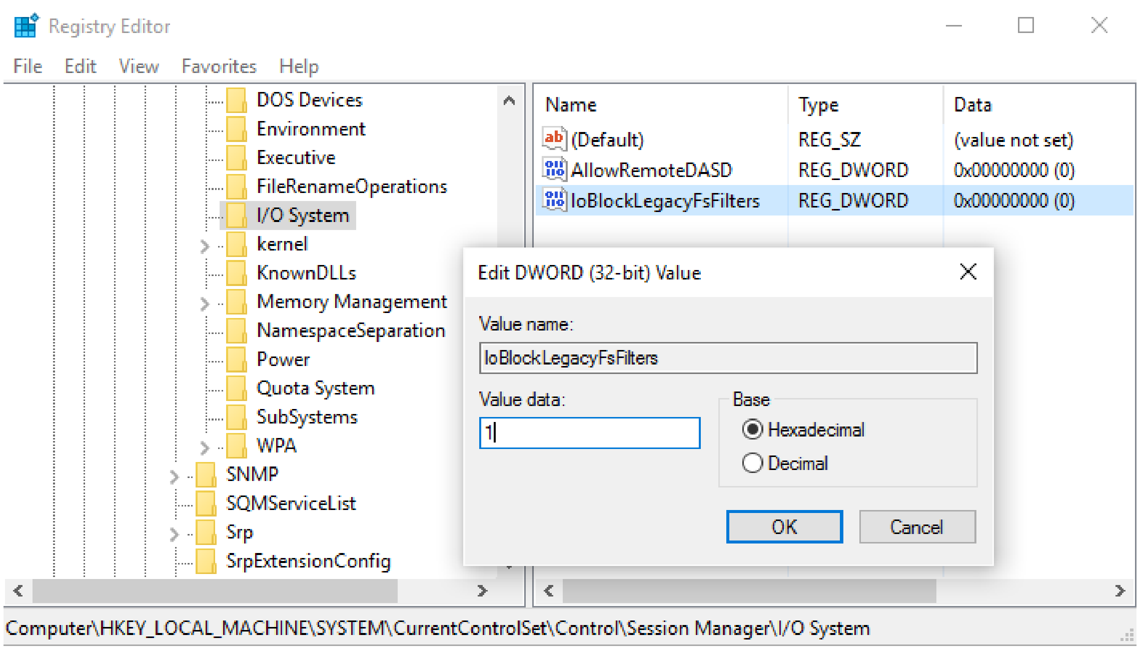
The registry key must be created under the following registry path:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SessionManager\I/O System
```

The valid DWORD values for the **IoBlockLegacyFsFilters** key are as follows:

IOBLOCKLEGACYFSFILTERS VALUE	DESCRIPTION
1	Legacy file system filter drivers are blocked from loading or attaching to storage volumes.
0	Legacy file system filter drivers are not blocked. In this release, this is the default behavior.

This is what the key looks like in Registry Editor:



Name	Type	Data
(Default)	REG_SZ	(value not set)
AllowRemoteDASD	REG_DWORD	0x00000000 (0)
IoBlockLegacyFsFilters	REG_DWORD	0x00000000 (0)

Value name: IoBlockLegacyFsFilters

Value data: 1

Base: [✓ Hexadecimal] Decimal

Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\I/O System

Example: when a legacy driver is blocked from loading

An **Error** event is logged to the System event log when a legacy file system filter driver is blocked from loading, as shown here:

EVENT PROPERTY	DESCRIPTION
Log Name	System
Source	Microsoft-Windows-Kernel-IO
Date	12/29/2015 2:55:05 PM
Event ID	1205
Task Category	None
Level	Error
Keywords	
User	CONTOSO\user
Computer	user.domain.corp.contoso.com
Description	Windows is configured to block legacy file system filters. Filter name: \Driver\sfilter

How to check if legacy drivers are running

If you're unsure which filters are legacy file system filter drivers or want to make sure that they're not running, you can perform the following:

To check if legacy file system filter drivers are running

1. Open an elevated Command Prompt by right-clicking a **cmd.exe** icon and clicking **Run as administrator**.

2. Type: `fltmc filters`
3. Look for legacy drivers, they're the ones with a **Frame** value of **<Legacy>**.

In this example, the legacy file system filter drivers, named AVLegacy and EncryptionLegacy, are marked with the **<Legacy>** Frame value. The file system driver named AVMiniFilter does not have the **<Legacy>** Frame value because it is a minifilter driver (it does not attach to the file system stack directly and uses Filter Manager).

```
C:\Windows\system32>fltmc filters

Filter Name                     Num Instances    Altitude    Frame
------------------------------  -------------  ------------  -----
AVLegacy                                        389998.99   <Legacy>
EncryptionLegacy                                149998.99   <Legacy>
AVMiniFilter                            3       328000         0
```

If you see that legacy drivers are still running after you block legacy file system filter drivers, make sure you reboot the system after setting the **IoBlockLegacyFsFilters** registry key. The setting will not take effect until after a reboot.

If your system has legacy file system filter drivers, work with the respective ISVs to get the Minifilter version of the file system driver. For info about porting legacy file system filter drivers to minifilter drivers that use the Filter Manager model, see Guidelines for Porting Legacy Filter Drivers.

File System Minifilter Drivers

4/26/2017 • 1 min to read • Edit Online

This section includes the following topics, which describe file system minifilter drivers:

Filter Manager and Minifilter Driver Architecture

4/26/2017 • 1 min to read • Edit Online

The filter manager is a kernel-mode driver that conforms to the legacy file system filter model and exposes functionality commonly required in file system filter drivers. By taking advantage of this functionality, third-party developers can write minifilter drivers, which are simpler to develop than legacy file system filter drivers, thus shortening the development process while producing higher-quality, more robust drivers.

This section includes:

Filter Manager Concepts

4/26/2017 • 3 min to read • Edit Online

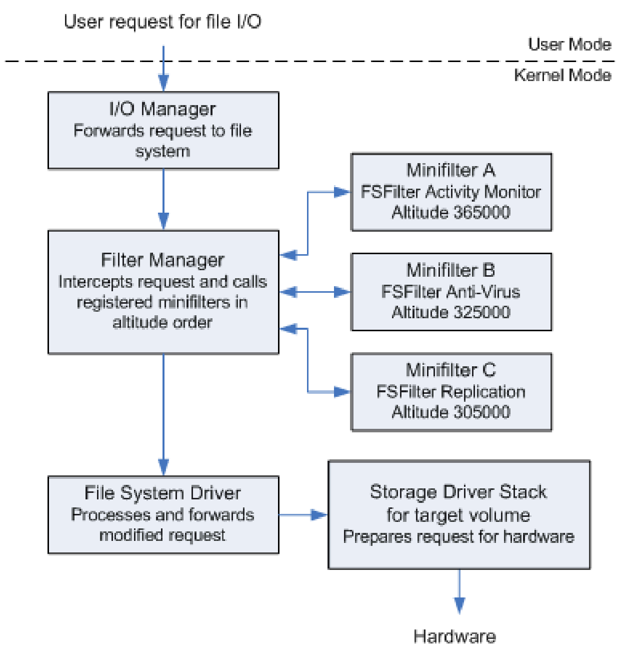
The filter manager is installed with Windows, but it becomes active only when a minifilter driver is loaded. The filter manager attaches to the file system stack for a target volume. A minifilter driver attaches to the file system stack indirectly, by registering with the filter manager for the I/O operations the minifilter driver chooses to filter.

A legacy filter driver's position in the file system I/O stack relative to other filter drivers is determined at system startup by its load order group. For example, an antivirus filter driver should be higher in the stack than a replication filter driver, so it can detect viruses and disinfect files before they are replicated to remote servers. Therefore, filter drivers in the FSFilter Anti-Virus load order group are loaded before filter drivers in the FSFilter Replication group. Each load order group has a corresponding system-defined class and class GUID used in the INF file for the filter driver.

Like legacy filter drivers, minifilter drivers attach in a particular order. However, the order of attachment is determined by a unique identifier called an *altitude*. The attachment of a minifilter driver at a particular altitude on a particular volume is called an *instance* of the minifilter driver.

A minifilter driver's altitude ensures that the instance of the minifilter driver is always loaded at the appropriate location relative to other minifilter driver instances, and it determines the order in which the filter manager calls the minifilter driver to handle I/O. Altitudes are allocated and managed by Microsoft.

The following figure shows a simplified I/O stack with the filter manager and three minifilter drivers.



A minifilter driver can filter IRP-based I/O operations as well as fast I/O and file system filter (FSFilter) callback operations. For each of the I/O operations it chooses to filter, a minifilter driver can register a preoperation callback routine, a postoperation callback routine, or both. When handling an I/O operation, the filter manager calls the appropriate callback routine for each minifilter driver that registered for that operation. When that callback routine returns, the filter manager calls the appropriate callback routine for the next minifilter driver that registered for the operation.

For example, assuming all three minifilter drivers in the above figure registered for the same I/O operation, the filter manager would call their preoperation callback routines in order of altitude from highest to lowest (A, B, C), then forward the I/O request to the next-lower driver for further processing. When the filter manager receives the

I/O request for completion, it calls each minifilter driver's postoperation callback routines in reverse order, from lowest to highest (C, B, A).

For interoperability with legacy filter drivers, the filter manager can attach filter device objects to a file system I/O stack in more than one location. Each of the filter manager's filter device objects is called a *frame*. From the perspective of a legacy filter driver, each filter manager frame is just another legacy filter driver.

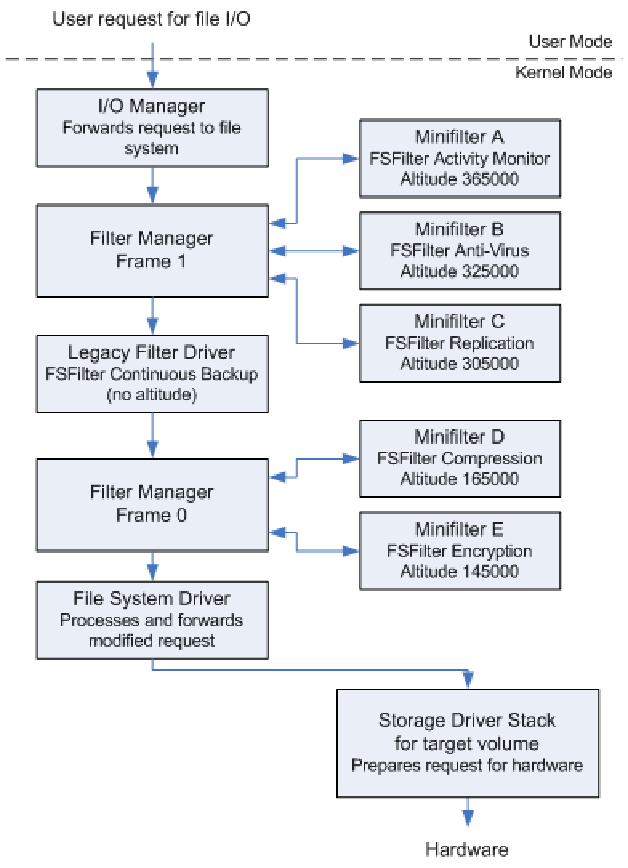
Each filter manager frame represents a range of altitudes. The filter manager can adjust an existing frame or create a new frame to allow minifilter drivers to attach at the correct location.

The filter manager cannot attach a minifilter between two attached legacy filters unless there is already a filter manager frame between them. If a minifilter is intended to be attached above a legacy filter, it can be attached below it, depending on the existence of a second attached legacy filter. A minifilter intended to be attached below a legacy filter could, instead, be attached above that legacy filter.

Important Always verify interoperability of legacy filters with minifilters or consider replacing legacy filters with minifilters. For more information, see Guidelines for Porting Legacy Filter Drivers.

If a minifilter driver is unloaded and reloaded, it is reloaded at the same altitude in the same frame from which it was unloaded.

The following figure shows a simplified I/O stack with a two filter manager frames, minifilter driver instances, and a legacy filter driver.



Advantages of the Filter Manager Model

4/26/2017 • 2 min to read • Edit Online

The filter manager model offers the following advantages over the existing legacy filter driver model:

- **Better control over filter load order.** Unlike a legacy filter driver, a minifilter driver can be loaded at any time and attached at the appropriate location as determined by its altitude.
- **Ability to unload while system is running.** Unlike a legacy filter driver, which cannot be unloaded while the system is running, a minifilter driver can be unloaded at any time, and it can prevent itself from being unloaded if necessary. The filter manager synchronizes safe removal of all minifilter driver attachments, and it handles operations that complete after the minifilter driver is unloaded.
- **Ability to process only necessary operations.** The filter manager uses a callback model in which a minifilter driver can choose which types of I/O operations (IRP-based, fast I/O, or FSFilter) to filter. The minifilter driver receives only I/O requests for which it has registered callback routines. A minifilter driver can register a unique preoperation or postoperation callback routine, or both, and it can ignore certain types of operations, such as paging I/O and cached I/O.
- **More efficient use of kernel stack.** The filter manager is optimized to reduce the amount of kernel stack it uses, and the callback model greatly reduces the impact of minifilter drivers on the stack. The filter manager reduces the impact of recursive I/O by supporting filter-initiated I/O that is seen only by lower drivers in the stack.
- **Less redundant code.** The filter manager reduces the amount of code required for a minifilter driver in a number of ways, such as providing infrastructure for name generation and caching file names for use by more than one minifilter driver. The filter manager attaches to volumes and notifies minifilter drivers when a volume is available. The filter manager is optimized to support multiprocessor systems, which makes locking both more efficient and less prone to error.
- **Reduced complexity.** The filter manager simplifies filtering I/O requests by providing support routines for common functionality, such as naming, context management, communication between user mode and kernel mode, and masking differences between file systems. The filter manager handles certain tasks on behalf of minifilter drivers, such as pending IRPs and enumerating and attaching to file system stacks.
- **Easier addition of new operations.** Because minifilter drivers register only for the I/O operations they will handle, support for new operations can be added to the filter manager without breaking existing minifilter drivers.
- **Better support for multiple platforms.** A minifilter driver can run on any version of Windows that supports the filter manager. If a minifilter driver registers for an I/O operation that isn't available at runtime, the filter manager simply doesn't call the minifilter driver for that operation. A minifilter driver can determine programmatically whether functions are available, and filter manager structures are designed to be extensible.
- **Better support for user-mode applications.** The filter manager provides common functionality for user-mode services and control programs that work with minifilter drivers. The filter manager user-mode library, Filterlib.dll, enables communication between a user-mode service or control program and a minifilter driver. Filterlib.dll also provides interfaces for management tools.

Filter Manager Support for Minifilter Drivers

4/26/2017 • 1 min to read • Edit Online

This section describes filter manager support for common tasks performed by minifilter drivers.

This section contains the following topics:

Loading and Unloading

4/26/2017 • 3 min to read • Edit Online

A minifilter driver can be loaded at any time while the system is running. If a minifilter driver's INF file specifies a driver start type of SERVICE_BOOT_START, SERVICE_SYSTEM_START, or SERVICE_AUTO_START, the minifilter driver is loaded according to existing load order group definitions for file system filter drivers, to support interoperability with legacy filter drivers. While the system is running, a minifilter driver can be loaded through a service start request (sc start, net start, or the service APIs), or through an explicit load request (fltmc load, **FltLoadFilter**, or **FilterLoad**).

A minifilter driver's DriverEntry routine is called when the minifilter driver is loaded, so the minifilter driver can perform initialization that will apply to all instances of the minifilter driver. Within its **DriverEntry** routine, the minifilter driver calls **FltRegisterFilter** to register callback routines with the filter manager and **FltStartFiltering** to notify the filter manager that the minifilter driver is ready to start attaching to volumes and filtering I/O requests.

Minifilter driver instances are defined in the INF file used to install the minifilter driver. A minifilter driver's INF file must define a default instance, and it can define additional instances. These definitions apply across all volumes. Each instance definition includes the instance name, its altitude, and flags that indicate whether the instance can be attached automatically, manually, or both. The default instance is used to order minifilter drivers so that the filter manager calls the minifilter driver's mount and instance setup callback routines in the correct order. The default instance is also used with explicit attachment requests when the caller doesn't specify an instance name.

The filter manager automatically notifies a minifilter driver about an available volume by calling its *InstanceSetupCallback* routine on the first create operation after the volume is mounted. This can occur before **FltStartFiltering** returns, when the filter manager enumerates existing volumes at system startup. It can also occur during runtime, when a volume is mounted or as a result of an explicit attachment request (fltmc attach, **FltAttachVolume**, or **FilterAttach**).

A minifilter driver instance is torn down when the minifilter driver is unloaded, the volume the instance is attached to is being dismounted, or as a result of an explicit detach request (fltmc detach, **FltDetachVolume**, or **FilterDetach**). If the minifilter driver registers an *InstanceQueryTeardownCallback* routine, it can fail an explicit detach request by calling **FilterDetach** or **FltDetachVolume**. The teardown proceeds as follows:

- If the minifilter driver registered an *InstanceTeardownStartCallback* callback routine, the filter manager calls it at the beginning of the teardown process. In this routine, the minifilter driver should complete all pending operations, cancel or complete other work such as I/O requests generated by the minifilter driver, and stop queuing new work items.

- During instance teardown, any currently executing preoperation or postoperation callback routines continue normal processing, any I/O request that is waiting for a postoperation callback can be "drained" or canceled, and any I/O requests generated by the minifilter driver continue normal processing until they are complete.

- If the minifilter driver registered an *InstanceTeardownCompleteCallback* routine, the filter manager calls this routine after all outstanding I/O operations have been completed. In this routine, the minifilter driver closes any files that are still open.

- After all outstanding references to the instance are released, the filter manager deletes remaining contexts and the instance is completely torn down.

While the system is running, a minifilter driver can be unloaded through a service stop request (sc stop, net stop, or the service APIs), or through an explicit unload request (fltmc unload, **FltUnloadFilter**, or **FilterUnload**).

A minifilter driver's *FilterUnloadCallback* routine is called when the minifilter driver is unloaded. This routine closes any open communication server ports, calls **FltUnregisterFilter**, and performs any needed cleanup. Registering this routine is optional. However, if the minifilter driver does not register a *FilterUnloadCallback* routine, the minifilter driver cannot be unloaded. For more information about this routine, see Writing a FilterUnloadCallback Routine.

Filter Manager Routines for Loading and Unloading Minifilter Drivers

The filter manager provides the following support routines for explicit load and unload requests, which can be issued from user mode or kernel mode:

FilterLoad

FilterUnload

FltLoadFilter

FltUnloadFilter

The following routines are used to register and unregister callback routines for instance setup and teardown:

FltRegisterFilter

FltStartFiltering

FltUnregisterFilter

Minifilter Driver Callback Routines for Instance Setup, Teardown, and Unload

The following minifilter driver callback routines are stored in the **FLT_REGISTRATION** structure that is passed as a parameter to **FltRegisterFilter**:

CALLBACK ROUTINE NAME	CALLBACK ROUTINE TYPE
InstanceSetupCallback	**PFLT_INSTANCE_SETUP_CALLBACK**
InstanceQueryTeardownCallback	**PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK**
InstanceTeardownStartCallback	**PFLT_INSTANCE_TEARDOWN_CALLBACK**
InstanceTeardownCompleteCallback	**PFLT_INSTANCE_TEARDOWN_CALLBACK**
FilterUnloadCallback	**PFLT_FILTER_UNLOAD_CALLBACK**

Processing I/O Operations

4/26/2017 • 3 min to read • Edit Online

The filter manager simplifies processing I/O operations for minifilter drivers. Unlike a legacy filter driver, which must correctly pass all I/O requests to the next-lower driver and correctly handle pending requests, synchronization, and I/O completion whether the legacy filter driver does any work related to the request, a minifilter driver registers only for the I/O operations it must handle.

For a given I/O operation, the filter manager calls only minifilter drivers that have registered a **preoperation callback** routine for that operation. The filter manager also handles certain IRP maintenance tasks on behalf of the minifilter driver, such as copying parameters to the next stack location and propagating the IRP **PendingReturned** flag.

In its preoperation callback routine, a minifilter driver does whatever processing is needed for the I/O operation and indicates what should be done to the IRP by returning the appropriate value from its preoperation callback routine. For example, to forward an IRP to the next-lower driver without a completion routine, the minifilter driver returns FLT_PREOP_SUCCESS_NO_CALLBACK; to do the same with a completion routine (the minifilter driver's postoperation callback routine for the I/O operation), the minifilter driver returns FLT_PREOP_SUCCESS_WITH_CALLBACK.

In its preoperation callback routine, the minifilter driver can queue the operation to a worker thread if needed by calling **FltQueueDeferredIoWorkItem**. After doing so, the minifilter driver returns FLT_PREOP_PENDING from its preoperation callback routine to indicate that the I/O operation is pending, and the minifilter driver is responsible for completing or resuming processing of the request. To resume processing, the minifilter driver calls **FltCompletePendedPreOperation** from the worker thread.

If the minifilter driver needs to maintain its own per-instance cancel-safe queue of outstanding I/O operations to be processed, it can set up such a queue by calling *FltCbdqInitialize* in its *InstanceSetupCallback* routine and calling *FltCbdqInsertIo* in its preoperation callback routine as needed to insert I/O operations into the queue.

The filter manager calls a minifilter driver's **postoperation callback** routine for an I/O operation when lower filter drivers (legacy filters and minifilter drivers) have finished completion processing.

In its postoperation callback routine, the minifilter driver can call **FltDoCompletionProcessingWhenSafe** to ensure that completion processing is performed at safe IRQL. Or it can queue the completion processing of the operation to a worker thread if needed by calling **FltQueueDeferredIoWorkItem**. After doing so, the minifilter driver returns FLT_POSTOP_MORE_PROCESSING_REQUIRED from its postoperation callback routine to halt the filter manager's completion processing for the I/O operation. To resume completion processing, the minifilter driver calls **FltCompletePendedPostOperation** from the worker thread.

The filter manager provides support for queuing of "generic" work items - work items that are associated with a minifilter driver or minifilter driver instance rather than an I/O operation. A minifilter driver can insert a work item into a system work queue by calling **FltQueueGenericWorkItem**. This routine is similar to routines such as **ExQueueWorkItem**; for example, work items (allocated by calling **FltAllocateGenericWorkItem**) can be reused. However, **FltQueueGenericWorkItem** is safer for minifilter drivers to use, because the filter manager does not allow the minifilter driver or minifilter driver instance to unload while outstanding work items are still being processed.

The filter manager also provides support for opportunistic lock (oplock) operations. For oplock operations, a minifilter driver can use such filter manager routines as **FltInitializeOplock** and **FltOplockFsctrl**, which are equivalent to the **FsRtlInitializeOplock** and **FsRtlOplockFsctrl** routines that are used by file systems and legacy filter drivers.

Filter Manager Routines for Processing I/O Operations

The filter manager provides the following support routines for pending I/O in **preoperation** and **postoperation** callback routines:

FltCompletePendedPostOperation

FltCompletePendedPreOperation

FltDoCompletionProcessingWhenSafe

The following routines are used for queuing work items in preoperation and postoperation callback routines:

FltAllocateDeferredIoWorkItem

FltAllocateGenericWorkItem

FltFreeDeferredIoWorkItem

FltFreeGenericWorkItem

FltQueueDeferredIoWorkItem

FltQueueGenericWorkItem

The following routines provide cancel-safe queue support:

FltCbdqDisable

FltCbdqEnable

FltCbdqInitialize

FltCbdqInsertIo

FltCbdqRemoveIo

FltCbdqRemoveNextIo

The following routines provide oplock support:

FltCheckOplock

FltCurrentBatchOplock

FltInitializeOplock

FltOplockFsctrl

FltOplockIsFastIoPossible

FltUninitializeOplock

Minifilter Driver Callback Routines for Processing I/O Operations

The following callback routines are stored in the **FLT_OPERATION_REGISTRATION** structure for each type of I/O operation that the minifilter driver handles:

CALLBACK ROUTINE NAME	CALLBACK ROUTINE TYPE
PostOperation	**PFLT_POST_OPERATION_CALLBACK**
PreOperation	**PFLT_PRE_OPERATION_CALLBACK**

Modifying Parameters

4/26/2017 • 2 min to read • Edit Online

A minifilter driver can modify certain parameters associated with an I/O operation, such as the target instance, target file object, and operation-specific parameters including buffer address and memory descriptor list (MDL) address. The minifilter driver usually modifies parameters in its preoperation callback for the I/O request. If the minifilter driver modifies parameters, it must call **FltSetCallbackDataDirty** to notify the filter manager that the parameters have changed. It should also record changes in the context passed from its preoperation callback so they are available to its postoperation callback.

A minifilter driver can change the I/O status for an operation when completing the operation in its preoperation callback or failing the operation in its postoperation callback (such as changing a STATUS_SUCCESS to an error status). It is not necessary to call **FltSetCallbackDataDirty** in this case.

For more information about modifying parameters, see Modifying the Parameters for an I/O Operation.

A minifilter driver can "swap buffers" by replacing the buffer field of an I/O request with its own buffer. Such a minifilter driver is responsible for keeping the MDL and buffer fields of the I/O request in sync. The filter manager sets the FLTFL_CALLBACK_DATA_SYSTEM_BUFFER_FLAG in the **FLT_CALLBACK_DATA** structure to indicate whether a buffer is a system buffer; if so, the minifilter driver must allocate the replacement buffer from nonpaged pool and set the MDL field to **NULL**. Otherwise, the buffer can be allocated from either paged or nonpaged pool, and the minifilter driver must always create and set an MDL. (In the case of a fast I/O operation, the new buffer can be allocated from either paged or nonpaged pool and the MDL should be **NULL**.) The minifilter driver must not free the buffer or MDL it is replacing, and it must not free any MDL it has successfully inserted into a callback data structure (the filter manager will free the MDL on behalf of the minifilter driver). After making a change to an MDL or buffer, the minifilter driver must call **FltSetCallbackDataDirty**.

A minifilter driver must register a postoperation callback for any operation in which it swaps buffers. In this callback routine, the minifilter driver must free any buffers it allocated. The filter manager will free the MDL unless the minifilter driver calls **FltRetainSwappedBufferMdlAddress**; in this case, the minifilter driver is responsible for freeing the MDL. The minifilter driver can call **FltGetSwappedBufferMdlAddress** to get the MDL for the buffer set in its preoperation callback.

If the minifilter driver is unloaded during an operation in which it has swapped buffers, the operation cannot be "drained"; instead, the operation is canceled and the filter manager waits for the operation to complete before unloading the minifilter driver.

See the SwapBuffers sample for an example of a minifilter driver that swaps buffers.

Filter Manager Routines for Modifying Parameters

The filter manager provides the following support routines for modifying I/O operation parameters in preoperation and postoperation callback routines:

FltClearCallbackDataDirty

FltIsCallbackDataDirty

FltSetCallbackDataDirty

The following routines provide support for swapping buffers:

FltGetSwappedBufferMdlAddress

FltRetainSwappedBufferMdlAddress

Accessing User Buffers

4/26/2017 • 1 min to read • Edit Online

All parameters specific to a given I/O operation, including buffers and memory descriptor lists (MDLs), are defined in an **FLT_PARAMETERS** union. This union is contained in an **FLT_IO_PARAMETER_BLOCK** structure that is accessed through the **Iopb** member of the **FLT_CALLBACK_DATA** structure that represents the I/O operation. Both the filter manager and minifilter drivers use **FLT_CALLBACK_DATA** structures to initiate and process I/O operations.

The **FLT_PARAMETERS** union also contains any parameter definitions for IRP-based operations that are specific to the buffering method used for that operation (buffered, direct I/O, or neither buffered nor direct I/O). It also contains parameter definitions for non-IRP-based operations (fast I/O and FsFilter callback routines).

A minifilter driver can call **FltDecodeParameters** to get pointers to the MDL address, buffer pointer, buffer length, and desired access parameters for an I/O operation. This saves minifilter drivers from having a switch statement to find the position of these parameters in helper routines that access these parameters across multiple I/O operations.

When processing an I/O operation that involves user buffers, a minifilter driver should always use an MDL if one is available. If so, the minifilter driver should call **MmGetSystemAddressForMdlSafe** to get a system address for the MDL and use the system address to access the user buffer.

If only a buffer address is available, the minifilter driver should always enclose any attempts to access the buffer in a try/except block. If the minifilter driver needs to access the buffer in a postoperation callback routine that is not synchronized, or if the I/O operation is posted to a worker thread, the minifilter driver should also lock the user buffer by calling **FltLockUserBuffer**. This function determines the appropriate access method to apply for the locked buffer based on the type of I/O operation and creates an MDL that points to the locked pages.

Filter Manager Routines for Accessing User Buffers

The filter manager provides the following support routines for accessing user buffers in preoperation and postoperation callback routines:

FltDecodeParameters

FltLockUserBuffer

Managing File Names

4/26/2017 • 1 min to read • Edit Online

The filter manager eliminates much of the work required for legacy filter drivers to retrieve and manage file names. When a name is requested, the filter manager provides the name in a reference-counted structure in the appropriate format for the current operation: normalized name, opened name, or short name.

A minifilter driver can call **FltGetDestinationFileNameInformation** to construct a full destination path name for a file or directory that is being renamed or for which an NTFS hard link is being created. This name can be returned in either normalized or opened-file format.

The filter manager also provides the **FltGetTunneledName** routine for retrieving normalized file name information that is invalidated due to file name tunneling.

To improve performance, the filter manager places names in a cache (where possible) that is shared among all minifilter drivers in the system. A minifilter driver can query either the cache or the file system, or both.

Minifilter drivers that modify the namespace can take advantage of the filter manager's support for name providers by registering callback routines to intercept name query operations, such as requests by upper minifilter drivers to generate or normalize a name.

Filter Manager Routines for Name Management

The filter manager provides the following support routines for name management:

FltGetDestinationFileNameInformation

FltGetFileNameInformation

FltGetFileNameInformationUnsafe

FltGetTunneledName

FltParseFileNameInformation

FltReleaseFileNameInformation

Minifilter Driver Callback Routines for Name Management

The following callback routines are stored in the **FLT_REGISTRATION** structure that is passed as a parameter to **FltRegisterFilter**, for minifilter drivers that modify the namespace:

CALLBACK ROUTINE NAME	CALLBACK ROUTINE TYPE
GenerateFileNameCallback	**PFLT_GENERATE_FILE_NAME**
NormalizeContextCleanupCallback	**PFLT_NORMALIZE_CONTEXT_CLEANUP**
NormalizeNameComponentCallback	**PFLT_NORMALIZE_NAME_COMPONENT**

Managing Contexts

4/26/2017 • 1 min to read • Edit Online

The filter manager enables minifilter drivers to associate contexts with objects to preserve state across I/O operations. Objects that can have contexts include volumes, instances, streams, and stream handles.

Third-party file systems must use the **FSRTL_ADVANCED_FCB_HEADER** structure (instead of the **FSRTL_COMMON_FCB_HEADER** structure) to work properly with stream and stream handle contexts.

Contexts can be allocated from paged or nonpaged pool except for volume contexts, which must be allocated from nonpaged pool.

Contexts are freed automatically when all outstanding references have been released. If the minifilter driver defines a context cleanup callback routine, the filter manager calls the routine before the context is freed.

The filter manager takes care of deleting contexts when the associated object is deleted, when an instance is detached, and when the minifilter driver is unloaded.

If a minifilter driver supports only one instance per volume, use instance contexts rather than volume contexts for better performance. You can also improve performance by storing a pointer to the minifilter driver instance context inside stream or stream handle contexts.

Contexts are not supported for paging files or during the following operations:

- Preoperation processing for create requests
- Postoperation processing for close requests
- Processing of IRP_MJ_NETWORK_QUERY_OPEN requests

See the CTX sample for an example of a minifilter driver that uses contexts.

Filter Manager Routines for Context Management

The filter manager provides the following support routines for creating, registering, and setting contexts:

FltAllocateContext

FltRegisterFilter

FltSetInstanceContext

FltSetStreamContext

FltSetStreamHandleContext

FltSetVolumeContext

The following routines are provided for querying context support:

FltSupportsStreamContexts

FltSupportsStreamHandleContexts

The following routines are provided for getting and referencing contexts:

FltGetContexts

FltGetInstanceContext

FltGetStreamContext

FltGetStreamHandleContext

FltGetVolumeContext

FltReferenceContext

The following routines are provided for releasing and deleting contexts:

FltDeleteContext

FltDeleteInstanceContext

FltDeleteStreamContext

FltDeleteStreamHandleContext

FltDeleteVolumeContext

FltReleaseContext

FltReleaseContexts

Minifilter Driver Callback Routines for Context Management

The following callback routines are stored in the **FLT_REGISTRATION** structure that is passed as a parameter to **FltRegisterFilter** for minifilter drivers that manage contexts:

CALLBACK ROUTINE NAME	CALLBACK ROUTINE TYPE
ContextAllocateCallback	**PFLT_CONTEXT_ALLOCATE_CALLBACK**
ContextCleanupCallback	**PFLT_CONTEXT_CLEANUP_CALLBACK**
ContextFreeCallback	**PFLT_CONTEXT_FREE_CALLBACK**

I/O Requests Generated by the Minifilter Driver

4/26/2017 • 1 min to read • Edit Online

A minifilter driver can generate and send IRP-based I/O requests from one of the minifilter driver's own instances on the current volume or on another volume. The generated I/O is seen only by the minifilter driver instances and legacy filter drivers attached below the specified instance, and by the file system. This solves many problems related to recursive I/O in the legacy filter driver model, in which I/O requests generated by a legacy filter driver must travel through the entire file system stack, starting with the topmost driver.

The filter manager doesn't unload a minifilter driver until all of its outstanding I/O operations are completed.

Filter Manager Routines for I/O Requests Generated by the Minifilter Driver

The filter manager provides the following support routines for creating, opening, reading, and writing files:

FltClose

FltCreateFile

FltCreateFileEx

FltReadFile

FltWriteFile

The following support routines are provided for setting and removing reparse points:

FltTagFile

FltUntagFile

The following support routines are provided for generating I/O requests:

FltAllocateCallbackData

FltFreeCallbackData

FltPerformAsynchronousIo

FltPerformSynchronousIo

FltReuseCallbackData

The following support routines are provided for canceling a file open request and for reissuing an I/O request:

FltCancelFileOpen

FltReissueSynchronousIo

The filter manager also provides the following general-purpose routines:

FltDeviceIoControlFile

FltFlushBuffers

FltFsControlFile

FltQueryInformationFile

FltQuerySecurityObject

FltQueryVolumeInformationFile

FltSetInformationFile

FltSetSecurityObject

Communication Between User Mode and Kernel Mode

4/26/2017 • 1 min to read • Edit Online

The filter manager supports communication between user mode and kernel mode through communication ports. The minifilter driver controls security on the port by specifying a security descriptor to be applied to the communication port object. Communication through a communication port is not buffered, so it is faster and more efficient. A user-mode application or service can reply to messages from a minifilter driver for bidirectional communication.

When the minifilter driver creates a communication server port, it implicitly begins to listen for incoming connections on the port. When a user-mode caller attempts to connect to the port, the filter manager calls the minifilter driver's **ConnectNotifyCallback** routine with a handle to the newly created connection. When the filter manager regains control, it passes the user-mode caller a separate file handle that represents the user-mode caller's endpoint to the connection. This handle can be used to associate I/O completion ports with the listener port.

A connection is accepted only if the user-mode caller has sufficient access as specified by the security descriptor on the port. Each connection to the port gets its own message queue and private endpoints.

Closing either endpoint (kernel or user) terminates that connection. When a user-mode caller closes its handle to the endpoint, the filter manager calls the minifilter driver's **DisconnectNotifyCallback** routine so the minifilter driver can close its handle to the connection.

Closing the communication server port prevents new connections but does not terminate existing connections. The filter manager terminates existing connections when the minifilter driver unloads.

Filter Manager Routines for Communication Between User Mode and Kernel Mode

The filter manager provides the following support routines for kernel-mode minifilter drivers to communicate with user-mode applications:

FltCloseClientPort

FltCloseCommunicationPort

FltCreateCommunicationPort

FltSendMessage

The following support routines are provided for user-mode applications to communicate with minifilter drivers:

FilterConnectCommunicationPort

FilterGetMessage

FilterReplyMessage

FilterSendMessage

Minifilter Driver Callback Routines for Communication Between User Mode and Kernel Mode

The following minifilter driver callback routines are passed as parameters to **FltCreateCommunicationPort**:

CALLBACK ROUTINE NAME	CALLBACK ROUTINE TYPE
ConnectNotifyCallback	PFLT_CONNECT_NOTIFY
DisconnectNotifyCallback	PFLT_DISCONNECT_NOTIFY
MessageNotifyCallback	PFLT_MESSAGE_NOTIFY

User-Mode Library

4/26/2017 • 1 min to read • Edit Online

The filter manager user mode interfaces provide common functionality for products that include filter drivers. The user-mode library is *Fltlib.dll*. Applications include the header files *FltUser.h* and *FltUserStructures.h*, and link to *FltLib.lib*.

These user-mode interfaces enable general control of the minifilter driver and communication between the user-mode service or control program and the filter driver. User-mode interfaces also provide interfaces for management tools that allow enumeration of filters, volumes, and instances.

For minifilters, user-mode communication APIs do not require administrator privileges. Instead, a minifilter defines the necessary privilege using an **ACL** defined on a port.

Filter Manager User-Mode Library Routines

The filter manager provides the following support routines for user-mode applications to use for loading and unloading minifilter drivers:

FilterLoad

FilterUnload

The following support routines are provided for creating and closing minifilter driver and instance handles:

FilterClose

FilterCreate

FilterInstanceClose

FilterInstanceCreate

The following support routines are provided for attaching and detaching minifilter driver instances:

FilterAttach

FilterAttachAtAltitude

FilterDetach

The following support routines are provided for enumerating filters, volumes, and instances:

FilterFindFirst

FilterFindNext

FilterInstanceFindFirst

FilterInstanceFindNext

FilterVolumeFindFirst

FilterVolumeFindNext

FilterVolumeInstanceFindFirst

FilterVolumeInstanceFindNext

The following support routines are provided for querying for information:

FilterGetDosName

FilterGetInformation

FilterInstanceGetInformation

The following support routines are provided for communication initiated by a user operation:

FilterConnectCommunicationPort

FilterSendMessage

The following support routines are provided for responding to communication initiated by a minifilter driver:

FilterGetMessage

FilterReplyMessage

Controlling Filter Manager Operation

4/26/2017 • 1 min to read • Edit Online

The operation of filter manager on versions of Windows earlier than Windows Vista is controlled by the REG_DWORD *AttachWhenLoaded* registry value stored under the following key:

```
HKLM\System\CurrentControlSet\Services\FltMgr
```

When *AttachWhenLoaded* is set to zero, the filter manager does not attach to any volumes until a minifilter driver registers with the filter manager. When *AttachWhenLoaded* is set to 1, the filter manager attaches at boot time to all volumes.

The default value for *AttachWhenLoaded* is zero with Windows XP with Service Pack 2 (SP2) and later.

The default value for *AttachWhenLoaded* is 1 on Windows Server 2003 with Service Pack 1 (SP1) and later versions.

The *AttachWhenLoaded* registry value does not exist on Windows Vista and only applies to previous versions of Windows.

When a minifilter driver is installed on Windows prior to Windows Vista, the software installer should set *AttachWhenLoaded* to 1 if this registry value is not currently set to 1. If the previous value of *AttachWhenLoaded* was zero, the installer should reboot the system after the installation of the minifilter driver.

Development and Testing Tools

4/26/2017 • 1 min to read • Edit Online

The filter manager tools described in this section are provided in the IFS Kit for Windows Server 2003 SP1 and in the Windows Driver Kit (WDK) for Windows Vista and later.

Minifilter driver developers are also encouraged to use general-purpose kernel-mode development and testing tools, such as PREfast with driver-specific rules.

Fltmc.exe Control Program

The Fltmc.exe control program is a command-line utility for common minifilter driver management operations. Developers can use Fltmc.exe to load and unload minifilter drivers, attach minifilter drivers to volumes or detach them from volumes, and enumerate minifilter drivers, instances, and volumes.

!fltkd Debugger Extension

The !fltkd debugger extension is provided in the Windows Debugging tools. Commonly used commands include the following:

COMMAND	DESCRIPTION
!cbd	The filter manager equivalent of !irp
!filter	Lists detailed information about the specified filter
!filters	Lists all attached minifilter drivers
!frames	Lists all filter manager frames and attached minifilter drivers
!instance	Lists detailed information about the specified instance
!volume	Lists detailed information about the specified volume
!volumes	Lists all volumes and attached minifilter driver instances

For additional debugging help, test the minifilter driver with the debug version of Fltmgr.sys, which contains numerous ASSERT statements to catch common errors.

Filter Verifier

Filter Verifier is an I/O Verification option in Driver Verifier that validates minifilter driver usage of filter manager functions. Filter Verifier is installed with the filter manager. Developers should always develop minifilter drivers with Driver Verifier and Filter Verifier enabled.

To use Filter Verifier, specify the minifilter driver's name and enable the I/O Verification option in Driver Verifier (Verifier.exe). Verification starts when the minifilter driver registers with the filter manager.

Filter Verifier validates the following usage in a minifilter driver:

- Correct use of parameters and calling context
- Correct return values from preoperation and postoperation callback routines
- Consistent and coherent changes to parameters in callback data

Filter Verifier tracks the following filter manager objects:

- Contexts
- Callback Data structures
- Queued Work Items
- NameInformation structures
- File Objects
- Filter Objects
- Instance Objects
- Volume Objects

Guidelines for Porting Legacy Filter Drivers

4/26/2017 • 4 min to read • Edit Online

Developers are encouraged to port legacy filter drivers to the filter manager model to obtain better functionality for their filter drivers and improve system reliability. Experienced developers should find it relatively easy to port a legacy filter driver to a minifilter driver. Filter driver developers at Microsoft recommend the following approach:

- Start with a reliable regression test suite to verify behavior between the legacy filter driver and the ported minifilter driver.
- Create a minifilter driver shell and systematically move functionality from the legacy filter driver to the minifilter driver. For example, get attachment working and then port one operation at a time, testing after each operation.
- Change user-mode/kernel-mode communication last, so you can use existing tools to test the minifilter driver.
- Compile with PREfast and test with the Filter Verifier I/O verification option in Driver Verifier enabled.

During the porting process, you should review all legacy filter driver code to take full advantage of filter manager capabilities. In particular, keep the following in mind:

- IRP-based I/O and fast I/O operations can come through the same operation when appropriate, which helps reduce duplication of code.
- When registering for operations, a minifilter driver can explicitly choose to ignore all paging I/O and cached I/O, which eliminates the need for code to check these.
- Instance notifications greatly simplify attach/detach logic.
- Register only for operations that your minifilter driver must handle; you can ignore everything else.
- Take advantage of filter manager context and name management support.
- Take advantage of filter manager support for issuing non-recursive I/O.
- Unlike legacy filter drivers, minifilter drivers cannot rely on local variables to maintain context from preoperation processing to postoperation processing. Consider allocating a lookaside list to store operation state.
- Be sure to release references when finished with a name or context.
- Completion ports in user mode add a powerful technique for building queues. You will probably need only a single connection to a single named port.

The following table lists common operations in a legacy filter driver and how they map to the filter manager model.

LEGACY FILTER DRIVER MODEL	FILTER MANAGER MODEL

LEGACY FILTER DRIVER MODEL	FILTER MANAGER MODEL
Pass-through operation with no completion routine	If your minifilter driver never does work for this type of I/O operation, do not register a preoperation or postoperation callback routine for this operation. Otherwise, return FLT_PREOP_SUCCESS_NO_CALLBACK from the preoperation callback routine registered for this operation. See Returning FLT_PREOP_SUCCESS_NO_CALLBACK.
Pass-through operation with a completion routine	Return FLT_PREOP_SUCCESS_WITH_CALLBACK from the preoperation callback routine. See Returning FLT_PREOP_SUCCESS_WITH_CALLBACK.
Pend operation in the preoperation callback routine	Call **FltLockUserBuffer** as needed to ensure that any user buffers are properly locked so that they can be accessed in a worker thread. Queue the work to a worker thread by calling support routines such as **FltAllocateDeferredIoWorkItem** and **FltQueueDeferredIoWorkItem**. Return FLT_PREOP_PENDING from the preoperation callback routine. When ready to return the I/O operation to the filter manager, call **FltCompletePendedPreOperation**. See Pending an I/O Operation in a Preoperation Callback Routine.
Pend operation in the postoperation callback routine	In the preoperation callback routine, call **FltLockUserBuffer** to ensure that user buffers are properly locked so that they can be accessed in a worker thread. Queue the work to a worker thread by calling support routines such as **FltAllocateGenericWorkItem** and **FltQueueGenericWorkItem**. Return FLT_POSTOP_MORE_PROCESSING_REQUIRED from the postoperation callback routine. When ready to return the I/O operation to the filter manager, call **FltCompletePendedPostOperation**. See Pending an I/O Operation in a Postoperation Callback Routine.
Synchronize the operation	Return FLT_PREOP_SYNCHRONIZE from the preoperation callback routine. See Returning FLT_PREOP_SYNCHRONIZE.

LEGACY FILTER DRIVER MODEL	FILTER MANAGER MODEL
Complete the operation in the preoperation callback routine	Set the final operation status and information in the **IoStatus** member of the **FLT_CALLBACK_DATA** structure for the operation. Return FLT_PREOP_COMPLETE from the preoperation callback routine. See Completing an I/O Operation in a Preoperation Callback Routine.
Complete the operation after it has been pended in the preoperation callback routine	Set the final operation status and information in the **IoStatus** member of the **FLT_CALLBACK_DATA** structure for the operation. Call **FltCompletePendedPreOperation** from the worker thread processing the I/O operation, passing FLT_PREOP_COMPLETE as the *CallbackStatus* parameter. See Completing an I/O Operation in a Preoperation Callback Routine.
Do all completion work in the completion routine	Return FLT_POSTOP_FINISHED_PROCESSING from the postoperation callback routine. See Writing Postoperation Callback Routines.
Do completion work at safe IRQL	Call **FltDoCompletionProcessingWhenSafe** from the postoperation callback routine. See Ensuring that Completion Processing is Performed at Safe IRQL.
Signal an event from the completion routine	Return FLT_PREOP_SYNCHRONIZE from the preoperation callback routine for this operation. The filter manager calls the postoperation callback routine in the same thread context as the preoperation callback routine, at IRQL <= APC_LEVEL. See Returning FLT_PREOP_SYNCHRONIZE.
Fail a successful create operation	Call **FltCancelFileOpen** from the postoperation callback routine for the create operation. Set an appropriate error NTSTATUS value in the **IoStatus** member of the **FLT_CALLBACK_DATA** structure for the operation. Return FLT_POSTOP_FINISHED_PROCESSING. See Failing an I/O Operation in a Postoperation Callback Routine.

LEGACY FILTER DRIVER MODEL	FILTER MANAGER MODEL
Disallow I/O through the fast I/O path for an I/O operation	Return FLT_STATUS_DISALLOW_FAST_IO from the preoperation callback routine for the operation. See Disallowing a Fast I/O Operation in a Preoperation Callback Routine.
Modify the parameters for an I/O operation	Set the modified parameter values in the **Iopb** member of the **FLT_CALLBACK_DATA** structure for the operation. Mark the FLT_CALLBACK_DATA structure as dirty by calling **FltSetCallbackDataDirty**, except when you have modified the contents of the **IoStatus** member of the FLT_CALLBACK_DATA structure. See Modifying the Parameters for an I/O Operation.
Lock the user buffer for the operation	Use the techniques and guidelines described in Accessing the User Buffers for an I/O Operation.

Installing a Minifilter Driver

4/26/2017 • 1 min to read • Edit Online

For Microsoft Windows XP and later operating systems, you should install your minifilter driver by using an INF file and an installation application. (On Windows 2000 and earlier operating systems, minifilter drivers were commonly installed by the Service Control Manager.)

In the future, INF-based installation is expected meet Windows Hardware Certification Kit requirements for minifilter drivers. Note that "INF-based installation" means only that you will need to use an INF file to copy files and to store information in the registry. You will not be required to install your entire product by using only an INF file, and you will not be required to provide a "right-click install" option for your driver.

This section includes:

Creating an INF File for a Minifilter Driver

Load Order Groups and Altitudes for Minifilter Drivers

Allocated Altitudes

Minifilter Altitude Request

Reparse Point Tag Request

Creating an INF File for a Minifilter Driver

4/26/2017 • 7 min to read • Edit Online

An INF file for a file system minifilter driver generally contains the following sections:

Version (required)

DestinationDirs (optional but recommended)

DefaultInstall (required)

DefaultInstall.Services (required)

ServiceInstall (required)

AddRegistry (required)

DefaultUninstall (optional)

DefaultUninstall.Services (optional)

Strings (required)

Note Starting with x64-based Windows Vista systems, all kernel-mode components, including non-PnP (Plug and Play) drivers, such as file system drivers (file system, legacy filter, and minifilter drivers), must be signed in order to load and execute. For this scenario, the following list contains information relevant to file system drivers:

- INF files for non-PnP drivers, including file system drivers, are not required to contain [Manufacturer] or [Models] sections.
- The **SignTool** command-line tool, located in the \bin\SelfSign directory of the WDK installation directory, can be used to directly "embed sign" a driver SYS executable file. For performance reasons, boot-start drivers must contain an embedded signature.
- Given an INF file, the Inf2cat command-line tool can be used to create a catalog (.cat) file for a driver package. Only catalog files can receive WHQL logo signatures.
- With Administrator privileges, an unsigned driver can still be installed on x64-based systems starting with Windows Vista. However, the driver will fail to load (and thus execute) because it is unsigned.
- For general information about signing drivers, see Driver Signing.
- For detailed information on the driving signing process, see Kernel-Mode Code Signing Walkthrough.
- All kernel-mode components, including custom kernel-mode development tools, must be signed. For more information, see Signing Drivers during Development and Test (Windows Vista and Later).

Version Section (required)

The **Version** section specifies a class and GUID that are determined by the type of minifilter driver, as shown in the following code example.

```
[Version]
Signature   = "$WINDOWS NT$"
Class       = "ActivityMonitor"
ClassGuid   = {b86dff51-a31e-4bac-b3cf-e8cfe75c9fc2}
Provider    = %Msft%
DriverVer   = 10/09/2001,1.0.0.0
CatalogFile =
```

The following table shows the values that file system minifilter drivers should specify in the **Version** section.

ENTRY	VALUE
Signature	"$WINDOWS NT$"
Class	See File System Filter Driver Classes and Class GUIDs.
ClassGuid	See File System Filter Driver Classes and Class GUIDs.
Provider	In your own INF file, you should specify a provider other than Microsoft.
DriverVer	See **INF DriverVer directive**.
CatalogFile	For antivirus minifilter drivers that are signed, this entry contains the name of a WHQL-supplied catalog file. All other minifilter drivers should leave this entry blank. For more information, see the description of the **CatalogFile** entry in **INF Version Section**.

DestinationDirs Section (optional but recommended)

The **DestinationDirs** section specifies the directories where minifilter driver and application files will be copied.

In this section and in the **ServiceInstall** section, you can specify well-known system directories by system-defined numeric values. For a list of these values, see **INF DestinationDirs Section**. In the following code example, the value 12 refers to the Drivers directory (%windir%\system32\drivers), and the value 10 refers to the Windows directory (%windir%).

```
[DestinationDirs]
DefaultDestDir = 12
Minispy.DriverFiles = 12
Minispy.UserFiles   = 10,FltMgr
```

DefaultInstall Section (required)

In the **DefaultInstall** section, a **CopyFiles** directive copies the minifilter driver's driver files and user-application files to the destinations that are specified in the **DestinationDirs** section.

Note The **CopyFiles** directive should not refer to the catalog file or the INF file itself. SetupAPI copies these files automatically.

You can create a single INF file to install your driver on multiple versions of the Windows operating system. You can create this type of INF file by creating additional **DefaultInstall**, **DefaultInstall.Services**, **DefaultUninstall**, and **DefaultUninstall.Services** sections for each operating system version. Each section is labeled with a *decoration* (for example, .ntx86, .ntia64, or .nt) that specifies the operating system version to which it applies. For more information about creating this type of INF file, see Creating INF Files for Multiple Platforms and Operating Systems.

The following code example shows a typical **DefaultInstall** section.

```
[DefaultInstall]
OptionDesc = %MinispyServiceDesc%
CopyFiles = Minispy.DriverFiles, Minispy.UserFiles
```

DefaultInstall.Services Section (required)

The **DefaultInstall.Services** section contains an **AddService** directive that controls how and when the services of a particular driver are loaded, as shown in the following code example.

```
[DefaultInstall.Services]
AddService = %MinispyServiceName%,,Minispy.Service
```

ServiceInstall Section (required)

The **ServiceInstall** section contains information used for loading the driver service. In the MiniSpy sample driver, this section is named "Minispy.Service", as shown in the following code example. The name of the **ServiceInstall** section must appear in an **AddService** directive in the **DefaultInstall.Services** section.

```
[Minispy.Service]
DisplayName    = %MinispyServiceName%
Description    = %MinispyServiceDesc%
ServiceBinary  = %12%\minispy.sys
ServiceType    = 2 ;    SERVICE_FILE_SYSTEM_DRIVER
StartType      = 3 ;    SERVICE_DEMAND_START
ErrorControl   = 1 ;    SERVICE_ERROR_NORMAL%
LoadOrderGroup = "FSFilter Activity Monitor"
AddReg         = Minispy.AddRegistry
Dependencies   = FltMgr
```

The **ServiceType** entry specifies the type of service. Minifilter drivers should specify a value of 2 (SERVICE_FILE_SYSTEM_DRIVER). For more information about the **ServiceType** entry, see **INF AddService Directive**.

The **StartType** entry specifies when to start the service. The following table lists the possible values for **StartType** and their corresponding start types.

VALUE	DESCRIPTION
0x00000000	SERVICE_BOOT_START
0x00000001	SERVICE_SYSTEM_START
0x00000002	SERVICE_AUTO_START
0x00000003	SERVICE_DEMAND_START

VALUE	DESCRIPTION
0x00000004	SERVICE_DISABLED

For more information about these start types, see "Driver Start Types" in What Determines When a Driver Is Loaded.

The **LoadOrderGroup** entry provides the filter manager with information that it needs to ensure interoperability between minifilter drivers and legacy file system filter drivers. You should specify a **LoadOrderGroup** value that is appropriate for the type of minifilter driver that you are developing. To choose a load order group, see Load Order Groups and Altitudes for Minifilter Drivers.

Note that you must specify a **LoadOrderGroup** value, even if your minifilter driver's start type is not SERVICE_BOOT_START. In this way, minifilter drivers are different from legacy file system filter drivers.

Note The filter manager's **StartType** value is SERVICE_BOOT_START, and its **LoadOrderGroup** value is FSFilter Infrastructure. These values ensure that the filter manager is always loaded before any minifilter drivers are loaded.

For more information about how the **StartType** and **LoadOrderGroup** entries determine when the driver is loaded, see What Determines When a Driver Is Loaded.

Note For minifilter drivers, unlike legacy file system filter drivers, the **StartType** and **LoadOrderGroup** values do not determine where the minifilter driver attaches in the minifilter instance stack. This location is determined by the altitude that is specified for the minifilter instance.

The **ErrorControl** entry specifies the action to be taken if the service fails to start during system startup. Minifilter drivers should specify a value of 1 (SERVICE_ERROR_NORMAL). For more information about the **ErrorControl** entry, see **INF AddService Directive**.

The **AddReg** directive refers to one or more INF writer-defined **AddRegistry** sections that contain information to be stored in the registry for the newly installed service. Minifilter drivers use **AddRegistry** sections to define minifilter driver instances and to specify a default instance.

The **Dependencies** entry specifies the names of any services or load order groups on which the driver depends. All minifilter drivers must specify FltMgr, which is the service name of the filter manager.

AddRegistry Section (required)

The **AddRegistry** section adds keys and values to the registry. Minifilter drivers use an **AddRegistry** section to define minifilter instances and to specify a default instance. This information is used whenever the filter manager creates a new instance for the minifilter driver.

In the MiniSpy sample driver, the following **AddRegistry** section, together with the %strkey% token definitions in the **Strings** section, defines three instances, one of which is named as the MiniSpy sample driver's default instance.

```
[Minispy.AddRegistry]
HKR,%RegInstancesSubkeyName%,%RegDefaultInstanceValueName%,0x00000000,%DefaultInstance%
HKR,%RegInstancesSubkeyName%"\"%Instance1.Name%,%RegAltitudeValueName%,0x00000000,%Instance1.Altitude%
HKR,%RegInstancesSubkeyName%"\"%Instance1.Name%,%RegFlagsValueName%,0x00010001,%Instance1.Flags%
HKR,%RegInstancesSubkeyName%"\"%Instance2.Name%,%RegAltitudeValueName%,0x00000000,%Instance2.Altitude%
HKR,%RegInstancesSubkeyName%"\"%Instance2.Name%,%RegFlagsValueName%,0x00010001,%Instance2.Flags%
HKR,%RegInstancesSubkeyName%"\"%Instance3.Name%,%RegAltitudeValueName%,0x00000000,%Instance3.Altitude%
HKR,%RegInstancesSubkeyName%"\"%Instance3.Name%,%RegFlagsValueName%,0x00010001,%Instance3.Flags%
```

DefaultUninstall Section (optional)

The **DefaultUninstall** section is optional but recommended if your driver can be uninstalled. It contains **DelFiles** and **DelReg** directives to remove files and registry entries, as shown in the following code example.

```
[DefaultUninstall]
DelFiles    = Minispy.DriverFiles, Minispy.UserFiles
DelReg      = Minispy.DelRegistry
```

DefaultUninstall.Services Section (optional)

The **DefaultUninstall.Services** section is optional but recommended if your driver can be uninstalled. It contains **DelService** directives to remove the minifilter driver's services, as shown in the following code example from the MiniSpy sample driver.

Note The **DelService** directive should always specify the SPSVCINST_STOPSERVICE flag (0x00000200) to stop the service before it is deleted.

```
[DefaultUninstall.Services]
DelService = Minispy,0x200
```

Strings Section (required)

The **Strings** section defines each %strkey% token that is used in the INF file.

You can create a single international INF file by creating additional locale-specific **Strings.***LanguageID* sections in the INF file. For more information about international INF files, see Creating International INF Files.

The following code example shows a typical **Strings** section.

```
[Strings]
Msft                = "Microsoft Corporation"
MinispyServiceDesc = "Minispy mini-filter driver"
MinispyServiceName = "Minispy"
RegInstancesSubkeyName = "Instances"
RegDefaultInstanceValueName  = "DefaultInstance"
RegAltitudeValueName    = "Altitude"
RegFlagsValueName  = "Flags"

DefaultInstance    = "Minispy - Top Instance"
Instance1.Name     = "Minispy - Middle Instance"
Instance1.Altitude = "370000"
Instance1.Flags    = 0x1 ; Suppress automatic attachments
Instance2.Name     = "Minispy - Bottom Instance"
Instance2.Altitude = "365000"
Instance2.Flags    = 0x1 ; Suppress automatic attachments
Instance3.Name     = "Minispy - Top Instance"
Instance3.Altitude = "385000"
Instance3.Flags    = 0x1 ; Suppress automatic attachments
```

Load Order Groups and Altitudes for Minifilter Drivers

4/26/2017 • 5 min to read • Edit Online

Windows uses a dedicated set of load order groups for file system filter drivers and minifilter drivers that are loaded at system startup.

Legacy file system filter drivers can attach only to the top of an existing file system driver stack and cannot attach in the middle of a stack. As a result, the start type for a driver and load order group are important to legacy file system filter drivers, because the earlier a filter driver loads, the lower it can attach on the file system driver stack.

Drivers are loaded first based on the start type for the driver, which represents phases of booting a system. For more information about start types, see "Driver Start Types" in What Determines When a Driver Is Loaded. All file system filter drivers and minifilter drivers that specify a start type of SERVICE_BOOT_START will be loaded before drivers with a start type of SERVICE_SYSTEM_START or SERVICE_AUTO_START. The start type is specified by the **StartType** entry in the ServiceInstall Section of an INF file that is used to install the minifilter driver. Within each start type category, the load order group determines when file system filter drivers and minifilter drivers will be loaded.

A minifilter driver can be loaded at any time. The concept of load order groups is still required by minifilter drivers for interoperability with legacy file system filter drivers. Every minifilter driver must have a unique identifier called *altitude*. The altitude of a minifilter driver defines its position relative to other minifilter drivers in the I/O stack when the minifilter driver is loaded. The altitude is an infinite-precision string interpreted as a decimal number. A minifilter driver that has a low numerical altitude is loaded into the I/O stack below a minifilter driver that has a higher numerical value.

Each load order group has a defined range of altitudes. The allocation of altitudes to minifilter drivers is managed by Microsoft. To request an altitude for your minifilter driver, send an email message to fsfcomm@microsoft.com asking for one to be assigned.

A minifilter driver must specify an altitude value from an altitude range that represents a load order group. Altitude values for a minifilter driver are specified in the Instance definitions of the Strings Section in the INF file that is used to install the minifilter driver. Instance definitions can also be specified in calls to the **InstanceSetupCallback** routine in the **FLT_REGISTRATION** structure. Multiple instances and altitudes can be defined for a minifilter driver. These instance definitions apply across all volumes.

The following rules about start type and load order groups determine when a minifilter driver will be loaded:

- A minifilter driver that specifies a particular start type and load order group is loaded at the same time as other file system filter drivers and minifilter drivers in that start type and load order group.
- Within each load order group, file system filter drivers and minifilter drivers are generally loaded in random order. This normally results in drivers being loaded based on the order in which the driver was installed.
- If a file system filter driver or minifilter driver does not specify a load order group, it is loaded after all the other drivers of the same start type that do specify a load order group.

The following table lists the system-defined load order groups and altitude ranges for minifilter drivers. For each load order group, the Load order group column contains the value that should be specified for that group in the **LoadOrderGroup** entry in the ServiceInstall Section of a minifilter's INF file. The Altitude range column contains the range of altitudes for a particular load order group. A minifilter driver must request an altitude allocation from Microsoft in the appropriate load order group or groups.

Note that the load order groups and altitude ranges are listed as they appear on the stack, which is the reverse of the order in which they are loaded.

LOAD ORDER GROUP	ALTITUDE RANGE	DESCRIPTION
Filter	420000-429999	This group is the same as the Filter load order group that was available on Windows 2000 and earlier. This group loads last and thus attaches furthest from the file system.
FSFilter Top	400000-409999	This group is provided for filter drivers that must attach above all other FSFilter types.
FSFilter Activity Monitor	360000-389999	This group includes filter drivers that observe and report on file I/O.
FSFilter Undelete	340000-349999	This group includes filters that recover deleted files.
FSFilter Anti-Virus	320000-329999	This group includes filter drivers that detect and disinfect viruses during file I/O.
FSFilter Replication	300000-309999	This group includes filter drivers that replicate file data to remote servers.
FSFilter Continuous Backup	280000-289999	This group includes filter drivers that replicate file data to backup media.
FSFilter Content Screener	260000-269999	This group includes filter drivers that prevent the creation of specific files or file content.
FSFilter Quota Management	240000-249999	This group includes filter drivers that provide enhanced file system quotas.
FSFilter System Recovery	220000-229999	This group includes filter drivers that perform operations to maintain operating system integrity, such as the System Restore (SR) filter.

LOAD ORDER GROUP	ALTITUDE RANGE	DESCRIPTION
FSFilter Cluster File System	200000-209999	This group includes filter drivers that are used in products that provide file server metadata across a network.
FSFilter HSM	180000-189999	This group includes filter drivers that perform hierarchical storage management.
FSFilter Imaging	170000-175000	This group includes ZIP-like filter drivers that provide a virtual namespace. This load group is available on Windows Vista and later versions of the operating system.
FSFilter Compression	160000-169999	This group includes filter drivers that perform file data compression.
FSFilter Encryption	140000-149999	This group includes filter drivers that encrypt and decrypt data during file I/O.
FSFilter Virtualization	130000- 139999	This group includes filter drivers that virtualize the file path, such as the Least Authorized User (LUA) filter driver added in Windows Vista. This load group is available on Windows Vista and later versions of the operating system.
FSFilter Physical Quota Management	120000-129999	This group includes filter drivers that manage quotas by using physical block counts.
FSFilter Open File	100000-109999	This group includes filter drivers that provide snapshots of already open files.
FSFilter Security Enhancer	80000-89999	This group includes filter drivers that apply lockdown and enhanced access control lists (ACLs).
FSFilter Copy Protection	60000-69999	This group includes filter drivers that check for out-of-band data on media.

LOAD ORDER GROUP	ALTITUDE RANGE	DESCRIPTION
FSFilter Bottom	40000-49999	This group is provided for filter drivers that must attach below all other FSFilter types.
FSFilter System	20000-29999	Reserved for internal use.
FSFilter Infrastructure		Reserved for internal use. This group loads first and thus attaches closest to the file system.

Allocated Altitudes

6/15/2017 • 29 min to read • Edit Online

A file system minifilter driver developed to the Filter Manager model must have a unique identifier called an altitude that defines its position relative to other minifilters present in the file system stack. Minifilter altitudes are allocated by Microsoft based on minifilter requirements and load order group.

The current altitude allocations are listed for each of the following load order groups.

420000 - 429999: Filter

MINIFILTER	ALTITUDE	COMPANY
ntoskrnl.exe	425500	Microsoft
ntoskrnl.exe	425000	Microsoft

400000 - 409999: FSFilter Top

MINIFILTER	ALTITUDE	COMPANY
wcnfs.sys	409900	Microsoft
iorate.sys	409010	Microsoft
ioqos.sys	409000	Microsoft
fsdepends.sys	407000	Microsoft
sftredir.sys	406000	Microsoft
dfs.sys	405000	Microsoft
csvnsflt.sys	404900	Microsoft
csvflt.sys	404800	Microsoft
Microsoft.Uev.AgentDriver.sys	404710	Microsoft
AppvVfs.sys	404700	Microsoft
eaw.sys	401900	Raytheon Cyber Solutions
TVFsfilter.sys	401800	TrustView
KKDiskProtecter.sys	401700	Goldmsg
AgentComm.sys	401600	Trustwave Holding

MINIFILTER	ALTITUDE	COMPANY
rvsavd.sys	401500	CJSC Returnil Software
dgdmk.sys	401400	Verdasys Inc.
tusbstorfilt.sys	401300	SimplyCore LLC
pcgenfam.sys	401200	Soluto
atrsdfw.sys	401100	Altiris
tpfilter.sys	401000	RedPhone Security
mbamwatchdog.sys	400900	Malwarebytes Corporation
edevmon.sys	400800	ESET
vmwflstor.sys	400700	VMware, Inc.
TsQBDrv.sys	400600	Tencent Technology

360000 - 389999: FSFilter Activity Monitor

MINIFILTER	ALTITUDE	COMPANY
drbdlock.sys	389090	Man Technology Inc
hsmltmon.sys	389080	Hitachi Solutions
AternityRegistryHook.sys	389070	Aternity Ltd
CSBFilter.sys	389060	Carbonite Inc
ChemometecFilter.sys	389050	ChemoMetec
SentinelMonitor.sys	389040	SentinelOne
DhWatchdog.sys	389030	Microsoft
edrsensor.sys	389025	Bitdefender SRL
NpEtw.sys	389020	Koby Kahane
OczMiniFilter.sys	389010	OCZ Storage
ielcp.sys	389004	Intel Corporation
IESlp.sys	389002	Intel Corporation
IntelCAS.sys	389000	Intel Corporation

MINIFILTER	ALTITUDE	COMPANY
boxifier.sys	388990	Kenubi
SamsungRapidFSFltr.sys	388980	NVELO
drsfile.sys	388970	MRY Inc.
CrUnCopy.sys	388964	Shenzhen CloudRiver
aictracedrv_am.sys	388960	AI Consulting
fiopolicyfilter.sys	388954	SanDisk
fcontrol.sys	388950	SODATSW spol. s r.o.
qfilter.sys	388940	Quorum Labs
Redlight.sys	388930	Trustware Ltd
eps.sys	388920	Lumension
VHDTrack.sys	388915	Intronis Inc
VHDDelta.sys	388912	Niriva LLC
FSTrace.sys	388910	Niriva LLC
YahooStorage.sys	388900	Yahoo Japan Corporation
KeWF.sys	388890	KEBA AG
epregflt.sys	388888	Check Point Software
zsfprt.sys	388880	k4solution Co.
dsflt.sys	388876	cEncrypt
bfaccess.sys	388872	bitFence Inc.
xcpl.sys	388870	X-Cloud Systems
RMPHVMonitor.sys	388865	ManageEngine Zoho
FAPMonitor.sys	388864	ManageEngine Zoho
EaseFlt.sys	388860	EaseVault Technologies Inc.
sieflt.sys	388852	Quick Heal Technologies Pvt. Ltd.
cssdlp.sys	388851	Quick Heal Technologies Pvt. Ltd.

MINIFILTER	ALTITUDE	COMPANY
cssdlp.sys	388850	CoSoSys
INISBDrv64.sys	388840	Initech Inc.
trace.sys	388831	Fitsec Ltd
SandDriver.sys	388830	Fitsec Ltd
dskmn.sys	388820	Honeycomb Technologies
xkfsfd.sys	388810	Jiransoft Co., Ltd
pcpifd.sys	388800	Jiransoft Co., Ltd
NNTInfo.sys	388790	New Net Technologies Limited
FsMonitor.sys	388780	IBM
CVCBT.sys	388770	CommVault Systems, Inc.
AwareCore.sys	388760	TaaSera
laFS.sys	388750	NetworkProfi Ltd.
fsnk.sys	388740	SoftPerfect Research
RGNT.sys	388730	HFN
fltRs329.sys	388720	Secured Globe Inc.
ospmon.sys	388710	SC ODEKIN SOLUTIONS SRL
edsigk.sys	388700	Enterprise Data Solutions, Inc.
fiometer.sys	388692	Fusion-io
dcSnapRestore.sys	388690	Fusion-io
fam.sys	388680	Quick Heal Technologies Pvt. Ltd.
vidderfs.sys	388675	Vidder Inc.
Tritiumfltr.sys	388670	Tritium Inc.
HexisFSMonitor.sys	388660	Hexis Cyber Solutions
BlackbirdFSA.sys	388650	BeyondTrust Inc.
TMUMS.sys	388642	Trend Micro Inc.

MINIFILTER	ALTITUDE	COMPANY
hfileflt.sys	388640	Trend Micro Inc.
TMUMH.sys	388630	Trend Micro Inc.
AcDriver.sys	388620	Trend Micro, Inc.
SakFile.sys	388610	Trend Micro, Inc.
SakMFile.sys	388600	Trend Micro, Inc.
rsfdrv.sys	388580	Clonix Co
alcapture.sys	388570	Airlock Digital Pty Ltd
kmNWCH.sys	388560	IGLOO SECURITY, Inc.
ISIRMFmon.sys	388550	ALPS SYSTEM INTERGRATION CO.
heimdall.sys	388540	Arkoon Network Security
thetta.sys	388532	Syncopate
thetta.sys	388531	Syncopate
thetta.sys	388530	Syncopate
DTPL.sys	388520	CONNECT SHIFT LTD
CyOptics.sys	388514	Cylance Inc.
CyProtectDrv32.sys	388510	Cylance Inc.
CyProtectDrv64.sys	388510	Cylance Inc.
tbfsfilt.sys	388500	Third Brigade
LDSecDrv.sys	388490	LANDESK Software
SGResFlt.sys	388480	Samsung SDS Ltd
CwMem2k64.sys	388470	ApSoft
axfltdrv.sys	388460	Axact Pvt Ltd
RMDiskMon.sys	388450	Qingdao Ruanmei Network Technology Co.
diskactmon.sys	388440	Qingdao Ruanmei Network Technology Co.

MINIFILTER	ALTITUDE	COMPANY
Codex.sys	388430	GameHi Co.
CatMF.sys	388420	Logichron
RW7FsFlt.sys	388410	PJSC KP VTI
aswSP.sys	388401	Avast Software
aswFsBlk.sys	388400	ALWIL Software
ThreatStackFIM.sys	388380	Threat Stack
BOsCmFlt.sys	388370	Barkly Protects Inc.
BOsFsFltr.sys	388370	Barkly Protects Inc.
FeKern.sys	388360	FireEye Inc.
libwamf.sys	388350	OPSWAT Inc.
szardrv.sys	388345	SaferZone Co.
szdfmdrv.sys	388340	SaferZone Co.
szdfmdrv_usb.sys	388331	SaferZone Co.
sprtdrv.sys	388330	SaferZone Co.
SWFsFltr.sys	388320	Solarwinds LLC
flashaccelfs.sys	388310	Network Appliance
changelog.sys	388300	Network Appliance
stcvsm.sys	388250	StorageCraft Tech
aUpDrv.sys	388240	ITSTATION Inc
fshs.sys	388222	F-Secure
fshs.sys	388221	F-Secure
fsatp.sys	388220	F-Secure
SecdoDriver.sys	388210	Secdo
TGFSMF.sys	388200	Tetraglyph Technologies
evscase.sys	388100	March Hare Software Ltd

MINIFILTER	ALTITUDE	COMPANY
VSScanner.sys	388050	VoodooSoft
tsifilemon.sys	388012	Intercom
MarSpy.sys	388010	Intercom
BrnFileLock.sys	388000	Blue Ridge Networks
BrnSecLock.sys	387990	Blue Ridge Networks
LCmPrintMon.sys	387978	YATEM Co. Ltd.
LCgAdMon.sys	387977	YATEM Co. Ltd.
LCmAdMon.sys	387976	YATEM Co. Ltd.
LCgFileMon.sys	387975	YATEM Co. Ltd.
LCmFile.sys	387974	YATEM Co. Ltd.
LCgFile.sys	387972	YATEM Co. Ltd.
LCmFileMon.sys	387970	YATEM Co. Ltd.
IridiumSwitch.sys	387960	Confio
scensemon.sys	387950	Scense
ruaff.sys	387940	RUNEXY
bbfilter.sys	387930	derivo GmbH
Bfmon.sys	387920	Baidu (Hong Kong) Limited
bdsysmon.sys	387912	Baidu Online Network
BdRdFolder.sys	387910	Baidu (beijing)
pscff.sys	387900	Weing Co.,Ltd.
fcnotify.sys	387880	TCXA Ltd.
aaf.sys	387860	Actifio Inc
gddcv.sys	387840	G Data Software AG
wgfile.sys	387820	ORANGE WERKS Inc
zesfsmf.sys	387800	Novell

MINIFILTER	ALTITUDE	COMPANY
uamflt.sys	387700	Sevtechnotrans
ehdrv.sys	387600	ESET, spol. s r.o.
Snilog.sys	387500	Systemneeds, Inc
tss.sys	387400	Tiversa Inc
LmDriver.sys	387390	in-soft Kft.
WDCFilter.sys	387330	Interset Inc.
altcbt.sys	387320	Altaro Ltd.
bapfecpt.sys	387310	BitArmor Systems, Inc
bamfltr.sys	387300	BitArmor Systems, Inc
TrustedEdgeFfd.sys	387200	FileTek, Inc.
MRxGoogle.sys	387100	Google, Inc.
YFSDR.SYS	387010	Yokogawa R&L Corp
YFSD.SYS	387000	Yokogawa R&L Corp
YFSRD.sys	386990	Yokogawa R&L Corp
psgfoctrl.sys	386990	Yokogawa R&L Corp
USBPDH.SYS	386901	Centre for Development of Advanced Computing
pecfilter.sys	386900	C-DAC Hyderabad
GKPFCB.sys	386810	INCA Internet Co.
GKPFCB64.sys	386810	INCA Internet Co.
TkPcFtCb.sys on 32bit	386800	INCA Internet Co.,Ltd.
TkPcFtCb64.sys on 64bit	386800	INCA Internet Co.,Ltd.
bmregdrv.sys	386731	Yandex LLC
bmfsdrv.sys	386730	Yandex LLC
CarbonBlackK.sys	386720	Bit9 Inc.
ScAuthFSFlt.sys	386710	Security Code LLC

MINIFILTER	ALTITUDE	COMPANY
ScAuthIoDrv.sys	386700	Security Code LLC
mfeaskm.sys	386610	McAfee
mfencfilter.sys	386600	McAfee
WinFLAHdrv.sys	386540	NewSoftwares.net
WinFLAdrv.sys	386530	NewSoftwares.net
WinDBdrv.sys	386520	NewSoftwares.net,Inc.
WinFLdrv.sys	386510	NewSoftwares.net,Inc.
WinFPdrv.sys	386500	NewSoftwares.net,Inc.
SkyAMDrv.sys	386430	Sky Co.
SheedSelfProtection.sys	386421	SheedSoft Ltd
arta.sys	386420	SheedSoft Ltd.
ApexSqlFilterDriver.sys	386410	ApexSQL LLC
stflt.sys	386400	Xacti
tbrdrv.sys	386390	Crawler Group
WinTeonMiniFilter.sys	386320	Dmitry Stefankov
wiper.sys	386310	Dmitry Stefankov
DevMonMiniFilter.sys	386300	Dmitry Stefankov
VMWVvpfsd.sys	386200	VMware, Inc.
RTOLogon.sys (Renamed)	386200	VMware, Inc.
Code42Filter.sys	386190	Code42
FileGuard.sys	386140	RES Software
NetGuard.sys	386130	RES Software
RegGuard.sys	386120	RES Software
ImgGuard.sys	386110	RES Software
AppGuard.sys	386100	RES Software

MINIFILTER	ALTITUDE	COMPANY
minitrc.sys	386020	Protected Networks
cpepmon.sys	386010	Checkpoint Software
CGWMF.sys	386000	NetIQ
ISRegFlt.sys	385990	Flexera Software
ISRegFlt64.sys	385990	Flexera Software
ctrPAMon.sys	385960	Comtrue Technology
shdlpMedia.sys	385950	Comtrue Technology
immflex.sys	385910	Immidio B.V.
StegoProtect.sys	385900	Stegosystems
brfilter.sys	385890	Bromium Inc
BrCow_x_x_x_x.sys	385889	Bromium Inc
secRMM.sys	385880	Squadra Technologies
dgfilter.sys	385870	DataGravity Inc.
WFP_MRT.sys	385860	FireEye Inc
mssecflt.sys	385600	Microsoft
Backupreader.sys	385500	Microsoft
AppVMon.sys	385400	Microsoft
DpmFilter.sys	385300	Microsoft
Sysmondrv.sys	385201	Microsoft
Procmon11.sys	385200	Microsoft
minispy.sys - Top	385100	Microsoft
fdrtrace.sys	385001	Microsoft
filetrace.sys	385000	Microsoft
uwfreg.sys	384910	Microsoft
uwfs.sys	384900	Microsoft

MINIFILTER	ALTITUDE	COMPANY
FilrDriver.sys	383340	Micro Focus
rwchangedrv.sys	383330	Rackware
airship-filter.sys	383320	AIRWare Technology Ltd
AeFilter.sys	383310	Faronics Corporation
QQProtect.sys	383300	Tencent (Shenzhen)
QQProtectX64.sys	383300	Tencent (Shenzhen)
KernelAgent32.sys	383260	ZoneFox
WRDWIZFILEPROT.SYS	383251	WardWiz
WRDWIZREGPROT.SYS	383250	WardWiz
groundling32.sys	383200	Dell Secureworks
groundling64.sys	383200	Dell Secureworks
avgtpx86.sys	383190	AVG Technologies CZ
avgtpx64.sys	383190	AVG Technologies CZ
DataNow_Driver.sys	383182	AppSense Ltd
UcaFltDriver.sys	383180	AppSense Ltd
YFSD2.sys	383170	Yokogawa Corpration
Kisknl.sys	383160	kingsoft
MWatcher.sys	383150	Neowiz Corporation
tsifilemon.sys	383140	Intercom
FIM.sys	383130	eIQnetworks
cFSfdrv	383120	Chaewool
ajfsprot.sys	383110	Analytik Jena AG
isafermon	383100	(c)SMS
kfac.sys	383000	Beijing CA-JinChen Software Co.
GUMHFilter.sys	382910	Glarysoft Ltd.

MINIFILTER	ALTITUDE	COMPANY
FJGSDis2.sys	382900	FUJITSU LIMITED
secure_os.sys	382890	FUJITSU SOCIAL SCIENCE
zqFilter	382800	magrasoft Ltd
ntps_fa.sys	382700	NTP Software
sConnect.sys	382600	I-O DATA DEVICE
AdaptivaClientCache32.sys	382500	Adaptiva
AdaptivaclientCache64.sys	382500	Adaptiva
GoFSMF.sys	382430	Gorizonty Rosta Ltd
SWCommFltr.sys	382420	SnoopWall LLC
atflt.sys	382410	Atlansys Software
amfd.sys	382400	Atlansys Software
iSafeKrnl.sys	382390	Elex Tech Inc
iSafeKrnlMon.sys	382380	Elex Tech Inc
360box.sys	382310	Qihoo 360
360fsflt.sys	382300	Beijing Qihoo Technology Co.
scred.sys	382210	SoftCamp Co.
PDGenFam.sys	382200	Soluto LTD
MCFileMon64.sys (x64 systems)	382100	Sumitomo Electric Ltd.
MCFileMon32.sys (x32 systems)	382100	Sumitomo Electric Ltd.
RyGuard.sys	382050	SHENZHEN UNNOO Information Techco.
FileShareMon.sys	382040	SHENZHEN UNNOO Information Techco.
ryfilter.sys	382030	SHENZHEN UNNOO Information Techco.
secufile.sys	382020	Shenzhen Unnoo LTD
XiaobaiFs.sys	382010	Shenzhen Unnoo LTD

MINIFILTER	ALTITUDE	COMPANY
XiaobaiFsR.sys	382000	Shenzhen Unnoo LTD
TWBDCFilter.sys	381910	Trustwave
VPDrvNt.sys	381900	AhnLab, Inc.
eetd32.sys	381800	Entrust Inc.
eetd64.sys	381800	Entrust Inc.
dnaFSMonitor.sys	381700	Dtex Systems
iwhlp2.sys on 2000	381610	InfoWatch
iwhlpxp.sys on XP	381610	InfoWatch
iwhlp.sys on Vista	381610	InfoWatch
iwdmfs.sys	381600	InfoWatch
IronGateFD.sys	381500	rubysoft
MagicBackupMonitor.sys	381400	Magic Softworks, Inc.
Sonar.sys	381337	IKARUS Security
IPFilter.sys	381310	Jinfengshuntai
MSpy.sys	381300	Ladislav Zezula
inuse.sys	381200	March Hare Software Ltd
qfmon.sys	381190	Quality Corporation
flyfs.sys	381160	NEC Soft
serfs.sys	381150	NEC Soft
hdrfs.sys	381140	NEC Soft
UVMCIFSF.sys	381130	NEC Corporation
ICFClientFlt.sys	381120	NEC System Technologies,Ltd.
IccFileIoAd.sys	381110	NEC System Technologies,Ltd.
IccFilterAudit.sys	381100	NEC System Technologies
IccFilterSc.sys	381090	InfoCage

MINIFILTER	ALTITUDE	COMPANY
mtsvcdf.sys	381000	CristaLink
SQLsafeFilterDriver.sys	380901	Idera Software
IderaFilterDriver.sys	380900	Idera
xhunter1.sys	380800	Wellbia.com
iGuard.sys	380720	i-Guard SAS
cbfltfs4.sys	380710	Backup Systems Ltd
PkgFilter.sys	380700	Scalable Software
snimg.sys	380600	Softnext Technologies
SK.sys	380520	HEAT Software
mpxmon.sys	380510	Positive Technologies
KC3.sys	380500	Infotecs
PLPOffDrv.sys	380492	SK Infosec Co
ISFPDrv.sys	380491	SK Infosec Co
ionmonwdrv.sys	380490	SK Infosec Co
CbSampleDrv.sys	380020	Microsoft
CbSampleDrv.sys	380010	Microsoft
CbSampleDrv.sys	380000	Microsoft
simrep.sys	371100	Microsoft
change.sys	370160	Microsoft
delete_flt.sys	370150	Microsoft
SmbResilFilter.sys	370140	Microsoft
usbtest.sys	370130	Microsoft
NameChanger.sys	370120	Microsoft
failMount.sys	370110	Microsoft
failAttach.sys	370100	Microsoft

MINIFILTER	ALTITUDE	COMPANY
stest.sys	370090	Microsoft
cdo.sys	370080	Microsoft
ctx.sys	370070	Microsoft
fmm.sys	370060	Microsoft
cancelSafe.sys	370050	Microsoft
message.sys	370040	Microsoft
passThrough.sys	370030	Microsoft
nullFilter.sys	370020	Microsoft
ntest.sys	370010	Microsoft
minispy.sys - Middle	370000	Microsoft
nravwka.sys	368400	NURILAB
bhkavki.sys	368390	NURILAB
bhkavka.sys	368390	NURILAB
docvmonk.sys	368380	NURILAB
docvmonk64.sys	368380	NURILAB
InvProtectDrv.sys	368370	Invincea
InvProtectDrv64.sys	368370	Invincea
browserMon.sys	368360	Adtrustmedia
SfdFilter.sys	368350	Sandoll Communication
phdcbtdrv.sys	368340	PHD Virtual Tech Inc.
sysdiag.sys	368330	HeroBravo Technology
cfrmd.sys	368320	Comodo Security Solutions
repdrv.sys	368310	Vision Solutions
repdrv.sys	368310	Vision Solutions
repmon.sys	368300	Vision Solutions

MINIFILTER	ALTITUDE	COMPANY
cvofflineFlt32.sys	368200	Quantum Corporation.
cvofflineFlt64.sys	368200	Quantum Corporation.
DsDriver.sys	368100	Warp Disk Software
nlcbhelpx86.sys	368000	NetLib
nlcbhelpx64.sys	368000	NetLib
nlcbhelpi64.sys	368000	NetLib
wbfilter.sys	367950	Whitebox Security
LRAgentMF.sys	367900	LogRhythm
Drwebfwflt.sys	367810	Doctor Web
EventMon.sys	367800	Doctor Web
soidriver.sys	367750	Sophos Plc
drvhookcsmf.sys	367700	GrammaTech, Inc.
drvhookcsmf_amd64.sys	367700	GrammaTech, Inc.
avipbb.sys	367600	Avira GmbH
FileSightMF.sys	367500	PA File Sight
csaam.sys	367400	Cisco Systems
FSMon.sys	367300	1mill
mfl.sys	367200	OSR Open Systems Resources, Inc.
filefilter.sys	367100	Beijing Zhong Hang Jiaxin Computer Technology Co.,Ltd.
iiscache.sys	367000	Microsoft
nowonmf.sys	366993	Diskeeper Corporation
dktlfsmf.sys	366992	Diskeeper Corporation
DKDrv.sys	366991	Diskeeper Corporation
DKRtWrt.sys - temp fix for XPSP3	366990	Diskeeper Corporation
HBFSFltr.sys	366980	Diskeeper Corporation

MINIFILTER	ALTITUDE	COMPANY
xoiv8x64.sys	366940	Arcserve
xomfcbt8x64.sys	366930	CA
KmxAgent.sys	366920	CA
KmxFile.sys	366910	CA
KmxSbx.sys	366900	CA
PointGuardVistaR32.sys	366810	Futuresoft
PointGuardVistaR64.sys	366810	Futuresoft
PointGuardVistaF.sys	366800	Futuresoft
PointGuardVista64F.sys	366800	Futuresoft
vintmfs.sys	366789	CondusivTechnologies
hiofs.sys	366782	Condusiv Technologies
intmfs.sys	366781	CondusivTechnologies
excfs.sys	366780	CondusivTechnologies
zampit_ml.sys	366700	Zampit
rflog.sys	366600	AppStream, Inc.
LivedriveFilter.sys	366500	Livedrive Internet Ltd
regmonex.sys	366410	Tranxition Corp
TXRegMon.sys	366400	Tranxition Corp
SDVFilter.sys	366300	Soliton Systems K.K.
eLock2FSCTLDriver.sys	366210	Egis Technology Inc.
msiodrv4.sys	366200	Centennial Software Ltd
mmPsy32.sys	366110	Resplendence Software Projects
mmPsy64.sys	366110	Resplendence Software Projects
rrMon32.sys	366100	Resplendence Software Projects
rrMon64.sys	366100	Resplendence Software Projects

MINIFILTER	ALTITUDE	COMPANY
cvsflt.sys	366000	March Hare Software Ltd
ktsyncfsflt.sys	365920	KnowledgeTree Inc.
nvmon.sys	365900	NetVision, Inc.
SnDacs.sys	365810	Informzaschita
SnExequota.sys	365800	Informzaschita
llfilter.sys	365700	SecureAxis Software
hafsnk.sys	365660	HA Unix Pt
DgeDriver.sys	365655	Dell Software Inc.
BWFSDrv.sys	365650	Quest Software Inc.
QFAPFlt.sys	365600	Quest Software
XendowFLT.sys	365570	Credant Technologies
fmdrive.sys	365500	Cigital, Inc.
EGMinFlt.sys	365400	WhiteCell Software Inc.
it2reg.sys	365315	Soliton Systems
it2drv.sys	365310	Soliton Systems
solitkm.sys	365300	Soliton Systems
pgpwdefs.sys	365270	Symantec
GEProtection.sys	365260	Symantec
diflt.sys	365260	Symantec Corp.
sysMon.sys	365250	Symantec
ssrfsf.sys	365210	Symantec
emxdrv2.sys	365200	Symantec
reghook.sys	365150	Symantec
spbbcdrv.sys	365100	Symantec
bhdrvx86.sys	365100	Symantec

MINIFILTER	ALTITUDE	COMPANY
bhdrvx64.sys	365100	Symantec
SISIPSFileFilter	365010	Symantec
symevent.sys	365000	Symantec
wrpfv.sys	364900	Microsoft
UpGuardRealTime.sys	364810	UpGuard
usbl_ifsfltr.sys	364800	SecureAxis
ntfsf.sys	364700	Sun&Moon Rise
BssAudit.sys	364600	ByStorm
GPMiniFIlter.sys	364500	Kalpataru
AlfaFF.sys	364400	Alfa
FSAFilter.sys	364300	ScriptLogic
GcfFilter.sys	364200	GemacmbH
FFCFILT.SYS	364100	FFC Limited
msnfsflt.sys	364000	Microsoft
mblmon.sys	363900	Packeteer
amsfilter.sys	363800	Axur Information Sec.
strapvista.sys (retired)	363700	AvSoft Technologies
SAFE-Agent.sys	363636	SAFE-Cyberdefense
EstPrmon.sys	363610	ESTsoft corp.
Estprp.sys - 64bit	363610	ESTsoft corp.
EstRegmon.sys	363600	ESTsoft corp.
EstRegp.sys - 64bit	363600	ESTsoft corp.
agfsmon.sys	363530	TechnoKom Ltd.
NlxFF.sys	363520	OnGuard Systems LLC
Sahara.sys	363511	Safend

MINIFILTER	ALTITUDE	COMPANY
Santa.sys	363510	Safend
vfdrv.sys	363500	Viewfinity
xhunter64.sys	363400	Wellbia.com
SPIMiniFilter.sys	363300	Software Pursuits
mracdrv.sys	363230	Mail.Ru
BEDaisy.sys	363220	BattlEye Innovations
NetAccCtrl.sys	363200	LINK co.
NetAccCtrl64.sys	363200	LINK co.
hpreg.sys	363130	HP
qfimdvr.sys	363120	Qualys Inc.
dsfemon.sys	363100	Topology Ltd
dsfemon.sys	363100	Topology Ltd
AmznMon.sys	363030	Amazon Web Services Inc
iothorfs.sys	363020	ioScience
ctamflt.sys	363010	ComTrade
psisolator.sys	363000	SharpCrafters
QmInspec.sys	362990	Beijing QiAnXin Tech.
GagSecurity.sys	362970	Beijing Shu Yan Science
CybKernelTracker.sys	362960	CyberArk Software
filemon.sys	362950	Temasoft S.R.L.
klifks.sys	362902	Kaspersky Lab
klifaa.sys	362901	Kaspersky Lab
Klifsm.sys	362900	Kaspersky Lab
minispy.sys - Bottom	361000	Microsoft

340000 - 349999: FSFilter Undelete

MINIFILTER	ALTITUDE	COMPANY
BSSFlt.sys	346000	Blue Shoe Software LLC
ThinIO.sys	345900	ThinScale Technology
hmpalert.sys	345800	SurfRight
nsffkmd64.sys	345700	NetSTAR Inc.
nsffkmd32.sys	345700	NetSTAR Inc.
xbprocfilter.sys	345600	Zrxb
ifileguard.sys	345500	I-O DATA DEVICE, INC.
undelex32.sys	345400	Resplendence Software Projects
undelex64.sys	345400	Resplendence Software Projects
starmon.sys	345300	Kantowitz Engineering, Inc.
mxRCycle.sys	345200	Avanquest
UdFilter.sys	345100	Diskeeper Corporation
it2prtc.sys	345040	Soliton Systems K.K.
SolRegFilter.sys	345030	Soliton Systems K.K.
SolSecBr.sys	345020	Soliton Systems K.K.
SolFCLLi.sys	345010	Soliton Systems K.K.
SolFCL.sys	345000	Soliton Smart Sec

320000 - 329998: FSFilter Anti-Virus

MINIFILTER	ALTITUDE	COMPANY
DeepInsFS.sys	329190	Deep Instinct
AppCheckD.sys	329180	CheckMAL Inc
spellmon.sys	329170	SpellSecurity
WhiteShield.sys	329160	Meidensha Corp
reaqtor.sys	329150	ReaQta Ltd.
SE46Filter.sys	329140	Technology Nexus AB

MINIFILTER	ALTITUDE	COMPANY
FileScan.sys	329130	NPcore Ltd
ECATDriver.sys	329120	EMC
pfkrnl.sys	329110	FXSEC LTD
epicFilter.sys	329100	Hidden Reflex
b9kernel.sys	329050	Bit9 Inc
eeCtrl.sys	329010	symantec
eraser.sys (Retired)	329010	symantec
SRTSP.sys,	329000	symantec
SRTSPIT.sys - ia64 systems	329000	symantec
SRTSP64.SYS - x64 systems	329000	symantec
a2ertpx86.sys	328920	Emsi Software GmbH
a2ertpx64.sys	328920	Emsi Software GmbH
a2gffx86.sys - x86	328910	Emsi Software GmbH
a2gffx64.sys - x64	328910	Emsi Software GmbH
a2gffi64.sys - IA64	328910	Emsi Software GmbH
a2acc.sys	328900	Emsi Software GmbH
a2acc64.sys on x64 systems	328900	Emsi Software GmbH
si32_file.sys	328810	Scargo Inc
si64_file.sys	328810	Scargo Inc
mbam.sys	328800	Malwarebytes Corp.
KUBWKSP.sys	328750	Netlor SAS
hcp_kernel_acq.sys	328740	refractionPOINT
SegiraAM.sys	328730	Segira LLC
wdocsafe.sys	328722	Cheetah Mobile Inc.
lbprotect.sys	328720	Cheetah Mobile Inc.

MINIFILTER	ALTITUDE	COMPANY
eamonm.sys	328700	ESET, spol. s r.o.
MaxProc64.sys	328620	Max Secure Software
MaxProtector.sys	328610	Max Secure Software
SDActMon.sys	328600	Max Secure Software
fileflt.sys	328540	Trend Micro Inc.
TmEsFlt.sys	328530	Trend Micro Inc.
tmevtmgr.sys	328510	Trend Micro Inc.
tmpreflt.sys	328500	Trend
vcMFilter.sys	328400	SGRI Co., LTD.
SAFsFilter.sys	328300	Lightspeed Systems
vsepflt.sys	328200	VMware
VFileFilter.sys(renamed)	328200	VMware
drivesentryfilterdriver2lite.sys	328100	DriveSentry Inc
WdFilter.sys	328010	Microsoft
mpFilter.sys	328000	Microsoft
vrSDetri.sys	327801	ETRI
vrSDetrix.sys	327800	ETRI
AhkSvPro.sys	327720	Ahkun Co.
AhkUsbFW.sys	327710	Ahkun Co.
AhkAMFlt.sys	327700	Ahkun Co.
PSINPROC.SYS	327620	Panda Security
PSINFILE.SYS	327610	Panda Security
amfsm.sys - Windows XP/2003 x64	327600	Panda Security
amm8660.sys - Windows Vista x86	327600	Panda Security
amm6460.sys - Windows Vista x64	327600	Panda Security

MINIFILTER	ALTITUDE	COMPANY
ADSpiderDoc.sys	327550	Digitalonnet
BkavSdFlt.sys	327540	Bkav Corporation
easyanticheat.sys	327530	EasyAntiCheat Solutions
5nine.cbt.sys	327520	5nine Software
caavFltr.sys	327510	Computer Assoc
ino_fltr.sys	327500	Computer Assoc
WCSDriver.sys	327410	White Cloud Security
360qpesv.sys	327404	360 Software (Beijing)
dsark.sys	327402	Qihoo 360
360avflt.sys	327400	Qihoo 360
ANVfsm.sys	327310	Arcdo
CDrRSFlt.sys	327300	Arcdo
EPSMn.sys	327200	SGA
VTSysFlt.sys	327150	Beijing Venus
TesMon.sys	327130	Tencent
QQSysMonX64.sys	327125	Tencent
QQSysMon.sys	327120	Tencent
TSysCare.sys	327110	Shenzhen Tencent Computer Systems Company Limited
TFsFlt.sys	327100	Shenzhen Tencent Computer Systems Company Limited
avmf.sys	327000	Authentium
BDFileDefend.sys	326916	Baidu (beijing)
BDsdKit.sys	326914	Baidu online network technology (beijing)Co.
bd0003.sys	326912	Baidu online network technology (beijing)Co.
Bfilter.sys	326910	Baidu (Hong Kong) Limited

MINIFILTER	ALTITUDE	COMPANY
NeoKerbyFilter	326900	NeoAutus
PLGFltr.sys	326800	Paretologic
WrdWizSecure64.sys	326730	WardWiz
wrdwizscanner.sys	326720	WardWiz
AshAvScan.sys	326700	Ashampoo GmbH & Co. KG
Zyfm.sys	326666	ZhengYong InfoTech LTD.
csaav.sys	326600	Cisco Systems
oavfm.sys	326550	HSM IT-Services Gmbh
SegMD.sys	326520	Segurmatica
SegMP.sys	326510	Segurmatica
SegF.sys	326500	Segurmatica
eeyehv.sys	326400	eEye Digital Security
eeyehv64.sys	326400	eEye Digital Security
CpAvFilter.sys	326311	CodeProof Technologies Inc
CpAvKernel.sys	326310	CodeProof Technologies Inc
NovaShield.sys	326300	Securitas Technologies,Inc.
SheedAntivirusFilterDriver.sys	326290	SheedSoft Ltd
bSyirmf.sys	326260	BLACKFORT SECURITY
bSymfdm.sys	326240	BLACKFORT SECURITY
bSyrtp.sys	326230	BLACKFORT SECURITY
bSyaed.sys	326220	BLACKFORT SECURITY
bSyar.sys	326210	BLACKFORT SECURITY
BdFileSpy.sys	326200	BullGuard
npxgd.sys	326160	INCA Internet Co.
npxgd64.sys	326160	INCA Internet Co.

MINIFILTER	ALTITUDE	COMPANY
tkpl2k.sys	326150	INCA Internet Co.
tkpl2k64.sys	326150	INCA Internet Co.
GKFF.sys	326140	INCA Internet Co.
GKFF64.sys	326140	INCA Internet Co.
tkdac2k.sys	326130	INCA Internet Co.
tkdacxp.sys	326130	INCA Internet Co.
tkdacxp64.sys	326130	INCA Internet Co.
tksp2k.sys	326120	INCA Internet Co.
tkspxp.sys	326120	INCA Internet Co.
tkspxp64.sys	326120	INCA Internet Co.
tkfsft.sys	326110	INCA Internet Co., Ltd
tkfsft64.sys - 64bit	326110	INCA Internet Co., Ltd
tkfsavxp.sys - 32bit	326100	INCA Internet Co., Ltd
tkfsavxp64.sys - 64bit	326100	INCA Internet Co., Ltd
SMDrvNt.sys	326040	AhnLab, Inc.
ATamptNt.sys	326030	AhnLab, Inc.
V3Flt2k.sys	326020	AhnLab, Inc.
V3MifiNt.sys	326010	Ahnlab
V3Ift2k.sys	326000	Ahnlab
V3IftmNt.sys (Old name)	326000	Ahnlab
ArfMonNt.sys	325990	Ahnlab
AhnRghLh.sys	325980	Ahnlab
AszFltNt.sys	325970	Ahnlab
OMFltLh.sys	325960	Ahnlab
V3Flu2k.sys	325950	Ahnlab

MINIFILTER	ALTITUDE	COMPANY
TfFregNt.sys	325940	AhnLab Inc.
vcdriv.sys	325820	Greatsoft Corp.Ltd
vcreg.sys	325810	Greatsoft Corp.Ltd
vchle.sys	325800	Greatsoft Corp.Ltd
NxFsMon.sys	325700	Novatix Corporation
AntiLeakFilter.sys	325600	Individual developer (Soft3304)
NanoAVMF.sys	325510	Panda Software
shldflt.sys	325500	Panda Software
nprosec.sys	325410	Norman ASA
nregsec.sys	325400	Norman ASA
issregistry.sys	325300	IBM
THFilter.sys	325200	Sybonic Systems Inc
pervac.sys	325100	PerSystems SA
avgmfx86.sys	325000	AVG Grisoft
avgmfx64.sys	325000	AVG Grisoft
avgmfi64.sys	325000	AVG Grisoft
avgmfrs.sys (retired)	325000	AVG Grisoft
FortiAptFilter.sys	324930	Fortinet Inc.
fortimon2.sys	324920	Fortinet Inc.
fortirmon.sys	324910	Fortinet Inc.
fortishield.sys	324900	Fortinet Inc.
mscan-rt.sys	324800	SecureBrain Corporation
sysdiag.sys	324600	Huorong Security
agentrtm64.sys	324510	WINS CO. LTD
rswmon.sys	324500	WINS CO. LTD

MINIFILTER	ALTITUDE	COMPANY
mwfsmfltr.sys	324420	MicroWorld Software Services Pvt. Ltd.
gtkdrv.sys	324410	GridinSoft LLC
GbpKm.sys	324400	GAS Tecnologia
crnsysm.sys	324310	Coranti
crncache32.sys	324300	Coranti
crncache64.sys	324300	Coranti
drwebfwft.sys	324210	Doctor Web
DwShield.sys	324200	Doctor Web
DwShield64.sys	324200	Doctor Web
IProtect.sys	324150	EveryZone
TvFiltr.sys	324140	EveryZone INC.
TvDriver.sys	324130	EveryZone INC.
TvSPFltr.sys	324120	EveryZone INC.
TvPtFile.sys	324110	EveryZone INC.
TvMFltr.sys	324100	Everyzone
SAVOnAccess.sys	324010	Sophos
savonaccess.sys	324000	Sophos
sld.sys	323990	Sophos
OADevice.sys	323900	Tall Emu
pwipf6.sys	323800	PWI, Inc.
EstRkmon.sys	323700	ESTsoft corp.
EstRkr.sys - 64bit	323700	ESTsoft corp.
dwprot.sys	323610	Doctor Web
Spiderg3.sys	323600	Doctor Web Ltd.
STKrnl64.sys	323500	Verdasys Inc

MINIFILTER	ALTITUDE	COMPANY
UFDFilter.sys	323400	Yoggie
SCFltr.sys	323300	SecurtiyCoverage, Inc.
fildds.sys	323200	Filseclab
fsfilter.sys	323100	MastedCode Ltd
fpav_rtp.sys	323000	f-protect
cwdriver.sys	322900	Leith Bade
AYFilter.sys	322810	ESTsoft
Rtw.sys	322800	ESTsoft
HookSys.sys	322700	Beijing Rising Information Technology Corporation Limited
snscore.sys	322600	S.N.Safe&Software
ssvhook.sys	322500	SecuLution GmbH
strapvista.sys	322400	AvSoft Technologies
strapvista64.sys	322400	AvSoft Technologies
sascan.sys	322300	SecureAge Technology
savant.sys	322200	Savant Protection, Inc.
VrBBDFlt.sys	322160	HAURI
vrSDfmx.sys	322153	HAURI
vrSDfmx.sys	322152	HAURI
vrSDam.sys	322151	HAURI
vrSDam.sys	322150	HAURI
VRAPTFLT.sys	322140	HAURI Inc.
VrAptDef.sys	322130	HAURI
VrSdCore.sys	322120	HAURI
VrFsFtM.sys	322110	HAURI
VrFsFtMX.sys(AMD64)	322110	HAURI

MINIFILTER	ALTITUDE	COMPANY
vradfil2.sys	322100	HAURI
fsgk.sys	322000	f-secure
bouncer.sys	321950	CoreTrace Corporation
PCTCore64.sys	321910	PC Tools Pty. Ltd.
PCTCore.sys (Old name)	321910	PC Tools Pty. Ltd.
ikfilesec.sys	321900	PC Tools Pty. Ltd.
ZxFsFilt.sys	321800	Australian Projects
antispyfilter.sys	321700	C-NetMedia Inc
PZDrvXP.sys	321600	VisionPower Co.,Ltd.
ggc.sys	321510	Quick Heal TechnologiesPvt. Ltd.
catflt.sys	321500	Quick Heal TechnologiesPvt. Ltd.
bdsflt.sys	321490	Quick Heal Technologies Pvt. Ltd.
arwflt.sys	321480	Quick Heal Technologies Pvt. Ltd.
csagent.sys	321410	CrowdStrike Ltd.
kmkuflt.sys	321400	Komoku Inc.
epdrv.sys	321320	McAfee Inc.
mfencoas.sys	321310	McAfee Inc.
mfehidk.sys	321300	McAfee Inc.
swin.sys	321250	McAfee Inc.
cmdccav.sys	321210	Comodo Group Inc.
cmdguard.sys	321200	Comodo Group Inc.
K7Sentry.sys	321100	K7 Computing Private Ltd.
nsminflt.sys	321050	NHN
nsminflt64.sys	321050	NHN
nvcmflt.sys	321000	Norman

MINIFILTER	ALTITUDE	COMPANY
dgsafe.sys	320950	KINGSOFT
issfltr.sys	320900	ISS
hbflt.sys	320840	BitDefender SRL
bdsvm.sys	320830	Bitdefender
gzflt.sys	320820	BitDefender SRL
bddevflt.sys	320812	BitDefender SRL
AVCKF.SYS	320810	BitDefender SRL
bdfsfltr.sys	320800	Softwin
bdfm.sys	320790	Softwin
Atc.sys	320781	BitDefender SRL
AVC3.SYS	320780	BitDefender SRL
TRUFOS.SYS	320770	BitDefender SRL
aswmonflt.sys	320700	Alwil
HookCentre.sys	320602	G Data
PktIcpt.sys	320601	G Data
MiniIcpt.sys	320600	G Data
avgntflt.sys	320500	Avira GmbH
klbg.sys	320440	Kaspersky
kldback.sys	320430	Kaspersky
kldlinf.sys	320420	Kaspersky
kldtool.sys	320410	Kaspersky
klif.sys	320401	Kaspersky Lab
klif.sys	320400	Kaspersky
avfsmn.sys	320310	Anvisoft
hssfwhl.sys	320330	Hitachi Solutions

MINIFILTER	ALTITUDE	COMPANY
DeepInsFS.sys	320320	Deep Instinct Ltd.
avfsmn.sys	320310	Anvisoft
lbd.sys	320300	Lavasoft AB
rvsmon.sys	320200	CJSC Returnil Software
ssfmonm.sys	320100	Webroot Software, Inc.
VirtualAgent.sys	320005	Symantec

300000 - 309998: FSFilter Replication

MINIFILTER	ALTITUDE	COMPANY
IntelCAS.sys	309100	Intel Corporation
mvfs.sys	309000	IBM Corporation
fsrecord.sys	305000	Microsoft
InstMon.sys	304201	Numecent Inc.
StreamingFSD.sys	304200	Numecent Inc.
ubcminifilterdriver.sys	304100	Ullmore Ltd.
replistor.sys	304000	Legato
stfsd.sys	303900	Endeavors Technologies
xomf.sys	303800	CA (XOSOFT)
nfid.sys	303700	Neverfail Group Ltd
sybfilter.sys	303600	Sybase, Inc.
rfsfilter.sys	303500	Evidian
cvmfsj.sys	303400	CommVault Systems, Inc.
iOraFilter.sys	303300	Infonic plc
bkbmfd32.sys (x86)	303200	BakBone Software, Inc
bkbmfd64.sys (x64)	303200	BakBone Software, Inc
mblvn.sys	303100	Packeteer

MINIFILTER	ALTITUDE	COMPANY
AV12NFNT.sys	303000	AhnLab
mDP_win_mini.sys	302900	Macro Impact
ctxubs.sys	302800	Citrix Systems
AxFilter.sys	301800	Axcient
vxfsrep.sys	301700	Symantec
dellcapfd.sys	301600	Dell
Sptres.sys	301500	Safend
OfficeBackup.sys	301400	Ushus Technologies
pcvnfilt.sys	301300	Blue Coat
repdac.sys	301200	NSI
repkap.sys	301100	NSI
repdrv.sys	301000	NSI

280000 - 289998: FSFilter Continuous Backup

MINIFILTER	ALTITUDE	COMPANY
Klcdp.sys	288900	Kaspersky Lab
splitinfmon.sys	288800	Split Infinity
versamatic.sys	288700	Acertant Tech
Yfilemon.sys	288690	Yarisoft
ibac.sys	288600	Idealstor, LLC.
fkdriver.sys	288500	Filekeeper
AAFileFilter.sys	288300	Dell
hyperoo.sys	288400	Hyperoo Ltd
HyperBacCA.sys	285000	Red Gate Software Ltd
ZMSFsFltr.sys	284400	Zenith InfoTech
aFsvDrv.sys	283100	ITSTATION Inc

MINIFILTER	ALTITUDE	COMPANY
AlfaSC.sys	284300	Alfa Corporation
hie_ifs.sys	284200	Hie Electronics, Inc.
AAFs.sys	284100	AppAssure Software
defilter.sys (old)	284000	Microsoft
tilana.sys	283000	Tilana Sys
VmDPFilter.sys	282900	Macro Impact
LbFilter.sys	281700	Linkverse S.r.l.
fbsfd.sys	281600	Ferro Software
dupleemf.sys	281500	Duplee SPI, S.L.
file_tracker.sys	281420	Acronis
exbackup.sys	281410	Acronis
afcdp.sys	281400	Acronis, Inc.
dcefltr.sys	281300	Cofio Software Ltd
ipmrsync_mfilter.sys	281200	OpenMars Enterprises
cascade.sys	281100	JP Software
filearchive.sys	281000	Code Mortem
syscdp.sys	280900	System OK AB
dpnedriver.sys (x86)	280850	HP
dpnedriver64.sys (x64)	280850	HP
hpchgflt.sys	280800	HP
VirtFile.sys	280700	Symantec
DeqoCPS.sys	280600	Deqo
LV_Tracker.sys	280500	LiveVault
cpbak.sys	280410	Checkpoint Software
tdmonxp.sys	280400	TimeData

MINIFILTER	ALTITUDE	COMPANY
nvfr_cpd	280310	Bakbone Software
nvfr_fdd	280300	Bakbone Software
Sptbkp.sys	280290	Safend

260000 - 269998: FSFilter Content Screener

MINIFILTER	ALTITUDE	COMPANY
Klshadow.sys	268300	Kaspersky Lab
isarsd.sys	268260	ISARS
zeoscanner.sys	268255	PCKeeper
fileHiders.sys	268250	PCKeeper
cbfltfs4-ObserveIT.sys	268240	ObserveIT
hipara.sys	268230	Allsum LLC
AliFileMonitorDriver.sys	268220	Alibaba
writeGuard.sys	268210	TCXA Ltd.
KKUDKProtectKer.sys	268200	Goldmessage technology co.
Atomizer.sys	268160	DragonFlyCodeWorks
farwflt.sys	268150	Malwarebytes
ADSpiderEx2.sys	268140	Digitalonnet
Safe.sys	268120	rian@alum.mit.edu
mydlpdelete-scanner.sys	268110	Medra Teknoloji
mydlpscanner.sys	268100	Medra Teknoloji
DLDriverNetMini.sys	268030	DeviceLock Inc
ENFFLTDRV.sys	268020	Enforcive Systems
crocopg.sys	268010	Infomaximum
sbapifs.sys	268000	Sunbelt Software
SGKD32.SYS	267910	NetSection Security

MINIFILTER	ALTITUDE	COMPANY
IccFilter.sys	267900	NEC System Technologies
tflbc.sys	267800	Tani Electronics Corporation
WBDrv.sys	266700	Axiana LLC
DMSamFilter.sys	266600	Digimarc Corp.
5nine.cbt.sys	266100	5nine Software
bsfs.sys	266000	Quick Heal TechnologiesPvt. Ltd.
XXRegSFilter.sys	265910	Zhe Jiang Xinxin Software Tech.
XXSFilter.sys	265900	Zhe Jiang Xinxin Software Tech.
AloahaUSBBlocker.sys	265800	Wrocklage Intermedia
frxdrv.sys	265700	FSLogix
FolderSecure.sys	265600	Max Secure Software
XendowFLTC.sys	265570	Credant Technologies
RepDac	265500	Vision Solutions
tbbdriver.sys	265400	Tedesi
spcgrd.sys	265300	FUJITSU BROAD SOLUTION
fdtlock.sys	265250	FUJITSU BROAD SOLUTION & CONSULTING Inc.
ssfFSC.sys	265200	SECUWARE S.L.
GagSecurity.sys	265120	Beijing Shu Yan Science
PrintDriver.sys	265110	Beijing Shu Yan Science
activ.sys	265100	Rapidware Pty Ltd
avscan.sys	265010	Microsoft
scanner.sys	265000	Microsoft
DI_fs.sys	264910	Soft-SB
wgnpos.sys	264900	Orchestria
odfltr.sys	264810	NetClean Technologies

MINIFILTER	ALTITUDE	COMPANY
ncpafltr.sys	264800	NetClean Technologies
ct.sys	264700	Haute Secure
fvefsmf.sys	264600	Fortisphere, Inc.
block.sys	264500	Autonomy Systems Limited
csascr.sys	264400	Cisco Systems
SymAFR.sys	264300	Symantec Corporation
cwnep.sys	264200	Websense Inc.
spywareremover.sys	264150	C-Netmedia
malwarebot.sys	264140	C-Netmedia
antispywarebot.sys	264130	2Squared Inc.
adwarebot.sys	264120	AntiSpyware LLC
antispyware.sys	264110	AntiSpyware LLC
spywarebot.sys	264100	C-Netmedia
nomp3.sys	264000	Hamish Speirs (private developer)
dlfilter.sys	263900	Starfield Software
sifsp.sys	263800	Secure Islands Technologies LTD
DLFsFlt.sys	263700	CenterTools Software GmbH
SamKeng.sys	263600	Syvik Co, Ltd.
rml.sys	263500	Logis IT Service Gmbh
vfsmfd.sys	263410	Vontu Inc.
vfsmfd.sys	263400	Vontu Inc.
acfilter.sys	263300	Avalere, Inc.
psecfilter.sys	263200	MDI Laboratory, Inc.
SolRedirect.sys	263110	Soliton Systems
solitkm.sys	263100	Soliton Systems

MINIFILTER	ALTITUDE	COMPANY
ipcfs.sys	263000	NetVeda
netgateav_access.sys	262910	NETGATE Tech. s.r.o.
spyemrg_access.sys	262900	NETGATE Tech. s.r.o.
pxrmcet.sys	262800	Proxure Inc.
EgisTecFF.sys	262700	Egis Technology Inc.
fgcpac.sys	262600	Fortres Grand Corp.
saappctl.sys	262510	SecureAge Technology
sadlp.sys	262500	SecureAge Technology
CRExecPrev.sys	262410	Cybereason
PEG2.sys	262400	PE GUARD
AdminRunFlt.sys	262300	Simon Jarvis
wvscr.sys	262200	Chengdu Wei Tech Inc.
psepfilter.sys	262100	Absolute Software
SAMDriver.sys	262000	Summit IT
wire_fsfilter.sys	261910	ThreatSpike Labs
AMFileSystemFilter.sys	261900	AppSense Ltd
mtflt.sys	261880	mTalos Inc.
nxrmflt.sys	261680	NextLabs, Inc.
hdlpflt.sys	261200	McAfee Inc.
CCFFilter.sys	261160	Microsoft
cbafilt.sys	261150	Microsoft
SmbBandwidthLimitFilter.sys	261110	Microsoft
DfsrRo.sys	261100	Microsoft
DataScrn.sys	261000	Microsoft
ldusbro.sys	260900	LANDesk Inc.

MINIFILTER	ALTITUDE	COMPANY
FileScreenFilter.sys	260800	Veritas
cpAcOnPnP.sys	260720	conpal GmbH
cpsgfsmf.sys	260710	conpal GmbH
psmmfilter.sys	260700	PolyServe
pctefa.sys	260650	PC Tools Pty. Ltd.
pctefa64.sys	260650	PC Tools Pty. Ltd.
symefasi.sys	260610	Symantec Corporation
symefa.sys	260600	Symantec
symefa64.sys	260600	Symantec
aictracedrv_cs.sys	260500	AI Consulting
DWFIxxxx.sys	260410	SciencePark Corporation
DWFIxxxx.sys	260400	SciencePark Corporation
FDriver.sys	260310	Fox-IT
iqpk.sys	260300	Secure Islands Technologies LTD
VHDFlt.sys	260240	Dell
VHDFlt.sys	260230	Dell
VHDFlt.sys	260220	Dell
VHDFlt.sys	260210	Dell

240000 - 249999: FSFilter Quota Management

MINIFILTER	ALTITUDE	COMPANY
ntps_qfs.sys	245100	NTP Software
PSSFsFilter.sys	245000	PSS Systems
Sptqmg.sys	245300	Safend
storqosflt.sys	244000	Microsoft

220000 - 229999: FSFilter System Recovery

MINIFILTER	ALTITUDE	COMPANY
file_protector.sys	227000	Acronis
fbwf.sys	226000	Microsoft
Klsysrec.sys	221500	Kaspersky Lab
SFDRV.SYS	221400	Utixo LLC
sp_prot.sys	221300	Xacti Corporation
nsfilep.sys	221200	Netsupport Limited
syscow.sys	221100	System OK AB
fsredir.sys	221000	Microsoft

200000 - 209999: FSFilter Cluster File System

MINIFILTER	ALTITUDE	COMPANY
CVCBT.sys	203400	CommVault Systems, Inc.
ResumeKeyFilter.sys	202000	Microsoft

180000 - 189999: FSFilter HSM

MINIFILTER	ALTITUDE	COMPANY
wcifs.sys	189900	Microsoft
gvflt.sys	189800	Microsoft
Svfsf.sys	186700	Spharsoft Technologies
gwmemory.sys	186600	Macrotec LLC
cteraflt.sys	186550	CTERA Networks Ltd.
dbx.sys	186500	Dropbox Inc.
quaddrasi.sys	186400	Quaddra Software
gdrive.sys	186300	Google
EaseTag.sys	186200	EaseVault Technologies Inc.
hcminifilter.sys	186100	Happy Cloud
PDFsFilter.sys	186000	Raxco Sfotware

MINIFILTER	ALTITUDE	COMPANY
camino.sys	185900	CaminoSoft Corp
C2C_AF1R.SYS	185810	C2C Systems
DFdriver.sys	185800	DataFirst Corporation
amfadrv.sys	185700	Quest Software Inc.
HSMdriver.sys	185600	Wim Vervoorn
kdfilter.sys	185555	Komprise Inc.
htdafd.sys	185500	Bridgehead Soft
SymHsm.sys	185400	Symantec
evmf.sys	185100	Symantec
otfilter.sys	185000	Overtone Soft
ithsmdrv.sys	184900	IBM
MfaFilter.sys	184800	Waterford Technologies
SonyHsmMinifilter.sys	184700	Sony Corporation
acahsm.sys	184600	Autonomy Corporation
zlhsm.sys	184500	ZL Technologies
Accesstracker.sys	183002	Microsoft
Changetracker.sys	183001	Microsoft
Fstier.sys	183000	Microsoft
hsmcdpflt.sys	182700	Metalogix
archivmgr.sys	182690	Metalogix
ntps_oddm.sys	182600	NTP Software
XDFileSys.sys	182500	XenData Limited
upmjit.sys	182400	Citrix Systems
AtmosFS.sys	182310	EMC Corporation
DxSpy.sys	182300	EMC Software Inc.

MINIFILTER	ALTITUDE	COMPANY
car_hsmflt.sys	182200	Caringo, Inc.
BRDriver.sys	182100	BitRaider
BRDriver64.sys	182100	BitRaider
autnhsm.sys	182000	Autonomy Corporation
cthsmflt.sys	181970	ComTrade
NxMini.sys	181900	NEXSAN
npfdaflt.sys	181800	Mimosa Systems Inc
AppStream.sys	181700	AppStream, Inc.
HPEDpHsmX64.sys	181620	Hewlett-Packard, Co.
HPArcHsmX64.sys	181610	Hewlett-Packard, Co.
hphsmflt.sys	181600	Hewlett-Packard, Co.
RepHsm.sys	181500	Double-Take Software, Inc.
RepSIS.sys	181490	Double-Take Software
SquashCompressionFsFilter.sys	181410	Squash Compression
GXHSM.sys	181400	Commvault Systems, Inc
EdsiHsm.sys	181300	Enterprise Data Solutions, Inc.
BkfMap.sys	181200	Data Storage Group
hsmfilter.sys	181100	GRAU Data Storage AG
mwi_dmflt.sys	181000	Moonwalk Univ
HcpAwfs.sys	181960	Hitachi Data Systems
sdrefltr.sys	180950	Hitachi Data Systems
fltasm.sys	180900	Global 360
cnet_hsm.sys	180850	Carroll-Net
pntvolflt.sys	180800	PoINT Software&Systems
appxstrm.sys	180710	Microsoft

MINIFILTER	ALTITUDE	COMPANY
wimmount.sys	180700	Microsoft
dfmflt.sys	180611	Microsoft
hsmflt.sys	180600	Microsoft
dfsrflt.sys	180500	Microsoft
odphflt.sys	180455	Microsoft
cldflt.sys	180451	Microsoft
dedup.sys	180450	Microsoft
dfmflt.sys	180410	Microsoft
sis.sys	180400	Microsoft
rbt_wfd.sys	180300	Riverbed Technology,Inc

170000 - 174999: *FSFilter Imaging (ex: .ZIP)

MINIFILTER	ALTITUDE	COMPANY
virtual_file.sys	172000	Acronis
wimFltr.sys	170500	Microsoft

160000 - 169999: FSFilter Compression

MINIFILTER	ALTITUDE	COMPANY
CmgFFC.sys	166000	Credant Technologies
compress.sys	165000	Microsoft
cmpflt.sys	162000	Microsoft
IridiumIO.sys	161700	Confio
logcompressor.sys	161600	VelociSQL
GcfFilter.sys	161500	GemacmbH
ssddoubler.sys	161400	Sinan Karaca
Sptcmp.sys	161300	Safend
wimfsf.sys	161000	Microsoft

MINIFILTER	ALTITUDE	COMPANY
GEFCMP.sys	160100	Symantec

140000 - 149999: FSFilter Encryption

MINIFILTER	ALTITUDE	COMPANY
EasyKryptMF.sys	149000	SoftKrypt LLC
padlock.sys	148910	IntSoft Inc.
ffecore.sys	148900	Winmagic
klvfs.sys	148810	Kaspersky Lab
Klfle.sys	148800	Kaspersky Lab
ISIRM.sys	148700	ALPS SYSTEM INTERGRATION CO.
ASUSSecDrive.sys	148650	ASUS
ABFilterDriver.sys	148640	AlertBoot
QDocumentFSF.sys	148630	BicDroid Inc.
bfusbenc.sys	148620	bitFence Inc.
sztgbfsf.sys	148610	SaferZone Co.
mwIPSDFilter.sys	148600	Egis Technology Inc.
csccvdrv.sys	148500	Computer Sciences Corporation
aefs.sys	148400	Angelltech Corporation Xi'an
IWCSEFlt.sys	148300	InfoWatch
GDDmk.sys	148250	G Data Software AG
clcxcore.sys	148210	AFORE Solutions Inc.
OrisLPDrv.sys	148200	CGS Publishing Tech
nlemsys.sys	148100	NETLIB
prvflder.sys	148000	Microsoft
ssefs.sys	147900	SecuLution GmbH
SePSed.sys	147800	Humming Heads, Inc.

MINIFILTER	ALTITUDE	COMPANY
dlmfencx.sys	147700	Data Encryption Ltd
psgcrypt.sys	147610	Yokogawa R&L Corp
bbfsflt.sys	147600	Bloombase
qx10efs.sys	147500	Quixxant
MEfefs.sys	147400	Eruces Inc.
medlpflt.sys	147310	Check Point Software Technologies Ltd
dsfa.sys	147308	Check Point Software Technologies Ltd
Snicrpt.sys	147300	Systemneeds, Inc
iCrypt.sys	147200	I-O DATA DEVICE, INC.
xdrmflt.sys	147100	bluefinsystems
dyFsFilter.sys	147000	Scrypto Media
thinairwin.sys	146960	Thin Air Inc"
UcaDataMgr.sys	146950	AppSense Ltd
zesocc.sys	146900	Novell
mfprom.sys	146800	McAfee Inc
MfeEEFF.sys	146790	McAfee
intefs.sys	146700	TianYu Software
leofs.sys	146600	Leotech
autocryptater.sys	146500	Richard Hagen
WavxDMgr.sys	146400	Scott Cochrane
eedmkxp32.sys	146300	Entrust
SbCe.sys	146200	SafeBoot
iSharedFsFilter	146100	Packeteer Inc
dlrmenc.sys	146010	DESlock
dlmfenc.sys	146000	DESlock+

MINIFILTER	ALTITUDE	COMPANY
aksdf.sys	145900	Aladdin Knowledge Systems
DDSFilter.sys	145800	WuHan Forworld Software
SecureShield.sys	145700	HMI
AifaFE.sys	145600	Alfa
GBFsMf.sys	145500	GreenBorder
jmefs.sys	145400	ShangHai Elec
emugufs.sys	145333	Emugu Secure FS
VFDriver.sys	145300	R Systems
EVSDecrypt64.sys	145230	Fortium Technologies Ltd
skycryptorencfs.sys	145220	Onecryptor CJSC.
AisLeg.sys	145210	Assured Information Security
windtalk.sys	145200	Hyland Software
TeamCryptor.sys	145190	iTwin Pte. Ltd.
CVDLP.sys	145180	CommVault Systems, Inc.
5nine.encryptor.sys	145170	5nine Software
ctpfile.sys	145160	Beijing Wondersoft Technology Co.
DPDrv.sys	145150	IBM Japan
tsdlp.sys	145140	Forware
KCDriver.sys	145130	Tallegra Ltd
CmgFFE.sys	145120	Credant Technologies
fgcenc.sys	145110	Fortres Grand Corp.
sview.sys	145100	Cinea
TalkeyFilterDriver.sys	145040	myTALKEY s.r.o.
FedsFilterDriver.sys	145010	Physical Optics Corp
stocc.sys	145000	Senforce Technologies

MINIFILTER	ALTITUDE	COMPANY
SnEfs.sys	144900	Informzaschita
ewSecureDox	144800	Echoworx Corporation
osrdmk.sys	144700	OSR Open Systems Resources, Inc.
uldcr.sys	144600	NCR Financial Solutions
Tkefsxp.sys - 32bit	144500	INCA Internet Co., Ltd
Tkefsxp64.sys - 64bit	144500	INCA Internet Co., Ltd
NmlAccf.sys	144400	NEC System Technologies, Ltd.
SolCrypt.sys	144300	Soliton Systems K.K.
IngDmk.sys	144200	Ingrian Networks, Inc.
llenc.sys	144100	SecureAxis Software
SecureData.sys	144030	SecureAge Technology
lockcube.sys	144020	SecureAge Technology Pte Ltd
sdmedia.sys	144010	SecureAge Technology
mysdrive.sys	144000	SecureAge Technology
FileArmor.sys	143900	Mobile Armor
VSTXEncr.sys	143800	VIA Technologies, Inc.
dgdmk.sys	143700	Verdasys Inc.
shandy.sys	143600	Safend Ltd.
C2knet.sys	143520	Secuware
C2kdef.sys	143510	Secuware
ssfFS.sys	143500	SECUWARE S.L.
PISRFE.sys	143400	Jilin University IT Co.
bapfecre.sys	143300	BitArmor Systems, Inc
KPSD.sys	143200	cihosoft
Fcfileio.sys	143100	Brainzsquare, Co. Ltd.

MINIFILTER	ALTITUDE	COMPANY
cpdrm.sys	143000	Pikewerks
vmfiltr.sys	142900	Vormetric Inc
VFSEnc.sys	142811	Symantec
pgpfs.sys	142810	Symantec
fencry.sys	142800	Symantec
TmFileEncDmk.sys	142700	Trend Micro Inc
cpefs.sys	142600	Crypto-Pro
dekfs.sys	142500	KasherLab co.,ltd
qlockfilter.sys	142400	Binqsoft Inc.
RRFilterDriverStack_d3.sys	142300	Rational Retention
cve.sys	142200	Absolute Software Corp.
spcflt.sys	142100	FUJITSU BSC Inc.
ldsecusb.sys	142000	LANDesk Inc.
fencr.sys	141900	SODATSW spol. s.r.o.
RubiFlt.sys	141800	Hitachi
CovertxFilter.sys	141700	Covertix
mfild.sys	141660	Penta Security Systems
TypeSquare.sys	141620	Morisawa inc.
xbdocfilter.sys	141610	Zrxb
EVSDecrypt32.sys	141600	Fortium Technologies Ltd
EVSDecrypt64.sys	141600	Fortium Technologies Ltd
EVSDecryptia64.sys	141600	Fortium Technologies Ltd
afdriver.sys	141500	ATUS Technology LLC
TrivalentFSFltr.sys	141430	Cyber Reliant
CmdMnEfs.sys	141420	Comodo Security

MINIFILTER	ALTITUDE	COMPANY
DWENxxxx.sys	141410	SciencePark Corporation
DWENxxxx.sys	141400	SciencePark Corporation
hdFileSentryDrv32.sys	141300	Heilig Defense
hdFileSentryDrv64.sys	141300	Heilig Defense
Filecrypt.sys	141100	Microsoft
encrypt.sys	141010	Microsoft
swapBuffers.sys	141000	Microsoft

130000 - 139999: FSFilter Virtualization

MINIFILTER	ALTITUDE	COMPANY
Klvirt.sys	138100	Kaspersky Lab
GetSAS.sys	138040	SAS Institute Inc
rqtNos.sys	138030	ReaQta Ltd.
HIPS64.sys	138020	Recrypt LLC
frxdrv.sys	138010	FSLogix
vzdrv.sys	138000	Altiris
sffsg.sys	137990	Starfish Storage Corp
AppStream.sys	137920	Symantec Corporation
boxifier.sys	137910	Kenubi
xorw.sys	137900	CA (XOsoft)
ctlua.sys	137800	SurfRight B.V.
fgccow.sys	137700	Fortres Grand Corp.
aswSnx.sys	137600	ALWIL Software
AppIsoFltr.sys	137500	Kernel Drivers
ptcvfsd.sys	137400	PTC
BDSandBox.sys	137300	BitDefender SRL

MINIFILTER	ALTITUDE	COMPANY
sxfpss-virt.sys	137200	Skanix AS
DKRtWrt.sys	137100	Diskeeper Corporation
ivm.sys	137000	RingCube Technologies
ivm.sys	136990	Citrix Systems
dtiof.sys	136900	Instavia Software Inc.
NxTopCP.sys	136800	Virtual Ccomputer Inc.
svdriver.sys	136700	VMware, Inc.
unifltr.sys	136600	Unidesk
unirsd.sys	136600	Unidesk
unidrive.sys (Renamed)	136600	Unidesk
ive.sys	136500	TrendMicro Inc.
odamf.sys	136450	Sony Corporation
SrMxfMf.sys	136440	Sony Corporation
pszmf.sys	136430	Sony Corporation
sxsudfmf.sys	136410	Sony Corporation
vfammf.sys	136400	Sony Corporation
VHDFlt.sys	136240	Dell
VHDFlt.sys	136230	Dell
VHDFlt.sys	136220	Dell
VHDFlt.sys	136210	Dell
ncfsfltr.sys	136200	NComputing
cmdguard.sys	136100	COMODO Security Solutions Inc
hpfsredir.sys	136000	HP
svhdxflt.sys	135100	Microsoft
luafv.sys	135000	Microsoft

MINIFILTER	ALTITUDE	COMPANY
ivm.sys	134000	RingCube Technologies
ivm.sys	133990	Citrix Systems
frxdrvvt.sys	132700	FSLogix Inc.
Stcvhdmf.sys	132600	StorageCraft Tech Corp
pfmfs_???.sys	132600	Pismo Technic Inc.
appdrv01.sys	132500	Protection Technology
virtual_file.sys	132400	Acronis
pdiFsFilter.sys	132300	Proximal Data Inc.
avgvtx86.sys	132200	AVG Technologies CZ
avgvtx64.sys	132200	AVG Technologies CZ
DataNet_Driver.sys	132100	AppSense Ltd
EgenPage.sys	132000	Egenera, Inc.
unidrive.sys-old	131900	Unidesk
ivm.sys.old	131800	RingCube Technologies
XiaobaiFsR.sys	131710	SHENZHEN UNNOO LTD
XiaobaiFs.sys	131700	SHENZHEN UNNOO LTD
iotfsflt.sys	131600	IO Turbine Inc
mhpvfs.sys	131500	Wunix Limited
svdriver.sys	131400	SnapVolumes
Sptvrt.sys	131300	Safend
aicvirt.sys	131200	AI Consulting
sfo.sys	130100	Microsoft
DeVolume.sys	130000	Microsoft

120000 - 129999: FSFilter Physical Quota Management

MINIFILTER	ALTITUDE	COMPANY
quota.sys	125000	Microsoft
qafilter.sys	124000	Veritas
DroboFlt.sys	123900	Data Robotics

100000 - 109999: FSFilter Filter Open File

MINIFILTER	ALTITUDE	COMPANY
insyncmf.sys	105000	InSync
SPILock8.sys	100900	Software Pursuits Inc.
Klbackupflt.sys	100800	Kaspersky
repkap	100700	Vision Solutions
symrg.sys	100600	Symantec
adsfilter.sys	100500	PolyServ

80000 - 89999: FSFilter Security Enhancer

MINIFILTER	ALTITUDE	COMPANY
dsbwnck.sys	88000	Easy Solution Inc.
rsbfsfilter.sys	87800	Corel Corporation
hsmltflt.sys	87720	Hitachi Solutions
hssfflt.sys	87710	Hitachi Solutions
acmnflt.sys	87708	Hitachi Solutions
ACSKFFD.sys	87700	Hitachi Solutions
MyDLPMF.sys	87600	Comodo Group Inc.
ScuaRaw.sys	87500	SCUA Segurança da Informação
HDSFilter.sys	87400	NeoAutus Automation System
ikfsmflt.sys	87300	IronKey Inc.
Klsec.sys	87200	Kaspersky Lab
XtimUSBFsFilterDrv.sys	87190	Dalian CP-SDT Ltd

MINIFILTER	ALTITUDE	COMPANY
RGFLT_FM.sys	87180	Hauri.inc
flockflt.sys	87170	Ahranta
ZdCore.sys	87160	Zends Technological Solutions
dcrypt.sys	87150	ReactOS Foundation
hpradeo.sys	87140	Pradeo
SDFSAGDRV.SYS	87130	ALPS SYSTEM INTERGRATION CO.
SDFSDEVFDRV.SYS	87120	ALPS SYSTEM INTERGRATION CO.
SDIFSFDRV.SYS	87110	ALPS SYSTEM INTERGRATION CO.
SDFSFDRV.SYS	87100	ALPS SYSTEM INTERGRATION CO.
HHRRPH.sys	87010	H+H Software GmbH
HHVolFltr.sys	87000	H+H Software GmbH
SbieDrv.sys	86900	Sandboxie L.T.D
assetpro.sys	86800	pyaprotect.com
dlp.sys	86700	Tellus Software AS
eps.sys	86600	Lumension Security
RapportPG64.sys	86500	Trusteer
amminifilter.sys	86400	AppSense
Sniflt.sys	86300	Systemneeds, Inc
SecFile.sys	86200	Secure By Design Inc.
philly.sys	86110	triCerat Inc.
reggy.sys	86100	triCerat Inc.
cygfilt.sys	86000	Livegrid Incorporated
prelaunch.sys	85900	D3L
csareg.sys	85810	Cisco Systems
csaenh.sys	85800	Cisco Systems

MINIFILTER	ALTITUDE	COMPANY
asEpsDrv.sys	85750	ASHINI Co. Ltd.
CIDACL.sys	85700	GE Aviation (Digital Systems Germantown)
CVDLP.sys	85610	CommVault Systems, Inc.
QGPEFlt.sys	85600	Quest Software
Drveng.sys	85500	CA
vracfil2.sys	85400	HAURI
TFsDisk.sys	85300	Teruten
rcMiniDrv.sys	85200	REDGATE CO.,LTD.
SnMc5xx.sys	85100	Informzaschita
FSPFltd.sys	85010	Alfa
AifaFFP.sys	85000	Alfa
EsAccCtlFE.sys	84901	EgoSecure GmbH
DpAccCtl.sys	84900	Softbroker GmbH
privman.sys	84800	BeyondTrust
eumntvol.sys	84700	Eugrid Inc
SoloEncFilter.sys	84600	Soliton Systems
sbfilter.sys	84500	UC4 Sofware
cposfw.sys	84450	Check Point Software Technologies Ltd.
vsdatant.sys	84400	Zone Labs LLC
SePnet.sys	84350	Humming Heads, Inc.
SePuld.sys	84340	Humming Heads, Inc.
SePpld.sys	84330	Humming Heads, Inc.
SePfsd.sys	84320	Humming Heads, Inc.
SePwld.sys	84310	Humming Heads, Inc.
SePprd.sys	84300	Humming Heads, Inc.

MINIFILTER	ALTITUDE	COMPANY
InPFlter.sys	84200	Humming Heads, Inc.
CProCtrl.sys	84100	Crypto-Pro
spyshelter.sys	84000	Datpol
clpinspprot.sys	83900	Information Technology Company Ltd.
uvmfsflt.sys	83376	NEC Corporation
dguard.sys	82300	Dmitry Varshavsky
NSUSBStorageFilter.sys	82200	NetSupport Ltd
RMSEFFMV.SYS	82100	CJSC Returnil Software
BoksFLAC.sys	82000	Fox Technologies
cpAcOnPnP.sys	81910	conpal GmbH
cpsgfsmf.sys	81900	conpal GmbH
ndevsec.sys	81800	Norman ASA
ViewIntus_RTDG.sys	81700	Pentego Technologies Ltd
airlock.sys	81630	Airlock Digital Pty Ltd
zam.sys	81620	
ANXfsm.sys	81610	Arcdo
CDrSDFlt.sys	81600	Arcdo
crnselfdefence32.sys	81500	Coranti
crnselfdefence64.sys	81500	Coranti
zlock_drv.sys	81400	SecurIT
f101fs.sys	81300	Fortres Grand Corp.
sysgar.sys	81200	Nucleus Data Recover
EmbargoM.sys	81100	ScriptLogic
ngssdef.sys	81050	Wontok
fsds2a.sys	81000	Splitstreem Ltd.

MINIFILTER	ALTITUDE	COMPANY
csacentr.sys	80900	Cisco Systems
ScvFLT50.sys	80850	Secuve Ltd
paritydriver.sys	80800	Bit9, Inc.
nkfsprot.sys	80710	Konneka
nkprot.sys	80700	KONNEKA Information Technologies
acpadlock.sys	80691	IntSoft Co
ksmf.sys	80690	IntSoft Co
im.sys	80680	CrowdStrike
SophosED.sys	80670	Sophos

60000 - 69999: FSFilter Copy Protection

MINIFILTER	ALTITUDE	COMPANY
CkProcess.sys	66100	KASHU SYSTEM DESIGN INC.
dlmfprot.sys	66000	Data Encrypt Sys
baprtsef.sys	65700	BitArmor Systems, Inc
sxfpss.sys	65600	Skanix AS
rgasdev.sys	65500	Macrovision
SkyFPDrv.sys	65410	Sky Co.,Ltd.
SkyWPDrv.sys	65400	Sky Co.,Ltd.
SnEraser.sys	65300	Informzaschita
vfilter.sys	65200	RSJ Software GmbH
COGOFlt32.sys	65100	Fortium Technologies Ltd
COGOFlt64.sys	65100	Fortium Technologies Ltd
COGOFLTia64.sys	65100	Fortium Technologies Ltd
scrubber.sys	65000	Microsoft
BRDriver.sys	64000	BitRaider LLC

MINIFILTER	ALTITUDE	COMPANY
BRDriver64.sys	64000	BitRaider LLC
LibertyFSF.sys	62300	Bayalink Solutions Co
axfsdrv2.sys	62100	Axence Software
sds.sys	62000	Egress Software
TotalSystemAuditor.sys	61600	ANRC LLC
MBAMApiary.sys	61500	Malwarebytes Corp.
WA_FSW.sys	61400	Programas Administracion y Mejoramiento
ViewIntus_RTAS	61300	Pentego Technologies
tffac.sys	61200	Toshiba Corporation
tccp.sys	61100	TrusCont Ltd
KomFS.sys	61000	KOM Networks

40000 - 49999: FSFilter Bottom

MINIFILTER	ALTITUDE	COMPANY
DLDriverMiniFlt.sys	47200	DeviceLock Inc
hsmltlib.sys	47110	Hitachi Solutions
hskdlib.sys	47100	Hitachi Solutions
acmnlib.sys	47090	Hitachi Solutions
aictracedrv_b.sys	47000	AI Consulting
hhdcfltr.sys	46900	Seagate Technology
Npsvctrig.sys	46000	Microsoft
fileinfo	45000	Microsoft
klvfs.sys	44900	Kaspersky Lab
rsfxdrv.sys	41000	Microsoft
defilter.sys	40900	Microsoft
AppVVemgr.sys	40800	Microsoft

MINIFILTER	ALTITUDE	COMPANY
wof.sys	40700	Microsoft

20000 - 29999: FSFilter System

None.

Minifilter Altitude Request

4/26/2017 • 1 min to read • Edit Online

A request for an minifilter altitude assignment is sent to Microsoft as an email. The body of the email must contain the following fields and corresponding information.

FIELD	COMMENT
Company name:	
Contact e-mail:	
Product name:	
Product URL:	
Product/Filter description:	A one paragraph summary to help Microsoft determine an appropriate altitude for the filter.
Filter filename:	
Filter type:	One of these values: Registry, FileSystem, Both
Filter start-type:	One of these values: Boot, System, Auto, Demand
Requested filter load order group:	See the File System Minifilter Allocated Altitudes for available load order groups.
Requested altitude:	Microsoft reserves the right to assign an altitude that is different from the requested altitude, depending on altitude availability and the filter driver functionality.
Additional information:	Use this field to let us know if there is any information you would like Microsoft to consider when assigning an altitude to this filter.

Send this information in an ASCII text e-mail message to fsfcomm@microsoft.com with the subject: “Minifilter altitude request”. An altitude for this filter will then be e-mailed back to the specified contact e-mail address.

The following is an example for the body of an allocation request email.

```
Hi,

Below is the request information to assign an altitude for our Contoso DataKleen file system minifilter.

Company name: Contoso Ltd.
Contact e-mail: filterdev@contoso.com
Product name: Contoso DataKleen
Product URL: http://fsfilters.contoso.com
Product/Filter Description:
    The Contoso DataKleen filter removes all occurences of any byte having a value
    between 128 and 255 during file reads. Our minifilter removes this value since
    it is not displayable on TTY devices.
Filter filename: ContosoDK.sys
Filter type: FileSystem
Filter start-type: Demand
Requested filter load order group: FSFilter Content Screener
Requested altitude: 268400
Additional information: None

Thanks,

FilterDev
```

Note

- All fields must be filled out.
- It may take Microsoft up to two weeks to process and assign altitudes. Any missing information may delay the assignment.
- The assigned altitude will eventually be reflected in the altitudes listed in File System Minifilter Allocated Altitudes. Please be aware that Microsoft only updates this list annually.

Reparse Point Tag Request

4/26/2017 • 1 min to read • Edit Online

This is the mechanism to obtain a Reparse Point tag, for those file system filter drivers that need one.

Reparse Point Tag Request

To obtain a Reparse Point tag, send the following information to Microsoft.

- Company name
- Company e-mail
- Company URL
- Contact e-mail
- Product name
- Product URL
- Product Description: (1 paragraph summary)
- Driver filename
- Driver device name
- Driver GUID
- High-latency bit enabled (yes/no)
- Name surrogate bit enabled (yes/no)

Send this information in an ASCII text e-mail message to rpid@microsoft.com. A *ReparseID* value for this driver will then be e-mailed back to the specified contact e-mail address.

The following list details some requirements for submitting a request.

- All fields must be filled out.
- For those who do not wish to use their own name and e-mail address in this publicly-viewable database, create an e-mail address for this use (for other companies which have products that include file system/filter drivers, which have interoperability issues with yours, and need to get the test/development teams of the two companies to communicate with each other). A suggested name is "*ntifskit@YourCompanyName.com*".
- If the "high latency" bit is enabled, this means the driver tags files and are expected to have a long latency. For example, this would be set by drivers drivers which use Reparse Points to implement heirarchical storage solutions, etc.
- If the "name surrogate" bit is enabled, this means the driver represents another named entity in the system. For example, the name of a volume mount point or of a directory junction.
- Reparse Points are a powerful feature of Windows, but developers should be aware that there can only be one reparse point per file, and some Windows mechanisms use reparse points (HSM, Native Structured Storage). Developers need to have fallback strategies for when the reparse point tag is already in use for a file.

Writing a DriverEntry Routine for a Minifilter Driver

4/26/2017 • 1 min to read • Edit Online

Every file system minifilter driver must have a **DriverEntry** routine. The **DriverEntry** routine is called when the minifilter driver is loaded.

The **DriverEntry** routine performs global initialization, registers the minifilter driver, and initiates filtering. This routine runs in a system thread context at IRQL PASSIVE_LEVEL.

The **DriverEntry** routine is defined as follows:

```
NTSTATUS
(*PDRIVER_INITIALIZE) (
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    );
```

DriverEntry has two input parameters. The first, *DriverObject*, is the driver object that was created when the minifilter driver was loaded. The second, *RegistryPath*, is a pointer to a counted Unicode string that contains a path to the minifilter driver's registry key.

A minifilter driver's **DriverEntry** routine must perform the following steps, in order:

1. Perform any needed global initialization for the minifilter driver.

2. Register the minifilter driver by calling **FltRegisterFilter**.

3. Initiate filtering by calling **FltStartFiltering**.

4. Return an appropriate NTSTATUS value.

This section includes:

Registering the Minifilter Driver

Initiating Filtering

Returning Status from a Minifilter DriverEntry Routine

Registering the Minifilter Driver

4/26/2017 • 1 min to read • Edit Online

Every minifilter driver must call **FltRegisterFilter** from its **DriverEntry** routine to add itself to the global list of registered minifilter drivers and to provide the filter manager with a list of callback routines and other information about the driver.

In the MiniSpy sample, the minifilter driver is registered as shown in the following code example:

```
NTSTATUS status;
status = FltRegisterFilter(
           DriverObject,                  //Driver
           &FilterRegistration,           //Registration
           &MiniSpyData.FilterHandle);    //RetFilter
```

FltRegisterFilter has two input parameters. The first, *Driver*, is the driver object pointer that the minifilter driver received as the *DriverObject* input parameter to its **DriverEntry** routine. The second, *Registration*, is a pointer to an **FLT_REGISTRATION** structure that contains entry points to the minifilter driver's callback routines.

In addition, **FltRegisterFilter** has an output parameter, *RetFilter*, that receives an opaque filter pointer for the minifilter driver. This filter pointer is a required input parameter for many **Flt***Xxx* support routines, including **FltStartFiltering** and **FltUnregisterFilter**.

Initiating Filtering

4/26/2017 • 1 min to read • Edit Online

After calling **FltRegisterFilter**, a minifilter driver's **DriverEntry** routine typically calls **FltStartFiltering** to begin filtering I/O operations.

Every minifilter driver must call **FltStartFiltering** from its **DriverEntry** routine to notify the filter manager that the minifilter driver is ready to begin attaching to volumes and filtering I/O requests. After the minifilter driver calls **FltStartFiltering**, the filter manager treats the minifilter driver as a fully active minifilter driver, presenting it with I/O requests and notifications of volumes to attach to. The minifilter driver must be prepared to begin receiving these I/O requests and notifications even before **FltStartFiltering** returns.

In the MiniSpy sample driver, **FltStartFiltering** is called as shown in the following code example:

```
status = FltStartFiltering( MiniSpyData.FilterHandle );
if( !NT_SUCCESS( status )) {
  FltUnregisterFilter( MiniSpyData.FilterHandle );
}
```

If the call to **FltStartFiltering** does not return STATUS_SUCCESS, the minifilter driver must call **FltUnregisterFilter** to unregister itself.

Returning Status from a Minifilter DriverEntry Routine

4/26/2017 • 1 min to read • Edit Online

A minifilter driver's **DriverEntry** routine normally returns STATUS_SUCCESS. But if minifilter initialization fails, the **DriverEntry** routine should return an appropriate error NTSTATUS value.

If the **DriverEntry** routine returns a status value that is not a success NTSTATUS value, the system responds by unloading the minifilter driver. The minifilter driver's **FilterUnloadCallback** routine is not called. For this reason, the **DriverEntry** routine must free any memory that was allocated for system resources before returning a status value that is not a success NTSTATUS value.

Writing a FilterUnloadCallback Routine for a Minifilter Driver

4/26/2017 • 1 min to read • Edit Online

A file system minifilter driver can optionally register a **PFLT_FILTER_UNLOAD_CALLBACK**-typed routine as the minifilter driver's *FilterUnloadCallback* routine. This callback routine is also referred to as the minifilter driver's *unload routine*.

Minifilter drivers are not required to register a *FilterUnloadCallback* routine. However, we strongly recommend that a minifilter driver registers this callback routine, because if a minifilter driver does not register a *FilterUnloadCallback* routine, the driver cannot be unloaded.

To register this callback routine, the minifilter driver stores the address of a PFLT_FILTER_UNLOAD_CALLBACK-typed routine in the **FilterUnloadCallback** member of the **FLT_REGISTRATION** structure that the minifilter driver passes as a parameter to **FltRegisterFilter** in its **DriverEntry** routine.

This section includes:

When the FilterUnloadCallback Routine Is Called

Writing a FilterUnloadCallback Routine

When the FilterUnloadCallback Routine Is Called

4/26/2017 • 1 min to read • Edit Online

The filter manager calls a minifilter driver's **FilterUnloadCallback** routine before unloading the minifilter driver in one of the following ways:

- *Non-mandatory unload*. This type of unload occurs when a user-mode application has called **FilterUnload** or a kernel-mode driver has called **FltUnloadFilter**. It also occurs when you type **fltmc unload** at the command prompt.
- *Mandatory unload*. This type of unload occurs when you issue a service stop request by typing **sc stop** or **net stop** at the command prompt. (For more information about the **sc stop** and **net stop** commands, click **Help and Support** on the Start menu.) It also occurs when a user-mode application calls the Microsoft Win32 **ControlService** function, passing the SERVICE_CONTROL_STOP control code as the *dwControl* parameter. (For more information about Win32 service functions, see the Microsoft Windows SDK documentation.)

For a non-mandatory unload, if the minifilter driver's *FilterUnloadCallback* routine returns an error or warning NTSTATUS value, such as STATUS_FLT_DO_NOT_DETACH, the filter manager does not unload the minifilter driver.

For a mandatory unload, the filter manager unloads the minifilter driver after the minifilter driver's *FilterUnloadCallback* routine is called, even if the *FilterUnloadCallback* routine returns an error or warning NTSTATUS value, such as STATUS_FLT_DO_NOT_DETACH.

To disable mandatory unloading for a minifilter driver, the minifilter driver sets the FLTFL_REGISTRATION_DO_NOT_SUPPORT_SERVICE_STOP flag in the **Flags** member of the **FLT_REGISTRATION** structure that the minifilter driver passes as a parameter to **FltRegisterFilter** in its **DriverEntry** routine. When this flag is set, the filter manager normally processes non-mandatory unload requests. However, mandatory unload requests will fail. The filter manager does not call the minifilter driver's *FilterUnloadCallback* routine for failed unload requests.

Note that if a minifilter driver's **DriverEntry** routine returns a warning or error NTSTATUS value, the *FilterUnloadCallback* routine is not called; the filter manager simply unloads the minifilter driver.

The *FilterUnloadCallback* routine is not called at system shutdown time. A minifilter driver that must perform shutdown processing should register a preoperation callback routine for IRP_MJ_SHUTDOWN operations.

Writing a FilterUnloadCallback Routine

4/26/2017 • 1 min to read • Edit Online

The *FilterUnloadCallback* routine is defined as follows:

```
typedef NTSTATUS
(*PFLT_FILTER_UNLOAD_CALLBACK) (
    FLT_FILTER_UNLOAD_FLAGS Flags
    );
```

The *FilterUnloadCallback* routine has one input parameter, *Flags*, which can be **NULL** or FLTFL_FILTER_UNLOAD_MANDATORY. The filter manager sets this parameter to FLTFL_FILTER_UNLOAD_MANDATORY to indicate that the unload operation is mandatory. For more information about this parameter, see **PFLT_FILTER_UNLOAD_CALLBACK**.

A minifilter driver's *FilterUnloadCallback* routine must perform the following steps:

- Close any open kernel-mode communication server port handles.
- Call **FltUnregisterFilter** to unregister the minifilter driver.
- Perform any needed global cleanup.
- Return an appropriate NTSTATUS value.

This section includes:

Closing the Communication Server Port

Unregistering the Minifilter

Performing Global Cleanup

Returning Status from a FilterUnloadCallback Routine

Closing the Communication Server Port

4/26/2017 • 1 min to read • Edit Online

If the minifilter driver previously opened a kernel-mode communication server port by calling **FltCreateCommunicationPort**, it must close the port by calling **FltCloseCommunicationPort**. To prevent the system from hanging during the unload process, the minifilter driver's **FilterUnloadCallback** routine must close this port before calling **FltUnregisterFilter**.

If a user-mode application has an open connection to the communication server port, any client port for that connection will remain open after **FltCloseCommunicationPort** returns. However, the filter manager will close any client ports when the minifilter driver is unloaded.

Unregistering the Minifilter

4/26/2017 • 1 min to read • Edit Online

A minifilter driver's **FilterUnloadCallback** routine must call **FltUnregisterFilter** to unregister the minifilter driver. Calling **FltUnregisterFilter** causes the following things to happen:

- The minifilter driver's callback routines are unregistered.
- The minifilter driver's instances are torn down, and the minifilter driver's **InstanceTeardownStartCallback** and **InstanceTeardownCompleteCallback** routines are called for each minifilter driver instance.
- If the minifilter driver set any contexts on volumes, instances, streams, or stream handles, these contexts are deleted. If the minifilter driver has registered a **CleanupContext** callback routine for a given context type, the filter manager calls the *CleanupContext* routine before deleting the context.

If there are outstanding rundown references on the minifilter driver's opaque filter pointer, **FltUnregisterFilter** enters a wait state until they are removed. Outstanding rundown references usually happen because the minifilter driver has called **FltQueueGenericWorkItem** to insert a work item into a system work queue, and the work item has not yet been dequeued and processed. (The filter manager adds the rundown reference when the minifilter driver calls **FltQueueGenericWorkItem** and removes it when the minifilter driver's work routine returns.)

Outstanding rundown references can also happen if the minifilter driver has called any routines that add a rundown reference to the minifilter driver's opaque filter pointer, such as **FltObjectReference** or **FltGetFilterFromInstance**, but did not subsequently call **FltObjectDereference**.

Performing Global Cleanup

4/26/2017 • 1 min to read • Edit Online

A minifilter driver's **FilterUnloadCallback** routine must perform any needed global cleanup. The following list includes examples of global cleanup tasks that a minifilter driver might perform:

- Call **ExDeleteResourceLite** to delete a global resource variable that was initialized by a previous call to **ExInitializeResourceLite**.
- Call **ExFreePool** or **ExFreePoolWithTag** to free global memory that was allocated by a previous call to a routine such as **ExAllocatePoolWithTag**.
- Call **ExDeleteNPagedLookasideList** or **ExDeletePagedLookasideList** to delete a lookaside list that was allocated by a previous call to **ExInitializeNPagedLookasideList** or **ExInitializePagedLookasideList**, respectively.
- Call **PsRemoveCreateThreadNotifyRoutine** or **PsRemoveLoadImageNotifyRoutine** to unregister a global callback routine that was registered by a previous call to **PsSetCreateThreadNotifyRoutine** or **PsSetLoadImageNotifyRoutine**, respectively.

Returning Status from a FilterUnloadCallback Routine

4/26/2017 • 1 min to read • Edit Online

A minifilter driver's **FilterUnloadCallback** routine normally returns STATUS_SUCCESS.

To refuse an unload operation that is not mandatory, the minifilter driver should return an appropriate warning or error NTSTATUS value such as STATUS_FLT_DO_NOT_DETACH. For more information about mandatory unload operations, see Writing a FilterUnloadCallback Routine and **PFLT_FILTER_UNLOAD_CALLBACK**.

If the *FilterUnloadCallback* routine returns a warning or error NTSTATUS value and the unload operation is not mandatory, the minifilter driver will not be unloaded.

Writing Preoperation and Postoperation Callback Routines

4/26/2017 • 2 min to read • Edit Online

In its **DriverEntry** routine, a minifilter driver can register up to one **preoperation callback routine** and up to one **postoperation callback routine** for each type of I/O operation that it needs to filter.

Unlike a legacy file system filter driver, a minifilter driver can choose which types of I/O operations to filter. A minifilter driver can register a preoperation callback routine for a given type of I/O operation without registering a postoperation callback, and vice versa. The minifilter driver receives only those I/O operations for which it has registered a preoperation or postoperation callback routine.

A *preoperation callback routine* is similar to a dispatch routine in the legacy filter driver model. When the filter manager processes an I/O operation, it calls the preoperation callback routine of each minifilter driver in the minifilter driver instance stack that has registered one for this type of I/O operation. The topmost minifilter driver in the stack--that is, the one whose instance has the highest altitude--receives the operation first. When that minifilter driver finishes processing the operation, it returns the operation to the filter manager, which then passes the operation to the next-highest minifilter driver, and so on. When all minifilter drivers in the minifilter driver instance stack have processed the I/O operation--unless a minifilter driver has completed the I/O operation--the filter manager sends the operation to legacy filters and the file system.

A *postoperation callback routine* is similar to a completion routine in the legacy filter driver model. Completion processing for an I/O operation begins when the I/O manager passes the operation to the file system and legacy filters that have registered completion routines for the operation. After these completion routines have finished, the filter manager performs completion processing for the operation. The filter manager then calls the postoperation callback routine of each minifilter driver in the minifilter driver instance stack that has registered one for this type of I/O operation. The bottom minifilter driver in the stack--that is, the one whose instance has the lowest altitude--receives the operation first. When that minifilter driver finishes processing the operation, it returns it to the filter manager, which then passes the operation to the next-lowest minifilter driver, and so on.

This section includes:

Registering Preoperation and Postoperation Callback Routines

Filtering I/O Operations in a Minifilter Driver

Writing Preoperation Callback Routines

Writing Postoperation Callback Routines

Modifying the Parameters for an I/O Operation

Determining the Buffering Method for an I/O Operation

Accessing the User Buffers for an I/O Operation

Registering Preoperation and Postoperation Callback Routines

4/26/2017 • 1 min to read • Edit Online

To register **preoperation callback routines** and **postoperation callback routines**, a minifilter driver makes a single call to **FltRegisterFilter** in its **DriverEntry** routine. For the *Registration* parameter in **FltRegisterFilter**, the minifilter driver passes a pointer to an **FLT_REGISTRATION** structure. The **OperationRegistration** member of this structure contains a pointer to an array of **FLT_OPERATION_REGISTRATION** structures, one for each type of I/O operation that the minifilter driver must filter.

Each FLT_OPERATION_REGISTRATION structure in the array, except for the last one, contains the following information:

- The major function code for the operation
- For read and write operations (IRP_MJ_READ and IRP_MJ_WRITE), a set of flags that specify whether to ignore cached I/O or paging I/O or both for IRP-based I/O operations
- Entry points for up to one preoperation callback routine and one postoperation callback routine

The last element in the array must be {IRP_MJ_OPERATION_END}.

The following code example, which is taken from the Scanner sample minifilter driver, shows an array of FLT_OPERATION_REGISTRATION structures. The Scanner sample minifilter driver registers preoperation and postoperation callback routines for IRP_MJ_CREATE and preoperation callback routines for IRP_MJ_CLEANUP and IRP_MJ_WRITE operations.

```
const FLT_OPERATION_REGISTRATION Callbacks[] = {
    {IRP_MJ_CREATE,
     0,
     ScannerPreCreate,
     ScannerPostCreate},
    {IRP_MJ_CLEANUP,
     0,
     ScannerPreCleanup,
     NULL},
    {IRP_MJ_WRITE,
     0,
     ScannerPreWrite,
     NULL},
    {IRP_MJ_OPERATION_END}
};
```

Filtering I/O Operations in a Minifilter Driver

4/26/2017 • 1 min to read • Edit Online

The following list describes several guidelines for filtering specific types of I/O operations in a file system minifilter driver:

- The **preoperation callback routine** for IRP_MJ_CREATE cannot query or set contexts for files, streams, or stream handles, because, at pre-create time, the file or stream (if any) that is going to be created has not yet been determined.
- The **postoperation callback routine** for IRP_MJ_CLOSE cannot set or query contexts for files, streams, or stream handles, because the system-internal structures that those items are associated with are freed before the post-close routine is called.
- Minifilter drivers must never fail IRP_MJ_CLEANUP or IRP_MJ_CLOSE operations. These operations can be pended, returned to the filter manager, or completed with STATUS_SUCCESS. However, a preoperation callback routine must never fail these operations.
- Minifilter drivers cannot register a postoperation callback routine for IRP_MJ_SHUTDOWN.

Writing Preoperation Callback Routines

4/26/2017 • 2 min to read • Edit Online

A file system minifilter driver uses one or more *preoperation callback routines* to filter I/O operations. **Preoperation callback routines** are similar to the dispatch routines that are used in legacy file system filter drivers.

A minifilter driver registers a preoperation callback routine for a particular type of I/O operation by storing the callback routine's entry point in the **OperationRegistration** member of the **FLT_REGISTRATION** structure. The minifilter driver passes this member as a parameter to **FltRegisterFilter** in its **DriverEntry** routine.

Minifilter drivers receive only those types of I/O operations for which they have registered a preoperation or postoperation callback routine. A minifilter driver can register a preoperation callback routine for a given type of I/O operation without registering a **postoperation callback routine**, and vice versa.

The following table shows the preoperation callback routine implementation for a specific usage scenario and its return value.

USAGE SCENARIO	IMPLEMENTATION	VALUE RETURNED
The routine is not relevant for the operation and does not require the final status of the operation or it has no postoperation callback.	Pass the I/O operation through without calling the minifilter's postoperation callback on completion.	FLT_PREOP_SUCCESS_NO_CALLBACK
The routine requires the final status of the operation.	Pass the operation through, requiring the minifilter to call the postoperation callback routine.	FLT_PREOP_SUCCESS_WITH_CALLBACK
The minifilter must complete or continue processing this operation in the future.	Put the operation into a pending state. Use **FltCompletePendedPreOperation** to complete the operation later.	FLT_PREOP_PENDING
The postoperation processing must occur in the context of the same thread that the dispatch routine was called. This ensures consistent IRQL and maintains your local variable state.	Synchronize the operation with the postoperation.	FLT_PREOP_SYNCHRONIZE
The preoperation callback routine needs to complete the operation.	Stop processing for the operation and assign final NTSTATUS value.	FLT_PREOP_COMPLETE

Every preoperation callback routine is defined as follows:

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

Like a dispatch routine, a preoperation callback routine can be called at IRQL = PASSIVE_LEVEL or at IRQL =

APC_LEVEL. Typically it is called at IRQL = PASSIVE_LEVEL, in the context of the thread that originated the I/O request. For fast I/O and file system filter (FsFilter) operations, the preoperation callback routine is always called at IRQL = PASSIVE_LEVEL. However, for an IRP-based operation, a minifilter driver's preoperation callback routine can be called in the context of a system worker thread if a higher filter or minifilter driver pends the operation for processing by the worker thread.

Context objects cannot be retrieved in postoperation routines at IRQL > APC_LEVEL. Instead, either get the context object during a preoperation routine and pass it to the postoperation routine or perform postoperation processing at IRQL <= APC_LEVEL. For more information about contexts, see Managing Contexts.

When the filter manager calls a minifilter driver's preoperation callback routine for a given I/O operation, the minifilter driver temporarily controls the I/O operation. The minifilter driver retains this control until it does one of the following:

- Returns a status value other than FLT_PREOP_PENDING from the preoperation callback routine.
- Calls **FltCompletePendedPreOperation** from a work routine that has processed an operation that was pended in the preoperation callback routine.

This section includes:

Passing an I/O Operation Down the Minifilter Instance Stack

Completing an I/O Operation in a Preoperation Callback Routine

Disallowing a Fast I/O Operation in a Preoperation Callback Routine

Pending an I/O Operation in a Preoperation Callback Routine

Passing I/O Operations Down the Minifilter Driver Instance Stack

4/26/2017 • 1 min to read • Edit Online

When a minifilter driver's **preoperation callback routine** or work routine returns an I/O operation to the filter manager, the filter manager sends the operation to minifilter drivers below the current minifilter driver in the minifilter driver instance stack and to legacy filters and the file system for further processing.

A minifilter driver's preoperation callback routine returns an I/O operation to the filter manager for further processing by returning one of the following status values:

- FLT_PREOP_SUCCESS_NO_CALLBACK (all operation types)
- FLT_PREOP_SUCCESS_WITH_CALLBACK (all operation types)
- FLT_PREOP_SYNCHRONIZE (IRP-based I/O operations only)

Note Although FLT_PREOP_SYNCHRONIZE should be returned only for IRP-based I/O operations, you can return this status value for other operation types. If it is returned for an I/O operation that is not an IRP-based I/O operation, the filter manager treats this return value as if it were FLT_PREOP_SUCCESS_WITH_CALLBACK.

Alternatively, the work routine for an operation that was pended in a preoperation callback routine returns an I/O operation to the filter manager by passing one of the preceding status values in the *CallbackStatus* parameter when it calls **FltCompletePendedPreOperation** to resume processing for the pended I/O operation.

This section includes:

Returning FLT_PREOP_SUCCESS_WITH_CALLBACK

Returning FLT_PREOP_SUCCESS_NO_CALLBACK

Returning FLT_PREOP_SYNCHRONIZE

Returning FLT_PREOP_SUCCESS_WITH_CALLBACK

7/21/2017 • 1 min to read • Edit Online

If a minifilter driver's **preoperation callback routine** returns FLT_PREOP_SUCCESS_WITH_CALLBACK, the filter manager calls the minifilter driver's **postoperation callback routine** during I/O completion.

Note If the minifilter driver's preoperation callback routine returns FLT_PREOP_SUCCESS_WITH_CALLBACK but the minifilter driver has not registered a postoperation callback routine for the operation, the system asserts on a checked build.

If the minifilter driver's preoperation callback routine returns FLT_PREOP_SUCCESS_WITH_CALLBACK, it can return a non-NULL value in its *CompletionContext* output parameter. This parameter is an optional context pointer that is passed to the corresponding postoperation callback routine. The postoperation callback routine receives this pointer in its *CompletionContext* input parameter.

The FLT_PREOP_SUCCESS_WITH_CALLBACK status value can be returned for all types of I/O operations.

Returning FLT_PREOP_SUCCESS_NO_CALLBACK

7/21/2017 • 1 min to read • Edit Online

If a minifilter driver's **preoperation callback routine** returns FLT_PREOP_SUCCESS_NO_CALLBACK, the filter manager does not call the minifilter driver's **postoperation callback routine**, if one exists, during I/O completion.

If the minifilter driver's preoperation callback routine returns FLT_PREOP_SUCCESS_NO_CALLBACK, it must return **NULL** in its *CompletionContext* output parameter.

The FLT_PREOP_SUCCESS_NO_CALLBACK status value can be returned for all types of I/O operations.

Returning FLT_PREOP_SYNCHRONIZE

7/21/2017 • 1 min to read • Edit Online

If a minifilter driver's **preoperation callback routine** synchronizes an I/O operation by returning FLT_PREOP_SYNCHRONIZE, the filter manager calls the minifilter driver's **postoperation callback routine** during I/O completion.

The filter manager calls the minifilter driver's postoperation callback routine in the same thread context as the preoperation callback, at IRQL <= APC_LEVEL. (Note that this thread context is not necessarily the context of the originating thread.)

Note If the minifilter driver's preoperation callback routine returns FLT_PREOP_SYNCHRONIZE, but the minifilter driver has not registered a postoperation callback routine for the operation, the system asserts on a checked build.

If the minifilter driver's preoperation callback routine returns FLT_PREOP_SYNCHRONIZE, it can return a non-NULL value in its *CompletionContext* output parameter. This parameter is an optional context pointer that is passed to the corresponding postoperation callback routine. The postoperation callback routine receives this pointer in its *CompletionContext* input parameter.

A minifilter driver's preoperation callback routine should return FLT_PREOP_SYNCHRONIZE only for IRP-based I/O operations. However, this status value can be returned for other operation types. If it is returned for an I/O operation that is not an IRP-based I/O operation, the filter manager treats this return value as if it were FLT_PREOP_SUCCESS_WITH_CALLBACK. To determine whether an operation is an IRP-based I/O operation, use the **FLT_IS_IRP_OPERATION** macro.

Minifilter drivers should not return FLT_PREOP_SYNCHRONIZE for create operations, because these operations are already synchronized by the filter manager. If a minifilter driver has registered preoperation and postoperation callback routines for IRP_MJ_CREATE operations, the post-create callback routine is called at IRQL = PASSIVE_LEVEL, in the same thread context as the pre-create callback routine.

Minifilter drivers must never return FLT_PREOP_SYNCHRONIZE for asynchronous read or write operations. Doing so can severely degrade both minifilter driver and system performance and can even cause deadlocks if, for example, the modified page writer thread is blocked. Before returning FLT_PREOP_SYNCHRONIZE for an IRP-based read or write operation, a minifilter driver should verify that the operation is synchronous by calling **FltIsOperationSynchronous**.

The following types of I/O operations cannot be synchronized:

- Oplock file system control (FSCTL) operations (**MajorFunction** is IRP_MJ_FILE_SYSTEM_CONTROL; **FsControlCode** is **FSCTL_REQUEST_FILTER_OPLOCK**, **FSCTL_REQUEST_BATCH_OPLOCK**, **FSCTL_REQUEST_OPLOCK_LEVEL_1**, or **FSCTL_REQUEST_OPLOCK_LEVEL_2**.)
- Notify change directory operations (**MajorFunction** is IRP_MJ_DIRECTORY_CONTROL; **MinorFunction** is IRP_MN_NOTIFY_CHANGE_DIRECTORY.)
- Byte-range lock requests (**MajorFunction** is IRP_MJ_LOCK_CONTROL; **MinorFunction** is IRP_MN_LOCK.)

FLT_PREOP_SYNCHRONIZE cannot be returned for any of these operations.

Completing an I/O Operation in a Preoperation Callback Routine

4/26/2017 • 1 min to read • Edit Online

To *complete* an I/O operation means to halt processing for the operation, assign it a final NTSTATUS value, and return it to the filter manager.

When a minifilter driver completes an I/O operation, the filter manager does the following:

- Does not send the operation to minifilter drivers below the current minifilter driver, to legacy filters, or to the file system.
- Calls the **postoperation callback routines** of the minifilter drivers above the current minifilter driver in the minifilter driver instance stack.
- Does not call the current minifilter driver's postoperation callback routine for the operation, if one exists.

A minifilter driver's **preoperation callback routine** completes an I/O operation by performing the following steps:

1. Setting the callback data structure's **IoStatus.Status** field to the final NTSTATUS value for the operation.
2. Returning FLT_PREOP_COMPLETE.

A preoperation callback routine that completes an I/O operation cannot set a non-NULL completion context (in the *CompletionContext* output parameter).

A minifilter driver can also complete an operation in the work routine for a previously pended I/O operation by performing the following steps:

1. Setting the callback data structure's **IoStatus.Status** field to the final NTSTATUS value for the operation.
2. Passing FLT_PREOP_COMPLETE in the *CallbackStatus* parameter when the work routine calls **FltCompletePendedPreOperation**.

When completing an I/O operation, a minifilter driver must set the callback data structure's **IoStatus.Status** field to the final NTSTATUS value for the operation, but this NTSTATUS value cannot be STATUS_PENDING or STATUS_FLT_DISALLOW_FAST_IO. For a cleanup or close operation, the field must be STATUS_SUCCESS. These operations cannot be completed with any other NTSTATUS value.

Completing an I/O operation is often referred to as succeeding or failing the operation, depending on the NTSTATUS value:

- To *succeed* an I/O operation means to complete it with a success or informational NTSTATUS value, such as STATUS_SUCCESS.
- To *fail* an I/O operation means to complete it with an error or warning NTSTATUS value, such as STATUS_INVALID_DEVICE_REQUEST or STATUS_BUFFER_OVERFLOW.

NTSTATUS values are defined in ntstatus.h. These values fall into four categories: success, informational, warning, and error. For more information about these values, see Using NTSTATUS Values.

Disallow a Fast I/O Operation in a Preoperation Callback Routine

4/26/2017 • 1 min to read • Edit Online

In certain circumstances, a minifilter driver might choose to disallow a fast I/O operation instead of completing it. Disallowing a fast I/O operation prevents the fast I/O path from being used for the operation.

Like completing an I/O operation, disallowing a fast I/O operation means to halt processing on it and return it to the filter manager. However, disallowing a fast I/O operation is different from completing it. If a minifilter driver disallows a fast I/O operation that was issued by the I/O manager, the I/O manager may reissue the same operation as an equivalent IRP-based operation.

When a minifilter driver's **preoperation callback routine** disallows a fast I/O operation, the filter manager does the following:

- Does not send the operation to minifilter drivers below the current minifilter driver, to legacy filters, or to the file system.
- Calls the **postoperation callback routines** of the minifilter drivers above the current minifilter driver in the minifilter driver instance stack.
- Does not call the current minifilter driver's postoperation callback routine for the operation, if one exists.

A minifilter driver disallows a fast I/O operation by returning FLT_PREOP_DISALLOW_FASTIO from the preoperation callback routine for the operation.

The preoperation callback routine should not set the callback data structure's **IoStatus.Status** field, because the filter manager automatically sets this field to STATUS_FLT_DISALLOW_FAST_IO.

FLT_PREOP_DISALLOW_FASTIO can only be returned for fast I/O operations. To determine whether an operation is a fast I/O operation, see **FLT_IS_FASTIO_OPERATION**.

Minifilter drivers cannot return FLT_PREOP_DISALLOW_FASTIO for IRP_MJ_SHUTDOWN, IRP_MJ_VOLUME_MOUNT, or IRP_MJ_VOLUME_DISMOUNT operations.

Pending an I/O Operation in a Preoperation Callback Routine

4/26/2017 • 1 min to read • Edit Online

A minifilter driver's **preoperation callback routine** can pend an I/O operation by posting the operation to a system work queue and returning FLT_PREOP_PENDING. Returning this status value indicates that the minifilter driver is retaining control of the I/O operation until it calls **FltCompletePendedPreOperation** to resume processing for the I/O operation.

A minifilter driver's preoperation callback routine pends an I/O operation by performing the following steps:

1. Posting the I/O operation to a system work queue by calling a routine such as **FltQueueDeferredIoWorkItem**.

2. Returning FLT_PREOP_PENDING.

A minifilter driver that must pend all (or most) incoming I/O operations should not use routines such as **FltQueueDeferredIoWorkItem** to pend operations, because calling this routine can cause the system work queues to be flooded. Instead, such a minifilter driver should use a cancel-safe queue. For more information about using cancel-safe queues, see *FltCbdqInitialize*.

Note that the call to **FltQueueDeferredIoWorkItem** will fail if any of the following conditions are true:

- The operation is not an IRP-based I/O operation.

- The operation is a paging I/O operation.

- The **TopLevelIrp** field of the current thread is not **NULL**. (For more information about how to find the value of this field, see **IoGetTopLevelIrp**.)

- The target instance for the I/O operation is being torn down.

If the minifilter driver's preoperation callback routine returns FLT_PREOP_PENDING, it must return **NULL** in the *CompletionContext* output parameter.

A minifilter driver can return FLT_PREOP_PENDING only for IRP-based I/O operations. To determine whether an operation is an IRP-based I/O operation, use the **FLT_IS_IRP_OPERATION** macro.

The work routine that dequeues and processes the I/O operation must call **FltCompletePendedPreOperation** to resume processing for the operation.

Writing Postoperation Callback Routines

4/26/2017 • 2 min to read • Edit Online

A file system minifilter driver uses one or more *postoperation callback routines* to filter I/O operations.

A postoperation callback routine can take one of the following actions:

- Accomplish completion work directly in postoperation routine. All the completion work can be accomplished at IRQL <= DISPATCH_LEVEL.
- Accomplish completion work at a safe IRQL. Return FLT_STATUS_MORE_PROCESSING_REQUIRED and queue a worker thread to allow processing at safe IRQL. When processing is complete, the worker thread calls **FltCompletePendedPostOperation** to continue postoperation processing.
- Cancel a successful CREATE operation.

Postoperation callback routines are similar to the *completion routines* that are used in legacy file system filter drivers.

A minifilter driver registers a postoperation callback routine for a particular type of I/O operation in the same way it registers a **preoperation callback routine**—that is, by storing the callback routine's entry point in the **OperationRegistration** member of the **FLT_REGISTRATION** structure that the minifilter driver passes as a parameter to **FltRegisterFilter** in its **DriverEntry** routine.

Minifilter drivers receive only those types of I/O operations for which they have registered a preoperation or postoperation callback routine. A minifilter driver can register a preoperation callback routine for a given type of I/O operation without registering a postoperation callback, and vice versa.

Every postoperation callback routine is defined as follows:

```
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
    );
```

Like a completion routine, a postoperation callback routine is called at IRQL <= DISPATCH_LEVEL, in an arbitrary thread context.

Because it can be called at IRQL = DISPATCH_LEVEL, a postoperation callback routine cannot call kernel-mode routines that must be called at a lower IRQL, such as **FltLockUserBuffer** or **RtlCompareUnicodeString**. For the same reason, any data structures that are used in a postoperation callback routine must be allocated from nonpaged pool.

The following situations are several exceptions to the preceding rule:

- If a minifilter driver's preoperation callback routine returns FLT_PREOP_SYNCHRONIZE for an IRP-based I/O operation, the corresponding postoperation callback routine is called at IRQL <= APC_LEVEL, in the same thread context as the preoperation callback routine.

- The postoperation callback routine for a fast I/O operation is called at IRQL = PASSIVE_LEVEL, in the same thread context as the preoperation callback routine.

- Post-create callback routines are called at IRQL = PASSIVE_LEVEL, in the context of the thread that originated the IRP_MJ_CREATE operation.

When the filter manager calls a minifilter driver's postoperation callback routine for a given I/O operation, the minifilter driver temporarily controls the I/O operation. The minifilter driver retains this control until it does one of the following:

- Returns FLT_POSTOP_FINISHED_PROCESSING from the postoperation callback routine.
- Calls **FltCompletePendedPostOperation** from a work routine that has processed an IRP-based I/O operation that was pended in the postoperation callback routine.

This section includes:

Performing Completion Processing for an I/O Operation

Pending an I/O Operation in a Postoperation Callback Routine

Failing an I/O Operation in a Postoperation Callback Routine

Performing Completion Processing for an I/O Operation

4/26/2017 • 1 min to read • Edit Online

A minifilter driver's **postoperation callback routine** is called when an I/O operation has been completed by the underlying file system, by a legacy filter, or by another minifilter driver that is at a lower altitude in the minifilter driver instance stack.

In addition, when a minifilter driver instance is being torn down, the filter manager "drains" any I/O operations for which the instance has received a **preoperation callback** and is awaiting a **postoperation callback**. In this situation, the filter manager calls the minifilter driver's postoperation callback routine, even if the I/O operation has not been completed, and sets the FLTFL_POST_OPERATION_DRAINING flag in the *Flags* input parameter.

When the FLTFL_POST_OPERATION_DRAINING flag is set, the minifilter driver must not perform normal completion processing. Instead, it should perform only necessary cleanup, such as freeing memory that the minifilter driver allocated for the *CompletionContext* parameter in its preoperation callback routine, and return FLT_POSTOP_FINISHED_PROCESSING.

This section includes the following topic:

Ensuring that Completion Processing is Performed at Safe IRQL

Ensuring that Completion Processing is Performed at Safe IRQL

4/26/2017 • 1 min to read • Edit Online

As noted in Writing Postoperation Callback Routines, the **postoperation callback routine** for an IRP-based I/O operation can be called at IRQL = DISPATCH_LEVEL, unless the minifilter driver's preoperation callback routine synchronized the operation by returning FLT_PREOP_SYNCHRONIZE or the operation is a create operation, which is inherently synchronous. (For more information about this return value, see Returning FLT_PREOP_SYNCHRONIZE.)

However, for IRP-based I/O operations that are not already synchronized, minifilter drivers can use to two techniques to ensure that completion processing is performed at IRQL <= APC_LEVEL.

The first technique is for the postoperation callback routine to pend the I/O operation until completion processing can be performed at IRQL <= APC_LEVEL. This technique is described in Pending an I/O Operation in a Postoperation Callback Routine.

The second technique is for the minifilter driver's postoperation callback routine to call **FltDoCompletionProcessingWhenSafe**. **FltDoCompletionProcessingWhenSafe** pends the I/O operation only if the current IRQL is >= DISPATCH_LEVEL. Otherwise, this routine executes the minifilter driver's **SafePostCallback** routine immediately. This technique is described in **FltDoCompletionProcessingWhenSafe**.

Pending an I/O Operation in a Postoperation Callback Routine

4/26/2017 • 1 min to read • Edit Online

A minifilter driver's **postoperation callback routine** can pend an I/O operation by performing the following steps:

1. Calling **FltAllocateDeferredIoWorkItem** to allocate a work item for the I/O operation.
2. Calling **FltQueueDeferredIoWorkItem** to post the I/O operation to a system work queue.
3. Returning FLT_POSTOP_MORE_PROCESSING_REQUIRED.

Note that the call to **FltQueueDeferredIoWorkItem** will fail if any of the following conditions are true:

- The operation is not an IRP-based I/O operation.
- The operation is a paging I/O operation.
- The **TopLevelIrp** field of the current thread is not **NULL**. (For more information about how to find the value of this field, see **IoGetTopLevelIrp**.)
- The target instance for the I/O operation is being torn down. (The filter manager indicates this situation by setting the FLTFL_POST_OPERATION_DRAINING flag in the *Flags* input parameter to the postoperation callback routine.)

Minifilter drivers must be prepared to handle this failure. If your minifilter driver cannot handle such failures, you should consider using the technique that is described in Returning FLT_PREOP_SYNCHRONIZE instead of pending the I/O operation.

After the minifilter driver's postoperation callback routine returns FLT_POSTOP_MORE_PROCESSING_REQUIRED, the filter manager will not perform any further completion processing for the I/O operation until the minifilter driver's work routine calls **FltCompletePendedPostOperation** to return control of the operation to the filter manager. The filter manager will not perform any further processing in this situation even if the work routine sets a failure NTSTATUS value in the **IoStatus.Status** field of the callback data structure for the operation.

The work routine that dequeues and performs completion processing for the I/O operation must call **FltCompletePendedPostOperation** to return control of the operation to the filter manager.

Failing an I/O Operation in a Postoperation Callback Routine

4/26/2017 • 1 min to read • Edit Online

A minifilter driver's **postoperation callback routine** can fail a successful I/O operation, but simply failing an I/O operation does not undo the effect of the operation. The minifilter driver is responsible for performing any processing that is needed to undo the operation.

For example, a minifilter driver's post-create callback routine can fail a successful IRP_MJ_CREATE operation by performing the following steps:

1. Calling **FltCancelFileOpen** to close the file that was created or opened by the create operation. Note that **FltCancelFileOpen** does not undo any modifications to the file. For example, **FltCancelFileOpen** does not delete a newly created file or restore a truncated file to its previous size.
2. Setting the callback data structure's **IoStatus.Status** field to the final NTSTATUS value for the operation. This value must be a valid error NTSTATUS value, such as STATUS_ACCESS_DENIED.
3. Setting the callback data structure's **IoStatus.Information** field to zero.
4. Returning FLT_POSTOP_FINISHED_PROCESSING.

When setting the callback data structure's **IoStatus.Status** field to the final NTSTATUS value for the operation, the minifilter driver must specify a valid error NTSTATUS value. Note that minifilter drivers cannot specify STATUS_FLT_DISALLOW_FAST_IO; only the filter manager can use this NTSTATUS value.

Callers of **FltCancelFileOpen** must be running at IRQL <= APC_LEVEL. However, a minifilter driver can safely call this routine from a post-create callback routine, because, for IRP_MJ_CREATE operations, the postoperation callback routine is called at IRQL = PASSIVE_LEVEL, in the context of the thread that originated the create operation.

Modifying the Parameters for an I/O Operation

4/26/2017 • 2 min to read • Edit Online

A minifilter driver can modify the parameters for an I/O operation. For example, a minifilter driver's **preoperation callback routine** can redirect an I/O operation to a different volume by changing the target instance for the operation. The new target instance must be an instance of the same minifilter driver at the same altitude on another volume.

The parameters for an I/O operation are found in the callback data (**FLT_CALLBACK_DATA**) structure and I/O parameter block (**FLT_IO_PARAMETER_BLOCK**) structure for the operation. The minifilter driver's **preoperation callback routine** and **postoperation callback routine** receive a pointer to the callback data structure for the operation in the *Data* input parameter. The *Iopb* member of the callback data structure is a pointer to an I/O parameter block structure that contains the parameters for the operation.

If a minifilter driver's preoperation callback routine modifies the parameters for an I/O operation, all minifilter drivers below that minifilter driver in the minifilter driver instance stack will receive the modified parameters in their preoperation and postoperation callback routines.

The modified parameters are not received by the current minifilter driver's postoperation callback routine or by any minifilter drivers above that minifilter driver in the minifilter driver instance stack. In all situations, a minifilter driver's preoperation and postoperation callback routines receive the same input parameter values for a given I/O operation.

After modifying the parameters for an I/O operation, the preoperation or postoperation callback routine must indicate that it has done so by calling **FltSetCallbackDataDirty**, unless it has changed the contents of the callback data structure's **IoStatus** field. Otherwise, the filter manager will ignore any changes to parameter values. **FltSetCallbackDataDirty** sets the FLTFL_CALLBACK_DATA_DIRTY flag in the callback data structure for the I/O operation. Minifilter drivers can test this flag by calling **FltIsCallbackDataDirty** or clear it by calling **FltClearCallbackDataDirty**.

If a minifilter driver's preoperation callback routine modifies the parameters for an I/O operation, all minifilter drivers below that minifilter driver in the minifilter driver instance stack will receive the modified parameters in the *Data* and *FltObjects* input parameters to their preoperation and postoperation callback routines. (Minifilter drivers cannot directly modify the contents of the **FLT_RELATED_OBJECTS** structure that is pointed to by the *FltObjects* parameter. However, if a minifilter driver modifies the target instance or target file object for an I/O operation, the filter manager modifies the value of the corresponding **Instance** or **FileObject** member of the FLT_RELATED_OBJECTS structure that is passed to lower minifilter drivers.)

Although any parameter changes that a minifilter driver's preoperation callback routine makes are not received by the minifilter driver's own postoperation callback routine, a preoperation callback routine is able to pass information about changed parameters to the minifilter driver's own postoperation callback routine. If the preoperation callback routine passes the I/O operation down the stack by returning FLT_PREOP_SUCCESS_WITH_CALLBACK or FLT_PREOP_SYNCHRONIZE, it can store information about changed parameter values into a minifilter driver-defined structure that is pointed to by the *CompletionContext* output parameter. The filter manager passes this structure pointer in the *CompletionContext* input parameter to the postoperation callback routine.

For more information about the parameters for an I/O operation, see **FLT_CALLBACK_DATA** and **FLT_IO_PARAMETER_BLOCK**.

Determining the Buffering Method for an I/O Operation

4/26/2017 • 1 min to read • Edit Online

Like device drivers, file systems are responsible for transferring data between user-mode applications and a system's devices. The operating system provides the following three methods for accessing data buffers:

- In *buffered I/O*, the I/O manager allocates a system buffer for the operation from nonpaged pool. The I/O manager copies data from this system buffer into the application's user buffer, and vice versa, in the context of the thread that initiated the I/O operation.
- In *direct I/O*, the I/O manager probes and locks the user buffer. It then creates a memory descriptor list (MDL) to map the locked buffer. The I/O manager accesses the buffer in the context of the thread that initiated the I/O operation.
- In *neither buffered nor direct I/O*, the I/O manager does not allocate a system buffer and does not lock or map the user buffer. Instead, it simply passes the buffer's original user-space virtual address to the file system stack. Drivers are responsible for ensuring that they are executing in the context of the initiating thread and that the buffer addresses are valid.

  Minifilter drivers must validate any address in user space before trying to use it. The I/O manager and filter manager do not validate such addresses and do not validate pointers that are embedded in buffers that are passed to minifilter drivers.

All standard Microsoft file systems use neither buffered nor direct I/O for most I/O processing.

For more information about buffering methods, see Methods for Accessing Data Buffers.

For IRP-based I/O operations, the buffering method used is operation-specific and is determined by the following factors:

- The type of I/O operation that is being performed
- The value of the **Flags** member of the **DEVICE_OBJECT** structure for the file system volume
- For I/O control (IOCTL) and file system control (FSCTL) operations, the value of the *TransferType* parameter that was passed to the CTL_CODE macro when the IOCTL or FSCTL was defined

Fast I/O operations that have buffers always use neither buffered nor direct I/O.

File system callback operations do not have buffers.

This section includes:

Operations That Can Be IRP-Based or Fast I/O

IRP-Based I/O Operations That Obey Device Object Flags

IRP-Based I/O Operations That Always Use Buffered I/O

IRP-Based I/O Operations That Always Use Neither Buffered Nor Direct I/O

IRP-Based IOCTL and FSCTL Operations

IRP-Based I/O Operations That Have No Buffers

Operations That Can Be IRP-Based or Fast I/O

4/26/2017 • 1 min to read • Edit Online

The following types of operations can be IRP-based or fast I/O operations:

- IRP_MJ_DEVICE_CONTROL. (Note that IRP_MJ_INTERNAL_DEVICE_CONTROL is always IRP-based.)
- IRP_MJ_QUERY_INFORMATION. This operation can be fast I/O if the **FileInformationClass** parameter is **FileBasicInformation**, **FileStandardInformation**, or **FileNetworkOpenInformation**.
- IRP_MJ_READ. Minifilter drivers can set the FLTFL_OPERATION_REGISTRATION_SKIP_CACHED_IO flag in the **FLT_OPERATION_REGISTRATION** structure to avoid receiving fast I/O IRP_MJ_READ operations and cached IRP-based reads.
- IRP_MJ_WRITE. Minifilter drivers can set the FLTFL_OPERATION_REGISTRATION_SKIP_CACHED_IO flag in the FLT_OPERATION_REGISTRATION structure to avoid receiving fast I/O IRP_MJ_WRITE operations and cached IRP-based writes.

When any of these operations is a fast I/O operation, it always uses neither buffered nor direct I/O, even if the equivalent IRP-based operation uses a different buffering method.

When IRP_MJ_DEVICE_CONTROL is a fast I/O operation, it always uses neither buffered nor direct I/O, regardless of the IOCTL's transfer type.

Although IRP_MJ_LOCK_CONTROL can be an IRP-based or fast I/O operation, it has no buffers.

IRP-Based I/O Operations That Obey Device Object Flags

4/26/2017 • 1 min to read • Edit Online

The buffering method for the following IRP-based I/O operations is determined by the value of the **Flags** member of the **DEVICE_OBJECT** structure for the file system volume:

- IRP_MJ_DIRECTORY_CONTROL
- IRP_MJ_QUERY_EA
- IRP_MJ_QUERY_QUOTA
- IRP_MJ_READ
- IRP_MJ_SET_EA
- IRP_MJ_SET_QUOTA
- IRP_MJ_WRITE

The DO_BUFFERED_IO and DO_DIRECT_IO flags in the **Flags** member are used as follows:

- If the DO_BUFFERED_IO flag is set, the operation uses buffered I/O.
- If the DO_DIRECT_IO flag is set and the DO_BUFFERED_IO flag is not set, the operation uses direct I/O.
- If neither flag is set, the operation uses neither buffered nor direct I/O.

For more information about device object flags, see **DEVICE_OBJECT** and Initializing a Device Object.

Note that IRP_MJ_READ and IRP_MJ_WRITE can be IRP-based or fast I/O operations. When they are IRP-based, the buffering method is determined by the device object flags as described above. When these operations are fast I/O, they always use neither buffered nor direct I/O. For more information about I/O operations that can be IRP-based or fast I/O operations, see Operations That Can Be IRP-Based or Fast I/O.

IRP-Based I/O Operations That Always Use Buffered I/O

4/26/2017 • 1 min to read • Edit Online

The following IRP-based I/O operations always use buffered I/O, regardless of the value of the **Flags** member of the **DEVICE_OBJECT** structure for the file system volume:

- IRP_MJ_CREATE (**EaBuffer parameter**)
- IRP_MJ_QUERY_INFORMATION
- IRP_MJ_QUERY_VOLUME_INFORMATION
- IRP_MJ_SET_INFORMATION
- IRP_MJ_SET_VOLUME_INFORMATION

Note that IRP_MJ_QUERY_INFORMATION can also be a fast I/O operation. When it is a fast I/O operation, it uses neither buffered nor direct I/O. For more information about I/O operations that can be IRP-based or fast I/O operations, see Operations That Can Be IRP-Based or Fast I/O.

IRP-Based I/O Operations That Always Use Neither Buffered Nor Direct I/O

4/26/2017 • 1 min to read • Edit Online

The following IRP-based I/O operations always use neither buffered nor direct I/O, regardless of the value of the **Flags** member of the **DEVICE_OBJECT** structure for the file system volume:

- IRP_MJ_PNP
- IRP_MJ_QUERY_SECURITY
- IRP_MJ_SET_SECURITY
- IRP_MJ_SYSTEM_CONTROL

IRP-Based IOCTL and FSCTL Operations

4/26/2017 • 1 min to read • Edit Online

The following IRP-based I/O operations use the buffering method that matches the transfer type that is specified in the definition of the I/O control code (IOCTL) or file system control code (FSCTL):

- IRP_MJ_DEVICE_CONTROL
- IRP_MJ_FILE_SYSTEM_CONTROL
- IRP_MJ_INTERNAL_DEVICE_CONTROL

The transfer type is specified in the *TransferType* parameter of the CTL_CODE macro. To obtain the transfer type for a given IOCTL or FSCTL, use the following macro:

```
#define METHOD_FROM_CTL_CODE(ctrlCode)         ((ULONG)(ctrlCode & 3))
```

This macro returns one of the following values:

```
#define METHOD_BUFFERED                 0
#define METHOD_IN_DIRECT                1
#define METHOD_OUT_DIRECT               2
#define METHOD_NEITHER                  3
```

For more information about the CTL_CODE macro, see Defining I/O Control Codes.

Note that IRP_MJ_DEVICE_CONTROL can also be a fast I/O operation. When it is a fast I/O operation, it always uses neither buffered nor direct I/O, regardless of the IOCTL's transfer type. For more information about when IRP_MJ_DEVICE_CONTROL can be a fast I/O operation, see Operations That Can Be IRP-Based or Fast I/O.

IRP-Based I/O Operations That Have No Buffers

4/26/2017 • 1 min to read • Edit Online

The following IRP-based I/O operations have no buffers and thus no buffering method:

- IRP_MJ_CREATE_MAILSLOT
- IRP_MJ_CREATE_NAMED_PIPE
- IRP_MJ_LOCK_CONTROL

Accessing the User Buffers for an I/O Operation

4/26/2017 • 1 min to read • Edit Online

The **FLT_PARAMETERS** structure for an I/O operation contains the operation-specific parameters for the operation, including buffer addresses and memory descriptor lists (MDL) for any buffers that are used in the operation.

For IRP-based I/O operations, the buffers for the operation can be specified by using:

- MDL only (typically for paging I/O)
- Buffer address only
- Buffer address and MDL

For fast I/O operations, only the user-space buffer address is specified. Fast I/O operations that have buffers always use neither buffered nor direct I/O and thus never have MDL parameters.

The following topics provide guidelines for handling buffer addresses and MDLs for IRP-based and fast I/O operations in minifilter driver **preoperation callback routines** and **postoperation callback routines**:

Accessing User Buffers in a Preoperation Callback Routine

Accessing User Buffers in a Postoperation Callback Routine

Accessing User Buffers in a Preoperation Callback Routine

4/26/2017 • 1 min to read • Edit Online

A minifilter driver's **preoperation callback routine** should treat a buffer in an IRP-based I/O operation as follows:

- Check whether an MDL exists for the buffer. The MDL pointer can be found in the *MdlAddress* or *OutputMdlAddress* parameter in the **FLT_PARAMETERS** for the operation. Minifilter drivers can call **FltDecodeParameters** to query for the MDL pointer.

  One method for obtaining a valid MDL is to look for the IRP_MN_MDL flag in the **MinorFunction** member of the I/O parameter block, **FLT_IO_PARAMETER_BLOCK**, in the callback data. The following example shows how to check for the IRP_MN_MDL flag.

  ```
  NTSTATUS status;
  PMDL *ReadMdl = NULL;
  PVOID ReadAddress = NULL;

  if (FlagOn(CallbackData->Iopb->MinorFunction, IRP_MN_MDL))
  {
      ReadMdl = &CallbackData->Iopb->Parameters.Read.MdlAddress;
  }
  ```

  However, the IRP_MN_MDL flag can be set only for read and write operations. It is best to use **FltDecodeParameters** to retrieve an MDL, because the routine checks for a valid MDL for any operation. In the following example, only the MDL parameter is returned if valid.

  ```
  NTSTATUS status;
  PMDL *ReadMdl = NULL;
  PVOID ReadAddress = NULL;

  status = FltDecodeParameters(CallbackData, &ReadMdl, NULL, NULL, NULL);
  ```

- If an MDL exists for the buffer, call **MmGetSystemAddressForMdlSafe** to obtain the system address for the buffer and then use this address to access the buffer.

  Continuing from the previous example, the following code obtains the system address.

  ```
  if (*ReadMdl != NULL)
  {
      ReadAddress = MmGetSystemAddressForMdlSafe(*ReadMdl, NormalPagePriority);
      if (ReadAddress == NULL)
      {
          CallbackData->IoStatus.Status = STATUS_INSUFFICIENT_RESOURCES;
          CallbackData->IoStatus.Information = 0;
      }
  }
  ```

- If there is no MDL for the buffer, use the buffer address to access the buffer. To ensure that a user-space buffer address is valid, the minifilter driver must use a routine such as **ProbeForRead** or **ProbeForWrite**, enclosing all buffer references in **try**/**except** blocks.

A preoperation callback routine should treat a buffer in a fast I/O operation as follows:

- Use the buffer address to access the buffer (because a fast I/O operation cannot have an MDL).
- To ensure that a user-space buffer address is valid, the minifilter driver must use a routine such as **ProbeForRead** or **ProbeForWrite**, enclosing all buffer references in **try**/**except** blocks.

For operations that can be fast I/O or IRP-based, all buffer references should be enclosed in **try**/**except** blocks. Although you do not have to enclose these references for IRP-based operations that use buffered I/O, the **try**/**except** blocks are a safe precaution.

Accessing User Buffers in a Postoperation Callback Routine

4/26/2017 • 2 min to read • Edit Online

A minifilter driver **postoperation callback routine** should treat a buffer in an IRP-based I/O operation as follows:

- Check whether an MDL exists for the buffer. The MDL pointer can be found in the *MdlAddress* or *OutputMdlAddress* parameter in the **FLT_PARAMETERS** for the operation. Minifilter drivers can call **FltDecodeParameters** to query for the MDL pointer.

  One method for obtaining a valid MDL is to look for the IRP_MN_MDL flag in the **MinorFunction** member of the I/O parameter block, **FLT_IO_PARAMETER_BLOCK**, in the callback data. The following example shows how to check for the IRP_MN_MDL flag.

  ```
  NTSTATUS status;
  PMDL *ReadMdl = NULL;
  PVOID ReadAddress = NULL;

  if (FlagOn(CallbackData->Iopb->MinorFunction, IRP_MN_MDL))
  {
      ReadMdl = &CallbackData->Iopb->Parameters.Read.MdlAddress;
  }
  ```

  However, the IRP_MN_MDL flag can be set only for read and write operations. It is best to use **FltDecodeParameters** to retrieve an MDL, because the routine checks for a valid MDL for any operation. In the following example, only the MDL parameter is returned if valid.

  ```
  NTSTATUS status;
  PMDL *ReadMdl = NULL;
  PVOID ReadAddress = NULL;

  status = FltDecodeParameters(CallbackData, &ReadMdl, NULL, NULL, NULL);
  ```

- If an MDL exists for the buffer, call **MmGetSystemAddressForMdlSafe** to obtain the system address for the buffer and then use this address to access the buffer. (**MmGetSystemAddressForMdlSafe** can be called at IRQL <= DISPATCH_LEVEL.)

  Continuing from the previous example, the following code obtains the system address.

  ```
  if (*ReadMdl != NULL)
  {
      ReadAddress = MmGetSystemAddressForMdlSafe(*ReadMdl, NormalPagePriority);
      if (ReadAddress == NULL)
      {
          CallbackData->IoStatus.Status = STATUS_INSUFFICIENT_RESOURCES;
          CallbackData->IoStatus.Information = 0;
      }
  }
  ```

- If there is no MDL for the buffer, check whether the system buffer flag is set for the operation by using the **FLT_IS_SYSTEM_BUFFER** macro.

  - If the FLT_IS_SYSTEM_BUFFER macro returns **TRUE**, the operation uses buffered I/O, and the buffer can safely be accessed at IRQL = DISPATCH_LEVEL.

  - If the FLT_IS_SYSTEM_BUFFER macro returns **FALSE**, the buffer cannot safely be accessed at IRQL = DISPATCH_LEVEL. If the postoperation callback routine can be called at DISPATCH_LEVEL, it must call **FltDoCompletionProcessingWhenSafe** to pend the operation until it can be processed at IRQL <= APC_LEVEL. The callback routine that is pointed to by the *SafePostCallback* parameter of **FltDoCompletionProcessingWhenSafe** should first call **FltLockUserBuffer** to lock the buffer and then call **MmGetSystemAddressForMdlSafe** to obtain the system address for the buffer.

A postoperation callback routine should treat a buffer in a fast I/O operation as follows:

- Use the buffer address to access the buffer (because a fast I/O operation cannot have an MDL).
- To ensure that a user-space buffer address is valid, the minifilter driver must use a routine such as **ProbeForRead** or **ProbeForWrite**, enclosing all buffer references in **try**/**except** blocks.
- The postoperation callback routine for a fast I/O operation is guaranteed to be called in the correct thread context.
- The postoperation callback routine for a fast I/O operation is guaranteed to be called at IRQL <= APC_LEVEL, so it can safely call routines such as **FltLockUserBuffer**.

The following example code fragment checks for the system buffer or fast I/O flags for a directory control operation and defers completion processing if necessary.

```
if (*DirectoryControlMdl == NULL)
{
    if (FLT_IS_SYSTEM_BUFFER(CallbackData) || FLT_IS_FASTIO_OPERATION(CallbackData))
    {
        dirBuffer = CallbackData->Iopb->Parameters.DirectoryControl.QueryDirectory.DirectoryBuffer;
    }
    else
    {
        // Defer processing until safe.
        if (!FltDoCompletionProcessingWhenSafe(CallbackData, FltObjects, CompletionContext, Flags,
ProcessPostDirCtrlWhenSafe, &retValue))
        {
            CallbackData->IoStatus.Status = STATUS_UNSUCCESSFUL;
            CallbackData->IoStatus.Information = 0;
        }
    }
}
```

For operations that can be either fast I/O or IRP-based, all buffer references should be enclosed in **try**/**except** blocks. Although you do not have to enclose these references for IRP-based operations that use buffered I/O, the **try**/**except** blocks are a safe precaution.

Managing Contexts in a Minifilter Driver

4/26/2017 • 1 min to read • Edit Online

A *context* is a structure that is defined by the minifilter driver and that can be associated with a filter manager object. Minifilter drivers can create and set contexts for the following objects:

- Files (Windows Vista and later only.)
- Instances
- Volumes
- Streams
- Stream handles (file objects)
- Transactions (Windows Vista and later only.)

Except for volume contexts, which must be allocated from nonpaged pool, contexts can be allocated from paged or nonpaged pool.

The filter manager deletes contexts automatically when the objects that they are attached to are deleted, when a minifilter driver instance is detached from a volume, or when the minifilter driver is unloaded.

This section includes:

Registering Context Types

4/26/2017 • 2 min to read • Edit Online

When a minifilter driver calls **FltRegisterFilter** from its **DriverEntry** routine, it must register each type of context that it uses.

To register context types, the minifilter driver creates a variable-length array of **FLT_CONTEXT_REGISTRATION** structures and stores a pointer to the array in the **ContextRegistration** member of the **FLT_REGISTRATION** structure that the minifilter driver passes in the *Registration* parameter of **FltRegisterFilter**. The order of the elements in this array does not matter. However, the last element in the array must be {FLT_CONTEXT_END}.

For each context type that the minifilter driver uses, it must supply at least one context definition in the form of an FLT_CONTEXT_REGISTRATION structure. Each FLT_CONTEXT_REGISTRATION structure defines the type, size, and other information for the context.

When the minifilter driver creates a new context by calling **FltAllocateContext**, the filter manager uses the *Size* parameter of the **FltAllocateContext** routine, as well as the **Size** and **Flags** members of the FLT_CONTEXT_REGISTRATION structure, to select the context definition to be used.

For fixed-size contexts, the **Size** member of the FLT_CONTEXT_REGISTRATION structure specifies the size, in bytes, of the portion of the context structure that is defined by the minifilter driver. The maximum size for a context is MAXUSHORT (64 KB). Zero is a valid size value. The filter manager implements fixed-size contexts using lookaside lists. The filter manager creates two lookaside lists for each size value: one paged and one nonpaged.

For variable-size contexts, the **Size** member must be set to FLT_VARIABLE_SIZED_CONTEXTS. The filter manager allocates variable-size contexts directly from paged or nonpaged pool.

In the **Flags** member of the FLT_CONTEXT_REGISTRATION structure, the FLTFL_CONTEXT_REGISTRATION_NO_EXACT_SIZE_MATCH flag can be specified. If the minifilter driver uses fixed-size contexts and this flag is specified, the filter manager allocates a context from the lookaside list if the context's size is greater than or equal to the requested size. Otherwise, the context's size must be equal to the requested size.

For a given context type, the minifilter driver can supply up to three fixed-size context definitions, each with a different size, and one variable-size definition. For more information, see **FLT_CONTEXT_REGISTRATION**.

The minifilter driver can optionally supply a context cleanup callback routine to be called before the context is freed. For more information, see **PFLT_CONTEXT_CLEANUP_CALLBACK**.

A minifilter driver can optionally define its own callback routines for allocating and freeing contexts, instead of relying on the filter manager to perform these tasks. However, this is very rarely necessary. For more information, see **PFLT_CONTEXT_ALLOCATE_CALLBACK** and **PFLT_CONTEXT_FREE_CALLBACK**.

The following code example, which is taken from the CTX sample minifilter driver, shows an array of FLT_CONTEXT_REGISTRATION structures that are used to register instance, file, stream, and file object (stream handle) contexts.

```
const FLT_CONTEXT_REGISTRATION contextRegistration[] =
{
    { FLT_INSTANCE_CONTEXT,              //ContextType
      0,                                 //Flags
      CtxContextCleanup,                 //ContextCleanupCallback
      CTX_INSTANCE_CONTEXT_SIZE,         //Size
      CTX_INSTANCE_CONTEXT_TAG           //PoolTag
    },
    { FLT_FILE_CONTEXT,                  //ContextType
      0,                                 //Flags
      CtxContextCleanup,                 //ContextCleanupCallback
      CTX_FILE_CONTEXT_SIZE,             //Size
      CTX_FILE_CONTEXT_TAG               //PoolTag
    },
    { FLT_STREAM_CONTEXT,                //ContextType
      0,                                 //Flags
      CtxContextCleanup,                 //ContextCleanupCallback
      CTX_STREAM_CONTEXT_SIZE,           //Size
      CTX_STREAM_CONTEXT_TAG             //PoolTag
    },
    { FLT_STREAMHANDLE_CONTEXT,          //ContextType
      0,                                 //Flags
      CtxContextCleanup,                 //ContextCleanupCallback
      CTX_STREAMHANDLE_CONTEXT_SIZE,     //Size
      CTX_STREAMHANDLE_CONTEXT_TAG       //PoolTag
    },
    { FLT_CONTEXT_END }
};
```

Creating Contexts

4/26/2017 • 1 min to read • Edit Online

Once a minifilter driver has registered the context types that it uses, it can create a context by calling **FltAllocateContext**. This routine selects the appropriate context definition to use according to the criteria described in Registering Context Types.

In the following code example, taken from the CTX sample minifilter driver, the **CtxInstanceSetup** routine calls **FltAllocateContext** to create an instance context:

```
status = FltAllocateContext(
 FltObjects->Filter,           //Filter
           FLT_INSTANCE_CONTEXT,         //ContextType
           CTX_INSTANCE_CONTEXT_SIZE,    //ContextSize
 NonPagedPool,                 //PoolType
           &instanceContext);            //ReturnedContext
```

In the CTX sample, the following context definition is registered for instance contexts:

```
{ FLT_INSTANCE_CONTEXT,              //ContextType
  0,                                 //Flags
 CtxContextCleanup,                 //ContextCleanupCallback
  CTX_INSTANCE_CONTEXT_SIZE,         //Size
  CTX_INSTANCE_CONTEXT_TAG },        //PoolTag
```

This is a fixed-size context definition, because the **Size** member is a constant. (If the **Size** member were FLT_VARIABLE_SIZED_CONTEXTS, it would be a variable-size context definition.) Note that the FLTFL_CONTEXT_REGISTRATION_NO_EXACT_SIZE_MATCH flag is not set in the **Flags** member. In this case, if the value of the *Size* parameter of **FltAllocateContext** matches that of the **Size** member of the context definition, **FltAllocateContext** allocates the instance context from the appropriate nonpaged lookaside list. If the values do not match, **FltAllocateContext** fails with a return value of STATUS_FLT_CONTEXT_ALLOCATION_NOT_FOUND.

FltAllocateContext initializes the reference count on the new context to one. When the context is no longer needed, the minifilter driver must release this reference. Thus, every call to **FltAllocateContext** must be matched by a subsequent call to **FltReleaseContext**.

Setting Contexts

4/26/2017 • 1 min to read • Edit Online

After creating a new context, a minifilter driver can attach it to an object by calling **FltSet*Xxx*Context**, where *Xxx* is the context type.

If the *Operation* parameter of the **FltSet*Xxx*Context** routine is set to FLT_SET_CONTEXT_KEEP_IF_EXISTS, **FltSet*Xxx*Context** attaches the newly allocated context to the object only if the minifilter driver has not already set a context for the object. If the minifilter driver has already set a context, **FltSet*Xxx*Context** returns STATUS_FLT_CONTEXT_ALREADY_DEFINED, which is an NTSTATUS error code, and does not replace the existing context. If the *OldContext* parameter of the **FltSet*Xxx*Context** routine is non-**NULL**, it receives a pointer to the existing context. When this pointer is no longer needed, the minifilter driver must release it by calling **FltReleaseContext**.

If the *Operation* parameter is set to FLT_SET_CONTEXT_REPLACE_IF_EXISTS, **FltSet*Xxx*Context** always attaches the new context to the object. If the minifilter driver has already set a context, **FltSet*Xxx*Context** deletes the existing context, sets the new context, and increments the reference count on the new context. If the *OldContext* parameter is non-**NULL**, it receives a pointer to the deleted context. When this pointer is no longer needed, the minifilter driver must release it by calling **FltReleaseContext**.

In the following code example, taken from the CTX sample minifilter driver, the **CtxInstanceSetup** routine creates and sets an instance context:

```
status = FltAllocateContext(
           FltObjects->Filter,           //Filter
           FLT_INSTANCE_CONTEXT,         //ContextType
           CTX_INSTANCE_CONTEXT_SIZE,    //ContextSize
           NonPagedPool,                 //PoolType
           &instanceContext);            //ReturnedContext
...
status = FltSetInstanceContext(
           FltObjects->Instance,              //Instance
           FLT_SET_CONTEXT_KEEP_IF_EXISTS,    //Operation
           instanceContext,                   //NewContext
           NULL);                             //OldContext

if (instanceContext != NULL) {
  FltReleaseContext(instanceContext);
}
return status;
```

Note that after the call to **FltSetInstanceContext**, there is a call to **FltReleaseContext** to release the reference count that was set by **FltAllocateContext** (*not* **FltSetInstanceContext**). This is explained in Releasing Contexts.

Getting Contexts

4/26/2017 • 1 min to read • Edit Online

Once a minifilter driver has set a context for an object, it can get the context by calling **FltGet*Xxx*Context**, where *Xxx* is the context type.

In the following code example, taken from the SwapBuffers sample minifilter driver, the minifilter driver calls **FltGetVolumeContext** to get a volume context:

```
status = FltGetVolumeContext(
 FltObjects->Filter,    //Filter
 FltObjects->Volume,    //Volume
               &volCtx);              //Context
...
if (volCtx != NULL) {
 FltReleaseContext(volCtx);
}
```

If the call to **FltGetVolumeContext** is successful, the *Context* parameter receives the address of the caller's volume context. **FltGetVolumeContext** increments the reference count on the *Context* pointer. Thus, when this pointer is no longer needed, the minifilter driver must release it by calling **FltReleaseContext**.

Referencing Contexts

4/26/2017 • 1 min to read • Edit Online

The filter manager uses reference counting to manage the lifetime of a minifilter driver context. A reference count is a number indicating the state of a context. Whenever a context is created, the reference count of the context is initialized to one (this is called the initial reference to the context). Whenever a context is referenced by a system component, the reference count of the context is incremented by one. When a context is no longer needed, its reference count is decremented. A positive reference count means that the context is usable. When the reference count becomes zero, the context is unusable, and the filter manager eventually frees it.

The initial reference to the context is typically released when the object is torn down. However, if a minifilter driver must remove a context from an object, the minifilter driver must somehow release that initial reference to the context. To safely release that initial reference to the context, the minifilter driver calls **FltDeleteContext**.

A minifilter driver can add its own reference to a context by calling **FltReferenceContext** to increment the context's reference count. This added reference must eventually be removed by calling **FltReleaseContext**.

Releasing Contexts

4/26/2017 • 1 min to read • Edit Online

A minifilter driver releases a context by calling **FltReleaseContext**. Every successful call to one of the following routines must eventually be matched by a call to **FltReleaseContext**:

FltAllocateContext

FltGetInstanceContext

FltGetFileContext

FltGetStreamContext

FltGetStreamHandleContext

FltGetTransactionContext

FltGetVolumeContext

FltReferenceContext

Note that the *OldContext* pointer returned by **FltSet*Xxx*Context** and the *Context* pointer returned by **FltDeleteContext** must also be released when they are no longer needed.

In the following code example, taken from the CTX sample minifilter driver, the **CtxInstanceSetup** routine creates and sets an instance context and then calls **FltReleaseContext**:

```
status = FltAllocateContext(
           FltObjects->Filter,           //Filter
           FLT_INSTANCE_CONTEXT,         //ContextType
           CTX_INSTANCE_CONTEXT_SIZE,    //ContextSize
           NonPagedPool,                 //PoolType
           &instanceContext);            //ReturnedContext
...
status = FltSetInstanceContext(
           FltObjects->Instance,              //Instance
           FLT_SET_CONTEXT_KEEP_IF_EXISTS,    //Operation
           instanceContext,                   //NewContext
           NULL);                             //OldContext

if (instanceContext != NULL) {
  FltReleaseContext(instanceContext);
}
return status;
```

Note that **FltReleaseContext** is called regardless of whether the call to **FltSetInstanceContext** succeeds. In both cases, the caller must call **FltReleaseContext** to release the reference set by **FltAllocateContext** (not **FltSetInstanceContext**).

If the context is successfully set for the instance, **FltSetInstanceContext** adds its own reference to the instance context. Thus, the reference set by **FltAllocateContext** is no longer needed, and the call to **FltReleaseContext** removes it.

If the call to **FltSetInstanceContext** fails, the instance context has only one reference, namely the one set by **FltAllocateContext**. When **FltReleaseContext** returns, the instance context has a reference count of zero, and it is freed by the filter manager.

Deleting Contexts

4/26/2017 • 1 min to read • Edit Online

Every context that is set by a successful call to **FltSet*Xxx*Context** must eventually be deleted. However, the filter manager deletes contexts automatically when the objects that they are attached to are deleted, when a minifilter driver instance is detached from a volume, or when the minifilter driver is unloaded. Thus, it is rarely necessary for a minifilter driver to explicitly delete a context.

A minifilter driver can delete a context by calling **FltDelete*Xxx*Context**, where *Xxx* is the context type, or by calling **FltDeleteContext**.

A context can be deleted only if it is currently set for an object. A context cannot be deleted if it has not yet been set, or if it has already been replaced by a successful call to **FltSet*Xxx*Context**.

In the call to **FltDelete*Xxx*Context**, the old context is returned in the *OldContext* parameter, if it is non-**NULL**. If the *OldContext* parameter is **NULL**, the filter manager decrements the reference count on the context, which is then freed unless the minifilter driver has an outstanding reference on it.

The following code example shows how to delete a stream context:

```
status = FltDeleteStreamContext(
 FltObjects->Instance,      //Instance
 FltObjects->FileObject,    //FileObject
           &oldContext);              //OldContext
...
if (oldContext != NULL) {
 FltReleaseContext(oldContext);
}
```

In this example, **FltDeleteStreamContext** removes the stream context from the stream, but it does not decrement the context's reference count, because the *OldContext* parameter is non-**NULL**. **FltDeleteStreamContext** returns the address of the deleted context in the *OldContext* parameter. After performing any needed processing, the caller must release the deleted context by calling **FltReleaseContext**.

Freeing Contexts

4/26/2017 • 1 min to read • Edit Online

A context is freed after it is deleted and all outstanding references to it have been released.

There is one exception to this rule: if a context has been created but has not been set by calling **FltSet*Xxx*Context**, it does not need to be deleted. It is freed when its reference count reaches zero. (See the code example in Releasing Contexts.)

When a minifilter driver registers its context types, each context definition can optionally include a context cleanup callback routine to be called before the context is freed. For more information, see **PFLT_CONTEXT_CLEANUP_CALLBACK**.

File System Support for Contexts

4/26/2017 • 1 min to read • Edit Online

To support file contexts (if applicable), stream contexts, and file object (stream handle) contexts, a file system must use the **FSRTL_ADVANCED_FCB_HEADER** structure. All Microsoft Windows file systems use this structure, and all third-party file system developers are strongly encouraged to do so as well. For more information, see **FsRtlSetupAdvancedHeader** and **FSRTL_ADVANCED_FCB_HEADER**.

The NTFS and FAT file systems do not support file, stream, or file object contexts on paging files, in the pre-create or post-close path, or for **IRP_MJ_NETWORK_QUERY_OPEN** operations.

A minifilter driver can determine whether a file system supports stream contexts and file object contexts for a given file object by calling **FltSupportsStreamContexts** and **FltSupportsStreamHandleContexts**, respectively.

File contexts are available on Windows Vista and later.

For file systems (such as FAT) that support only a single data stream per file, file contexts are equivalent to stream contexts. Such file systems usually support stream contexts but do not support file contexts. Instead, the filter manager provides this support, using the file system's existing support for stream contexts. For minifilter driver instances attached to these file systems, **FltSupportsFileContexts** returns **FALSE**, while **FltSupportsFileContextsEx** returns **TRUE** (when a valid non-**NULL** value is passed for the *Instance* parameter).

To support file contexts, a file system must:

- Embed a **FileContextSupportPointer** member of type PVOID in its file context structure, usually the file context block (FCB). The file system must initialize this member to **NULL**.
- Use **FsRtlSetupAdvancedHeaderEx** (instead of **FsRtlSetupAdvancedHeader**) to initialize its stream context structure, passing a valid pointer to the **FileContextSupportPointer** member (embedded in the corresponding file context structure) for the *FileContextSupportPointer* parameter. For more information, see **FsRtlSetupAdvancedHeaderEx** and **FSRTL_ADVANCED_FCB_HEADER**.
- Call **FsRtlTeardownPerFileContexts** to free all file context structures that filter and minifilter drivers have associated with a file when the file system deletes its file context structure for the file.

Best Practices

4/26/2017 • 1 min to read • Edit Online

If a minifilter driver creates only one minifilter driver instance per volume, it should use instance contexts rather than volume contexts for better performance.

A minifilter driver can also improve performance by storing a pointer to the minifilter driver instance context inside its stream or stream handle contexts.

Network Redirector Drivers

4/26/2017 • 1 min to read • Edit Online

This section includes the following topics, which describe network redirector drivers:

Introduction to Remote File Systems

4/26/2017 • 1 min to read • Edit Online

Remote file systems enable an application that runs on a client computer to access files stored on a different computer. Remote file systems also often make other resources (remote printers, for example) accessible from a client computer. The remote file and resource access takes place using some form of local area network (LAN), wide area network (WAN), point-to-point link, or other communication mechanism. These file systems are often referred as network file systems or distributed file systems.

Microsoft Networks is the native remote file system included in Windows Server 2003, Windows XP, and Windows 2000. Microsoft Networks provides remote access to files as well as access to remote printers and plotters. Microsoft Networks was called LAN Manager Network in earlier versions of Windows.

Microsoft also includes support for several other remote file systems on Windows:

- NetWare Core Protocol (NCP)
- WebDAV (file access that uses remote web servers)
- AppleTalk file and printer sharing (supports connections from Macintosh clients to systems running Windows Server 2003 and Windows 2000 Server)
- Network File System (NFS)
- IBM mainframe VSAM and AS/400 file access, included with Microsoft Host Integration Server 2000

Note In versions of the Windows operating system prior to Windows Server 2003 R2, you could obtain NFS by installing Microsoft Services for UNIX. With Windows Server 2003 R2 and later versions of the Windows operating system, NFS is included as an optional Windows component.

Remote file systems are implemented by a collection of software components. The number and complexity of the software components required varies based on the design and complexity of the remote file system.

Software on the client system provides remote file and resource access. This client software functions as a "network redirector" forwarding local calls for file operations to some remote server. This network redirector makes the remote resources appear as if they are local.

Software on the server system implements the remote file server operations that access local file storage or resources on the server. Requests are received from client network redirectors that are processed on the file server, and the responses are sent back to the client.

A system can act as both a client to some systems and as a server to other clients. So, it is common to find both client and server software running on a single system.

This section contains the following topic:

Components of a Remote File System

Components of a Remote File System

4/26/2017 • 1 min to read • Edit Online

The following basic elements are required to implement a remote file system:

- Network redirector software installed on the client.
- A well-defined transport protocol used for communication.
- File server software installed on the server.

For example, the Microsoft Network remote file system is implemented as follows:

- The client for Microsoft Networks software provides the client network redirector components.
- The Common Internet File System (CIFS), which is also called the Server Message Block (SMB) protocol, defines the network transport protocol used for communication.
- The LAN Manager Server (sometimes called SMB Server) provides the file server service.

The network redirector software installed on the client consists of several software components, some that operate in user-mode and some that operate in kernel-mode.

The following sections discuss concepts that are important to developers of remote file systems on Windows Server 2003, Windows XP, and Windows 2000. The following topics are discussed:

Introduction to Network Redirectors

Kernel Network Redirector Driver Components

Introduction to Network Redirectors

4/26/2017 • 1 min to read • Edit Online

This section introduces fundamental concepts that are important to developers implementing network redirector software for the client of a remote file system. This section includes:

What is a Network Redirector?

4/26/2017 • 1 min to read • Edit Online

A network redirector consists of software components installed on a client computer that is used for accessing files and other resources (printers and plotters, for example) on a remote system. The network redirector sends (or redirects) requests for file operations from local client applications to a remote server where the requests are processed. The network redirector receives responses from the remote server that are then returned to the local application. The network redirector software creates the appearance on the client system that remote files and resources are the same as local files and resources and allows them to be used and manipulated in the same ways.

The network redirector tries to make access to remote resources as transparent as possible for the local client application. If there are network problems, the request may be retried several times before the network redirector gives up and returns an error to the client application.

This section includes the following topic:

Basic Architecture of a Network Redirector

Basic Architecture of a Network Redirector

4/26/2017 • 4 min to read • Edit Online

The following basic software components are required as part of a network redirector:

- A kernel-mode file system device driver (SYS) that provides the network redirector functions.
- A user-mode dynamic link library (DLL) that provides access for client user-applications to the Network Provider interface for non-file operations and enables communication with the kernel-mode file system driver providing the network redirector functions.

The kernel-mode file system driver component of the network redirector interacts with the Object Manager, Cache Manager, Memory Manager, I/O System, and other kernel systems in order to integrate remote file access into the operating system. The driver receives requests from a client application through the I/O Manager for remote file access and sends the necessary information over the network to the remote file server. The driver also receives the responses from the remote file server and passes the information back to the client application through the I/O Manager. The network redirector driver must also implement the network protocol used for remote access or provide calls to another kernel driver, user-mode application, or service that implements this network protocol or communication mechanism. The most common method to implement this network communication is to use the Transport Driver Interface (TDI) to call an existing network protocol stack (TCP/IP, for example). The network protocol stack then communicates at the bottom layer using the Network Driver Interface Specification (NDIS) with the network interface card. It is also possible to use an application-specific interface for this communication mechanism.

The user-mode DLL receives special requests from the client through the Network Provider interface that are not file operations. These requests can be used for determining capabilities of the network provider, enumerating remote network resources, logging on to a remote network, changing network passwords, connecting with remote servers, mounting remote network shares or resources, and disconnecting from remote shares. The user-mode DLL implements the equivalent of the WNet user-mode APIs for accessing the remote file system. This DLL then communicates with the kernel-mode network redirector driver through special IOCTL calls to fulfill these client requests. For example, this DLL is called by Windows Explorer when enumerating network resources (browsing the Network Neighborhood) or when connecting to remote file shares and resources (printers). This DLL is also called from the NET command-line tool to view remote resources (**net view** \\computername) and connect or disconnect to remote file shares (**net use**). This DLL can also be called by custom applications using the exposed WNet APIs of the Multiple Provider Router (MPR).

A network redirector may also need several other components:

- A service application may be needed to act as an intermediary between the user-mode DLL and the kernel-mode driver if any global data tables need to be securely stored in the user-mode DLL. Since the user-mode DLL is instantiated in the process space of the client application, there is no way to secure any internal data tables that need to be global across all client applications. It is possible to have per-client data that is private to each client application when connecting to the user-mode network provider DLL, but not global data. A service application (an .exe file) can be used as the intermediary between the user-mode DLL and the kernel driver where global data tables are stored. For example, Microsoft Networks uses the Workstation service built into the svchost.exe to store a global table of "net use" connections and act as an intermediary to the kernel-mode driver. In the case of Microsoft Networks, the user-mode DLL, ntlanman.dll, calls the Workstation service in svchost.exe, which then calls the kernel-mode driver, mrxsmb.sys, which provides the network redirector functions.
- A system for installing and properly registering the network redirector components in the system, which

must include creating any necessary registry entries. There should also be a method to uninstall the network redirector if the software is no longer needed. It is possible to use the Microsoft Systems Installer (MSI) or Windows INF script files to perform this setup. This installation and setup could also be accomplished using a custom installation program.

- A custom kernel-mode TDI driver needs to be installed if the network redirector uses a network protocol not included in the operating system (Xerox Network Services, for example). The new TDI driver would need to be installed that implements the network protocol to use for communication. The kernel-mode network redirector driver connects to the upper edge of the custom TDI driver. The custom TDI driver would in turn communicate at its lower edge with an NDIS driver.

  Note that if the network redirector uses the TCP/IP protocol, a custom TDI driver is not required. The network redirector would connect to the upper edge of the TCP/IP protocol TDI driver in this case.

- An administration tool is occasionally needed to provide access from user-mode to the kernel-mode driver for special configuration, diagnostics, and administration. This tool can also be used to enable or disable tracing and logging for troubleshooting problems. The tool would communicate with the kernel driver by using various custom private IOCTLs. This tool might also provide access to the Service Control Manager (SCM) to manage an intermediary service where global data is stored securely for the user-mode Network Provider DLL.

Note TDI will not be supported in Microsoft Windows versions after Windows Vista. Use Windows Filtering Platform or Winsock Kernel instead.

The Kernel-Mode Network Redirector Driver

4/26/2017 • 1 min to read • Edit Online

This section introduces concepts that are important to developers of kernel-mode drivers for network redirectors. This section includes:

Network Redirector Design and Performance

4/26/2017 • 3 min to read • Edit Online

There are two different paths to achieving high performance in a network redirector. The first approach stresses getting quickly to the network, above all else. The second approach tries to use sophisticated techniques for minimizing network operations; these operations are often time consuming and use up scarce network resources (available bandwidth, for example). Developers generally turn to the second approach when they are unsuccessful at the first.

While never discounting the benefit of short code paths, the "run-to-the-wire" approach provides better performance only when system operation does not encounter some fundamental bottleneck. For example, consider a 10 megabit Ethernet network where the actual transfer rate is limited to 1.25 MB/second. While this is faster than some disks, it is still considerably slower than memory access speeds. For this reason, a system that provides in-memory caching of data that is accessed frequently will provide better performance, in many cases, than one that does not. Conversely, performance may suffer if the network redirector on the client insists on trying to cache in unfavorable situations. For example, if the cache manager relies on the page granularity of the x86-based hardware, then this may mean that a full 4-KB block would be read and written in response to each 32-byte write that is initiated by the application. Clearly, to get the best performance, a network redirector must implement sophisticated cache management strategies, but must also be equally prepared to determine dynamically whether to use them. And, if caching is disabled, a network redirector must be prepared to run for the wire.

A second example is provided in the case of a CPU-saturated server. The performance of the remote file system may be significantly hampered if measures are not taken to relieve congestion at the server. For example, a server under saturation conditions may have a greater likelihood for dropping packets than an unloaded server. The client network redirector would be well advised to pace the packets sent in such a way as to reduce the probability of dropped packets. Even better, the redirector could take measures to attempt to combine several operations into a single packet. Combining operations so that the server gets more operations per packet is a win in the saturated server case.

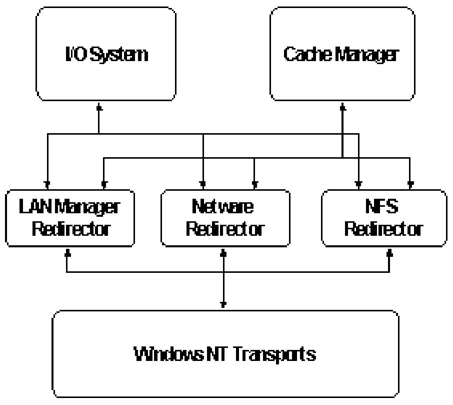
Performance is also reduced when a system uses a networking mode that is inappropriate for the type of traffic and the likely network conditions. Experience suggests a possible correlation between using a datagram-based mode and having better performance for small I/O requests. It is not clear whether this improvement is across-the-board or whether it only applies in those scenarios where the probability of datagram delivery is very high. But the available data suggests that implementation of a "connectionless" mode is certainly worthwhile in addition to a connection-oriented mode. The ideal would be a network redirector where both modes would be supported by the network protocol and available for use, so the redirector could dynamically select how the traffic should be carried.

Finally, performance may be enhanced if appropriate data obtained during operations is available for making smart decisions. There are numerous examples of this in Windows. For example, Cache Manager keeps a short history of read commands and tracks cache hits and misses, which is used to guide decisions on read-ahead scheduling. A second example might be the use of "environmental information" for making decisions. There might be information available in the file name, the share name, or from a user or administrator that could greatly increase performance. For example, response can sometimes be tuned to improve performance by just changing cache size parameters to capture the data file being used. This "environmental knowledge" of access patterns for certain scenarios could potentially be used to suggest dynamic changes (hints) for improving performance.

Network Redirector Design in Windows NT

4/26/2017 • 1 min to read • Edit Online

The architecture that implements a kernel-mode driver for network redirectors has changed over time. The general model for network redirectors has typically been based on the architecture that implements the kernel driver for the Client for Microsoft Networks (LAN Manager Client). The original scheme introduced in Windows NT 3.0 used no shared components and was limited. This model is often called the original rdr driver model (rdr was an abbreviation for redirector). No special support from the operating system was provided to simplify the process of writing a network redirector. Each kernel-mode driver implemented all of the functions required for a network redirector. Consequently, each kernel driver would include a large amount of code for interactions with the I/O Manager, Cache Manager, and Memory Manager. Each network redirector (LAN Manager, NetWare, and NFS, for example) installed on Windows had to implement all of these functions themselves. This design model was used for drivers for network redirectors through Windows NT 4.0. The following is a diagram of this architecture, with multiple redirectors.



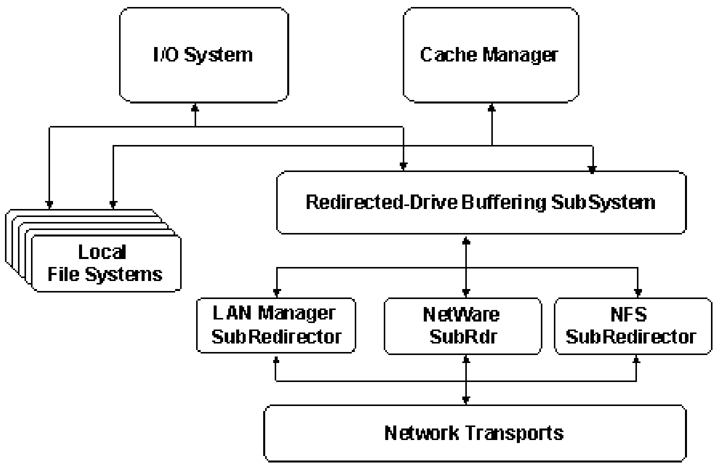
Network Redirector Design in Windows 2000

4/26/2017 • 1 min to read • Edit Online

A significant challenge in the design of network redirectors is the relatively complex translation that is performed from user-initiated operations to low-level network operations, both with respect to operation selection and timing. Dealing with the Windows I/O System, Cache Manager, and Memory Manager is a relatively complex undertaking. This is especially true when considering the variety of buffering modes that may be appropriate for a remote communication mechanism, such as a computer network where the speed and reliability may vary considerably. The implementation of these buffering operations in a network redirector represents a significant investment in function that would ideally be shared and reused by drivers.

Windows 2000 introduced a new driver model (often called the mini-redirector architecture, or rdr2) for network redirectors based on a layered or miniport driver approach. Rather than having to re-implement the complex code used for buffering and interaction with the I/O Manager and Cache Manager in each driver, this large block of code was pulled out and made available to all potential network redirectors. The shared common buffering code is called the Redirected Drive Buffering SubSystem (RDBSS).

A model of this architecture with multiple redirectors is shown below.



This RDBSS design change offers several of the following benefits:

- Simplifies the process of writing the kernel driver for a network redirector since a large amount of common code that was needed to deal with the I/O System, Cache Manager, and Memory Manager was provided.
- Makes available to other network redirector drivers a considerable amount of performance improvements based on buffering algorithms and kernel optimizations developed for Microsoft Networks.
- Simplifies maintenance since only one copy of the buffering code needs to be developed and maintained. In the older model, a copy was required for each redirector.
- Provides a strong encapsulation of the network protocol-specific component of a network redirector, so driver developers can focus on these aspects of the network redirector that are unique and specific to their application or product.
- Simplifies debugging drivers for network redirectors by providing decoupling based on this layered approach.

The RDBSS model was introduced with Windows 2000. This same model is also used on Windows Server 2003 and Windows XP.

The Redirected Drive Buffering SubSystem

4/26/2017 • 4 min to read • Edit Online

The Redirected Drive Buffering SubSystem (RDBSS) is provided as a kernel-mode file system driver, *rdbss.sys*, which is included with the operating system and as a static library, *rdbsslib.lib*, which is included with the Windows Driver Kit (WDK). The static library duplicates the code in the *rdbss.sys* kernel-mode driver.

The original scheme was that all network mini-redirectors would link dynamically with the *rdbss.sys* kernel driver. Drivers that linked dynamically to *rdbss.sys* are called non-monolithic drivers. However, this feature was never fully implemented, so only the Microsoft SMB network mini-redirector links dynamically to *rdbss.sys* and is a non-monolithic driver. All other network mini-redirectors, including other Microsoft redirectors included with the operating system, link against the *rdbsslib.lib* library and statically include the same code as the *rdbss.sys* kernel driver. While the same RDBSS source code is used in both the *rdbss.sys* driver and the *rdbsslib.lib* library, the size of a monolithic network mini-redirector driver is larger since most of the code in *rdbss.sys* is included with each driver. For example, on Windows Server 2003, the file sizes for various drivers are as follows:

DRIVER	FILE SIZE	DESCRIPTION
rdbss.sys	156 KB	The RDBSS device driver used only by the Microsoft SMB mini-redirector.
mrxdav.sys	179 KB	The Microsoft WebDAV network mini-redirector driver, which links in much of *rdbsslib.lib*.
rdrpdr.sys	185 KB	The Microsoft Terminal Services network mini-redirector driver, which links in much of *rdbsslib.lib*.

The RDBSS communicates with the I/O Manager, Cache Manager, Memory Manager, network mini-redirectors, and other kernel systems. RDBSS supports the following features:

- A well-defined mechanism is defined for communication between RDBSS and mini-redirector drivers.
- Multiple buffering modes are supported. There are also support routines to help the network mini-redirector change the buffering modes dynamically to enable live sharing.
- The network mini-redirector can indicate support for various options, including case-sensitive names, UNC naming, printing, pipes, mailslots, and scavenging (reusing some data structures).
- The network mini-redirector provides support for namespace, file information, and connection caching to reduce unnecessary network operations.

The RDBSS uses a well-defined mechanism for communication. RDBSS exports a large number of functions that can be called by a mini-redirector driver and other kernel systems to set options and perform various operations.

A mini-redirector implements a number of call-back functions that are used by RDBSS to communicate with the mini-redirector driver. When a mini-redirector driver first starts (in its **DriverEntry** routine), the driver calls an RDBSS function for registering mini-redirector drivers and passes in a structure with pointers to these callback functions (dispatch table) along with some configuration data values.

The callback functions that must be implemented in a mini-redirector driver fall into the following basic categories:

- Callbacks to manage network mini-redirector data structures.
- Callbacks for file I/O operations.
  - Synchronous file I/O routines (create, close, and cleanup, for example).
  - Asynchronous file I/O routines (read, write. FSTCL, IOCTL, and lock, for example). For historical reasons, these synchronous file I/O routines in a network mini-redirector were often called low I/O operations.

The file I/O callbacks are expected to be synchronous, except for the functions that fall in the low I/O category. The low I/O routines can operate asynchronously and therefore must support cancellation.

RDBSS and mini-redirectors commonly use a fundamental data structure, the RX_CONTEXT, to pass information. The RX_CONTEXT encapsulates a large amount of information and includes a pointer to the current IRP. There is a default size for this RX_CONTEXT data structure. The size of this RX_CONTEXT structure can be extended based on the requirements of the remote file system and network mini-redirector. The size to use for this extended RX_CONTEXT is defined when a mini-redirector first registers with RDBSS (one of the data values specified).

RDBSS and mini-redirectors also make extensive use of the following data structure abstractions:

- Server Call Context (SRV_CALL)--The context associated with each known file system server.
- Net Roots (NET_ROOT)--The root of a share opened by the user.
- Virtual Net Roots (V_NET_ROOT)--The view of a share on a server. The view can be constrained along multiple dimensions. As an example the view can be associated with a logon ID, which constrains the operations that can be performed on the share.
- File Control Blocks (FCB)--The RDBSS data structure associated with each unique file opened on the client.
- File Object Extensions (FOBX)--The RDBSS extension to the NT FILE_OBJECT.
- ServerSide Open Context (SRV_OPEN)--The open handle to a file on a remote share.

RDBSS has very flexible support for buffering. RDBSS can be set to separately buffer read and write requests. File size and file time can also be cached. Explicit buffering strategies are allowed and these can be set when a share on a file server is opened. Buffering can also be set on an individual file basis.

RDBSS will try to use access history to reduce the need for unnecessary network requests. For example, if an open request for a remote file fails, this information will be cached by RDBSS. If the client immediately requests to open a similar file, with a change of case in the file name, RDBSS will fail the request for remote file systems that ignore case in file names.

Oplock Semantics

4/26/2017 • 1 min to read • Edit Online

This section contains the following topics:

Overview

4/26/2017 • 5 min to read • Edit Online

Opportunistic locks, or oplocks, provide a mechanism that allows file server clients (such as those utilizing the SMB and SMB2 protocols) to dynamically alter buffering strategy for a given file or stream in a consistent manner to increase performance and to reduce network use. To increase the network performance for remote file operations, a client can locally buffer file data, which reduces or eliminates the need to send and receive network packets. For example, a client may not have to write information into a file on a remote server if the client knows that no other process is accessing the data. Likewise, the client may buffer read-ahead data from the remote file if the client knows that no other process is writing data to the remote file. Applications and drivers can also use oplocks to transparently access files without affecting other applications that might need to use those files.

File systems like NTFS support multiple data streams per file. Oplocks are granted on stream handles, meaning that the operations apply to the given open stream of a file and, in general, operations on one stream do not affect oplocks on a different stream. There are exceptions to this, which are explicitly listed below. For file systems that do not support alternate data streams, such as FAT, think of "file" when this document refers to "stream".

There are currently eight different types of oplocks:

- A **Level 2** (or shared) oplock indicates that there are multiple readers of a stream and no writers. This supports client read caching.
- A **Level 1** (or exclusive) oplock allows a client to open a stream for exclusive access and allows the client to perform arbitrary buffering. This supports client read caching and write caching.
- A **Batch** oplock (also exclusive) allows a client to keep a stream open on the server even though the local accessor on the client machine has closed the stream. This supports scenarios where the client needs to repeatedly open and close the same file, such as during batch script execution. This supports client read caching, write caching, and handle caching.
- A **Filter** oplock (also exclusive) allows applications and file system filters (including minifilters), which open and read stream data, a way to "back out" when other applications, clients, or both try to access the same stream. This supports client read caching and write caching.
- A **Read** (R) oplock (shared) indicates that there are multiple readers of a stream and no writers. This supports client read caching.
- A **Read-Handle** (RH) oplock (shared) indicates that there are multiple readers of a stream, no writers, and that a client can keep a stream open on the server even though the local accessor on the client machine has closed the stream. This supports client read caching and handle caching.
- A **Read-Write** (RW) oplock (exclusive) allows a client to open a stream for exclusive access and allows the client to perform arbitrary buffering. This supports client read caching and write caching.
- A **Read-Write-Handle** (RWH) oplock (exclusive) allows a client to keep a stream open on the server even though the local accessor on the client machine has closed the stream. This supports client read caching, write caching, and handle caching.

Level 1, Level 2, and Batch oplocks were implemented in Windows NT 3.1. The Filter oplock was added in Windows 2000. R, RH, RW, and RWH oplocks have been added in Windows 7.

Some oplocks seem quite similar. In particular, R seems similar to Level 2, RW seems similar to Level 1, and RWH seems similar to Batch. The R, RH, RW, and RWH oplocks (hereinafter referred to collectively as "Windows 7 oplocks") have been added to Windows 7 to provide greater flexibility for the caller to express caching intentions, and to allow oplock breaks and upgrades (that is, modification of the oplock state from one level to a level of greater caching; for example, upgrading a Read oplock to a Read-Write oplock). This flexibility is not achievable with the Level 2, Level 1, Batch, and Filter oplocks (hereinafter referred to collectively as "legacy oplocks"). Differences between the Windows 7 oplocks and the legacy oplocks are discussed later in this documentation.

The core functionality of the oplock package is implemented in the kernel (primarily through *FsRtlXxx* routines). File systems call into this package to implement the oplock functionality in their file system. This document describes how the NTFS file system interoperates with the kernel oplock package. Other file systems function in a similar manner though there might be minor differences.

Oplocks are granted on stream handles. This means an oplock is granted for a given open of a stream. Starting with Windows 7, the stream handle can be associated with an *oplock key*, that is, a GUID value that is used to identify multiple handles that belong to the same client cache view (see the note later in this topic). The oplock key can be explicitly provided (to **IoCreateFileEx**) when the handle is created. If an oplock key is not explicitly specified when the handle is created, the system will treat the handle as having a unique oplock key associated with it, such that its key will differ from any other key on any other handle. If a file operation is received on a handle other than the one on which the oplock was granted, and the oplock key that is associated with the oplock's handle differs from the key that is associated with the operation's handle, and that operation is not compatible with the currently granted oplock, then that oplock is broken. The oplock breaks even if it is the same process or thread performing the incompatible operation. For example, if a process opens a stream for which an exclusive oplock is granted and the same process then opens the same stream again, using a different (or no) oplock key, the exclusive oplock is broken immediately.

Remember that oplock keys exist on handles, and they are "put on" the handle when the handle is created. You can associate a handle with an oplock key even if no oplocks are granted.

Note It is more accurate to say that the oplock key is associated with the **FILE_OBJECT** structure that the stream handle refers to. This distinction is important when the handle is duplicated, such as with DuplicateHandle. Each of the duplicate handles refers to the same underlying **FILE_OBJECT** structure.

Granting Oplocks

4/26/2017 • 7 min to read • Edit Online

Oplocks are requested through FSCTLs. The following list shows the FSCTLs for the different oplock types (which user-mode applications and kernel-mode drivers can issue):

- FSCTL_REQUEST_OPLOCK_LEVEL_1
- FSCTL_REQUEST_OPLOCK_LEVEL_2
- FSCTL_REQUEST_BATCH_OPLOCK
- FSCTL_REQUEST_FILTER_OPLOCK
- FSCTL_REQUEST_OPLOCK

The first four FSCTLs in the list are used to request legacy oplocks. The last FSCTL is used to request Windows 7 oplocks with the REQUEST_OPLOCK_INPUT_FLAG_REQUEST flag specified in the **Flags** member of the REQUEST_OPLOCK_INPUT_BUFFER structure, passed as the *lpInBuffer* parameter of DeviceIoControl. In a similar manner, **ZwFsControlFile** can be used to request Windows 7 oplocks from kernel mode. A file system minifilter must use **FltAllocateCallbackData** and **FltPerformAsynchronousIo** to request a Windows 7 oplock. To specify which of the four Windows 7 oplocks is required, one or more of the flags OPLOCK_LEVEL_CACHE_READ, OPLOCK_LEVEL_CACHE_HANDLE, or OPLOCK_LEVEL_CACHE_WRITE is set in the **RequestedOplockLevel** member of the REQUEST_OPLOCK_INPUT_BUFFER structure. For more information, see **FSCTL_REQUEST_OPLOCK**.

When a request is made for an oplock and the oplock can be granted, the file system returns STATUS_PENDING (because of this, oplocks are never granted for synchronous I/O). The FSCTL IRP does not complete until the oplock is broken. If the oplock cannot be granted, an appropriate error code is returned. The most commonly returned error codes are STATUS_OPLOCK_NOT_GRANTED and STATUS_INVALID_PARAMETER (and their equivalent user-mode analogs).

As mentioned previously, the Filter oplock allows an application to "back out" when other applications/clients try to access the same stream. This mechanism allows an application to access a stream without causing other accessors of the stream to receive sharing violations when attempting to open the stream. To avoid sharing violations, a special three-step procedure should be used to request a Filter oplock (FSCTL_REQUEST_FILTER_OPLOCK):

1. Open the file with a required access of FILE_READ_ATTRIBUTES and a share mode of FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE.
2. Request a Filter oplock on the handle from step 1.
3. Open the file again for read access.

The handle opened in step 1 will not cause other applications to receive sharing violations, since it is open only for attribute access (FILE_READ_ATTRIBUTES), and not data access (FILE_READ_DATA). This handle is suitable for requesting the Filter oplock, but not for performing actual I/O on the data stream. The handle opened in step 3 allows the holder of the oplock to perform I/O on the stream, while the oplock granted in step 2 allows the holder of the oplock to "get out of the way" without causing a sharing violation to another application that attempts to access the stream.

The NTFS file system provides an optimization for this procedure through the FILE_RESERVE_OPFILTER create option flag. If this flag is specified in step 1 of the previous procedure, it allows the file system to fail the create request with STATUS_OPLOCK_NOT_GRANTED if the file system can determine that step 2 will fail. Be aware that if step 1 succeeds, there is no guarantee that step 2 will succeed, even if FILE_RESERVE_OPFILTER was specified for the create request.

The following table identifies the required conditions necessary to grant an oplock.

REQUEST TYPE	CONDITIONS
Level 1 Filter Batch	Granted only if all of the following conditions are true: • The request is for a given stream of a file. ◦ If a directory, STATUS_INVALID_PARAMETER is returned. • The stream is opened for ASYNCHRONOUS access. ◦ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned (oplocks are not granted for synchronous I/O requests). • There are no TxF transactions on any stream of the file. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. • There are no other opens on the stream (even by the same thread). ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. Be aware that if the current oplock state is: • No Oplock: The request is granted. • Level 2: The original Level 2 request is broken with FILE_OPLOCK_BROKEN_TO_NONE. The requested exclusive oplock is then granted. • Level 1, Batch, Filter, Read, Read-Handle, Read-Write, or Read-Write-Handle: STATUS_OPLOCK_NOT_GRANTED is returned.

REQUEST TYPE	CONDITIONS
Level 2	Granted only if all of the following conditions are true: • The request is for a given stream of a file. ◦ If a directory, STATUS_INVALID_PARAMETER is returned. • The stream is opened for ASYNCHRONOUS access. ◦ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned. • There are no TxF transactions on the file. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. • There are no current Byte Range Locks on the stream. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. ◦ Be aware that prior to Windows 7, the operating system verifies if a byte range lock ever existed on the stream since the last time it was opened, and fails the request if so. Be aware that if the current oplock state is: • No Oplock: The request is granted. • Level 2 and/or Read: The request is granted. You can have multiple Level 2/Read oplocks granted on the same stream at the same time. Multiple Level 2 (but not Read) oplocks can even exist on the same handle. ◦ If a Read oplock is requested on a handle that already has a Read oplock granted to it, the first Read oplock's IRP is completed with STATUS_OPLOCK_SWITCHED_TO_NEW_HANDLE before the second Read oplock is granted. • Level 1, Batch, Filter, Read-Handle, Read-Write, Read-Write-Handle: STATUS_OPLOCK_NOT_GRANTED is returned.

REQUEST TYPE	CONDITIONS
Read	Granted only if all of the following conditions are true: • The request is for a given stream of a file. ◦ If a directory, STATUS_INVALID_PARAMETER is returned. • The stream is opened for ASYNCHRONOUS access. ◦ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned. • There are no TxF transactions on the file. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. • There are no current Byte Range Locks on the stream. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. Be aware that if the current oplock state is: • No Oplock: The request is granted. • Level 2 and/or Read: The request is granted. You can have multiple Level 2/Read oplocks granted on the same stream at the same time. ◦ Additionally, if an existing oplock has the same oplock key as the new request, its IRP is completed with STATUS_OPLOCK_SWITCHED_TO_NEW_HANDLE. • Read-Handle and the existing oplock have a different oplock key from the new request: The request is granted. Multiple Read and Read-Handle oplocks can coexist on the same stream (see the note following this table). ◦ Else (oplock keys are the same) STATUS_OPLOCK_NOT_GRANTED is returned. • Level 1, Batch, Filter, Read-Write, Read-Write-Handle: STATUS_OPLOCK_NOT_GRANTED is returned.

REQUEST TYPE	CONDITIONS
Read-Handle	Granted only if all of the following conditions are true: • The request is for a given stream of a file. ◦ If a directory, STATUS_INVALID_PARAMETER is returned. • The stream is opened for ASYNCHRONOUS access. ◦ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned. • There are no TxF transactions on the file. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. • There are no current Byte Range Locks on the stream. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. Be aware that if the current oplock state is: • No Oplock: the request is granted. • Read: the request is granted. ◦ If an existing Read oplock has the same oplock key as the new request, its IRP is completed with STATUS_OPLOCK_SWITCHED_TO_NEW_HANDLE. This means that the oplock is upgraded from Read to Read-Handle. ◦ Any existing Read oplock that does not have the same oplock key as the new request remains unchanged. • Level 2, Level 1, Batch, Filter, Read-Write, Read-Write-Handle: STATUS_OPLOCK_NOT_GRANTED is returned.

REQUEST TYPE	CONDITIONS
Read-Write	Granted only if all of the following conditions are true: • The request is for a given stream of a file. ◦ If a directory, STATUS_INVALID_PARAMETER is returned. • The stream is opened for ASYNCHRONOUS access. ◦ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned. • There are no TxF transactions on the file. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. • If there are other opens on the stream (even by the same thread) they must have the same oplock key. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. Be aware that if the current oplock state is: • No Oplock: the request is granted. • Read or Read-Write and the existing oplock has the same oplock key as the request: the existing oplock's IRP is completed with STATUS_OPLOCK_SWITCHED_TO_NEW_HANDLE, the request is granted. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. • Level 2, Level 1, Batch, Filter, Read-Handle, Read-Write-Handle: STATUS_OPLOCK_NOT_GRANTED is returned.

REQUEST TYPE	CONDITIONS
Read-Write-Handle	Granted only if all of the following are true: • The request is for a given stream of a file. ◦ If a directory, STATUS_INVALID_PARAMETER is returned. • The stream is opened for ASYNCHRONOUS access. ◦ If opened for SYNCHRONOUS access, STATUS_OPLOCK_NOT_GRANTED is returned. • There are no TxF transactions on the file. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. • If there are other open requests on the stream (even by the same thread) they must have the same oplock key. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. Be aware that if the current oplock state is: • No Oplock: the request is granted. • Read, Read-Handle, Read-Write, or Read-Write-Handle and the existing oplock has the same oplock key as the request: the existing oplock's IRP is completed with STATUS_OPLOCK_SWITCHED_TO_NEW_HANDLE, the request is granted. ◦ Else STATUS_OPLOCK_NOT_GRANTED is returned. • Level 2, Level 1, Batch, Filter: STATUS_OPLOCK_NOT_GRANTED is returned.

Note Read and Level 2 oplocks may coexist on the same stream, and Read and Read-Handle oplocks may coexist, but Level 2 and Read-Handle oplocks may not coexist.

Breaking Oplocks

4/26/2017 • 4 min to read • Edit Online

After an oplock is granted, the owner of that oplock has access to the stream (based on the type of oplock that was requested). If the operation received is not compatible with the current oplock, the oplock is broken.

When an oplock is granted, the requesting IRP is pended. When an oplock is broken, the pended oplock request IRP is completed with STATUS_SUCCESS. For Level 1, Batch, and Filter oplocks the **IoStatus.Information** member of the IRP is set to indicate the level to which the oplock is breaking. These levels are:

- FILE_OPLOCK_BROKEN_TO_NONE - The oplock was broken and there is no current oplock on the stream. The oplock is said to have been "broken to None".
- FILE_OPLOCK_BROKEN_TO_LEVEL_2 - The current oplock (Level 1 or Batch) was converted to a Level 2 oplock. Note that Filter oplocks never break to Level 2, they always break to None.

For Read-Handle, Read-Write, and Read-Write-Handle oplocks, the level to which the oplock is breaking is described as a combination of zero or more of the flags OPLOCK_LEVEL_CACHE_READ, OPLOCK_LEVEL_CACHE_HANDLE, or OPLOCK_LEVEL_CACHE_WRITE in the **NewOplockLevel** member of the REQUEST_OPLOCK_OUTPUT_BUFFER structure passed as the *lpOutBuffer* parameter of DeviceIoControl. In a similar manner, **FltFsControlFile** and **ZwFsControlFile** can be used to request Windows 7 oplocks from kernel mode. For more information, see **FSCTL_REQUEST_OPLOCK**.

When breaking a Level 1, Batch, Filter, Read-Write, Read-Write-Handle, or, under certain circumstances (see note), a Read-Handle oplock, the pended oplock request IRP is completed by the oplock package and the operation that caused the oplock break is itself pended (note that if the operation is issued on a synchronous handle, or it is an IRP_MJ_CREATE, which is always synchronous, the I/O manager causes the operation to block, rather than return STATUS_PENDING), waiting for an acknowledgment from the owner of the oplock to tell the oplock package that they have finished their processing and it is safe for the pended operation to proceed. The purpose of this delay is to allow the owner of the oplock to put the stream back into a consistent state before the current operation proceeds. The system waits forever to receive the acknowledgment as there is no timeout. It is therefore incumbent on the owner of the oplock to acknowledge the break in a timely manner. The pended operation's IRP is set into a cancelable state. If the application or driver performing the wait terminates, the oploack package immediately completes the IRP with STATUS_CANCELLED.

An IRP_MJ_CREATE IRP may specify the FILE_COMPLETE_IF_OPLOCKED create option to avoid being blocked as part of oplock break acknowledgment. This option tells the oplock package not to block the create IRP until the oplock break acknowledgment is received. Instead, the create is allowed to proceed. If a successful create results in an oplock break, the return code is STATUS_OPLOCK_BREAK_IN_PROGRESS, rather than STATUS_SUCCESS. The FILE_COMPLETE_IF_OPLOCKED flag is typically used to avoid deadlocks. For example, if a client owns an oplock on a stream and the same client subsequently opens the same stream, the client would block waiting for itself to acknowledge the oplock break. In this scenario, use of the FILE_COMPLETE_IF_OPLOCKED flag avoids the deadlock.

Because the NTFS file system initiates oplock breaks for Batch and Filter oplocks before checking for sharing violations, it is possible for a create that specified FILE_COMPLETE_IF_OPLOCKED to fail with STATUS_SHARING_VIOLATION but still cause a Batch or Filter oplock to break. In this case, the information member of the **IO_STATUS_BLOCK** structure is set to FILE_OPBATCH_BREAK_UNDERWAY to allow the caller to detect this case.

For Read-Handle and Read-Write-Handle oplocks, the oplock break is initiated after NTFS checks for and detects a sharing violation. This gives holders of these oplocks the opportunity to close their handles and get out of the way, thus allowing for the possibility of not returning the sharing violation to the user. It also avoids unconditionally breaking the oplock in cases where the handle that the oplock caches does not conflict with the new create.

When Level 2, Read, and, under certain circumstances (see note), Read-Handle oplocks break, the system does not wait for an acknowledgment. This is because there should be no cached state on the stream that needs to be restored to the file before allowing other clients access to it.

There are certain file system operations which check the current oplock state to determine if the oplock needs to be broken. The following sections list each operation and describe what triggers an oplock break, what determines the level to which the oplock breaks, and whether an acknowledgment of the break is required:

- IRP_MJ_CREATE
- IRP_MJ_READ
- IRP_MJ_WRITE
- IRP_MJ_CLEANUP
- IRP_MJ_LOCK_CONTROL
- IRP_MJ_SET_INFORMATION
- IRP_MJ_FILE_SYSTEM_CONTROL

A break of a Windows 7 oplock requires an acknowledgment if the REQUEST_OPLOCK_OUTPUT_FLAG_ACK_REQUIRED flag is set in the **Flags** member of the REQUEST_OPLOCK_OUTPUT_BUFFER structure passed as the output parameter of DeviceIoControl(*lpOutBuffer*), **FltFsControlFile**(*OutBuffer*) or **ZwFsControlFile**(*OutBuffer*). For more information, see **FSCTL_REQUEST_OPLOCK**.

Note The above listed per-operation topics describe the details of when a break of a Read-Handle oplock results in the pending of the operation that broke the oplock. For example, the IRP_MJ_CREATE topic contains the associated Read-Handle details.

Checking the Oplock State of an IRP_MJ_CREATE operation

4/26/2017 • 5 min to read • Edit Online

The following only applies when an existing stream of a file is being opened (that is, newly created streams cannot have pre-existing oplocks on them).

Note When processing IRP_MJ_CREATE for any oplock, if the desired access contains nothing other than FILE_READ_ATTRIBUTES, FILE_WRITE_ATTRIBUTES, or SYNCHRONIZE, the oplock does not break unless FILE_RESERVE_OPFILTER is specified. Specifying FILE_RESERVE_OPFILTER always results in an oplock break if the create succeeds. For brevity and simplicity, the following table omits the foregoing, since it applies to all oplocks.

REQUEST TYPE	CONDITIONS
Level 1	Broken on IRP_MJ_CREATE when: • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.

	If the oplock is broken:
	• Break to None **IF**:
	  ◦ The FILE_RESERVE_OPFILTER flag is set
	  OR
	  ◦ Any of the following create disposition values are specified:
	    ◦ FILE_SUPERSEDE
	    ◦ FILE_OVERWRITE
	    ◦ FILE_OVERWRITE_IF
	  ELSE:
	  ◦ Break to Level 2.
	• An acknowledgment must be received before the operation continues.

Level 2	Broken on IRP_MJ_CREATE when:
	• The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.
	• **AND:**
	  ◦ The FILE_RESERVE_OPFILTER flag is set
	  OR
	  ◦ Any of the following create disposition values are specified:
	    ◦ FILE_SUPERSEDE
	    ◦ FILE_OVERWRITE
	    ◦ FILE_OVERWRITE_IF

	If the oplock is broken: • Break to None. • No acknowledgment is required, the operation proceeds immediately.
Batch	Broken on IRP_MJ_CREATE when: • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.

	If the oplock is broken:
	• Break to None **IF**:
	  ◦ The FILE_RESERVE_OPFILTER flag is set.
	  OR
	  ◦ Any of the following create disposition values are specified:
	    ◦ FILE_SUPERSEDE
	    ◦ FILE_OVERWRITE
	    ◦ FILE_OVERWRITE_IF
	  ELSE:
	  ◦ Break to Level 2.
	• An acknowledgment must be received before the operation continues.

Filter	Broken on IRP_MJ_CREATE when: • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock. • **AND:** ◦ A "writable" desired access was requested on the stream which was not opened for FILE_SHARE_READ access. Note that "writeable" access is defined as any attribute other than: ◦ FILE_READ_ATTRIBUTES ◦ FILE_WRITE_ATTRIBUTES ◦ FILE_READ_DATA ◦ FILE_READ_EA ◦ FILE_EXECUTE ◦ SYNCHRONIZE ◦ READ_CONTROL
	If the oplock is broken: • Break to None. • An acknowledgment must be received before the operation continues.
Read	Broken on IRP_MJ_CREATE when: • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock. • **AND:** ◦ The FILE_RESERVE_OPFILTER flag is set **OR** ◦ Any of the following create disposition values are specified: ◦ FILE_SUPERSEDE ◦ FILE_OVERWRITE ◦ FILE_OVERWRITE_IF
	If the oplock is broken: • Break to None. • No acknowledgment is required, the operation proceeds immediately.

Read-Handle	Broken on IRP_MJ_CREATE when: • The current open conflicts with an existing open such that a sharing violation would occur. **OR** • The FILE_RESERVE_OPFILTER flag is set. **OR** • Any of the following create disposition values are specified: ◦ FILE_SUPERSEDE ◦ FILE_OVERWRITE ◦ FILE_OVERWRITE_IF **AND** (for any of the above three conditions) • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.
	If the oplock is broken: • Break to None **IF**: ◦ The FILE_RESERVE_OPFILTER flag is set. **OR** ◦ Any of the following create disposition values are specified: ◦ FILE_SUPERSEDE ◦ FILE_OVERWRITE ◦ FILE_OVERWRITE_IF **ELSE:** ◦ Break to Read. • If the oplock broke because the current open conflicts with an existing open such that a sharing violation would occur, an acknowledgment must be received before the operation continues. • If the oplock broke for any other reason, although acknowledgment of the break is required, the operation continues immediately (for example, without waiting for the acknowledgment).
Read-Write	Broken on IRP_MJ_CREATE when: • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.

	If the oplock is broken:
	• Break to None **IF**:
	  ◦ The FILE_RESERVE_OPFILTER flag is set.
	  OR
	  ◦ Any of the following create disposition values are specified:
	    ◦ FILE_SUPERSEDE
	    ◦ FILE_OVERWRITE
	    ◦ FILE_OVERWRITE_IF
	  ELSE:
	  ◦ Break to Read.
	• An acknowledgment must be received before the operation continues.

Read-Write-Handle	Broken on IRP_MJ_CREATE when: • The oplock key associated with the FILE_OBJECT on which the open is occurring is different from the oplock key associated with the FILE_OBJECT that owns the oplock.

	If the oplock is broken:
	• Break to None **IF**:
	  ◦ The FILE_RESERVE_OPFILTER flag is set.
	  OR
	  ◦ Any of the following create disposition values are specified:
	    ◦ FILE_SUPERSEDE
	    ◦ FILE_OVERWRITE
	    ◦ FILE_OVERWRITE_IF
	  ELSE:
	  ◦ Break to Read-Write if the current open conflicts with an existing open such that a sharing violation would occur. Otherwise, break to Read-Handle.
	• An acknowledgment must be received before the operation continues.

The file system performs additional checks for Batch and Filter oplocks (rather than the oplock package itself) when processing an IRP_MJ_CREATE operation, which impact whether the file system asks the oplock package to perform oplock break processing. This is a case where operations on one data stream can impact the oplocks on other data streams of the same file (that is, the last two list items of the following criteria list). If one or more of the following criteria are met, the file system sends a request to the oplock package to perform oplock break processing:

- Request a break if this is a network query open and a KTM transaction is present. Otherwise, do not request a break on network query open.
- If a SUPERSEDE, OVERWRITE or OVERWRITE_IF operation is performed on an alternate data stream and FILE_SHARE_DELETE is not specified and there is a Batch or Filter oplock on the primary data stream, request

a break of the Batch or Filter oplock on the primary data stream.

- If a SUPERSEDE, OVERWRITE or OVERWRITE_IF operation is performed on the primary data stream and DELETE access has been requested and there are Batch or Filter oplocks on any alternate data stream, request a break of the Batch or Filter oplocks on all alternate data streams that have them.

When the file system decides to ask the oplock package to perform oplock break processing, the rules laid out in the preceding table apply.

The check to break Batch and Filter oplocks occurs before the share access checks are made. This means the Batch or Filter oplock is broken even if the open request ultimately fails due to a sharing violation.

Checking the Oplock State of an IRP_MJ_READ operation

4/26/2017 • 1 min to read • Edit Online

The following only applies when a *stream* is being read. If a TxF transacted reader performs the read, this check is not made since a transacted reader excludes a writer (that is, a writer holding an oplock cannot be present at all).

REQUEST TYPE	CONDITIONS
Level 1 Batch	Broken on IRP_MJ_READ when: • The read operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to Level 2. • An acknowledgment must be received before the operation continues.
Read-Write	Broken on IRP_MJ_READ when: • The read operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to Read. • An acknowledgment must be received before the operation continues.
Read-Write-Handle	Broken on IRP_MJ_READ when: • The read operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to Read-Handle. • An acknowledgment must be received before the operation continues.

Level 2 Filter Read Read-Handle	• The oplock is not broken, no acknowledgment is required, and the operation proceeds immediately.

Checking the Oplock State of an IRP_MJ_WRITE operation

4/26/2017 • 1 min to read • Edit Online

The following only applies when a *stream* is being written and the write is not a paging I/O.

REQUEST TYPE	CONDITIONS
Level 1 Batch Filter Read-Handle Read-Write Read-Write-Handle	Broken on IRP_MJ_WRITE when: • The write operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • For the Read-Handle request: Although acknowledgment of the break is required, the operation continues immediately (for example, without waiting for the acknowledgment). • For all other request types: An acknowledgment must be received before the operation continues.
Read	Broken on IRP_MJ_WRITE when: • The write operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • No acknowledgment is required, the operation proceeds immediately.
Level 2	• Always break to None. • No acknowledgment is required, the operation proceeds immediately.

Checking the Oplock State of an IRP_MJ_CLEANUP operation

4/26/2017 • 1 min to read • Edit Online

The following only applies when a *stream* is being closed.

REQUEST TYPE	**CONDITIONS**
Level 1 Batch Filter Read-Handle Read-Write Read-Write-Handle	• Always break to None. • No acknowledgment is required, the operation proceeds immediately. Note that any I/O operations (IRPs) waiting for an acknowledgment from a pending break request are completed immediately.
Level 2 Read	• Always break to None. Note that other Level 2 or Read oplocks on the same stream are not affected; only the Level 2 or Read oplock associated with this FILE_OBJECT is broken. • No acknowledgment is required, the operation proceeds immediately.

Checking the Oplock State of an IRP_MJ_LOCK_CONTROL operation

4/26/2017 • 1 min to read • Edit Online

The following applies on every byte range lock operation on the given stream.

REQUEST TYPE	CONDITIONS
Level 1 Batch Read-Handle Read-Write Read-Write-Handle	Broken on IRP_MJ_LOCK_CONTROL when: • The lock operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • For the Handle request: Although acknowledgment of the break is required, the operation continues immediately (for example, without waiting for the acknowledgment). • For all other request types: An acknowledgment must be received before the operation continues.
Read	Broken on IIRP_MJ_LOCK_CONTROL when: • The lock operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • No acknowledgment is required, the operation proceeds immediately.
Filter	• The oplock is not broken, no acknowledgment is required, and the operation proceeds immediately.
Level 2	• Always break to None. • No acknowledgment is required, the operation proceeds immediately.

Checking the Oplock State of an IRP_MJ_SET_INFORMATION operation

4/26/2017 • 2 min to read • Edit Online

Certain IRP_MJ_SET_INFORMATION operations check oplock state. The following six operations perform this check:

FileEndOfFileInformation, FileAllocationInformation, and FileValidDataLengthInformation

This information applies when the following operations are being performed on a file or stream:

- A caller attempts to change the logical size of the stream. Note that when the cache manager's lazy writer thread attempts to set a new end of file, no oplock check is made. This is because the check is made previously when the real write request is received.
- A caller attempts to change the allocated size of the stream.

REQUEST TYPE	CONDITIONS
Level 1 Batch Filter Read-Handle Read-Write Read-Write-Handle	Broken on IRP_MJ_SET_INFORMATION (for FileEndOfFileInformtion, FileAllocationInformation, and FileValidDataLengthInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • For the Read-Handle request: Although acknowledgment of the break is required, the operation continues immediately (i.e., without waiting for the acknowledgment). • For all other request types: An acknowledgment must be received before the operation continues.
Read	Broken on IRP_MJ_SET_INFORMATION (for FileEndOfFileInformtion, FileAllocationInformation, and FileValidDataLengthInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • No acknowledgment is required, the operation proceeds immediately.

Level 2	• Always break to None. • No acknowledgment is required, the operation proceeds immediately.

FileRenameInformation, FileShortNameInformation, and FileLinkInformation

This information applies when the following operations are being performed on a file or stream:

- The file or stream is being renamed.
- A short name is being set for the file.
- A hard link is being created for the file. This affects oplock state if the new hard link is superseding an existing link to a different file, and the oplock exists on the link being superseded.
- An ancestor directory of the stream on which the oplock exists is being renamed, or the ancestor directory's short name is being set.

REQUEST TYPE	CONDITIONS
Batch Filter	Broken on IRP_MJ_SET_INFORMATION (for FileRenameInformation, FileShortNameInformation, and FileLinkInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to None. • An acknowledgment must be received before the operation continues.
Read-Handle	Broken on IRP_MJ_SET_INFORMATION (for FileRenameInformation, FileShortNameInformation, and FileLinkInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.
	If the oplock is broken: • Break to Read. • An acknowledgment must be received before the operation continues.
Read-Write-Handle	Broken on IRP_MJ_SET_INFORMATION (for FileRenameInformation, FileShortNameInformation, and FileLinkInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock.

	If the oplock is broken: • Break to Read-Write. • An acknowledgment must be received before the operation continues.
Level 1 Level 2 Read Read-Write	• The oplock is not broken, no acknowledgment is required, and the operation proceeds immediately.

FileDispositionInformation

This information applies when a caller tries to delete the file.

REQUEST TYPE	CONDITIONS
Read-Handle	Broken on IRP_MJ_SET_INFORMATION (for FileDispositionInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock. **AND** • **FILE_DISPOSITION_INFORMATION**.DeleteFile is **TRUE**.
	If the oplock is broken: • Break to Read. • An acknowledgment must be received before the operation continues.
Read-Write-Handle	Broken on IRP_MJ_SET_INFORMATION (for FileDispositionInformation) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock. **AND** • **FILE_DISPOSITION_INFORMATION**.DeleteFile is **TRUE**.
	If the oplock is broken: • Break to Read-Write. • An acknowledgment must be received before the operation continues.

Checking the Oplock State of IRP_MJ_FILE_SYSTEM_CONTROL

4/26/2017 • 1 min to read • Edit Online

Certain IRP_MJ_FILE_SYSTEM_CONTROL operations check oplock state. The following operation(s) perform this check:

- **FSCTL_SET_ZERO_DATA**

This information applies when a caller wants to zero the current contents of the given stream.

REQUEST TYPE	CONDITIONS
Level 1 Batch Filter Read-Handle Read-Write Read-Write-Handle	Broken on IRP_MJ_FILE_SYSTEM_CONTROL (for FSCTL_SET_ZERO_DATA) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock. If the oplock is broken: • Break to None • For the Read-Handle request: Although acknowledgment of the break is required, the operation continues immediately (for example, without waiting for the acknowledgment). • For all other request types: An acknowledgment must be received before the operation continues.
Read	Broken on IRP_MJ_FILE_SYSTEM_CONTROL (for FSCTL_SET_ZERO_DATA) when: • The operation occurs on a FILE_OBJECT with a different oplock key from the FILE_OBJECT which owns the oplock. If the oplock is broken: • Break to None • No acknowledgment is required, the operation proceeds immediately.
Level 2	• Always break to None. • No acknowledgment is required, the operation proceeds immediately.

Acknowledging Oplock Breaks

4/26/2017 • 1 min to read • Edit Online

There are different types of acknowledgments that the owner of an oplock can return. Similar to the grant requests, these acknowledgments are sent as file system control codes (that is, FSCTLs). They are:

- FSCTL_OPLOCK_BREAK_ACKNOWLEDGE
  - This FSCTL indicates that the oplock owner has completed stream synchronization and they accept the level to which the oplock was broken (either Level 2 or None).
- FSCTL_OPLOCK_BREAK_ACK_NO_2
  - This FSCTL indicates that the oplock owner has completed stream synchronization but does not want a Level 2 oplock. Instead, the oplock should be broken to None (that is, the oplock is to be relinquished entirely).
- FSCTL_OPBATCH_ACK_CLOSE_PENDING
  - For a Level 1 oplock, this FSCTL indicates that the oplock owner has completed stream synchronization and is relinquishing the oplock entirely (no Level 2 oplock may result from this acknowledgment).
  - For a Batch or Filter oplock, this FSCTL indicates that the oplock owner intends to close the stream handle on which the oplock was granted. Operations blocked, awaiting acknowledgment of the oplock break, continue to wait until the oplock owner's handle is closed.
- FSCTL_REQUEST_OPLOCK
  - By specifying REQUEST_OPLOCK_INPUT_FLAG_ACK in the **Flags** member of the REQUEST_OPLOCK_INPUT_BUFFER structure passed as the *lpInBuffer* parameter of DeviceIoControl, this FSCTL is used to acknowledge breaks of Windows 7 oplocks. The acknowledgment is required only if the REQUEST_OPLOCK_OUTPUT_FLAG_ACK_REQUIRED flag is set in the **Flags** member of the REQUEST_OPLOCK_OUTPUT_BUFFER structure passed as the *lpOutBuffer* parameter of DeviceIoControl. In a similar manner, **FltFsControlFile** and **ZwFsControlFile** can be used to acknowledge Windows 7 oplocks from kernel-mode. For more information, see **FSCTL_REQUEST_OPLOCK**.

A related FSCTL code is FSCTL_OPLOCK_BREAK_NOTIFY. This code is used when the caller wants to be notified when an oplock break on the given stream completes. This call may block. When the FSCTL_OPLOCK_BREAK_NOTIFY call returns STATUS_SUCCESS, this signifies one of the following:

- No oplock granted.
- No oplock break was in progress at the time of the call.
- Any oplock break that was in progress is now complete.

To send an acknowledgment when no acknowledgment is expected is an error and the acknowledgment FSCTL IRP is failed with STATUS_INVALID_OPLOCK_PROTOCOL.

Closing the handle of the file for which the oplock break is requested will implicitly acknowledge the break. In the case of an oplock break for a sharing violation, the oplock holder can close the file handle, which acknowledges the oplock break, and prevent a sharing violation for the other user of the file.

Kernel Network Redirector Driver Components

4/26/2017 • 1 min to read • Edit Online

This section discusses the various components that are important to developers who are implementing kernel-mode network redirector drivers for the client of a remote file system. This section includes:

The RDBSS Driver and Library

The Kernel Network Mini-Redirector Driver

The RDBSS Driver and Library

4/26/2017 • 2 min to read • Edit Online

The Redirected Drive Buffering Subsystem (RDBSS) is implemented in two forms:

- A file system driver (*rdbss.sys*) supplied with the operating system.
- A static library (*rdbsslib.lib*) supplied with the Windows Driver Kit (WDK).

The *rdbss.sys* driver is automatically loaded if any non-monolithic network mini-redirectors are registered on the system. The Microsoft Server Message Block (SMB) redirector (*mrxsmb sys*) is currently the only driver that can be built as a non-monolithic network mini-redirector driver.

All other network mini-redirector drivers, including other Microsoft network mini-redirectors supplied with the operating system, must be implemented as monolithic drivers that link with the *rdbsslib.lib* static library provided with the WDK.

The RDBSS uses a well-defined mechanism for communication with network mini-redirector drivers, I/O Manager, Cache Manager, Memory Manager, and other kernel systems.

RDBSS exports a large number of routines that can be called by a network mini-redirector and other kernel systems to set options and perform various operations. To call the routines exported by RDBSS, a network mini-redirector driver (or other kernel driver) includes the appropriate WDK header files, calls the exported RDBSS routine by name, and links with the appropriate *rdbsslib.lib* file installed with the WDK. Note that different *rdbsslib.lib* files are provided with the WDK for Window Vista, Windows Server 2003, Windows XP, and Windows 2000.

The WDK header files for RDBSS also define a number of macros that are recommended for use by network mini-redirector drivers, rather than calling some of the RDBSS routines directly.

All of the data structures defined and used by RDBSS have a special 4-byte signature at the beginning of the data structure that is used extensively in validation. The values for these RDBSS data structures signatures are defined in the WDK header file, *nodetype.h*. These data structure signatures are used for troubleshooting and debugging RDBSS and network mini-redirector drivers.

The following sections discuss in detail each of the categories of routines exported by RDBSS and the macros defined to call these routines. We begin with a list of all of the routines provided by RDBSS and a similar list of macros defined by RDBSS:

- Routines Provided by RDBSS
- Macros Defined by RDBSS

The routines exported by RDBSS and the RDBSS macros defined to call these routines can be organized into a number of different categories, including the following:

- Driver Registration and Start/Stop Control
- Pool Allocation and Free Routines
- Timer and Worker Thread Management
- Work Queue Dispatching Mechanisms
- Diagnostics and Debugging
- Logging Routines and Macros

- Miscellaneous Routines
- RX_CONTEXT and IRP Management
- Connection and File Structure Management
- FCB Resource Synchronization
- Buffering State Control
- Low I/O Routines
- Name Cache Management
- Prefix Table Management
- Purging and Scavenging Control
- Multiplex ID Management
- Connection Engine Management

Routines Provided by RDBSS

4/26/2017 • 16 min to read • Edit Online

The following routines are exported by RDBSS.

ROUTINE	DESCRIPTION
RxAcquireExclusiveFcbResourceInMRx	This resource acquisition routine acquires the File Control Block (FCB) resource in exclusive mode. This routine will wait for the FCB resource to be free, so this routine does not return control until the resource has been acquired. This routine acquires the FCB resource even if the RX_CONTEXT associated with this FCB has been canceled.
RxAcquireSharedFcbResourceInMRx	This resource acquisition routine acquires the FCB resource in shared mode. This routine will wait for the FCB resource to be free if it was previously acquired exclusively, so this routine does not return control until the resource has been acquired. This routine acquires the FCB resource even if the RX_CONTEXT associated with this FCB has been canceled.
RxAcquireSharedFcbResourceInMRxEx	This resource acquisition routine acquires the FCB resource in shared mode. This routine will wait for the FCB resource to be freed if it was previously acquired exclusively. This routine does not return control until the resource has been acquired. This routine acquires the FCB resource even if the RX_CONTEXT associated with this FCB has been canceled. This routine is only available on Windows Server 2003 Service Pack 1 (SP1) and later.
RxAssert	This routine sends an assert string in RDBSS checked builds to a kernel debugger if one is installed.
RxAssociateContextWithMid	This routine associates the supplied opaque context with an available multiplex ID (MID) from a MID_ATLAS data structure.
RxCancelTimerRequest	This routine cancels a timer request. The request to be canceled is identified by the routine and context.
RxCeAllocateIrpWithMDL	This routine allocates an IRP for use by the connection engine and associates an MDL with the IRP. This routine is only available on Windows XP.

ROUTINE	DESCRIPTION
RxCeBuildAddress	This routine associates a transport address with a transport binding.
RxCeBuildConnection	This routine establishes a connection between a local RDBSS connection address and a given remote address. This routine should be called in the context of a system worker thread.
RxCeBuildConnectionOverMultipleTransports	This routine establishes a connection between a local RDBSS connection address and a given remote address and supports multiple transports. A set of local addresses are specified and this routine attempts to connect to the target server using all of the transports associated with the local addresses. One connection is chosen as the winner depending on the connection options. This routine must be called in the context of a system worker thread.
RxCeBuildTransport	This routine binds an RDBSS transport to a specified transport name.
RxCeBuildVC	This routine adds a virtual circuit to a specified connection.
RxCeCancelConnectRequest	This routine cancels a previously issued connection request. Note that this routine is not currently implemented.
RxCeFreeIrp	This routine frees an IRP used by the connection engine. This routine is only available on Windows XP.
RxCeInitiateVCDisconnect	This routine initiates a disconnect on the virtual circuit. This routine must be called in the context of a system worker thread.
RxCeQueryAdapterStatus	This routine returns the ADAPTER_STATUS structure for a given transport.
RxCeQueryInformation	This routine queries for information about a given virtual circuit.
RxCeQueryTransportInformation	This routine queries a given transport for information about the connection count and quality of service.
RxCeSend	This routine sends a transport service data unit (TSDU) along the specified connection on a virtual circuit.

ROUTINE	DESCRIPTION
RxCeSendDatagram	This routine sends a TSDU to a specified transport address.
RxCeTearDownAddress	This routine removes a transport address from a transport binding.
RxCeTearDownConnection	This routine tears down a given connection.
RxCeTearDownTransport	This routine unbinds from the transport specified.
RxCeTearDownVC	This routine tears down a virtual connection.
RxChangeBufferingState	This routine is called to process a buffering state change request.
RxCompleteRequest	This routine is used to complete an IRP associated with an RX_CONTEXT structure. This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.
RxCompleteRequest_Real	This routine is used to complete an IRP associated with an RX_CONTEXT structure. This routine should not be used by network mini-redirectors.
RxCreateMidAtlas	This routine allocates a new instance of a MID_ATLAS data structure and initializes it. RDBSS uses the multiplex ID (MID) defined in this data structure as a way that both the network client (mini-redirector) and the server can distinguish between the concurrently active requests on any connection.
RxCreateNetFcb	This routine allocates, initializes, and inserts a new FCB structure into the in-memory data structures for a NET_ROOT structure on which this FCB is opened. The structure allocated has space for a SRV_OPEN and an FOBX structure. This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.
RxCreateNetFobx	This routine allocates, initializes, and inserts a new file object extension (FOBX) structure. Network mini-redirectors should call this routine to create an FOBX at the end of a successful create operation.

ROUTINE	DESCRIPTION
RxCreateNetRoot	This routine builds a node representing a NET_ROOT structure and inserts the name into the net name table on the associated device object. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxCreateRxContext	This routine allocates a new RX_CONTEXT structure and initializes the data structure.
RxCreateSrvCall	This routine builds a node that represents a server call context and inserts the name into the net name table maintained by RDBSS. This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.
RxCreateSrvOpen	This routine allocates, initializes, and inserts a new SRV_OPEN structure into the in-memory data structures used by RDBSS. If a new structure has to be allocated, it has space for an FOBX structure. This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.
RxCreateVNetRoot	This routine builds a node that represents a V_NET_ROOT structure and inserts the name into the net name table. This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.
RxDbgBreakPoint	This routine raises an exception that is handled by the kernel debugger if one is installed; otherwise, it is handled by the debug system.
RxDereference	This routine decrements the reference count on an instance of several of the reference-counted data structures used by RDBSS.
RxDereferenceAndDeleteRxContext_Real	This routine dereferences an RX_CONTEXT structure and if the reference count goes to zero, then it deallocates and removes the specified RX_CONTEXT structure from the RDBSS in-memory data structures.
RxDestroyMidAtlas	This routine destroys an existing instance of a MID_ATLAS data structure and frees the memory allocated.
RxDispatchToWorkerThread	This routine invokes a routine in the context of a worker thread.

ROUTINE	DESCRIPTION
RxDriverEntry	This routine is called by a monolithic network mini-redirector driver from its **DriverEntry** to initialize RDBSS. For non-monolithic drivers, this initialization routine is equivalent to the **DriverEntry** routine of the *rdbss.sys* device driver.
RxFinalizeConnection	This routine deletes a connection to a share. Any files open on the connection are closed depending on the level of force specified. The network mini-redirector might choose to keep the transport connection open for performance reasons, unless some option is specified to force a close of connection.
RxFinalizeNetFcb	This routine finalizes the given FCB structure. The caller must have an exclusive lock on the NET_ROOT structure associated with FCB. This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.
RxFinalizeNetFobx	This routine finalizes the given FOBX structure. The caller must have an exclusive lock on the FCB associated with this FOBX. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxFinalizeNetRoot	This routine finalizes the given NET_ROOT structure. The caller should have exclusive access to the lock on the NetName table of the device object associated with this NET_ROOT structure (through the SRV_CALL structure). This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.
RxFinalizeSrvCall	This routine finalizes the given SRV_CALL structure. The caller should have exclusive access to the lock on the NetName table of the device object associated with this SRV_CALL structure. This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.
RxFinalizeSrvOpen	This routine finalizes the given SRV_OPEN structure. This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.
RxFinalizeVNetRoot	This routine finalizes the given V_NET_ROOT structure. The caller must have exclusive access to the lock on the NetName table of the device object associated with this V_NET_ROOT structure. This routine is used internally by RDBSS and should not be used by network mini-redirectors.

ROUTINE	DESCRIPTION
RxFinishFcbInitialization	This routine is used to finish initializing an FCB after the successful completion of a create operation by the network mini-redirector.
RxForceFinalizeAllVNetRoots	This routine force finalizes all of the V_NET_ROOT structures associated with a given NET_ROOT structure. The caller must have exclusive access to the lock on the NetName table of the device object associated with this V_NET_ROOT structure. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxFsdDispatch	This routine implements the file system driver (FSD) dispatch for RDBSS to process an I/O request packet (IRP). This routine is called by a network mini-redirector in the driver dispatch routines to initiate RDBSS processing of a request.
RxFsdPostRequest	This routine queues the I/O request packet (IRP) specified by an RX_CONTEXT structure to the worker queue for processing by the file system process (FSP).
RxGetFileSizeWithLock	This routine gets the file size from the FCB structure using a lock to ensure that the 64-bit value is read consistently.
RxGetRDBSSProcess	This routine returns a pointer to the process of the main thread used by the RDBSS kernel process.
RxIndicateChangeOfBufferingState	This routine is called to register a buffering state change request (an oplock break indication, for example) for later processing.
RxIndicateChangeOfBufferingStateForSrvOpen	This routine is called to register a buffering state change request (an oplock break indication, for example) for later processing.
RxInferFileType	This routine tries to infer the file type (directory or non-directory) from the **CreateOptions** (**Create.NtCreateParameters.CreateOptions**) field in the RX_CONTEXT structure.
RxInitializeContext	This routine initializes a newly allocated RX_CONTEXT structure.
RxIsThisACscAgentOpen	This routine determines if a file open was made by a user-mode client-side caching agent. This routine is only available on Windows Server 2003.

ROUTINE	DESCRIPTION
RxLockEnumerator	This routine is called from a network mini-redirector to enumerate the file locks on an FCB.
RxLogEventDirect	This routine is called to log an error to the I/O error log. It is recommended that the **RxLogEvent** or **RxLogFailure** macro be used instead of calling this routine directly.
RxLogEventWithAnnotation	This routine allocates an I/O error log structure, fills in the log structure, and writes this structure to the I/O error log.
RxLogEventWithBufferDirect	This routine is called to log an error to an I/O error log. This routine encodes the line number and status into the data buffer stored in the I/O error log structure.
RxLowIoCompletion	This routine must be called by the low I/O routines of a network mini-redirector driver when processing is complete, if the routine initially returned pending.
RxLowIoGetBufferAddress	This routine returns the buffer that corresponds to the MDL from the **LowIoContext** structure of an RX_CONTEXT structure.
RxMakeLateDeviceAvailable	This routine modifies the device object to make a "late device" available. A late device is one that is not created in the driver's load routine.
RxMapAndDissociateMidFromContext	This routine maps a MID to its associated context in a MID_ATLAS data structure and then disassociates the MID from the context.
RxMapMidToContext	This routine maps a MID to its associated context in a MID_ATLAS data structure.
RxMapSystemBuffer	This routine returns the system buffer address from the I/O request packet (IRP).
RxNameCacheActivateEntry	This routine takes a name cache entry and updates the expiration time and the network mini-redirector context. It then puts the entry on the active list.
RxNameCacheCheckEntry	This routine checks a NAME_CACHE entry for validity.

ROUTINE	DESCRIPTION
RxNameCacheCreateEntry	This routine allocates and initializes a NAME_CACHE structure with the given name string. It is expected that the caller will then initialize any additional network mini-redirector elements of the name cache context and then put the entry on the name cache active list.
RxNameCacheExpireEntry	This routine puts a NAME_CACHE entry on the free list.
RxNameCacheExpireEntryWithShortName	This routine expires all of the NAME_CACHE entries whose name prefix matches the given short file name.
RxNameCacheFetchEntry	This routine looks for a match with a specified name string for a NAME_CACHE entry.
RxNameCacheFinalize	This routine releases the storage for all of the NAME_CACHE entries associated with a NAME_CACHE_CONTROL structure.
RxNameCacheFreeEntry	This routine releases the storage for a NAME_CACHE entry and decrements the count of NAME_CACHE cache entries associated with a NAME_CACHE_CONTROL structure.
RxNameCacheInitialize	This routine initializes a NAME_CACHE structure and associates it with a NAME_CACHE_CONTROL structure.
RxNewMapUserBuffer	This routine returns the address of the user buffer used for low I/O. This routine is only available on Windows XP and Windows 2000.
RxpAcquirePrefixTableLockExclusive	This routine acquires an exclusive lock on a prefix table used to catalog SRV_CALL and NET_ROOT names. This routine is only available on Windows XP and Windows 2000. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxpAcquirePrefixTableLockShared	This routine acquires a shared lock on a prefix table used to catalog SRV_CALL and NET_ROOT names. This routine is only available on Windows XP and Windows 2000. This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.

ROUTINE	DESCRIPTION
RxpDereferenceAndFinalizeNetFcb	This routine dereferences the reference count and finalizes an FCB. This routine is only available on Windows Server 2003 Service Pack 1 (SP1) and later.
RxpDereferenceNetFcb	This routine decrements the reference count on an FCB.
RxPostOneShotTimerRequest	This routine is used by drivers to initialize a one-shot timer request. The worker thread routine passed to this routine is called once when the timer expires.
RxPostRecurrentTimerRequest	This routine is used to initialize a recurrent timer request. The worker thread routine passed to this routine is called at regular intervals when the recurrent timer fires based on the input parameters to this routine.
RxPostToWorkerThread	This routine invokes the routine in the context of a worker thread.
RxpReferenceNetFcb	This routine increments the reference count on an FCB.
RxPrefixTableLookupName	The routine looks up a name in a prefix table used to catalog SRV_CALL and NET_ROOT names and converts from the underlying pointer to the containing structure. This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.
RxpReleasePrefixTableLock	This routine releases a lock on a prefix table used to catalog SRV_CALL and NET_ROOT names. This routine is only available on Windows XP and Windows 2000. This routine is used internally by RDBSS and should not be used by network mini-redirector drivers.
RxPrepareContextForReuse	This routine prepares an RX_CONTEXT structure for reuse by resetting all operation-specific allocations and acquisitions that have been made. The parameters obtained from the IRP are not modified. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxPrepareToReparseSymbolicLink	This routine sets up the file object name to facilitate a reparse. This routine is used by the network mini-redirectors to traverse symbolic links. This routine is used internally by RDBSS and should not be used by network mini-redirectors.

ROUTINE	DESCRIPTION
RxpTrackDereference	This routine is used to track requests to dereference SRV_CALL, NET_ROOT, V_NET_ROOT, FOBX, FCB, and SRV_OPEN structures in checked builds. A log of these dereference requests can be accessed by the logging system and WMI. For retail builds, this routine does nothing.
RxpTrackReference	This routine is used to track requests to reference SRV_CALL, NET_ROOT, V_NET_ROOT, FOBX, FCB, and SRV_OPEN structures in checked builds. A log of these reference requests can be accessed by the logging system and WMI. For retail builds, this routine does nothing.
RxpUnregisterMinirdr	The routine is called by a network mini-redirector driver to unregister the driver with RDBSS and remove the registration information from the internal RDBSS registration table.
RxPurgeAllFobxs	This routine purges all the FOBX structures associated with a network mini-redirector.
RxPurgeRelatedFobxs	This routine purges all of the FOBX structures associated with a NET_ROOT structure.
RxReassociateMid	This routine reassociates a MID with an alternate context.
RxReference	This routine increments the reference count on an instance of several of the reference-counted data structures used by RDBSS.
RxRegisterMinirdr	This routine is called by a network mini-redirector driver to register the driver with RDBSS, which adds the registration information to an internal registration table. RDBSS also builds a device object for the network mini-redirector.
RxReleaseFcbResourceInMRx	This routine frees the FCB resource acquired using **RxAcquireExclusiveFcbResourceInMRx** or **RxAcquireSharedFcbResourceInMRx**.
RxReleaseFcbResourceForThreadInMRx	This routine frees the FCB resource acquired using **RxAcquireSharedFcbResourceInMRxEx** This routine is only available on Windows Server 2003 Service Pack 1 (SP1) and later.

ROUTINE	DESCRIPTION
RxResumeBlockedOperations_Serially	This routine wakes up the next waiting thread, if any, on the serialized blocking I/O queue.
RxScavengeAllFobxs	This routine scavenges all of the FOBX structures associated with a given network mini-redirector device object.
RxScavengeFobxsForNetRoot	This routine scavenges all of the FOBX structures that pertain to the given NET_ROOT structure.
RxSetDomainForMailslotBroadcast	This routine is called by a network mini-redirector driver to set the domain used for mailslot broadcasts if mailslots are supported by the driver.
RxSetMinirdrCancelRoutine	This routine sets up a network mini-redirector cancel routine for an RX_CONTEXT structure.
RxSetSrvCallDomainName	This routine sets the domain name associated with any given server (SRV_CALL structure).
RxSpinDownMRxDispatcher	This routine tears down the dispatcher context for a network mini-redirector. This routine is only available on Windows XP and later.
RxStartMinirdr	This routine starts up a network mini-redirector that called to register itself. RDBSS will also register the network mini-redirector driver as a Universal Naming Convention (UNC) provider with the Multiple UNC Provider (MUP) if the driver indicates support for UNC names.
RxStopMinirdr	This routine stops a network mini-redirector driver. A driver that is stopped will no longer accept new commands.
RxUnregisterMinirdr	This routine is an inline function defined in *rxstruc.h* that is called by a network mini-redirector driver to unregister the driver with RDBSS and remove the registration information from the internal RDBSS registration table. The **RxUnregisterMinirdr** inline function internally calls **RxpUnregisterMinirdr**.
_RxAllocatePoolWithTag	This routine allocates memory from a pool with a four-byte tag at the beginning of the block that can be used to help catch instances of memory problems. It is recommended that the **RxAllocatePoolWithTag** macro be used instead of calling this routine directly.

ROUTINE	DESCRIPTION
_RxCheckMemoryBlock	This routine checks a memory block for a special RX_POOL_HEADER header signature. Note that a network mini-redirector driver would need to add this special signature block to memory allocated in order to use the routine. This routine should not be used since this special header block has not been implemented.
_RxFreePool	This routine frees a memory pool. It is recommended that the **RxFreePool** macro be used instead of calling this routine directly.
_RxLog	This routine takes a format string and variable number of parameters and formats an output string for structureing as an I/O error log entry if logging is enabled. It is recommended that the **RxLog** macro be used instead of calling this routine directly. This routine is only available on checked builds of RDBSS on Windows Server 2003, Windows XP, and Windows 2000.
__RxFillAndInstallFastIoDispatch	This routine fills out a fast I/O dispatch vector to be identical with the normal dispatch I/O vector and installs it into the driver object that is associated with the device object passed. This routine is implemented only for non-monolithic drivers and does nothing on monolithic drivers.
__RxSynchronizeBlockingOperations	This routine is used to synchronize blocking I/O to the same work queue. This routine is used internally by RDBSS to synchronize named pipe operations. This routine may be used by a network mini-redirector to synchronize operations on a separate queue that is maintained by the network mini-redirector. This routine is only available on Windows Server 2003.
__RxSynchronizeBlockingOperationsMaybeDroppingFcbLock	This routine is used to synchronize blocking I/O to the same work queue. This routine is used internally by RDBSS to synchronize named pipe operations. This routine may be used by a network mini-redirector to synchronize operations on a separate queue that is maintained by the network mini-redirector. This routine is only available on Windows XP and Windows 2000.

Macros Defined by RDBSS

4/26/2017 • 4 min to read • Edit Online

A number of useful macros are defined in the Window Driver Kit (WDK)header files that call these RDBSS routines or other kernel routines. Some of these macros are normally used instead of calling the RDBSS routines directly. Some of these macros are used as convenience routines.

The following macros are defined by RDBSS.

MACRO	DESCRIPTION
RxAcquirePrefixTableLockExclusive (*TABLE*, *WAIT*)	This macro acquires the prefix table lock in exclusive mode for change operations.
RxAcquirePrefixTableLockShared (*TABLE*, *WAIT*)	This macro acquires the prefix table lock in shared mode for lookup operations.
RxAllocatePoolWithTag (*type*, *size*, *tag*)	On checked builds, this macro allocates memory from a pool with a four-byte tag at the beginning of the block that can be used to help catch instances of memory trashing. On retail builds, this macro becomes a direct call to **ExAllocatePoolWithTag**.
RxCheckMemoryBlock (*ptr*)	On checked builds, this macro checks a memory block for a special RX_POOL_HEADER header signature. On retail builds, this macro does nothing.
RxDereferenceAndFinalizeNetFcb (*Fcb* ,*RxContext*, *RecursiveFinalize*, *ForceFinalize*)	This macro is used to track dereference operations on FCB structures. Note that this macro manipulates the reference count and also returns the status of the final dereference call.
RxDereferenceNetFcb (*Fcb*)	This macro is used to track dereference operations on FCB structures. Note that this macro manipulates the reference count and also returns the status of the final dereference call.
RxDereferenceNetFobx (*Fobx,LockHoldingState*)	This macro is used to track dereference operations on FOBX structures.
RxDereferenceNetRoot (*NetRoot*, *LockHoldingState*)	This macro is used to track dereference operations on NET_ROOT structures.

MACRO	DESCRIPTION
RxDereferenceSrvCall (*SrvCall*, *LockHoldingState*)	This macro is used to track dereference operations on SRV_CALL structures.
RxDereferenceSrvOpen (*SrvOpen*, *LockHoldingState*)	This macro is used to track dereference operations on SRV_OPEN structures.
RxDereferenceVNetRoot (*VNetRoot*, *LockHoldingState*)	This macro is used to track dereference operations on V_NET_ROOT structures.
RxFcbAcquiredShared (*RXCONTEXT*, *FCB*)	This macro checks if the current thread has access to the regular resource in shared mode. This macro calls the **ExIsResourceAcquiredSharedLite** routine.
RxFillAndInstallFastIoDispatch(*__devobj*, *__fastiodisp*)	This macro calls **__RxFillAndInstallFastIoDispatch** to fill out a fast I/O dispatch vector to be identical with the normal dispatch I/O vector and installs it into the driver object associated with the device object passed.
RxFreePool (*ptr*)	On checked builds, this macro frees a memory pool. On retail builds, this macro becomes a direct call to **ExFreePool**.
RxIsFcbAcquiredShared (*FCB*)	This macro checks if the current thread has access to the regular resource in shared mode. This macro calls the **ExIsResourceAcquiredSharedLite** routine.
RxIsFcbAcquiredExclusive (*FCB*)	This macro checks if the current thread has access to the regular resource in exclusive mode. This macro calls the **ExIsResourceAcquiredExclusiveLite** routine.
RxIsFcbAcquired (*FCB*)	This macro checks if the current thread has access to the regular resource in either shared or exclusive mode. This macro calls the **ExIsResourceAcquiredSharedLite** and **ExIsResourceAcquiredExclusiveLite** routines.
RxIsPrefixTableLockAcquired (*TABLE*)	This macro indicates if the prefix table lock was acquired in either exclusive or shared mode.
RxIsPrefixTableLockExclusive (*TABLE*)	This macro indicates if the prefix table lock was acquired in exclusive mode.

MACRO	DESCRIPTION
RxLog(*Args*)	On checked builds, this macro calls the **_RxLog** routine. On retail builds, this macro does nothing. Note that the arguments to **RxLog** must be enclosed with an additional pair of parenthesis to enable translation into a null call when logging should be turned off.
RxLogEvent (*_DeviceObject*, *_OriginatorId*, *_EventId*, *_Status*)	This macro calls the **RxLogEventDirect** routine.
RxLogFailure (*_DeviceObject*, *_OriginatorId*, *_EventId*, *_Status*)	This macro calls the **RxLogEventDirect** routine.
RxLogFailureWithBuffer (*_DeviceObject*, *_OriginatorId*, *_EventId*, *_Status*, *_Buffer*, *_Length*)	This macro calls the **RxLogEventWithBufferDirect** routine.
RxLogRetail(*Args*)	On checked builds, this macro calls the **_RxLog** routine. On retail builds, this macro does nothing. Note that the arguments to **RxLogRetail** must be enclosed with an additional pair of parenthesis to enable translation into a null call when logging should be turned off.
RxReferenceNetFcb (*Fcb*)	This macro is used to track reference operations on FCB structures.
RxReferenceNetFobx (*Fobx*)	This macro is used to track reference operations on FOBX structures. A log of these reference operations can be accessed by the logging system and WMI.
RxReferenceNetRoot (*NetRoot*)	This macro is used to track reference operations on NET_ROOT structures. A log of these reference operations can be accessed by the logging system and Windows Management Instrumentation (WMI).
RxReferenceSrvCall (*SrvCall*)	This macro is used to track reference operations on SRV_CALL structures that are not at Deferred Procedure Call (DPC) level.
RxReferenceSrvCallAtDpc (*SrvCall*)	This macro is used to track reference operations on SRV_CALL structures at DPC level.
RxReferenceSrvOpen (*SrvOpen*)	This macro is used to track reference operations on SRV_OPEN structures.

MACRO	DESCRIPTION
RxReferenceVNetRoot (*VNetRoot*)	This macro is used to track reference operations on V_NET_ROOT structures.
RxReleasePrefixTableLock (*TABLE*)	This macro frees the prefix table lock.
RxSynchronizeBlockingOperations(*RXCONTEXT,FCB,IO QUEUE*)	This macro synchronizes blocking I/O requests to the same work queue. On Windows Server 2003, this macro calls the **__RxSynchronizeBlockingOperations** routine with the *DropFcbLock* parameter set to **FALSE**. On Windows XP and Windows 2000, this macro calls the **__RxSynchronizeBlockingOperationsMaybeDroppingFcbLock** routine with the *DropFcbLock* parameter set to **FALSE**.
RxSynchronizeBlockingOperations(*RXCONTEXT,FCB,IO QUEUE*)	This macro synchronizes blocking I/O requests to the same work queue. On Windows Server 2003, this macro calls the **__RxSynchronizeBlockingOperations** routine with the *DropFcbLock* parameter set to **TRUE**. On Windows XP and Windows 2000, this macro calls the **__RxSynchronizeBlockingOperationsMaybeDroppingFcbLock** routine with the *DropFcbLock* parameter set to **TRUE**.

Driver Registration and Start/Stop Control

4/26/2017 • 5 min to read • Edit Online

When the operating system is started, Windows loads RDBSS and any network mini-redirector drivers based on settings in the registry. For a monolithic network mini-redirector driver, which is linked statically with rdbsslib.lib, the driver must call the **RxDriverEntry** routine from its **DriverEntry** routine to initialize the copy of the RDBSSLIB library linked with the network driver. In this case, the **RxDriverEntry** routine must be called before any other RDBSS routines are called and used. For a non-monolithic network mini-redirector driver (the Microsoft SMB redirector), the rdbss.sys device driver is initialized in its own **DriverEntry** routine when loaded.

A network mini-redirector registers with RDBSS when the driver is loaded by the kernel and unregisters with RDBSS when the driver is unloaded. A network mini-redirector informs RDBSS that it has been loaded by calling **RxRegisterMinirdr**, the registration routine exported from RDBSS. As part of this registration process, the network mini-redirector passes a parameter to **RxRegisterMinirdr** that is a pointer to a large structure, MINIRDR_DISPATCH. This structure contains configuration information for the network mini-redirector and a dispatch table of pointers to callback routines implemented by the network mini-redirector kernel driver. RDBSS makes calls to the network mini-redirector driver through this list of callback routines.

The **RxRegisterMinirdr** routine sets all of the driver dispatch routines of the network mini-redirector driver to point to the top-level RDBSS dispatch routine, **RxFsdDispatch**. A network mini-redirector can override this behavior by saving its own entry points and rewriting the driver dispatch with its own entry points after the call to **RxRegisterMinirdr** returns or by setting a special parameter when calling **RxRegisterMinirdr**.

The network mini-redirector driver does not actually start operation until it receives a call to its **MRxStart** routine, one of the callback routines passed in the MINIRDR_DISPATCH structure. The **MrxStart** callback routine must be implemented by the network mini-redirector driver if it wishes to receive callback routines for operations unless the network mini-redirector preserves its own driver dispatch entry points. Otherwise, RDBSS will only allow the following I/O request packets through to the driver until **MrxStart** returns successfully:

- IRP requests for device creates and device operations where the FileObject->FileName.Length on the IRPSP is zero and the FileObject->RelatedFileObject is **NULL**.

For any other IRP request, the RDBSS dispatch routine **RxFsdDispatch** will return a status of STATUS_REDIRECTOR_NOT_STARTED.

The RDBSS dispatch routine will also fail any requests for the following I/O request packets:

- IRP_MJ_CREATE_MAILSLOT
- IRP_MJ_CREATE_NAMED_PIPE

The **MrxStart** callback routine implemented by the network mini-redirector is called by RDBSS when the **RxStartMinirdr** routine is called. The RDBSS **RxStartMinirdr** routine is usually called as a result of a file system control code (FSCTL) or I/O control code (IOCTL) request from a user-mode application or service to start the network mini-redirector. The call to **RxStartMinirdr** cannot be made from the **DriverEntry** routine of the network mini-redirector after a successful call to **RxRegisterMinirdr**since some of the start processing requires that the driver initialization be completed. Once the **RxStartMinirdr** call is received, RDBSS completes the start process by calling the **MrxStart** routine of the network mini-redirector. If the call to **MrxStart** returns success, RDBSS sets the internal state of the mini-redirector in RDBSS to RDBSS_STARTED.

RDBSS exports a routine, **RxSetDomainForMailslotBroadcast**, to set the domain for mailslot broadcasts. This routine is used during registration if the network mini-redirector supports mailslots.

A convenience routine, **__RxFillAndInstallFastIoDispatch**, exported by RDBSS can be used to copy all of the IRP_MJ_XXX driver routine pointers for handling I/O request processing to the comparable fast I/O dispatch vectors, but this routine only works for non-monolithic drivers.

RDBSS also exports routines to notify RDBSS that the network mini-redirector is starting or stopping. These calls are used if a network mini-redirector includes a user-mode administration service or utility application that starts and stops the redirector. This user-mode service or application can send custom FSCTL or IOCTL requests to the network mini-redirector driver to indicate that it should start or stop. The redirector can call the RDBSS **RxStartMinirdr** or **RxStopMinirdr** routines to notify RDBSS to start or stop this network mini-redirector.

The following table lists the RDBSS driver registration and start/stop control routines.

ROUTINE	DESCRIPTION
RxDriverEntry	This routine is called by a monolithic network mini-redirector driver from its **DriverEntry** routine to initialize RDBSS. For non-monolithic drivers, this initialization routine is equivalent to the **DriverEntry** routine of the rbss.sys device driver.
RxRegisterMinirdr	This routine is called by a network mini-redirector driver to register the driver with RDBSS, which adds the registration information to an internal registration table. RDBSS also builds a device object for the network mini-redirector.
RxSetDomainForMailslotBroadcast	This routine is called by a network mini-redirector driver to set the domain used for mailslot broadcasts, if mailslots are supported by the driver.
RxStartMinirdr	This routine starts up a network mini-redirector that called to register itself. RDBSS will also register the network mini-redirector driver as a UNC provider with the MUP if the driver indicates support for UNC names.
RxStopMinirdr	This routine stops a network mini-redirector driver. A driver that is stopped will no longer receive new commands except IOCTL or FSCTL requests.
RxpUnregisterMinirdr	This routine is called by a network mini-redirector driver to de-register the driver with RDBSS and remove the registration information from the internal RDBSS registration table.
RxUnregisterMinirdr	This routine is an inline function defined in rxstruc.h that is called by a network mini-redirector driver to de-register the driver with RDBSS and remove the registration information from the internal RDBSS registration table. The **RxUnregisterMinirdr** inline function internally calls **RxpUnregisterMinirdr**.

ROUTINE	DESCRIPTION
__RxFillAndInstallFastIoDispatch	This routine fills out a fast I/O dispatch vector to be identical with the normal dispatch I/O vector and installs it into the driver object associated with the device object passed.

The following macro is defined in the mrx.h header file that calls one of these routines. This macro is normally used instead of calling the **__RxFillAndInstallFastIoDispatch** routine directly.

MACRO	DESCRIPTION
RxFillAndInstallFastIoDispatch(*__devobj*, *__fastiodisp*)	This macro calls **__RxFillAndInstallFastIoDispatch**to fill out a fast I/O dispatch vector to be identical with the normal dispatch I/O vector and installs it into the driver object associated with the device object passed.

Pool Allocation and Free Routines

4/26/2017 • 1 min to read • Edit Online

RDBSS provides a number of routines to use for pool allocation. Normally, these routines are called using macros, not by calling these routines directly. The macros automatically handle the differences between retail and checked builds.

On a checked build, these routines were designed to add wrappers around the normal kernel allocation and free routines. These wrappers for pool allocation and free routines provide additional debugging information and call a set of routines that perform various kinds of checking and guarding before calling the kernel pool allocation and free routines. However, these features are not currently implemented in these allocation and free routines, but might be added in future releases.

On a free build, these routines become direct calls to the kernel allocation and free routines, **ExAllocatePoolWithTag** and **ExFreePool**.

The following table lists the RDBSS pool allocation and free routines.

ROUTINE	DESCRIPTION
_RxAllocatePoolWithTag	This routine allocates memory from a pool with a four-byte tag at the beginning of the block that can help catch memory problems. It is recommended that the **RxAllocatePoolWithTag** macro be called instead of using this routine directly.
_RxCheckMemoryBlock	This routine checks a memory block for a special RX_POOL_HEADER header signature. Note that a network mini-redirector driver would need to add this special signature block to memory allocated in order to use the routine. This routine should not be used since this special header block has not been implemented.
_RxFreePool	This routine frees a memory pool. It is recommended that the **RxFreePool** macro be called instead of using this routine directly.

A number of macros, which are defined in the *ntrxdef.h* header file, call these routines. Instead of calling the routines listed in the previous table directly, the following macros are normally used.

MACRO	DESCRIPTION
RxAllocatePoolWithTag (*type*, *size*, *tag*)	On checked builds, this macro allocates memory from a pool with a four-byte tag at the beginning of the block that can help catch instances of memory trashing. On retail builds, this macro becomes a direct call to **ExAllocatePoolWithTag**.

MACRO	DESCRIPTION
RxCheckMemoryBlock (*ptr*)	On checked builds, this macro checks a memory block for a special RX_POOL_HEADER header signature. On retail builds, this macro does nothing.
RxFreePool (*ptr*)	On checked builds, this macro frees a memory pool. On retail builds, this macro becomes a direct call to **ExFreePool**.

Timer and Worker Thread Management

4/26/2017 • 1 min to read • Edit Online

RDBSS provides several timer routines for worker thread management. These services are provided to all network mini-redirector drivers. The following types of timer routines are available:

- A periodic trigger
- A one-shot notification

A timer is associated with a device object and a worker thread routine. When a timer expires, a worker thread routine passed as an input parameter to the initial **RxPostOneShotTimerRequest** or **RxPostRecurrentTimerRequest** routine is called.

The following RDBSS timer routines are included.

ROUTINE	DESCRIPTION
RxCancelTimerRequest	This routine cancels a timer request. The request to be canceled is identified by a pointer to the routine and a context parameter.
RxPostOneShotTimerRequest	This routine is used by drivers to initialize a one-shot timer request. The worker thread routine passed to this routine is called once when the timer expires.
RxPostRecurrentTimerRequest	This routine initializes a recurrent timer request. The worker thread routine passed to this routine is called at regular intervals when the recurrent timer fires based on the input parameters to this routine.

Work Queue Dispatching Mechanisms

4/26/2017 • 4 min to read • Edit Online

RDBSS uses Windows kernel work queues to dispatch operations on multiple threads for later execution. Network mini-redirector drivers can use the work queues maintained by RDBSS for dispatching operations for later execution.

RDBSS provides several routines that implement the dispatching mechanism used in RDBSS. These routines can also be used by network mini-redirector drivers.

RDBSS keeps track of the work items on a per-device-object basis. This allows RDBSS to handle the race conditions associated with loading and unloading network mini-redirectors. This also provides a mechanism in RDBSS for preventing a single network mini-redirector from unfairly using all of the resources.

There are certain scenarios in which dispatching of work items is inevitable. To avoid frequent memory allocation and to free operations in these scenarios, the WORK_QUEUE_ITEM is allocated as part of another data. In other scenarios where dispatching is rare, it pays to avoid memory allocation until it is required. The RDBSS work queue implementation provides for both of these scenarios in the form of dispatching and posting work queue requests. In the case of dispatching using the **RxDispatchToWorkerThread** routine, no memory for the WORK_QUEUE_ITEM needs to be allocated by the caller. For posting using the **RxPostToWorkerThread** routine, the memory for the WORK_QUEUE_ITEM needs to be allocated by the caller.

There are two common cases of dispatching operations to worker threads:

- For a very infrequent operation, use the **RxDispatchToWorkerThread** routine to conserve memory use by dynamically allocating and freeing memory for the work queue item when it is needed.
- When an operation is going to be repeatedly dispatched, use the **RxPostToWorkerThread** routine to conserve time by allocating in advance the WORK_QUEUE_ITEM as part of the data structure to be dispatched.

The trade off between the two dispatching operations is time versus space (memory use).

The dispatching mechanism in RDBSS provides for multiple levels of work queues on a per-processor basis. The following levels of work queues currently supported:

- Critical
- Delayed
- HyperCritical

The distinction between Critical and Delayed is one of priority. The HyperCritical level is different from the other two in that the routines should not block (wait for any resource). This requirement cannot be enforced, so the effectiveness of the dispatching mechanism relies on the implicit cooperation of the clients.

The work queue implementation in RDBSS is built around a KQUEUE implementation. The additional support involves the regulation of a number of threads that are actively waiting for the work items. Each work queue data structure is allocated from nonpaged pool memory and has its own synchronization mechanism (a spinlock).

In addition to bookkeeping information (queue state and type, for example), RDBSS also maintains statistics that are gathered over the lifetime of the work queue. This can provide valuable information in tuning a work queue. The number of items that have been processed , the number of items that have to be processed, and the cumulative queue length is structureed. The cumulative queue length is an important metric and represents the sum of the number of items waiting to be processed each time an additional work item was queued. The cumulative queue length divided by the sum of the total number of items processed and the number of items to be processed gives an indication of the average queue length. A value much greater than one signifies that the minimum number of worker threads associated with the work queue can be increased. A value much less than one signifies that the maximum number of work threads associated with the queue can be decreased.

The work queue typically start in an active state and continue until either a non-recoverable situation is encountered (lack of system resources, for example) or when it transitions to the inactive state. When a rundown is initiated, it transitions to the rundown-in-progress state.

The rundown of work queues is not complete when the threads have been spun down. The termination of the threads needs to be ensured before the data structures can be torn down. The work queue implementation in RDBSS follows a protocol in which each of the threads being spun down saves a reference to the thread object in the rundown context. The rundown issuing thread (which does not belong to the work queue) waits for the completion of all of the threads spun down before tearing down the data structures.

The current implementation of **RxDispatchToWorkerThread** and **RxPostToWorkerThread** queues work onto the same processor from which the call originated.

The following RDBSS routines for work queue dispatching include.

ROUTINE	DESCRIPTION
RxDispatchToWorkerThread	This routine invokes a routine in the context of a worker thread. The memory for the WORK_QUEUE_ITEM is allocated by this routine.
RxPostToWorkerThread	This routine invokes the routine in the context of a worker thread. Memory for the WORK_QUEUE_ITEM must be allocated by the caller.
RxSpinDownMRxDispatcher	This routine tears down the dispatcher context for a network mini-redirector. Note that this routine is only available on Windows Server 2003 and Windows XP.

Diagnostics and Debugging

4/26/2017 • 2 min to read • Edit Online

RDBSS provides a number of routines for diagnostic and debugging purposes. These routines fall into two categories:

- Assert and debug routines
- Reference and dereference tracking routines

These routines include the items in the following table.

ROUTINE	DESCRIPTION
RxAssert	This routine sends an assert string in checked builds of RDBSS to a kernel debugger if one is installed. When the rxAssert.h include file is used, Windows kernel **RtlAssert** calls will be redefined to call this **RxAssert** routine as well. For retail builds, calls to this routine will bug check.
RxDbgBreakPoint	This routine raises an exception that is handled by the kernel debugger if one is installed; otherwise, it is handled by the debug system.
RxpTrackDereference	This routine is used to track a request to reference SRV_CALL, NET_ROOT, V_NET_ROOT, FOBX, FCB, and SRV_OPEN structures in checked builds. A log of these reference requests can be accessed by the logging system and WMI. This routine does not perform the dereference operation. For retail builds, this routine does nothing.
RxpTrackReference	This routine is used to track a request to dereference SRV_CALL, NET_ROOT, V_NET_ROOT, FOBX, FCB, and SRV_OPEN structures in checked builds. A log of these dereference requests can be accessed by the logging system and WMI. This routine does not perform the reference operation. For retail builds, this routine does nothing.

In addition to the routines listed in the previous table, a number of macros that call these routines are defined for debugging. These macros, which are listed in the following table, provide a wrapper around the **RxReference** or **RxDereference** routines used for file structure management operations on SRV_CALL, NET_ROOT, V_NET_ROOT, FOBX, FCB, and SRV_OPEN structures. These macros first call the corresponding **RxpTrackReference** or **RxpTrackDereference** routine to log diagnostic information before calling the corresponding **RxReference** or **RxDeference** routine. A log of the reference and dereference requests can be accessed by the RDBSS logging system and WMI.

MACRO	DESCRIPTION
RxDereferenceAndFinalizeNetFcb (*Fcb* ,*RxContext*, *RecursiveFinalize*, *ForceFinalize*)	This macro is used to track dereference operations on FCB structures. Note that this macro manipulates the reference count and also returns the status of the finalize call.
RxDereferenceNetFcb (*Fcb*)	This macro is used to track dereference operations on FCB structures. Note that this macro manipulates the reference count and also returns the status of the final dereference call.
RxDereferenceNetFobx (*Fobx,LockHoldingState*)	This macro is used to track dereference operations on FOBX structures.
RxDereferenceNetRoot (*NetRoot*, *LockHoldingState*)	This macro is used to track dereference operations on NET_ROOT structures.
RxDereferenceSrvCall (*SrvCall*, *LockHoldingState*)	This macro is used to track dereference operations on SRV_CALL structures.
RxDereferenceSrvOpen (*SrvOpen*, *LockHoldingState*)	This macro is used to track dereference operations on SRV_OPEN structures.
RxDereferenceVNetRoot (*VNetRoot*, *LockHoldingState*)	This macro is used to track dereference operations on NET_ROOT structures.
RxReferenceNetFcb (*Fcb*)	This macro is used to track reference operations on FCB structures.
RxReferenceNetFobx (*Fobx*)	This macro is used to track reference operations on FOBX structures.
RxReferenceNetRoot (*NetRoot*)	This macro is used to track reference operations on NET_ROOT structures.
RxReferenceSrvCall (*SrvCall*)	This macro is used to track reference operations on SRV_CALL structures that are not at DPC level.
RxReferenceSrvCallAtDpc (*SrvCall*)	This macro is used to track reference operations on SRV_CALL structures at DPC level.
RxReferenceSrvOpen (*SrvOpen*)	This macro is used to track reference operations on SRV_OPEN structures.

MACRO	DESCRIPTION
RxReferenceVNetRoot (*VNetRoot*)	This macro is used to track reference operations on V_NET_ROOT structures.

Logging Routines and Macros

4/26/2017 • 2 min to read • Edit Online

RDBSS provides a number of routines for logging. These logging facilities are always present. When the RDBSSLOG macro is defined, a generation of the logging calls on checked builds is enabled. When NO_RDBSSLOG is set, the logging calls are disabled.

The logging routines create log records that are stored in a circular buffer. Each record is bounded on either side by a record descriptor. This record descriptor is four bytes long.

The following table includes logging routines.

ROUTINE	DESCRIPTION
RxLogEventDirect	This routine is called to log an error to the I/O error log. It is recommended that the **RxLogFailure** or **RxLogEvent** macro be used instead of calling this routine directly.
RxLogEventWithAnnotation	This routine allocates an I/O error log record, fills in the log record, and writes this record to the I/O error log.
RxLogEventWithBufferDirect	This routine allocates an I/O error log record, fills in the log record, and writes this record to the I/O error log. This routine encodes the line number and status into the raw data buffer stored in the I/O error log record. It is recommended that the **RxLogFailureWithBuffer** macro be used instead of calling this routine directly.
_RxLog	This routine takes a format string and variable number of parameters and formats an output string for recording as an I/O error log entry if logging is enabled. It is recommended that the **RxLog** macro be used instead of calling this routine directly. This routine is only available on checked builds of RDBSS on Windows Server 2003, Windows XP, and Windows 2000.

The following macros are defined in the rxlog.h and rxprocs.h header files that call the routines listed in the previous table. These macros are normally used instead of calling these routines directly.

MACRO	DESCRIPTION
RxLog(*Args*)	On checked builds, this macro calls the **_RxLog** routine. On retail builds, this macro does nothing. Note that the arguments to **RxLog** must be enclosed with an additional pair of parenthesis to enable translation into a null call when logging should be turned off.

MACRO	DESCRIPTION
RxLogEvent (*_DeviceObject*, *_OriginatorId*, *_EventId*, *_Status*)	This macro calls the **RxLogEventDirect** routine.
RxLogFailure (*_DeviceObject*, *_OriginatorId*, *_EventId*, *_Status*)	This macro calls the **RxLogEventDirect** routine.
RxLogFailureWithBuffer (*_DeviceObject*, *_OriginatorId*, *_EventId*, *_Status*, *_Buffer*, *_Length*)	This macro calls the **RxLogEventWithBufferDirect** routine.
RxLogRetail(*Args*)	On checked builds, this macro calls the **_RxLog** routine. On retail builds, this macro does nothing. Note that the arguments to **RxLogRetail** must be enclosed with an additional pair of parenthesis to enable translation into a null call when logging should be turned off.

Miscellaneous Routines

4/26/2017 • 1 min to read • Edit Online

RDBSS includes a number of utility routines that do not fall into a particular category.

The RDBSS miscellaneous routines include the following:

ROUTINE	DESCRIPTION
RxFsdDispatch	This routine implements the file system driver (FSD) dispatch for RDBSS to process an I/O request packet (IRP). This routine is called by a network mini-redirector in the driver dispatch routines to initiate RDBSS processing of a request.
RxFsdPostRequest	This routine queues the IRP specified by an RX_CONTEXT structure to the worker queue for processing by the file system process (FSP).
RxGetRDBSSProcess	This routine returns a pointer to the process of the main thread used by the RDBSS kernel process.
RxIsThisACscAgentOpen	This routine determines if a file open request was made by a user-mode client-side caching agent. This routine is only available on Windows Server 2003.
RxMakeLateDeviceAvailable	This routine modifies the device object to make a "late device" available. A late device is one that is not created in the driver's load routine.
RxPrepareToReparseSymbolicLink	This routine sets up the file object name to facilitate a reparse. This routine is used by the network mini-redirectors to traverse symbolic links. This routine should not be used by network mini-redirectors.

RX_CONTEXT and IRP Management

7/21/2017 • 4 min to read • Edit Online

The RX_CONTEXT structure is one of the fundamental data structures used by RDBSS and network mini-redirectors to manage an I/O request packet (IRP). An RX_CONTEXT structure describes an IRP while it is being processed and contains state information that allows global resources to be released as the IRP is completed. The RX_CONTEXT data structure encapsulates an IRP for use by RDBSS, network mini-redirectors, and the file system. An RX_CONTEXT structure includes a pointer to a single IRP and all of the context required to process the IRP.

An RX_CONTEXT structure is sometimes referred to as an IRP Context or RxContext in the Windows Driver Kit (WDK) header files and other resources used for developing network mini-redirector drivers.

The RX_CONTEXT is a data structure to which additional information provided by the various network mini redirectors is attached. From a design standpoint, this additional information can be handled in one of several ways:

- Allow for context pointers to be defined as part of RX_CONTEXT, which network mini-redirectors use to store away their information. This implies that every time an RX_CONTEXT structure is allocated and destroyed, the network mini-redirector driver must perform a separate associated allocation or destruction of the memory block that contains the additional network mini-redirector information. Since RX_CONTEXT structures are created and destroyed in large numbers, this is not an acceptable solution from a performance standpoint.
- Another approach consists of over allocating the size of each RX_CONTEXT structure by a pre-specified amount for each network mini redirector, which is then reserved for use by the mini redirector. Such an approach avoids the additional allocation and destruction but complicates the RX_CONTEXT management code in RDBSS.
- The third approach consists of allocating a pre-specified area, which is the same for all network mini redirectors as part of each RX_CONTEXT. This is an unformatted area on top of which any desired structure can be imposed by the various network mini redirectors. Such an approach overcomes the disadvantages associated with previous approaches. This is the approach currently implemented in RDBSS.

The third approach is the scheme used by RDBSS. Consequently, developers of network mini-redirector drivers should try and define the associated private context to fit into this pre-specified area defined in the RX_CONTEXT data structure. Network mini-redirector drivers that violate this rule will incur a significant performance penalty.

Many RDBSS routines and routines exported by a network mini-redirector make reference to RX_CONTEXT structures in either the initiating thread or in some other thread used by the routine. Thus, the RX_CONTEXT structures are reference counted to manage their use for asynchronous operations. When the reference count goes to zero, the RX_CONTEXT structure can be finalized and released on the last dereference operation.

RDBSS provides a number of routines that are used to manipulate an RX_CONTEXT structure and the associated IRP. These routines are used to allocate, initialize, and delete an RX_CONTEXT structure. These routines are also used to complete the IRP associated with an RX_CONTEXT and set up a cancel routine for an RX_CONTEXT.

The following routines manipulate RX_CONTEXT structures:

ROUTINE	DESCRIPTION

ROUTINE	DESCRIPTION
RxCompleteRequest	This routine is used to complete an IRP associated with an RX_CONTEXT structure. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxCompleteRequest_Real	This routine is used to complete an IRP associated with an RX_CONTEXT structure. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxCreateRxContext	This routine allocates a new RX_CONTEXT structure and initializes the data structure.
RxDereferenceAndDeleteRxContext_Real	This routine dereferences an RX_CONTEXT structure, and if the reference count goes to zero, then it deallocates and removes the specified RX_CONTEXT structure from the RDBSS in-memory data structures.
RxInitializeContext	This routine initializes a newly allocated RX_CONTEXT structure.
RxPrepareContextForReuse	This routine prepares an RX_CONTEXT structure for reuse by resetting all operation-specific allocations and acquisitions previously made. The parameters obtained from the IRP are not modified. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxResumeBlockedOperations_Serially	This routine wakes up the next waiting thread, if any, on the serialized blocking I/O queue.
RxSetMinirdrCancelRoutine	The routine sets up a network mini-redirector cancel routine for an RX_CONTEXT structure.
__RxSynchronizeBlockingOperations	This routine is used to synchronize blocking I/O to the same work queue. This routine is used internally by RDBSS to synchronize named pipe operations. This routine may be used by a network mini-redirector to synchronize operations on a separate queue that is maintained by the network mini-redirector. The routine is only available on Windows Server 2003.

ROUTINE	DESCRIPTION
__RxSynchronizeBlockingOperationsMaybeDroppingFcbLock	This routine is used to synchronize blocking I/O to the same work queue. This routine is used internally by RDBSS to synchronize named pipe operations. This routine may be used by a network mini-redirector to synchronize operations on a separate queue that is maintained by the network mini-redirector. The routine is only available on Windows XP and Windows 2000.

The following macros are defined in the *rxcontx.h* header file that call the routines listed in the previous table. These macros are normally used instead of calling these routines directly.

MACRO	DESCRIPTION
RxSynchronizeBlockingOperations(*RXCONTEXT,FCB,IO QUEUE*)	This macro synchronizes blocking I/O requests to the same work queue. On Windows Server 2003, this macro calls the **__RxSynchronizeBlockingOperations** routine with the *DropFcbLock* parameter set to **FALSE**. On Windows XP and Windows 2000, this macro calls the **__RxSynchronizeBlockingOperationsMaybeDroppingFcbLock** routine with the *DropFcbLock* parameter set to **FALSE**.
RxSynchronizeBlockingOperations(*RXCONTEXT,FCB,IO QUEUE*)	This macro synchronizes blocking I/O requests to the same work queue. On Windows Server 2003, this macro calls the **__RxSynchronizeBlockingOperations** routine with the *DropFcbLock* parameter set to **TRUE**. On Windows XP and Windows 2000, this macro calls the **__RxSynchronizeBlockingOperationsMaybeDroppingFcbLock** routine with the *DropFcbLock* parameter set to **TRUE**.

Connection and File Structure Management

4/26/2017 • 3 min to read • Edit Online

There are six fundamental data structures used by RDBSS for managing connections and file structures. These data structures are used internally by RDBSS and by the various network mini-redirectors. There are two versions of these data structures. The network mini-redirector version contains fields that can be manipulated by a network mini-redirector driver. The network mini-redirector version of these data structures starts with the MRX_ prefix. The RDBSS version contains additional fields that can only be manipulated by RDBSS.

These six fundamental data structures are as follows:

- SRV_CALL--server call context. This structure provides the abstraction for a remote server.
- NET_ROOT--net root. This structure abstracts a connection to a share.
- V_NET_ROOT--view of net roots (also referred to as virtual netroots).
- FCB--file control block. This structure represents an open file on a share.
- SRV_OPEN--server-side open context. This structure encapsulates an open handle on the server.
- FOBX--file object extension. This structure is an RDBSS extension to the **FILE_OBJECT** structure.

These data structures are organized in the following hierarchy:

```
           SRV_CALL
FCB   <------> NET_ROOT
   SRV_OPEN  <---> V_NET_ROOT
       FOBX
           FILE_OBJECT
```

In response to kernel file system calls, RDBSS normally creates and finalizes for a network mini-redirector driver all of the previously mentioned structures except the FOBX structure. So, a network mini-redirector driver will normally only call a few of the RDBSS routines used for connection and file structure management. Most of these routines are called internally by RDBSS.

All of these data structures are reference counted. The reference counts on a data structure are as follows:

DATA STRUCTURE	DESCRIPTION OF REFERENCE COUNT
SRV_CALL	The number of NET_ROOT entries that point to SRV_CALL, plus some dynamic value.
NET_ROOT	The number of FCB entries and V_NET_ROOT entries that point to NET_ROOT, plus some dynamic value.
V_NET_ROOT	The number of SRV_OPEN entries that point to V_NET_ROOT, plus some dynamic value.
FCB	The number of SRV_OPEN entries that point to FCB, plus some dynamic value.

DATA STRUCTURE	DESCRIPTION OF REFERENCE COUNT
SRV_OPEN	The number of FOBX entries that point to SRV_OPEN, plus some dynamic value.
FOBX	Some dynamic value.

In each case, the dynamic value refers to the number of callers that have referenced the structure without dereferencing it. The static part of the reference count is maintained by the routines themselves. For example, **RxCreateNetRoot** increments the reference count for the associated SRV_CALL structure.

Reference calls and successful lookups increment the reference counts; dereference calls decrements the count. Create routine calls allocate a structure and set the reference count to 1.

The reference count associated with any data structure is at least 1 plus the number of instances of the data structure at the next lower level associated with it. For example, the reference count associated with a SRV_CALL, which has two NET_ROOTs associated with it, is at least 3. In addition to the references held by the RDBSS internal **NameTable** structures and the data structure at the next lower level, there are additional references that may have been acquired.

These restrictions ensure that a data structure at any given level cannot be finalized (released and the associated memory block freed) until all of the data structures at the next level below have been finalized or have released their references. For example, if a reference to an FCB is held, then it is safe to access the V_NET_ROOT, NET_ROOT, and SRV_CALL structures associated with it.

The two important abstractions used in the interface between the network mini-redirectors and RDBSS are SRV_CALL and NET_ROOT structures. A SRV_CALL structure corresponds to the context associated with a server with which a connection has been established, and the NET_ROOT structure corresponds to a share on a server (this could also be viewed as a portion of the namespace, which has been claimed by a network mini-redirector).

The creation of SRV_CALL and NET_ROOT structures typically involves at least one network round trip. To provide for asynchronous operations to continue, these operations are modeled as a two-phase activity. Each call-down to a network mini-redirector for creating a SRV_CALL and a NET_ROOT structure is accompanied by a call-up from the network mini-redirector to the RDBSS to notify the completion status of the request. Currently these are synchronous.

The creation of a SRV_CALL structure is further complicated by the fact that the RDBSS must choose from a number of network mini-redirectors to establish a connection with a server. To provide the RDBSS with maximum flexibility in choosing the network mini-redirector that it wishes to deploy, the creation of a SRV_CALL structure involves a third phase in which the RDBSS notifies the network mini-redirector of a winner. All of the losing network mini-redirectors destroy the associated context.

This section contains the following topics:

The SRV_CALL Structure

7/21/2017 • 1 min to read • Edit Online

The server call context structure, SRV_CALL, maintains information about each specific network server connection maintained by a network mini-redirector.

A global list of the SRV_CALL structures is maintained in global data by RDBSS. Each SRV_CALL structure has a few elements common with other RDBSS structures, along with elements that are unique to a SRV_CALL structure. The RDBSS routines that manage SRV_CALL structures only modify the following elements:

- Signature and reference count
- A name and associated table information
- A list of associated NET_ROOT entries
- A set of timing parameters that control how often the network mini-redirector wants to be called by RDBSS in different circumstances (idle timeouts, for example)
- The associated network mini-redirector driver ID
- Whatever additional storage is request by the network mini-redirector (or the creator of the SRV_CALL data structure)

The Unicode name of the SRV_CALL structure is carried in the structure itself at the end. Extra space reserved for use by the network mini-redirector begins at the end of the known SRV_CALL data structure so that a network mini-redirector can simply refer to this extra space using context fields from an include file.

The finalization of a SRV_CALL structure consists of two parts:

1. Destroying the association with all NET_ROOTS
2. Freeing the memory

There can be a delay between these two actions, and a field in the SRV_CALL structure prevents the first step from being duplicated.

The NET_ROOT Structure

4/26/2017 • 1 min to read • Edit Online

A net root structure, NET_ROOT, contains information for each specific network server\share connection maintained by a network mini-redirector.

A NET_ROOT is what the RDBSS and a network mini-redirector driver want to deal with, not a server. Accordingly, RDBSS normally creates and opens a NET_ROOT structure and calls the network mini-redirector driver responsible for opening the server. The network mini-redirector driver is expected to populate the appropriate fields in the passed in NET_ROOT structure.

A list of the NET_ROOT structures is maintained by RDBSS for each SRV_CALL. Each NET_ROOT structure has a few elements common with other RDBSS structures, along with elements that are unique to a NET_ROOT structure. The RDBSS routines that manage NET_ROOT structures only modify the following elements:

- Signature and reference count
- A name and associated table information
- A back pointer to the associated SRV_CALL structure
- Size information for the various substructures
- A lookup table of associated FCB structures
- Whatever additional storage is request by the network mini-redirector (or the creator of the NET_ROOT data structure)

A NET_ROOT structure also contains a list of RX_CONTEXT structures that are waiting for the NET_ROOT transitioning to be completed before resumption of IRP processing. This typically happens when concurrent requests are directed at a server. One of these requests is initiated while the other requests are queued. Extra space reserved for use by the network mini-redirector begins at the end of the known NET_ROOT data structure so that a network mini-redirector can simply refer to this extra space using context fields from an include file.

The finalization of a NET_ROOT structure consists of two parts:

1. Destroying the association with all V_NET_ROOTS
2. Freeing the memory

There can be a delay between these two actions, and a field in the NET_ROOT structure prevents the first step from being duplicated.

The V_NET_ROOT Structure

7/21/2017 • 1 min to read • Edit Online

The V_NET_ROOT structure provides a mechanism for mapping into a share (for example, a user drive mapping that points below the root of the associated share point). The V_NET_ROOT name can be in one of the following formats:

```
\server\share\d1\d2
\;m:\server\share\d1\d2
```

The format of the name depends on whether there is a local device ("X:", for example) associated with this V_NET_ROOT structure. In the case of a local drive mapping (d1\d2, for example), the local drive mapping gets prefixed onto each **CreateFile** that is opened on this V_NET_ROOT structure.

V_NET_ROOT structures are also used to supply alternate credentials. The purpose for this kind of a V_NET_ROOT structure is to propagate the alternate credentials into the NET_ROOT as the default. For this to work, there must be no other references.

A list of the V_NET_ROOT structures is maintained by RDBSS for each NET_ROOT. Each V_NET_ROOT structure has a few elements common with other RDBSS structures, along with elements that are unique to a V_NET_ROOT structure. The RDBSS routines that manage V_NET_ROOT structures only modify the following elements:

- Signature and reference count
- A pointer to the associated NET_ROOT structure and links
- Name information for table lookup (prefix)
- Name for a prefix to be added to whatever name the user sees (this is for simulating a NET_ROOT structure that is not mapped at the root of the actual NET_ROOT structure)

The finalization of a V_NET_ROOT structure consists of two parts:

1. Destroying the association with all SRV_OPEN structures
2. Freeing the memory

There can be a delay between these two actions, and a field in the V_NET_ROOT structure prevents the first step from being duplicated.

The FCB Structure

4/26/2017 • 1 min to read • Edit Online

The file control block (FCB) structure is pointed to by the *FsContext* field in the file object. All of the operations that share an FCB refer to the same file. Unfortunately, SMB servers are implemented today in such a way that a name can be an alias, so that two different names could be the same file. The FCB is the focal point of file operations. Since operations on the same FCB are actually on the same file, synchronization is based on the FCB rather than some higher level object.

Whenever an FCB structure is created, a corresponding SRV_OPEN and FOBX structure is also created. More than one SRV_OPEN structure can be associated with a given FCB structure, and more than one FOBX structure is associated with a given SRV_OPEN structure. In most cases, the one SRV_OPEN structure is associated with an FCB, and the number of FOBX structures associated with a given SRV_OPEN structure is 1. To improve the spatial locality and the paging behavior in such cases, the allocation for an FCB structure also involves an allocation for one associated SRV_OPEN and FOBX structure.

RDBSS tries to allocate the associated FCB, SRV_OPEN, and FOBX structures together in memory to improve paging behavior. RDBSS does not allocate the FCB and NET_ROOT structures together because the NET_ROOT structures are not paged, but FCB structures usually are paged (unless they are paging files).

The FCB structure corresponds to every open file and directory. The FCB structure is split up into the following two portions:

- A non-paged part allocated in non-paged pool
- A paged part

The former is the NON_PAGED_FCB and the later is referred to as FCB.

The FCB contains a pointer to the corresponding NON_PAGED_FCB part. A backpointer is maintained from the NON_PAGED_FCB to the FCB for debugging purposes in checked builds.

The NON_PAGED_FCB contains a structure of special pointers used by Memory Manager and Cache Manager to manipulate section objects. Note that the values for these pointers are normally set outside of the file system.

An FCB structure contains the following:

- An FSRTL_COMMON_HEADER structure
- A signature and reference count
- A name and associated table information
- A backpointer to the associated NET_ROOT structure
- A list of associated SRV_OPEN structures
- The device object
- Any additional storage requested by the network mini-redirector or the creator of the FCB structure

The SRV_OPEN Structure

7/21/2017 • 1 min to read • Edit Online

The SRV_OPEN structure describes a specific open on the server. Multiple file objects and file object extensions (FOBXs) can share the same SRV_OPEN structure if the access rights match. For example, where the file ID is stored for SMBs. A list of the file IDs is associated with the FCB. Similarly, all file object extensions that share the same server-side open are listed together here. Also, information is stored about whether a new open of the FCB can share the server-side open context.

The flag values that affect SRV_OPEN operations are split into two groups:

- Flags visible to network mini-redirectors
- Private flags used internally by RDBSS and invisible to network mini-redirectors

The flags visible to network mini-redirectors consist of the lower 16 bits of the possible SRV_OPEN flags. The upper 16 bits are reserved for use internally by RDBSS.

A SRV_OPEN structure contains the following:

- Signature and reference count
- A backpointer to the FCB structure
- A backpointer to the V_NET_ROOT structure (usually)
- A list of FOBX structures
- Access rights and collapsibility status
- Additional storage requested by the network mini-redirector or the creator of the SRV_OPEN structure

The FOBX Structure

4/26/2017 • 1 min to read • Edit Online

A file object extension (FOBX) structure is an RDBSS extension to the **FILE_OBJECT** structure. The FOBX structure is pointed to by the **FileObjectExtension** field in the file object. An FOBX structure contains the following:

- A signature and reference count
- A backpointer to the associated FCB structure
- A backpointer to the associated SRV_OPEN structure
- Context information about this open structure
- Additional storage requested by the network mini-redirector or the creator of the FOBX structure

The FOBX structure contains all of the information that is needed, per file object, which is not normally stored by the I/O system. Information about file objects is stored by the I/O system in fixed-size file system objects. The FOBX structure handles the other information needed on file objects by network mini-redirectors.

The FOBX structure for any file object is referenced by the **FsContext2** field in the file object. Even though the FOBX structure is ordinarily a terminus in the RDBSS structure, the FOBX structure is currently reference counted anyway.

The FOBX flags are split into two groups:

- Flags visible to network mini-redirectors
- Flags used internally by RDBSS and invisible to network mini-redirectors

The flags visible to network mini-redirectors consist of the lower 16 bits of the possible FOBX flags. The upper 16 bits are reserved for use internally by RDBSS.

Connections and File Structure Locking

4/26/2017 • 3 min to read • Edit Online

For locking purposes, there are two levels of lookup tables used:

- A per-device object table for SRV_CALL and NET_ROOT structures (prefix table)
- A table-per-NET_ROOT structure for FCB structures (FCB table)

These separate tables allow directory operations on different NET_ROOT structures to be almost completely non-interfering once the connections are established. Directory operations on the same NET_ROOT structure do interfere slightly, however. The following table describes what locks are needed for specific operations:

OPERATION	DATA TYPES	LOCK REQUIRED
Create or Finalize	SRV_CALL NET_ROOT V_NET_ROOT	An exclusive lock on the NetName table (the TableLock field of RxContext->RxDeviceObject->pRxNetNameTable).
Reference, Dereference, or Lookup	SRV_CALL NET_ROOT V_NET_ROOT	A shared or exclusive lock on the NetName table (the TableLock field of RxContext->RxDeviceObject->pRxNetNameTable).
Create or Finalize	FCB SRV_OPEN FOBX	An exclusive lock on the FCB table (the TableLock field of NET_ROOT->FcbTable).
Reference, Dereference, or Lookup	FCB SRV_OPEN FOBX	A shared or exclusive lock on the FCB table (the TableLock field of NET_ROOT->FcbTable).

Note that manipulations of SRV_OPEN and FOBX data structures currently require the same lock as needed for manipulations of FCB data structures. This is simply a memory saving idea. Future versions of Windows may add another resource at the FCB level to remove this restriction so that a set of shared resources could be used to decrease the probability of a collision to an acceptably low level.

If you require both locks (FinalizeNetFcb, for example), you must take the lock on NetName table first and then the lock on the FCB table. You must release the locks in the opposite order.

The SRV_CALL, NET_ROOT, and V_NET_ROOT creation and finalization process is governed by the acquisition and release of the RDBSS lock on the NetName table.

The FCB creation and finalization process is governed by the acquisition and release of the lock on the NetName table associated with the NET_ROOT structure.

The FOBX and SRVOPEN creation and finalization process is governed by the acquisition and release of the lock on the FCB table.

The following table summarizes the locks and the modes in which the locks need to be acquired for creation and finalization of the various data structures:

TYPE OF OPERATION	SRV_CALL	NET_ROOT	FCB	SRV_OPEN	FOBX
Create	Exclusive lock on NetName table	Exclusive lock on NetName table	Exclusive lock on FCB table	Exclusive lock on FCB table	Exclusive lock on FCB table
Finalize	Exclusive lock on NetName table	Exclusive lock on NetName table	Exclusive lock on FCB table	Exclusive lock on FCB table	Exclusive lock on FCB table

Referencing and dereferencing these data structures must adhere to certain conventions as well.

When the reference count associated with any of the data structures drops to 1 (the sole reference being held by the name table in most cases), the data structure is a potential candidate for finalization. The data structure can be either finalized immediately or it can be marked for scavenging. Both of these methods are implemented in RDBSS. When the locking requirements are met during dereferencing, the data structures are finalized immediately. The one exception to this is when delayed operation optimization is implemented (dereferencing the FCB structure, for example). Otherwise, the data structure is marked for scavenging.

A network mini-redirector should have an exclusive lock on the NetName table in order to call a finalization routine.

To execute a Create on one of these data structures, a network mini-redirector driver should do something similar to the following:

```
getshared();lookup();
if (failed) {
    release(); getexclusive(); lookup();
        if ((failed) { create(); }
    }
deref();
release();
```

When you have successfully acquired the lock, insert the node into the table, release the lock, and then see if the server is available. If the server is available, set up the rest of the information and unblock anyone who is waiting on the same server (the SRV_CALL or NET_ROOT structures).

Connection and File Control Block Management Routines

4/26/2017 • 4 min to read • Edit Online

The connection and file control block management routines are used by RDBSS to manage structures used to represent connections and file control blocks.

RDBSS provides the following routines for connection and file control block management that can be used by network mini-redirector drivers:

ROUTINE	DESCRIPTION
RxCreateNetFcb	This routine allocates, initializes, and inserts a new FCB structure into the in-memory data structures for a NET_ROOT structure on which this FCB is being opened. The structure allocated has space for a SRV_OPEN and an FOBX structure. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxCreateNetFobx	This routine allocates, initializes, and inserts a new file object extension (FOBX) structure. Network mini-redirectors should call this routine to create an FOBX at the end of a successful create operation.
RxCreateNetRoot	This routine builds a node that represents a NET_ROOT structure and inserts the name into the net name table on the associated device object. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxCreateSrvCall	This routine builds a node that represents a server call context and inserts the name into the net name table maintained by RDBSS. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxCreateSrvOpen	This routine allocates, initializes, and inserts a new SRV_OPEN structure into the in-memory data structures used by RDBSS. If a new structure must be allocated, it has space for an FOBX structure. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxCreateVNetRoot	This routine builds a node that represents a V_NET_ROOT structure and inserts the name into the net name table. This routine is used internally by RDBSS and should not be used by network mini-redirectors.

ROUTINE	DESCRIPTION
RxDereference	This routine decrements the reference count on an instance of several of the reference-counted data structures used by RDBSS.
RxFinalizeConnection	This routine deletes a connection to a share. Any files open on the connection are closed depending on the level of force specified. The network mini-redirector might choose to keep the transport connection open for performance reasons, unless some option is specified to force a close of the connection.
RxFinalizeNetFcb	This routine finalizes the given FCB structure. The caller must have an exclusive lock on the NET_ROOT structure associated with this FCB. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxFinalizeNetFobx	This routine finalizes the given FOBX structure. The caller must have an exclusive lock on the FCB associated with this FOBX. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxFinalizeNetRoot	This routine finalizes the given NET_ROOT structure. The caller should have exclusive lock on the NetName table of the device object associated with this NET_ROOT structure (through the SRV_CALL structure). This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxFinalizeSrvCall	This routine finalizes the given SRV_CALL structure. The caller should have exclusive access to the lock on the NetName table of the device object associated with this SRV_CALL structure. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxFinalizeSrvOpen	This routine finalizes the given SRV_OPEN structure. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxFinalizeVNetRoot	This routine finalizes the given V_NET_ROOT structure. The caller must have exclusive access to the lock on the NetName table of the device object associated with this V_NET_ROOT structure. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxFinishFcbInitialization	This routine is used to finish initializing an FCB after the successful completion of a create operation by the network mini-redirector.

ROUTINE	DESCRIPTION
RxForceFinalizeAllVNetRoots	This routine force finalizes all of the V_NET_ROOT structures associated with a given NET_ROOT structure. The caller must have exclusive access to the lock on the NetName table of the device object associated with this V_NET_ROOT structure. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxGetFileSizeWithLock	This routine gets the file size in the FCB header, using a lock to ensure that the 64-bit value is read consistently.
RxInferFileType	This routine tries to infer the file type (directory or non-directory) from a field in the RX_CONTEXT structure.
RxLockEnumerator	This routine is called from a network mini-redirector to enumerate the file locks on an FCB.
RxpDereferenceAndFinalizeNetFcb	This routine decrements the reference count and finalizes an FCB. This routine is only available on Windows Server 2003 Service Pack 1 (SP1) and later.
RxpDereferenceNetFcb	This routine decrements the reference count on an FCB.
RxpReferenceNetFcb	This routine increments the reference count on an FCB.
RxReference	This routine increments the reference count on an instance of several of the reference-counted data structures used by RDBSS.
RxSetSrvCallDomainName	This routine sets the domain name associated with any given server (SRV_CALL structure).

Note that a number of macros are also defined that provide wrappers around the **RxReference** and **RxDeference** routines for debugging. For more information about these macros, see Diagnostics and Debugging.

FCB Resource Synchronization

4/26/2017 • 2 min to read • Edit Online

The synchronization resources of interest to mini-redirector drivers are primarily associated with the FCB. There is a paging I/O resource and a regular resource. The paging I/O resource is managed internally by RDBSS. The only resource accessible to mini-redirector drivers is the regular resource, which should be accessed using the following supplied routines:

ROUTINE	DESCRIPTION
RxAcquireExclusiveFcbResourceInMRx	This routine acquires the FCB resource in the exclusive mode. This routine will wait for the FCB resource to be free if it was previously acquired; this routine does not return control until the exclusive resource has been acquired. This routine acquires the FCB resource even if the RX_CONTEXT structure associated with this FCB has been canceled.
RxAcquireSharedFcbResourceInMRx	This routine acquires the FCB resource in shared mode. This routine will wait for the FCB resource to be free if it was previously acquired exclusively; this routine does not return control until the shared resource has been acquired. This routine acquires the FCB resource even if the RX_CONTEXT structure associated with this FCB has been canceled.
RxAcquireSharedFcbResourceInMRxEx	This routine acquires the FCB resource in shared mode. This routine will wait for the FCB resource to be free if it was previously acquired exclusively; this routine does not return control until the shared resource has been acquired. This routine acquires the FCB resource even if the RX_CONTEXT structure associated with this FCB has been canceled. This routine is only available on Windows Server 2003 Service Pack 1 (SP1) and later.
RxReleaseFcbResourceForThreadInMRx	This routine frees the FCB resource previously acquired using **RxAcquireSharedFcbResourceInMRxEx**. This routine is only available on Windows Server 2003 Service Pack 1 and later.
RxReleaseFcbResourceInMRx	This routine frees the FCB resource previously acquired using **RxAcquireExclusiveFcbResourceInMRx** or **RxAcquireSharedFcbResourceInMRx**.

The following macros are defined in the rxprocs.h header file to determine whether the current thread has access to the FCB regular resource.

MACRO	DESCRIPTION
RxFcbAcquiredShared (*RXCONTEXT*, *FCB*)	This macro checks if the current thread has access to the regular resource in shared mode. This macro calls the **ExIsResourceAcquiredSharedLite** routine.
RxIsFcbAcquiredShared (*FCB*)	This macro checks if the current thread has access to the regular resource in shared mode. This macro calls the **ExIsResourceAcquiredSharedLite** routine.
RxIsFcbAcquiredExclusive (*FCB*)	This macro checks if the current thread has access to the regular resource in exclusive mode. This macro calls the **ExIsResourceAcquiredExclusiveLite** routine.
RxIsFcbAcquired (*FCB*)	This macro checks if the current thread has access to the regular resource in either shared or exclusive mode. This macro calls the **ExIsResourceAcquiredSharedLite** and **ExIsResourceAcquiredExclusiveLite** routine.

Buffering State Control

4/26/2017 • 3 min to read • Edit Online

RDBSS provides a buffering manager, a mechanism for providing distributed cache coherency in conjunction with the various network mini-redirectors. This service is encapsulated in the buffering manager in RDBSS that processes requests to change the buffering state. The buffering manager in RDBSS tracks and initiates actions on all change buffering state requests generated by the various network mini redirectors as well as by RDBSS.

There are several common components in the implementation of cache coherency in a typical network mini-redirector:

- Modifications to create and open file routines.

  In this path, the type of buffering request is determined and the appropriate request is made to the server. On return from the network mini-redirector, and possibly the server, the buffering state associated with the FCB is updated based on the result of the create or open call.

- Modifications to receive an indication code to handle change buffering state notifications from the server.

  If such a request is detected, then the local mechanism to coordinate the buffering states must be triggered.

- A mechanism for changing the buffering state that is implemented as part of RDBSS. Any change buffering state request must identify the SRV_OPEN structure to which the request applies.

The amount of computational effort involved in identifying the SRV_OPEN structure depends upon the protocol and details of the network mini-redirector. In the SMB protocol, opportunistic locks (oplocks) provide the necessary infrastructure for cache coherency. In the SMB protocol implementation in Windows, the multiplex ID APIs provided by RDBSS are used. The server gets to pick the multiplex ID used for identifying a file opened at the server. The multiplex IDs are relative to the NET_ROOT (share) on which they are opened. Thus, every change buffering state request is identified by two keys: the NetRootKey and the SrvOpenKey, which need to be translated to the appropriate NET_ROOT and SRV_OPEN structure, respectively. To provide better integration with the resource acquisition/release mechanism and to avoid duplication of this effort in the various network mini redirectors, the RDBSS provides this service.

There are two routines provided in RDBSS for indicating buffering state changes to SRV_OPEN structures:

- **RxIndicateChangeOfBufferingState** for registering the request

- **RxIndicateChangeOfBufferingStateForSrvOpen** for associating a SRV_OPEN structure with the key

Note that the key associations are irreversible and will last the lifetime of the associated SRV_OPEN structure.

Network mini-redirectors that need an auxiliary mechanism for establishing the mapping from multiplex IDs to the SRV_OPEN structure can use **RxIndicateChangeOfBufferingState**, while the network mini-redirectors that do not require this assistance can use **RxIndicateChangeOfBufferingStateForSrvOpen**.

The buffering manager in RDBSS processes these requests in different stages. It maintains the requests received from the various underlying network mini-redirectors in one of several lists.

- The Dispatcher list contains all of the requests for which the appropriate mapping to a SRV_OPEN structure has not been established.

- The Handler list contains all of the requests for which the appropriate mapping has been established, but has not yet been processed.

- The LastChanceHandlerList contains all of the requests for which the initial processing was unsuccessful. This typically happens when the FCB was acquired in a shared mode at the time the change buffering state request was received. In such cases, the oplock break request can only be processed by a delayed worker thread.

The change buffering state request processing in a network mini-redirector driver is intertwined with the FCB acquisition and release protocol. This helps to ensure shorter turnaround times.

RDBSS provides the following routines to manage buffering state that can be used by network mini-redirector drivers:

ROUTINE	DESCRIPTION
RxChangeBufferingState	This routine is called to process a buffering state change request.
RxIndicateChangeOfBufferingState	This routine is called to register a buffering state change request (an oplock break indication, for example) for later processing.
RxIndicateChangeOfBufferingStateForSrvOpen	This routine is called to register a buffering state change request (an oplock break indication, for example) for later processing.

Low I/O Routines

4/26/2017 • 1 min to read • Edit Online

Low I/O routines represent the basic IRP_MJ_XXX asynchronous operations on a file object (open, close, read, and write, for example). RDBSS provides some convenience routines that are used with low I/O operations by a network mini-redirector. The RDBSS low I/O routines include the following:

ROUTINE	DESCRIPTION
RxLowIoCompletion	This routine must be called by the low I/O routines of a network mini-redirector driver when processing is complete, if the routine initially returned as pending.
RxLowIoGetBufferAddress	This routine returns the buffer that corresponds to the MDL from the **LowIoContext** structure of an RX_CONTEXT structure.
RxMapSystemBuffer	This routine returns the system buffer address from the I/O request packet (IRP).
RxNewMapUserBuffer	This routine returns the address of the user buffer used for low I/O. Note that this routine is only available on Windows XP and Windows 2000.

Name Cache Management

4/26/2017 • 3 min to read • Edit Online

The NAME_CACHE structure caches the name strings of recent operations performed at the server so the client can suppress redundant requests. For example, if an open request has recently failed with a "file not found" message and the client application tries the open request again with an upper-case string, and the network mini-redirector does not support case-sensitive names, RDBSS can fail the request immediately without hitting the server.

In general, the algorithm is to put a time window and operation count limit on the NAME_CACHE entry. The time window is usually two seconds. So if the NAME_CACHE entry is greater than two seconds, the match will fail and the request will go to the server. If the request fails again at the server, the NAME_CACHE entry is updated with another two-second window. If the request operation count doesn't match, then one or more requests have been sent to the server, which could make this NAME_CACHE entry invalid. So again, this operation will be sent to the server.

A NAME_CACHE structure has a public portion exposed to the network mini-redirector, MRX_NAME_CACHE, and a private section for use solely by RDBSS. The mini-redirector portion has a context field, NTSTATUS, for the result of a prior server operation on this name entry and a context extension pointer for some additional mini-redirector specific storage that can be co-allocated with the NAME_CACHE structure. For more information, see **RxNameCacheInitialize**.

For Windows networking, the SMB operation count is an example of a mini-redirector-specific state, which could be saved in the context field of MRX_NAME_CACHE. When **RxNameCacheCheckEntry** is called, it will perform an equality check between the context field and a supplied parameter as part of finding a match in the name cache. When a NAME_CACHE entry is created or updated, it is the network mini-redirector's job to supply an appropriate value for this field and the lifetime, in seconds, for the NAME_CACHE entry.

The private RDBSS portion of the NAME_CACHE structure contains the name as a Unicode string, a hash value of the name to speed lookups, an expiration time of the entry, and a flag that indicates whether the server supports case-sensitive names.

The NAME_CACHE_CONTROL structure manages a given name cache. It has a free list, an active list, and a lock to synchronize updates. The NAME_CACHE_CONTROL structure also has fields to store the current number of NAME_CACHE entries allocated, a value for the maximum number of entries to be allocated, the size of any additional network-mini-redirector storage used for each NAME_CACHE entry, and values for statistics (the number of times the cache was updated, checked, a valid match was returned, and when the network-mini-redirector saved a network operation). The **MaximumEntries** field limits the number of NAME_CACHE entries created in case a poorly behaved program were to generate a large number of open requests with bad file names that consume large quantities of memory.

Currently there are name caches maintained by RDBSS for OBJECT_NAME_NOT_FOUND. For this name cache, a two-second window is maintained, which is invalidated if any operation is sent to the server. This could happen when the client application has a file (sample1) open that an application on the server could use to signal the creation of a different file (sample2) on the server. When the client reads the first file (sample1) and learns that the second file (sample2) has been created on the server, then a hit in the name cache that matches the second file (sample2) cannot return an error. This optimization only handles the case of successive file open operations on the same file that does not yet exist. This scenario happens using Microsoft Word.

The RDBSS name cache management routines include the following:

ROUTINE	DESCRIPTION
RxNameCacheActivateEntry	This routine takes a name cache entry and updates the expiration time and the network mini-redirector context. It then puts the entry on the active list.
RxNameCacheCheckEntry	This routine checks a NAME_CACHE entry for validity.
RxNameCacheCreateEntry	This routine allocates and initializes a NAME_CACHE structure with the given name string. It is expected that the caller will then initialize any additional network mini-redirector elements of the name cache context and then put the entry on the name cache active list.
RxNameCacheExpireEntry	This routine puts a NAME_CACHE entry on the free list.
RxNameCacheExpireEntryWithShortName	This routine expires all of the NAME_CACHE entries whose name prefix matches the given short file name.
RxNameCacheFetchEntry	This routine looks for a match with a specified name string for a NAME_CACHE entry.
RxNameCacheFinalize	This routine releases the storage for all of the NAME_CACHE entries associated with a NAME_CACHE_CONTROL structure.
RxNameCacheFreeEntry	This routine releases the storage for a NAME_CACHE entry and decrements the count of NAME_CACHE cache entries associated with a NAME_CACHE_CONTROL structure.
RxNameCacheInitialize	This routine initializes a name cache (a NAME_CACHE_CONTROL structure).

Prefix Table Management

4/26/2017 • 2 min to read • Edit Online

RDBSS defines data structures that enable use of prefix tables to catalog SRV_CALL, NET_ROOT, and V_NET_ROOT names.

The current implementation of name management in RDBSS uses a table that has the following components:

- A queue of inserted names
- A version stamp
- A table lock resource that controls table access
- A value that indicates whether name matches are case insensitive
- A bucket of hash value entries for this prefix table

The table lock resource is used in the normal way: shared for lookup operations, exclusive for change operations.

The version stamp changes with each change. The reason for the queue is that the prefix table package allows multiple callers to be enumerating at a time. The queue of inserted names and version stamp allow multiple callers to be enumerating simultaneously. The queue could be used as a faster lookup for file names, but the prefix table is definitely the correct approach for NET_ROOT structures.

These prefix table management routines are used internally by RDBSS in response to a call from MUP to claim a name or to form the create path for a NET_ROOT structure. These RDBSS prefix table management routines can also be used by network mini-redirectors, as long as the appropriate lock is acquired before accessing the table and the lock is released when work is completed. The normal use by a driver would be as follows:

- Acquire a shared lock by calling **RxAcquirePrefixTableLockShared**.
- Look up a name by calling **RxPrefixTableLookupName**.
- Release the shared lock by calling **RxReleasePrefixTableLock**.

Note that certain routines are implemented only on Windows XP and previous versions of Windows. **RxPrefixTableLookupName** is the only prefix table management routine implemented on all versions of Windows

The RDBSS prefix table management routines include the following:

ROUTINE	DESCRIPTION
RxpAcquirePrefixTableLockExclusive	This routine acquires an exclusive lock on a prefix table used to catalog SRV_CALL and NET_ROOT names. This routine is only available on Windows XP and Windows 2000. This routine is used internally by RDBSS and should not be used by network mini-redirectors.

ROUTINE	DESCRIPTION
RxpAcquirePrefixTableLockShared	This routine acquires a shared lock on a prefix table used to catalog SRV_CALL and NET_ROOT names. This routine is only available on Windows XP and Windows 2000. This routine is used internally by RDBSS and should not be used by network mini-redirectors.
RxPrefixTableLookupName	The routine looks up a name in a prefix table used to catalog SRV_CALL and NET_ROOT names and converts from the underlying pointer to the containing structure.
RxpReleasePrefixTableLock	This routine releases a lock on a prefix table used to catalog SRV_CALL and NET_ROOT names. This routine is only available on Windows XP and Windows 2000. This routine is used internally by RDBSS and should not be used by network mini-redirectors.

Starting with Windows Server 2003, the routines mentioned in the previous table, except **RxPrefixTableLookupName**, are replaced by macros.The following macros are defined that call the prefix table routines with fewer parameters.

MACRO	DESCRIPTION
RxAcquirePrefixTableLockExclusive (*TABLE*, *WAIT*)	This macro acquires the prefix table lock in exclusive mode for change operations.
RxAcquirePrefixTableLockShared (*TABLE*, *WAIT*)	This macro acquires the prefix table lock in shared mode for lookup operations.
RxIsPrefixTableLockAcquired (*TABLE*)	This macro indicates if the prefix table lock was acquired in either exclusive or shared mode.
RxIsPrefixTableLockExclusive (*TABLE*)	This macro indicates if the prefix table lock was acquired in exclusive mode.
RxReleasePrefixTableLock (*TABLE*)	This macro frees the prefix table lock.

Purging and Scavenging Control

4/26/2017 • 1 min to read • Edit Online

RDBSS provides a number of routines to purge and scavenge FOBX structures when they are no longer needed.

At cleanup, there are no more user handles associated with the file object. In such cases, the time window between close and cleanup is dictated by the additional references maintained by Memory Manager and Cache Manager. RDBSS uses a scavenger process that runs on a separate thread to scavenge and purge unneeded FOBX and other structures.

Currently, scavenging has been implemented for SRV_CALL, NET_ROOT and V_NET_ROOT structures. The FCB scavenging is handled separately. The FOBX can and should always be synchronously finalized. The only data structure that will have to be potentially enabled for scavenged finalization are SRV_OPEN structures.

The scavenger process as it is implemented in RDBSS currently will not consume any system resources until there is a need for scavenged finalization. The first entry to be marked for scavenged finalization will result in a timer request being posted for the scavenger. In the current implementation, the timer requests are posted as one-shot timer requests. This implies that there are no guarantees regarding the time interval within which the entries will be finalized. The scavenger activation mechanism is a potential candidate for fine tuning at a later stage.

The RDBSS purging and scavenging routines include the following:

ROUTINE	DESCRIPTION
RxPurgeAllFobxs	This routine purges all of the FOBX structures associated with a network mini-redirector.
RxPurgeRelatedFobxs	This routine purges all of the FOBX structures associated with a NET_ROOT structure.
RxScavengeAllFobxs	This routine scavenges all of the FOBX structures associated with a given network mini-redirector device object.
RxScavengeFobxsForNetRoot	This routine scavenges all of the FOBX structures associated with a given NET_ROOT structure.

Multiplex ID Management

4/26/2017 • 3 min to read • Edit Online

RDBSS defines a multiplex ID (MID), a 16-bit value, that can be used by both the network client (mini-redirector) and the server to distinguish between the concurrently active requests on any connection. A network redirector can associate a MID with any arbitrary context or internal data structure that it uses. It is completely at the option of the network redirector whether MIDs are allocated and used.

A MID, as defined by RDBSS, is part of a MID_ATLAS data structure that has been designed to meet several criterion. Associated with a MID_ATLAS data structure are a series of one or more MID_MAP data structures used to map MIDs to associated contexts.

The MID_ATLAS data structure, the MID_MAP structure, and MIDs should scale well to handle the differing capabilities of various remote servers. For example, the typical LAN Manager server on Windows permits 50 outstanding requests on any connection. Some types of servers may support as few as one outstanding request, while gateway servers may desire this number to be very high (on the order of thousands of outstanding connections).

The two primary operations that need to be handled well are:

- Mapping a MID to the context associated with it. This routine will be invoked to process every packet received along any connection at both the client and the server (assuming the servers use MIDs).
- Generating a new MID for sending requests to the server. This routine will be used at the client for enforcement of maximum connection limits as well as for tagging each concurrent request with a unique ID.

The MID must be able to efficiently manage the unique tagging and identification of a number of MIDs (typically 50) from a possible combination of 65,536 values. In some cases, it might make sense to create a small MID_ATLAS structure to save kernel memory used by the MID_MAP structure and expand the size of the MID_ATLAS structure if needed to efficiently handle greater usage. To ensure a proper time-space tradeoff, the lookup is organized as a three-level hierarchy. The 16 bits used to represent a MID are split up into three-bitfields. The length of the right-most field (least significant) is decided by the maximum number of MIDs that are allowed in the initial atlas. This maximum value is a parameter passed to the **RxCreateMidAtlas** routine when the MID_ATLAS data structure is first created. This maximum value determines the initial size of the MID_ATLAS data structure that is created and how many MID_MAP data structures can be accommodated. The remaining length is split up equally between the next two fields, which determine the maximum size of possible subordinate MID_ATLAS structures that can be defined to expand and extend an existing MID_ATLAS into a three-level hierarchy of MID_MAP data structures. So, each MID_ATLAS data structure can contain the maximum number of MID_MAP structures or a pointer to one subordinate MID_ATLAS and the MID_MAP structures.

For example, if a maximum of 50 MIDs are allocated on creation , the length of the first field is 6 (64 (2 ** 6) is greater than 50). The remaining length is split into two fields of 5 bits each for the second and third hierarchical levels so that an existing MID_ATLAS data structure can be expanded to accommodate more MID_MAP entries.

RDBSS provides the following routines for creating and manipulating a MID_ATLAS data structure, associated MID_MAP data structures, and Multiplex IDs.

ROUTINE	DESCRIPTION
RxAssociateContextWithMid	This routine associates the supplied opaque context with an available MID from a MID_ATLAS structure.

ROUTINE	DESCRIPTION
RxCreateMidAtlas	This routine allocates a new instance of the MID_ATLAS data structure and initializes it.
RxDestroyMidAtlas	This routine destroys an existing instance of a MID_ATLAS data structure and frees the memory allocated to it. As a side effect it invokes the passed in context destructor on every valid context in the MID_ATLAS structure.
RxMapMidToContext	This routine maps a MID to its associated context in a MID_ATLAS structure.
RxMapAndDissociateMidFromContext	This routine maps a MID to its associated context in a MID_ATLAS structure and then disassociates the MID from the context.
RxReassociateMid	This routine reassociates a MID with an alternate context.

Connection Engine Management

4/26/2017 • 3 min to read • Edit Online

In RDBSS, the connection engine is designed to map and emulate the TDI specifications as closely as possible. This provides an efficient mechanism that fully exploits the underlying TDI implementation for use by network mini-redirectors.

While the RDBSS connection engine does abstract TDI, network redirectors are also free to communicate directly with TDI instead of using these RDBSS connection engine routines. The existing RDBSS connection engine routines that provide wrappers for TDI were developed to support Microsoft Networks, so they are very Windows-centric and may not be appropriate for other network directors. Also, the connection engine routines in RDBSS are to be removed from Windows operating systems released after Windows Server 2003. In the future, each network redirector will be responsible for developing the connection engine routines needed (to TDI or some other transport). For example, a WebDAV redirector could talk to some user-mode reflector process to send out HTTP packets (standard TCP/IP) rather than TDI.

The RDBSS connection engine routines deal with the following entities:

- Transports
- Transport addresses
- Transport connections
- Virtual circuits on a connection

The transports are bindings to the various transport service providers on any system. The transport addresses are the local connection end points. The connections are transport connections between endpoints. Each connection encapsulates a number of virtual circuits (typically one).

The following important data structures are created and manipulated by the various connection engine routines associated with RDBSS:

- RXCE_TRANSPORT--encapsulates all of the parameters for a transport
- RXCE_ADDRESS--encapsulates all of the parameters for a transport address
- RXCE_CONNECTION--encapsulates all of the parameters for a transport connection
- RXCE_VC--encapsulates all of the parameters for a virtual circuit on a transport connection

Network mini-redirector drivers can use these data structures and invoke the routines provided for each type to build and tear down the connection engine portions. These routines do not allocate or free the memory associated with these structures. This provides a flexible mechanism for mini-redirector drivers to manage instances of these connection engine data structures.

The four connection engine types described above are tagged at the beginning of each data structure with a special RXCE_SIGNATURE signature that is used extensively by RDBSS for validation.

RDBSS provides the following connection engine routines that can be used by network mini-redirector drivers.

ROUTINE	DESCRIPTION

ROUTINE	DESCRIPTION
RxCeAllocateIrpWithMDL	This routine allocates an IRP for use by the connection engine and associates an MDL with the IRP. This routine is only available on Windows XP.
RxCeBuildAddress	This routine associates a transport address with a transport binding.
RxCeBuildConnection	This routine establishes a connection between a local RDBSS connection address and a given remote address. This routine should be called in the context of a system worker thread.
RxCeBuildConnectionOverMultipleTransports	This routine establishes a connection between a local RDBSS connection address and a given remote address and supports multiple transports. A set of local addresses are specified and this routine attempts to connect to the target server through all of the transports associated with the local addresses. One connection is chosen as the winner depending on the connect options. This routine must be called in the context of a system worker thread.
RxCeBuildTransport	This routine binds an RDBSS transport to a specified transport name.
RxCeBuildVC	This routine adds a virtual circuit to a specified connection.
RxCeCancelConnectRequest	This routine cancels a previously issued connection request. Note that this routine is not currently implemented.
RxCeFreeIrp	This routine frees an IRP used by the connection engine. This routine is only available on Windows XP.
RxCeInitiateVCDisconnect	This routine initiates a disconnect on the virtual circuit. This routine must be called in the context of a system worker thread.
RxCeQueryAdapterStatus	This routine returns the ADAPTER_STATUS structure for a given transport.
RxCeQueryInformation	This routine queries information that pertains to a connection.

ROUTINE	DESCRIPTION
RxCeQueryTransportInformation	This routine returns the transport information about the connection count and quality of service for a given transport.
RxCeSend	This routine sends a TSDU along the specified connection on a virtual circuit.
RxCeSendDatagram	This routine sends a TSDU to a specified transport address.
RxCeTearDownAddress	This routine removes a transport address from a transport binding.
RxCeTearDownConnection	This routine tears down a given connection.
RxCeTearDownTransport	This routine unbinds from the transport specified.
RxCeTearDownVC	This routine tears down a virtual connection.

Note TDI will not be supported in Microsoft Windows versions after Windows Vista. Use Windows Filtering Platform or Winsock Kernel instead.

The Kernel Network Mini-Redirector Driver

4/26/2017 • 3 min to read • Edit Online

A kernel network mini-redirector driver implements a number of callback routines that are used by the Redirected Drive Buffering Subsystem (RDBSS) to communicate with the driver. In the remainder of this document, a kernel network mini-redirector driver will be referred to as a network mini-redirector driver.

When a network mini-redirector driver first starts (in its **DriverEntry** routine), the driver calls the RDBSS **RxRegisterMinirdr** routine to register the network mini-redirector driver with RDBSS. The network mini-redirector driver passes in a MINIRDR_DISPATCH structure, which includes configuration data along with pointers to the routines that the network mini-redirector driver implements (a dispatch table).

A network mini-redirector can choose to implement only some of these routines. Any routine that is not implemented by the network mini-redirector should be set to a **NULL** pointer in the MINIRDR_DISPATCH structure passed to **RxRegisterMinirdr**. RDBSS will only call routines implemented by the network mini-redirector.

One special category of routines implemented by a network mini-redirector are the low I/O operations that represent the traditional file I/O calls for read, write, and other file operations. All of the low I/O routines can be called asynchronously by RDBSS. A kernel driver for a network mini-redirector must make certain that any low I/O routines that are implemented can be safely called asynchronously. The low I/O routines are passed in as an array of routine pointers as part of the MINIRDR_DISPATCH structure from the **DriverEntry** routine. The value of the array entry is the low I/O operation to perform. All of the low I/O routines expect a pointer to an RX_CONTEXT structure to be passed in as a parameter. The RX_CONTEXT data structure has a **LowIoContext.Operation** member that also specifies the low I/O operation to perform. It is possible for several of the low I/O routines to point to the same routine in a network mini-redirector driver since this **LowIoContext.Operation** member can be used to specify the low I/O operation requested. For example, all of the low I/O calls related to file locks could call the same low I/O routine in the network mini-redirector and this routine could use the **LowIoContext.Operation** member to specify the lock or unlock operation requested.

RDBSS also assumes asynchronous operation for a few other routines implemented by a network mini-redirector. These routines are used for establishing a connection with a remote resource. Since connection operations can take a considerable amount of time to complete, RDBSS assumes these routines are implemented as asynchronous operations.

RDBSS assumes that all routines implemented by a network mini-redirector other than the low I/O and connection-related routines are based on synchronous calls. However, this is subject to change in future releases of the Windows operating system.

All of the routines implemented by a network mini-redirector return an NTSTATUS value on completion. Most routines return STATUS_SUCCESS on success or an appropriate NTSTATUS value. In addition to return values specific to a particular routine, there are two generic categories of errors that can be returned for most routines :

- Network errors
- Authentication errors

Possible network errors include the following:

STATUS_IO_TIMEOUT
The I/O request to the remote server has timed out.

STATUS_BAD_NETWORK_PATH
The I/O request was to a network path that does not exist. This error can occur if a directory was renamed or

deleted.

STATUS_NETWORK_UNREACHABLE
The network is unreachable from the client.

STATUS_REMOTE_NOT_LISTENING
The remote server is not listening for connections.

STATUS_USER_SESSION_DELETED
The user session on the server has been deleted. The session may have timed out, the server may have been restarted causing all existing user sessions to be deleted, or an administrator on the server may have forced a delete of the user session .

STATUS_CONNECTION_DISCONNECTED
The connection to the remote server was disconnected.

STATUS_NETWORK_NAME_DELETED
The I/O request is for a network name that has been deleted.

Possible authentication errors include the following:

STATUS_LOGON_FAILURE
The login request to the remote server failed.

STATUS_NETWORK_CREDENTIAL_CONFLICT
There was a conflict with the network credentials that were presented.

STATUS_DOWNGRADE_DETECTED
A change of the network protocol used by the client to communicate with the server was detected by the server and the change was to an older version of the protocol.

STATUS_LOGIN_WKSTA_RESTRICTION
There is a restriction on workstation logins to the server.

The following sections discuss in detail the routines that can be implemented by a network mini-redirector:

Routines Implemented by the Kernel Network Mini-Redirector

Routines Not Used by RDBSS

The routines implemented by a network mini-redirector can be organized into the following categories based on their function:

Connection and Name Resolution

Driver Start, Stop, and Device Control

File System Object Creation and Deletion

File System Object I/O Routines

File System Object Query and Set Routines

Miscellaneous Network Mini-Redirector Routines

Routines Implemented by the Kernel Network Mini-Redirector

4/26/2017 • 7 min to read • Edit Online

The following routines can be implemented by a network mini-redirector:

ROUTINE	DESCRIPTION
MRxAreFilesAliased	RDBSS calls this routine to query the network mini-redirector if two file control block (FCB) structures represent the same file. RDBSS calls this routine when processing two files that appear to be the same but have different names (an MS-DOS short name and a long name, for example).
MRxCleanupFobx	RDBSS calls this routine to request that the network mini-redirector close a file system object extension structure. RDBSS issues this call in response to receiving an **IRP_MJ_CLEANUP** on a file object.
MRxCloseSrvOpen	RDBSS calls this routine to request that the network mini-redirector close a SRV_OPEN structure.
MRxCollapseOpen	RDBSS calls this routine to request that the network mini-redirector collapse an open file system request onto an existing SRV_OPEN.
MRxCompleteBufferingStateChangeRequest	RDBSS calls this routine to notify the network mini-redirector that a buffering state change request has been completed. For example, the SMB redirector uses this routine to send an oplock break response or to close the handle on an oplock break if the file is no longer in use. Byte range locks that need to be flushed out to the server are passed to the network mini-redirector in the **LowIoContext.ParamsFor.Locks.LockList** member of the RX_CONTEXT structure.
MRxComputeNewBufferingState	RDBSS calls this routine to request that the network mini-redirector compute a new buffering state change.
MRxCreate	RDBSS calls this routine to request that the network mini-redirector create a file system object. RDBSS issues this call in response to receiving an **IRP_MJ_CREATE**.
MRxCreateSrvCall	RDBSS calls this routine to request that the network mini-redirector create a SRV_CALL structure and establish a connection with a server.

ROUTINE	DESCRIPTION
MRxCreateVNetRoot	RDBSS calls this routine to request that the network mini-redirector create a V_NET_ROOT structure.
MRxDeallocateForFcb	RDBSS calls this routine to request that the network mini-redirector deallocate an FCB. This call is in response to a request to close a file system object.
MRxDeallocateForFobx	RDBSS calls this routine to request that the network mini-redirector deallocate an FOBX structure. This call is in response to a request to close a file system object.
MRxDevFcbXXXControlFile	RDBSS calls this routine to pass a device FCB control request to the network mini-redirector. RDBSS issues this call in response to receiving an **IRP_MJ_DEVICE_CONTROL**, **IRP_MJ_FILE_SYSTEM_CONTROL**, or **IRP_MJ_INTERNAL_DEVICE_CONTROL** on a device FCB.
MRxExtendForCache	RDBSS calls this routine to request that a network mini-redirector extend a file when the file is being cached by the cache manager.
MRxExtendForNonCache	RDBSS calls this routine to request that a network mini-redirector extend a file when the file is not being cached by the cache manager.
MRxExtractNetRootName	RDBSS calls this routine to request that a network mini-redirector extract the name of the NET_ROOT from a given pathname.
MRxFinalizeNetRoot	RDBSS calls this routine to request that a network mini-redirector finalize a NET_ROOT object.
MRxFinalizeSrvCall	RDBSS calls this routine to request that a network mini-redirector finalize a SRV_CALL structure used for establishing connection with a server.
MRxFinalizeVNetRoot	RDBSS calls this routine to request that a network mini-redirector finalize a V_NET_ROOT object.
MRxFlush	RDBSS calls this routine to request that a network mini-redirector flush a file system object. RDBSS issues this call in response to receiving an **IRP_MJ_FLUSH_BUFFERS**.

ROUTINE	DESCRIPTION
MRxForceClosed	RDBSS calls this routine to request that a network mini-redirector force a close. This routine is called when the condition of the SRV_OPEN structure is bad or the SRV_OPEN structure is marked as closed.
MRxGetConnectionId	RDBSS calls this routine to request that a network mini-redirector return a connection ID for the connection which can be used for handling multiple sessions. If connection IDs are supported by the network mini-redirector, then the returned connection ID is appended to the connection structures stored in the name table. RDBSS considers the connection ID an opaque blob, and does a byte-by-byte comparison of the connection ID blob while looking up the net-name table for a given name.
MRxIsLockRealizable	RDBSS calls this routine to request that a network mini-redirector indicate whether byte-range locks are supported on specific NET_ROOT structure.
MRxIsValidDirectory	RDBSS calls this routine to request that a network mini-redirector indicate if a path is a valid directory.
MRxLowIOSubmit[LOWIO_OP_EXCLUSIVELOCK]	RDBSS calls this routine to request that a network mini-redirector open an exclusive lock on a file object. RDBSS issues this call in response to receiving an **IRP_MJ_LOCK_CONTROL** with a minor code of IRP_MN_LOCK and when **IrpSp->Flags** has the SL_EXCLUSIVE_LOCK bit set.
MRxLowIOSubmit[LOWIO_OP_FSCTL]	RDBSS calls this routine to pass a file system control request to the network mini-redirector. RDBSS issues this call in response to receiving an **IRP_MJ_FILE_SYSTEM_CONTROL**.
MRxLowIOSubmit[LOWIO_OP_IOCTL]	RDBSS calls this routine to pass an I/O system control request to the network mini-redirector. RDBSS issues this call in response to receiving an **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL**.
MRxLowIOSubmit[LOWIO_OP_NOTIFY_CHANGE_DIRECTORY]	RDBSS calls this routine to issue a request to the network mini-redirector for a directory change notification operation. RDBSS issues this call in response to receiving an **IRP_MJ_DIRECTORY_CONTROL**.
MRxLowIOSubmit[LOWIO_OP_READ]	RDBSS calls this routine to issue a read request to the network mini-redirector. RDBSS issues this call in response to receiving an **IRP_MJ_READ**.

ROUTINE	DESCRIPTION
MRxLowIOSubmit[LOWIO_OP_SHAREDLOCK]	RDBSS calls this routine to request that a network redirector open a shared lock on a file object. RDBSS issues this call in response to receiving an **IRP_MJ_LOCK_CONTROL** with a minor code of IRP_MN_LOCK and when **IrpSp->Flags** has the SL_EXCLUSIVE_LOCK bit set.
MRxLowIOSubmit[LOWIO_OP_UNLOCK]	RDBSS calls this routine to request that a network mini-redirector remove a single lock on a file object. RDBSS issues this call in response to receiving an IRP_MJ_LOCK_CONTROL with a minor code of IRP_MN_UNLOCK_SINGLE.
MRxLowIOSubmit[LOWIO_OP_UNLOCK_MULTIPLE]	RDBSS calls this routine to request that the network mini-redirector remove multiple locks held on a file object. RDBSS issues this call in response to receiving an IRP_MJ_LOCK_CONTROL with a minor code of IRP_MN_UNLOCK_ALL or IRP_MN_UNLOCK_ALL_BY_KEY. The ranges to be unlocked are specified in the **LowIoContext.ParamsFor.Locks.LockList** member of the RX_CONTEXT.
MRxLowIOSubmit[LOWIO_OP_WRITE]	RDBSS calls this routine to issue a write request to the network mini-redirector. RDBSS issues this call in response to receiving an **IRP_MJ_WRITE**.
MRxPreparseName	RDBSS calls this routine to give a network mini-redirector the opportunity to preparse a name.
MRxQueryDirectory	RDBSS calls this routine to request that a network mini-redirector query information on a file directory system object.
MRxQueryEaInfo	RDBSS calls this routine to request that a network mini-redirector query extended attribute information on a file system object. RDBSS issues this call in response to receiving an **IRP_MJ_QUERY_EA**.
MRxQueryFileInfo	RDBSS calls this routine to request that a network mini-redirector query file information on a file system object. RDBSS issues this call in response to receiving an **IRP_MJ_QUERY_INFORMATION**.
MRxQueryQuotaInfo	RDBSS calls this routine to request that a network mini-redirector query quota information on a file system object. RDBSS issues this call in response to receiving an **IRP_MJ_QUERY_QUOTA**.

ROUTINE	DESCRIPTION
MRxQuerySdInfo	RDBSS calls this routine to request that a network mini-redirector query security descriptor information on a file system object. RDBSS issues this call in response to receiving an **IRP_MJ_QUERY_SECURITY**.
MRxQueryVolumeInfo	RDBSS calls this routine to request that a network mini-redirector query volume information. RDBSS issues this call in response to receiving an **IRP_MJ_QUERY_VOLUME_INFORMATION**.
MRxSetEaInfo	RDBSS calls this routine to request that a network mini-redirector set extended attribute information on a file system object. RDBSS issues this call in response to receiving an **IRP_MJ_SET_EA**.
MRxSetFileInfo	RDBSS calls this routine to request that a network mini-redirector set file information on a file system object. RDBSS issues this call in response to receiving an **IRP_MJ_SET_INFORMATION**.
MRxSetFileInfoAtCleanup	RDBSS calls this routine to request that a network mini-redirector set file information on a file system object at cleanup. RDBSS issues this call during cleanup when an application closes a handle but before the file object is closed by the I/O Manager.
MRxSetQuotaInfo	RDBSS calls this routine to request that a network mini-redirector set quota information on a file system object. RDBSS issues this call in response to receiving an **IRP_MJ_SET_QUOTA**.
MRxSetSdInfo	RDBSS calls this routine to request that a network mini-redirector set security descriptor information on a file system object. RDBSS issues this call in response to receiving an **IRP_MJ_SET_SECURITY**.
MRxSetVolumeInfo	RDBSS calls this routine to request that a network mini-redirector set volume information. RDBSS issues this call in response to receiving an **IRP_MJ_SET_VOLUME_INFORMATION**.
MRxShouldTryToCollapseThisOpen	RDBSS calls this routine to request that a network mini-redirector indicate if RDBSS should try and collapse an open request onto an existing file system object.

ROUTINE	DESCRIPTION
MRxSrvCallWinnerNotify	This routine was originally designed to be called by RDBSS to notify a network mini-redirector that it was "the winner" when multiple redirectors could fulfill the request. The winning network mini-redirector is expected to create the SRV_CALL structure and establish a connection with the server. Under the current implementation of RDBSS, each network mini-redirector has its own copy of RDBSS, so there are no competing network redirectors at the RDBSS layer. This routine is called after every request to create a SRV_CALL structure. When multiple redirectors are installed for handling the same Universal Naming Convention (UNC) namespace, the redirector to service a request is chosen by the Multiple UNC Provider (MUP) based on the order of redirectors specified in the registry.
MRxStart	RDBSS calls this routine to start the network mini-redirector.
MRxStop	RDBSS calls this routine to stop the network mini-redirector.
MRxTruncate	RDBSS calls this routine to request that a network mini-redirector truncate a file system object.
MRxZeroExtend	RDBSS calls this routine to request that a network mini-redirector extend a file system object filling the file with zeros on cleanup if the file size is greater than the valid data length of the FCB.

Routines Not Used by RDBSS

4/26/2017 • 1 min to read • Edit Online

A number of routines listed in the MINIRDR_DISPATCH structure are not called or used by RDBSS. It is unnecessary for a network mini-redirector to implement any of these routines since they will never be called. A network mini-redirector should set a **NULL** pointer for all these routines in the MINIRDR_DISPATCH structure passed in to **RxRegisterMinirdr** from its **DriverEntry** routine.

The following is a complete list of the routines not used by RDBSS:

- **MRxCancel**
- **MRxCancelCreateSrvCall**
- **MRxClosedFcbTimeOut**
- **MRxClosedSrvOpenTimeOut**
- **MRxClosePrintFile**
- **MRxEnumeratePrintQueue**
- **MRxLowIOSubmit[LOWIO_OP_CLEAROUT]**
- **MRxOpenPrintFile**
- **MRxUpdateNetRootState**
- **MRxWritePrintFile**

Connection and Name Resolution

4/26/2017 • 3 min to read • Edit Online

Network mini-redirectors establish connections to remote servers and handle name resolution using several routines. RDBSS abstracts this process into several structures including the SRV_CALL, NET_ROOT, and V_NET_ROOT structures that are reference counted. In network mini-redirector terms, the establishment of a connection is often called a "tree connect." This connection establishment requires creating a SRV_CALL, a NET_ROOT, and a V_NET_ROOT structure. So the normal procedure would be a call to the network mini-redirector **MRxCreateSrvCall** routine followed by a call to the **MRxSrvCallWinnerNotify** and **MRxCreateVNetRoot** routines. These calls are usually issued in response to a request from a user-mode application or service to mount a drive (**net use x: \\server\public**, for example). These calls can also result from a request for a UNC file object (**notepad \\server\public\readme.txt**, for example). RDBSS handles both of these cases internally for the network mini-redirector and initiates the **MRxCreateSrvCall** sequence.

When a connection is deleted, similar finalize calls are made to tear down the SRV_CALL, and NET_ROOT structures and release any memory used. These calls to the network mini-redirector include **MRxFinalizeVNetRoot**, **MRxFinalizeNetRoot**, and **MRxFinalizeSrvCall**.

The original design for RDBSS was that various network mini-redirectors would all share a single copy of RDBSS, but this was not implemented. A holdover from this original design is that various network mini-redirectors compete to fulfill the request for a network connection (\\server\share, for example). The **MRxSrvCallWinnerNotify** routine is used by RDBSS to notify a network mini-redirector that it was the winner when multiple redirectors could fulfill the request. The winning network mini-redirector is expected to create the SRV_CALL and establish a connection with the server.

Under the implementation of RDBSS in Windows Server 2003, Windows XP, and Windows 2000, each network mini-redirector has its own copy of RDBSS so there are no competing network redirectors at the RDBSS layer. Each network mini-redirector (network provider) and its local copy of RDBSS is called in turn based on the order in a registry setting: The order in which providers are queried is controlled by the following registry value:

```
ProviderOrder
```

This registry value is located under the following registry key:

```
HKLM\CurrentControlSet\Control\NetworkProvider\Order
```

The **MRxSrvCallWinnerNotify** routine will be called after every request to create a SRV_CALL structure.

The following table lists the routines that can be implemented by a network mini-redirector for connection and name resolution operations.

ROUTINE	DESCRIPTION
MRxCreateSrvCall	RDBSS calls this routine to request that the network mini-redirector create a SRV_CALL structure and establish connection with a server.

ROUTINE	DESCRIPTION
MRxCreateVNetRoot	RDBSS calls this routine to request that the network mini-redirector create a V_NET_ROOT structure.
MRxExtractNetRootName	RDBSS calls this routine to request that a network mini-redirector extract the name from the NET_ROOT structure for a given pathname.
MRxFinalizeNetRoot	RDBSS calls this routine to request that a network mini-redirector finalize a NET_ROOT object.
MRxFinalizeSrvCall	RDBSS calls this routine to request that a network mini-redirector finalize a SRV_CALL structure used for establishing connection with a server.
MRxFinalizeVNetRoot	RDBSS calls this routine to request that a network mini-redirector finalize a V_NET_ROOT structure.
MRxPreparseName	RDBSS calls this routine to give a network mini-redirector the opportunity to preparse a name.
MRxSrvCallWinnerNotify	This routine was originally designed to be called by RDBSS to notify a network mini-redirector that it was the winner when multiple redirectors could fulfill the request. The winning network mini-redirector is expected to create the SRV_CALL structure and establish a connection with the server. Under the current implementation of RDBSS each network mini-redirector has its own copy of RDBSS, so there are no competing network redirectors at the RDBSS layer. This routine will be called before every request to create a SRV_CALL structure. When multiple redirectors are installed for handling the same UNC namespace, the redirector to service a request is chosen by MUP based on the order of redirectors specified in the registry.

Driver Start, Stop, and Device Control

4/26/2017 • 2 min to read • Edit Online

Driver registration is handled in the **DriverEntry** routine of a network mini-redirector driver. When a network mini-redirector first starts (in its **DriverEntry** routine), the driver must call the RDBSS **RxRegisterMinirdr** routine to register the network mini-redirector with RDBSS. The network mini-redirector passes in a MINIRDR_DISPATCH structure which includes configuration data and a table of routine pointers (a dispatch table) to the routines that the network mini-redirector driver implements.

The **MRxStart** and **MRxStop** routines must be implemented by the network mini-redirector driver to allow the driver to be started and stopped.

The sequence to start or stop the network mini-redirector is complex. This sequence is normally initiated by a user-mode application or service supplied with network mini-redirector driver to control the driver for administration and management purposes. The network mini-redirector can use a service that is configured to start automatically when the operating system starts. This service can request that the network mini-redirector be started whenever the operating system starts.

MRxStart is called by RDBSS when the **RxStartMinirdr** routine is called. The **RxStartMinirdr** routine is usually called as a result of an FSCTL or IOCTL request from a user-mode application or service to start the network mini-redirector. The call to **RxStartMinirdr** cannot be made from the **DriverEntry** routine of the network mini-redirector after a successful call to **RxRegisterMinirdr**because some of the start processing requires that the driver initialization be completed. Once the **RxStartMinirdr** call is received, RDBSS completes the start process by calling the **MRxStart** routine of the network mini-redirector. If the call to **MRxStart** returns success, RDBSS sets the internal state of the mini-redirector in RDBSS to RDBSS_STARTED.

MRxStop is called by RDBSS when the **RxStopMinirdr** routine is called. The RDBSS **RxStopMinirdr** routine is usually called as a result of an FSCTL or IOCTL request from a user-mode application or service to stop the network mini-redirector. This call can also be made from the network mini-redirector or, as part of shutdown process, by the operating system. Once the **RxStopMinirdr** call is received, RDBSS completes the process by calling the **MRxStop** routine of the network mini-redirector.

The **MRxDevFcbXXXControlFile** routine is used to receive requests from a user-mode application or service to control the network mini-redirector by making IOCTL or FSCTL calls on a device FCB.

In addition, there are two low I/O routines that handle IOCTL and FSCTL operations on the driver object: **MRxLowIOSubmit[LOWIO_OP_FSCTL]** and **MRxLowIOSubmit[LOWIO_OP_IOCTL]**.

A network mini-redirector can also use these low I/O routines to provide control and management of the network mini-redirector from a user-mode application or service.

The following table lists the routines that can be implemented by a network mini-redirector for start, stop, and device control operations.

ROUTINE	DESCRIPTION
MRxDevFcbXXXControlFile	RDBSS calls this routine to pass a device FCB control request to the network mini-redirector. RDBSS issues this call in response to receiving an IRP_MJ_DEVICE_CONTROL, IRP_MJ_FILE_SYSTEM_CONTROL, or IRP_MJ_INTERNAL_DEVICE_CONTROL on a device FCB.

ROUTINE	DESCRIPTION
MRxStart	RDBSS calls this routine to start the network mini-redirector.
MRxStop	RDBSS calls this routine to stop the network mini-redirector.

File System Object Creation and Deletion

4/26/2017 • 3 min to read • Edit Online

A number of routines implemented by a network mini-redirector are for file creation and deletion. RDBSS abstracts this process by creating several structures including the SRV_OPEN, FCB, and FOBX structure that are reference counted. Creating or opening a file system object normally requires creating SRV_OPEN, FCB, and FOBX structures. The normal procedure would be for RDBSS to call the **MRxCreate** routine implemented by the network mini-redirector to fill information into these structures. It is also possible to collapse an open/create file request on an existing SRV_OPEN structure using the **MRxShouldTryToCollapseThisOpen** and **MRxCollapseOpen** routines.

In the context of a network redirector, a file object refers to a file control block (FCB) structure and a file object extension (FOBX) structure. There is a one to one correspondence between file objects and FOBXs. Many file objects will refer to the same FCB structure, which represents a single file somewhere on a remote server. A client can have several different open requests (NtCreateFile requests) on the same FCB and each of these will create a new file object. RDBSS and network mini-redirectors can choose to send fewer **MRxCreate** requests than the NtCreateFile requests that are received, in effect sharing a server open request (SRV_OPEN) among several FOBXs.

RDBSS and a network mini-redirector do not necessarily close the SRV_OPEN structures when the user closes a file. RDBSS can try to reuse the SRV_OPEN structure and the data without any contact with the server. Some Windows applications can open, read, and close a file and then quickly reopen the same file. In these cases, reusing the SRV_OPEN structures can improve performance.

There are several routines used to close and finalize these data structures and free any memory or other resources used when they are no longer needed. These routines include **MRxCleanupFobx**, **MRxCloseSrvOpen**, **MRxDeallocateForFcb**, **MRxDeallocateForFobx**, and **MRxForceClosed**.

The following table lists the routines that can be implemented by a network mini-redirector for file system object creation and deletion operations.

ROUTINE	DESCRIPTION
MRxAreFilesAliased	RDBSS calls this routine to query the network mini-redirector if two FCB objects represent the same file. RDBSS calls this routine when processing two files that appear to be the same but have different names (an MS-DOS short name and a long name, for example).
MRxCleanupFobx	RDBSS calls this routine to request that the network mini-redirector close a file system object extension. RDBSS issues this call in response to receiving an IRP_MJ_CLEANUP on a file object.
MRxCloseSrvOpen	RDBSS calls this routine to request that the network mini-redirector close a SRV_OPEN object.
MRxCollapseOpen	RDBSS calls this routine to request that the network mini-redirector collapse an open file system request onto an existing SRV_OPEN.

ROUTINE	DESCRIPTION
MRxCreate	RDBSS calls this routine to request that the network mini-redirector create a file system object. This call is issued by RDBSS in response to receiving an IRP_MJ_CREATE.
MRxDeallocateForFcb	RDBSS calls this routine to request that the network mini-redirector deallocate an FCB. This call is in response to a request to close a file system object.
MRxDeallocateForFobx	RDBSS calls this routine to request that the network mini-redirector deallocate an FOBX. This call is in response to a request to close a file system object.
MRxExtendForCache	RDBSS calls this routine to request that a network mini-redirector extend a file when the file is being cached by cache manager.
MRxExtendForNonCache	RDBSS calls this routine to request that a network mini-redirector extend a file when the file is not being cached by cache manager.
MRxFlush	RDBSS calls this routine to request that a network mini-redirector flush a file system object. RDBSS issue this call in response to receiving an IRP_MJ_FLUSH_BUFFERS.
MRxForceClosed	RDBSS calls this routine to request that a network mini-redirector force a close. This routine is called when the condition of the SRV_OPEN is not good or the SRV_OPEN is marked as closed.
MRxIsLockRealizable	RDBSS calls this routine to request that a network mini-redirector indicate whether byte-range locks are supported on this NET_ROOT.
MRxShouldTryToCollapseThisOpen	RDBSS calls this routine to request that a network mini-redirector indicate if RDBSS should try and collapse an open request onto an existing file system object.
MRxTruncate	RDBSS calls this routine to request that a network mini-redirector truncate a file system object.
MRxZeroExtend	RDBSS calls this routine to request that a network mini-redirector extend a file system object filling the file with zeros on cleanup if the file size is greater than the valid data length of the FCB.

File System Object I/O Routines

4/26/2017 • 3 min to read • Edit Online

The file system object I/O routines represent the traditional file I/O calls for read, write, and other file operations. These routines are often called "low I/O routines" in the network mini-redirector. RDBSS calls these routines in response to receiving a specific IRP

The low I/O routines are passed in as an array of routine pointers as part of the MINIRDR_DISPATCH structure passed to the **RxRegisterMinirdr** routine that is used to register the network mini-redirector at startup (in the **DriverEntry** routine). The value of the array entry is the low I/O operation to perform.

These low I/O or file system object I/O routines are normally called asynchronously by RDBSS. So a network mini-redirector must make certain that any low I/O routines that are implemented can be safely called asynchronously. It is also possible for a network mini-redirector to ignore the request for an asynchronous call and implement a routine to only operate synchronously. However, for certain calls that may take time to complete (read and write, for example), implementing these as synchronous operations can significantly reduce I/O performance for the entire operating system.

All the file system object I/O routines expect a pointer to an RX_CONTEXT structure to be passed in as a parameter. Before calling these routines, RDBSS sets the value of a number of members in the **LowIoContext** structure of the RX_CONTEXT. Some of the members in the **LowIoContext** structure are set for all routines, while some members are only set for specific routines. The RX_CONTEXT data structure contains the IRP that is being processed and has a **LowIoContext.Operation** member that specifies the low I/O operation to perform. It is possible for several of the low I/O routines to point to the same routine in a network mini-redirector because this **LowIoContext.Operation** member can be used to differentiate the low I/O operation requested. For example, all the I/O calls related to file locks could call the same low I/O routine in the network mini-redirector which uses the **LowIoContext.Operation** member to differentiate the lock or unlock operation requested.

Other file I/O routines other than the low I/O routines are based on synchronous calls, although this is subject to change in future releases of the Windows operating system.

While in the LowIo path, the **LowIoContext.ResourceThreadId** member of the RX_CONTEXT is guaranteed to indicate the owning thread that initiated the operation in RDBSS. The **LowIoContext.ResourceThreadId** member can be used to release the FCB resource on behalf of another thread. When an asynchronous routine completes, the FCB resource that was acquired from the initial thread can be released.

If an **MrxLowIoSubmit[LOWIO_OP_XXX]** routine is likely to take a long time to complete, the network mini-redirector driver should release the FCB resource before initiating the network communication. The FCB resource can be released by calling **RxReleaseFcbResourceForThreadInMRx**.

The following table lists the routines that can be implemented by a network mini-redirector for file system object I/O (low I/O) operations.

ROUTINE	DESCRIPTION
MRxLowIOSubmit[LOWIO_OP_EXCLUSIVELOCK]	RDBSS calls this routine to request that a network mini-redirector open an exclusive lock on a file object. RDBSS issues this call in response to receiving an IRP_MJ_LOCK_CONTROL with a minor code of IRP_MN_LOCK and IrpSp->Flags has the SL_EXCLUSIVE_LOCK bit set.

ROUTINE	DESCRIPTION
MRxLowIOSubmit[LOWIO_OP_FSCTL]	RDBSS calls this routine to pass a file system control request to the network mini-redirector. RDBSS issues this call in response to receiving an IRP_MJ_FILE_SYSTEM_CONTROL.
MRxLowIOSubmit[LOWIO_OP_IOCTL]	RDBSS calls this routine to pass an I/O system control request to the network mini-redirector. RDBSS issues this call in response to receiving an IRP_MJ_DEVICE_CONTROL or IRP_MJ_INTERNAL_DEVICE_CONTROL..
MRxLowIOSubmit[LOWIO_OP_NOTIFY_CHANGE_DIRECTORY]	RDBSS calls this routine to issue a request to the network mini-redirector for a directory change notification operation. RDBSS issues this call in response to receiving an IRP_MJ_DIRECTORY_CONTROL.
MRxLowIOSubmit[LOWIO_OP_READ]	RDBSS calls this routine to issue a read request to the network mini-redirector. RDBSS issues this call in response to receiving an IRP_MJ_READ.
MRxLowIOSubmit[LOWIO_OP_SHAREDLOCK]	RDBSS calls this routine to request that a network redirector open a shared lock on a file object. RDBSS issues this call in response to receiving an IRP_MJ_LOCK_CONTROL with a minor code of IRP_MN_LOCK and IrpSp->Flags does not have the SL_EXCLUSIVE_LOCK bit set.
MRxLowIOSubmit[LOWIO_OP_UNLOCK]	RDBSS calls this routine to request that a network mini-redirector remove a single lock on a file object. RDBSS issues this call in response to receiving an IRP_MJ_LOCK_CONTROL with a minor code of IRP_MN_UNLOCK_SINGLE.
MRxLowIOSubmit[LOWIO_OP_UNLOCK_MULTIPLE]	RDBSS calls this routine to request that the network mini-redirector remove multiple locks held on a file object. RDBSS issues this call in response to receiving an IRP_MJ_LOCK_CONTROL with a minor code of IRP_MN_UNLOCK_ALL or IRP_MN_UNLOCK_ALL_BY_KEY. The byte ranges to be unlocked are specified in the **LowIoContext.ParamsFor.Locks.LockList** member of the RX_CONTEXT.
MRxLowIOSubmit[LOWIO_OP_WRITE]	RDBSS calls this routine to issue a write request to the network mini-redirector. RDBSS issues this call in response to receiving an IRP_MJ_WRITE.

File System Object Query and Set Routines

4/26/2017 • 2 min to read • Edit Online

A number of routines that can be implemented by a network mini-redirector are for query and set operations on file system objects. RDBSS issues most of these calls in response to receiving an IRP to query or set information on a file object. So there is a direct correspondence between the IRP that RDBSS receives and the MRx query or set operation that RDBSS calls.

The following table lists the routines that can be implemented by a network mini-redirector for file system object query and set operations.

ROUTINE	DESCRIPTION
MRxIsValidDirectory	RDBSS calls this routine to request that a network mini-redirector indicate if a path is a valid directory.
MRxQueryDirectory	RDBSS calls this routine to request that a network mini-redirector query information on a file directory system object.
MRxQueryEaInfo	RDBSS calls this routine to request that a network mini-redirector query extended attribute information on a file system object. RDBSS issues this call in response to receiving an IRP_MJ_QUERY_EA.
MRxQueryFileInfo	RDBSS calls this routine to request that a network mini-redirector query file information on a file system object. RDBSS issues this call in response to receiving an IRP_MJ_QUERY_INFORMATION.
MRxQueryQuotaInfo	RDBSS calls this routine to request that a network mini-redirector query quota information on a file system object. RDBSS issues this call in response to receiving an IRP_MJ_QUERY_QUOTA.
MRxQuerySdInfo	RDBSS calls this routine to request that a network mini-redirector query security descriptor information on a file system object. RDBSS issues this call in response to receiving an IRP_MJ_QUERY_SECURITY.
MRxQueryVolumeInfo	RDBSS calls this routine to request that a network mini-redirector query volume information. RDBSS issues this call in response to receiving an IRP_MJ_QUERY_VOLUME_INFORMATION.

ROUTINE	DESCRIPTION
MRxSetEaInfo	RDBSS calls this routine to request that a network mini-redirector set extended attribute information on a file system object. RDBSS issues this call in response to receiving an IRP_MJ_SET_EA.
MRxSetFileInfo	RDBSS calls this routine to request that a network mini-redirector set file information on a file system object. RDBSS issues this call in response to receiving an IRP_MJ_SET_INFORMATION.
MRxSetFileInfoAtCleanup	RDBSS calls this routine to request that a network mini-redirector to set file information on a file system object at cleanup. RDBSS issues this call during cleanup when an application closes a handle but before the close.
MRxSetQuotaInfo	RDBSS calls this routine to request that a network mini-redirector set quota information on a file system object. RDBSS issues this call in response to receiving an IRP_MJ_SET_QUOTA.
MRxSetSdInfo	RDBSS calls this routine to request that a network mini-redirector set security descriptor information on a file system object. RDBSS issues this call in response to receiving an IRP_MJ_SET_SECURITY.
MRxSetVolumeInfo	RDBSS calls this routine to request that a network mini-redirector set volume information. RDBSS issues this call in response to receiving an IRP_MJ_SET_VOLUME_INFORMATION.

Miscellaneous Network Mini-Redirector Routines

4/26/2017 • 1 min to read • Edit Online

A few other routines implemented by a network mini-redirector deal with buffering state management and connection IDs. The buffering state management routines can be used by a network mini-redirector to handle various buffering schemes implemented in the redirector and the server and to communicate any changes in buffering state. For example, the **MRxCompleteBufferingStateChangeRequest** routine is used by the SMB redirector as part of the mechanism to send an oplock break to the server.

Connection IDs can be used for multiplexing multiple connections. It is unnecessary for a network mini-redirector to support connection IDs, so **MRxGetConnectionId** is optional.

The following table lists the routines that can be implemented by a network mini-redirector for miscellaneous operations.

ROUTINE	DESCRIPTION
MRxCompleteBufferingStateChangeRequest	RDBSS calls this routine to notify the network mini-redirector that a buffering state change request has been completed. For example, the SMB redirector uses this routine to send an oplock break response or to close the handle on an oplock break if the file is no longer in use. Byte range locks that need to be flushed out to the server are passed to the network mini-redirector in the **LowIoContext.ParamsFor.Locks.LockList** member of the RX_CONTEXT.
MRxComputeNewBufferingState	RDBSS calls this routine to request that the network mini-redirector compute a new buffering state change.
MRxGetConnectionId	RDBSS calls this routine to request that a network mini-redirector return a connection ID for the connection which can be used for handling multiple sessions. If connection IDs are supported by the network mini-redirector, then the returned connection ID is appended to the connection structures stored in the name table. RDBSS considers the connection ID as an opaque blob, and does a byte-by-byte comparison of the connection ID blob while looking up the net-name table for a given name.

Writing a Kernel Network Redirector

4/26/2017 • 1 min to read • Edit Online

This section discusses the important issues to consider when writing a kernel network redirector. The following topics are discussed:

Support for UNC Naming and MUP

MUP and DFS Interactions

MUP Changes in Microsoft Windows Vista

Support for Transactions in Microsoft Windows Vista

Network Redirectors and the File System Process

Support for UNC Naming and MUP

9/26/2017 • 11 min to read • Edit Online

The multiple UNC provider (MUP) is a kernel-mode component responsible for channeling all remote file system accesses using a Universal Naming Convention (UNC) name to a network redirector (the UNC provider) that is capable of handling the remote file system requests. MUP is involved when a UNC path is used by an application as illustrated by the following example that could be executed from a command line:

```
notepad \\server\public\readme.txt
```

MUP is not involved during an operation that creates a mapped drive letter (the "NET USE" command, for example). This operation is handled by the multiple provider router (MPR) and a user-mode WNet provider DLL for the network redirector. However, a user-mode WNet provider DLL can communicate directly with a kernel-mode network redirector driver during this operation.

On Microsoft Windows Server 2003, Windows XP, and Windows 2000, remote file operations performed on a mapped drive that does not represent a Distributed File System (DFS) drive don't go through MUP. These operations go directly to the network provider that handled the drive letter mapping.

For network redirectors that conform to the Windows Vista redirector model, MUP is involved even when a mapped network drive is used. File operations performed on the mapped drive go through MUP to the network redirector. Note that in this case MUP simply passes the operation to the network redirector involved.

MUP is part of the *mup.sys* binary, which also includes the Microsoft DFS client in Windows Server 2003, Windows XP, and Windows 2000.

A kernel network redirector will normally also have a user-mode WNet provider DLL to support establishing connections to remote resources (mapping drive letters to remote resources, for example). The MPR is a user-mode DLL that establishes network connections based on queries to WNet providers. Calls to the MPR would result from any of the following:

A "net use x: \\server\share" command issued from a command prompt.

A network drive letter connection established from Windows Explorer

Direct calls to WNet functions.

A network redirector must register with MUP to handle UNC names. There can be multiple UNC providers registered with MUP. These UNC providers can be one or more of the following:

- Network mini-redirectors based on RDBSS
- Legacy redirectors not based on RDBSS

MUP determines which provider can handle a UNC path in a name-based operation, typically an IRP_MJ_CREATE request. This is referred to as "prefix resolution." Prior to Windows Vista, the prefix resolution operation serves two purposes:

- The name-based operation which resulted in the prefix resolution is routed to the provider claiming the prefix. If successful, MUP ensures that subsequent handle-based operations (IRP_MJ_READ and IRP_MJ_WRITE, for example) go to the same provider completely bypassing MUP.
- The provider and the prefix it claimed are entered in a prefix cache maintained by MUP. For subsequent name-based operations, MUP uses this prefix cache to determine whether a provider has already claimed a

prefix before attempting to perform a prefix resolution. Each entry in this prefix cache is subject to a timeout (referred to as TTL) once it is added to the cache. An entry is thrown away after this timeout expires, at which point of time MUP will perform prefix resolution again for this prefix on a subsequent name-based operation.

MUP performs prefix resolution by issuing an **IOCTL_REDIR_QUERY_PATH** request to network redirectors registered with MUP. The input and output buffers for IOCTL_REDIR_QUERY_PATH are as follows:

	PARAMETER AVAILABLE AT	DATA STRUCTURE FORMAT
Input buffer	IrpSp->Parameters.DeviceIoControl.Type3InputBuffer	QUERY_PATH_REQUEST
Output buffer	IRP->UserBuffer	QUERY_PATH_RESPONSE

The IOCTL and the data structures are defined in *ntifs.h*. The buffers are allocated from non-paged pool.

Network redirectors should only allow kernel-mode senders of this IOCTL, by verifying that the **RequesterMode** member of the IRP structure is **KernelMode**.

MUP uses the QUERY_PATH_REQUEST data structure for the request information.

```
typedef struct _QUERY_PATH_REQUEST {
    ULONG               PathNameLength;
    PIO_SECURITY_CONTEXT SecurityContext;
    WCHAR               FilePathName[1];
} QUERY_PATH_REQUEST, *PQUERY_PATH_REQUEST;
```

STRUCTURE MEMBER	DESCRIPTION
PathNameLength	The length, in bytes, of the Unicode string contained in the **FilePathName** member.
SecurityContext	A pointer to the security context.
FilePathName	A non-NULL terminated Unicode string of the form &lt;server>&lt;share>&lt;path>. The length of the string, in bytes, is specified by the **PathNameLength** member.

UNC providers should use the QUERY_PATH_RESPONSE data structure for the response information.

```
typedef struct _QUERY_PATH_RESPONSE {
    ULONG  LengthAccepted;
} QUERY_PATH_RESPONSE, *PQUERY_PATH_RESPONSE;
```

STRUCTURE MEMBER	DESCRIPTION

STRUCTURE MEMBER	DESCRIPTION
LengthAccepted	The length, in bytes, of the prefix claimed by the provider from the Unicode string path specified in the **FilePathName** member of the QUERY_PATH_REQUEST structure.

Note that IOCTL_REDIR_QUERY_PATH is a METHOD_NEITHER IOCTL. This means that the input and output buffers might not be at the same address. A common mistake by UNC providers is to assume that the input buffer and the output buffer are the same and use the input buffer pointer to provide the response.

When a UNC provider receives an IOCTL_REDIR_QUERY_PATH request, it has to determine whether it can handle the UNC path specified in the **FilePathName** member of the QUERY_PATH_REQUEST structure. If so, it has to update the **LengthAccepted** member of the QUERY_PATH_RESPONSE structure with the length, in bytes, of the prefix that it has claimed, and complete the IRP with STATUS_SUCCESS. If the provider cannot handle the specified UNC path, it must fail the IOCTL_REDIR_QUERY_PATH request with an appropriate NTSTATUS error code and must not update the **LengthAccepted** member of the QUERY_PATH_RESPONSE structure. Providers must not modify any of the other members or the **FilePathName** string under any condition.

If the \\server\share prefix name is not recognized in response to an IRP_MJ_CREATE or other IRPs that are using UNC names, the recommended NTSTATUS code to return is one of the following:

STATUS_BAD_NETWORK_PATH
The network path cannot be located. The machine name (\\server, for example) is not valid or the network redirector cannot resolve the machine name (using whatever name resolution mechanisms are available).

STATUS_BAD_NETWORK_NAME
The specified share name cannot be found on the remote server. The machine name (\\server, for example) is valid, but specified share name cannot be found on the remote server.

STATUS_INSUFFICIENT_RESOURCES
There were insufficient resources available to allocate memory for buffers.

STATUS_INVALID_DEVICE_REQUEST
An IOCTL_REDIR_QUERY_PATH request should only come from MUP and the requester mode of the IRP should always be **KernelMode**. This error code is returned if the requester mode of the calling thread was not **KernelMode**.

STATUS_INVALID_PARAMETER
The **PathNameLength** member in the QUERY_PATH_REQUEST structure exceeds the maximum allowable length, UNICODE_STRING_MAX_BYTES, for a Unicode string.

STATUS_LOGON_FAILURE or STATUS_ACCESS_DENIED
If the prefix resolution operation failed due to invalid or incorrect credentials, the provider should return the exact error code returned by the remote server; these error codes must not be translated to STATUS_BAD_NETWORK_NAME or STATUS_BAD_NETWORK_PATH. Error codes like STATUS_LOGON_FAILURE and STATUS_ACCESS_DENIED serve as a feedback mechanism to the user indicating the requirement to use appropriate credentials. These error codes are also used in certain cases to prompt the user automatically for credentials. Without these error codes, the user might assume that the machine is not accessible.

If the network redirector is unable to resolve a prefix, it must return an NTSTATUS code that closely matches the intended semantics from the above list of recommended NTSTATUS codes. A network redirector must not return the actual encountered error (STATUS_CONNECTION_REFUSED, for example) directly to MUP if the NTSTATUS code is not from the above list.

The length of the prefix claimed by the provider depends on an individual UNC provider. Most providers usually claim the \\<servername>\<sharename > part of a path of the form \\<servername>\<sharename>\<path>. For example, if a provider claimed \\server\public given a path \\server\public\dir1\dir2, all name-based operations for the prefix \\server\public (\server\public\file1, for example) will be routed to that provider automatically without any prefix resolution since the prefix is already in the prefix cache. However, a path with the prefix \server\marketing\presentation will go through prefix resolution.

If a network redirector claims a server name (\\server, for example), all requests for shares on this server will go to this network redirector. This behavior is only acceptable if there is no possibility of another share on the same server being accessed by a different network redirector. For example, a network redirector claiming \\server of a UNC path will prevent access by other network redirectors to other shares on this server (WebDAV access to \\server\web, for example).

Any legacy network redirector (not based on using RDBSS) that registers as a UNC provider with MUP by calling **FsRtlRegisterUncProvider** will receive the IOCTL_REDIR_QUERY_PATH request.

A network mini-redirector that indicates support as a UNC provider will receive this prefix claim as if it were an IRP_MJ_CREATE call. This create request is similar to a user-mode **Createfile** call with FILE_CREATE_TREE_CONNECTION flag set on. A network mini-redirector will not receive the prefix claim as a call to **MRxLowIOSubmit[LOWIO_OP_IOCTL]**. For a prefix claim, RDBSS will send an **MRxCreateSrvCall** request to the network mini-redirector followed by a call to **MRxSrvCallWinnerNotify** and **MRxCreateVNetRoot**. When a network mini-redirector registers with RDBSS, the driver dispatch table for the network mini-redirector will be copied over by RDBSS to point to internal RDBSS entry points. RDBSS then receives this IOCTL_REDIR_QUERY_PATH internally for the network mini-redirector and calls **MRxCreateSrvCall**, **MRxSrvCallWinnerNotify**, and **MRxCreateVNetRoot**. The original IOCTL_REDIR_QUERY_PATH IRP will be contained in the RX_CONTEXT structure passed to the **MRxCreateSrvCall** routine. In addition, the following members in the RX_CONTEXT passed to **MRxCreateSrvCall** will be modified:

The **MajorFunction** member is set to IRP_MJ_CREATE even though the original IRP was IRP_MJ_DEVICE_CONTROL.

The **PrefixClaim.SuppliedPathName.Buffer** member is set to the **FilePathName** member of the QUERY_PATH_REQUEST structure.

The **PrefixClaim.SuppliedPathName.Length** member is set to the **PathNameLength** member of the QUERY_PATH_REQUEST structure.

The **Create.NtCreateParameters.SecurityContext** member is set to the **SecurityContext** member of the QUERY_PATH_REQUEST structure.

The **Create.ThisIsATreeConnectOpen** member is set to **TRUE**.

The **Create.Flags** member has the RX_CONTEXT_CREATE_FLAG_UNC_NAME bit set.

If the network mini-redirector wants to see details of the prefix claim, it can read these members in the RX_CONTEXT passed to **MRxCreateSrvCall**. Otherwise it can just attempt to connect to the server share and return STATUS_SUCCESS if the **MRxCreateSrvCall** call was successful. RDBSS will make the prefix claim on behalf of the network mini-redirector.

There is one case where a network mini-redirector could receive this IOCTL directly. A network mini-redirector could save a copy of its driver dispatch table before initializing and registering with RDBSS. After calling **RxRegisterMinirdr** to register with RDBSS, the network mini-redirector could save a copy of the new driver dispatch table entry points installed by RDBSS and restore its original driver dispatch table. The restored driver dispatch table would need to be modified so that after checking the received IRP for those of interest to the network mini-redirector, the call is forwarded to the RDBSS driver dispatch entry points. RDBSS will copy over the driver dispatch table of a network mini-redirector when the driver initializes RDBSS and calls **RxRegisterMinrdr**. A network mini-redirector that links against *rdbsslib.lib* must save its original driver dispatch table before calling **RxDriverEntry** from its **DriverEntry** routine to initialize the RDBSS static library and restore its driver dispatch table after calling **RxRegisterMinrdr**. This is because RDBSS copies over the network mini-redirector dispatch table in both the **RxDriverEntry** and **RxRegisterMinrdr** routines.

The order in which providers are queried during prefix resolution is controlled by the REG_SZ ProviderOrder registry value stored under the following key:

```
HKLM\System\CurrentControlSet\Control\NetworkProvider\Order
```

Individual provider names in the ProviderOrder registry value are separated by commas without any leading or trailing white space.

For example, this value could contain the string:

```
RDPNP,LanmanWorkstation,WebClient
```

Given a UNC path \\<server>\<share>\<path>, MUP issues a prefix resolution request if the prefix (\\server\share or \\server, for example) is not found in the MUP prefix cache. MUP sends a prefix resolution request to each provider in the following order until a provider claims the prefix (or all providers have been queried):

1. TS client (RDPNP)
2. SMB redirector (LanmanWorkstation)
3. WebDAV redirector (WebClient)

Changes to the ProviderOrder registry value require a reboot to take effect in MUP on Windows Server 2003, Windows XP, and Windows 2000.

MUP uses each provider name listed to find the provider's registry key under the following registry key:

```
HKLM\System\CurrentControlSet\Services\<ProviderName>
```

MUP then reads the DeviceName value under the NetworkProvider subkey to find the device name with which the provider will register. When the provider actually registers, MUP matches the device name passed in with the list of device names of known providers and places the provider in an ordered list for the purposes of prefix resolution. The order of providers in this list is based on the order specified in the ProviderOrder registry value discussed above.

Note that this provider order is also honored by the Multiple Provider Router (MPR), the user-mode DLL that establishes network connections based on queries to WNet providers.

Prior to Windows Server 2003 with Service Pack 1 and Windows XP with Service Pack 2, MUP behavior was to issue prefix resolution requests to all the providers "in parallel" in the order specified in the ProviderOrder registry value, and then wait for all of the providers to complete the prefix resolution operation. Thus, even if the first provider claimed the prefix, MUP still waits for all the other providers to complete the prefix resolution operation. When multiple providers respond with a prefix claim, MUP selects the provider based on the order specified in ProviderOrder registry value.

On Windows XP Service Pack 2 and later and on Windows Server 2003 Service Pack 1 and later, this behavior changed slightly. MUP issues the prefix resolution request serially and stops as soon as the first provider claims the prefix. Thus, in the above example, if RDPNP claims a prefix, MUP will not call the SMB or WebDAV redirectors.

The primary reason this behavior was changed is that a "serial prefix resolution" scheme prevents cases of a network redirector with lower priority in the ProviderOrder value from causing performance issues for a network redirector of higher priority in the ProviderOrder value. For example, consider a remote server, with a firewall in place, configured to block certain types of TCP/IP packets (access to HTTP, for example), but to allow others (SMB access, for example). In this case, even if the SMB network redirector is configured as the first provider in the ProviderOrder value and claims the prefix quickly, the WebDAV redirector might significantly delay the completion of the prefix resolution by waiting for the TCP connection to timeout.

MUP and DFS Interactions

4/26/2017 • 1 min to read • Edit Online

When a Universal Naming Convention (UNC) path is used by an application, a request is sent to the multiple UNC provider (MUP) to determine which network provider to use. MUP channels the request to the appropriate network redirector (the UNC provider) that is capable of handling the remote file system request. If the Distributed File System (DFS) client is enabled, MUP first passes the request for a particular "\\server\share" to the DFS client to determine if the request is for a DFS share rather than a normal remote file share.

The default behavior is that the DFS client is enabled. The DFS client is disabled depending on the value of a registry entry located below the following:

```
HKLM\System\CurrentControlSet\Services\Mup
```

When the DisableDfs registry entry has a DWORD value of 1, the DFS client is disabled.

Assuming the prefix of a path specified in a name-based operation is not in the prefix cache maintained by MUP, the DFS client (when enabled) implicitly takes precedence over all redirectors. The DFS client attempts to determine whether a UNC path specified is a DFS path. It does this by sending a referral request to the IPC$ share of an appropriate server. If the path is determined to be a DFS path, the DFS client handles the operation. Otherwise, the DFS client passes the name-based request to MUP for prefix resolution to be handled by an appropriate redirector.

When a request to access the IPC$ share is sent to a system where the LAN Manager Server (sometimes called SMB Server), srv.sys, is disabled or not installed (a UNIX system, for example), there may be a delay introduced because several attempts are made to connect to the IPC$ share. This delay is typically 5-7 seconds, but can be longer based on the speed and latency of the connecting network infrastructure and other conditions.

MUP Changes in Microsoft Windows Vista

4/26/2017 • 10 min to read • Edit Online

Windows Vista implements a number of changes to the multiple UNC provider (MUP) that can affect network redirectors.

MUP and the Distributed File System (DFS) client are in separate binary files. The MUP component is in mup.sys and the DFS client is in dfsc.sys. On Windows Server 2003, Windows XP, and Windows 2000, the MUP kernel component, mup.sys, also contained the DFS client.

A new redirector model is defined on Windows Vista:

- MUP registers as a file system with the I/O manager by calling **IoRegisterFileSystem**.
- A network redirector registers with MUP using **FsRtlRegisterUncProviderEx** , a new routine introduced in Windows Vista.
- A network redirector passes an unnamed device object to **FsRtlRegisterUncProviderEx**.
- A network redirector passes a device name to **FsRtlRegisterUncProviderEx**.
- A network redirector does not register as a file system with the I/O manager (does not call **IoRegisterFileSystem**).
- All calls from MUP to a network redirector, including prefix resolution, IOCTLs, and FSCTLs, are made with APCs enabled. All calls from other components to MUP are expected to be made with APCs enabled. When calls are used with **FsRtlCancellableWaitForSingleObject** or **FsRtlCancellableWaitForMultipleObjects**, new routines introduced in Windows Vista, this will ensure that long waits can be aborted if a thread that issued an I/O request is terminated.
- Prefix resolution is performed using IOCTL_REDIR_QUERY_PATH_EX, a new IOCTL introduced in Windows Vista.
- A network redirector device name registered with MUP becomes a symbolic link to the MUP device object.

For a network redirector conforming to the Windows Vista redirector model, MUP creates a symbolic link in the object manager namespace with the device name specified by the network redirector in the call to **FsRtlRegisterUncProviderEx**. The target of this symbolic link is the MUP device object (\Device\Mup).

The advantage of registering MUP as a file system and the device name of the network redirector being a symbolic link to the MUP device object is that all remote file system I/O operations, and not just name-based operations, go through MUP. So file system filter drivers that need to be on the remote file system stack can simply attach to the MUP device object. It is not necessary for file system filter drivers to hardcode provider device object names (\Device\LanmanRedirector, for example) into their driver anymore. In this way, file system filter drivers can monitor all I/O operations issued to all network redirectors by a single attachment. This also eliminates duplicate I/O operations seen by file system filter drivers prior to Windows Vista, which attached separately to DFS (mup.sys) and individual network redirectors (\Device\LanmanRedirector, for example) in order to monitor I/O operations to both.

A file system filter driver that is attached to the MUP device object can selectively filter the traffic that is sent to specific network redirectors. In this situation, the filter driver maps the device names of the network redirectors of interest to provider identifiers by calling the **FsRtlMupGetProviderIdFromName** routine. The filter driver can then determine whether it should filter the traffic for a particular file object by comparing the provider identifier that is obtained by calling the **FsRtlMupGetProviderInfoFromFileObject** routine with the provider identifiers of the network directors of interest.

For a network redirector conforming to the Windows Vista redirector model:

- All file objects on the remote file system stack resolve to MUP. Hence, **IoGetDeviceAttachmentBaseRef** returns the device object for MUP, not the network redirector that owns the file object. However, the content of the file object is still owned by the network redirector.
- An IRP_MJ_CREATE issued to the device name of a network redirector (\Device\LanmanRedirector\server\share, for example) will be targeted to that network redirector without going through MUP prefix resolution, exactly as it was on Windows Server 2003, Windows XP, and Windows 2000.

Network redirectors that are not based on the Windows Vista RDBSS (linking dynamically or statically) are termed "legacy redirectors". These legacy network redirectors include:

- Network redirectors written for Windows Server 2003, Windows XP, and Windows 2000 that register directly with MUP using **FsRtlRegisterUncProvider**.
- Network mini-redirectors written for Windows Server 2003, Windows XP, and Windows 2000 that statically link with the rdbsslib.lib library for Windows Server 2003, Windows XP, or Windows 2000.
- Network redirectors written for Windows Vista that register directly with MUP using **FsRtlRegisterUncProviderEx**.

Network mini-redirectors that dynamically link against the Windows Vista RDBSS (rdbss.sys) automatically conform to the Windows Vista redirector model because RDBSS registers with MUP using *FsRtlRegisterUncProviderEx*. Network mini-redirectors that statically link against the Windows Vista RDBSS (rdbsslib.lib) also automatically conform to the Windows Vista redirector model because RDBSS registers with MUP using **FsRtlRegisterUncProviderEx**.

A legacy network redirector written for Windows Vista that registers directly with MUP must comply with the Windows Vista redirector model.

Network redirectors written for Windows Server 2003, Windows XP, and Windows 2000 that register with MUP directly using the **FsRtlRegisterUncProvider** continue to work exactly the same way as they did on Windows Server 2003, Windows XP, and Windows 2000. Network mini-redirectors written for Windows Server 2003, Windows XP, and Windows 2000 that statically link with the rdbsslib.lib library for Windows Server 2003, Windows XP, and Windows 2000 continue to work exactly the same way as they did on Windows Server 2003, Windows XP, and Windows 2000. These legacy network redirectors and mini-redirectors exhibit the following behavior:

- They will be visible to file system filter drivers that monitor file system registration.
- Their device objects are named. The device names are not symbolic links and do not point to \Device\MUP.
- File objects resolve to the named device object of the network redirector.
- MUP is involved only in the prefix resolution operation. Once the network provider has been identified, MUP "gets out of the way" by returning STATUS_REPARSE. All subsequent operations will not pass through MUP.

This behavior has been retained to prevent double filtering that would otherwise happen if the provider device name were a symbolic link to \Device\MUP. This double filtering would occur for the following reasons:

- The file system filter driver is already attached to \Device\MUP.
- The file system filter driver attaches to any registering file system. Since network redirectors that use named device objects register themselves as file systems, a file system filter driver would end up filtering the same I/O twice.

Calls to and from MUP on Windows Vista are made with APCs enabled, which has the following impacts:

- It is important to protect, if required, code paths that get called from MUP against thread suspension by appropriate means, especially IOCTL_REDIR_QUERY_PATH handlers. Note that a thread suspend is a potentially "unbounded wait" operation that can last a long time.
- It is important to ensure that any "wait for I/O" operation involving user-mode threads (as opposed to system threads) always uses "cancellable waits". See the **FsRtlCancellableWaitForSingleObject** and **FsRtlCancellableWaitForMultipleObjects** routines for details.
- Deadlocks might occur when a thread gets suspended holding some important lock. It is important to run tests in the presence of user-mode threads getting suspended arbitrarily to check for deadlock conditions.
- It is important to run tests to verify whether "wait for I/O operations" are really cancellable and that a user-mode application can terminate a thread quickly enough so that the application does not appear to be in a "non-responding" state when attempting to terminate said thread.

The prefix cache size and timeout used by MUP on Windows Vista are now controlled by the following registry values:

- PrefixCacheSizeInKB
- PrefixCacheTimeoutInSeconds.

These registry values can be changed dynamically without a reboot. These registry values are under the following registry key:

```
HKLM\System\CurrentControlSet\Services\Mup\Parameters.
```

The ProviderOrder registry value that determines the order in which MUP issues prefix resolution requests to individual redirectors can be changed dynamically without rebooting the system. This registry value is located under the following registry key:

```
HKLM\CurrentControlSet\Control\NetworkProvider\Order
```

On Windows Vista, MUP performs prefix resolution differently depending on whether the network redirector registered with MUP by calling **FsRtlRegisterUncProvider** or **FsRtlRegisterUncProviderEx**. Legacy network redirectors that register with MUP by calling **FsRtlRegisterUncProvider** will receive an **IOCTL_REDIR_QUERY_PATH** request for prefix resolution. This is the same method that is used on Windows Server 2003, Windows XP, and Windows 2000.

Network redirectors that conform to the Windows Vista redirector model and register with MUP by calling **FsRtlRegisterUncProviderEx** will receive an **IOCTL_REDIR_QUERY_PATH_EX** request for prefix resolution. Note that on Windows Vista, network mini-redirectors statically linked with rdbsslib.lib or dynamically linked with rdbss.sys will call **FsRtlRegisterUncProviderEx** indirectly through RDBSS.

The input and output buffers for IOCTL_REDIR_QUERY_PATH_EX are as follows:

	Parameter available at	Data structure format
Input buffer	IrpSp->Parameters.DeviceIoControl.Type3InputBuffer	QUERY_PATH_REQUEST_EX
Output buffer	IRP->UserBuffer	QUERY_PATH_RESPONSE

The IOCTL and the data structures are defined in ntifs.h. The buffers are allocated from non-paged pool.

Network redirectors should only honor kernel-mode senders of this IOCTL, by verifying that **Irp->RequestorMode** is **KernelMode**.

MUP uses the QUERY_PATH_REQUEST_EX data structure for the request information.

```
typedef struct _QUERY_PATH_REQUEST_EX {
  PIO_SECURITY_CONTEXT  pSecurityContext;
 ULONG  EaLength;
 PVOID  pEaBuffer;
  UNICODE_STRING  PathName;
} QUERY_PATH_REQUEST_EX, *PQUERY_PATH_REQUEST_EX;
```

STRUCTURE MEMBER	DESCRIPTION
pSecurityContext	A pointer to the security context.
EaLength	The length, in bytes, of the extended attributes buffer.
pEaBuffer	Pointer to the extended attributes buffer.
PathName	A non-NULL terminated Unicode string of the form &lt;server>&lt;share>&lt;path>.

UNC providers should use the QUERY_PATH_RESPONSE data structure for the response information.

```
typedef struct _QUERY_PATH_RESPONSE {
 ULONG  LengthAccepted;
} QUERY_PATH_RESPONSE, *PQUERY_PATH_RESPONSE;
```

STRUCTURE MEMBER	DESCRIPTION
LengthAccepted	The length, in bytes, of the prefix claimed by the provider from the Unicode string path specified in the **PathName** member of the QUERY_PATH_REQUEST_EX structure.

Note that IOCTL_REDIR_QUERY_PATH_EX is a METHOD_NEITHER IOCTL. This means that the input and output buffers might not be at the same address. A common mistake by UNC providers is to assume that the input buffer and the output buffer are the same and use the input buffer pointer to provide the response.

When a UNC provider receives an IOCTL_REDIR_QUERY_PATH_EX request, it has to determine whether it can handle the UNC path specified in the **PathName** member of the QUERY_PATH_REQUEST_EX structure. If so, the UNC provider has to update the **LengthAccepted** member of the QUERY_PATH_RESPONSE structure with the length, in bytes, of the prefix it has claimed and complete the IRP with STATUS_SUCCESS. If the provider cannot handle the UNC path specified, it must fail the IOCTL_REDIR_QUERY_PATH_EX request with an appropriate NTSTATUS error code and must not update the **LengthAccepted** member of the QUERY_PATH_RESPONSE structure. Providers must not modify any of the other members or the **PathName** string under any condition.

On Windows Vista, a network mini-redirector based on using RDBSS that indicates support as a UNC provider will receive this prefix claim as if it were a regular tree connect create, similar to a user-mode Createfile call with FILE_CREATE_TREE_CONNECTION flag set. RDBSS will send an **MRxCreateSrvCall** request to the network mini-redirector followed by a call to **MRxSrvCallWinnerNotify** and **MRxCreateVNetRoot**. This prefix claim will not be received as a call to **MRxLowIOSubmit[LOWIO_OP_IOCTL]**. When a network mini-redirector registers with RDBSS, the driver dispatch table for the network mini-redirector will be copied over by RDBSS to point to internal RDBSS entry points. RDBSS then receives this IOCTL_REDIR_QUERY_PATH_EX internally for the network mini-redirector and calls **MRxCreateSrvCall**, **MRxSrvCallWinnerNotify**, and **MRxCreateVNetRoot**. The original IOCTL_REDIR_QUERY_PATH_EX IRP will be contained in the RX_CONTEXT passed to the **MRxCreateSrvCall** routine. In addition, the following members in the RX_CONTEXT passed to **MRxCreateSrvCall** will be modified:

The **MajorFunction** member is set to IRP_MJ_CREATE even though the original IRP was IRP_MJ_DEVICE_CONTROL.

The **PrefixClaim.SuppliedPathName.Buffer** member is set to the **PathName.Buffer** member of the QUERY_PATH_REQUEST_EX structure.

The **PrefixClaim.SuppliedPathName.Length** member is set to the **PathName.Length** member of the QUERY_PATH_REQUEST_EX structure.

The **Create.ThisIsATreeConnectOpen** member is set to the **TRUE**.

The **Create.ThisIsAPrefixClaim** member is set to the **TRUE**.

The **Create.NtCreateParameters.SecurityContext** member is set to the **SecurityContext** member of the QUERY_PATH_REQUEST_EX structure.

The **Create.EaBuffer** member is set to the **pEaBuffer** member of the QUERY_PATH_REQUEST_EX structure.

The **Create.EaLength** member is set to the **EaLength** member of the QUERY_PATH_REQUEST_EX structure.

The **Create.Flags** member will have the RX_CONTEXT_CREATE_FLAG_UNC_NAME bit set.

If the network mini-redirector wants to see details of the prefix claim, it can read these members in the RX_CONTEXT structure that is passed to **MRxCreateSrvCall**. Otherwise, it can just attempt to connect to the server share and return STATUS_SUCCESS if the **MRxCreateSrvCall** call was successful. RDBSS will make the prefix claim on behalf of the network mini-redirector.

Support for Transactions in Microsoft Windows Vista

4/26/2017 • 1 min to read • Edit Online

On Windows Vista, there is currently no support for transactions for network mini-redirectors based on RDBSS.

Network Redirectors and the File System Process

4/26/2017 • 4 min to read • Edit Online

Most of the file operations performed by a file system driver are usually completed in the user's thread context. These operations include all of the synchronous file I/O calls to the file system. In these cases, all of the work is done inline. The operation might block in the kernel, but the work is performed on the same thread. File systems on Windows often call this work using the user's thread context as the File System Dispatch (FSD). Most of the time, a file system will try to complete its work in the FSD.

There are some cases where an application does not want to block (asynchronous reads or asynchronous writes, for example). In these cases, the file I/O operation needs to be dispatched to a system worker thread for completion. File systems on Windows often call this work using a system worker thread context as the File System Process (FSP). The following example of an asynchronous IRP_MJ_WRITE request sent to a network mini-redirector illustrates a case where the FSP needs to be used.

To execute an asynchronous IRP_MJ_WRITE request, RDBSS or a network mini-redirector needs to acquire the file object's FCB resource (a synchronization object used for changing the file object). When the FCB resource is already held by another thread and RDBSS or the network mini-redirector try to acquire the FCB resource in the user's thread context (the FSD), this operation will block. Asynchronous requests to a file system are not supposed to block. For asynchronous requests (IRP_MJ_WRITE request, for example), a file system driver will first check if the needed resource is available (the FCB resource for a network mini-redirector, for example). If the resource is not available, then the file system driver posts the request to a work queue for later completion by a system worker thread (the FSP) and the file system returns STATUS_PENDING from the FSD thread. To the user application, the FSD will return STATUS_PENDING immediately, while the actual work will be handled by a system worker thread in the FSP.

Several tasks must be completed before a file system driver posts work to the FSP. The file system driver needs to capture the security context from the user's thread in the FSD since the work will be completed by a system worker thread. RDBSS does this automatically in the **RxFsdPostRequest** routine for network mini-redirectors. This routine is called by RDBSS whenever a network mini-redirector returns STATUS_PENDING with the **PostRequest** member of the RX_CONTEXT structure set to **TRUE**. If work is posted to a work queue, the file system driver must also make sure that the user buffers will be available for later use by the system worker thread. There are two methods to accomplish this task:

1. A file system driver can map the user buffers into kernel memory space before posting to the FSP so the buffers can be accessed later by a system worker thread (FSP). The method commonly used by file system drivers is to lock the user buffers in the FSP because the memory pages can always be mapped later by the system worker thread.

2. A file system can save the thread of the calling process from the FSP and the system worker thread can attach to this calling process while in the FSP. Using the **KeStackAttachProcess**, the system worker thread would attach to the user's calling process and access the user buffers and then detach from the user's calling process using **KeUnstackDetachProcess** when the work is done.

RDBSS will automatically lock user buffers using method 1 in the **RxFsdPostRequest** routine for a number of IRP requests as long as the RX_CONTEXT_FLAG_NO_PREPOSTING_NEEDED bit is not set in the **Flags** member of the RX_CONTEXT structure. User buffers will be locked for the following requests:

- IRP_MJ_DIRECTORY_CONTROL with a minor function of IRP_MN_QUERY_DIRECTORY

- IRP_MJ_QUERY_EA

- IRP_MJ_READ as long as the minor function does not include IRP_MN_MDL
- IRP_MJ_SET_EA
- IRP_MJ_WRITE as long as the minor function does not include IRP_MN_MDL

Method 2 is commonly used for processing IRPs that use METHOD_NEITHER when only a small amount of information is normally passed and returned. These IRPs would include the following:

- IRP_MJ_DEVICE_CONTROL
- IRP_MJ_FILE_SYSTEM_CONTROL
- IRP_MJ_INTERNAL_DEVICE_CONTROL

RDBSS only supports asynchronous calls for the **MrxLowIoSubmit** array of operations. If a network mini-redirector wants to implement other operations (**MRxQueryFileInfo**, for example) as an asynchronous call, the network mini-redirector needs to post the request to the FSP. If a network mini-redirector receives a request for **MrxQueryFileInfo** with the RX_CONTEXT_ASYNC_OPERATION bit set in the **Flags** member of the RX_CONTEXT structure, the network mini-redirector would need to post this request to the FSP for asynchronous operation. In the operation of the **MrxQueryFileInfo** routine, the network mini-redirector would first need to capture the security context of the user's thread and map the user buffers into kernel space (or set the system worker thread to attach to the user's calling process while executing in the FSP). Then the network mini-redirector would set the **PostRequest** member of the RX_CONTEXT structure to **TRUE** and return STATUS_PENDING from the FSD. The work would be dispatched by RDBSS to a work queue for operation by a system worker thread (the FSP).

Security Considerations for File Systems

4/26/2017 • 1 min to read • Edit Online

This section provides security guidance for file system developers. Follow these guidelines when you use the Windows Driver Kit (WDK) to implement file systems and file system filter drivers for Microsoft Windows operating systems. The following topics are discussed:

Introduction to File Systems Security

General Driver Security Issues

Security Features for File Systems

File System Security Issues

Security Considerations for File System Filter Drivers

Introduction to File Systems Security

4/26/2017 • 1 min to read • Edit Online

File system developers who use the WDK should be aware of security considerations during the implementation of their file systems. File system filter driver developers should also be aware of these issues, and should be familiar with the Microsoft Windows security interfaces in order to incorporate security considerations into their products.

This section focuses on Windows security and file systems. It is not intended to be a primer for those who want to develop security extensions for Windows, or for file system developers who want to implement a non-Windows security model within the Windows environment.

This section includes a number of code samples. The samples are extracted from real-world code, but are presented here in abbreviated form. Developers should adapt these samples for use in their particular environment.

General Driver Security Issues

4/26/2017 • 1 min to read • Edit Online

This section discusses security issues that apply to all drivers, including file systems and file system filter drivers.

This section contains:

What is Driver Security

4/26/2017 • 1 min to read • Edit Online

With respect to drivers, anything a user can do that causes a driver to malfunction in such a way that it causes the system to crash or become unusable is a security flaw. When most developers are working on their driver, their focus is on getting the driver to work properly and not whether a malicious intruder will attempt to exploit holes within the system. This is even more the case for file systems and file system filter drivers, which are some of the most complicated types of drivers to write.

However, after a product release, there are users who attempt to probe and identify security weaknesses. Thus, it makes sense for developers to consider these issues during the design and implementation phase in order to minimize the likelihood that such holes exist. The goal is to eliminate as many security holes as possible before they become part of a released product.

Achieving secure drivers requires the cooperation of the designer (consciously thinking of potential threats to the driver), the implementer (defensively coding common operations that can be the source of exploits), and the test team (proactively attempting to find exploits). By properly coordinating all of these activities, the security of the driver will be dramatically enhanced.

Designing with Security Threat Models

4/26/2017 • 1 min to read • Edit Online

In considering security, a common methodology is to create specific threat models that attempt to describe the types of attacks that are possible. This technique is useful when designing a file system or file system filter driver because it forces the developer to consider the potential attack vectors against a driver. Having identified potential threats, a driver developer can then consider means of defending against these threats in order to bolster the overall security of the driver component.

When considering security threat models, it is also important to differentiate between the actions drivers manage on behalf of user I/O requests (which are subject to security checks) and I/O operations initiated by drivers themselves (which are by default not subject to security checks). User-mode requests to one driver may also be passed on to another driver through internal FSCTL or IOCTL requests (the Srv.sys driver, for example), further complicating these issues.

For developers of kernel-mode drivers, the following important issues should be considered:

- I/O operations initiated by drivers bypass security checks on the local system
- I/O operations initiated by drivers bypass most parameter validation checks.
- Drivers are part of the trusted computing base and thus can control the entire system.

Imagine, for example, a rogue program that was able to successfully load a driver on the system. This would provide it with tremendous control and would essentially compromise the system. If an application can exploit a feature of your driver to achieve the same thing, you have handed over control to the application and compromised the system.

Microsoft uses the "STRIDE" model when considering security:

- S--Spoofing Identity.
- T--Tampering with Data.
- R--Repudiation.
- I--Information Disclosure.
- D--Denial of Service.
- E--Elevation of Privilege.

The basic principles here identify specific types of system compromise that can occur. Some of these principles have validity to drivers in general, but all have validity to file system and file system filter drivers.

Spoofing Identity

4/26/2017 • 1 min to read • Edit Online

The concept of spoofing identity is allowing unprivileged code to use someone else's identity, and hence, their security credentials. For example, a driver that uses some form of a password mechanism is subject to this type of attack. Not all such drivers have security flaws, although, they are vulnerable to security flaws based on spoofing identity. The designers and implementers of the driver need to evaluate the level of vulnerability.

Other more subtle examples of spoofing identity exist. For example, an encryption filter driver that relies upon a smart card for the decryption key is subject to a physical spoofing attack if the smart card is lost or stolen. So, a filter driver might add some additional requirement, such as a biometric confirmation or a password, to protect against this category of attack.

A driver that attempts to perform its own security checks must be particularly vigilant to ensure that it uses the proper credentials during the security check. A failure to do so can easily provide a spoofing exploit that would allow a malicious user who discovered it to perform operations that would appear to have been done by someone else, since the security descriptor would be correct.

In general, drivers are best designed and implemented if they take advantage of the existing security mechanisms within the operating system, rather than constructing their own. This minimizes the number of potential locations where the implementation may contain errors.

Tampering with Data

4/26/2017 • 2 min to read • Edit Online

Data tampering is a threat for drivers, but a serious threat for file system and file system filter drivers. For all drivers, there is a potential threat that any control structure shared between user-mode and kernel-mode components may be modified by the user-mode component while it is in use by the kernel-mode component. File system and filter drivers are particularly vulnerable to this class of attack because they have a strong reliance on METHOD_NEITHER data transfer types that directly access the user buffer through its virtual address. Fast I/O also directly accesses raw user-mode buffers. The risk here is that a driver might be using this data while the application is modifying the data. Typically, a driver attempts to protect against this by validating the data. However, data validation only works if the data cannot change after it has been validated.

For file system and file system filter drivers, there are numerous IOCTL and FSCTL operations that are used to transfer information between user-mode applications and the various kernel-mode drivers. Further, it is quite common for such drivers to have private IOCTL and FSCTL operations that also transfer control and data information between user-mode services and their kernel-mode drivers.

Inherent in this design model is an assumption that only the driver's service or application will take advantage of its interface. Such a design and implementation is at risk for the following reasons:

- A malicious application could send a valid buffer to the driver, and then subsequently modify the data in an attempt to probe and find the weaknesses of the driver.
- Failures within the control application could cause the data contents of a control buffer to become invalid. For example, a stack-based control region might become invalid if an exception were to change the flow of control and cause a reuse of the stack region.

Protecting against this category of problem requires constant vigilance in terms of the implementation. Buffers must be located in non-volatile memory (either kernel-only or read-only memory) prior to being validated. If the buffer contains references to any other buffers or control structures, they too must be located in non-volatile memory before they are validated.

Developers should also be aware that IOCTL using the **FastIoDeviceControl** dispatch will pass data in raw user buffers. So drivers implementing the fast I/O version for IOCTLs should take appropriate steps to prevent problems.

Note that validating the data by itself is not enough. For example, a successful call to **ProbeForWrite** might indicate that a buffer is valid, but subsequent changes in the application address space could cause that state to change. The application could terminate, for example, prior to the driver actually using the buffer directly. Thus, the driver must protect against any change within the application's address space. Normally this is done using structured exception handling using **__try** and **__except** around any code that accesses a user buffer address directly.

Repudiation

4/26/2017 • 1 min to read • Edit Online

The concept of repudiation is that a user might perform a particular operation, and then subsequently deny having performed it. For most drivers this is an unusual type of issue. For a file system, however, logging is used to track operations (deletion of important files, for example) and ensure that there is a clear trail of operations. This provides a mechanism for ensuring against such repudiation.

Additionally, the operating system can assign ownership of objects to specific security identifiers. The ownership information cannot be changed without appropriate privileges (**SeTakeOwnershipPrivilege**) in order to ensure that ownership of specific objects can be tracked. Object ownership provides another form of protection against repudiation.

Information Disclosure

4/26/2017 • 2 min to read • Edit Online

For a driver, information disclosure typically relates to exposing information to an application through poor buffer handling. For example, a driver with a buffered I/O request normally indicates the amount of data being returned by setting the **Information** member of the *IOStatus* block structure. The I/O manager then uses that information to copy the results back into the application's buffer.

If the driver indicates more data is being returned, the I/O manager will copy additional data from the *SystemBuffer*. However, if the driver did not fill in the balance of the *SystemBuffer*, then whatever was in that memory will be returned back to the application, potentially exposing sensitive data to the application. Witness the problem with network drivers where extra information might be sent to other systems because the data buffers used were not cleared. For example, an ICMP ping response might include extra information. This problem of exposing data inadvertently is very real and happens in a wide variety of systems.

For a file system or file system filter driver, there is the added risk of disclosing file information to users who should not be allowed to access the data. This can be done in many different ways:

- A filter driver that uses **ZwCreateFile** to open the file, and then provides access to the data through its intermediate handle. The **ZwCreateFile** function will, by default, open the file and bypassing security checks because the request is coming from kernel mode. So, access using this handle may disclose information that would not normally be available to an application.

  If the filter driver wants to enforce access checks to ensure it is not exposing data that should not be exposed, then the filter driver should specify OBJ_FORCE_ACCESS_CHECK in the *ObjectAttributes* parameter of the **ZwCreateFile** function.

- A filter driver that opens a handle in kernel mode (bypassing access checks) but does not specify OBJ_KERNEL_HANDLE. So the handle created is placed in the current process's handle table. This handle, with full access to the data, is then visible in user mode. A rogue application can be watching for such handles and try to use them to access the data.

- A file system or file system filter driver that posts an IRP_MJ_CREATE request, and then processes it in the context of a system worker thread. When the IRP is processed by the system worker thread, the create operation will normally be completed using the privileges of the system thread. So, access using these system privileges can disclose information that would not normally be available to the application.

- A file system or file system filter driver that performs file object-based I/O without confirming that the calling process should be allowed access to the given data.

Because of their unique role in managing and protecting information, file systems and file system filter drivers must be particularly vigilant in ensuring their protection of information.

Denial of Service

4/26/2017 • 3 min to read • Edit Online

A denial-of-service occurs when access to a particular service should have been granted, but in fact was improperly rejected. For example, any operation that an unprivileged application can perform that causes the system to become unusable is effectively a denial-of-service. This would include any operation or sequence of operations that:

- Crashes the system.
- Causes premature termination of threads or processes.
- Creates a deadlock condition. A deadlock occurs when two or more threads are stopped waiting in a permanent state of stalemate. Each thread is waiting for a resource held by one of the other threads.
- Creates a live lock condition. A live lock can occur when two or more processors cannot progress because they are waiting to acquire a resource (typically a lock on a queue) and the thread which owns that resource is in a similar non-progressing state.

Such problems often arise within drivers because they contain latent bugs that can be exploited by normal applications. Exploits of this type can be simple and are difficult to protect against. Common causes of such problems in drivers include:

- Improper user buffer validation.
- Buffer overflow or underflow.

For file systems and file system filter drivers, there are numerous cases of such problems. For example, the MAX_PATH value is defined as 260 for historical reasons on the Win32 subsystem. Many driver components assume that this indicates the size of the largest path. Unfortunately, this is not the case as the maximum path on an NTFS file system is 32,767 Unicode characters (65,534 bytes). If a filter driver were to encode a MAX_PATH length assumption into its code base, a simple denial-of-service attack would arise from an application creating a path larger than this within a path managed by the filter driver.

Another common problem is that applications often embed user-mode pointers into private FSCTL requests. A file system is subject to three broad categories of denial-of-service attacks:

- Consuming all available disk space.
- Using all available disk bandwidth.
- Blocking access to files to which users should have access.

Typically, there is little that a file system developer can do to prevent these types of attacks. However, there are steps that can be taken by developers to allow administrators to limit these types of denial-of-service attacks.

The simplest denial-of-service attack involving the file system is to use up all of the free disk space. It is simple to write an application to do this and the consequences are far reaching. Many applications and services in the system will not function if they can no longer write to disk. The mitigating technology is disk quotas, which can limit how much disk space is available for files owned by a user when used properly by administrators. Therefore, it makes sense to include support for disk quotas when developing a file system.

A malicious or poorly written application could also try to consume all disk bandwidth. The consequences for regular users subject to this type of attack is a sluggish or unresponsive system. Currently, the operating system has no mechanism to throttle the bandwidth consumed by applications. A file system also consumes kernel memory for each open file object and file handle. A malicious application could try to continuously open a large number of files and keep them open until memory is exhausted. The primary mitigating technique for these issues is auditing and logging so that an administrator can monitor the computer for applications that perform a large amount of I/O or use large amounts of other resources. Again, it would be judicious for file system and file system filter drivers to include auditing support to be better able to minimize this type of denial of service.

A malicious application can try to prevent other users from accessing files needed for normal use. An important strategy to minimize these issues is to ensure that security information associated with file objects is properly implemented when developing file systems.

Finally, all drivers need to be concerned about consuming all available memory or other resources in response to requests from a malicious or aberrant application.

Elevation of Privilege

4/26/2017 • 1 min to read • Edit Online

An elevation-of-privilege occurs when an application gains rights or privileges that should not be available to them. Many of the elevation-of-privilege exploits are similar to exploits for other threats. For example, buffer overrun attacks that cleverly attempt to write executable code. This works on the x86-based architecture when a buffer is allocated from the stack as a local variable. The stack also contains the return address of the current procedure call. If a malicious developer ascertains that there is a buffer overflow potential, data can be placed in the buffer so that it overwrites the return address. When the CPU executes the "ret" instruction to return back to the previous caller, it will return control to the location specified by the malicious developer and not the real caller.

For file systems and file system filter drivers, the possibility of an elevation--of-privilege attack is quite high due to a combination of the following reasons:

- File systems and file system filter drivers are actively involved in managing access to data, including privileges.
- File systems and file system filter drivers exploit special privileges and access rights to implement their features.
- Many of the operating system privileges directly relate to file systems (**SeChangeNotifyPrivilege**, which controls the ability to traverse directories, for example).

This type of exploit is most important for those implementing file systems. This exploit can be an issue to file system filter drivers that are actively managing data storage (encryption filters, for example) that might circumvent or bypass normal file system security operations.

Implementing File Systems to Minimize Security Threats

4/26/2017 • 1 min to read • Edit Online

Implementation problems that pose security threats fall into a set of common issues:

- Buffer handling.
- Authentication and identification.
- Access control.
- Handle management.

None of these issues is particularly novel. These issues are well known, yet these problems recur in drivers. Part of the problem is that most existing development tools do not warn users or mitigate against these types of problems. However, using judicious defensive development techniques, most of these problems can be eliminated.

This section includes the following topics:

Buffer Handling

Authentication and Identification

Access Control

Handle Management

Buffer Handling

4/26/2017 • 4 min to read • Edit Online

Perhaps the most common error within any driver relates to buffer handling where buffers are invalid or too small. These errors can allow buffer overflows or cause system crashes, which constitute security compromises for the system.

From the perspective of a driver, buffers come in one of two varieties:

- Paged buffers, which may or may not be resident in memory.
- Non-paged buffers, which must be resident in memory.

Of course, an invalid address is neither paged nor nonpaged, but as the operating system begins to work toward resolving the page fault such a buffer causes, it will isolate the invalid address into one of the "standard" address ranges (paged kernel addresses, non-paged kernel addresses, or user addresses) and raise the appropriate type of error. Buffer errors are always handled either by a bug check (PAGE_FAULT_IN_NONPAGED_AREA, for example) or by an exception (STATUS_ACCESS_VIOLATION, for example). In the case of a bug check, the system will halt operation. In the case of an exception, the stack-based exception handlers will be invoked, and if none of them handle the exception, then a bug check will be invoked.

Regardless, any access path that may be called by an application program that causes the driver to lead to a bug check is a security violation within the driver. This allows an application to cause denial-of-service attacks to the entire system.

One of the most common problems in this area is that driver writers assume too much about the operating environment. This could include:

- Checking that the high bit is set in the address. This does not work on x86-based computers where the system is using Four Gigabyte Tuning (4GT) by setting the /3GB option in the Boot.ini file. In that case, user-mode addresses set the high bit for the third gigabyte (GB) of the address space.
- Using **ProbeForRead** and **ProbeForWrite** to validate the address. While this will ensure that the address is a valid user-mode address at the time of the probe, there is nothing that requires it to remain valid after the probe operation. Thus, this technique introduces a subtle race condition that can lead to periodic irreproducible crashes. **ProbeForRead** and **ProbeForWrite** calls are necessary for a different reason: to validate whether the address is a user-mode address and that the length of the buffer is within the user address range. If the probe is omitted, users can pass in valid kernel-mode addresses, which will not be caught by a __try and __except block (structured exception handling) and will open up a large security hole. So **ProbeForRead** and **ProbeForWrite** calls are necessary to ensure alignment and that the user-mode address, plus the length, is within the user address range. However, a __try and __except block is needed to guard against access.

  Note that **ProbeForRead** only validates that the address and length fall within the possible user-mode address range (slightly under 2 GB for a system without 4GT, for example), not whether the memory address is valid. In contrast, **ProbeForWrite** will try to access the first byte in each page of the length specified to verify that these are valid memory addresses.
- Relying on memory manager functions (**MmIsAddressValid**, for example) to ensure that the address is valid. As with the probe functions, this introduces a race condition that can lead to irreproducible crashes.
- Failing to use structured exception handling. The __try and __except functions within the compiler use operating system-level support for exception handling. Exceptions at kernel level are thrown back by calling

ExRaiseStatus, or one of the related functions. A driver failing to use structured exception handling around any call that may raise an exception will lead to a bug check (typically KMODE_EXCEPTION_NOT_HANDLED).

  Note that it is mistake to use structured exception handling around code that is not expected to raise errors. This will just mask real bugs that would otherwise be found. Putting a __try and __except wrapper at the top dispatch level of your routine is not the correct solution to this problem, although it is sometimes the reflex solution tried by driver writers.

- Relying upon the contents of user memory remaining stable. For example, suppose a driver were to write a value into a user-mode memory location, and then later in the same routine refer to that memory location. A malicious application could actively modify that memory and, as a result, cause the driver to crash.

For file systems, these problems are particularly severe because they typically rely upon directly accessing user buffers (the METHOD_NEITHER transfer method). Such drivers directly manipulate user buffers and thus must incorporate precautionary methods for buffer handling in order to avoid operating system-level crashes. Fast I/O always passes raw memory pointers, so drivers need to protect against similar problems if fast I/O is supported.

The WDK contains numerous examples of buffer validation in the FASTFAT and CDFS file system sample code, including:

- The **FatLockUserBuffer** function in fastfat\deviosup.c uses **MmProbeAndLockPages** to lock down the physical pages behind the user buffer and **MmGetSystemAddressForMdlSafe** in **FatMapUserBuffer** to create a virtual mapping for the pages that are locked down.

- The **FatGetVolumeBitmap** function in fastfat\fsctl.c uses **ProbeForRead** and **ProbeForWrite** to validate user buffers in the defragmentation API.

- The **CdCommonRead** function in cdfs\read.c uses __try and __except around code to zero user buffers. Note that the sample code in **CdCommonRead** appears to use the try and except keywords. In the WDK environment, these keywords in C are defined in terms of the compiler extensions __try and __except. Anyone using C++ code must use the native compiler types to handle exceptions properly, as __try is a C++ keyword, but not a C keyword, and will provide a form of C++ exception handling that is not valid for kernel drivers.

Authentication and Identification

4/26/2017 • 1 min to read • Edit Online

Most drivers are not involved in authentication or identification issues, leaving this task to individual services. One case where a driver might become involved in authentication or identification processing is in access management. In this case, the authentication step is usually handled through calls to the Security Reference Monitor. Authentication and identification information is normally tracked by the operating system by the security token, an internal data structure that encapsulates the security credentials for a given thread or process.

Access Control

4/26/2017 • 1 min to read • Edit Online

To protect themselves from inappropriate access, most drivers rely upon the default access controls applied by the I/O manager against their device objects. Other mechanisms are available to drivers. Perhaps the simplest for normal drivers is to apply an explicit security descriptor when they install their driver. An example of applying security descriptors to the device object are discussed in a later section.

A driver that implements its own security policy could rely upon the standard Windows APIs for assistance managing security access. In this case, the driver manages the storage of security descriptors and is responsible for invoking the security reference monitor routines to validate security. These include numerous routines, such as the following:

- **SeAccessCheck**--this routine compares the security descriptor against the security credentials of the caller.
- **SePrivilegeCheck**--this routine determines if the given privileges are enabled for the caller.
- **SeSinglePrivilegeCheck**--this routine determines if a specific privilege is enabled for the caller.
- **SeAuditingFileOrGlobalEvents**--this routine indicates if the system has enabled auditing.
- **SeOpenObjectAuditAlarm**--this routine audits open object events.

This list is incomplete, but it describes a number of the key functions that can be used within a driver to perform access validation.

Handle Management

4/26/2017 • 2 min to read • Edit Online

A significant source of security issues within drivers is the use of handles passed between user-mode and kernel-mode components. There are a number of known problems with handle usage within the kernel environment, including the following:

- An application that passes the wrong type of handle to a kernel driver. The kernel driver might crash trying to use an event object where a file object is needed.
- An application that passes a handle to an object for which it does not have the necessary access. The kernel driver might perform an operation that works because the call comes from kernel mode, even though the user does not have adequate permissions to do so.
- An application that passes a value that is not a valid handle in its address space, but is marked as a system handle to perform a malicious operation against the system.
- An application that passes a value that is not an appropriate handle for the device object (a handle that this driver didn't create).

To protect against these issues, a kernel driver must be particularly careful to ensure that handles passed to it are valid. The safest policy is to create any needed handles within the context of the driver. These handles, created by kernel drivers, should specify the OBJ_KERNEL_HANDLE option, which will create a handle valid in arbitrary process context and one that can only be accessed from a kernel-mode caller.

For drivers that use handles created by an application program, the use of these handles must be done with extreme care:

- The best practice is to convert the handle to an object pointer by calling **ObReferenceObjectByHandle**, specifying the correct *AccessMode* (usually from Irp->RequestorMode), *DesiredAccess*, and *ObjectType* parameters, such as IoFileObjectType or ExEventObjectType.
- If a handle must be used directly within a call, it is best to use the Nt variants of functions rather than the Zw variants of functions. This will enforce parameter checking and handle validation by the operating system since the previous mode will be **UserMode** and hence untrusted. Note that parameters passed to Nt functions that are pointers may fail validation if the previous mode is **UserMode**. The Nt and Zw routines return an *IoStatusBlock* parameterwith error information that you should check for errors.
- Errors must be appropriately trapped as well using __try and __except as necessary. Many of the cache manager (Cc), memory manager (Mm), and file system runtime library routines (FsRtl) raise an exception when an error occurs.

No driver should ever rely on handles or parameters passed from a user-mode application without taking appropriate precautions.

Note that if the Nt variant is used to open a file, then the Nt variant must also be used to close the file.

Testing for Security

4/26/2017 • 1 min to read • Edit Online

Testing for security is not an automated process. Rather, it combines the use of existing tools as well as a thorough threat analysis of the given driver or drivers. Testing a driver thus consists of many separate steps:

- A thorough threat analysis to actively identify the types of attacks that might be made within the environment where the driver is used. For example, drivers that are present in highly controlled environments may have simpler threat analyses than drivers for mass distribution, which will be subject to arbitrary attacks.
- Testing with existing tools including the "Designed for Windows" logo tests in the WDK, such as Device Path Exerciser, a device control attack testing utility. The IFS tests in the WDK should also be used to thoroughly test for storage stack issues.
- Additional tests should be developed as part of normal driver development for specifically testing scenarios identified as part of the threat analysis. These tests would usually be unique to the driver and attempt to probe the particular driver.

Ideally, for testing purposes, the development of validation tests would include input from the original designers of the software, as well as unrelated development resources familiar with the specific type of file system or file system filter driver product being developed, and one or more people familiar with security intrusion analysis and prevention.

Security Features for File Systems

4/26/2017 • 1 min to read • Edit Online

Unlike most other types of drivers, file systems are intimately involved in normal security processing. This is because of the nature of security and its implementation within Microsoft Windows. The general Windows security model associates a security descriptor with an object--in this case, the FILE_OBJECT. File systems that support Windows security are responsible for storing and retrieving security descriptors. In addition, file systems are responsible for handing several other special security considerations that fall outside the normal scope of standard kernel-mode drivers.

This section discusses key features that may be added to a file system to support Windows security. None of these are mandatory and file systems can be constructed without using any of these interfaces. Further, it is possible to implement some security features while ignoring others--this is specific to the implementation of the file system.

This section includes the following topics:

Security Descriptors

4/26/2017 • 2 min to read • Edit Online

Windows file systems may support the storage and management of security descriptors associated with individual storage units within the file system. The granularity of security control is entirely up to the file system. For example, one file system might maintain a single security descriptor that covers everything on a given storage volume, while another might provide security descriptors that cover different parts of a single given file. The models with which most developers will be comfortable are those provided by the existing Windows file systems:

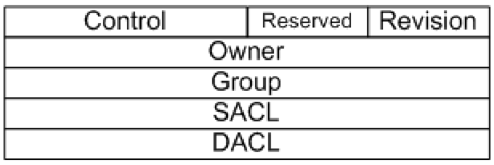
- NTFS—supports a per-file (or directory) security descriptor model. NTFS is efficient in its storage of security descriptors, storing only a single copy of each security descriptor, even if it is used by many different files.
- FAT, CDFS, UDFS—do not support security descriptors.
- RDBSS and the SMB Network Redirector—provide support comparable to that provided by the remote volume.

These file systems, however, do not represent all possible implementations of Windows security for file systems.

A Windows security descriptor consists of four distinct pieces:

- The security identifier (SID) of the owner of the object. An object's owner always has the ability to reset the security on the object. This is a good way to ensure that, for example, all access to an object can be removed. Because even if owners remove their ability to perform all operations, this inherent right allows them to restore their security rights on the object.
- An optional security identifier (SID) of the default group of the object. The concept of group ownership is one that is not required in Windows, but is useful for some applications.
- The system access control list (SACL) that describes the auditing policy of the security descriptor.
- The discretionary access control list (DACL) that describes the access policy of the security descriptor.

The following figure illustrates a windows security descriptor.



Security descriptors are variable-sized objects, with each of the individual sub-components being variable in size as well. To facilitate offline storage of security descriptors, a security descriptor may be in self-relative format, in which case the header is the offset within the buffer to the specific component of the security descriptor. An in-memory format consists of pointer values to the various parts of the security descriptor. For a file system, the self-relative format is normally the most useful because it allows for simple storage and retrieval of the security descriptor from persistent storage. Applications that build security descriptors are more likely to use the in-memory format. The security reference monitor provides conversion routines to convert from one format to the other.

This section includes the following topics:

Security Identifier

Access Mask

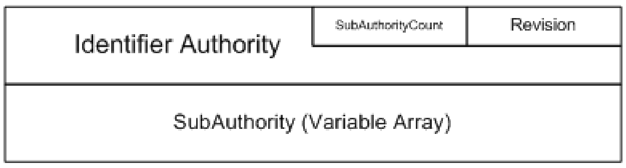
Access Control Entry

Security Identifier

4/26/2017 • 2 min to read • Edit Online

The security identifier is used by Windows as the definitive value to distinguish security entities from one another. For example, a unique security identifier is assigned to each new account created for individual users on the system. For a file system, only this SID is actually used.

The following figure illustrates the security identifier structure.



In addition to unique SIDs, the Windows system defines a set of well known identifiers. For example, the local Administrator is such a well-known SID. Windows provides an in-kernel mechanism for converting between SIDs and user names within the kernel environment. These function calls are available from the ksecdd driver, which implements these functions by using user-mode helper services. Accordingly, their use within file systems must obey the usual rules for communication with user-mode services. These calls cannot be used during paging file I/O.

The functions include the following:

- **SecMakeSPN**—creates a service provider name string that can be used when communicating with specific security service providers.
- **SecMakeSPNEx**—an augmented version of **SecMakeSPN**. This function is available on Microsoft Windows XP and later versions of Windows.
- **SecMakeSPNEx2**—an augmented version of **SecMakeSPNEx**. This function is available on Windows Vista, Windows Server 2008, and later versions of Windows.
- **SecLookupAccountSid**—given a SID, this routine will return an account name. This function is available on Windows XP and later.
- **SecLookupAccountName**—given an account name, this routine will retrieve the SID. This function is available on Windows XP and later.
- **SecLookupWellKnownSid**—given a well known SID type, this routine will return the correct SID. This function is available on Windows Server 2003 and later.

In addition, any kernel driver may create a SID by using the following standard runtime library routines:

- **RtlInitializeSid**—initializes a buffer for a new SID.
- **RtlLengthSid**—determines the size of the SID stored within the given buffer.
- **RtlValidSid**—determines if the given SID buffer is a valid formatted buffer.

Note that **RtlLengthSid** and **RtlValidSid** assume that the 8-byte fixed header for a SID is present. So a driver should check for this minimum length for a SID header before calling these functions.

While there are several other RTL functions, these are the primary functions necessary when constructing a SID.

The following code example demonstrates how to create a SID for the "local system" entity:

```
{
    //
    // temporary stack-based storage for an SID
    //
    UCHAR sidBuffer[128];
    PISID localSid = (PISID) sidBuffer;
    SID_IDENTIFIER_AUTHORITY localSidAuthority =
        SECURITY_NT_AUTHORITY;

    //
    // build the local system SID
    //
    RtlZeroMemory(sidBuffer, sizeof(sidBuffer));

    localSid->Revision = SID_REVISION;
    localSid->SubAuthorityCount = 1;
    localSid->IdentifierAuthority = localSidAuthority;
    localSid->SubAuthority[0] = SECURITY_LOCAL_SYSTEM_RID;

    //
    // make sure it is valid
    //
    if (!RtlValidSid(localSid)) {
        DbgPrint("no dice - SID is invalid\n");
        return(1);
    }
}
```

Note that this could have also been done using the simpler function **SecLookupWellKnownSid** introduced in Windows Server 2003.

The following code example demonstrates how to create a SID using the **SecLookupWellKnownSid** function for the "local system" entity:

```
{
    UCHAR sidBuffer[128];
    PISID localSid = (PISID) sidBuffer;
    SIZE_T sidSize;
    status = SecLookupWellKnownSid(WinLocalSid,
                                   &localSid,
                                   sizeof(sidBuffer),
                                   &sidSize);

    if (!NT_SUCCESS(status)) {
      //
      // error handling
      //
    }
  }
```

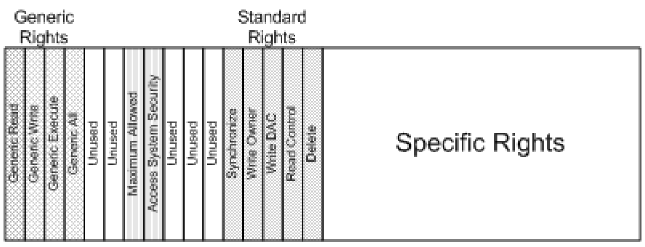
Either of these approaches are valid, although the latter code is preferred. Note that these code examples use local buffers for storing the SID. These buffers cannot be used outside the current call context. If the SID buffer needed to be persistent, the buffer should be allocated from pool memory.

Access Mask

4/26/2017 • 3 min to read • Edit Online

The function of the access mask is to describe access rights in a compact form. To simplify access management, the access mask contains a set of four bits, the *generic rights*, which are translated into a set of more detailed rights by using the function **RtlMapGenericMask**.

The following figure illustrates the access mask.



The generic rights are one of the following:

- GENERIC_READ—the right to read the information maintained by the object.
- GENERIC_WRITE—the right to write the information maintained by the object.
- GENERIC_EXECUTE—the right to execute or alternatively look into the object.
- GENERIC_ALL—the right to read, write, and execute the object.

Note that these rights can be combined (GENERIC_READ and GENERIC_WRITE can both be requested, for example) with the resulting mapping requiring the union of the rights needed for each generic right. This paradigm mimics UNIX "rwx" access bits that are used to control access to UNIX resources. The generic rights in the access mask simplify application development on Windows since these rights mask the different security rights for various object types.

The following set of *standard rights* are applicable to all object types:

- DELETE—the right to delete the particular object.
- READ_CONTROL—the right to read the control (security) information for the object.
- WRITE_DAC—the right to modify the control (security) information for the object.
- WRITE_OWNER—the right to modify the owner SID of the object. Recall that owners always have the right to modify the object.
- SYNCHRONIZE—the right to wait on the given object (assuming that this is a valid concept for the object).

The lower 16 bits of the access mask represent the specific rights. The meaning of these specific rights is unique to the object in question. For file systems, the primary interests are the specific rights for file objects. For file objects, specific rights are normally interpreted differently, depending upon whether the file object represents a file or a directory. For files, the normal interpretation is:

- **FILE_READ_DATA**—the right to read data from the given file.
- **FILE_WRITE_DATA**—the right to write data to the given file (within the existing range of the file).
- **FILE_APPEND_DATA**—the right to extend the given file.

- **FILE_READ_EA**—the right to read the extended attributes of the file.
- **FILE_WRITE_EA**—the right to modify the extended attributes of the file.
- **FILE_EXECUTE**—the right to locally execute the given file. Executing a file stored on a remote share requires read permission, since the file is read from the server, but executed on the client.
- **FILE_READ_ATTRIBUTES**—the right to read the file's attribute information.
- **FILE_WRITE_ATTRIBUTES**—the right to modify the file's attribute information.

For directories, the same bit values are used, but their interpretation is different in some of the following cases:

- **FILE_LIST_DIRECTORY**—the right to list the contents of the directory.
- **FILE_ADD_FILE**—the right to create a new file within the directory.
- **FILE_ADD_SUBDIRECTORY**—the right to create a new directory (subdirectory) within the directory.
- **FILE_READ_EA**—the right to read the extended attributes of the given directory.
- **FILE_WRITE_EA**—the right to write the extended attributes of the given directory.
- **FILE_TRAVERSE**—the right to access objects within the directory. The FILE_TRAVERSE access right is different than the FILE_LIST_DIRECTORY access right. Holding the FILE_LIST_DIRECTORY access right allows an entity to obtain a list of the contents of a directory, while the FILE_TRAVERSE access right gives an entity the right to access the object. A caller without the FILE_LIST_DIRECTORY access right could open a file that it knew already existed, but would not be able to obtain a list of the contents of the directory.
- **FILE_DELETE_CHILD**—the right to delete a file or directory within the current directory.
- **FILE_READ_ATTRIBUTES**—the right to read a directory's attribute information.
- **FILE_WRITE_ATTRIBUTES**—the right to modify a directory's attribute information.

The actual mapping of generic rights to standard and specific rights for file objects is defined by the I/O manager. This mapping can be retrieved by a file system using **IoGetFileObjectGenericMapping**. Normally, this mapping is done during IRP_MJ_CREATE processing by the I/O manager prior to calling the file system. But this might be done by a file system checking security on specific operations (specialized FSCTL operations, for example).

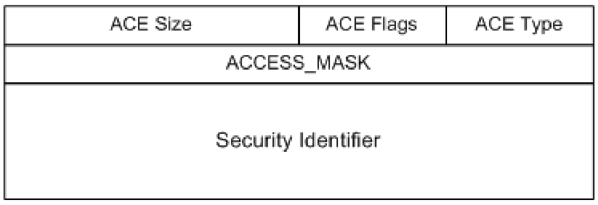
Access Control Entry

4/26/2017 • 1 min to read • Edit Online

An access control entry (ACE) describes access rights associated with a particular SID. The access control entry is evaluated by the operating system in order to compute the effective access granted to a particular program based upon its credentials. For example, when a user logs on to the computer, and then executes a program, the program uses the credentials associated with that particular user's account.

Thus, when a program attempts to open an object, Windows compares the credentials associated with the program against the security controls associated with the object. The security reference monitor then uses the ACE information to determine if the program should be allowed or denied access to the given object. Thus, the ACE determines the behavior of the security subsystem.

The following figure illustrates the access control entry.



There are five types of ACEs used by the security subsystem. The **Type** member of the ACE structure controls the interpretation of the ACE. The defined types are:

- **ACCESS_ALLOWED_ACE_TYPE**—this type indicates that the ACE specifies access rights that will be granted to the specific SID.
- **ACCESS_DENIED_ACE_TYPE**—this type indicates that the ACE specifies access rights that are to be denied to the specific SID.
- **SYSTEM_AUDIT_ACE_TYPE**—this type indicates that the ACE specifies auditing behavior.
- **SYSTEM_ALARM_ACE_TYPE**—this type indicates that the ACE specifies alarm behavior.
- **ACCESS_ALLOWED_COMPOUND_ACE_TYPE**—this type indicates that the ACE is tied to a particular server and the entity it is impersonating.

Thus, three of the types are used to control programmatic access to an object, while the other two are used to control the audit and alarm behavior of the security subsystem when the object is accessed. Note that the actual behavior of the security subsystem is computed by combining the information for some or all of the ACEs associated with the object.

A driver may construct an access control entry of ACCESS_ALLOWED_ACE_TYPE using the routine **RtlAddAccessAllowedAce**. For adding the other types of ACE entries, driver writers must construct their own functions because the WDK does not provide any other support routines.

Access Control List

4/26/2017 • 2 min to read • Edit Online

An access control list (ACL) is a list of ACEs created by the operating system to control the security behavior associated with a given (protected) object of some sort. In Windows there are two types of ACLs:

- **Discretionary ACL**--this is a list of zero or more ACEs that describe the access rights for a protected object. It is discretionary because the access granted is at the discretion of the owner or any user with appropriate rights.
- **System ACL**--this is a list of zero or more ACEs that describe the auditing and alarm policy for a protected object.

The term "discretionary" refers to the differentiation between mandatory and discretionary control. In an environment that uses mandatory controls, the owner of an object may not be able to grant access to the object. In a discretionary environment, such as Windows, the owner of an object is allowed to grant such access. Mandatory controls are normally associated with very tight security environments, such as those using compartmentalized security, where the system must prevent disclosure of sensitive information between users on the same system.

A driver constructing an ACL would follow a few key steps:

1. Allocate storage for the ACL.
2. Initialize the ACL.
3. Add zero (or more) ACEs to the ACL.

The following code examples demonstrate how to construct an ACL:

```
dacl = ExAllocatePool(PagedPool, PAGE_SIZE);
if (!dacl) {
    return;
}
status = RtlCreateAcl(dacl, PAGE_SIZE, ACL_REVISION);
if (!NT_SUCCESS(status)) {
    ExFreePool(dacl);
    return;
}
```

The previous code fragment creates an empty ACL. The code sample allocates a significant amount of memory, since we do not know the size required for the ACL.

At this point, the ACL is empty because it has no ACE entries. An empty ACL denies access to anyone trying to access the object because there are no entries that grant such access. The following code fragment adds an ACE to this ACL:

```
status = RtlAddAccessAllowedAce(dacl, ACL_REVISION,  FILE_ALL_ACCESS, SeExports->SeWorldSid);
if (!NT_SUCCESS(status)) {
    ExFreePool(dacl);
    return;
}
```

This entry would now grant access to any entity that accessed the object. This is the purpose of world access SID (SeWorldSid), which is usually represented as "Everyone" access in other Windows system utilities.

Note that when constructing ACLs, it is important to order access denied ACE entries at the beginning of the ACL, and then access allowed ACE entries at the end of the ACL. This is because when the security reference monitor does the evaluation of the ACL it will grant access if it finds an ACE granting access, before it finds the denied ACEs. This behavior is well documented in the Microsoft Windows SDK, but it relates to the specific mechanism the security reference monitor uses to determine whether access should be granted or denied.

Privileges

4/26/2017 • 2 min to read • Edit Online

Privilege is a completely separate mechanism that is used by the operating system to control specific operations. Each privilege has associated with it particular operations that may be performed if the privilege is held and enabled by the caller. Note the two conditions here:

- The privilege must be held by the caller.
- The privilege must also be enabled.

The principle here, known as least privilege, requires that privileges be enabled prior to their use, rather than simply assumed, in order to minimize the chance that a user might inadvertently perform an operation they did not intend. For example, **SeRestorePrivilege** would normally allow the caller to bypass the usual checks for write access to a file. An administrator may not wish to actually override the normal security checks when copying a file, but would wish to do so when restoring that same file using a backup/restore utility.

For file systems, there are a number of privileges that are often used to modify the normal behavior (notably, security checks) for the system. These privileges are:

- **SeBackupPrivilege**--allows file content retrieval, even if the security descriptor on the file might not grant such access. A caller with **SeBackupPrivilege** enabled obviates the need for any ACL-based security check.
- **SeRestorePrivilege**--allows file content modification, even if the security descriptor on the file might not grant such access. This function can also be used to change the owner and protection.
- **SeChangeNotifyPrivilege**--allows traverse right. This privilege is an important optimization in Windows, since the cost of performing a security check on every single directory in a path is obviated by holding this privilege.
- **SeManageVolumePrivilege**--allows specific volume-level management operations, such as lock volume, defragmenting, volume dismount, and setting valid data length on Windows XP and later. Note that this particular privilege is explicitly enforced by a file system driver primarily based upon FSCTL operations. In this case, the file system makes a policy decision to enforce this privilege. The determination of whether this privilege is held by the caller is made by the security reference monitor as part of the normal privilege check.

While there are numerous other privileges, they are normally opaque to file systems and are thus only interpreted by the security reference monitor.

The key Windows routines for managing privileges within a file system are:

- **SePrivilegeCheck**--this routine performs a check for a specific set of necessary privileges.
- **SeSinglePrivilegeCheck**--this routine performs a check for a single specific privilege; it is an optimized version of **SePrivilegeCheck**.
- **SeAccessCheck**--this routine performs normal access checking on an object (normally a file object for a file system).
- **SeFreePrivileges**--this routine frees the privilege block returned by a previous call to **SeAccessCheck**.
- **SeAppendPrivileges**--this routine adds enabled privileges to an ACCESS_STATE structure. Typically, a file system would use the ACCESS_STATE passed to it during IRP_MJ_CREATE processing.

Auditing

4/26/2017 • 1 min to read • Edit Online

One of the tenets of good security design is to admit that there is no such thing as a secure system. Developers for the system must be aware that people will circumvent whatever security is present. This could be done actively, for example, by probing the security subsystem to find and exploit holes within the system. Or this could be accidental, for example, inadvertently overwriting or deleting critical data. Whatever the cause, it is imperative to construct a system that can detect such breaches.

The auditing system within Windows provides a mechanism for tracking specific security events so that the log can be analyzed at a later time to perform post-mortem analysis of a damaged or compromised system. The auditing mechanism on Windows intimately involves the file system because the file system is responsible for maintaining the persistent storage of system data. For many systems, security needs are much lower, and in those cases, auditing is disabled. File systems must be implemented in such a way that they can address the concerns of both these environments.

Key routines for auditing include:

- **SeAuditingFileEvents**--this routine determines whether file auditing has been enabled on the system; this is a global policy check to determine whether a full audit check should be done. This routine was introduced to optimize the security system operations.
- **SeOpenObjectAuditAlarm**--this routine performs the primary audit operations in the Windows system (audits an attempt to open an object). Note that it is the attempt to access the object that is audited, not whether access to the object was successful or unsuccessful.

There is no requirement for auditing. None of the sample file systems (FAT or CDFS, for example) in the IFS section of the WDK implement auditing. However, from a security perspective, auditing is important because it allows administrators to monitor the security behavior of the system.

Kernel Extended Attributes

4/26/2017 • 4 min to read • Edit Online

Kernel Extended Attributes (Kernel EA's) are a feature added to NTFS in Windows 8 as a way to boost the performance of image file signature validation. It is an expensive operation to verify an images signature. Therefore, storing information about whether a binary, which has previously been validated, has been changed or not would reduce the number of instances where an image would have to undergo a full signature check.

Overview

EA's with the name prefix `$Kernel` can only be modified from kernel mode. Any EA that begins with this string is considered a Kernel EA. Before retrieving the necessary update sequence number (USN), it is recommended that **FSCTL_WRITE_USN_CLOSE_RECORD** be issued first as this will commit any pending USN Journal updates on the file that may have occurred earlier. Without this, the **FileUSN** value may change shortly after setting of the Kernel EA.

It is recommended that a Kernel EA contains at least the following information:

- USN UsnJournalID
  - The **UsnJournalID** field is a GUID that identifies the current incarnation of USN Journal File. The USN Journal can be deleted and created from user mode per volume. Each time the USN Journal is created a new **UsnJournalID** GUID will be generated. With this field, you can tell if there was a period of time where the USN Journal was disabled and can revalidate.
    - This value can be retrieved using FSCTL_QUERY_USN_JOURNAL.
- USN FileUSN
  - The **FileUSN** value contains the USN ID of the last change that was made to the file and is tracked inside the Master File Table (MFT) record for the given file.
    - When the USN Journal is deleted, **FileUSN** is reset to zero.

This information, along with any other a given usage might need, is then set on the file as a Kernel EA.

Setting a Kernel Extended Attribute

In order to set a Kernel EA, it must begin with the prefix `"$Kernel."` and be trailed by a valid EA name string. An attempt to set a Kernel EA from user mode will be silently ignored. The request will return **STATUS_SUCCESS** but no actual EA modification will be made. To set a Kernel EA calling an API like ZwSetEaFile or FltSetEaFile from kernel mode is not sufficient. This is because SMB supports the setting of EA's across the network and those requests will be issued from kernel mode on the server.

To set a Kernel EA the caller must also set the **IRP_MN_KERNEL_CALL** value in the MinorFunction field of the IRP (I/O request packet). Since the only way to set this field is by generating a custom IRP, the routine FsRtlSetKernelEaFile has been exported from the FsRtl package as a support function to set up a Kernel EA.

You may not intermix the setting of normal and kernel EA's in the same call to FsRtlSetKernelEaFile. If you do this the operation will be failed with **STATUS_INTERMIXED_KERNEL_EA_OPERATION**. Setting a Kernel EA will not generate a **USN_REASON_EA_CHANGE** record to the USN Journal; as a consequence, kernel EA's and regular EA's cannot be used in the same operation.

Querying an Extended Attribute

Querying the EA's on a file from user mode will return both normal and Kernel EA's. They are returned to user mode to minimize any application compatibility issues. The normal ZwQueryEaFile and FltQueryEaFile operations will return both normal and kernel EA's from both user and kernel modes.

When only a **FileObject** is available, using FsRtlQueryKernelEaFile may be more convenient for use to query for Kernel EA's from kernel mode.

Querying Update Sequence Number Journal Information

The FSCTL_QUERY_USN_JOURNAL operation requires **SE_MANAGE_VOLUME_PRIVILEGE** even when issued from kernel mode unless the **IRP_MN_KERNEL_CALL** value was set in the MinorFunction field of the IRP. The routine **FsRtlKernelFsControlFile** has been exported from the FsRtl package in the Kernel to easily allow kernel mode components to issue this USN request.

NOTE Starting with Windows 10, version 1703 and later this operation no longer requires SE_MANAGE_VOLUME_PRIVILEGE.

Auto-Deletion of Kernel Extended Attributes

Simply rescanning a file because the USN ID of the file changed can be expensive as there are many benign reasons a USN update may be posted to the file. To simplify this, an auto delete of Kernel EA's feature was added to NTFS.

Because not all Kernel EA's may want to be deleted in this scenario, an extended EA prefix name is used. If a Kernel EA begins with: `"$Kernel.Purge."` then if any of the following USN reasons are written to the USN journal, NTFS will delete all kernel EAs that exist on that file that conforms to the given naming syntax:

- USN_REASON_DATA_OVERWRITE
- USN_REASON_DATA_EXTEND
- USN_REASON_DATA_TRUNCATION
- USN_REASON_REPARSE_POINT_CHANGE

This delete of Kernel EA's will be successful even in low memory situations.

Remarks

- Kernel EA's cannot be tampered with by user mode components.
- Kernel EA's can exist in the same file as a normal EA.

See Also

FltQueryEaFile
FltSetEaFile
FSCTL_QUERY_USN_JOURNAL
FsRtlQueryKernelEaFile
FsRtlSetKernelEaFile
ZwQueryEaFile
ZwSetEaFile

File System Security Issues

4/26/2017 • 1 min to read • Edit Online

The previous section described security considerations in general terms. In addition to general security issues of interest to all drivers, there are specific security issues related to file systems. This section attempts to provide a guide for those file systems looking to implement Windows-style security within their driver. This section discusses those specific issues and how they can be addressed within a file system. This section is not a reference guide nor does it provide a complete list of all possible security threats and how they may be mitigated. But rather, the goal of this section is to identify well known threats and key issues related to security that should be addressed by all file systems. Because the area of security itself is sufficiently broad, this document does not cover all possible security issues or implementations.

This section includes the following topics:

Semantic Model Checks

Security Checks

Semantic Model Checks

4/26/2017 • 1 min to read • Edit Online

In discussing file systems and security, we distinguish between those checks that the file system makes that are part of its semantic model, such as shared file access or special file attributes, and those checks that the file system makes that are part of the security information of the file. This section focuses on checks needed to comply with the semantic model. A later section discusses specific steps needed by a file system to perform security checks that are specific to the security information policies of the file system.

This section includes the following topics:

Create Processing

4/26/2017 • 5 min to read • Edit Online

For a file system, most of the interesting security work occurs during **IRP_MJ_CREATE** processing. It is this step that must analyze the incoming request, determine whether the caller has appropriate rights to perform the operation, and grant or deny the operation as appropriate. Fortunately, for file system developers, most of the decision mechanism is implemented within the Security Reference Monitor. Thus, in most cases, the file system need only call the appropriate Security Reference Monitor routines to properly determine access. The risk for a file system occurs when it fails to call these routines as necessary and inappropriately grants access to a caller.

For a standard file system, such as the FAT file system, the checks that are made as part of IRP_MJ_CREATE are primarily semantics checks. For example, the FAT file system has numerous checks to ensure that IRP_MJ_CREATE processing is allowed based upon the state of the file or directory. These checks made by the FAT file system include checks for read-only media (for example, attempts to perform destructive "create" operations, such as overwrite or supersede, on read-only media are not allowed), share access checks, and oplock checks. One of the most difficult parts of this analysis is to realize that an operation at one level (the file level, for example) may in fact be disallowed because of the state of a different level resource (the volume level, for example). For example, a file may not be opened if another process has exclusively locked the volume. Common cases to check would include:

- Is the file level open compatible with the volume level state? Volume-level locking must be obeyed. Thus, if one process holds an exclusive volume level lock, only threads within that process can open files. Threads from other processes must not be allowed to open files.

- Is the file level open compatible with the media state? Certain "create" operations modify the file as part of the "create" operation. This would include overwrite, supersede, and even updating the last access time on the file. These "create" operations are not allowed on read-only media and the last access time is not updated.

- Is the volume level open compatible with the file level state? An exclusive volume open would not be allowed if there are existing files opened on the volume. This is a common problem for new developers because they attempt to open the volume and find that it fails. When this fails, FSCTL_DISMOUNT_VOLUME can be used to invalidate open handles and force a dismount, allowing exclusive access to the newly mounted volume.

In addition, file attributes must be compatible. A file with the read-only attribute cannot be opened for write access. Note that the desired access should be checked after expansion of the generic rights are expanded. For example, this check within the FASTFAT file system is in the **FatCheckFileAccess** function (see the Acchksup.c source file from the fastfat samples that the WDK contains).

The following code example is specific to the FAT semantics. A file system that implements DACLs as well, would do an additional security check using the Security Reference Monitor routines (**SeAccessCheck**, for example.)

```
    //
    //  check for a read-only Dirent
    //

    if (FlagOn(DirentAttributes, FAT_DIRENT_ATTR_READ_ONLY)) {

        //
        //  Check the desired access for a read-only Dirent
        // Don&#39;t allow
        //  WRITE, FILE_APPEND_DATA, FILE_ADD_FILE,
        //  FILE_ADD_SUBDIRECTORY, and FILE_DELETE_CHILD
        //

        if (FlagOn(*DesiredAccess, ~(DELETE |
                                     READ_CONTROL |
                                     WRITE_OWNER |
                                     WRITE_DAC |
                                     SYNCHRONIZE |
                                     ACCESS_SYSTEM_SECURITY |
                                     FILE_READ_DATA |
                                     FILE_READ_EA |
                                     FILE_WRITE_EA |
                                     FILE_READ_ATTRIBUTES |
                                     FILE_WRITE_ATTRIBUTES |
                                     FILE_EXECUTE |
                                     FILE_LIST_DIRECTORY |
                                     FILE_TRAVERSE))) {

            DebugTrace(0, Dbg, "Cannot open readonly\n", 0);

            try_return( Result = FALSE );
        }
```

A more subtle check implemented by FASTFAT is to ensure that the access requested by the caller is something about which the FAT file system is aware (in the **FatCheckFileAccess** function in Acchksup.c from the fastfat sample that the WDK contains):

The following code example demonstrates an important concept for file system security. Check to ensure that what is passed to your file system does not fall outside the bounds of what you expect. The conservative and proper approach from the perspective of security is that if you do not understand an access request, you should reject that request.

```
    //
    // Check the desired access for the object.
    // Reject what we do not understand.
    // The model of file systems using ACLs is that
    // they do not type the ACL to the object that the
    // ACL is on.
    // Permissions are not checked for consistency vs.
    // the object type - dir/file.
    //

    if (FlagOn(*DesiredAccess, ~(DELETE |
                             READ_CONTROL |
                             WRITE_OWNER |
                             WRITE_DAC |
                             SYNCHRONIZE |
                             ACCESS_SYSTEM_SECURITY |
                             FILE_WRITE_DATA |
                             FILE_READ_EA |
                             FILE_WRITE_EA |
                             FILE_READ_ATTRIBUTES |
                             FILE_WRITE_ATTRIBUTES |
                             FILE_LIST_DIRECTORY |
                             FILE_TRAVERSE |
                             FILE_DELETE_CHILD |
                             FILE_APPEND_DATA))) {

        DebugTrace(0, Dbg, "Cannot open object\n", 0);

        try_return( Result = FALSE );
    }
```

Fortunately for file systems, once the security check has been done during the initial create processing, subsequent security checks are performed by the I/O manager. Thus, for example, the I/O manager ensures that user-mode applications do not perform a write operation against a file that has been opened only for read access. In fact, a file system should not attempt to enforce read-only semantics against the file object, even if it was opened only for read access, during the IRP_MJ_WRITE dispatch routine. This is due to the way the memory manager associates a specific file object with a given section object. Subsequent writing through that section will be sent as IRP_MJ_WRITE operations on the file object, even though the file was opened read-only. In other words, the access enforcement is done when a file handle is converted into the corresponding file object at Nt system service entry points by **ObReferenceObjectByHandle**.

There are two additional places within a file system where semantic security checks must be made similar to "create" processing:

- During rename or hard link processing.
- When processing file system control operations.

Rename processing and file system control processing is discussed in subsequent sections.

Note that this is not an exhaustive list of semantic issues related to "create" processing. The intent of this section is to draw attention to these issues for file system developers. All semantic issues must be identified for a specific file system, implemented to meet the specific semantics, and tested to ensure that the implementation handles the various cases.

Delete on Close

4/26/2017 • 1 min to read • Edit Online

When a caller specifies the **FILE_DELETE_ON_CLOSE** option, it is necessary for the file system check to ensure that the caller has delete permission on the file or delete child permission on the parent directory. Either permission is sufficient to allow a file to be deleted. This is an important case for file systems to handle. The semantics of the operation, which delete the file when it is closed, are not enforced by the I/O manager but by the file system.

The file system may also need to check that the volume is not write protected and that this operation does not apply to a directory where this operation is not allowed. For example, the FASTFAT file system code does checks for a write-protected volume and does not allow the root directory to be deleted using FILE_DELETE_ON_CLOSE. An example of these checks can be found in the **FatCommonCreate** function in the Create.c source file from the fastfat sample that the WDK contains.

Executable Images

4/26/2017 • 1 min to read • Edit Online

Executable files are loaded into the address space of a process using a memory mapped image file. The file itself is not required to be opened nor does a handle need to be created because the mapping is done by means of a section. File systems must check to enforce these special semantics, assuming that they support memory mapped files. For example, the FASTFAT file system code to check for this case can be found in the **FatOpenExistingFCB** function in the Create.c source file from the fastfat samples that the WDK contains:

```
//
//  If the user wants write access to the file, make sure there
//  is not a process mapping this file as an image. Any attempt to
//  delete the file will be stopped in fileinfo.c
//
//  If the user wants to delete on close, check at this
//  point though.
//

if (FlagOn(*DesiredAccess, FILE_WRITE_DATA) || DeleteOnClose) {

    Fcb->OpenCount += 1;
    DecrementFcbOpenCount = TRUE;

    if (!MmFlushImageSection( &Fcb->NonPaged->SectionObjectPointers,
                              MmFlushForWrite )) {

        Iosb.Status = DeleteOnClose ? STATUS_CANNOT_DELETE :STATUS_SHARING_VIOLATION;
        try_return( Iosb );
    }
}
```

Thus, the file system ensures that a memory mapped file, including an executable image, cannot be deleted even though the file is not open.

Rename and Hard Link Processing

4/26/2017 • 2 min to read • Edit Online

An area of particular concern for file systems is the proper handling of rename operations. A similar area of concern is hard link creation for file systems that support hard links. For rename operations, it is possible for a file system to delete a file (the target of the rename operation), which requires additional security checks by the file system.

When looking at the control structure for a rename operation, one of the structure fields is the **ReplaceIfExists** option:

```
typedef struct _FILE_RENAME_INFORMATION {
    BOOLEAN ReplaceIfExists;
    HANDLE RootDirectory;
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_RENAME_INFORMATION, *PFILE_RENAME_INFORMATION;
```

Similarly, in the hard link operation's control structure, one of the structure fields is the **ReplaceIfExists** option:

```
typedef struct _FILE_LINK_INFORMATION {
    BOOLEAN ReplaceIfExists;
    HANDLE RootDirectory;
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_LINK_INFORMATION, *PFILE_LINK_INFORMATION;
```

In both cases, the option is to replace the target of the operation, if it exists. While the FASTFAT file system does not support hard links, it does support rename operations. These semantics and behavior can be seen within the FASTFAT file system source code in the **FatSetRenameInfo** function (see the *Fileinfo.c* source file from the fastfat samples that the WDK contains).

The following code example to handle a rename operation mimics the file system checks for deleting the file. For a file system with a more robust security model (NTFS, for example), this check would also require security checking to ensure that the caller was allowed to delete the given file (the caller had the appropriate permissions required for deletion).

```
    //
    //  The name already exists. Check if the user wants
    //  to overwrite the name and has access to do the overwrite.
    //  We cannot overwrite a directory.
    //

    if ((!ReplaceIfExists) ||
        (FlagOn(TargetDirent->Attributes, FAT_DIRENT_ATTR_DIRECTORY)) ||
        (FlagOn(TargetDirent->Attributes, FAT_DIRENT_ATTR_READ_ONLY))) {

        try_return( Status = STATUS_OBJECT_NAME_COLLISION );
    }

    //
    //  Check that the file has no open user handles; otherwise,
    //  access will be denied. To do the check, search
    //  the list of FCBs opened under the parent Dcb, and make
    //  sure that none of the matching FCBs have a nonzero unclean count or
    //  outstanding image sections.
    //

    for (Links = TargetDcb->Specific.Dcb.ParentDcbQueue.Flink;
            Links != &TargetDcb->Specific.Dcb.ParentDcbQueue; ) {

        TempFcb = CONTAINING_RECORD( Links, FCB, ParentDcbLinks );

        //
        //  Advance now. The image section flush may cause the final
        //  close, which will recursively happen underneath of us here.
        //  It would be unfortunate if we looked through free memory.
        //

        Links = Links->Flink;

        if ((TempFcb->DirentOffsetWithinDirectory == TargetDirentOffset) &&
                ((TempFcb->UncleanCount != 0) ||
                !MmFlushImageSection( &TempFcb->NonPaged->SectionObjectPointers,
                MmFlushForDelete))) {

            //
            //  If there are batch oplocks on this file, then break the
            //  oplocks before failing the rename.
            //

            Status = STATUS_ACCESS_DENIED;

            if ((NodeType(TempFcb) == FAT_NTC_FCB) &&
                    FsRtlCurrentBatchOplock( &TempFcb->Specific.Fcb.Oplock )) {

                //
                //  Do all of the cleanup now since the IrpContext
                //  could go away when this request is posted.
                //

                FatUnpinBcb( IrpContext, TargetDirentBcb );

                Status = FsRtlCheckOplock( &TempFcb->Specific.Fcb.Oplock,
                    Irp,
                    IrpContext,
                    FatOplockComplete,
                    NULL );

                if (Status != STATUS_PENDING) {

                    Status = STATUS_ACCESS_DENIED;
                }
            }

            try_return( NOTHING );
        }
    }

    //
    //  OK, this target is finished. Remember the Lfn offset.
    //

    TargetLfnOffset = TargetDirentOffset -
        FAT_LFN_DIRENTS_NEEDED(&TargetLfn) * sizeof(DIRENT);

    DeleteTarget = TRUE;
}
```

Set File Information Processing

4/26/2017 • 1 min to read • Edit Online

The I/O manager executes some additional checks for a subset of the information classes supported by **IRP_MJ_SET_INFORMATION**. Specifically, for FileRenameInformation, FileLinkInformation and FileMoveClusterInformation, the I/O manager issues an open to the parent directory of the target name to ensure that the user has access to create a child under that parent before it sends down the IRP_MJ_SET_INFORMATION request to the file system.

Neither I/O Operations

4/26/2017 • 4 min to read • Edit Online

A file system must handle operations that typically involve directly manipulating user buffers. Such operations are inherently risky because the user address might not be valid. File systems must be particularly conscious of such operations and ensure that they protect them appropriately. The following operations rely upon the **Flags** member of the file system's device object to specify how the I/O manager is to transfer data between user and kernel address space:

- **IRP_MJ_DIRECTORY_CONTROL**
- **IRP_MJ_QUERY_EA**
- **IRP_MJ_QUERY_QUOTA**
- **IRP_MJ_READ**
- **IRP_MJ_SET_EA**
- **IRP_MJ_SET_QUOTA**
- **IRP_MJ_WRITE**

Typically, a file system chooses neither I/O implicitly by setting neither DO_DIRECT_IO nor DO_BUFFERED_IO in the **Flags** member of the volume device object that it creates.

The following operation ignores the **Flags** member of the file system's device object and uses neither I/O to transfer data between user and kernel address space:

- **IRP_MJ_QUERY_SECURITY**

Using neither I/O, the file system is responsible for handling its own data transfer operations. This allows a file system to satisfy an operation by directly placing the data into the user-space buffer of an application. The file system must thus ensure that the user's buffer is valid when the operation begins and gracefully handle the buffer becoming invalid while the operation is ongoing. Fast I/O also passes raw pointers. Developers should be aware that checking the validity of the buffer at the beginning of the operation is not sufficient to ensure that it remains valid throughout the operation. For example, a malicious application could map a block of memory (through a section, for example), issue an I/O operation, and unmap the block of memory while the I/O operation is ongoing.

There are several ways for a file system to handle this situation. One mechanism is to lock down the physical memory that corresponds to the user's address and create a second mapping in the operating system's address space. This ensures that the file system uses a virtual address that it controls. So even if the user address becomes invalid, the address created by the file system will remain valid. The FASTFAT file system code uses two different functions to achieve this. The first function locks down the user's buffer:

```
VOID
FatLockUserBuffer (
    IN PIRP_CONTEXT IrpContext,
    IN OUT PIRP Irp,
    IN LOCK_OPERATION Operation,
    IN ULONG BufferLength
    )

/*++
Routine Description:

    This routine locks the specified buffer for the specified type of
    access. The file system requires this routine because it does not
    ask the I/O system to lock its buffers for direct I/O. This routine
    can only be called from the file system driver (FSD) while still in the user context.

    Note that this is the *input/output* buffer.

Arguments:
    Irp - Pointer to the Irp for which the buffer will be locked.
    Operation - IoWriteAccess for read operations, or IoReadAccess for
                write operations.
    BufferLength - Length of user buffer.

Return Value:
    None
--*/

{
    PMDL Mdl = NULL;

    if (Irp->MdlAddress == NULL) {
        //
        // Allocate the Mdl and Raise if the allocation fails.
        //
        Mdl = IoAllocateMdl( Irp->UserBuffer, BufferLength, FALSE, FALSE, Irp );
        if (Mdl == NULL) {
            FatRaiseStatus( IrpContext, STATUS_INSUFFICIENT_RESOURCES );
        }

        //
        // now probe the buffer described by the Irp. If there is an exception,
        // deallocate the Mdl and return the appropriate "expected" status.
        //
        try {
            MmProbeAndLockPages( Mdl,
                                 Irp->RequestorMode,
                                 Operation );
        } except(EXCEPTION_EXECUTE_HANDLER) {
            NTSTATUS Status;
            Status = GetExceptionCode();
            IoFreeMdl( Mdl );
            Irp->MdlAddress = NULL;
            FatRaiseStatus( IrpContext,
                            FsRtlIsNtstatusExpected(Status) ? Status : STATUS_INVALID_USER_BUFFER );
        }
    }

    UNREFERENCED_PARAMETER( IrpContext );
}
```

This routine ensures that the physical memory that backs a user's address will not be reused for any other purpose while the operation is ongoing. A file system might do this in order to send the I/O operation to the underlying volume management or disk class layer to satisfy a non-cached user I/O. In such a case, the file system does not need its own virtual address to the buffer. A second function creates the file system's mapping into the kernel address space:

```
PVOID
FatMapUserBuffer (
    IN PIRP_CONTEXT IrpContext,
    IN OUT PIRP Irp
    )
/*++
Routine Description:
    This routine conditionally maps the user buffer for the current I/O
    request in the specified mode. If the buffer is already mapped, it
    just returns its address.

    Note that this is the *input/output* buffer.

Arguments:
    Irp - Pointer to the Irp for the request.

Return Value:
    Mapped address
--*/
{
    UNREFERENCED_PARAMETER( IrpContext );

    //
    // If there is no Mdl, then we must be in  the FSD, and can simply
    // return the UserBuffer field from the Irp.
    //
    if (Irp->MdlAddress == NULL) {
        return Irp->UserBuffer;
    } else {
        PVOID Address = MmGetSystemAddressForMdlSafe( Irp->MdlAddress, NormalPagePriority );
        if (Address == NULL) {
            ExRaiseStatus( STATUS_INSUFFICIENT_RESOURCES );
        }
        return Address;
    }
}
```

The FASTFAT implementation allows the second routine to return the user-level address as well, which requires that the FAT file system ensure that the address returned (user or kernel) must be valid. It does this by using the __try and __except keywords to create a protected block.

These routines are in the deviosup.c source file from the fastfat samples that the WDK contains.

Another critical issue occurs when the request is not satisfied in the context of the caller. If a file system posts the request to a worker thread, the driver must lock down the buffer with an MDL to not lose track of it. The **FatPrePostIrp** function in the workque.c source file from the fastfat samples provides an example of how this issue is handled by the FASTFAT file system.

The FASTFAT file system protects against a broad range of failures, not simply invalid user buffers, by using these routines. While this is a very powerful technique, it also involves ensuring that all protected code blocks properly release any resources they might be holding. The resources to release include memory, synchronization objects, or some other resource of the file system itself. A failure to do so would give a would-be attacker the ability to cause resource starvation by making many repetitive calls into the operating system to exhaust the resource.

File System Control Processing

4/26/2017 • 4 min to read • Edit Online

Handling the **IRP_MJ_FILE_SYSTEM_CONTROL** operation is different from the data buffer handling required by other operations within the file system. This is because each operation establishes its specific data transfer mechanism for the I/O manager as part of its control code by means of the CTL_CODE macro. In addition, the control code specifies the file access that is required by the caller. A file system should be particularly cognizant of this issue when defining the control code, because this access is enforced by the I/O manager. Some I/O control codes (FSCTL_MOVE_FILE , for example) specify FILE_SPECIAL_ACCESS, which is a mechanism for allowing the file system to indicate that the operation's security will be checked by the file system directly. FILE_SPECIAL_ACCESS is numerically equivalent to FILE_ANY_ACCESS, so the I/O manager does not provide any specific security checks, deferring instead to the file system. FILE_SPECIAL_ACCESS mainly provides documentation that additional checks will be made by the file system.

Several file system operations specify FILE_SPECIAL_ACCESS. The FSCTL_MOVE_FILE operation is used as part of the defragmentation interface for file systems and it specifies FILE_SPECIAL_ACCESS. Since you want to be able to defragment open files that are actively being read and written, the handle to be used has only FILE_READ_ATTRIBUTES granted access to avoid share access conflicts. However, this operation needs to be a privileged operation as the disk is being modified on a low level. The solution is to verify that the handle used to issue the FSCTL_MOVE_FILE is a direct-access storage device (DASD) user volume open, which is a privileged handle. The FASTFAT file system code that ensures this operation is being done against a user volume open is in the **FatMoveFile** function (see the fsctrl.c source file from the fastfat sample that the WDK contains):

```
    //
    //  extract and decode the file object and check for type of open
    //

    if (FatDecodeFileObject( IrpSp->FileObject, &Vcb, &FcbOrDcb, &Ccb ) != UserVolumeOpen) {

        FatCompleteRequest( IrpContext, Irp, STATUS_INVALID_PARAMETER );

        DebugTrace(-1, Dbg, "FatMoveFile -> %08lx\n", STATUS_INVALID_PARAMETER);
        return STATUS_INVALID_PARAMETER;
    }
```

The structure used by the FSCTL_MOVE_FILE operation specifies the file being moved:

```
typedef struct {
    HANDLE FileHandle;
    LARGE_INTEGER StartingVcn;
    LARGE_INTEGER StartingLcn;
    ULONG ClusterCount;
} MOVE_FILE_DATA, *PMOVE_FILE_DATA;
```

As previously noted, the handle used to issue the FSCTL_MOVE_FILE is an "open" operation of the entire volume, while the operation actually applies to the file handle specified in the MOVE_FILE_DATA input buffer. This makes the security checks for this operation somewhat complex. For example, this interface must convert the file handle to a file object that represents the file being moved. This requires careful consideration on the part of any driver. FASTFAT does this using **ObReferenceObject** in a guarded fashion in the **FatMoveFile** function in the fsctrl.c source file in the fastfat sample that the WDK contains:

```
    //
    //  Try to get a pointer to the file object from the handle passed in.
    //

    Status = ObReferenceObjectByHandle( InputBuffer->FileHandle,
                                        0,
                                        *IoFileObjectType,
                                        Irp->RequestorMode,
                                        &FileObject,
                                        NULL );

    if (!NT_SUCCESS(Status)) {

        FatCompleteRequest( IrpContext, Irp, Status );

        DebugTrace(-1, Dbg, "FatMoveFile -> %08lx\n", Status);
        return Status;
    }
    //  Complete the following steps to ensure that this is not an invalid attempt
    //
    //    - check that the file object is opened on the same volume as the
    //      DASD handle used to call this routine.
    //
    //    - extract and decode the file object and check for type of open.
    //
    //    - if this is a directory, verify that it&#39;s not the root and that
    //      you are not trying to move the first cluster.  You cannot move the
    //      first cluster because sub-directories have this cluster number
    //      in them and there is no safe way to simultaneously update them
    //      all.
    //
    //  Allow movefile on the root directory if it&#39;s FAT32, since the root dir
    //  is a real chained file.
    //    //
```

Note the use of Irp->RequestorMode to ensure that if the caller is a user-mode application, the handle cannot be a kernel handle. The required access is 0 so that a file can be moved while it is being actively accessed. And finally note that this call must be made in the correct process context if the call originated in user mode. The source code from the FASTFAT file system enforces this as well in the **FatMoveFile** function in fsctrl.c:

```
    //
    //  Force WAIT to true. There is a handle in the input buffer that can only
    //  be referenced within the originating process.
    //

    SetFlag( IrpContext->Flags, IRP_CONTEXT_FLAG_WAIT );
```

These semantic security checks performed by the FAT file system are typical of those required by a file system for any operation that passes a handle. In addition, the FAT file system must also perform sanity checks specific to the operation. These sanity checks are to ensure that the disparate parameters are compatible (the file being moved is on the volume that was opened, for example) in order to prevent the caller from performing a privileged operation when it should not be allowed.

For any file system, correct security is an essential part of file system control operations, which include:

- Validating user handles appropriately.
- Protecting user buffer access.
- Validating semantics of the specific operation.

In many cases, the code necessary to perform proper validation and security can constitute a substantial portion of the code within the given function.

Media Validation

4/26/2017 • 1 min to read • Edit Online

A major concern when developing a file system that supports removable media (FASTFAT, for example) is guarding against the "disk of death" attack. When implementing a file system, the driver must guard against maliciously malformed structures since anyone can insert a removable disk (CD-ROM, DVD-ROM, or USB flash memory disk, for example) into the system.

Security Checks

4/26/2017 • 1 min to read • Edit Online

The bulk of the file system's responsibility with respect to security is in the area of security checks. These are implemented within the file system because it is the part of Windows that actually "owns" the object. The goal of the security implementation is to separate the policy (implemented by the file system) for protecting its objects, and the mechanism (implemented by the Security Reference Monitor) for making access decisions.

In other words, the file system developer is responsible for making calls to the Security Reference Monitor at the appropriate time to validate correct access to a file system resource. The file system need not understand the details of how the Security Reference Monitor makes these security decisions. This section describes points where a file system might consider adding security checks.

This section includes the following topics:

Applying Security Descriptors on the Device Object

4/26/2017 • 5 min to read • Edit Online

Most drivers use the access controls applied by the I/O manager against their device objects to protect themselves from inappropriate access. The simplest approach for most drivers is to apply an explicit security descriptor when the driver is installed. In an INF file, such security descriptors are described by the "Security" entry in the AddReg section. For more information about the complete language used to describe security descriptors, see Security Descriptor Definition Language in the Microsoft Windows SDK documentation.

The basic format of a security descriptor using the Security Descriptor Definition Language (SDDL) includes the following distinct pieces of a standard security descriptor:

- The owner SID.
- The group SID.
- The discretionary access control list (DACL).
- The system access control list (SACL).

Thus, the description within an INF script for a security descriptor is:

```
O:owner-sidG:group-sidD:dacl-flags(ace)(ace)S:sacl-flags(ace)(ace)
```

The individual access control entries describe the access to be granted or denied to a particular group or user as specified by the security identifier or SID. For example, an INF file might contain a line such as:

```
"D:P(A;CI;GR;;;BU)(A;CI;GR;;;PU)(A;CI;GA;;;BA)(A;CI;GA;;;SY)(A;CI;GA;;;NS)(A;CI;GA;;;LS)
(A;CI;CCDCLCSWRPSDRC;;;S-1-5-32-556)"
```

The example above came from a NETTCPIP.INF file from a Microsoft Windows XP Service Pack 1 (SP1) system.

In this instance, there is no owner or group specified, so they default to the predefined or default values. The **D** indicates that this is a DACL. The **P** indicates that this is a protected ACL and does not inherit any rights from the containing object's security descriptor. The protected ACL prevents more lenient security on the parent from being inherited. The parenthetical expression indicates a single access control entry (ACE). An access control entry that uses the SDDL is made up of several distinct semicolon-separated components. In order, they are as follows:

- The type indicator for the ACE. There are four unique types for DACLs and four different types for SACLs.
- The **flags** field used to describe inheritance of this ACE for child objects or auditing and alarm policy for SACLs.
- The **rights** field indicates which rights are granted or denied by the ACE. This field can either specify a specific numeric value indicating the generic, standard, and specific rights applicable to this ACE or use a string description of common access rights.
- An object GUID if the DACL is an object-specific ACE structure.
- An inherited object GUID if the DACL is an object-specific ACE structure
- A SID indicating the security entity to which this ACE applies.

Thus, interpreting the sample security descriptor, the "A" leading up the ACE indicates that this is an "access allowed" entry. The alternative is an "access denied" entry, which is only infrequently used and is denoted by a leading "D" character. The **flags** field specifies Container Inherit (CI), which indicates that this ACE is inherited by sub-objects.

The **rights** field values encode specific rights that include generic rights and standard rights. For example, "GR" indicates "generic read" access and "GA" indicates "generic all" access, both of which are generic rights. A number of special rights follow these generic rights. In the sample above, "CC" indicates create child access, which is specific to file and directory rights. The sample above also includes other standard rights after the "CC" string including "DC" for delete child access, "LC" for list child access, "SW" for self-right access, "RP" for read property access, "SD" for standard delete access, and "RC" for read control access.

The SID entry strings in the sample above include "PU" for power users, "BU" for built-in users, "BA" for "built-in administrators, "LS" for the local service account, "SY" for the local system, and "NS" for the network service account. So in the example above, users are given generic read access on the object. In contrast, built-in administrators, the local service account, the local system, and the network service are given generic all access (read, write, and execute). The complete set of all possible rights and standard SID strings are documented in the Windows SDK.

These ACLs will be applied to all device objects created by a given driver. A driver can also control the security settings of specific objects by using the new function, **IoCreateDeviceSecure**, when creating a named device object. The **IoCreateDeviceSecure** function is available on Windows XP Service Pack 1 and Windows Server 2003 and later. Using **IoCreateDeviceSecure**, the security descriptor to be applied to the device object is described using a subset of the full Security Descriptor Definition Language that is appropriate to device objects.

The purpose of applying specific security descriptors to device objects is to ensure that appropriate security checks are done against the device whenever an application attempts to access the device itself. For device objects that contain a name structure (the namespace of a file system, for example), the details of managing access to this device namespace belong to the driver, not to the I/O manager.

An interesting issue in these cases is how to handle security at the boundary between the I/O manager responsible for checking access to the driver device object and the device driver, which implements whatever security policy is appropriate for the driver. Traditionally, if the object being opened is the name of the device itself, the I/O manager will perform a full access check against the device object directly using its security descriptor. However, if the object being opened indicates a path inside the driver itself, the I/O manager will only check to ensure that traverse access is granted to the device object. Typically, this traverse right is granted because most threads have been granted **SeChangeNotifyPrivilege**, which corresponds with granting the traverse right to the directory. A device that does not support name structure would normally request that the I/O manager perform a full security check. This is done by setting the **FILE_DEVICE_SECURE_OPEN** bit in the device characteristics field. A driver that includes a mix of such device objects should set this characteristic for those devices that do not support name structure. For example, a file system would set this option on its named device object (which does not support a naming structure), but would not set this option on its unnamed device objects (a volume, for example), which do support naming structure. Failing to set this bit correctly is a common bug in drivers and can allow inappropriate access to the device. For drivers that use the attachment interface (**IoAttachDeviceToDeviceStackSafe**, for example), the **FILE_DEVICE_SECURE_OPEN** bit is set if this field is set in the device to which the driver is attaching. So, filter drivers don't need to worry about this particular aspect of security checking.

IRP_MJ_CREATE Dispatch Routine

7/21/2017 • 1 min to read • Edit Online

A major portion of Windows security checking occurs inside the **IRP_MJ_CREATE** dispatch routine. This is because the bulk of the Windows security model is related to access validation. Access validation results are stored as part of the handle that is created as a result of this operation. Subsequent operations are validated against the rights computed at this point.

If the access rights on the file change after the file or directory has been opened, the original access rights provided during the IRP_MJ_CREATE operation continue to be valid. These access rights are associated with the handle, so as long as the handle persists, the access granted under it governs subsequent operations.

This section includes the following topics:

Checking for Traverse Privilege on IRP_MJ_CREATE

7/21/2017 • 2 min to read • Edit Online

One of the primary concerns **IRP_MJ_CREATE** checks is whether the caller has traverse privilege (does the caller have the right to access the path to the object). Since most callers have traverse privilege, one of the first checks normally done within the file system is checking for the traverse privilege:

```
BOOLEAN traverseCheck =
    !(IrpContext->IrpSp->Parameters.Create.SecurityContext->AccessState->Flags
        & TOKEN_HAS_TRAVERSE_PRIVILEGE);
```

Note that the traverse privilege check relies upon the state information passed to the file system from the I/O manager. This information is based upon whether the caller holds SeChangeNotifyPrivilege. If the caller does not hold this privilege, a traverse check must be done on each directory. In the example below, the traverse check is done using a generic routine, typically used for most security checks:

{

```
    SeLockSubjectContext(
        &accessParams.AccessState->SubjectSecurityContext);
//
// Note: AccessParams is passed to us and is normally based on
//       the fields of the same name from the IRP
//

    granted = SeAccessCheck( Fcb->SecurityDescriptor,
        &AccessParams.AccessState->SubjectSecurityContext,
        TRUE,
        AccessParams.desiredAccess,
        0,
        &Privileges,
        IoGetFileObjectGenericMapping(),
        AccessParams.AccessMode,
        &AccessParams.GrantedAccess,
        &AccessParams.status );

    if (Privileges != NULL) {
        //
        // this changes the AccessState
        //
        (void) SeAppendPrivileges(AccessParams.AccessState, Privileges );
        SeFreePrivileges( Privileges );
        Privileges = NULL;
    }

    if (granted) {
        //
        // delete granted bits from desired bits
        //
        AccessParams.desiredAccess &=
            ~(AccessParams.GrantedAccess | MAXIMUM_ALLOWED);

        if (!checkOnly) {
        //
        // the caller wants to modify the access state for this
        // request
        //
            AccessParams.AccessState->PreviouslyGrantedAccess |=
                AccessParams.GrantedAccess;

        if (maxDesired) {

            maxDelete =
                (BOOLEAN)(AccessParams.AccessState->PreviouslyGrantedAccess &
                    DELETE);
            maxReadAttr =
                (BOOLEAN)(AccessParams.AccessState->PreviouslyGrantedAccess &
                    FILE_READ_ATTRIBUTES);
        }
        AccessParams.AccessState->RemainingDesiredAccess &=
            ~(AaccessParams.GrantedAccess | MAXIMUM_ALLOWED);
    }
    SeUnlockSubjectContext(&accessParams.AccessState->SubjectSecurityContext);
}
```

This function performs a generic security check. This function must deal with the following issues in doing so:

- It must specify the correct security descriptor to use for the check.
- It must pass along the security context (these are the credentials of the entity performing the operation).
- It must update the access state based upon the results of the security check.
- It must account for the MAXIMUM_ALLOWED option (see the ntifs.h include file that the WDK includes for

details). The MAXIMUM_ALLOWED option specifies that the file system should set the access to the maximum possible access allowed by the file system (read/write/delete, for example). Very few applications use the MAXIMUM_ALLOWED option because this option is not supported on the FASTFAT file system. Because the MAXIMUM_ALLOWED option bit is not one of the access bits that the FASTFAT file system recognizes, it rejects access requests to the given file. An application that attempts to open a file on a FASTFAT volume with the MAXIMUM_ALLOWED option set will find that the request fails. For details, see the **FatCheckFileAccess** function in the Acchksup.c source file of the FASTFAT sample code that the WDK contains.

Note that for a simple traverse check, the requested access would be FILE_TRAVERSE and the security descriptor would be that of the directory through which the caller is attempting to traverse, not the requested access from the original IRP_MJ_CREATE IRP.

Checking for Other Special Cases on IRP_MJ_CREATE

7/21/2017 • 2 min to read • Edit Online

Other special case handling must occur during the IRP_MJ_CREATE handling within the file system if the caller does not have SeChangeNotifyPrivilege. For example, a file that can be opened by using its file ID or object ID may not allow the caller to obtain the path to that file, since the caller might not have traverse permission to get to the object by means of the directory tree structure.

The cases you may wish to consider in your file system:

- **FILE_OPEN_BY_FILE_ID** — the name of the file should not be made available to the caller. Since this information (traverse permission) is only known during the create operation, and will not be available during a file information query call, information on traverse permission must be computed by the file system during the IRP_MJ_CREATE.
- **SL_OPEN_TARGET_DIRECTORY** — the target file might not exist and the directory must be checked for file creation access. If the target does exist, it might require a delete access check. Normally the delete access check is done at the time of the rename operation during IRP_MJ_SET_INFORMATION processing.
- **FILE_SUPERSEDE** and **FILE_OVERWRITE** — destructive operations require additional access rights, even if the caller did not explicitly request them. For example, a file system might require DELETE access in order to perform a supersede operation.
- **FILE_CREATE**, **FILE_OPEN_IF**, and **FILE_OVERWRITE_IF** — constructive operations require access to the parent directory where the new object is being created. This is a check on the containing directory, not on the object itself, and thus similar to the earlier code sample.
- **SL_FORCE_ACCESS_CHECK** — if this value is set in the I/O stack location, the check must be performed as if the call were from user mode, not kernel mode. Thus, calls to security monitor routines should specify **UserMode** even if the call originated within a kernel mode server.
- *File/directory deletion* — this might be based upon the ACL state of the file (for example, FILE_WRITE_DATA access implicitly allows delete; if you can delete the data, you have effective delete permissions on the file) as well as the ACL state of the directory (FILE_DELETE_CHILD permission on the containing directory, for example).

The following code example demonstrates the special case handling for FILE_SUPERSEDE and FILE_OVERWRITE, both cases where additional access is implicitly required by the caller, even if it was not requested.

```
{
ULONG NewAccess = Supersede ? DELETE : FILE_WRITE_DATA;
ACCESS_MASK AddedAccess = 0;
PACCESS_MASK DesiredAccess =
    &IrpSp->Paramters.Create.SecurityContext->DesiredAccess;

//
// If the caller does not have restore privilege, they must have write
// access to the EA and attributes for overwrite or supersede.
//
if (0 == (AccessState->Flags & TOKEN_HAS_RESTORE_PRIVILEGE)) {
    *DesiredAccess |= FILE_WRITE_EA | FILE_WRITE_ATTRIBUTES;

    //
    // Does the caller already have this access?
    //
    if (AccessState->PreviouslyGrantedAccess & NewAccess) {

        //
        // No - they need this as well
        //
        *DesiredAccess |= NewAccess;

    }

    //
    // Now check access using SeAccessCheck (omitted)
    //

}
```

This code sample is a good example of where file system policy takes precedence. The caller did not request DELETE access or FILE_WRITE_DATA access, but such access is inherent in the operation being performed based upon the semantics of the file system.

Adding Auditing on IRP_MJ_CREATE

7/21/2017 • 1 min to read • Edit Online

Another important aspect of the security checks within a file system is to add auditing (if necessary). Typically, this is done as part of the same set of routines that make security decisions, since the purpose of auditing is to record the security decisions made by the system. For example, the following code could be used to implement auditing within a file system after completing the access checks:

```
{
UNICODE_STRING FileAuditObjectName;

RtlInitUnicodeString(&FileAuditObjectName, L"File");

if ( SeAuditingFileOrGlobalEvents (AccessGranted,
        &Fcb->SecurityDescriptor,
        &AccessState->SubjectSecurityContext)) {
    //
    // Must pass complete Windows path name, including device name.
    //
    ConstructAuditFileName(Irp, Fcb, &AuditName);

    if (IrpSp->Parameters.Create.SecurityContext->FullCreateOptions
            & FILE_DELETE_ON_CLOSE) {
        SeOpenObjectForDeleteAuditAlarm(&FileAuditObjectName,
                                        NULL,
                                        &AuditName,
                                        &Fcb->SecurityDescriptor,
                                        AccessState,
                                        FALSE, // Object not created.
                                        // Was it  successful?
                                        // Based on SeAccessCheck
                                        SeAccessCheckAccessGranted,
                                        // UserMode or KernelMode
                                        EffectiveMode,
                                        &AccessState->GenerateOnClose
                                        );
    } else {
        SeOpenObjectAuditAlarm(&FileAuditObjectName,
                               NULL,
                               &AuditName,
                               &Fcb->SecurityDescriptor,
                               AccessState,
                               FALSE, // object not created
                               // Was it successful?
                               // Based on SeAccessCheck
                               AccessGranted,
                               // UserMode or KernelMode
                               EffectiveMode,
                               &AccessState->GenerateOnClose
                               );
    }

    //
    // Free file name here if needed.
    //
}
```

Management of Access Control Lists on IRP_MJ_CREATE

7/21/2017 • 1 min to read • Edit Online

There are numerous additional security-related issues that can be addressed within the file system. For example, the management of access control lists on disk is a major security issue. Given that security information might be identical on thousands of files, it is often useful for the file system to implement a sharing model for security descriptors. Thus, all files that use the same security descriptor share a single on-disk (and possibly in-memory) copy of the security descriptor. The NTFS file system uses this model.

An additional option would be for the file system to cache results. While not strictly related to security, it is important to realize that security operations can add substantial cost to ordinary operations, such as opening the file. Thus, caching security results from previous operations can allow the file system to rely upon previous decisions. For example, a new call that requests a subset of access previously granted to the same user on the same file could be summarily granted. Of course, the risk of adding any such mechanism is the potential for adding bugs, which allow improper access. It is important to ensure that any security implementation be thoroughly tested to ensure that it works in the manner expected.

Assigning Security to a New File on IRP_MJ_CREATE

7/21/2017 • 1 min to read • Edit Online

The final task in create handling is assigning security to the new file. While the Windows security model supports inheritance (individual ACE entries are marked in such a way that they are inherited when new files or directories are created) this is implemented outside the file system. Thus, the bulk of the logic within the file system is dedicated to storing the new security descriptor. Here is a sample routine:

```
NTSTATUS FsdAssignInitialSecurity( PIRP_CONTEXT IrpContext,
        PFCB Fcb, PFCB Directory)
{
    NTSTATUS status = STATUS_SUCCESS;
    BOOLEAN CreateDir = ((IrpContext->IrpSp->Parameters.Create.Options
        & FILE_DIRECTORY_FILE)==FILE_DIRECTORY_FILE);
    PACCESS_STATE AccessState =
    IrpContext->IrpSp->Parameters.Create.SecurityContext->AccessState;
    PSECURITY_DESCRIPTOR SecurityDescriptor = NULL;

    //
    // Make sure the parent directory&#39;s security descriptor is loaded.
    //
    (void) FsdLoadSecurityDescriptor(IrpContext, Directory);

    //
    // don&#39;t care about the return code here, as it is handled later
    //
    if (Directory->SecurityDescriptor == NULL) {

        //
        // If the parent has no security, then we are outside
        // of the normal Windows paradigm.
        //
        // The child (that is, the target of the create) will also have
        // a NULL SD.
        //
        // Note that you can always assign security to the file object
        // explicitly at later on.
        //
        return STATUS_SUCCESS;

    }

    //
    // Now create the security descriptor.
    //
    status = SeAssignSecurity(Directory->SecurityDescriptor,
                              AccessState->SecurityDescriptor,
                              &SecurityDescriptor,
                              CreateDir,
                              &AccessState->SubjectSecurityContext,
                              IoGetFileObjectGenericMapping(),
                              PagedPool);

    if (!NT_SUCCESS(status)) {

        return status;
    }

    //
    // Associate the SD with the file; use our own storage so when
    // cleanup occurs it is unnecessary to know if the storage came from the
    // security reference monitor.
     //
    Fcb->SecurityDescriptorLength =
        RtlLengthSecurityDescriptor( SecurityDescriptor );

    Fcb->SecurityDescriptor = ExAllocatePoolWithTag(PagedPool,
        Fcb->SecurityDescriptorLength, &#39;DSyM&#39;);

    if (!Fcb->SecurityDescriptor) {
        //
        // There is no paged pool.
        //
        SeDeassignSecurity(&SecurityDescriptor);
        Fcb->SecurityDescriptorLength = 0;
        return STATUS_NO_MEMORY;
    }

    RtlCopyMemory(Fcb->SecurityDescriptor, SecurityDescriptor,
        Fcb->SecurityDescriptorLength);

    SeDeassignSecurity(&SecurityDescriptor);

    //
    // Store the SD persistently (this is file system specific).
    //
    (void) FsdStoreSecurityDescriptor(IrpContext, Fcb);

    return STATUS_SUCCESS;
}
```

Note that the logic of constructing the initial security descriptor (understanding inheritance, for example) is not handled within the file system. This is in keeping with the simple model for handling security descriptors within the file systems layer.

Handling Quotas on IRP_MJ_CREATE

7/21/2017 • 1 min to read • Edit Online

Some logic could also be included to acquire quota information if the file system supports quotas. One strategy a file system could adopt would be to acquire a block of quota information about **IRP_MJ_CREATE** that is later checked and updated by dispatch routines for other IRP requests that can change the size of a file (delete and write operations, for example).

IRP_MJ_QUERY_SECURITY and IRP_MJ_SET_SECURITY

7/21/2017 • 3 min to read • Edit Online

Fortunately for a file system, the actual storage and retrieval of security descriptors is relatively opaque. This is due to the nature of security descriptors in a self-relative format that does not require any understanding of the descriptor by the file system. Thus, processing a query operation is normally a very simple exercise. Here is an example from a file system implementation:

```
NTSTATUS FsdCommonQuerySecurity( PIRP_CONTEXT IrpContext)
{
    NTSTATUS status = STATUS_SUCCESS;
    PSECURITY_DESCRIPTOR LocalPointer;

    // Need to add code to lock the FCB here

    status = FsdLoadSecurityDescriptor(IrpContext, IrpContext->Fcb);

    if (NT_SUCCESS(status) ) {

        //
        // copy the SecurityDescriptor into the callers buffer
        // note that this copy can throw an exception that must be handled
        // (code to handle the exception was omitted here for brevity)
        //
        LocalPointer = IrpContext->Fcb->SecurityDescriptor;

        status = SeQuerySecurityDescriptorInfo(
     &IrpContext->IrpSp->Parameters.QuerySecurity.SecurityInformation,
            (PSECURITY_DESCRIPTOR)IrpContext->Irp->UserBuffer,
            &IrpContext->IrpSp->Parameters.QuerySecurity.Length,
            &LocalPointer );

        //
        // CACLS utility expects OVERFLOW
        //
        if (status == STATUS_BUFFER_TOO_SMALL ) {
            status = STATUS_BUFFER_OVERFLOW;
        }
    }

    // Need to add code to unlock the FCB here

    return status;
}
```

Note that this routine relies on an external function to load the actual security descriptor from persistent storage (in this implementation, that routine only loads the security descriptor if it has not previously been loaded). Since the security descriptor is opaque to the file system, the security reference monitor must be used to copy the descriptor into the user's buffer. We note two points with respect to this code sample:

1. The conversion of the error code STATUS_BUFFER_TOO_SMALL into the warning code STATUS_BUFFER_OVERFLOW is necessary in order to provide correct behavior for some Windows security tools.

2. Errors in handling the user buffer can, and will, arise because both query and set security operations are

normally done using the user buffer directly. Note that this is controlled by the **Flags** member of the DEVICE_OBJECT created by the file system. In a file system implementation based on this code, the calling function would need to use a __try block to protect against an invalid user buffer.

The specifics of how the file system loads a security descriptor from storage (the **FsdLoadSecurityDescriptor** function in this example) will depend entirely on the implementation of security descriptor storage in the file system.

Storing a security descriptor is a bit more involved. File systems may need to determine whether the security descriptor matches an existing security descriptor if the file system supports security descriptor sharing. For non-matching security descriptors, the file system may need to allocate new storage for this new security descriptor. Below is a sample routine for replacing the security descriptor on a file.

```
NTSTATUS FsdCommonSetSecurity(PIRP_CONTEXT IrpContext)
{
    NTSTATUS status = STATUS_SUCCESS;
    PSECURITY_DESCRIPTOR SavedDescriptorPtr =
        IrpContext->Fcb->SecurityDescriptor;
    ULONG SavedDescriptorLength =
        IrpContext->Fcb->SecurityDescriptorLength;
    PSECURITY_DESCRIPTOR newSD = NULL;
    POW_FCB Fcb = IrpContext->Fcb;
    ULONG Information = IrpContext->Irp->IoStatus.Information;

    //
    // make sure that the FCB security descriptor is up to date
    //
    status = FsdLoadSecurityDescriptor(IrpContext, Fcb);

    if (!NT_SUCCESS(status)) {
      //
      // Something is seriously wrong
      //
      IrpContext->Irp->IoStatus.Status = status;
      IrpContext->Irp->IoStatus.Information = 0;
      return status;
    }

    status = SeSetSecurityDescriptorInfo(
       NULL,
       &IrpContext->IrpSp->Parameters.SetSecurity.SecurityInformation,
       IrpContext->IrpSp->Parameters.SetSecurity.SecurityDescriptor,
       &Fcb->SecurityDescriptor,
       PagedPool,
       IoGetFileObjectGenericMapping()
       );

    if (!NT_SUCCESS(status)) {

        //
        // restore things  and return
        //
        Fcb->SecurityDescriptorLength = SavedDescriptorLength;
        Fcb->SecurityDescriptor = SavedDescriptorPtr;
        IrpContext->Irp->IoStatus.Status = status;
        IrpContext->Irp->IoStatus.Information = 0;

        return status;
    }

    //
    // get the new length
    //
    Fcb->SecurityDescriptorLength =
        RtlLengthSecurityDescriptor(Fcb->SecurityDescriptor);
    //
    // allocate our own private SD to replace the one from
    // SeSetSecurityDescriptorInfo so we can track our memory usage
    //
    newSD = ExAllocatePoolWithTag(PagedPool,
        Fcb->SecurityDescriptorLength, &#39;DSyM&#39;);

    if (!newSD) {

      //
      // paged pool is empty
      //
      SeDeassignSecurity(&Fcb->SecurityDescriptor);
      status = STATUS_NO_MEMORY;
      Fcb->SecurityDescriptorLength = SavedDescriptorLength;
      Fcb->SecurityDescriptor = SavedDescriptorPtr;

      //
      // make sure FCB security is in a valid state
      //
      IrpContext->Irp->IoStatus.Status = status;
      IrpContext->Irp->IoStatus.Information = 0;

      return status;

    }

    //
    // store the new security on disk
    //
    status = FsdStoreSecurityDescriptor(IrpContext, Fcb);

    if (!NT_SUCCESS(status)) {
      //
      // great- modified the in-core SD but couldn&#39;t get it out
      // to disk. undo everything.
      //
      ExFreePool(newSD);
      SeDeassignSecurity(&Fcb->SecurityDescriptor);
      status = STATUS_NO_MEMORY;
      Fcb->SecurityDescriptorLength = SavedDescriptorLength;
      Fcb->SecurityDescriptor = SavedDescriptorPtr;
      IrpContext->Irp->IoStatus.Status = status;
      IrpContext->Irp->IoStatus.Information = 0;

      return status;
    }

    //
    // if we get here everything worked!
    //
    RtlCopyMemory(newSD, Fcb->SecurityDescriptor,
        Fcb->SecurityDescriptorLength);

    //
    // deallocate the security descriptor
    //
    SeDeassignSecurity(&Fcb->SecurityDescriptor);

    //
    // this either is the new private SD or NULL if
    // memory allocation failed
    //
    Fcb->SecurityDescriptor = newSD;

    //
    // free the memory from the previous descriptor
    //
    if (SavedDescriptorPtr) {
      //
      // this  must always be from private allocation
      //
      ExFreePool(SavedDescriptorPtr);

    }

    IrpContext->Irp.IoStatus = status;
    IrpContext->Irp.Information = Information;

    return status;
}
```

Note that this is an area in which implementation varies dramatically from file system to file system. For example, a file system that supports security descriptor sharing would need to add explicit logic to find a matching security descriptor. This sample is only an attempt to provide guidance to implementers.

IRP_MJ_DIRECTORY_CONTROL

7/21/2017 • 3 min to read • Edit Online

Security is a consideration when processing certain directory control operations, notably those dealing with change notifications. The security concern is that a directory change notification might return information about specific files that have changed. If the user does not have the privilege to traverse the path to the directory, information about the change cannot be returned to the user. Otherwise, the user now has a mechanism for learning additional information about the directory that the user should not have.

Support for directory change notification by the file system run-time library allows a file system to specify a callback function for performing a traverse check before it returns a directory change notification. This callback function takes a large number of parameters. For security considerations, the following three parameters are important:

- *NotifyContext*--the context of the directory where the change notification is active. This will be the *FsContext* parameter that is passed in to the call to **FsRtlNotifyFilterChangeDirectory**. Note that **FsRtlNotifyFilterChangeDirectory** is available on Windows XP and later. Windows 2000 systems used the **FsRtlNotifyFullChangeDirectory** function, which is similar.
- *TargetContext*--the context of the file that has changed. This will be the *TargetContext* parameter passed by the file system when it calls **FsRtlNotifyFilterReportChange**.
- *SubjectContext*--the security context of the thread requesting the directory change notification. This is the subject security context captured by the file system at the time the directory change notification call is made to **FsRtlNotifyFilterChangeDirectory**.

When a change occurs, the file system indicates this to the file system run-time library. The file system run-time library will then call the callback function provided by the file system to verify that the caller can be given information about the change. Note that the file system only needs to register a callback function if the check is required for the caller. This is the case if the caller does not have SeChangeNotifyPrivilege enabled, as indicated by the TOKEN_HAS_TRAVERSE_PRIVILEGE in the caller's security token.

Inside the callback function, the file system must perform a traverse check from the directory specified by the *NotifyContext* parameter, to the file that changed, specified by the *TargetContext* parameter. The sample routine below performs such a check.

```
BOOLEAN
FsdNotifyTraverseCheck (
    IN PDIRECTORY_CONTEXT OriginalDirectoryContext,
    IN PFILE_CONTEXT ModifiedDirectoryContext,
    IN PSECURITY_SUBJECT_CONTEXT SubjectContext
    )
{
  BOOLEAN AccessGranted = TRUE;
  PFILE_CONTEXT CurrentDirectoryContext;
  ACCESS_MASK GrantedAccess;
  NTSTATUS Status;
  PPRIVILEGE_SET Privileges = NULL;
  PFILE_CONTEXT TopDirectory;


  //
  //  Nothing to do  if there is no file context.
  //
  if (ModifiedDirectoryContext == NULL) {
    return TRUE;
  }

  //
  // If the directory that changed is the original directory,
  // we can return , since the caller has access.
  // Note that the directory  context is unique to the specific
  // open instance, while the modified directory context
  // represents the per-file/directory context.
  // How these data structures work in your file system will vary.
  //
  if (OriginalDirectoryContext->FileContext == ModifiedDirectoryContext) {
    return TRUE;
  }

  //
  // Lock the subject context.
  //
  SeLockSubjectContext(SubjectContext);


  for( TopDirectory = OriginalDirectoryContext->FileContext,
          CurrentDirectoryContext = ModifiedDirectoryContext;
          CurrentDirectoryContext == TopDirectory || !AccessGranted;
          CurrentDirectoryContext = CurrentDirectoryContext->ParentDirectory) {
    //
    // Ensure we have the current security descriptor loaded for
    // this directory.
    //
    FsdLoadSecurity( NULL, CurrentDirectoryContext);

    //
    // Perform traverse check.
    //
    AccessGranted = SeAccessCheck(
            CurrentDirectoryContext->SecurityDescriptor,
            SubjectContext,
            TRUE,
            FILE_TRAVERSE,
            0,
            &Privileges,
            IoGetFileObjectGenericMapping(),
            UserMode,
            &GrantedAccess,
            &Status);

    //
    // At this point, exit the loop if access was not granted,
    // or if the parent directory is the same as where the change
    // notification was made.
    //

  }

  //
  // Unlock subject context.
  //
  SeUnlockSubjectContext(SubjectContext);

  return AccessGranted;
}
```

This routine is likely to be substantially different for file systems that cache security information, or that have different data structures for tracking files and directories (for example, files that use a structure for tracking links between files and directories). File systems supporting links are not considered in this sample in an attempt to simplify the example.

IRP_MJ_FILE_SYSTEM_CONTROL

4/26/2017 • 1 min to read • Edit Online

A file system control allows the file system to perform essentially any specialized operation. The existing Windows file systems have a number of specialized controls and for them, as well as any third-party file systems developed for Windows, it is imperative to aggressively check all parameters. In addition, FSCTL operations often have restricted security rights. These can be seen in the FASTFAT sample code that the WDK contains (see the **FatInvalidateVolumes** function in fsctrl.c, for example). This is an example of a privilege check. The policy of the FASTFAT file system in this case is to require that the given privilege be enabled on the system.

The I/O manager will enforce FILE_READ_DATA and FILE_WRITE_DATA permissions on specific FSCTL operations, if the file system has set these bits in the file system operation definition using the CTL_CODE macro. All other permissions required must be checked by the file system (FILE_READ_ATTRIBUTES permissions, for example) if this is the policy of the file system.

IRP_MJ_SET_INFORMATION

7/21/2017 • 1 min to read • Edit Online

The rename and hard link cases in set information might require a security check under certain circumstances. Specifically, if the caller wants to delete the target of the rename or hard link by setting the **ReplaceIfExists** field to **TRUE**, the file system must perform a security check to ensure that the caller has appropriate permission to delete the target. In addition, there can be certain types of files that the file system, as a matter of policy, does not wish to allow to be deleted in this fashion (registry hives and paging files, for example). The following code example determines if the caller has the appropriate security permissions to delete the file:

```
NTSTATUS FsdCheckDeleteFileAccess(POW_IRP_CONTEXT IrpContext,
                                  PSECURITY_DESCRIPTOR targetSD,
                                  PFCB ParentFcb)
{
    SECURITY_SUBJECT_CONTEXT SubjectContext;
    BOOLEAN Granted;
    NTSTATUS status = STATUS_SUCCESS;
    PPRIVILEGE_SET Privileges = NULL;
    ACCESS_MASK GrantedAccess;

    //
    // See if the user has DELETE access to the target.
    //
    SeCaptureSubjectContext( &SubjectContext );

    SeLockSubjectContext( &SubjectContext );

    Granted = SeAccessCheck(targetSD,           // Target&#39;s SD.
                            &SubjectContext,    // Captured security context.
                            TRUE,               // Tokens are locked.
                            DELETE,             // we only care about delete
                            0,                  // previously granted access.
                            &Privileges,        // privilege_set
                            IoGetFileObjectGenericMapping(), // Generic mappings.
                            UserMode,           // Mode
                            &GrantedAccess,     // Granted access mask
                            &status );          // Error code

    //
    // Do not need privilege set, so release it.
    //
    if (Privileges != NULL) {

        SeFreePrivileges( Privileges );
        Privileges = NULL;
    }

    if (!Granted) {

        status = STATUS_SUCCESS;

        //
        // The user does not have DELETE access to the target, but
        // could have FILE_DELETE_CHILD access to the parent directory.
        //
        (void) FsdLoadSecurityDescriptor(IrpContext, ParentFcb);
        if (!ParentFcb->SecurityDescriptor) {
            //
            // fine - no security is fine - he gets to do what he wants
            //
            SeUnlockSubjectContext( &SubjectContext );
            SeUnlockSubjectContext( &SubjectContext );
            SeReleaseSubjectContext( &SubjectContext );
            return STATUS_SUCCESS;
        }


        Granted = SeAccessCheck(&ParentFcb->SecurityDescriptor,
                                &SubjectContext,   // Captured security context.
                                TRUE,              // Tokens are locked.
                                FILE_DELETE_CHILD, // we only care about delete
                                0,                 // Previously granted access.
                                &Privileges,       // privilege_set
                                IoGetFileObjectGenericMapping(), // Generic mappings
                                UserMode,          // mode
                                &GrantedAccess,    // Granted access mask
                                &status );         // Error code
        //
        // Release privileges
        //
        if (Privileges != NULL) {
            SeFreePrivileges( Privileges );
            Privileges = NULL;
        }
    }
    SeUnlockSubjectContext( &SubjectContext );
    SeReleaseSubjectContext( &SubjectContext );
    return status;
}
```

This code can be used for both the rename and hard link creation case.

Note that it is outside the scope of this document to discuss policy level code where the file system decides to disallow the delete based upon the type of file being deleted.

Impersonation

4/26/2017 • 1 min to read • Edit Online

Some file systems might find it useful to perform operations on behalf of the original caller. For example, a network file system might need to capture the caller's security information at the time a file is opened so that a subsequent operation can be performed using the appropriate credentials. No doubt there are numerous other special cases where this type of feature is useful, both within a file system as well as in specific applications.

The key routines needed for impersonation include:

- **PsImpersonateClient SeImpersonateClientEx**--initiates impersonation. Unless a specific thread is indicated, the impersonation is done in the current thread context.
- **PsRevertToSelf**--terminates impersonation within the current thread context.
- **PsReferencePrimaryToken**--holds a reference on the primary (process) token for the specified process. This function may be used to capture the token for any process on the system.
- **PsDereferencePrimaryToken**--releases a reference on a previously referenced primary token.
- **SeCreateClientSecurityFromSubjectContext**--returns a client security context useful for impersonation from a subject context (provided to the FSD during the **IRP_MJ_CREATE** handling, for example).
- **SeCreateClientSecurity**--creates a client security context based upon the security credentials of an existing thread on the system.
- **ImpersonateSecurityContext**--impersonates security context within ksecdd.sys, the kernel security service.
- **RevertSecurityContext**--terminates impersonation within ksecdd.sys, the kernel security service.

Impersonation is straight-forward to implement. The following code example demonstrates basic impersonation:

```
NTSTATUS PerformSpecialTask(IN PFSD_CONTEXT Context)
{
  BOOLEAN CopyOnOpen;
  BOOLEAN EffectiveOnly;
  SECURITY_IMPERSONATION_LEVEL ImpersonationLevel;
  NTSTATUS Status;
  PACCESS_TOKEN oldToken;

  //
  // We need to perform a task in the system process context
  //
  if (NULL == Context->SystemProcess) {

    return STATUS_NO_TOKEN;

  }

  //
  // Save the existing token, if any (otherwise NULL)
  //
  oldToken = PsReferenceImpersonationToken(PsGetCurrentThread(),
                                           &CopyOnOpen,
                                           &EffectiveOnly,
                                           &ImpersonationLevel);

  Status = PsImpersonateClient( PsGetCurrentThread(),
                                Context->SystemProcess,
                                TRUE,
                                TRUE,
                                SecurityImpersonation);
  if (!NT_SUCCESS(Status)) {

    if (oldToken)
        PsDereferenceImpersonationToken(oldToken);
    return Status;

  }

  //
  // Perform task - whatever it is
  //


  //
  // Restore to previous impersonation level
  //
  if (oldToken) {
    Status = PsImpersonateClient(PsGetCurrentThread(),
                                 oldToken,
                                 CopyOnOpen,
                                 EffectiveOnly,
                                 ImpersonationLevel);

    if (!NT_SUCCESS(Status)) {
      //
      // This is bad - we can&#39;t restore, we can&#39;t leave it this way
      //
      PsRevertToSelf();
    }
    PsDereferenceImpersonationToken(oldToken);
  } else {
    PsRevertToSelf();
  }

  return Status;
}
```

There are numerous variants of this impersonation code that are available to file systems developers, but this provides a basic illustration of the technique.

Process and Thread Termination Issues

4/26/2017 • 1 min to read • Edit Online

File systems that store state information related to specific users might need to watch for process and thread termination conditions. For example, encryption keys associated with a particular user might need to be discarded on the termination (whether planned or premature) of a specialized control application. For more information about the routines used to handle these conditions, see **PsSetCreateProcessNotifyRoutine** and **PsSetCreateThreadNotifyRoutine**.

Security Considerations for File System Filter Drivers

4/26/2017 • 1 min to read • Edit Online

File system filter drivers have a distinct set of security issues in addition to those that apply to all drivers and to file system specific issues. This section discusses key issues that need to be addressed by file system filter driver writers, although it does not provide a complete list of all possible security issues.

This section includes the following topics:

Proxy Operations in File System Filter Drivers

4/26/2017 • 1 min to read • Edit Online

File system filter drivers must frequently perform operations on behalf of the original (user-mode) caller. When doing so, it is imperative that the file system filter driver not perform an operation that might take an action that the original user could not. For example, suppose that a user attempts to open a file for FILE_SUPERSEDE. A filter driver cannot then attempt to open the file specifying the same type of access using **ZwCreateFile**, for example, because even if the user did not have permission to supersede the file, the operation would succeed when performed by the file system filter driver.

The ways in which a file system filter driver might introduce such problems are many. They can occur any time a file system filter driver performs an operation on the underlying file system without knowing the results of the user's operation. A file system filter driver must thus identify such cases and ensure that it has either determined the outcome of the operation, or it has a mechanism for recovering from an error in the actual user operation. For example, in the case of a request to supersede the file, a filter driver might need to open the file on behalf of the original caller using **IoCreateFile** indicating security rights on the file that would be sufficient for the supersede operation. If the filter driver were to use the IO_FORCE_ACCESS_CHECK option, for example, a security check would be done with the credentials of the current thread, even though the call was from a kernel driver.

It is essential for a file system filter driver to identify instances where the driver is performing an operation on behalf of, or as a result of, some user level operation. In these cases, a clear strategy for how to ensure correct operation needs to be identified.

Impersonation in a File System Filter Driver

4/26/2017 • 1 min to read • Edit Online

Another operation a file system filter driver might attempt to use is impersonation. While impersonation is a very powerful technique for handling security on behalf of other threads, it also requires appropriate care for use on behalf of any component. For a file system filter driver, it is important to identify the operations that need to be done using impersonation. Then, it is essential to ensure that other operations that are performed by the file system filter driver should not be done using impersonation. The risk with impersonation is typically that the caller has fewer privileges than the driver making the call. Thus, if a call is made with impersonation, it might fail, while it would succeed without impersonation.

Impersonation is needed for any operation that creates a new handle because the handle represents the reference to the object and is the point at which the security check has been performed. For example, impersonation is necessary when opening a file or other object (using **ZwCreateSection**, **ZwCreateEvent**, and **ZwCreateFile**, for example). In these calls, the filter driver calling them must ensure that the parameters being passed are valid because other operating system operations will assume that calls originating from kernel mode will have valid parameters. Thus, a filter driver cannot safely pass a user buffer address to any of these functions, even when impersonating.

In the case of **ZwCreateFile**, there is a corresponding I/O manager call, **IoCreateFile**, that should be used with impersonation because it allows the filter driver to specify IO_FORCE_ACCESS_CHECK. Absent this option, the I/O manager will not enforce proper user level access checks.

Coexistence with other File System Filter Drivers

4/26/2017 • 1 min to read • Edit Online

One of the most insidious problems that must be properly handled by a file system filter driver is coexistence with other filter drivers. When building a file system filter driver to coexist with other file system filter drivers, it is best to consider the following issues:

- Filter drivers must consider the presence of other filter drivers in their operations. Any operation performed by the filter driver should be robust enough to survive an additional filter driver using the same or different technique.
- Filter drivers may impact the behavior of other filters by changing the base behavior of the system.
- Increasing the number of filter drivers increases the consumption of scarce resources, notably stack space. File system filter drivers must strive to minimize their use of such scarce resources. Otherwise, malicious user applications can take advantage of such weaknesses to cause the system to fail. Developers should be particularly careful about completion paths and error paths.
- Filter drivers should be conservative in what they send to the lower driver (filter driver or file system) and should be liberal in what they accept. Whenever possible, the filter driver should try to ensure the operations they send to the underlying driver are simple and not complicated (do not perform rename operations during create operations, for example).
- Filter drivers must be cautious about locking. Locks must never be held across file system calls. Various components of the system make very precise and explicit assumptions about lock ordering and functions that can and cannot block. Disturbing this by adding another layer of locking can easily lead to deadlocks. I/O originating from Srv.sys exposes these problems particularly quickly, but they can be seen during normal stress testing as well.

It is imperative that any file system filter driver developer not only design and implement to coexist cleanly with other filter drivers, but also test the filter driver with other filter drivers to ensure that the driver does not introduce security problems within the system.

Memory Mapped Files in a File System Filter Driver

4/26/2017 • 1 min to read • Edit Online

A file system filter driver must be cognizant of the fact that files may be accessed via virtual memory mappings of the files, rather than via the read and write paths. A file system filter driver monitoring changes in the file will miss changes to such files. In general a file system filter driver that wants to deal with memory-mapped I/O has to filter paging I/O. A number of techniques for dealing with this are discussed in the Windows File System Developers List (NTFSD) newsgroup hosted by OSR.

Reparse Points in a File System Filter Driver

4/26/2017 • 1 min to read • Edit Online

A filter driver that processes reparse points must be aware of the risk that an application program might create invalid reparse points. To ensure the strictest security, a driver that handles reparse points must ensure that the data contents of the reparse point itself are verifiable, whether through a secure checksum, encrypted contents, or some other mechanism that ensures invalid reparse points cannot be created by unprivileged applications. For example, a filter driver might require that its reparse points be encrypted using a password shared between an application (or the local security authority, for example), and the driver, in order to ensure that the data contents of the reparse point are valid.

Otherwise, it is possible that a malicious application could create reparse points that have invalid reparse point information. In this case, the file system filter driver must be prepared to handle invalid reparse point data, including self-referential data (data that creates reference loops that might cause some sort of overflow, for example), data overflow issues, and invalid data contents.

Miscellaneous Information

4/26/2017 • 1 min to read • Edit Online

This section contains additional information relevant to minifilter development.

This section includes:

- Anti-virus optimization for Windows Containers
- Offloaded Data Transfers
- Locating Disk Images Correctly
- Understanding Volume Enumerations with Duplicate Volume Names

Anti-virus optimization for Windows Containers

6/14/2017 • 6 min to read • Edit Online

Applies to:

- Windows 10, version 1607
- Windows Server 2016
- AV products running on the Host

This topic describes optimizations that anti-virus products can utilize to avoid redundant scanning of Windows Container files and help improve Container start up time.

Container overview

The Windows Container feature is designed to simplify the distribution and deployment of applications. For more information, see the introduction to Windows Containers.

Containers are constructed from any number of package layers. The Windows base OS package forms the first layer.

Each container has an isolated volume that represents the system volume to that container. A container isolation filter (**wcifs.sys**) provides a virtual overlay of package layers onto this container volume. The overlay is achieved using placeholders (reparse points). The volume is seeded with placeholders before the container first accesses the overlain path. Reads of placeholder files are directed to the backing package file. In this way multiple container volumes can access the same underlying package file data stream.

If a container modifies a file, the isolation filter performs copy-on-write and replaces the placeholder with the contents of the package file. This breaks the "linkage" to the package file for that particular container.

Read redirection

Reads from a placeholder file are redirected to the appropriate package layer by the isolation filter. The redirection is performed at the filter’s level. Since the filter is below the AV range, AV filters will not see the read redirection. AV will also not see the opens of package files performed to set up the redirection.

An AV filter does have full view of all operations on the container system volume. It sees operations on placeholder files as well as the file modifications or new file additions.

Redundant scanning problem

There will likely be many containers depending on the same package layers. The same data stream of a given package file will provide the data for placeholders on multiple container system volumes. As a result, there is potential for redundant AV scans of the same data in every container. This has an unnecessary negative impact on the performance of containers. This is a signification cost given that containers are expected to start quickly and may be short-lived.

Recommended approach

To avoid redundant scanning on containers it is suggested that an AV product modify its behavior as described below. It is up to the AV product to determine the risk/reward benefit to its customers for this approach. For more about that, see Benefits and risks.

1. Package install

During package installation, the management tools will lay out files in the package under the layer root. The AV Filter should continue to scan the files as they are being placed in the package root and it normally would. This ensures that all the files in the layers are initially clean with respect to malware.

2. Container start and execution

For real-time scanning of a container volume, AVs should scan in a way that avoids redundancy. Placeholder files need special consideration. Files modified by the container or new files created in the container are not redirected so redundant scanning is not a concern.

To avoid redundant scans, the AV filter first needs to identify container volumes and placeholders on those volumes. For various reasons there is no direct way for an AV filter to query if a volume is a container volume or if a given file is a placeholder file. The isolation filter hides the placeholder reparse point for application compatibility reasons (some applications do not behave correctly if they are aware they are accessing reparse points). Also, a volume is only a container volume while a container is running. The container may be stopped and the volume may remain remounted. Instead, in pre-create, the AV filter must query the file object to determine if it is being opened in the context of a container. It can then attach and ECP to the create and receive the placeholder state on create completion.

The following changes are need in the AV product:

- **During pre-create on a container volume, attach an ECP to the Create CallbackData that will receive the placeholder information.** These creates can be identified by querying the SILO parameters from the fileobject using **IoGetSiloParameters**. Note the filter must specify the size in the **WCIFS_REDIRECTION_ECP_CONTEXT** structure. All the other fields are out fields set if the ECP is acknowledged.

- **In post-create, if the ECP is acknowledged, examine the ECP redirection flags.** The flags will indicate if the open was serviced from package layer or from the scratch root (new or modified files). Flags will also indicate if the package layer is registered and whether it is remote.

  - For opens that are serviced from a remote layer, AV should skip scanning the file. This is indicated by the redirection flags:

    ```
    WCIFS_REDIRECTION_FLAGS_CREATE_SERVICED_FROM_LAYER &&
    WCIFS_REDIRECTION_FLAGS_CREATE_SERVICED_FROM_REMOTE_LAYER
    ```

    Remote layers can be assumed to have been scanned on remote host. Hyper-V Container packages are remote to the utility VM that hosts the container. Those packages will be scanned normally on the Hyper-V host when they are accessed by the utility VM over SMB loop-back.

    Since VolumeGUID and FileId do not apply over remote, these fields will not be set.

  - For opens that are serviced from a registered layer, AV should skip scanning the file. This is indicated by the redirection flags:

    ```
    WCIFS_REDIRECTION_FLAGS_CREATE_SERVICED_FROM_LAYER &&
    WCIFS_REDIRECTION_FLAGS_CREATE_SERVICED_FROM_REGISTERED_LAYER
    ```

    The registered layer should be scanned asynchronously during package installation and after signature update.

    Note Registered layers may not be identified by the system in the future. In this case, local layer files must be individually identified as described in the last bullet.

  - For opens that are serviced from a local package layer, AV should use the provided VolumeGUID and FileId of the layer file to determine if the file needs to be scanned. This will likely require AV to build a cache of scanned files indexed by volume GUID and FileId. This is indicated by the redirection flag:

`WCIFS_REDIRECTION_FLAGS_CREATE_SERVICED_FROM_LAYER`

  - For new/modified files in the scratch location, the AV product should scan the files and perform its normal remediation. This is indicated by the redirection flag:

    `WCIFS_REDIRECTION_FLAGS_CREATE_SERVICED_FROM_SCRATCH`

    Since there is no layer file in this case, VolumeGUID and FileId will not be set.

- **Don't save "this file is serviced from layer" as a permanent marker in its stream context.** A file that is initially serviced from the layer root may be modified after the create. In this case, a subsequent create for the same file may indicate that the create is being serviced from the container volume. The AV Filter needs to understand that this can happen.

Don't use the LayerRootLocations registry key

In the past, we recommended using the `LayerRootLocations` registry key to get the location of base image. AV products should no longer use this registry key. Instead, use the approach recommended in this topic to avoid redundant scanning.

The registry location that had been used to register package layers:

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Virtualization\LayerRootLocations`

Benefits and risks

Consider the following benefits and risks to using these new optimizations for AV products.

Benefits

- No impact to container start or execution time (even for the first container).
- Avoids scanning of the same content in multiple containers.
- Works for Windows Server Containers. For Hyper-V Container, this works for the packages but requires additional work for running Container.

Risks

If a container is launched in the time between signature update and the next scheduled proactive anti-malware scan, the files executed in the container are not being scanned with respect to the latest anti-malware signatures. To mitigate this risk, the AV product could skip scans for redirected files only if there has not been a signature update since the last proactive scan. This would limit container performance degradation until a proactive scan is completed with the latest signatures. Optionally, the AV product can trigger a proactive scan in this situation so that subsequent container launches are more efficient.

Offloaded Data Transfers

4/26/2017 • 11 min to read • Edit Online

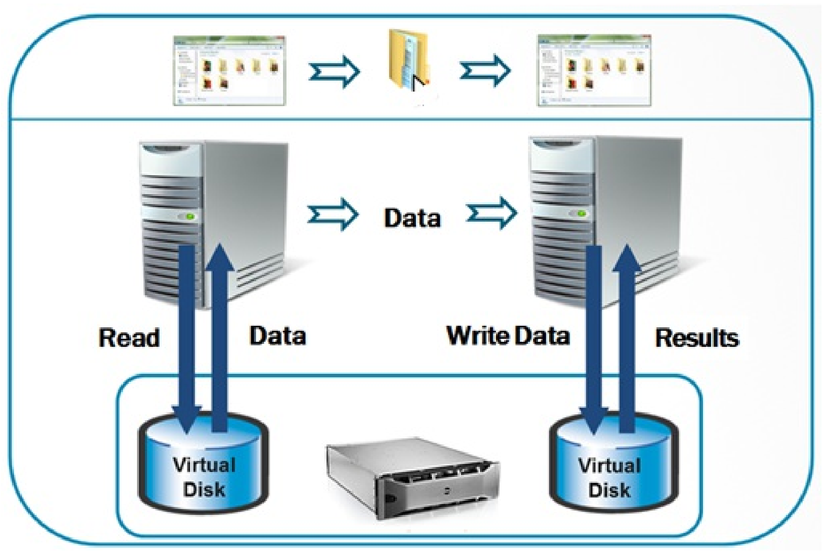
Transferring data between computers or within the same computer is a frequent file system activity. Using the standard **ReadFile** and **WriteFile** functions work well from a functional point of view, but it involves heavy data movement through all levels of the system and potentially across a network. This can affect the availability of the systems involved in the transfer and the network connecting the systems. The advanced capabilities available with many storage subsystems provide a more efficient means of performing the 'heavy lifting' task of data movement.

Starting with Windows 8, applications can take advantage of these capabilities to help offload the process of data movement to the storage subsystem. File system filters can typically monitor these actions by intercepting read and write requests to a volume. Additional actions are required for filters to be aware of offloaded data transfers.

Typical Data Transfers

Moving data around in an application scenario today is quite simple. It involves reading the data into local memory, then writing it back out to a new location. The following diagram illustrates this scenario.

This scenario involves copying a file between two locations on two different file servers, each with its own virtual disk exposed through an Intelligent Storage Array (ISA). The initiating system first needs to read the data from the source virtual disk into a local buffer. It then packages and transmits the data through some transport and protocol (like SMB over 1GbE) to the second system, which then receives the data and outputs it to a local buffer. Then, the target system writes the data to the destination virtual disk. This scenario describes a very typical Read/Write method of data transfer that is performed multiple times by many different applications every day.



While standard reads and writes work well in most scenarios, the data intending to be copied may be located on virtual disks managed by the same Intelligent Storage Array. This means that the data is moved out of the array, onto a server, across a network transport, onto another server, and back into the same array once again. The act of moving data within a server and across a network transport can significantly impact the availability of those systems; not to mention the fact that the throughput of the data movement is limited by the throughput and availability of the network.

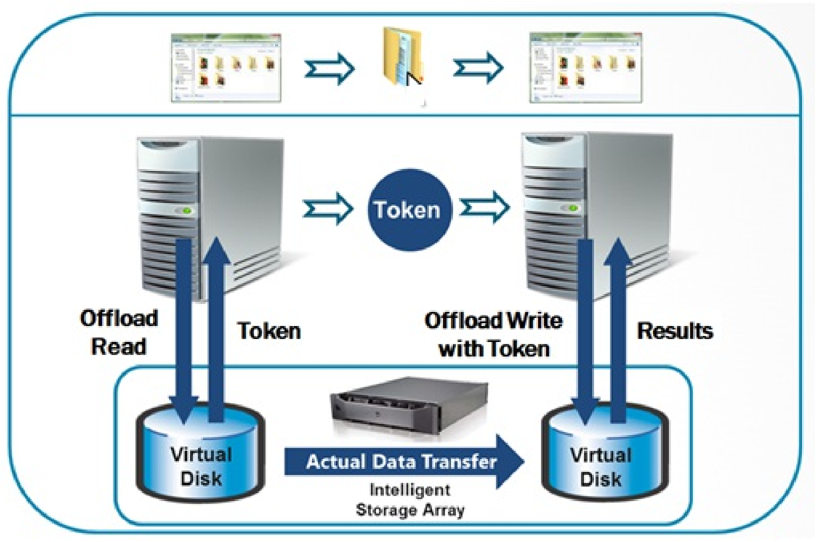
Offloaded Data Transfers (ODX)

Offloading the Data Transfer

Two new FSCTLs are introduced in Windows 8 that facilitate a method of offloading the data transfer. This shifts the burden of bit movement away from servers to bit movement that occurs intelligently within the storage subsystems. The best way to visualize the command semantics is to think of them as analogous to an unbuffered read and an unbuffered write.

FSCTL_OFFLOAD_READ
This control request takes an offset within the file to be read and a desired length in the **FSCTL_OFFLOAD_READ_INPUT** structure. If supported, the storage subsystem hosting the file receives the associated offload read storage command and generates a token, which is a logical representation of the data intended to be read at the time of the offload read command. This token string is returned to the caller in the **FSCTL_OFFLOAD_READ_OUTPUT** structure.

FSCTL_OFFLOAD_WRITE
This control request takes an offset within the file to be written to, the desired length of the write, and the token that is a logical representation of the data to be written. If supported, the storage subsystem hosting the file to be written receives the associated offload write storage command. It first attempts to recognize the given token, and then performs the write operation if possible. The write operation is completed underneath Windows, and therefore components on the file system and storage stacks will not see the data movement. Once the data movement is complete, the number of bytes written is returned to the caller.



Similar to the first diagram, a simple file copy between two virtual disks on two different servers is shown. Instead of doing normal reads and writes, we offload the heavy lifting of bit movement to the storage array. The first system issues the offload read operation, requesting the array to generate a token representing a point-in-time view of the data to be read within the region of the first virtual disk. The first system then transmits the token to the second system, which in turn issues an offload write operation to the second virtual disk with the token. The array then interprets the token and attempts to perform the data movement between the virtual disks. You'll notice that the actual data transfer occurs within the intelligent storage array, and not between the two hosts. This significantly improves availability of the two systems while virtually eliminating network traffic between the systems.

Integration with the Copy Engine

The core copy engine in Windows is used by **CopyFile** and related functions. Starting with Windows 8, the copy engine will transparently attempt to use offloaded data transfers before the traditional copy file code path. Since the copy APIs are used by most applications, utilities, and by the shell, these callers are able to use offloaded data transfer capabilities by default with little, if any, code modification or user intervention.

The following steps summarize how the copy engine attempts an offloaded data transfer:

1. The copy engine issues a **FSCTL_OFFLOAD_READ** on the source file to get a read token.
2. If there was a failure in retrieving the read token, the copy engine falls back to traditional reads and writes (the traditional copy file code path). If the failure indicates that the source volume does not support offload, the copy

engine also marks the volume in a per-process cache. The copy engine will not try offload any more for the volumes in the per-process cache.

3. If the token was successfully retrieved, the copy engine attempts to issue **FSCTL_OFFLOAD_WRITE** commands on the target file in large chunks until all the data that is logically represented by the token has been offload written.
4. Any errors in performing the offload read or write results in the copy engine falling back to the traditional code path of reads and write, starting from where the offload code path ended off (where the read or write was truncated). If the failure indicates that the destination volume does not support offload, or the source volume cannot reach the destination volume, the copy engine updates the same per-process cache so it will not try offload on these volumes. This per-process cache will reset periodically.

The following functions support offloaded data transfers:

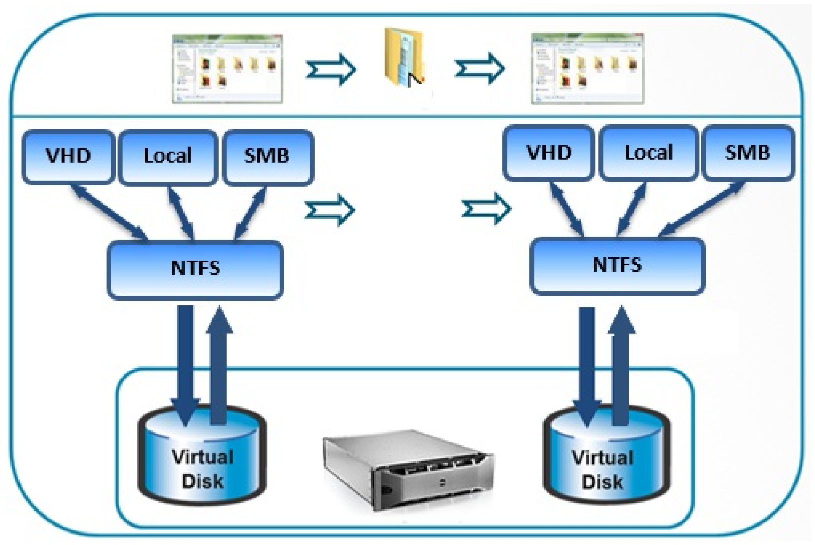
- **CopyFile**
- **CopyFileEx**
- **MoveFile**
- **MoveFileEx**
- **CopyFile2**

The following functions do not support offloaded data transfers:

- **CopyFileTransacted**
- **MoveFileTransacted**

Supported Offload Data Transfer Scenarios

Support for the offload operations is provided in the Hyper-V storage stack and in the Windows SMB File Server. Where the backing physical storage supports ODX operations, callers can issue **FSCTL_OFFLOAD_READ** and **FSCTL_OFFLOAD_WRITE** to files residing on VHDs or on remote file shares, whether from within a virtual machine or on physical hardware. The following diagram illustrates the most basic supported source and destination targets for offloaded data transfers.



File System Filter Opt-In Model and Impact to Applications

Filter Manager, starting with Windows 8, allows a filter to specify offload capability as a supported feature. File system filters attached to a volume can collectively determine if a certain offloaded operation is supported or not; if it is not, the operation fails with an appropriate error code.

A filter must indicate that it supports **FSCTL_OFFLOAD_READ** and **FSCTL_OFFLOAD_WRITE** through a registry **DWORD** value named **SupportedFeatures**, located in the driver service definition in the registry at

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\<filter driver name>\. This value contains bitfields where the bits determine which functionality is opted-in, and should be set during filter installation.

Currently, the defined bits are:

FLAG	MEANING
SUPPORTED_FS_FEATURES_OFFLOAD_READ 0x00000001	Filter supports FSCTL_OFFLOAD_READ
SUPPORTED_FS_FEATURES_OFFLOAD_WRITE 0x00000002	Filter supports FSCTL_OFFLOAD_WRITE

The filter opt-in model can be enabled or disabled based on the value present in the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\FileSystem\FilterSupportedFeaturesMode registry key, which has the following values:

FILTERSUPPORTEDFEATURESMODE VALUE	MEANING
0 (Default)	Do normal opt-in processing.
1	Never opt-in (equivalent to setting SupportedFeatures to 0 on all filters attached)

Testing

To check the supported features of the stack, there is an updated command within the fltmc utility. Run **fltmc instances –v [volume]:** as an elevated user, and check the *SprtFtrs* column. If the *SprtFtrs* value for a filter is set to 0, it implies that the filter is blocking offload on this volume. If the *SprtFtrs* field is set to 3, both offload operations are supported.

Checking Feature Support in IRP Processing

As part of IRP processing, the **FsRtlGetSupportedFeatures** routine retrieves the aggregated **SupportedFeatures** state for all filters attached to the given volume stack. Components such as I/O Manager and SRV (SMB) call this routine to validate the **SupportedFeatures** state for all the filters on the stack. Components that roll their own offload IRPs should call this function to validate opt-in support for that operation.

Considerations for Filter Drivers

Offloaded data transfer is a new way to move data around in the data center. Due to the integration of offload logic in the core copy engine, many applications by default will have the ability to perform offloaded data movement without explicitly opting in. As a result, filter developers need to understand how these new operations impact filters. Not understanding these operations fully or not evaluating the new data flow can potentially result in scenarios where data can become inconsistent or corrupted. The following list summarizes a set of action items for filter developers to take note of with offload:

- Understand the new data flow, the impact to the filter, and the ability of the filter to support these offloaded operations.
- Update your filter installer to add a REG_DWORD value for **SupportedFeatures** to the HKLM\System\CurrentControlSet\Services\[filter] subkey. Initialize it to specify offload functionality.
- For filters that want to act upon offload operations, update the registration to **IRP_MJ_FILE_SYSTEM_CONTROL** to handle **FSCTL_OFFLOAD_READ** and **FSCTL_OFFLOAD_WRITE**.
- For filters that need to block offloaded operations, return the status code STATUS_NOT_SUPPORTED from within the filter. Do not rely upon the registry value to enforce blocking offload operations since it can be changed by end users. A filter should explicitly allow or disallow offload operations.

Copy Tokens

With offloaded operations, the file data is not seen by the I/O stack. Instead it is seen as a 512-byte token that is the logical proxy for the data. This token is either an opaque and unique string of a vendor-specific format generated by the storage subsystem, or can be of a well-known type to represent a pattern of data (such as a data range that is logically equivalent to zero). Modifications to the data that the token is a proxy for will either result in invalidating the token, or for the storage subsystem to persist the original data by some vendor-specific means (such as through a snapshot mechanism). Subsequent offload read requests to a specified range in a file results in unique tokens.

There are classes of tokens that represent a pattern of data that is well defined. The most common well known token is the Zero Token which is equivalent to zero. When a token is defined as a Well Known Token, the **TokenType** member in the **STORAGE_OFFLOAD_TOKEN** structure will be set to STORAGE_OFFLOAD_TOKEN_TYPE_WELL_KNOWN. When this field is set, the **WellKnownPattern** member determines which pattern of data the token is.

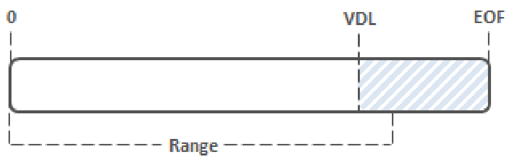
- When the **WellKnownPattern** field is set to STORAGE_OFFLOAD_PATTERN_ZERO or STORAGE_OFFLOAD_PATTERN_ZERO_WITH_PROTECTION_INFORMATION, it indicates the Zero Token. When this token is returned by a **FSCTL_OFFLOAD_READ** operation, it indicates that the data contained within the desired file range is logically equivalent to zero. When this token is provided to a **FSCTL_OFFLOAD_WRITE** operation, it indicates that the desired range of the file to be written to should be logically zeroed.
- Other than the Zero Token, there are no other Well Known Token patterns currently defined. It is not recommended that users define their own Well Known Token patterns.

Truncation

The underlying storage subsystem that Windows communicates with can process less data that was desired in an offload operation. This is called truncation. With offload read, this means that the returned token represents a range of the data less than that which was requested. This is indicated by the **TransferLength** member in the **FSCTL_OFFLOAD_READ_OUTPUT** structure, which is a byte count from the beginning of the range of the file to be read. For offload write, a truncation indicates that less data was written than was desired. This is indicated by the **LengthWritten** member in the **FSCTL_OFFLOAD_WRITE_OUTPUT** structure, which is a byte count from the beginning of the range of the file to be written. Errors in command processing, or limitations within the stack for large ranges, result in truncation.

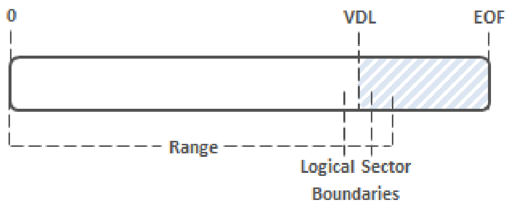
There are two scenarios in which NTFS truncates the range to be offload read or written:

1. The copy range will be truncated to Valid Data Length (VDL) if VDL is before the End of the File (EOF). This assumes that VDL is aligned to a logical sector boundary, otherwise see scenario.

   During a **FSCTL_OFFLOAD_READ** operation, the flag OFFLOAD_READ_FLAG_ALL_ZERO_BEYOND_CURRENT_RANGE is set in the **FSCTL_OFFLOAD_READ_OUTPUT** structure indicating that the rest of the file contains zeros, and the **TransferLength** member is truncated to VDL.

2. Similar to Scenario 1, but when VDL is not aligned to a logical sector boundary, the desired range is truncated by NTFS to the next logical sector boundary.



Limitations

- Offload operations are supported only on NTFS volumes.
- Offload operations are supported through remote file servers if the remote share is a NTFS volume and if the server is running Windows Server 2012 (assuming the remote stack also supports offload operations).
- NTFS does not support offload FSCTLs performed on files encrypted with Bitlocker or NTFS encryption (EFS), de-duplicated files, compressed files, resident files, sparse files, or files participating in TxF transactions.
- NTFS does not support offload FSCTLs performed on files within a volsnap snapshot.
- NTFS will fail the offload FSCTL if the desired file range is unaligned to the logical sector size on the source device or if the desired file range is unaligned to the logical sector size on the destination device. This follows the same semantics as non-cached IO.
- The destination file must be pre-allocated (**SetEndOfFile** and not **SetAllocation**) before **FSCTL_OFFLOAD_WRITE**.
- In processing offload read and offload write, NTFS first calls **CcCoherencyFlushAndPurgeCache** to commit any modified data in the system cache. This is the same semantic as non-cached IO.

Locating Disk Images Correctly

4/26/2017 • 1 min to read • Edit Online

Disk imaging and volume management tools must be aware of where to locate the image data area on disk and the impact on user data when using a snapshot created by Volsnap. The snapshot will contain corrupted data when the disk image is located improperly.

Some imaging products make an incorrect assumption that the BIOS Parameter Block (BPB) on the disk is only 4K bytes when in fact it will always extend to the first 8K bytes of the disk. As a result, depending on cluster size of the volume, if an imaging tool locates user data in last 4K byte portion of the BPB, it can result in data loss after Volsnap snapshots are created for the volume. This is due to the fact that the imaging tool placed user data in a reserved area of the disk.

The logical sector size set for the device, such as 512 bytes or 4K bytes, does not determine the size of the BPB as it is always 8K bytes.

The following table describes the sector layout and byte offset requirements for the data area of a disk image.

BCB Reserved Area Image Data Area 0 (512 byte sector) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 + 0 (byte offset) 4096 8192

Understanding Volume Enumerations with Duplicate Volume Names

4/26/2017 • 1 min to read • Edit Online

When enumerating volumes, it is possible for duplicate volume names to appear in a resulting volume information list.

To help understand why this can occur, consider the following scenario: the volume enumeration routine **FltEnumerateVolumeInformation** is used to enumerate all system volumes. This results in a buffer filled with volume information structures - one for each volume known to filter manager. In this buffer, each volume information structure can be of type **FILTER_VOLUME_BASIC_INFORMATION** or **FILTER_VOLUME_STANDARD_INFORMATION**, but not both.

Given this list of volume information structures, it is possible for multiple list elements to contain the same volume name. That is, the **FilterVolumeName** members of two or more list elements could be identical. This is possible because all filter manager enumeration routines, such as **FltEnumerateVolumes**, enumerate volumes including those that have been dismounted but have not been torn-down (due to the fact that open files still exist on the volume). Thus, when a volume becomes dismounted, its name can appear more than once in a volume information list - once for its current mounted state and once for its prior dismounted but non-torn-down state, in the simplest case.

If duplicate volume names appear in a volume information list, each group of identical names is explained by the above description. However, it is possible to confirm the above scenario by using the following procedures:

- If the list is populated with structures of type FILTER_VOLUME_STANDARD_INFORMATION, identify a group of structures whose **FilterVolumeName** members are equal. If one or more of the structures in this group has the FLTFL_VSI_DETACHED_VOLUME flag set in its **Flags** member, the volume associated with the group was in a dismounted but non-torn-down state. This confirms why duplicate volume names exist. Repeat this procedure for all such remaining groups, if applicable.
- If the list is populated with structures of type FILTER_VOLUME_BASIC_INFORMATION, convert this list to its equivalent FILTER_VOLUME_STANDARD_INFORMATION structure form and proceed as in the previous bullet point.

Note The FILTER_VOLUME_STANDARD_INFORMATION structure is only available starting with Windows Vista.

Routines and structures affected by this topic include the following:

FILTER_VOLUME_BASIC_INFORMATION

FILTER_VOLUME_STANDARD_INFORMATION

FilterVolumeFindFirst

FilterVolumeFindNext

FltEnumerateVolumeInformation

FltEnumerateVolumes

FltGetVolumeInformation