In this chapter:

- *Why Use Filter Drivers?*
- *Basic Steps in Filtering*
- *Some Dos and Don 'ts in Filtering*

12

Filter Drivers

The Windows NT I/O subsystem was designed to be extensible. One of the ways in which the capabilities of the I/O subsystem can be extended is by developing filter drivers. Chapter 2, *File System Driver Development,* provided an introduction to filter drivers in Windows NT. This chapter takes a detailed look at designing and implementing filter drivers for the Windows NT operating system.

First, we'll discuss why you may want to use filter drivers to achieve some of your objectives. This is followed by a discussion of some fundamental steps involved in developing filter drivers, including how to attach to a target device object, how to create your own IRP structures, how to use completion routines to perform postprocessing upon IRP completion, and how to stop filtering by detaching from a target device object.

We'll conclude with a discussion of some issues you should be familiar with when attempting filter driver design and development. The diskette accompanying this book provides a complete sample filter driver implementation that can be used as a template in designing your own kernel-mode filter driver.

Why Use Filter Drivers?

The fundamental reason for any of us to design and develop kernel-mode software for Windows NT is to provide added value beyond what is provided with the core operating system environment. This is also the motivating factor behind the design and development of filter drivers.

Two design principles adopted by the NT I/O Manager make developing value-added software easier than with other operating systems.

First, the I/O Manager design implements a client-server model for the I/O subsystem. Any user- or kernel-mode component can request the services of practically any other loaded kernel-mode driver. The requesting module is then the client of the target driver that will satisfy the request. There are few restrictions mandated by the I/O Manager on when a client component can invoke a driver (the server for the request) and what kind of services can be requested.

One example of the usage of this client-server model is when file system drivers request services from lower-level device drivers. What is more unusual, though certainly possible, is for file systems to request services from other file system drivers installed on the machine, or even for lower-level intermediate drivers to request services from higher-level file system drivers.

You should give careful consideration to the following whenever you design a driver that requests services from either other higher-level kernel-mode drivers or kernel-mode drivers at the same level in the calling hierarchy:

- The different scenarios under which your driver can be invoked
- The different scenarios under which your driver would request services from other kernel-mode modules
- Restrictions on when you can incur page faults within your driver module
- Assumptions made by higher-level drivers, such as file systems; the file systems adapt their behavior depending on what the top-level component is for an I/O request
- Resource acquisition hierarchies that must be defined and strictly implemented for resource acquisition across kernel-mode modules

Second, the NT I/O Manager supports a layered driver model. As each IRP is processed, it passes through various layers of the driver hierarchy until it is finally resolved by some driver via a call to loCompleteRequest ( ) . Therefore, it's easy for a third-party driver to insert itself into the existing calling hierarchy and get the opportunity to process the I/O Request Packets.

In order to cooperate with the I/O Manager in supporting such a layered driver module, your design must conform to the following basic requirements:

- Always invoke the services of other kernel-mode drivers in the standard manner by using the loCallDriver () function.
- Once an IRP has been sent on to another driver, do not touch it.
- You can, however, register a completion routine to be invoked when the IRP has been completed.
- Unless you develop tightly coupled drivers that use privately defined lOCTLs to communicate with each other, you must never depend on whether the

request you have forwarded goes directly to your target driver or is intercepted by another filter driver module.

- The filter driver module must present the same interfaces as those presented by the original target of the request.
- Treat other driver modules as black boxes.
- Your driver must not be dependent upon how the target driver implements processing for your I/O request.

What Is a Filter Driver?

A filter driver is a kernel-mode driver. It is developed primarily to intercept requests targeted to an existing kernel-mode driver, to allow the addition of new functionality beyond what is currently available.

Figure 12-1 illustrates this concept.

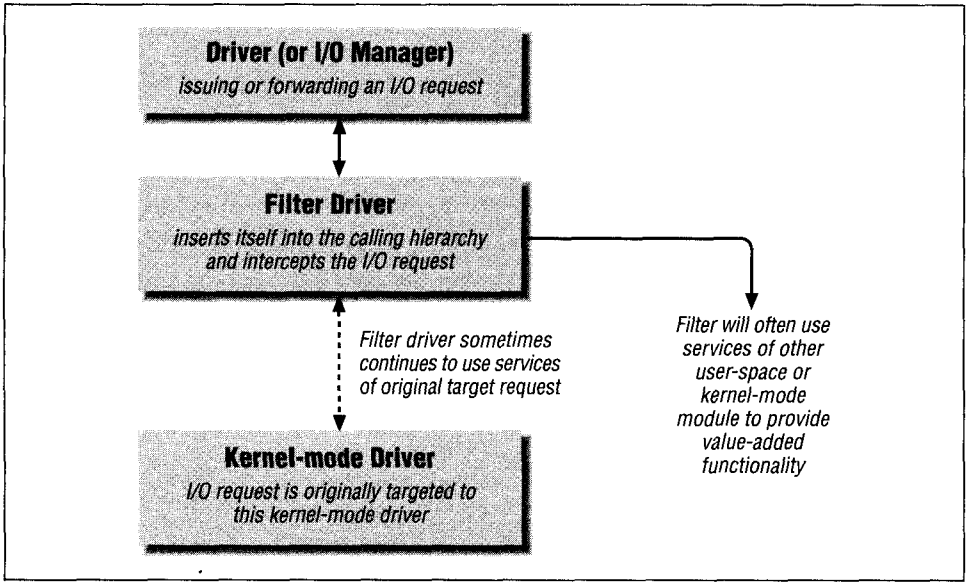

Figure 12-1. Inserting a filter driver to intercept requests

As shown in Figure 12-1, I/O requests targeted to a specific driver are intercepted by the filter driver module. The filter driver may either use the services of the original target of the I/O request, or use the services of other user-mode or kernel-mode software to provide value-added functionality.

When Can I Use a Filter Driver?

You should consider designing a filter driver whenever you wish to affect the current flow of processing for certain I/O requests. Therefore, if you want to provide some software that will extend, modify, or completely supplant an existing module and if you wish to maintain complete transparency to the user when providing your specialized functionality, consider designing a filter driver.

For example, suppose you decide to design and implement on-line encryption/decryption functionality for the data stored on existing Windows NT file systems. Currently, the operating system does not provide any such functionality. However, hypothesize that you possess the technology to implement a secure encryption algorithm. What you would really like to do is the following:

- Use the services of the Windows NT native file systems to store and retrieve user data

  It would not be cost-effective to design your own file system implementation to store encrypted data on disk. Besides, users would typically wish to continue to use native Windows NT file system services for storing their data and would like to use your software only to encrypt sensitive data stored on such file systems.

- Intercept all user write requests and encrypt data being stored to disk for targeted files, directories, or complete mounted logical volumes

  Given that you will not design a new file system driver, you will want to intercept existing file system requests so that the user can specify files, directories, or even entire mounted logical volumes to be encrypted on-the-fly. When a write request issued by the user is received, your software module should somehow be able to intercept such write requests and encrypt the user-supplied data before it is stored to disk.

- Intercept all user read requests and decrypt data (if required) before returning it to the user

  Now that you have successfully encrypted user-supplied data and stored it on disk using the services of the native file systems, you must also provide the services of decrypting the data whenever an authorized user tries to read it.

It seems obvious that a filter driver would serve your purposes admirably in the preceding example problem. The filter driver would allow you to intercept user I/O requests and perform your encryption/decryption processing on the data, transparently to the user. Furthermore, it is not necessary to design your own file system or special device driver to manage and transfer data on secondary storage devices, and therefore your filter driver would continue to use the services provided by existing drivers on the system.

More Examples of Filter Drivers

Other examples of situations where a filter driver could be used include:

To provide virus detection functionality

Imagine for a minute that you want to provide a new virus-detection module for the Windows NT operating system. This virus-detecting module performs its tasks in real-time; therefore, it will attempt to detect any viruses in files being copied to a mounted logical volume and refuse the data transfer if such a virus is found. How would you go about doing this?

Figure 12-2 illustrates how a filter driver that layers itself above a mounted logical volume device object managed by a file system driver can perform the virus detection functionality.

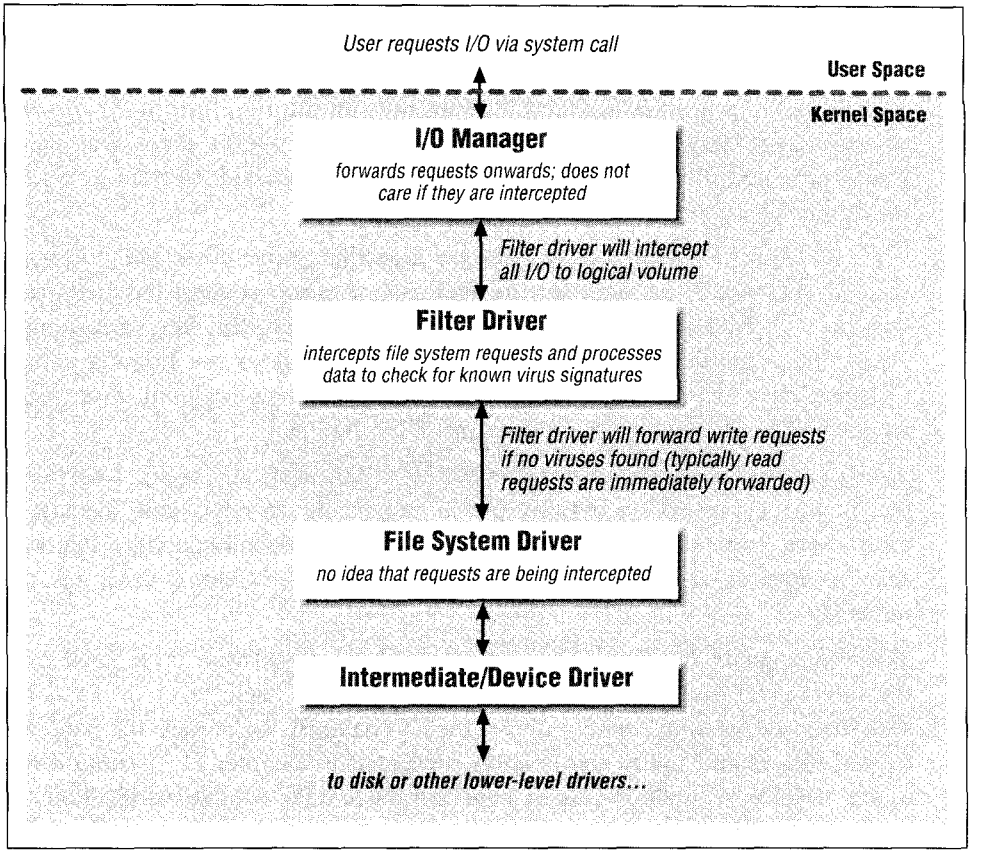

Figure 12-2. Filter driver used in virus detection

The virus detection software module can be implemented as a filter driver that intercepts I/O targeted to one or more mounted logical volumes. When-

ever any user's I/O request is received by the Windows NT I/O Manager for a file residing on a mounted logical volume, the I/O Manager normally forwards the request to the file system driver managing the mounted logical volume.

Before forwarding the request, however, the I/O Manager also checks to see if any other device object has layered itself over the device object representing the mounted logical volume and redirects the request to that device object, which is at the top of the layered list of device objects. In order to intercept I/O requests, the virus-detecting filter driver module has to create a device object that layers (or attaches) itself to the device object representing the mounted logical volume.

Therefore, the filter driver module intercepts the I/O before it reaches the file system. Now, the virus-detection module can check for any virus signatures in the data being written out to disk. Note that in most cases, read requests can be immediately forwarded by the filter driver to the file system.

If any virus signature is detected, the filter driver can reject the write request, protecting the user's physical disks from possible corruption. If no virus signature is detected, the filter driver can safely forward the IRP to the file system for further processing.

Note that the file system driver has no idea that some other filter driver is layered above it. It behaves (as always) as if the user request has been sent directly to it by the I/O Manager. By the same token, the filter driver must always be cognizant of the fact that the file system does not know about its existence and must therefore ensure that it does not do anything that would violate any fundamental assumptions made by the FSD.

Virus-detection software must also be able to automatically check for viruses that might be present on existing media (especially on removable media). In most cases, virus-detection software will also provide functionality that will scan removable media whenever they are reinserted into a drive on the machine.

This functionality requires that your virus-detection software layer itself over the lower-level disk driver (for the removable drive) itself, layer over the file system (in order to accurately detect media changes), or require the software to understand and utilize information presented in Chapter 11, *Writing a File System Driver III,* on how file system drivers handle volume verification for removable media.

Implement HSMfunctionality.

Hierarchical storage management (HSM) means different things to different people. However, it often involves automatic transfer of infrequently used

data to slower but cheaper secondary storage media and an automatic transfer back to regular storage if the migrated data files are accessed. Figure 12-3 illustrates how a filter driver could be part of such an HSM solution.

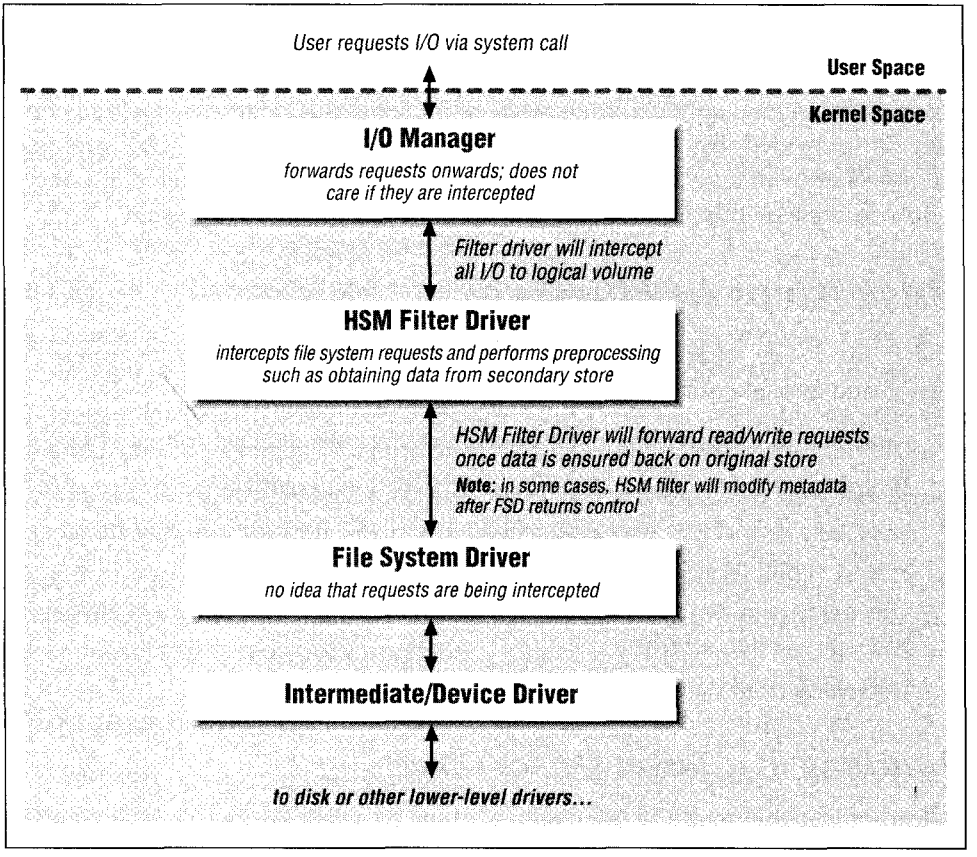

Figure 12-3. HSMfilter driver

Consider an HSM filter driver that migrates older, infrequently accessed files to a slower device. If a user now wishes to access or modify the migrated file, the HSM driver will typically transfer data back to the local file system before forwarding the request to the FSD. In this way, the FSD can be completely ignorant about the migration/retrieval of data performed by the HSM driver.

Often, HSM drivers leave a little *stubfile* on the original file system as a placeholder, once data has been migrated. The actual size of this stub file is generally 0 bytes, although it may contain some metadata stored by the HSM driver for administrative convenience. If a user tries to list the directory entries for a directory whose files have been migrated, the HSM module may have to massage the information returned by the FSD (e.g., file size) in

response to a file information request in order to maintain complete transparency about the migration operation that was performed. Therefore, the HSM module may choose to register a completion routine before forwarding directory control or file information requests to the FSD, allowing it to perform appropriate modification of the information returned by the FSD before it is finally returned to the caller.

You could undoubtedly come up with many additional ways in which the functionality provided by the I/O subsystem could be extended. The preceding examples are simply a sample of the number of ways in which filter drivers can help you implement your ideas for providing added value to the system.

Basic Steps in Filtering

There are a few operations that you should become familiar with when you design and implement a new filter driver:

- Attaching to a target device object, to intercept calls directed to that object
- Building IRPs that can be dispatched to drivers managing target device objects

  Note that your driver may either build associated IRPs for a master IRP sent to you or create new master IRP structures.
- Specifying a completion routine to be invoked when an attached driver finishes processing an IRP
- Detaching from the target device object when appropriate

Attaching to a Target Device Object

Before proceeding with the discussion on how to attach to a target device object, it is useful to understand the following terms:

- The *filter driver* is the kernel-mode driver that you design and implement.
- The *source device object* (also known as the *filter-driver device object)* is the device object that you create in order to perform a logical attachment between your driver and the original target of the I/O requests.
- The *target device object* is the device object, representing a physical, virtual, or logical device, to which I/O request packets are currently directed.

  Your goal is to intercept the I/O request packets sent to the target device object.
- The *target driver* is the kernel-mode driver that manages (and provides the dispatch functions for) the target device object.

The process of associating a source device object created by your filter driver with the target device object, such that the I/O Manager will automatically redirect requests to your driver is called an *attach* operation.

As mentioned earlier, filter drivers provide their value-added functionality by intercepting requests targeted to an existing driver. Once a filter driver has begun intercepting all of the requests targeted to the existing driver module, it can either augment the functionality provided by the existing driver or supplant it altogether.

There are a few simple steps your driver must perform to successfully attach to a target device object:

1. Get a pointer to the target device object.
2. Create your own device object that will be used in the attach operation.
3. Ensure that your driver is set up to process the I/O requests, originally directed to the target device object, that will now be redirected to it instead.
4. Ensure that the fields in your device object are set correctly to maintain complete transparency to the modules that normally invoke the target driver.
5. Request the I/O Manager to create an attachment between the two device objects.

Once the attach operation has been completed, the I/O Manager will begin redirecting I/O requests to your device object instead of forwarding them to the driver managing the target device object.

Code Fragment

The following code fragment illustrates how the attach operation can be implemented by your filter driver.

```
NTSTATUS SFilterAttachTarget(
PUNICODE_STRING                TargetDeviceName,
ACCESS_MASK                    DesiredAccess,
BOOLEAN                        InvokedFromDriverEntry)
{
    // Declarations ...

    try {

        // Get a pointer to the target device object. Read the discussion
        // provided later in this chapter for information on how to
        // ensure that a file system is always mounted
        // before you attach to the underlying device object.
        if (!NT_SUCCESS(RC = IoGetDeviceObjectPointer(TargetDeviceName,
                                    DesiredAccess,
                                    &PtrTargetFileObject,
                                    StPtrTargetDeviceObject))) {
```

```
    try_return(RC);
}

// File Object has been referenced. No need to reference the
// device object, since a successful attach operation will ensure
// that the device object is not deleted without our being
// notified. Further, if you do reference the underlying device
// object, you will effectively exclude all new open operations,
// just in case the device object can be opened exclusively only
// (since the reference will count as an open operation).

// Now, create a new device object for the attach operation.
if ( !NT_SUCCESS(RC = loCreateDevice(
                            SFilterGlobalData.SFilterDriverObject,
                            sizeof(SFilterDeviceExtension),
                            NULL,         // unnamed device object
                            PtrTargetDeviceObject->DeviceType,
                            PtrTargetDeviceObject->Characteristics,
                            FALSE,
                            &(PtrNewDeviceObject)))){
    // failed to create a device object, leave.
    try_return(RC);
}

// Initialize the extension for the device object. The extension
// stores any device-object-specific (global) data (e.g., a
// pointer to the device object to which you performed the attach
// operation).
PtrDeviceExtension = (PtrSFilterDeviceExtension)
                                PtrNewDeviceObject->DeviceExtension;
SFilterlnitDevExtension(PtrDeviceExtension,
                        SFILTER_NODE_TYPE_ATTACHED_DEVICE );
InitializedDeviceObject = TRUE;

// If we were not invoked from the DriverEntry() function, mark the
// fact that this device object is no longer being initialized.
// If the device object is created during driver initialization,
// the I/O Manager will do this for us.
if ( !InvokedFromDriverEntry) {
    PtrNewDeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
}

// Acquire the resource exclusively for our newly created device
// object to ensure that dispatch routines requests are not
// processed until we are really ready.
ExAcquireResourceExclusiveLite  (
        &(PtrDeviceExtension->DeviceExtensionResource), TRUE);
AcquiredDeviceObject = TRUE;

// The new device object has been created. Perform the attachment.
RC = IoAttachDeviceByPointer(PtrNewDeviceObject,
                                        PtrTargetDeviceObject)   ;
// The only reason we would fail (and possibly get STATUS_NO_SUCH_
// DEVICE)
```

```
        II is if the target was being initialized or unloaded and neither
        // should be happening at this time.
        ASSERT(NT_SUCCESS(RC));

        // Note that the AlignmentRequirement, the StackSize, and the
        // SectorSize values will have been automatically initialized for
        // us in the source device object (the I/O Manager does this as
        // part of processing the IoAttachDeviceByPointer( ) request).

        // We should set the Flags values correctly to indicate whether
        // direct I/O, buffered I/O, or neither is required. Typically,
        // FSDs (especially native FSD implementations) do not want the I/O
        // Manager to touch the user buffer at all.
        PtrNewDeviceObject->Flags = (PtrTargetDeviceObject->Flags &
                        (DO_BUFFERED_IO | DO_DIRECT_IO));

        // Initialize the TargetDeviceObject field in the extension.
        // This is used by us when (if) we wish to forward I/O requests
        // to the target device object.
        PtrDeviceExtension->TargetDeviceObject = PtrTargetDeviceObject;
        PtrDeviceExtension->TargetDriverObject =
                                    PtrTargetDeviceObject->DriverObject;
        // Some bookkeeping.
        SFilterSetFlag (PtrDeviceExtension->DeviceExtensionFlags,
                              SFILTER_DEV_EXT_ATTACHED) ;

        // We are there now. All I/O requests will start being redirected
        // to us until we detach ourselves.

    try_exit:    NOTHING;

    } finally {
        // Cleanup stuff goes here.
        if (AcquiredDeviceObject) {
            SFilterReleaseResource (&(PtrDeviceExtension->
                                                DeviceExtensionResource) ) ;
        }

        if ( !NT_SUCCESS(RC) && PtrNewDeviceObject) {
            if (InitializedDeviceObject) {
                // The detach routine will take care of everything.
                //A code fragment for the detach routine is provided
                // later in this chapter.
                SFilterDetachTarget (PtrNewDeviceObject,
                                            PtrTargetDeviceObject,
                                            PtrDeviceExtension)  ;
            }
        }

        // Dereference the file object. Once you have done so, you can
        // forget all about the target file object. But please remember to
        // always do this! Failure to dereference the file object will
        // result in a dangling file object structure that in turn will
        // prevent unloading/dismounting of the target device object.
```

```
        if (PtrTargetFileObject) {
            ObDereferenceObject(PtrTargetFileObject);
            PtrTargetFileObject = NULL;
        }
    }

    return(RC);
}
```

The following definition of a device extension structure, as defined by the sample filter driver code, will be useful in understanding the previous code fragment:

```
typedef struct _SFilterDeviceExtension {
    // A signature (including device size).
    SFilterIdentifier                NodeIdentifier;
    // This is used to synchronize access to the device extension
    // structure.
    ERESOURCE                        DeviceExtensionResource;
    // The sample filter driver keeps a private doubly linked list of all
    // device objects created by the driver.
    LIST_ENTRY                       NextDeviceObject;
    // See Flag definitions below.
    uint32                           DeviceExtensionFlags;
    // The device object we are attached to.
    PDEVICE_OBJECT                   TargetDeviceObject;
    // Stored for convenience. A pointer to the driver object for the
    // target device object (you can always obtain this information from
    // the target device object).
    PDRIVER_OBJECT                   TargetDriverObject;
    // You can associate other information here.
} SFilterDeviceExtension, *PtrSFilterDeviceExtension;

#define    SFILTER_DEV_EXT_RESOURCE_INITIALIZED        (0x00000001)
#define    SFILTER_DEV_EXT_INSERTED_GLOBAL_LIST        (0x00000002)
#define    SFILTER_DEV_EXT_ATTACHED                    (0x00000004)
```

Notes

The code fragment for the SFilterAttachTarget() function illustrates how simple it is to perform an attachment between a device object created by your driver and another named device object. The code fragment follows closely the sequence of steps listed earlier for performing the attach operation.

You should note that the attachment can be performed as easily if the target device object is not a named device object. However, you cannot open an unnamed device object; therefore, in order to be able to attach to an unnamed device object, your driver must have some other (driver-specific) method devised to obtain a pointer to the target device object.

The following sections describe some of the support functions provided by the I/O Manager that will prove useful to you in developing your filter driver (to perform an attachment between your device object and the target device object).

IoGetDeviceObjectPointer()

The arguments for this function are well-described in the DDK:

```
NTSTATUS
IoGetDeviceObjectPointer(
    IN PUNICODE_STRING        ObjectName,
    IN ACCESS_MASK            DesiredAccess,
    OUT PFILE_OBJECT          *FileObject,
    OUT PDEVICE_OBJECT        *DeviceObject
);
```

The IoGetDeviceObjectPointer() function is often used by filter drivers to obtain a pointer to a target physical/virtual device object or to the highest-layered device object attached to the target device object. Here are the steps executed by the I/O Manager to implement this function:

1. The I/O Manager performs an open operation on the target object, identified by the ObjectName argument (e.g., \Device\C:).

   Note that the open request will typically recurse back into the NT I/O Manager. The DesiredAccess value determines whether or not a mount sequence is initiated by the I/O Manager in processing the open operation; the I/O Manager may choose to initiate a mount sequence if no logical volume has yet been mounted on the target physical/virtual device when the open request is being processed.

2. The I/O Manager then obtains a pointer to the file object that is created as a result of processing the open request.

   The open request (if successful) returns a file handle. The I/O Manager uses the ObReferenceObjectByHandle() function to obtain a pointer to the associated (referenced) file object.

3. The I/O Manager uses the IoGetRelatedDeviceObject() function to get a pointer to the highest-layered device object that may be attached to the target device.

   The argument to the IoGetRelatedDeviceObject() function is the file object pointer obtained in Step 2.

4. Finally, the I/O Manager closes the file handle obtained in Step 1 before returning control to the caller.

The I/O Manager can safely return pointers to the file object representing the successful open operations, as well as to the associated device object, even though the handle obtained from the open operation has been closed, because the file object structure is referenced in Step 2.

What Happens After the Attach Operation?

In order to appreciate the value of attempting an attach operation, you should understand what happens once you have performed the attach. You know that the I/O Manager will now reroute the IRPs destined for the target device object to your driver (and your source device object) instead. But how does the I/O Manager do this? To answer this question, let's look at the attach operation in greater detail.

The attach operation

Recall from Chapter 4, *The NT I/O Manager,* that each device object structure has a field called AttachedDevice. This field is used by the I/O Manager to keep track of the linked list of attached devices for a particular target device object. Note that I mentioned a *linked list* of attached device objects and not just a single attached device object; the clear implication is that multiple filter device objects could potentially exist that are attached to a specific target device object. Therefore, you can conceive of a chain (or a layer) of attached filter device objects; each of these attached device objects will have an opportunity to process IRPs sent to the target device object.

There are three ways in which your driver can request an attach operation:

`loAttachDeviceByPointer()`

When your driver invokes the I/O Manager to perform an attach between the target device object and your source device object using loAttachDeviceByPointer(), the I/O Manager performs the following sequence of operations: loAttachDeviceByPointer(), loAttachDeviceToDeviceStack(), and loAttachDevice().

a. The I/O Manager will get a pointer to the topmost device object that had been previously attached to the target device object.

   The code used to do this is encapsulated within an I/O Manager routine called loGetAttachedDevice(), which is available to third-party developers as well:

```
PDEVICE_OBJECT
loGetAttachedDevice(
    IN PDEVICE_OBJECT DeviceObject
);
```

   The implementation of this function appears to be pretty trivial and is demonstrated in this code fragment:*

---

* Note that the actual code implemented by the I/O Manager is probably slightly different than the fragment presented here; however, the logic presented here is accurate.

```
PDEVICE_OBJECT
IoGetAttachedDevice(PDEVICE_OBJECT TargetDeviceObject) {
    PDEVICE_OBJECT    ReturnedDeviceObject = TargetDeviceObject;

    while (ReturnedDeviceObject->AttachedDevice) {
        ReturnedDeviceObject = TargetDeviceObject->AttachedDevice;
    }

    return(ReturnedDeviceObject);
}
```

Think of the attached list of device objects as a stack-based list. The last object inserted into the list will be at the head of the list. Extend this analogy a bit further, and you can see that the last device object to perform the attach operation will be the first object to get a crack at the IRPs sent to the target device object.

In order to maintain this last-in-first-chance-at-IRP ordering, the I/O Manager gets a pointer to the topmost device object in the linked list of device objects in order to continue processing the attach request. If, however, yours happens to be the first attach request for the target device object, the I/O Manager will directly use the pointer to the target device object (supplied by you) in the following steps.

Now, the I/O Manager will ensure that the device object you are attempting to attach to is not being deleted.

If the device object is being deleted or if the corresponding driver has an unload pending against it, the I/O Manager will immediately reject your attach request. You can expect to get an error such as STATUS_NO_SUCH_ DEVICE from the I/O Manager. If everything seems to be in order, the I/O Manager proceeds to the next step.

b. The I/O Manager will physically complete the attach operation.

   The following steps are executed by the I/O Manager to complete the attach operation:

   i. The ReturnedDeviceObject->AttachedDevice field is set to point to the source device object.

   ii. The StackSize field in the source device object is set to (ReturnedDeviceObject->StackSize + 1).

   Note that once the attach has been completed, the I/O Manager will redirect all IRPs sent to the target device object to your driver. The I/O Manager does not know what you will do with the IRPs; it can assume the worst case, however, (in terms of I/O stack location usage) where you may simply perform some preprocessing or register a completion routine and forward the IRP to the next driver

in the list of layered drivers. Since the attaching of your device object could require the IRP to be routed through one more layered driver, the I/O Manager ensures that the number of stack locations that will be allocated for all subsequent IRPs directed to the target device object will be enough to last through all the drivers that may process the IRP. The I/O Manager will set the AlignmentRecruirement field and the SectorSize field in the source device object created by your driver to be the same as those in the target device object.

IoAttachDeviceToDeviceStack()

This function call was first made available in Windows NT Version 4.0. It is functionally similar to the preceding loAttachDeviceByPointer() routine and is invoked in the same manner (i.e., your driver must supply both the source and target device object pointer values). However, this function performs one additional task: if the attach operation completes successfully, IoAttachDeviceToDeviceStack() will return a pointer to the previous highest-layered device object to which your source device object was attached.

The returned device object pointer value can prove to be useful if your filter driver forwards any intercepted IRPs to the next driver in the calling hierarchy.

Note that if your driver happened to be the first to perform an attach operation to the target device object, the returned device object pointer will be the same as the target device object pointer supplied by your driver when invoking the IoAttachDeviceToDeviceStack() function.

NOTE Prior to Version 4.0, your driver could first invoke loGetAttachedDevice() followed by loAttachedDeviceByPointer() to achieve practically the same functionality as is now provided by the IoAttachDeviceToDeviceStack() function. Also, the loGetDeviceObjectByPointer() function returns a pointer to the highest-layered device object attached to the target device object.

---

loAttachDevice()

This function is defined as follows:

```
NTSTATUS
loAttachDevice(
    IN PDEVICE_OBJECT        SourceDevice,
    IN PUNICODE_STRING       TargetDevice,
    OUT PDEVICE_OBJECT       *AttachedDevice
);
```

The loAttachDevice() function also performs an attachment between two device objects. However, this function accepts the target device name instead of a pointer to the target device object. It will open the target device object on behalf of your driver and use the target device object pointer to perform the actual attach operation.

The steps executed by this function are as follows:

a. The loAttachDevice() function invokes loGetDeviceObject-Pointer() internally, to obtain a pointer to the target device object.

   The DesiredAccess value is set to FILE_READ_ATTRIBUTES.

b. The loAttachDevice() function executes the same sequence of steps as those described earlier, in performing an attach operation between the source device object and the target device object.

   Just as in the case of the loAttachDeviceByPointer() function, the I/O Manager initializes the StackSize, AlignmentRecruirement, and SectorSize fields in the SourceDevice object structure.

c. The I/O Manager dereferences the file object pointer returned from the internal call to loGetDeviceObjectPointer().

d. A pointer to the previous highest-layered device object (for the target device) is returned to the caller in the AttachedDevice argument.

Note that your driver must have created the source device object before you can invoke the loAttachDevice() function.

You may be wondering whether it would be preferable to invoke loAttachDevice() directly, instead of invoking loGetDeviceObjectPointer() in your driver followed by a call to loAttachDeviceByPointer() or loAttachDeviceToDeviceStack().

There is one subtle difference between the two methods of performing an attach operation. This difference is only important if your driver wishes to layer over an FSD logical volume device object (as opposed to layering over a lower-level device driver disk device object).

The loAttachDevice() function will always open the target device (identified by the device name that you supply to the function) with the DesiredAccess value set to FILE_READ_ATTRIBUTES. This type of open request will not result in a mount operation being initiated by the I/O Manager on the target physical/virtual/logical device if such a mount operation has not yet taken place. Therefore, if your driver wishes to attach to the device object representing the mounted logical volume on drive C: and if the name supplied by your driver is \Device\C:, you cannot really be sure that loAttachDevice() will do what

you expect, since you may actually end up with your source device object having been attached to the physical device object identified by \Device\C:.

However, if your driver wishes to ensure that it is always attached to an FSD device object representing a logical volume mounted on the target drive, then you can invoke loGetDeviceObjectPointer() function directly from your driver by specifying the DesiredAccess to some value like FILE_READ_ACCESS. This type of DesiredAccess value will result in the I/O Manager initiating a mount process (if no logical volume has yet been mounted on the target device), and the returned device object pointer will refer to the device object representing the mounted logical volume.* Your FSD can then request the attach by invoking loAttachDeviceByPointer() orloAttachDeviceToDeviceStack().

Once you invoke one of preceding three functions, loAttachDeviceByPointer(), IoAttachDeviceToDeviceStack(), and loAttachDevice() successfully, you can be assured that the attach operation has been completed by the NT I/O Manager.

You must be careful whenever you request an attach operation, since all new IRPs targeted to the target device object will immediately begin getting rerouted to you, instead of being sent to the original target of the I/O request. Therefore, be prepared to handle such requests immediately or block them until you complete all your initialization.

IRP routing after the attach

Now that you have performed the attach, you should start getting first access to all the IRPs sent to the target device object, right? Well, not quite. You may get first chance at IRPs, or you may get called after some other driver has had its way with the IRP, or your driver may never be called for an IRP sent to the target device object.

Why? To understand when your filter driver is invoked (and when it is not), you need to understand the I/O Manager-supplied utility function called loGetRelatedDeviceObject(). This function is also available to you when you develop a kernel-mode driver and is defined as:

```
PDEVICE_OBJECT
loGetRelatedDeviceObject(
    IN PFILE_OBJECT FileObject
);
```

* Another method that you eould use to ensure that a mount is always initiated (if required) by the I/O Manager is to speeify a name such as \Device\C:\ instead of \Device\C: only. The trailing \ indicates that you wish to open the root directory (as opposed to performing a *direct device open* of the target device) and will force the I/O Manager to initiate a mount sequence.

The function `IoGetRelatedDeviceObject()` is always invoked internally by the NT I/O Manager whenever it needs to determine where it should send an IRP for a user-initiated I/O operation (e.g., the NtReadFile ( ) function invoked by a thread).* The following steps are executed in this function:

1. The I/O Manager checks whether the supplied FileObject has a mounted Volume Parameter Block (VPB) associated with it.

   VPB structures were discussed in detail earlier in this book. You may recall that when a file system successfully mounts a logical volume, a pointer to the device object (created by the FSD) representing the mounted logical volume is stored in the VPB->DeviceObject field.

   If the FileObject->Vpb field is nonnull and if the FileObject->Vpb->DeviceObject field is nonnull, the I/O Manager will invoke the loGetAttachedDevice () function for the FileObject->Vpb->DeviceObject structure and use the returned device object pointer when invoking loCallDriver().

   The implication here is that if a logical volume has been mounted on a physical/virtual/logical device object, the I/O Manager will redirect I/O requests to the highest-layered driver that has performed an attach operation on the device object created by the FSD to represent the mounted logical volume.

   When will the Vpb pointer for a file object not be set to NULL? Well, recall that the Vpb pointer for a file object is set by the FSD whenever a successful create/open operation has been performed on the file object (as was described in Chapter 11, *Writing a File System Driver III).* Therefore, you should infer that if a file stream residing on a logical volume has been successfully opened, and subsequently an I/O operation is received for the file stream, this particular check made by the I/O Manager will succeed and the IRP will be appropriately dispatched.

2. If the preceding check fails because the Vpb pointer is set to NULL, then the I/O Manager tries harder to determine where to send the IRP.

   In the previous case, the Vpb pointer was nonnull because the file stream had been opened. However, for certain file objects, the Vpb pointer may still be NULL. In this case, the I/O Manager checks whether the file object has an associated device object that was mounted by some file system. This can be done by checking the FileObject->DeviceObject field. If nonnull (indicating that the file object is associated with some "real" device object), and if

---

* Note that the I/O Manager also uses the loGetRelatedDeviceObject () function internally when processing a synchronous/asynchronous page write or a synchronous page read request. Therefore, filter drivers layered over a FSD will get the opportunity to process page faults and/or paging I/O writes (including those initiated due to memory-mapped files).

the FileObject->DeviceObject->Vpb->DeviceObject is nonnull (indicating that a file system has mounted this device object), then the I/O Manager will invoke the loGetAttachedDevice() function on the FileObject->DeviceObject->Vpb->DeviceObject structure and use the returned device object pointer when invoking loCallDriver().

3. If both of these checks fail to yield a device object structure pointer, the I/O Manager uses the device object associated with the file object.

   When both the preceding checks fail, more than likely the I/O request is being issued to an open physical/virtual/logical device that has not yet been mounted. If an I/O operation is being issued directly to this device object (e.g., for raw access to the device), the I/O Manager will invoke the loGetAttachedDevice() function on the FileObject->DeviceObject structure and use the returned device object pointer when invoking loCallDriver().

Given this information, you can see that even after you attach to a target device object, it is not guaranteed that you will receive the IRP before any other driver in the calling hierarchy. If some other driver has attached itself to your device object, that driver is ahead of yours in the call chain. Then, it is no longer certain that you will ever see the IRP, since it is left completely to the discretion of each driver whether or not it will forward the IRP to the next driver or complete the IRP itself.

You should also note one important point: what happens if you attach to a device object representing a physical disk partition after a file system has mounted itself onto the device object? Well, as you can easily infer, you will not get to process most IRPs because the I/O Manager will always send the IRP to the file system driver first (or rather, to the highest-layered device object attached to the file system volume device object). The FSD, in turn, will forward the IRP (for actual, physical I/O operations) directly to the target physical/virtual device object (to which you have attached yourself) via an invocation to loCallDriver(). Your device object will not even be considered to receive the IRP.

NOTE Most FSD implementations store a pointer to the target physical/virtual device object when they mount a logical volume on the device object in their VCB structure. They use this device object pointer when invoking loCallDriver(). They do not, however, invoke loGetAttachedDevice() on the target device object pointer before invoking loCallDriver().

Create/open requests

The I/O Manager performs the following actions to determine the target driver to which the create/open request should be sent, before actually forwarding the request to a target FSD or filter driver:

- For relative create/open requests, the I/O Manager determines the target driver from the related file object specified in the create/open request.

  Recall from earlier chapters that create/open requests can be specified with a filename relative to the name contained in the (supplied) previously open directory file object. For relative file create/open requests, the I/O Manager obtains a pointer to the target device object by invoking the loGetRelatedDeviceObject () function on the related file object.

- For all other create/open requests, the I/O Manager sends the request either to the highest-layered driver attached to the device object representing the mounted logical volume or directly to the device object representing the target physical/virtual/logical device.

  A create/open request can specify either a device open (e.g., \Device\C:) or a file/directory on a logical volume mounted on the physical/virtual/logical device object.

  When a request is received by the I/O Manager for a direct device open operation, the I/O Manager uses the target device object supplied by the Object Manager and forwards the create/open request to the device driver managing this device object. Note that any filter driver attached to this target device object will not get to intercept this create/open request.

  For all other create/open requests, the I/O Manager always tries to ensure that an FSD mounts a logical volume on the target physical/virtual/logical device. Once an FSD has claimed the device and mounted a logical volume on the target device object, the I/O Manager uses the loGetAttachedDevice () function to get a pointer to the highest-layered device object attached to the volume device object created by the FSD. If no filter driver has attached itself to the volume device object, the create/open request is forwarded directly to the responsible FSD.

If your driver wishes to intercept all create/open requests that may be sent to a particular logical volume, ensure that your filter driver creates a device object that attaches itself to the target device object representing the mounted logical volume before the IRP for the mount operation is completed but after the FSD has completed mount-related processing.

The obvious way to accomplish this is to intercept mount requests issued to the target FSD, register a completion routine to be invoked once the mount request

(IRP) has been completed, and initiate (and complete) the attach sequence from within your completion routine before returning control to the I/O Manager.

Building IRPs

Whether you develop a file system driver or a filter driver, you will undoubtedly find it necessary to create IRPs that your driver will subsequently use in dispatching I/O requests. You can either decide to allocate and initialize such IRPs yourself, or you could decide to use one of the I/O Manager-supplied utility functions to help you in these tasks.

The following routines can prove useful when you start creating your own I/O request packets. Note that the Windows NT DDK also provides a description of the functions presented here.

loAllocatelrpO

The loAllocatelrp() function is used internally by the I/O Manager to allocate IRPs. It is also available to third-party driver developers. This function is defined as follows:

```
PIRP
loAllocatelrp(
    IN CCHAR        StackSize,
    IN BOOLEAN      ChargeQuota
);
```

Parameters:

StackSize
: The I/O Manager uses the value contained in this argument to determine the number of I/O stack locations that could possibly be used in processing this IRP. The I/O Manager must allocate sufficient memory to contain the specified number of I/O stack locations. Your driver can use the TargetDeviceObject->StackSize value to pass to the loAllocatelrp() function.

ChargeQuota
: This determines whether the memory allocated for the IRP should be charged to the quota allocated to the requesting process. Typically, filter drivers will set this argument to FALSE (the I/O Manager generally sets it to TRUE when invoking the function internally to forward user I/O requests to a target FSD).

Functionality Provided:

The I/O Manager will allocate an IRP either from a zone containing preallocated IRPs* or by directly invoking ExAllocatePoolWithQuotaTag()/ExAllocatePoolWithTag(). Once this function returns a success code back to your driver, you can check the value of the Zoned field in the IRP to determine whether or not the IRP was allocated from a zone (if you are curious enough to do so). All IRP structures are always allocated from nonpaged pool.

For reasons of efficiency and to avoid kernel memory fragmentation, the I/O Manager preallocates two separate zones for IRP structures that require only a single I/O stack location and for those that require four (or fewer) I/O stack locations. If, in an invocation to loAllocatelrp(), a thread requests more than four I/O stack locations, the I/O Manager cannot use either of the two preallocated zones. In this situation, the I/O Manager requests memory via a call to the NT Executive ExAllocatePool() function.

Of course, in high-stress situations, it is always possible that the IRP zones may be exhausted and the I/O Manager will resort to requesting memory from the NT Executive pool management support package.

The loAllocatelrp() function also initializes certain fields in the IRP before returning the IRP to your driver. Note that the entire structure is zeroed by the I/O Manager before any fields are initialized. The initialized fields include the Type, Size, StackCount, CurrentLocation, ApcEnvironment, and Tail.Overlay.CurrentStackLocation fields.

The I/O Manager tries to ensure that a valid IRP pointer is returned to the thread that invokes this function. If the IRP can be allocated from a zone, the I/O Manager tries to get a free IRP structure from the appropriate zone. If the number of I/O stack locations requested precludes allocation from a zone (i.e., it is greater than 4)† or if the appropriate zone is exhausted, the I/O Manager allocates the IRP by invoking the appropriate NT Executive function (listed previously). If no memory is available for the IRP structure and if the previous mode of the caller is kernel mode, the I/O Manager will request memory from the NonPagedPoolMustSucceed memory pool. Therefore, although it is possible that the loAllocatelrp() function will return NULL if the previous mode happened to

---

* Beginning with Windows NT Version 4.0, the I/O Manager may decide to use lookaside lists instead of zones. The Zoned field in the IRP has been renamed to AllocationFlags. The flag value in this field determines whether the IRP has been allocated from a fixed-size block of memory (e.g., a zone or lookaside list), from the nonpaged-must-succeed pool, or from the system nonpaged pool. This change, however, does not fundamentally affect the discussion presented in the chapter.

† The number of I/O stack locations associated with preallocated IRPs is subject to change. Therefore, your driver must never depend on the fact that the I/O Manager will allocate IRPs with a certain number of stack locations from a zone.

be user mode and if system memory was seriously depleted, failure to obtain memory for an IRP when the caller executes in the context of the system process will result in a bugcheck.

WARNING Contrary to the documentation in the Windows NT DDK, you should not invoke the lolnitializelrp() function (described below) for the new IRP structure obtained by calling loAllocatelrp().

As a matter of fact, the lolnitializelrp() function performs exactly the same initialization as will have already been performed by IoAllocateIrp() for you. Also, part of the initialization performed by lolnitializelrp() involves zeroing the entire IRP structure. This is unfortunate for those unwary developers that do call lolnitializelrp() on IRPs obtained via loAllocatelrp(), since zeroing the IRP structure will erroneously clear the Zoned flag in the IRP and will subsequently often lead to a system crash at very unexpected times."

---

lolnitializelrp()

This function is provided to support drivers that allocate IRP structures themselves (instead of requesting an IRP from the I/O Manager) and is defined as follows:

```
VOID
lolnitializelrp(
    IN OUT PIRP    Irp,
    IN USHORT      PacketSize,
    IN CCHAR       StackSize
);
```

Parameters:

Irp

This is the IRP structure to be initialized.

PacketSize

This is the size of the IRP to be initialized. Typically, this will be the value computed by the `IoSizeOfIrp()` macro supplied in the DDK.

StackSize

This is the number of I/O stack locations for which memory has been allocated by your driver.

---

* When the I/O Manager tries to release memory allocated for the IRP, it will check the Zoned flag value to determine whether memory should be returned back to the zone or should be released back to the system nonpagecl pool. Even if the IRP had been allocated from a zone, the Zoned flag will have been cleared by lolnitializelrp(), and the I/O Manager will erroneously return the memory back to the system nonpaged pool leading to a subsequent system crash.

Functionality Provided:

Typically, your driver will invoke the `IoInitializeIrp()` after it has allocated an IRP by directly invoking `ExAllocatePool()` (or from some zone/lookaside list maintained by your driver), instead of requesting that the I/O Manager allocate the IRP structure on your behalf.

The `IoInitializeIrp()` function initializes the Type, Size, StackCount, CurrentLocation, ApcEnvironment, and Tail.Overlay.CurrentStackLocation fields. These are exactly the same fields as those initialized by invoking `IoAllocateIrp()` (described previously). The IRP is zeroed before any fields are initialized.

Note that the IRP initialization performed by both the `IoAllocateIrp()` and the `IoInitializeIrp()` functions is rudimentary. Therefore, your driver is responsible for performing all of the additional initialization for the IRP. The actual fields that your filter driver will initialize depends heavily upon the type of I/O request that you are issuing and the target device object (kernel-mode driver) to which you will be issuing the request. Read Chapter 4 to understand the nature of the various fields in the IRP. You should also review the sample filter driver code provided in the accompanying diskette to see how some of the fields are initialized.

IoBuildAsynchronousFsdRequest()

```
PIRP
IoBuildAsynchronousFsdRequest(
    IN ULONG                    MajorFunction,
    IN PDEVICE_OBJECT           DeviceObject,
    IN OUT PVOID                Buffer OPTIONAL,
    IN ULONG                    Length OPTIONAL,
    IN PLARGE_INTEGER           StartingOffset OPTIONAL,
    IN PIO_STATUS_BLOCK         IoStatusBlock OPTIONAL
);
```

Parameters:

`MajorFunction`

The I/O Manager initializes the first I/O stack location with the MajorFunction code value.

`DeviceObject`

This is a pointer to the device object that will be the immediate target for the I/O request. The I/O Manager obtains the number of stack locations to be allocated from the StackSize field in the DeviceObject structure. Furthermore, for read/write I/O requests (for which the IRP is being created), the I/O Manager determines the type of buffering required for the target driver (DO_DIRECT_IO, DO_BUFFERED_IO, or neither of the two).

Buffer
: Your driver can supply the buffer pointer, which is only required for requests of type IRP_MJ_READ and IRP_MJ_WRITE. Note that for these two MajorFunction types, the Buffer argument is not optional.

Length
: This is the length of any Buffer that may have been supplied.

StartingOffset
: This is the starting offset for a read/write operation.

loStatusBlock
: This contains the results of the operation are returned in this structure (if supplied).

Functionality Provided:

loBuildAsynchronousFsdRequest () will create and initialize a new IRP that can be used by your driver to issue an IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_SHUTDOWN, or IRP_MJ_FLUSH_BUFFERS request to another kernel-mode driver. This function executes the following sequence of steps:

1. It allocates a new IRP using loAllocatelrp ().
2. The MajorFunction field in the first I/O stack location is initialized to the value supplied by your driver.
3. The Userlosb field in the IRP is initialized to the value contained in the loStatusBlock argument.

   Note that upon IRP completion, the I/O Manager uses the field (pointer) value to return the status of the I/O operation.
4. For read/write requests, the I/O Manager performs some additional initialization of the IRP.

   The I/O Manager initializes the appropriate values in the first I/O stack location for read/write requests. For write requests, the Parameters .Write.Length and Parameters .Write.ByteOffset fields in the first I/O stack location are initialized to the Length and StartingOffset arguments (respectively) supplied by your driver, and for read requests, the Parameters .Read. Length and Parameters .Read. ByteOffset fields are initialized.

   If the Flags field in the DeviceObject structure for the target device object specifies DO_BUFFERED_IO, the I/O Manager allocates a system buffer (of the supplied Length) and initializes the AssociatedIrp-> SystemBuffer field to refer to the newly allocated buffer. This buffer will be automatically deallocated by the I/O Manager when the IRP has been

completed (and any data obtained has been copied into the supplied Buffer for IRP_MJ_READ I/O requests). Furthermore, the I/O Manager will copy data from the supplied Buffer to the allocated system buffer for IRP_MJ_WRITE requests before returning the newly allocated IRP to your driver.

If the Flags field in the target DeviceObject structure specifies D0_DIRECT_IO instead, the I/O Manager allocates an MDL describing the supplied Buffer. The I/O Manager also probes and locks the pages for the MDL (for write access in the case of IRP_MJ_READ requests and for read access otherwise). The MdlAddress field in the IRP is initialized to point to this allocated MDL. You should note that the I/O Manager always frees all MDLs associated with an IRP as part of the postprocessing performed in the loCompleteRequest ( ) function.

If neither direct I/O nor buffered I/O has been specified, the I/O Manager will simply set the UserBuffer field in the IRP to point to the supplied buffer.

5. The I/O Manager will initialize the Tail .Overlay.Thread field with the value obtained from KeGetCurrentThread ( ) .

   This is required for subsequent, asynchronous processing of media-verify requests that may be initiated by lower-level disk drivers (in the case of removable media), or for reporting a hard error to the user.

It is important that your driver be aware of those fields in the IRP that the loBuildAsynchronousFsdRequest ( ) function does not initialize. The I/O Manager expects that your driver will initialize the following fields (if appropriate):

`RequestorMode`

Your driver should typically set this value to KernelMode if you are executing in the context of a system worker thread. Otherwise, your driver can use the ExGetPreviousMode ( ) function to determine the value to be set in this field.

`Tail.Overlay.OriginalFileObject`

Set this field to point to the file object structure associated with the I/O request. You will need to do this for all requests except the IRP_MJ_SHUTDOWN IRP.

`FileObject`

Set the field to point to the same value as Tail .Overlay .Original-FileObject.

Your driver can invoke the loBuildAsynchronousFsdRequest ( ) routine at a high IRQL (e.g., IRQL DISPATCH_LEVEL). Furthermore, your driver will set a completion routine to be invoked when the IRP completes, allowing you to

trigger any postprocessing that may be required. You can also free the IRP using the loFreelrp () function after you have completed postprocessing for the request.

WARNING Remember to set a completion routine that will free the IRP allocated via a call to loBuildAsynchronousFsdRequest ( ) . Failure to do so will result in the I/O Manager performing normal completion-related postprocessing on the IRP (see Chapter 4 for details on the postprocessing performed by the I/O Manager). This will lead to unexpected system crashes, since the IRP is not typically set up correctly for such postprocessing.

IoBuildSynchronousFsdRequest()

```
PIRP
loBuildSynchronousFsdRequest(
    IN ULONG                    MajorFunction,
    IN PDEVICE_OBJECT           DeviceObject,
    IN OUT PVOID                Buffer OPTIONAL,
    IN ULONG                    Length OPTIONAL,
    IN PLARGE_INTEGER           StartingOffset OPTIONAL,
    IN PREVENT                  Event,
    OUT PIO_STATUS_BLOCK        loStatusBlock
);
```

Parameters:

As you can observe in the preceding function definition, this routine takes virtually the same arguments as those expected by the loBuildAsynchronousFsd-Request (). The only caveats that you must be aware of are as follows:

- The loStatusBlock argument is no longer optional.

  The I/O Manager expects to complete any IRP allocated using loBuildSynchronousFsdRecruest ( ) . Therefore, you should provide a valid pointer to an IO_STATUS_BLOCK structure when invoking this function. The results of the I/O operation will be returned to you in this structure.

- Your driver must provide a pointer to an initialized Event object.

  Note that by definition, the IRP created by the I/O Manager is expected to be used for a synchronous call to a some kernel-mode driver. Therefore, the I/O Manager expects that the caller (your driver) will wish to wait for completion of the IRP. When the IRP is completed, the I/O Manager will signal the event object supplied by your driver. Remember to initialize the event object before invoking the loBuildSynchronousFsdRequest ( ) function (and to set the event object to the not-signaled state).

Your driver may choose not to wait for the completion of the request. However, you must have some means of deallocating the event structure (and the I/O status block) in this case.

Functionality Provided:

loBuildSynchronousFsdRequest ( ) will create and initialize a new IRP that can be used by your driver to issue a synchronous I/O request to another kernel-mode driver. Internally, this routine invokes loBuildAsynchronousFsd-Request ( ) to do most of the work of allocating and initializing the IRP structure.

After obtaining an IRP structure from the call to IoBuildAsynchronousFsd-Request (), this function initializes the UserEvent field in the IRP with the supplied Event pointer value. Finally, the loBuildSynchronousFsd-Request () function inserts the allocated IRP into the list of pending IRPs for the current thread using the ThreadListEntry field in the IRP. The IRP is automatically dequeued by the I/O Manager from the list of pending IRPs as part of the postprocessing performed on the IRP during loCompleteRequest ( ) .*

Your driver can associate a completion routine for synchronous I/O requests created using the loBuildSynchronousFsdRequest () function. However, you must be careful if you wish to prevent the I/O Manager from completing the IRP by returning STATUS_MORE_PROCESSING_REQUIRED from your completion routine. This is because the IRP is inserted into the list of pending IRPs for the thread that invoked loBuildSynchronousFsdRequest () and failure to remove the IRP from this list could cause a system crash at some later time.

TIP If you need to ensure that the IRP is safely removed from the list of pending IRPs associated with a thread, you should execute the following steps:

— Ensure that you perform the next step in the context of the thread identified by the Tail .Overlay .Thread field in the IRP. You can do this by issuing a kernel-mode APC to the target thread (if required).

— At IRQL APC_LEVEL or higher, invoke the RemoveEntryList () macro on the Irp->ThreadListEntry field.

IoBuildDeviceIoControlRequest()

This function is defined as follows (consult the DDK also for information on this function).

---

* Actually, the dequeue operation takes place in the context of the thread that requested the I/O operation (when performing final postprocessing as part of the APC executed in the context of the requesting thread). Chapter 4 describes the postprocessing performed by the I/O Manager in greater detail.

```
PIRP
loBuildDeviceloControlRequest(
    IN ULONG                    loControlCode,
    IN PDEVICE_OBJECT           DeviceObject,
    IN PVOID                    InputBuffer OPTIONAL,
    IN ULONG                    InputBufferLength,
    OUT PVOID                   OutputBuffer OPTIONAL,
    IN ULONG                    OutputBufferLength,
    IN BOOLEAN                  InternalDeviceloControl,
    IN PREVENT                  Event,
    OUT PIO_STATUS_BLOCK        loStatusBlock
);
```

Parameters:

`loControlCode`

This is the IOCTL code value that will be placed in the Parameters .DeviceloControl. loControlCode field is the first I/O stack location of the newly allocated IRP. The I/O Manager also uses this code to determine the manner in which data should be transferred between the calling module (your driver) and the target for the request.

`DeviceObject`

This is a pointer to the target device object for the request.

`InputBuffer`

This is used by your driver to send data to the target driver. Supplying an input buffer is optional, unless the InputBufferLength contains a nonzero value.

`InputBufferLength`

This is the length of any InputBuffer supplied by you.

OutputBuffer

Your driver can supply such a buffer to receive data from the target driver. You can also use this buffer to send information to the target driver if the method of data transfer is METHOD_IN_DIRECT or METHOD_OUT_DIRECT.* You must supply a valid buffer pointer if OutputBufferLength contains a nonzero value.

`OutputBufferLength`

If this field contains a nonzero value, you must supply a valid OutputBuffer pointer. This field contains the length of the supplied OutputBuffer (if any).

---

* The contents of this buffer (used to send data to the target driver) will naturally be overwritten if the target driver returns information back to you.

`InternalDeviceIoControl`

If set to TRUE, the MajorFunction code value in the first I/O stack location is set to IRP_MJ_INTERNAL_DEVICE_CONTROL, otherwise it is set to IRP_MJ_DEVICE_CONTROL.

Event

IOCTL requests are considered inherently synchronous; therefore, the I/O Manager expects you to supply a valid, initialized event object pointer. This event will be signaled by the I/O Manager when the IRP is completed.

`IoStatusBlock`

Upon IRP completion, the I/O Manager will return the results of the operation in this argument, supplied by your driver.

Functionality Provided:

The IoBuildDeviceIoControlRequest() function allocates and initializes an IRP that can subsequently be used to issue an IOCTL to another kernel-mode driver. Internally, this function uses the services of IoAllocateIrp() to allocate a new IRP structure. This function initializes the following fields in the allocated IRP (in addition to those initialized by IoAllocateIrp()):

UserEvent

This field is initialized to the pointer value supplied in the Event argument. The I/O Manager will set this event to the signaled state upon completion of the I/O request packet.

UserIosb

This field is initialized to the passed-in IoStatusBlock value.

Parameters.DeviceIoControl.OutputBufferLength

This field is initialized to the value supplied in the OutputBufferLength argument.

Parameters.DeviceIoControl.InputBufferLength

This field is initialized to the value supplied in the InputBufferLength argument.

Parameters.DeviceIoControl.IoControlCode

This field is initialized to the passed-in IoControlCode value.

Furthermore, the loBuildDeviceloControlRequest() function also determines the method of data transfer, based upon the loControlCode value:*

- If the loControlCode indicates that the data transfer method is 0 (METHOD_NEITHER), the I/O Manager allocates a system buffer if either InputBufferLength or OutputBufferLength are nonzero.

  The system buffer allocated has a length that is the greater of the InputBufferLength and OutputBufferlength values. The loBuildDeviceloControlRecruest() function initializes the AssociatedIrp. SystemBuffer field in the IRP to point to the allocated system buffer.

  If InputBufferLength is nonzero, the loBuildDeviceloControlRequest() function will copy the contents of the InputBuffer into the allocated system buffer. If the OutputBufferLength is nonzero, the I/O Manager will set the IRP_INPUT_OPERATION flag in the IRP, indicating that the loCompleteRequest() function must copy the contents of the allocated system buffer into the caller supplied output buffer.

  Note that the I/O Manager keeps track of the caller-supplied output buffer by setting the UserBuffer field in the IRP to point to the OutputBuffer.t The system buffer allocated by the I/O Manager is automatically deallocated upon IRP completion.

- If the loControlCode indicates METHOD_IN_DIRECT (value = 1) or METHOD_OUT_DIRECT (value = 2), the I/O Manager allocates a system buffer for the InputBuffer and/or creates an MDL to describe the OutputBuffer.

  If the InputBuffer pointer is nonnull, the loBuildDeviceloControlRequest() function allocates a system buffer of length InputBufferLength. The Associatedlrp.SystemBuffer field in the IRP is set to point to this allocated buffer. The I/O Manager copies the contents of the caller-supplied InputBuffer into the allocated system buffer. Note that the system buffer will be automatically deallocated upon IRP completion.

  If the OutputBuffer pointer is nonnull, the loBuildDeviceloControlRequest() function will create an MDL to describe the supplied Output-

---

* Recall from Chapter 11, *Writing a File System Driver III,* that the IOCTL code value determines the method used in data transfer. The possible methods are METHOD_BUFFERED, METHOD_IN_DIRECT, METHOD_OUT_DIRECT, or METHOD_NEITHER. The two least-significant bits in the IOCTL code determine the data transfer method.

t The target driver must not use this buffer pointer directly unless it is completely sure that it has been invoked in the context of the original user thread. Trying to access this buffer in the context of any other thread will lead to system memory/data corruption and also probably a system crash. Moreover, there is no real reason to use the pointer, since the target driver can access the caller-supplied output buffer directly via the I/O-Manager-provided MDL.

Buffer. Furthermore, the I/O Manager will lock the pages described by the MDL. The MDL will be automatically destroyed by the I/O Manager (and pages unlocked) upon IRP completion.

- If the loControlCode indicates METHOD_NEITHER (value = 3), the I/O Manager will initialize the IRP with the caller-supplied buffer pointer values.

  The Parameters.DeviceIoControl.Type3InputBuffer field is set to the pointer value supplied in the OutputBuffer argument. The UserBuffer field is set to the InputBuffer value.

IoMakeAssociatedIrp()

Filter drivers and file system drivers can use this function to create one or more associated IRPs for a given master IRP. An associated IRP is just like any other IRP, except for the fact that it is logically associated with a single master IRP. An associated IRP can be easily identified by checking for the presence of the IRP_ASSOCIATED_IRP flag in the IRP.

A master IRP can potentially have several IRPs associated with it, but each associated IRP must be uniquely associated with a single master IRP (that is, there exists a one-to-many relationship between a master IRP and its associated IRPs). Associated IRPs cannot become master IRPs themselves, so an associated IRP cannot have other IRPs associated with it. The number of associated IRPs outstanding for a given master IRP can be ascertained by checking the IrpCount field in the master IRP structure.

The IoMakeAssociatedIrp() function is defined as follows:

```
PIRP
IoMakeAssociatedIrp(
    IN PIRP    Irp,
    IN CCHAR   StackSize
);
```

Parameters:

Irp
: This is a pointer to the master IRP for this associated IRP (to be created).

StackSize
: This is the number of stack locations to be allocated for the associated IRP.

Functionality Provided:

The IoMakeAssociatedIrp() function returns a newly allocated associated IRP to your driver. The following steps are executed by the I/O Manager when you invoke this function.

1. The I/O Manager allocates an IRP either from a zone/lookaside list or by requesting nonpaged memory from the NT Executive pool management package.
2. This IRP is initialized in exactly the same manner as described for lolnitializelrp().
3. The I/O Manager sets the IRP_ASSOCIATED_IRP flag value in the newly createdIRP.
4. The Associatedlrp.Masterlrp field is initialized to the Irp argument supplied by your driver.
5. The Tail.Overlay.Thread field is initialized to the Irp-> Tail.Overlay.Thread field value (obtained from the master IRP structure).

If, however, the I/O Manager fails to obtain memory for an IRP structure, it will return NULL to your driver.

Uses of associated IRP structures

Imagine that you have designed an FSD that breaks up a rather large I/O request into fixed-sized pieces and issues the I/O requests in parallel to underlying disk device drivers. You could then decide to simply create multiple associated IRP structures, each describing a subset of the total I/O request, and then dispatch them concurrently (for asynchronous I/O) to the underlying device drivers.

Another use for these structures could be an intermediate driver that provides disk-striping functionality below the FSD. Now, whenever you receive an I/O request from an FSD to a striped device, you will need to break up this request into little stripes, and you would probably like to issue each of these I/O requests concurrently (since typically, each request will be issued to a different physical disk). Associated IRP structures are a natural choice at this time.

Note that you do not have to create associated IRP structures only when executing multiple I/O requests concurrently. You could just as well create associated IRPs that are used in sequential processing. However, associated IRPs lend themselves well to issuing multiple I/O requests in parallel to satisfy a specific user request.

Restrictions on the use of associated IRP structures

If you examine the IRP structure defined in the DDK/IPS kit closely, you will notice that information about associated IRPs is maintained in the following structure:

```
union {
    struct _IRP         *MasterIrp;
    LONG                IrpCount;
```

```
    PVOID              SystemBuffer;
} Associatedlrp;
```

For the master IRP, the count of associated IRPs is maintained in the IrpCount field. For an associated IRP, a pointer leading back to the master IRP is maintained in the Masterlrp field. If neither of these fields are used, the SystemBuffer field can potentially contain a pointer to any system buffer allocated by the I/O Manager for buffered I/O requests.

From the structure definition, certain restrictions can immediately be ascertained:

- If your driver supports buffered I/O and receives a system buffer allocated by the I/O Manager, you will lose the pointer to this buffer in trying to maintain the associated IRP count in your master IRP.
- An associated IRP cannot be dispatched to a driver that expects to receive buffered I/O requests.
- An associated IRP cannot become a master IRP.

  If you develop a filter/intermediate driver that resides below an FSD, it is quite possible that the FSD will create an associated IRP and dispatch it to your driver. If your code tries to create an associated IRP itself (for the IRP received by you), you will run into all sorts of problems.

Completion of associated IRPs

The I/O Manager invokes completion routines for each of the stack locations contained in the associated IRP structure. However, once the completion routines have been invoked (and assuming that none of the completion routines returns STATUS_MORE_PROCESSING_REQUIRED), the loCompleteRequest() function performs the following steps for an associated IRP structure:

- The I/O Manager obtains a pointer to the master IRP for the associated IRP being completed.
- The Associatedlrp.Count field in the master IRP is decremented by 1.
- The memory for the associated IRP structure is freed and so are any MDLs referred to by the associated IRP.
- If the Associatedlrp.Count field in the master IRP is equal to 0, the I/O Manager internally invokes loCompleteRequest() on the master IRP.

As is obvious from this list, many of the steps that would normally be performed when completing a regular IRP structure are skipped by the I/O Manager when processing the completion for an associated IRP.

NOTE As described earlier, associated IRP structures cannot be used in conjunction with buffered I/O data transfers. Therefore, the I/O Manager does not have to worry about copying over any data from a system buffer to a driver/user-supplied buffer. Associated IRPs are typically only used with the direct I/O method of data transfer, in which case an MDL describing the user buffer would probably have been utilized for the data transfer operation (if any).

There are two things your driver can do to prevent this automatic completion of the master IRP by the I/O Manager (in case you wish to control when the master IRP is actually freed):

- Your driver can specify a completion routine for the associated IRP.

  Your driver should return STATUS_MORE_PROCESSING_REQUIRED from this completion routine, which will cause the I/O Manager to immediately terminate further processing of the associated IRP. This will prevent manipulation of the associated IRP count by the I/O Manager and thereby also prevent completion of the master IRP. Completion routines are described in greater detail in the next section.

- Your driver can increase the Associatedlrp.Count field in the master IRP before dispatching the associated IRP to a lower-level driver.

  Although this may sound repugnant (and like bad software engineering), it does work. Your driver can simply increase the Associatedlrp.Count field in the master IRP by 1 before dispatching any associated IRPs that may have been created. This will result in the count not being equal to 0, even after IRP-completion processing of all associated IRPs has been performed by the I/O Manager. Since the count does not equal 0, the I/O Manager will not complete the master IRP.

Completion Routines

The I/O Manager allows kernel-mode drivers to register completion routines associated with I/O stack locations in an IRP. This allows the kernel-mode driver the opportunity to perform any required postprocessing on the IRP after `IoCompleteRequest()` has been invoked either by the driver itself or by some other kernel-mode driver.*

* Completion routines are used by many types of kernel-mode drivers, including file system drivers, filter drivers, and other intermediate drivers.

There can be multiple completion routines associated with each IRP, because completion routines are associated with stack locations in the IRP (and most IRPs have multiple stack locations). However, only one kernel-mode driver can process any particular stack location and therefore, only one completion routine can be associated with each such stack location.

The I/O Manager invokes completion routines in sequence in the loCompleteRequest() function, starting with invoking the completion routine associated with the last stack location to be processed (before loCompleteRequest() was invoked) and proceeding in reverse sequence until the completion routine associated with the first I/O stack location in the IRP has been invoked. This allows for a natural unraveling of I/O stack locations with the last-in-first-out order being preserved.

Figure 12-4 illustrates the sequence in which completion routines are invoked for an IRP with four stack locations.

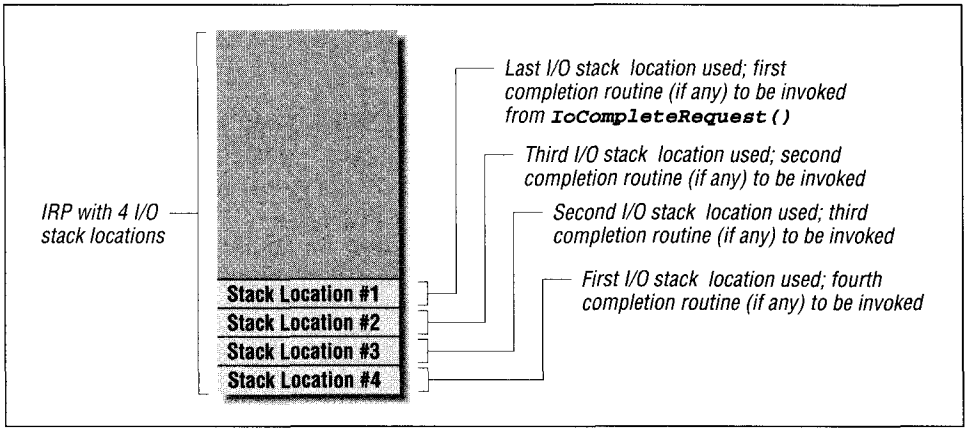

Figure 12-4. I/O manager sequence for invoking completion routines

Specifying a completion routine for IRPs

Your driver should use the loSetCompletionRoutine() macro, which is made available by the NT I/O Manager. Currently, this macro is defined as follows:

```
#define loSetCompletionRoutine( Irp, Routine, CompletionContext,          \
                                Success, Error, Cancel )                  \
{                                                                         \
    PIO_STACK_LOCATION irpSp;                                             \
    ASSERT( (Success) | (Error) | (Cancel) ? (Routine) != NULL : TRUE );  \
    irpSp = IoGetNextIrpStackLocation( (Irp) );                           \
    irpSp->CompletionRoutine = (Routine);                                 \
    irpSp->Context = (CompletionContext);                                 \
    irpSp->Control = 0;                                                   \
    if ((Success)) { irpSp->Control = SL_INVOKE_ON_SUCCESS; }             \
```

```
if ((Error)) { irpSp->Control |= SL_INVOKE_ON_ERROR;}                \
if ((Cancel)) { irpSp->Control |= SL_INVOKE_ON_CANCEL; } }
```

Parameters:

Irp
This is a pointer to the IRP structure.

CompletionRoutine
This is a pointer to the completion routine, supplied by your driver, of type PIO_COMPLETION_ROUTINE. This completion function must be defined as follows:

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

CompletionContext
This is an opaque pointer value that is passed to the completion routine by the I/O Manager.

InvokeOnSuccess
If set to TRUE and if the returned status code to loCompleteRecruest () is STATUS_SUCCESS, the completion routine will be invoked.

InvokeOnError
If set to TRUE and if the returned status code to loCompleteRequest () does not evaluate to a success value, the completion routine will be invoked.

InvokeOnCancel
If set to TRUE and the IRP has been canceled (i.e., Irp->Cancel is TRUE), the completion routine will be invoked.

Typically, your driver will request that the completion routine be invoked regardless of why the loCompleteRecruest ( ) function was called. Therefore, you should set InvokeOnSuccess, InvokeOnError, and InvokeOnCancel to TRUE.

Notice that the completion routine information is placed in the next I/O stack location. This is logical, since that is the I/O stack location to be initialized for the next driver in the calling hierarchy.

Many kernel-mode drivers (especially filter drivers) execute the following steps:

- Allocate a new IRP structure using any one of the I/O Manager-supplied functions described earlier in this chapter.

- Initialize the first I/O stack location (obtained by using the loGetNextlrp-StackLocation() function) with appropriate values and set a completion routine using the loSetCompletionRoutine ( ) function.

The problem with this approach is that when the filter driver completion routine does get invoked, you will find that the device object pointer supplied to your completion routine is NULL. The reason for this will become obvious as you read the following discussion on how the I/O Manager invokes completion routines. Basically, the problem is that your driver neglected to create a stack location for itself in the newly allocated IRP structure, and hence the I/O Manager has no way of determining the device object pointer it should pass on to your completion routine.

To avoid this potential problem (especially if your driver plans to use the passed-in device object pointer in the completion routine), ensure that your driver always creates and initializes a stack location for itself. To do this, you must execute the following steps after obtaining a new IRP structure:

- Use loSetCurrentStackLocation () to set the IRP pointers to the first stack location in the IRP.
- Initialize the first stack location (use loGetCurrentStackLocation() to obtain a pointer to this stack location) with appropriate values. Note that the I/O Manager will update this stack location with a pointer to your device object when you invoke loCallDriver().
- Use loGetNextlrpStackLocation ( ) to get a pointer to the next stack location (to be used by the driver you will invoke with the newly allocated IRP).
- Initialize the next IRP stack location with appropriate values and use loSet-CompletionRoutine ( ) to set your completion routine for the next I/O stack location.

Invoking completion routines

Completion routines are invoked by the loCompleteRequest ( ) function, implemented by the I/O Manager. The loCompleteRequest ( ) function, in turn, is invoked by the kernel-mode driver that will complete processing for the current IRP. The following pseudocode extract illustrates how the I/O Manager invokes completion routines associated with the IRP being completed.

```
while (PtrIrp->CurrentLocation < (PtrIrp->StackCount + 1)) {
    currentStackLocation = loGetCurrentlrpStackLocation(Ptrlrp);

    ...

    // Prepare to process beginning at the next I/O stack location.
    // If any completion routine returns
```

```
    II STATUS_MORE_PROCESSING_REQUIRED and later reissues the
    // loCompleteRequest() call, the I/O Manager will begin processing
    // at the next stack location (which is the correct thing to do).
    (PtrIrp->Tail.Overlay.CurrentStackLocation)++;
    (PtrIrp->CurrentLocation)++;

    if (PtrIrp->CurrentLocation == (PtrIrp->StackCount + 1)) {
        // Some driver has set up a completion routine for the
        // last valid I/O stack location itself (probably using an
        // associated IRP).
        PtrDeviceObject = NULL;
    } else {
        // Device Object of the driver that set the completion routine.
        // Notice that PtrIrp->Tail.Overlay.CurrentStackLocation was
        // incremented before we use IoGetCurrentIrpStackLocation()
        // here.
        PtrDeviceObject =
                    loGetCurrentlrpStackLocation(Ptrlrp)->DeviceObject;
    }
    PtrContext = currentStackLocation->Context;

    if ((NT_SUCCESS(PtrIrp->IoStatus.Status) &&
            currentStackLocation->Control & SL_INVOKE_ON_SUCCESS)
            II
          (!NT_SUCCESS(PtrIrp->IoStatus.Status) &&
            currentStackLocation->Control & SL_INVOKE_ON_FAILURE)
            II
          (PtrIrp->Cancel && currentStackLocation->Control
                                & SL_INVOKE_ON_CANCEL)) {
            // Invoke the completion routine.
            RC = currentStackLocation->
CompletionRoutine(PtrDeviceObject,PtrIrp, PtrContext);
            if (RC == STATUS_MORE_PROCESSING_REQUIRED) {
                return;
            }
    }

    ...
} // end of while more stack locations to process.
```

Notice that the flag values in the Control field for the current stack location, when combined with state information about why the IRP was completed and the status code saved in the IRP, determine whether or not the I/O Manager will invoke a completion routine for that particular stack location. Also note that the I/O Manager simply starts processing the IRP beginning at the current stack location (i.e., the stack location for the driver that invoked IoCompleteRequest()) and continues on until all stack locations have been processed.

WARNING The I/O Manager is meticulous about supplying the correct device object pointer to the driver that sets a completion routine. If your driver (that must have some device object created to even receive the IRP in the first place) sets a completion routine, then your completion routine will be invoked with a pointer to your own device object. It is possible, however, for your driver to create a new IRP and immediately set a completion routine in the first I/O stack location (to set up for the next driver in the calling hierarchy). In this case, your completion routine will be invoked with the device object pointer set to NULL (since there was no stack location set up for your driver, there is no device object pointer that the I/O Manager can supply to you).

Finally, you may have noticed something strange about the preceding pseudocode fragment. If the completion routine invoked returns a special status code (STATUS_MORE_PROCESSING_REQUIRED), the I/O Manager simply stops postprocessing for the particular IRP and returns control immediately to the caller. This should give you some ideas on how IRPs can be reused by a higher-level driver even after they have been completed by a lower-level kernel-mode driver. This issue is discussed in greater detail later in this chapter.

Be careful about a particular bug that manifests itself when your filter driver specifies a completion routine in an IRP and then forwards the IRP to the next driver in the calling hierarchy. The following code fragment illustrates a methodology used sometimes by higher-level Windows NT drivers that can result in incorrect execution:

```
PtrCurrentloStackLocation = loGetCurrentlrpStackLocation(Ptrlrp);
PtrNextloStackLocation = loGetNextlrpStackLocation(Ptrlrp);
// The following code can cause problems for the driver above
// in the calling hierarchy!!!
*PtrNextIoStackLocation = *PtrCurrentIoStackLocation;
RC = loCallDriver(...);
```

If you examine this code fragment carefully, you will notice that the driver executing this code has literally copied the entire contents of the current I/O stack location into the stack location passed on to the next driver in the calling hierarchy. The copied data includes information contained in the Control field (the SL_INVOKE_XXX flag values), as well as the function pointer and context contained in the CompletionRoutine and Context fields respectively.

The net result is that the completion routine associated with the current I/O stack location is now also associated with the next I/O stack location and will therefore

be invoked twice for the same IRP. Both NTFS and FASTFAT implementations in Windows NT (up until Version 4.0 SP2) contain this bug.*

To protect yourself against such badly behaved drivers, execute the following code sequence in your completion routine:

```
NTSTATUS    SFilterSampleCompletionRoutine (
PDEVICE_OBJECT            PtrSentDeviceObj ect;
PIRP                       Ptrlrp;
PVOID                      SFilterContext)
{
    // Some declarations above.
    PDEVICE_OBJECT         SFilterDeviceObject = NULL;
    ...

    // The following line must exist in all completion routines. Read
    // Chapter 4 for more information.
    if (PtrIrp->PendingReturned) {
        IoMarkIrpPending( Ptrlrp) ;
    }

    //To protect myself from bugs in other drivers ... ! ! !
    // Ensure that you have some way to get a pointer to your device
    // object.
    SFilterDeviceObject = . . . ; // assume that we get the value from
                                  // the context.
    if (PtrSentDeviceObject != SFilterDeviceObject) {
        // We were called erroneously. Return control back to the I/O
        // Manager.
        return;
    }

    // Other processing goes here.
    ...
}
```

Some points to consider regarding completion routines

Completion routines are directly invoked in the context of the thread that calls loCompleteReguest ( ) . Since your driver cannot be sure about the thread execution context in which the completion routine is invoked, it must be especially careful with regard to the memory accessed by the driver or other resources (e.g., pointers, object handles) that may be accessed. Ensure that the processing performed by the completion routine can be executed in any arbitrary thread context.

Also note that completion routines are often invoked at a high IRQL. It is not unusual to have your completion routine invoked at IRQL DISPATCH_LEVEL.

* It appears as though this behavior is exhibited only in the dispatch routine for IRP_MJ_DEVICE_CONTROL as implemented by FASTFAT and NTFS.

Therefore, your completion routine code cannot be made pageable, nor can it access paged memory.

Although you may sometimes be able to get away with invoking IoCallDriver() from within your completion routine, many kernel-mode driver dispatch entry points are not equipped to handle being invoked at a high IRQL. Therefore, try to avoid invoking driver dispatch entry points directly from your completion routine. You could, instead, initiate such processing asynchronously using a worker thread.

WARNING In Chapter 4, we saw how your driver must always propagate the pending returned information from your completion routine. Failure to do this will result in unexpected system behavior, including system hangs and crashes. Review the information provided in Chapter 4 to ensure that your completion routine does behave correctly about propagating such information.

Using completion routines

Filter drivers often use completion routines to perform postprocessing of data returned by the target driver. For example, an encryption module that you may develop could decrypt data on-the-fly in a completion routine after the compressed data has been retrieved by the file system from secondary storage.

There are less esoteric things, as well, that are done using completion routines. For example, an intermediate driver or a file system driver could break up a relatively large I/O request into more manageable pieces and issue multiple I/O requests to lower-level disk drivers. The data returned from the disk drivers can then be collated in a completion routine associated with each IRP sent to the lower-level drivers.

Sometimes FSDs, filter drivers, or fault-tolerant drivers will use completion routines to determine whether a specific I/O request must be reissued to the lower-level driver, in case the I/O failed. This type of retry operation might make sense under certain circumstances.

Subject to the restrictions discussed previously, the kind of processing that you can perform in your completion routine is only limited by your imagination.

About this STATUS_MORE_PROCESSINGJREQUIRED business ...

As you must have observed from the pseudocode fragment presented earlier, when IRP completion postprocessing is being performed by the I/O Manager, the

postprocessing is abruptly terminated if any completion routine invoked returns a special return status of type STATUS_MORE_PROCESSING_REQUIRED.

This is a method provided by the I/O Manager to allow any kernel-mode driver in the calling hierarchy to interrupt the IRP completion. It is possible for the same IRP to be completed, via loCompleteRecjuest ( ), once again at some later time, and the I/O Manager will begin processing (once again) starting at the current I/O stack location.

By returning STATUS_MORE_PROCESSING_REQUIRED, your kernel-mode driver essentially informs the I/O Manager that it needs to hold on to and use the IRP for some additional time. Managing the IRP from that point onward is the responsibility of your driver. This is no different from how your driver would manage an IRP received in a dispatch routine for the very first time. The only point to note is that the kind of processing you can perform directly in your completion routine is limited, since the completion routine is invoked in the context of an arbitrary thread (possibly) at a high IRQL. However, you can certainly dispatch the same IRP to some worker thread for further asynchronous processing.

You should note that the only action performed by the I/O Manager, before it stops the postprocessing of the IRP, is to invoke any completion routines for stack locations lower in the calling hierarchy. Therefore, the IRP state is completely maintained when your completion routine gains control of the IRP. The reason that the I/O Manager terminates processing of the IRP so completely once your driver returns STATUS_MORE_PROCESSING_REQUIRED is because the I/O Manager has no idea what your driver intends to do to the IRP (or has already done to the IRP). Your driver may have just freed the memory allocated for the IRP (by invoking loFreelrp ()) before returning STATUS_MORE_PROCESSING_ REQUIRED to the I/O Manager and hence any attempt by the I/O Manager to even read any field in the IRP structure could lead to a system crash.

Synchronous I/O requests and STATUS_MORE_PROCESSING_REQUIRED

There is one potential problem that you must understand if you expect to return STATUS_MORE_PROCESSING_REQUIRED from a completion routine provided by your driver. Recall from Chapter 4 that the NT I/O Manager tries to optimize processing of user I/O requests that are considered *inherently synchronous.* In the case of these types of I/O requests, the I/O Manager always blocks the invoking thread until the request has been completed via loCompleteRequest ( ). Because of this, the I/O Manager avoids issuing an APC to perform the final post-processing in the context of the thread issuing the I/O request. Instead, the loCompleteRequest () code simply returns control to the caller (after performing some basic postprocessing), and the invoking thread that is blocked,

awaiting completion of the request, performs the final postprocessing by invoking lopCompleteRequest ( ) directly.

For inherently synchronous I/O requests, the I/O Manager code that initially creates an IRP and forwards it onward to the first kernel-mode driver (typically, a filter driver that intercepts FSD requests or the FSD itself) executes the following code sequence:

```
// Invoke the first driver in the calling hierarchy to process the IRP.
RC = loCallDriver(...);
if (RC == STATUS_PENDING) {
    // Wait until the request is completed. The loCompleteRequest()
    // code will now be forced to use a kernel-mode APC to complete
    // the request.
    KeWaitForSingleObject(...);
} else {
    // This request completed synchronously. Therefore, I can safely
    // assume that the IRP is no longer required. Furthermore, the
    // loCompleteRequest() has not issued an APC to perform the final
    // postprocessing. Therefore, let me perform such postprocessing
    // by invoking the appropriate (internal) routine directly.
    lopCompleteRequest(...);
    // Note that the call to lopCompleteRequest() above will result in
    // memory for the IRP being freed.
    }
```

Sometimes, filter drivers that layer themselves above an FSD write code as follows:

```
NTSTATUS SFilterBadFSDInterceptRoutine(
...)
{
    // Assume appropriate declarations, etc.
    ...

    // The filter driver sets a completion routine called
    // SFilterCompletion().
    loSetCompletionRoutine(Ptrlrp, SFilterCompletion,
                                   SFilterCompletionContext    ,
                                   TRUE, TRUE, TRUE);

    // Now, simply dispatch the call and return whatever the FSD returns.
    // The problem with this (described below) is that the FSD may not
    // return STATUS_PENDING. This may cause us headaches later.
    return(loCallDriver(...));
}

NTSTATUS SFilterCompletion(
...)
{
    // Assume appropriate declarations, etc.

    ...
```

```
    // Hardcoded return of STATUS_MORE_PROCESSING_REQUIRED.
    return(STATUS_MORE_PROCESSING_REQUIRED);
}
```

Consider the following situation. The FSD synchronously processes the IRP and returns an appropriate status, either STATUS_SUCCESS or an error (except STATUS_PENDING because, from the FSD's perspective, IRP processing has been completed synchronously). Your filter driver passes the returned status code to the I/O Manager, believing that the completion routine will be able to intercept IRP postprocessing by returning STATUS_MORE_PROCESSING_REQUIRED.

Unfortunately, although your filter driver believes that it has stopped IRP completion postprocessing by returning STATUS_MORE_PROCESSING_REQUIRED from the completion routine, the previous I/O Manager code fragment will not care about the abrupt stoppage of the IRP completion and will invoke IopCompleteRequest() directly, which, in turn, will free the memory for the IRP. This will lead to a system crash (or corruption) when your filter driver continues processing the IRP.

To avoid this problem, you may consider the following guiding principles:*

- If you complete an IRP in your driver synchronously, do not invoke IoMarkIrpPending() and do not return STATUS_PENDING from your dispatch routine.
- If you pass the IRP to a lower layer, protect yourself from the preceding problem by always marking the IRP pending and always returning STATUS_PENDING.

  The other way of protecting yourself is to ensure that if you inadvertently forwarded to the I/O Manager a return code of STATUS_PENDING, you cannot return STATUS_MORE_PROCESSING_REQUIRED from your completion routine (unless you are really sure that this is not an inherently synchronous I/O operation from the I/O Manager's perspective). The next rule formalizes this behavior.
- If you ever return STATUS_PENDING (regardless of whether it is because you decide to return this status code yourself or because some lower-level driver does so and you simply pass-on the return code), you must have marked the IRP pending.
- If you ever mark the IRP pending, you must return STATUS_PENDING.

---

* These ideas/guidelines came out of a discussion on a Usenet newsgroup where this topic was hotly debated. Appendix F, *Additional Sources for Help,* lists some sources for help during FSD or filter driver development, including a Usenet newsgroup.

Here's a simplistic method that can be followed by any filter driver that layers itself on top of an FSD:

```
NTSTATUS SFilterBetterFSDInterceptRoutine(
...)
{
    // Assume appropriate declarations, etc.
    ...

    // The filter driver sets a completion routine called
    // SFilterCompletion().
    IoSetCompletionRoutine(PtrIrp, SFilterCompletion,
                                   SFilterCompletionContext,
                                   TRUE, TRUE, TRUE);

    // Now, invoke the lower-level driver but force synchronous requests to
    // always be completed via an APC.
    loMarklrpPending(Ptrlrp);
    loCallDriver(...)
    return(STATUS_PENDING);
}
```

This code will degrade performance somewhat (though whether such degradation will be noticeable is debatable), but will always lead to correct handling of the IRP, even if your completion routine returns STATUS_MORE_PROCESSING_REQUIRED. As soon as you return STATUS_PENDING (after having marked the IRP pending), the I/O Manager code invoking your driver will wait for the completion of the IRP using the KeWaitForXXX() function, and IRP completion (via `IopCompleteRequest()`) will only be finished when the IRP is finally completed and no completion routine returns STATUS_MORE_PROCESSING_REQUIRED. The downside is that the I/O Manager will be forced to issue an APC to complete the IRP, which incurs a performance penalty even for synchronous I/O requests.

Detaching from a Target Device Object

There will be occasions when your driver may wish to stop filtering and would like to detach itself from the target device object. This may also happen because the target device object could be in the process of being deleted by the driver that created it. An example of this is when file system drivers managing removable media delete a device object representing a mounted instance of a logical volume because the user has replaced the media in the drive. If a user decides to format a disk device, the FSD will dismount the logical volume mounted on the device (if any) at the request of the user application. This will also result in deletion of the logical volume device object and your driver must be prepared to delete the attached device object in this case.

To request that your device object be detached from a target device object, use the IoDetachDevice() function (described in the DDK) supplied by the I/O Manager. This function expects a single argument, the target device object you wish to detach from. You must supply the target device object pointer that you obtained when first attaching to the target.*

Note that your driver must not ever try to detach from a target device object if another driver has layered itself on top of your device object. You can always check for this case by examining the AttachedDevice field in your own device object and declining to detach from the target if the field contents are nonnull. Failure to do this will not only result in memory leaks, but will break the drivers that are layered above yours, since their device objects will also get detached abruptly (without their knowledge or consent). Detaching (when it is not initiated by the I/O Manager) can only be performed safely in a last-in-first-out fashion, starting with the highest-layered device object attached to a specific target device object. The only exception to this rule is when the I/O Manager asks your driver to perform the following detach.

Starting with Version 4.0 of the operating system, the I/O Manager will request that you detach from a target device object if the driver managing the target device object decides to delete the object for some reason. For example, as mentioned earlier, an FSD may dismount a volume if requested to do so by a user application and will therefore delete the device object representing the particular instance of the mounted logical volume.† In this case, the FSD will invoke IoDeleteDevice() to perform the delete operation. In turn, the I/O Manager will ask the first driver that has attached a device object to the one being deleted to detach its own device object. This call will be sent to your driver in the form of a fast I/O function call.‡

Your driver must detach its device object at this time and probably also delete it as well. If you do choose to delete your device object using IoDeleteDevice(), the I/O Manager will now call any driver that has a device object attached to your device object (being deleted) to detach itself from your device object. Note that the fast I/O detach call does not accept failure (there is no return

---

* This is one reason why your driver should always store the target device address in some deviee object extension field. Also note that the address you supply is the address of the highest-layered device object that you received when (for example.) your driver invoked the IoAttachDeviceToDeviceStack() function.

† In Windows NT Version 3.51 and earlier, if the I/O Manager detected a nonnull AttachedDevice field for a device object on which IoDeleteDevice() was invoked, it would bugcheck the system. This was not very conducive to supporting filter drivers cleanly.

‡ Note that I said the first driver will be asked to perform the detach and not the top-layered driver. This is because the I/O Manager expects each driver (starting with the first one) that had attached to the target device object to first detach itself and then invoke IoDeleteDevice() on itself, resulting in a recursive detach for the next (higher-layered) attached device and so on.

value for the function definition), so your driver has no choice but to do the I/O Manager's bidding. If you fail to perform the detach operation, the I/O Manager will bugcheck the system.

TIP Even if multiple drivers have attached themselves to a particular target device object, the method described previously allows each such driver to cleanly detach and delete its own (attached) device object when the target device object is being deleted. To make this happen, however, each driver that has its fast I/O detach function invoked must first perform the detach operation and then immediately perform a delete operation for its device object.

Some Dos and Don'ts in Filtering

Designing filter drivers is often an iterative process. Your filter driver is likely to encounter unique problems and issues that are specific to the type of driver you are trying to design and the type of target driver that your filter will attach itself to. However, there are certain fundamental principles that you should keep in mind when you begin the process of designing and implementing a filter driver. Many of these principles were mentioned earlier in this chapter. Here then, is a recap of some of the basic principles that you should always keep in mind when designing your filter driver:

Always understand the nature of the driver you wish to filter

This may seem obvious to some of you but it cannot be stressed often enough. There are some who believe that using a canned approach to designing filter drivers may be adequate. This may well be the case—sometimes. However, in most cases, if you fail to understand the characteristics of your target driver, you will end up with problems late in the cycle that could be difficult to rectify easily.

As an example, consider the situation where you decide to filter all requests targeted to a specific logical volume managed by a native FSD (e.g., NTFS). You will layer your own device object over the target device object representing the mounted logical volume. So far, so good. However, you should now understand the various ways in which the FSD gets invoked, since those are exactly the situations in which your dispatch entry points will be invoked.

For example, in Chapter 10, *Writing A File System Driver II,* you read that the FSD read/write entry points can be invoked in many different ways: via a system call from a user application, due to a page fault on a mapped file, from the Cache Manager due to asynchronous read/write requests, recursively via the FSD and the Cache Manager due to cached I/O being performed by

the user application, and so on. The important point to note here is that in some situations, it may be acceptable for your filter driver to post an I/O request to be handled asynchronously, but in other situations (e.g., when servicing a page fault), your filter driver should never try to post the request for asynchronous handling, since this could lead to a deadlock or hang. (Remember that the VMM or the Cache Manager may have preacquired FSD resources.)

Similarly, you must be extremely careful about how your filter driver performs synchronization, especially when it filters file system I/O requests. As you read in earlier chapters, the NT VMM and the Cache Manager often preacquire FSD resources via fast I/O calls. This is necessary in order to maintain the system locking hierarchy and avoid deadlock. However, if your filter driver layers itself on top of an FSD, then by definition your filter driver becomes part of the intertwined set of kernel modules that are affected by the locking sequence implemented in processing I/O requests, and therefore, you must somehow ensure that your filter driver does not violate the locking hierarchy in any way. You could do this by ensuring that any resources acquired when processing read/write/create/cleanup/close requests are end-resources (i.e., you would acquire such a resource and not acquire any other until the resource was released; furthermore, for filter drivers, you would not pass-on an IRP unless the acquired resource was released), or you could preacquire resources yourself in the context of the invoking Cache Manager or VMM thread when the fast I/O call is intercepted.*

Note that in the event that your filter driver decides to filter lower-level device objects (e.g., one representing a physical disk device), you might be able to ignore many complicated issues that you would otherwise face when intercepting file system I/O. There are other issues that a filter driver layering itself over a disk driver must be careful of; for example, IRPs sent to a lower-level disk driver may often be associated IRPs created by an FSD, and therefore a filter driver layering itself over the disk driver must not try to create associated IRPs itself. Similarly, lower-level disk drivers are often expected to complete their processing asynchronously by queuing the request and returning control immediately to the higher-level kernel-mode drivers. Your filter driver must conform to such expectations or you could risk destabilizing the entire system.

In all situations, more knowledge and experience about the target driver will prove to be better than less.

---

* The problem with the second approach is that replacing the lazy-write/read-ahead callbacks obtained from the file object could turn out to be very difficult.

Know what your driver attaches itself to

As described earlier in this chapter, your driver may try to attach to a file system (mounted) logical volume device object but may actually end up attaching to the physical disk device object if the volume is not mounted. Therefore, be careful about how you open the target device as a prerequisite to performing the attach operation.

Beware of maintaining unnecessary references

If you inadvertently maintain an extra reference on a target device object (or file object obtained when performing the attach), you may prevent all further open requests to the target and defeat your original goal of intercepting I/O requests (there will be none to intercept). Furthermore, in the case of FSD-created device/file objects, you may end up preventing a volume dismount/lock operation because of such unnecessary references maintained by you and prevent a user from doing useful things like reformatting a drive or ejecting a removable piece of media.

Be careful about the thread context in which your dispatch entry point executes

This is a problem that many of us encounter when beginning to design and implement filter drivers. You may have implemented a kernel-mode filter driver function that executes as follows:

```
NTSTATUS SFilterBadFilterRead(
...)
{
    // Declarations, etc. go here.
    IO_STATUS_BLOCK         LocalIoStatus;
    void                    *LocalBufferPointer;
    ...

    // Here, I will make the caller wait until I read some data
    // from another logical volume. This data will somehow help me in
    // processing the caller's request. ZwReadFileO is easy to use
    // and therefore I will try to use it.
    ZwReadFile(GlobalFileHandle, ...,
                            &LocalIoStatus, LocalBufferPointer, ...);

    ...
}
```

This code fragment seems reasonable, but there are two problems (at least) with it that will result in the ZwReadFile ( ) routine returning an error status back to you. First, the fragment attempts to use a GlobalFileHandle that was presumably obtained when the target of the read operation was first opened (probably during driver initialization). Unfortunately, file handles are process-specific, and therefore, it is highly likely that the previously described file handle will be invalid in the context of most user threads that invoke the system service to request I/O. Instead of using the global file handle directly,

you could create a new file handle in the context of the caller thread (use ObOpenObjectByPointer ( ) described in Chapter 5, *The NT Virtual Memory Manager,* if you had previously stored a pointer to the underlying file object), or you may open a new handle to the target object in the context of the calling thread, or you could post the request to be handled in the context of a thread that can use the handle, or finally, you could avoid using ZwReadFile ( ) and instead create IRPs that you dispatch directly to the target of the request.

Similarly, you will notice that the code attempts to use pointers to memory that is either off the kernel-mode stack or allocated in kernel-mode. When you try to use ZwReadFile ( ) , the recipient of the request will check the previous mode of the caller (which in all likelihood is user mode) and will reject the request if passed-in addresses are kernel-mode virtual addresses (virtual address with a value > Ox7FFFFFF).

The point to note, once again, is that the context of the thread in which your filter driver dispatch routine executes must always be kept in mind as you design and implement your driver.

Be creative

Imagine that you wish to prevent the Windows NT I/O Manager from automatically assigning drive letters during system boot-up to certain drives connected to the system.* How would you go about doing that?

If you go back and read the system boot-up sequence overview described in Chapter 4, you will note that the I/O Manager opens the device object (representing the physical device) created by the disk device driver in order to obtain the characteristics of the device before assigning a drive letter to it. You can then deduce that, perhaps by designing and implementing a filter driver that layers itself, early in the boot sequence, over the target disk device objects, you could somehow prevent this drive letter assignment. How? Maybe, if your driver recognized the I/O Manager open request and failed this open operation for each target disk device object, the I/O Manager may conclude that the disks were unusable and (hopefully) decide not to assign drive letters to these disk drives.t

---

* This may be necessary if you have a large disk farm connected to the system. You know that there are a finite number of drive letters available, and you may decide that one method to allow a user to utilize all of the disks in this large disk farm would be to present logical groupings of disks under a single drive letter. There are other alternatives, as well, that you may decide to implement that are beyond the scope of this book.

t Unfortunately, it is not as easy to prevent drive letter assignment as is described here. But the description provided here is definitely a good starting point.

There are many ways in which filter drivers can *be* used. Not all of these ways have as yet been exploited. Therein lies an opportunity for you to be creative and design stable, safe software modules that extend the capabilities of the Windows NT I/O subsystem and provide substantial added value to your customers.

In this chapter, we discussed filter driver design and development. In order to understand how to design filter drivers that are reliable and useful, you should understand the kernel-mode environment in which your filter driver will execute. The contents of this book and the sample filter driver implementation provided on the accompanying diskette should help you in creatively designing and implementing your own value-added software for the Windows NT operating system.