Handling IRPs: What Every Driver Writer Needs to Know

July 31, 2006

Abstract

This paper presents an overview of the I/O request packet (IRP) mechanism that is used in the Microsoft® Windows® family of operating systems. It is intended to provide driver writers with a greater understanding of how I/O works in the operating system and how their drivers should manage and respond to I/O requests.

This information applies for the following operating systems:

Microsoft Windows Vista™
Microsoft Windows Server® 2003
Microsoft Windows XP
Microsoft Windows 2000

The current version of this paper is maintained at:

http://www.microsoft.com/whdc/driver/kernel/IRPs.mspx

Contents

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2006 Microsoft Corporation. All rights reserved.

Microsoft, Windows, Windows Server, Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Introduction

The Microsoft® Windows® family of operating systems communicates with drivers by sending I/O request packets (IRPs). The data structure that encapsulates the IRP not only describes an I/O request but also maintains information about the status of the request as it passes through the drivers that handle it. Because the data structure serves two purposes, an IRP can be defined as:

- a container for an I/O request

  – or –

- a thread-independent call stack

Considering IRPs from these two perspectives may help driver writers understand what their drivers must do to respond correctly to I/O requests.

For current documentation on routines and issues discussed in this paper, see the most recent version of the Microsoft Windows Driver Kit (WDK).

Definition 1: IRP as a Container for an I/O Request

The operating system presents most I/O requests to drivers using IRPs. IRPs are appropriate for this purpose because:

- IRPs can be processed asynchronously.
- IRPs can be cancelled.
- IRPs are designed for I/O that involves more than one driver.

The IRP data structure packages the information that a driver requires to respond to an I/O request. The request might be from user mode or from kernel mode; regardless of the request's origin, a driver requires the same information.

Every IRP has two parts, shown in Figure 1:

- A header that describes the primary I/O request
- An array of parameters that describe subordinate requests (sometimes called sub-requests)

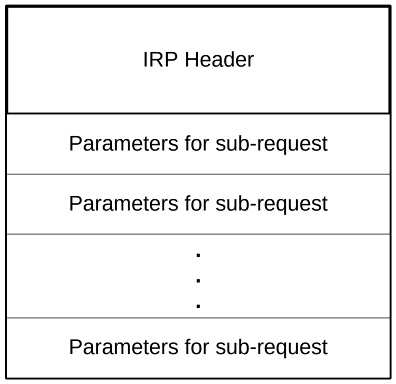
IRP Header
Parameters for sub-request
Parameters for sub-request
.
.
.
Parameters for sub-request

Figure 1. Structure of an IRP

The size of the header is fixed and is the same for every IRP. The size of the array of parameters depends on the number of drivers that will handle the request.

© 2006 Microsoft Corporation. All rights reserved.

Contents of the IRP Header

An IRP is usually handled by a stack of drivers. The header of each IRP contains data that is used by each driver that handles the IRP. While a given driver is handling an IRP, that driver is considered to be the *current owner* of the IRP.

The header of each IRP contains pointers to the following:

- Buffers to read the input and write the output of the IRP
- A memory area for the driver that currently owns the IRP
- A routine, supplied by the driver that currently owns the IRP, which the operating system calls if the IRP is canceled
- The parameters for the current sub-request

In addition to the pointers, the IRP header contains other data that describes the nature and state of the request.

IRP Parameters

Following the IRP header is an array of sub-requests. An IRP can have more than one sub-request because IRPs are usually handled by a stack of drivers. Each IRP is allocated with a fixed number of such sub-requests, usually one for each driver in the device stack. This number typically matches the **StackSize** field of the top device object in the stack, though a driver in the middle of a stack could allocate fewer. If a driver must forward a request to a different device stack, it must allocate a new IRP.

Each sub-request is represented as an I/O stack location (a structure of type IO_STACK_LOCATION), and the IRP typically contains one such I/O stack location for each driver in the device stack to which the IRP is sent. A field in the IRP header identifies the I/O stack location that is currently in use. The value of this field is called the *IRP stack pointer* or the *current stack location*.

The IO_STACK_LOCATION structure includes the following:

- The major and minor function codes for the IRP
- Arguments specific to these codes
- A pointer to the device object for the corresponding driver
- A pointer to an *IoCompletion* routine if the driver has set one
- A pointer to the file object associated with the request
- Various flags and context areas

The IO_STACK_LOCATION does not contain the pointers to the input and output locations; these pointers are in the IRP itself. All the sub-requests operate on the same buffers.

© 2006 Microsoft Corporation. All rights reserved.

Definition 2: IRP as a Thread-Independent Call Stack

Performing an I/O operation typically requires more than one driver for a device. Each driver for a device creates a device object, and these device objects are organized hierarchically into a *device stack*. IRPs are passed down the device stack from one driver to the next. For each driver in the stack, the IRP contains a pointer to an I/O stack location. Because the drivers can handle the requests asynchronously, an IRP is similar to a thread-independent call stack, as Figure 2 shows.

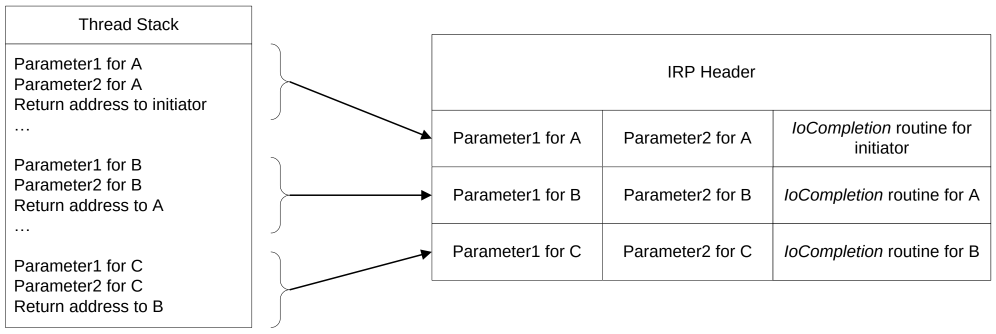

Figure 2. IRP as Thread-independent Call Stack

On the left side of Figure 2, the thread stack shows how the parameters and return address for drivers A, B, and C might be organized into a call stack. On the right, the figure shows how these parameters and return addresses correspond to the I/O stack locations and *IoCompletion* routines in an IRP.

The asynchronous nature of IRP handling is critical to the operating system and the Windows Driver Model (WDM). In a synchronous, single-threaded I/O design, the application that issues a request and each driver through which the request passes must wait until all lower components have completed the request. Such a design uses system resources inefficiently, thus decreasing system performance.

The structure of the IRP provides for an inherently asynchronous design, enabling applications to queue one or more I/O requests without waiting. While the I/O request is in progress, the application thread is free to perform calculations or queue additional I/O requests. Because all the information required to process the request is encapsulated in the IRP, the requesting thread's call stack can be decoupled from the I/O request.

Passing an IRP to the Next Lower Driver

Passing an IRP to the next lower driver (also called *forwarding an IRP*) is the IRP equivalent of a subroutine call. When a driver forwards an IRP, it must populate the next I/O stack location with parameters, advance the IRP stack pointer, and invoke the next driver's dispatch routine. In essence, the driver is *calling down the IRP stack*.

To pass an IRP, a driver typically takes the following steps:

1. Set up the parameters for the next I/O stack location. The driver can either:
   - Call the **IoGetNextIrpStackLocation** routine to get a pointer to the next I/O stack location, and then copy the required parameters to that location.
   - Call the **IoCopyCurrentIrpStackLocationToNext** routine (if the driver sets an *IoCompletion* routine in step 2), or the **IoSkipCurrentIrpStackLocation**

© 2006 Microsoft Corporation. All rights reserved.

routine (if the driver does not set an *IoCompletion* routine in step 2) to pass the same parameters used for the current location.

Note: Drivers must not use the **RtlCopyMemory** routine to copy the current parameters. This routine copies the pointer to the current driver's *IoCompletion* routine, thus causing the *IoCompletion* routine to be called more than once.

2. Set an *IoCompletion* routine for post-processing, if necessary, by calling the **IoSetCompletionRoutine** routine. If the driver sets an *IoCompletion* routine, it must call **IoCopyCurrentIrpStackLocationToNext** in step 1.
3. Pass the request to the next driver by calling the **IoCallDriver** routine. This routine automatically advances the IRP stack pointer and invokes the next driver's dispatch routine.

After a driver passes the IRP to the next driver, it no longer owns the IRP and must not attempt to access it. The IRP could be freed or completed by another driver or on another thread. Attempting to access an IRP in such a situation can cause a system crash. If the driver requires access to the IRP after passing it down the stack, the driver must set an *IoCompletion* routine. When the I/O Manager calls the *IoCompletion* routine, the driver regains ownership of the IRP for the duration of the *IoCompletion* routine. Thus the *IoCompletion* routine can access the fields in the IRP.

If the driver's dispatch routine must also process the IRP after lower drivers have completed it, the *IoCompletion* routine must return STATUS_MORE_PROCESSING_REQUIRED, which returns ownership of the IRP to the dispatch routine. As a result, the I/O Manager stops processing the IRP and leaves the ultimate completion of the IRP to the dispatch routine. The dispatch routine can later call **IoCompleteRequest** to finish completion or can mark the IRP pending for further processing.

Completing an IRP

When I/O is complete, the driver that completed the I/O calls the **IoCompleteRequest** routine. This routine moves the IRP stack pointer to point to the next higher location in the IRP stack, as Figure 3 shows.

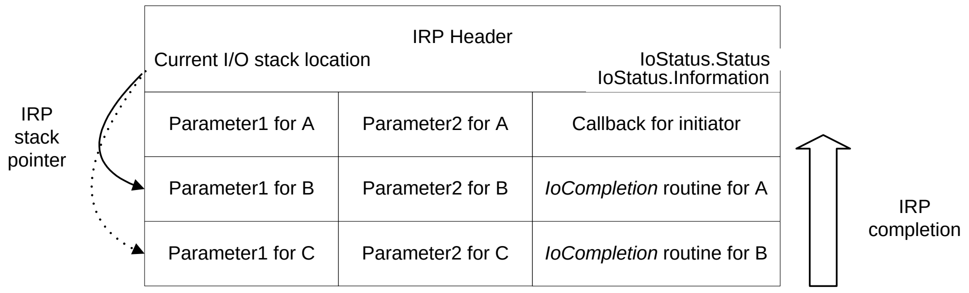

Figure 3. IRP completion and stack pointer

Figure 3 shows the current I/O stack location after driver C has called **IoCompleteRequest**. The solid arrow on the left indicates that the stack pointer now points to the parameters and callback for driver B. The dotted arrow indicates the previous stack location. The hollow arrow on the right indicates the order in which the *IoCompletion* routines are called.

Note: For ease of explanation, this paper shows the I/O stack locations in the IRP "upside-down," that is, in inverted order from A to C instead of from C to A. Using an inverted diagram enables calls that proceed "down" the device stack to point downwards.

© 2006 Microsoft Corporation. All rights reserved.

If a driver set an *IoCompletion* routine as it passed the IRP down the device stack, the I/O Manager calls that routine when the IRP stack pointer once again points to the I/O stack location for the driver. In this way, *IoCompletion* routines act as return addresses for the drivers that handled the IRP as it traversed the device stack.

An *IoCompletion* routine can return either of two status values:

- STATUS_CONTINUE_COMPLETION—continues the upward completion of the IRP. The I/O Manager advances the IRP stack pointer and calls the next-higher driver's *IoCompletion* routine.
- STATUS_MORE_PROCESSING_REQUIRED—stops the upward completion of the IRP and leaves the IRP stack pointer at its current location. Drivers that return this status usually restart the upward completion of the IRP later by calling the **IoCompleteRequest** routine.

When every driver has completed its corresponding sub-request, the I/O request is complete. The I/O Manager retrieves the status of the request from the **Irp->IoStatus.Status** field and retrieves the number of bytes transferred from the **Irp->IoStatus.Information** field.

Synchronous I/O Responses

Although the Windows operating system is designed for asynchronous I/O, most applications issue synchronous I/O requests. Drivers can also issue both synchronous and asynchronous requests and can respond to requests either synchronously or asynchronously.

To determine whether a request was completed synchronously or asynchronously, a driver checks the status returned by **IoCallDriver**, as the following code sample shows:

```
KEVENT   event;

KeInitializeEvent(&event, NotificationEvent, FALSE);

IoCopyCurrentIrpStackLocationToNext(Irp);

IoSetCompletionRoutine(Irp,
                       CatchIrpRoutine,
                       &event,
                       TRUE,
                       TRUE,
                       TRUE
                       );

status = IoCallDriver(DeviceObject, Irp);

//
// Check for synchronous or asynchronous completion.
//

    if (status == STATUS_PENDING) {
    // Code to handle asynchronous response
    // omitted for now.
    }

return status;
```

© 2006 Microsoft Corporation. All rights reserved.

The driver initializes an event, sets the I/O stack location, sets an *IoCompletion* routine, and calls **IoCallDriver** to forward the IRP. The status returned by **IoCallDriver** indicates whether lower drivers are handling the IRP synchronously or asynchronously. If the request is being handled asynchronously, **IoCallDriver** returns STATUS_PENDING. If lower drivers respond synchronously, **IoCallDriver** returns the completion status that was returned by the next lower driver. As the code sample shows, the driver simply returns that same completion status.

When an IRP is completed synchronously, the driver returns the IRP's completion status from its dispatch routine. Drivers above it in the device stack can get the status in either of two ways:

- In the dispatch routine, from the value returned by **IoCallDriver**.
- In the *IoCompletion* routine, from the **IoStatus.Status** field of the IRP.

When the I/O Manager calls the driver's *IoCompletion* routine, the driver owns the IRP and thus can access the **IoStatus.Status** field. If the driver does not set an *IoCompletion* routine, it does not own the IRP after it calls **IoCallDriver** and therefore must not access fields within the IRP.

Figure 4 shows the two ways a driver or application can get the status of an IRP. For ease of explanation, the figure shows the *IoCompletion* routines in the same I/O stack location as the parameters with which they are called, instead of one location lower.

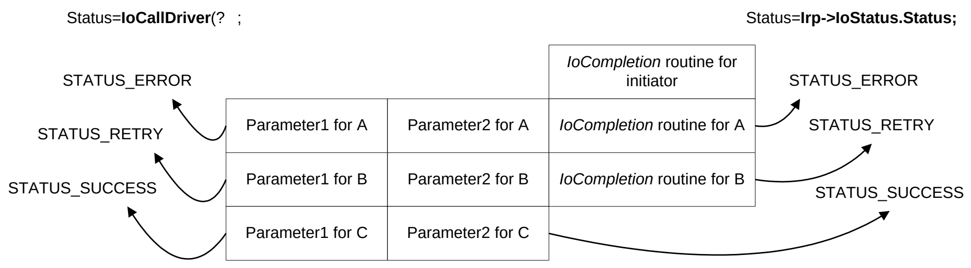

Figure 4. Status returned by IoCallDriver and available to *IoCompletion* routine

On the left side of Figure 4, the **IoCallDriver** routine returns the completion status reported by the next lower driver. On the right, the *IoCompletion* routines read the status from the **IoStatus.Status** field of the IRP. If the IRP completes synchronously, the *IoCompletion* routine for each driver is called before **IoCallDriver** returns, so the status value is available to the *IoCompletion* routine before it is available to the dispatch routine.

Figure 4 shows that driver C returns STATUS_SUCCESS, driver B returns STATUS_RETRY, and driver A returns STATUS_ERROR. The final status of the IRP is available only to the initiator of the request; other drivers can read only the status returned by the next-lower driver.

Asynchronous I/O Responses

A driver should return STATUS_PENDING from a dispatch routine when it cannot complete an I/O request synchronously in a timely manner. Understanding when to return STATUS_PENDING is a problem for many driver writers.

A driver must return STATUS_PENDING if:

- Its dispatch routine for an IRP might return before the IRP is completed.
- It completes the IRP on another thread.

- The dispatch routine cannot determine the IRP's completion status before it returns.

The driver must call the **IoMarkIrpPending** macro before it releases control of the IRP and before it returns STATUS_PENDING. **IoMarkIrpPending** sets the SL_PENDING_RETURNED bit in the **Control** field of the current I/O stack location. Each time an I/O stack location is completed, the I/O Manager copies the value of this bit to the **Irp->PendingReturned** field in the IRP header, as Figure 5 shows.

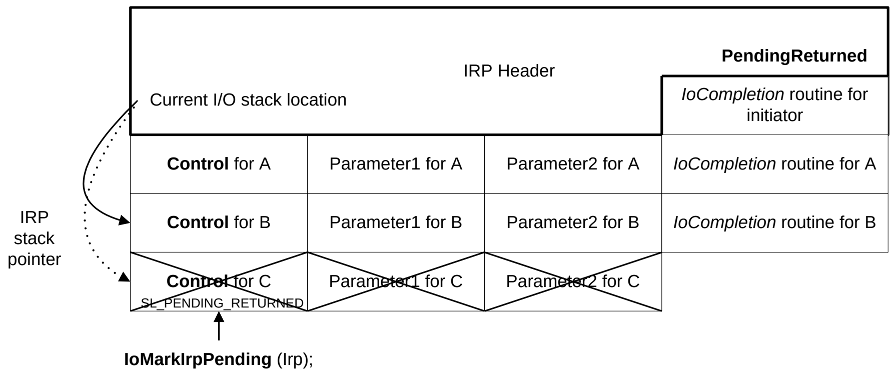

Figure 5. Propagating the pending bit

In Figure 5, Driver C's call to the **IoMarkIrpPending** macro sets the SL_PENDING_RETURNED bit in the **Control** field of Driver C's I/O stack location. When Driver C completes the IRP, the I/O Manager changes the IRP stack pointer to point to driver B and propagates the value of the SL_PENDING_RETURNED bit to the **PendingReturned** field in the IRP header.

IoCompletion Routines and Asynchronous I/O Responses

If a driver sets an *IoCompletion* routine for an IRP, the *IoCompletion* routine can check the value of the **Irp->PendingReturned** field to determine whether the IRP will be completed asynchronously.

If the value of the **Irp->PendingReturned** field is TRUE, the **IoCallDriver** routine will return (or has already returned) STATUS_PENDING. If **Irp->PendingReturned** is FALSE, **IoCallDriver** has already returned with the value in the **Irp->IoStatus.Status** field. *IoCompletion* routines for intermediate drivers can similarly test **Irp->PendingReturned** to determine how the result of their forwarded request is being handled.

Drivers that complete I/O requests asynchronously sometimes must perform additional processing as the IRP moves back up the device stack. The following code sets an *IoCompletion* routine when it forwards an IRP to the next lower driver, then waits on an event. Thus, it handles an asynchronous response in the same manner as a synchronous one:

```
KEVENT   event;

KeInitializeEvent(&event, NotificationEvent, FALSE);

IoCopyCurrentIrpStackLocationToNext(Irp);
```

© 2006 Microsoft Corporation. All rights reserved.

```
IoSetCompletionRoutine(Irp,
                       CatchIrpRoutine,
                       &event,
                       TRUE,
                       TRUE,
                       TRUE
                       );

status = IoCallDriver(DeviceObject, Irp);

//
// Wait for lower drivers to be done with the Irp.
// It’s important to note here that when you allocate
// memory for an event on the stack you must do a
// KernelMode wait instead of UserMode to prevent
// the stack from being paged out.
//

    if (status == STATUS_PENDING) {
       status = KeWaitForSingleObject(&event,
                                      Executive,
                                      KernelMode,
                                      FALSE,
                                      NULL
                                      );
       ASSERT(NT_SUCCESS(status));
       status = Irp->IoStatus.Status;
    }

return status;
```

In this case, the driver waits for an event that is set by its *IoCompletion* routine. The driver must wait in kernel mode because the event was allocated on the stack. For performance reasons, drivers should wait on events only when IRPs complete asynchronously. The **KeWaitForSingleObject** routine uses the system-wide dispatcher lock. This lock protects the signal state of events, semaphores, and mutexes, and consequently is used frequently throughout the operating system. Requiring the use of this lock for every synchronous I/O operation would unacceptably hinder performance.

The following code shows the *IoCompletion* routine set in the preceding fragment:

```
NTSTATUS
CatchIrpRoutine(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP       Irp,
    IN PKEVENT    Event
    )
{
    if (Irp→PendingReturned) {

        // Release waiting thread
        KeSetEvent( Event, IO_NO_INCREMENT, FALSE );
    }

    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

© 2006 Microsoft Corporation. All rights reserved.

The *IoCompletion* routine tests the value of the **Irp->PendingReturned** field. Based on this value, it sets an event only if STATUS_PENDING has been or will be returned to the caller. This avoids a call to the **KeSetEvent** routine, which uses the system-wide dispatcher lock.

The *IoCompletion* routine returns STATUS_MORE_PROCESSING_REQUIRED. This status indicates to the I/O Manager that the current driver must perform additional processing while it owns the IRP. The I/O Manager stops the upward completion of the IRP, leaving the I/O stack location in its current position. The current driver still owns the IRP and can continue to process it in another routine. When the driver has completely processed the IRP, it should call the **IoCompleteRequest** routine to continue IRP completion.

Ownership of the IRP is important because it determines whether the driver can access the IRP. As soon as a driver relinquishes ownership of the IRP, the IRP can be completed or freed on another thread, and any attempt by the driver to access the IRP can result in a system crash.

Propagating the Pending Bit

Any time a driver handling an I/O request returns the response of the next-lower driver, the value of the pending bit in its I/O stack location (SL_PENDING_RETURNED in the **Control** field of the IO_STACK_LOCATION structure) must be the same as that of the next-lower driver. If the driver does not set an *IoCompletion* routine, the I/O Manager automatically propagates the value of the bit, freeing the driver of this responsibility. However, if the driver sets an *IoCompletion* routine, and the next-lower driver returns STATUS_PENDING, the current driver must mark its own I/O stack location as pending. For example:

```
// Forward request to next driver
IoCopyCurrentIrpStackLocationToNext( Irp );

// Send the IRP down
status = IoCallDriver( nextDevice, Irp );

// Return the lower driver’s status
return status;
```

Because this example does not set an *IoCompletion* routine, the driver must not call the **IoMarkIrpPending** macro. The driver simply returns the same status as the next-lower driver and the I/O Manager copies the value of the pending bit.

However, if the driver sets an *IoCompletion* routine, this code is insufficient for situations in which the lower driver returns STATUS_PENDING. In such situations, the driver must call the **IoMarkIrpPending** macro to set the SL_PENDING_RETURNED bit for its own I/O stack location. The driver must call **IoMarkIrpPending** from an *IoCompletion* routine; it must not make this call from its dispatch routine. Thus, the following code is incorrect:

```
// Forward request to next driver
IoCopyCurrentIrpStackLocationToNext( Irp );

// Send the IRP down
status = IoCallDriver( nextDevice, Irp );
```

© 2006 Microsoft Corporation. All rights reserved.

```
// The following is an error because this driver
// no longer owns the IRP.
If (status == STATUS_PENDING) {

    IoMarkIrpPending( Irp );
}

// Return the lower driver’s status
return status;
```

This approach is incorrect because **IoMarkIrpPending** operates on the current I/O stack location. After **IoCallDriver** passes the IRP to the next lower driver, the current driver no longer owns the IRP. Thus, when the call to **IoMarkIrpPending** is executed, there is no current I/O stack location; in fact, if a lower driver completed the IRP, then the pointer is no longer valid. To avoid this problem, the driver must call **IoMarkIrpPending** from an *IoCompletion* routine. For example:

```
NTSTATUS
CompletionRoutine( ... )
{
   if (Irp→PendingReturned) {

      // Return the lower driver’s result. If the
      // lower driver marked its stack location pending,
      // so do we.
      IoMarkIrpPending( Irp );
   }

… //additional processing in IoCompletion routine

   return STATUS_CONTINUE_COMPLETION;
}
```

Drivers should use this code sequence only when returning the same status as the lower driver. If the driver does not set an *IoCompletion* routine, the I/O Manager automatically propagates the value of the SL_PENDING_RETURNED bit upwards to the next I/O stack location. A driver is not required to use an *IoCompletion* routine simply to call **IoMarkIrpPending**; but if a driver does have an *IoCompletion* routine, and a lower driver returns STATUS_PENDING, the *IoCompletion* routine must call **IoMarkIrpPending**.

Summary of Guidelines for Pending IRPs

Driver writers must follow certain guidelines when handling IRPs for which STATUS_PENDING can be returned. Ignoring these guidelines may cause post-processing to occur twice, resulting in a system crash, or prevent post-processing from occurring, resulting in a system hang.

The following are the fundamental guidelines for returning STATUS_PENDING:

- If a driver returns STATUS_PENDING, it must first call the **IoMarkIrpPending** macro to mark the I/O stack location as *pending*.
- Conversely, if a driver calls **IoMarkIrpPending**, it must return STATUS_PENDING.

© 2006 Microsoft Corporation. All rights reserved.

In addition:

- If a driver returns the same status as the next lower driver and sets an *IoCompletion* routine, the *IoCompletion* routine must call **IoMarkIrpPending** if the value of the **Irp->PendingReturned** field is TRUE.
- If a driver completes an I/O request on a different thread from that on which it received the request, its dispatch routine or *IoCompletion* routine must call **IoMarkIrpPending** and its dispatch routine must return STATUS_PENDING.

Optimizations

By testing the value of the **Irp->PendingReturned** field, a driver can take advantage of the pending bit to optimize post-processing work for an I/O request. For example, a driver can use the processor cache efficiently, thus improving throughput, by post-processing the IRP after the **IoCallDriver** routine returns a synchronous response. The logic of such an optimization follows:

- In a synchronous I/O response, the thread that initiated the I/O request can perform post-processing if **IoCallDriver** does not return STATUS_PENDING. In this case, the IRP is complete when **IoCallDriver** returns, so the dispatch routine can perform any required processing.
- In an asynchronous I/O response, the *IoCompletion* routine should perform the post-processing if **IoCallDriver** returns STATUS_PENDING. The *IoCompletion* routine must test the value of **Irp->PendingReturned**, as described in **"**IoCompletion Routines and Asynchronous I/O Responses" earlier in this paper. If the value of **Irp->PendingReturned** is TRUE, the *IoCompletion* routine performs the required post-processing.

The operating system uses this exact technique for read requests, write requests, and some I/O control codes (IOCTLs). Consequently, if a driver fails to follow the guidelines in "Summary of Guidelines for Pending IRPs" earlier in this paper, the operating system will perform post-processing twice or not at all.

Life Cycle of a File Object

A file object is created each time a device is opened. Each file object represents a single use of an individual file, and maintains state information for that use (such as current file offset). When the I/O Manager creates and opens a file object, it creates a handle to refer to the object. The IRP_MJ_CREATE, IRP_MJ_CLEANUP, and IRP_MJ_CLOSE requests define the life cycle of a file object.

IRP_MJ_CREATE Requests

An IRP_MJ_CREATE request notifies the driver that a new file object has been created. The I/O Manager typically sends an IRP_MJ_CREATE request when a user-mode application calls the **CreateFile** function or a kernel-mode driver calls the **ZwCreateFile** routine.

In an IRP_MJ_CREATE request, the current I/O stack location contains a **FileObject** structure, which identifies the file object to open. The **FileObject** structure specifies:

- The name of the file to open in the **FileObject->FileName** field.
- Two pointers for driver use in the **FileObject->FsContext** and **FileObject->FsContext2** fields.

© 2006 Microsoft Corporation. All rights reserved.

Note: In a WDM device stack, only the functional device object (FDO) can use the two context pointers. File system drivers use special functions to share these fields among multiple drivers.

IRP_MJ_CREATE requests arrive in the context of the thread and the process that opened or created the file. A driver can record per-caller information, if required.

In response to a create request, the operating system checks access rights as follows:

- If the request does not include a file name, the operating system checks the rights against the ACL for the device object opened by name.
- If the request includes a file name, the operating system checks security only if the FILE_DEVICE_SECURE_OPEN characteristic is set in the **DeviceObject->Characteristics** field. Otherwise, the driver is responsible for checking security.

If the request succeeds, the operating system saves the granted rights in the handle to the object for use in subsequent I/O requests. For detailed information on driver security, see the WDK.

IRP_MJ_CLEANUP Requests

An IRP_MJ_CLEANUP request notifies a driver that the last handle to a file object has been closed. This request does not indicate that the file object has been deleted.

The I/O Manager sends an IRP_MJ_CLEANUP request for a file object after the last handle to the object is closed. When a driver receives an IRP_MJ_CLEANUP request, the driver must complete any pending IRPs for the specified file object. After the IRPs have been completed, the I/O Manager destroys the file object. While the IRPs are pending, the I/O Manager cannot delete the object.

IRP_MJ_CLEANUP requests arrive in the context of the caller that attempts to close the last handle. If the driver recorded any information about the caller when the IRP_MJ_CREATE request arrived, it should release that information when it processes the IRP_MJ_CLEANUP request. However, if the file handle has been duplicated, the context for the IRP_MJ_CLEANUP request might not be the same as that for the corresponding IRP_MJ_CREATE request. Consequently, driver writers must be careful to determine the appropriate context when using process information during cleanup operations.

For example, an application can call the **DuplicateHandle()** function to duplicate a file handle into another process. If the new handle is closed last, after all handles in the original process, the driver will receive an IRP_MJ_CLEANUP request on a handle in the new process context. The context still represents the caller, but it is not the same as the original caller that created the file object.

IRP_MJ_CLOSE Requests

An IRP_MJ_CLOSE request notifies a driver that a file object has been deleted.

The I/O Manager sends an IRP_MJ_CLOSE request for a file object when both of the following are true:

- All handles to the file object are closed.
- No outstanding references to the object, such as those caused by a pending IRP, remain.

© 2006 Microsoft Corporation. All rights reserved.

Unlike IRP_MJ_CREATE and IRP_MJ_CLEANUP requests, an IRP_MJ_CLOSE request does not arrive in the context of the caller.

In response to an IRP_MJ_CLOSE request, a driver should reverse whatever actions it performed in response to the corresponding IRP_MJ_CREATE request. Additional tasks depend on the design of the driver and the type of hardware it supports.

Data Transfer Mechanisms

The Windows family of operating systems supports three data transfer mechanisms:

- Buffered I/O operates on a kernel-mode copy of the user's data.
- Direct I/O directly accesses the user's data through Memory Descriptor Lists (MDLs) and kernel-mode pointers.
- Method neither I/O (neither buffered nor direct I/O) accesses the user's data through user-mode pointers.

For standard I/O requests, such as IRP_MJ_READ and IRP_MJ_WRITE, drivers specify which transfer mechanism they support by modifying the value of the **DeviceObject->Flags** field soon after creating the device.

Buffered I/O

To receive read and write requests as buffered I/O, the driver sets the DO_BUFFERED_IO flag in the **DeviceObject->Flags** field during initialization. When a driver receives a buffered I/O request, the **Irp->AssociatedIrp.SystemBuffer** field contains the address of the kernel-mode buffer on which the driver should operate. The I/O Manager copies data from the kernel-mode buffer to the user-mode buffer during a read request, or from the user-mode buffer to the kernel-mode buffer during a write request.

Direct I/O

To receive read and write requests as direct I/O, the driver sets the DO_DIRECT_IO flag in the **DeviceObject->Flags** field during initialization. When a driver receives a direct I/O request, the **Irp->MdlAddress** field contains the address of an MDL that describes the request buffer. The MDL lists the buffer's virtual address and size, along with the physical pages in the buffer. The I/O Manager locks these physical pages before issuing the request to the driver and unlocks them during completion. The driver must not use the user-mode buffer address specified in the MDL; instead, it must get a kernel-mode address by calling the **MmGetSystemAddressForMdlSafe** macro.

Neither Buffered nor Direct I/O

To receive neither buffered nor direct I/O requests, the driver sets neither the DO_BUFFERED_IO flag nor the DO_DIRECT_IO flag in the **DeviceObject->Flags** field. When a driver receives such a request, the **Irp->UserBuffer** field contains the address of the data pertaining to the request. Because this buffer is in the user address space, the driver must validate the address before using it. To validate the pointer, a driver calls the **ProbeForRead** or **ProbeForWrite** function within a **try/except** block. The driver must also perform all access to the buffer within a **try/except** block.

© 2006 Microsoft Corporation. All rights reserved.

In addition, the driver must copy the data to a safe kernel-mode address in the pool or on the stack before manipulating it. Copying the data to a kernel-mode buffer ensures that the user-mode caller cannot change the data after the driver has validated it.

Note: For detailed information on probing and on problems commonly seen in driver I/O paths, see the white paper "Common Driver Reliability Issues."

I/O Control Codes (IOCTLs)

The I/O Manager sends an I/O control code (IOCTL) as part of the IRP for requests other than read or write requests. An IOCTL is a 32-bit control code that identifies an I/O or device operation. Requests that specify IOCTLs can have both input and output buffers.

The operating system supports two types of IOCTLs, which are sent in two different IRPs:

- IRP_MJ_DEVICE_CONTROL requests can be sent from user mode or kernel mode. These requests are sometimes called public IOCTLs.
- IRP_MJ_INTERNAL_DEVICE_CONTROL requests can be sent by kernel-mode components only. These requests are typically used for driver-to-driver communication and are sometimes called private IOCTLs.

For an IOCTL, the transfer mechanism is specified in the **Method** field of the control code. IOCTLs support the following transfer mechanisms:

- METHOD_BUFFERED
- METHOD_OUT_DIRECT
- METHOD_IN_DIRECT
- METHOD_NEITHER

METHOD_BUFFERED IOCTLs

In a METHOD_BUFFERED IOCTL, like a buffered read or write request, data transfer is performed through a copy of the user's buffer passed in the **Irp->AssociatedIrp.SystemBuffer** field. The lengths of the input and output buffers are passed in the driver's IO_STACK_LOCATION structure in the **Parameters.DeviceIoControl.InputBufferLength** field and the **Parameters.DeviceIoControl.OutputBufferLength** field. These values represent the maximum number of bytes the driver should read or write in response to the buffered IOCTL.

METHOD_BUFFERED IOCTLs are the most secure IOCTLs because the buffer pointer is guaranteed to be valid and aligned on a natural processor boundary, and the data in the buffer cannot change.

The I/O Manager does not zero-initialize the output buffer before issuing the request. The driver is responsible for writing either valid data or zeroes in the output buffer up to the return byte count it specifies in the **Irp->IoStatus.Information** field. Failing to write valid data or zeroes could result in returning private kernel data to the user-mode application. Because this data could belong to another user, this error is considered a breach of system security.

© 2006 Microsoft Corporation. All rights reserved.

METHOD_OUT_DIRECT IOCTLs

An IOCTL that specifies METHOD_OUT_DIRECT or METHOD_DIRECT_FROM_HARDWARE represents a read operation from the hardware. METHOD_OUT_DIRECT and METHOD_DIRECT_FROM_HARDWARE can be used interchangeably.

In METHOD_OUT_DIRECT requests, the **Irp->AssociatedIrp.SystemBuffer** field contains a kernel-mode copy of the requestor's input buffer. The **Irp->MdlAddress** field contains an MDL that describes the requestor's output buffer. The I/O Manager readies this buffer for the driver to write. As in read and write operations, the driver must call the **MmGetSystemAddressForMdlSafe** macro to get a kernel-mode pointer to the buffer described by the MDL.

The requestor's input buffer typically contains a pointer to a command that the driver should interpret or send to the device. The requestor's output buffer typically is the location to which the driver should transfer the result of the operation.

METHOD_IN_DIRECT IOCTLs

An IOCTL that specifies METHOD_IN_DIRECT or METHOD_DIRECT_TO_HARDWARE requests a write operation to the hardware. METHOD_DIRECT_TO_HARDWARE and METHOD_IN_DIRECT can be used interchangeably.

In METHOD_IN_DIRECT requests, the **Irp->AssociatedIrp.SystemBuffer** field contains a kernel-mode copy of the requestor's input buffer. The **Irp->MdlAddress** field contains an MDL that describes the requestor's output buffer. The I/O Manager readies this buffer for the driver to read. As in read and write operations, the driver must call the **MmGetSystemAddressForMdlSafe** macro to get a kernel-mode pointer to the buffer described by the MDL.

The input and output buffers are typically used in similar ways for METHOD_OUT_DIRECT and METHOD_IN_DIRECT IOCTLs. The requestor's input buffer contains a command for the driver or device. However, the requestor's output buffer contains the data for the driver to transfer *to* the device. In effect, it is a second input buffer.

METHOD_NEITHER IOCTLs

A driver can define IOCTLs that use neither direct nor buffered I/O. METHOD_NEITHER IOCTLs have separate user-mode pointers for input and output buffers:

- **IrpSp->Parameters.DeviceIoControl.Type3InputBuffer** points to the input buffer.
- **Irp->UserBuffer** points to the output buffer.

The input and output buffer addresses are user-mode pointers. Therefore, drivers must validate these pointers before using them, by calling the **ProbeForRead** and **ProbeForWrite** routines within a **try/except** block. In addition, the driver must copy all parameters to kernel-mode memory (either in the pool or on the stack) before validating them.

Note: For detailed information on probing and on problems commonly seen in driver I/O paths, see the white paper "Common Driver Reliability Issues."

© 2006 Microsoft Corporation. All rights reserved.

Success, Error, and Warning Status for IRP Completion

When a driver completes an IRP with a success or warning status, the I/O Manager:

- Copies the data from the buffer specified in the IRP back to the user’s buffer, if the IRP specified METHOD_BUFFERED I/O. The driver specifies the number of bytes to be copied in the **Irp->IoStatus.Information** field.
- Copies the results from the IRP’s status block (**Irp->IoStatus.Status** and **Irp->IoStatus.Information**) to the caller’s original request block.
- Signals the event specified by the caller that initiated the request.

Not all of these actions occur in all cases. For example, if a driver completes an IRP with an error status, the I/O Manager does not copy any data back to the user’s buffer, and it copies only the value of the **Irp->IoStatus.Status** field, not the value of the **IoStatus.Information** field. If a driver completes an IRP synchronously, the I/O Manager does not signal the event. If a driver completes an IRP asynchronously, the I/O Manager might or might not signal the event. Therefore, a driver that calls **ZwReadFile**, **ZwDeviceIoControl**, or similar **Zw*Xxx*** routines should wait on the event only if the **Zw*Xxx*** routine returns STATUS_PENDING.

The following values indicate error and warning status codes:

- NTSTATUS codes 0xC0000000 - 0xFFFFFFFF are errors.
- NTSTATUS codes 0x80000000 - 0xBFFFFFFF are warnings.

If the value of the **Irp->IoStatus.Status** field is an error code, the operating system does not return any data, so the contents of the **Irp->IoStatus.Information** field should always be zero. If the value of the **Irp->IoStatus.Status** field is a warning code, the operating system can return data, so the contents of the **Irp->IoStatus.Information** field can be nonzero.

A simple scenario can help explain this situation. Assume that an IRP requires a driver to return data in a buffer that is defined with the following two fields:

```
ULONG Length;
UCHAR Data [];  // Variable length array
```

The *Length* field specifies the size required to retrieve the data. The application sends a ULONG request to get the *Length*, and then sends a second request with a bigger buffer to retrieve all the data. The driver, in turn, always expects the buffer to be at least the size of a ULONG data item.

If the buffer is large enough, the driver completes the IRP with STATUS_SUCCESS, and *Length* and **Irp->IoStatus.Information** receive the number of bytes transferred.

If the buffer is not large enough to hold the data, the driver completes the IRP with the warning STATUS_BUFFER_OVERFLOW. In this case, the data is too large for the buffer. The driver updates the *Length* with the size required and writes **sizeof**(ULONG) into **Irp->IoStatus.Information**.

If the buffer is too small to write the required length (that is, the buffer is smaller than **sizeof**(ULONG)), the driver completes the IRP with the error STATUS_BUFFER_TOO_SMALL and sets **Irp->IoStatus.Information** to 0.

© 2006 Microsoft Corporation. All rights reserved.

Building IRPs

Drivers can create two types of IRPs:

- Threaded IRPs, also called synchronous requests.
- Nonthreaded IRPs, also called asynchronous requests.

Threaded IRPs

Threaded IRPs are bound to the current thread of execution when they are created. When the thread terminates, any threaded IRPs associated with it are automatically canceled.

Drivers create threaded IRPs by calling one of the following routines:

- **IoBuildSynchronousFsdRequest**
- **IoBuildDeviceIoControlRequest**

IoBuildSynchronousFsdRequest allocates and builds a threaded IRP for a read or write request. **IoBuildDeviceIoControlRequest** allocates and builds a threaded IRP to send an IOCTL.

When creating a threaded IRP, a driver specifies an event to signal when the IRP is complete and provides pointers to the buffers in which to return the requested data. When every sub-request in the IRP is complete (that is, when all the required drivers have completed the IRP), the entire request is complete. The I/O Manager then performs post-processing as follows:

- Copies kernel-mode data to user-mode buffers to be returned to the caller. For example, **ZwReadFile**, **IoBuildDeviceIoControlRequest**, and similar system support routines pass user-mode buffers.
- Signals the event, if the response is asynchronous. If the **IoCallDriver** routine returns STATUS_PENDING, the caller should wait on the event.
- Copies the value of the **Irp->IoStatus.Status** field to the **Irp->UserIosb.Status** field of the user I/O status block, if the response is asynchronous. If **IoCallDriver** returns STATUS_PENDING, the caller can read this field after the event is signaled.
- Copies the value of the **Irp->IoStatus.Information** field to the **Irp->UserIosb.Information** field of the user I/O status block, if the response is asynchronous and the I/O request succeeded. If **IoCallDriver** returns STATUS_PENDING, the caller can read this field after the event is signaled. If the request returns an error status, the I/O Manager does not transfer any data to the user buffers, so the caller should treat the value at **Irp->UserIosb.Information** as zero; some driver writers might prefer to zero-initialize the field as an added precaution.
- Frees the IRP.

Nonthreaded IRPs

Nonthreaded IRPs are not associated with any thread. The driver that initiates a nonthreaded IRP must set a completion routine to “catch” the IRP when it is complete. The I/O Manager does not free nonthreaded IRPs; the driver that initiated the IRP must free it. Nonthreaded IRPs are intended for driver-to-driver communication.

© 2006 Microsoft Corporation. All rights reserved.

Drivers create nonthreaded IRPs by calling one of the following routines:

- **IoBuildAsynchronousFsdRequest**
- **IoAllocateIrp**

IoBuildAsynchronousFsdRequest allocates and builds a nonthreaded IRP for a read or write request. **IoAllocateIrp** allocates an IRP for a driver to send to lower drivers in the same device stack or to another device stack.

IRP Cancellation

IRPs can be canceled (sometimes described as “recalled”). When an IRP is canceled, the driver that currently owns the IRP must complete it immediately.

A driver that requires special processing to cancel an IRP must either:

- Supply an *IoCancel* routine to be notified when cancellation occurs and test the **Cancel** flag in the IRP at various times during processing to determine whether the IRP has been canceled.
- Use the new cancel-safe IRP queuing routines (**IoCsq*Xxx***).

IRP cancellation can be difficult to code correctly because cancellation is inherently asynchronous and race conditions can occur at numerous points. The cancel-safe IRP queues greatly simplify cancellation logic and are strongly recommended for all new drivers. IRP cancellation is covered in the WDK and in the white paper titled “Cancel Logic in Windows Drivers.”

Debugging I/O Problems

Driver writers can use the Driver Verifier and extensions to the Microsoft debuggers to debug problems in handling IRPs.

Driver Verifier can catch errors in every aspect of IRP handling. Driver Verifier is provided with every version of the operating system. Driver verifier runs best with a debugger attached. However, all driver writers must still provide and use test tools that exercise their driver in order for Driver Verifier to detect errors. Use of the Driver Verifier and driver-specific test tools should be a standard part of debugging and development.

The **!irp** and **!irpfind** debugger extensions can help in tracing IRPs while debugging. The **!irp** extension displays detailed information about a specified IRP, and the **!irpfind** extension displays information about all IRPs in the system or about one or more IRPs that meet specified criteria. For detailed information about these extensions, see the documentation in the Debugging Tools for Windows package. You can download this package from the WHDC web site, at the location listed in the Call to Action and Resources section of this paper.

Call to Action and Resources

Call to action for driver developers:

- Return STATUS_CONTINUE_COMPLETION or STATUS_MORE_PROCESSING_REQUIRED from all *IoCompletion* routines.
- Understand when to return STATUS_PENDING for an IRP. Return STATUS_PENDING only if you have called the **IoMarkIrpPending** macro for the IRP, and conversely, call **IoMarkIrpPending** for an IRP only if you will return STATUS_PENDING.

© 2006 Microsoft Corporation. All rights reserved.

- Test the value of the **Irp->PendingReturned** flag in *IoCompletion* routines to optimize post-processing of IRPs.
- Understand the difference between error status and warning status, and what data the IRP contains for each type of status.
- Use Driver Verifier to catch errors in IRP handling, and use debugger extensions to understand how individual IRPs are processed.

Resources:

WHDC Web site
http://www.microsoft.com/whdc/default.mspx

Current White Papers
I/O Completion/Cancellation Guidelines
http://www.microsoft.com/whdc/driver/kernel/Iocancel.mspx

Cancel Logic in Windows Drivers
http://www.microsoft.com/whdc/driver/kernel/cancel_logic.mspx

Scheduling, Thread Context, and IRQL
http://www.microsoft.com/whdc/driver/kernel/IRQL.mspx

Locks, Deadlocks, and Synchronization
http://www.microsoft.com/whdc/driver/kernel/locks.mspx

Common Driver Reliability Issues
http://www.microsoft.com/whdc/driver/security/drvqa.mspx

Debugging Tools for Windows
http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx

Microsoft Knowledge Base
http://support.microsoft.com/

Knowledge Base Article 320275, “Different Ways of Handling IRPs—Cheat Sheet (Part 1 of 2)”

Knowledge Base Article 326315, “Different Ways of Handling IRPs—Cheat Sheet (Part 2 of 2)”

Microsoft Windows Driver Kit (WDK)
http://www.microsoft.com/whdc/driver/WDK/default.mspx

Microsoft Platform Software Development Kit (SDK)
http://www.microsoft.com/downloads/details.aspx?FamilyId=484269E2-3B89-47E3-8EB7-1F2BE6D7123A&displaylang=en

© 2006 Microsoft Corporation. All rights reserved.