Different ways of handling IRPs - cheat sheet (part 1 of 2)

This article was previously published under Q320275

Forwarding IRPs to a lower driver from the dispatch routine

This article examines the following scenarios:

SUMMARY

Introduction

One of the most frequently done tasks in Windows Driver Model (WDM) drivers is sending input/output request packets (IRPs) from one driver to another driver. A driver either creates its own IRP and sends it to a lower driver, or the driver forwards the IRPs that it receives from another driver that is attached above.

This article discusses all the possible ways that a driver can send IRPs to a lower driver with annotated sample code. Depending on the need, driver writers can follow one of the templates given in this article and not be affected by old IRP handling rules.

Part 1 of this subject shows 5 scenarios about how to forward an IRP to another driver from a dispatch routine, and the remaining 7 scenarios (listed in part 2 of this subject) discuss different ways of creating an IRP and sending it to another driver. Part 2 of this subject is contained in the following Knowledge Base article:

> 326315 Different ways of handling IRPs - cheat sheet (part 2 of 2)

Before you examine the various scenarios, note the following about the STATUS that is returned by completion routines:

> An IRP completion routine can return either STATUS_MORE_PROCESSING_REQUIRED or STATUS_SUCCESS.

The I/O manager uses the following rules when it examines the status:

- If the status is STATUS_MORE_PROCESSING_REQUIRED, stop completing the IRP, leave the stack location unchanged and return.

- If the status is anything other than STATUS_MORE_PROCESSING_REQUIRED, continue completing the IRP upward.

Because the I/O Manager does not have to know which non-STATUS_MORE_PROCESSING_REQUIRED value is used, use STATUS_SUCCESS (because the value 0 is efficiently loadable on most processor architectures).

To improve the readability of the code, Windows XP SP1 and Windows XP .NET Driver Development Kit Ntddk.h and Wdm.h header files will have a new **#define** that is named STATUS_CONTINUE_COMPLETION, which is aliased to STATUS_SUCCESS as shown in the following code:

```
//
// This value should be returned from completion routines to continue
// completing the IRP upwards. Otherwise, STATUS_MORE_PROCESSING_REQUIR
ED
// should be returned.
//
#define STATUS_CONTINUE_COMPLETION      STATUS_SUCCESS
//
```

```
// Completion routines can also use this enumeration instead of status
codes.
//
typedef enum _IO_COMPLETION_ROUTINE_RESULT {

    ContinueCompletion = STATUS_CONTINUE_COMPLETION,
    StopCompletion = STATUS_MORE_PROCESSING_REQUIRED

} IO_COMPLETION_ROUTINE_RESULT, *PIO_COMPLETION_ROUTINE_RESULT;
```

back to the top

MORE INFORMATION

Scenario 1: Forward and forget

Use the following code if a driver just wants to forward the IRP down and take no additional action. The driver does not have to set a completion routine in this case. If the driver is a top level driver, the IRP can be completed synchronously or asynchronously, depending on the status that is returned by the lower driver.

```
NTSTATUS
DispatchRoutine_1(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    //
    // You are not setting a completion routine, so just skip the stack
    // location because it provides better performance.
    //
    IoSkipCurrentIrpStackLocation (Irp);
    return IoCallDriver(TopOfDeviceStack, Irp);
}
```

back to the top

Scenario 2: Forward and wait

Use the following code if a driver wants to forward the IRP to a lower driver and wait for it to return so that it can process the IRP. This is frequently done when handling PNP IRPs. For example, when you receive a IRP_MN_START_DEVICE IRP, you must forward the IRP down to the bus driver and wait for it to complete before you can start your device. The Windows XP system has a new function named **IoForwardIrpSynchronously** that you can use to do this operation easily.

```
NTSTATUS
DispatchRoutine_2(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    KEVENT   event;
    NTSTATUS status;

    KeInitializeEvent(&event, NotificationEvent, FALSE);

    //
    // You are setting completion routine, so you must copy
    // current stack location to the next. You cannot skip a location
    // here.
    //
    IoCopyCurrentIrpStackLocationToNext(Irp);

    IoSetCompletionRoutine(Irp,
                           CompletionRoutine_2,
                           &event,
                           TRUE,
                           TRUE,
                           TRUE
                           );

    status = IoCallDriver(TopOfDeviceStack, Irp);

    if (status == STATUS_PENDING) {

       KeWaitForSingleObject(&event,
                             Executive, // WaitReason
                             KernelMode, // must be Kernelmode to preve
nt the stack getting paged out
                             FALSE,
                             NULL // indefinite wait
                             );
       status = Irp->IoStatus.Status;
    }

    // <---- Do your own work here.


    //
    // Because you stopped the completion of the IRP in the CompletionR
outine
    // by returning STATUS_MORE_PROCESSING_REQUIRED, you must call
    // IoCompleteRequest here.
    //
    IoCompleteRequest (Irp, IO_NO_INCREMENT);
    return status;

}
NTSTATUS
CompletionRoutine_2(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
  if (Irp->PendingReturned == TRUE) {
    //
    // You will set the event only if the lower driver has returned
    // STATUS_PENDING earlier. This optimization removes the need to
    // call KeSetEvent unnecessarily and improves performance because t
he
    // system does not have to acquire an internal lock.
    //
    KeSetEvent ((PKEVENT) Context, IO_NO_INCREMENT, FALSE);
  }
  // This is the only status you can return.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

back to the top

Scenario 3: Forward with a completion routine

In this case, the driver sets a completion routine, forwards the IRP down, and then returns the status of lower driver as is. The purpose of setting the completion routine is to modify the content of the IRP on its way back.

```
NTSTATUS
DispathRoutine_3(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    NTSTATUS status;

    //
    // Because you are setting completion routine, you must copy the
    // current stack location to the next. You cannot skip a location
    // here.
    //
    IoCopyCurrentIrpStackLocationToNext(Irp);

    IoSetCompletionRoutine(Irp,
                           CompletionRoutine_31,// or CompletionRoutin
e_32
                           NULL,
                           TRUE,
                           TRUE,
                           TRUE
                           );

    return IoCallDriver(TopOfDeviceStack, Irp);

}
```

If you return the status of the lower driver from your dispatch routine:

- You must not change the status of the IRP in the completion routine. This is to make sure that the status values set in the IRP's IoStatus block (Irp->IoStatus.Status) are the same as the return status of the lower drivers.

- You must propagate the pending status of the IRP as indicated by Irp->PendingReturned.

- You must not change the synchronicity of the IRP.

As a result, there are only 2 valid versions of the completion routine in this scenario (31 and 32):

```
NTSTATUS
CompletionRoutine_31 (
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{

    //
    // Because the dispatch routine is returning the status of lower driver
    // as is, you must do the following:
    //
    if (Irp->PendingReturned) {

        IoMarkIrpPending( Irp );
    }

    return STATUS_CONTINUE_COMPLETION ; // Make sure of same synchronicity
}

NTSTATUS
CompletionRoutine_32 (
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    //
    // Because the dispatch routine is returning the status of lower driver
    // as is, you must do the following:
    //
    if (Irp->PendingReturned) {

        IoMarkIrpPending( Irp );
    }

    //
    // To make sure of the same synchronicity, complete the IRP here.
    // You cannot complete the IRP later in another thread because the
    // the dispatch routine is returning the status returned by the lower
    // driver as is.
    //
    IoCompleteRequest( Irp,  IO_NO_INCREMENT);

    //
    // Although this is an unusual completion routine that you rarely see,
    // it is discussed here to address all possible ways to handle IRPs.
    //
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

back to the top

Scenario 4: Queue for later, or forward and reuse

Use the following code snippet in a situation where the driver wants to either queue an IRP and process it later or forward the IRP to the lower driver and reuse it for a specific number of times before completing the IRP. The dispatch routine marks the IRP pending and returns STATUS_PENDING because the IRP is going to be completed later in a different thread. Here, the completion routine can change the status of the IRP if necessary (in contrast to the previous scenario).

```
NTSTATUS
DispathRoutine_4(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    NTSTATUS status;

    //
    // You mark the IRP pending if you are intending to queue the IRP
    // and process it later. If you are intending to forward the IRP
    // directly, use one of the methods discussed earlier in this artic
le.
    //
    IoMarkIrpPending( Irp );

    //
    // For demonstration purposes: this IRP is forwarded to the lower d
river.
    //
    IoCopyCurrentIrpStackLocationToNext(Irp);

    IoSetCompletionRoutine(Irp,
                           CompletionRoutine_41, // or CompletionRoutin
e_42
                           NULL,
                           TRUE,
                           TRUE,
                           TRUE
                           );
    IoCallDriver(TopOfDeviceStack, Irp);

    //
    // Because you marked the IRP pending, you must return pending,
    // regardless of the status of returned by IoCallDriver.
    //
    return STATUS_PENDING ;

}
```

The completion routine can either return STATUS_CONTINUE_COMPLETION or STATUS_MORE_PROCESSING_REQUIRED. You return STATUS_MORE_PROCESSING_REQUIRED only if you intend to reuse the IRP from another thread and complete it later.

```
NTSTATUS
CompletionRoutine_41(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    //
    // By returning STATUS_CONTINUE_COMPLETION , you are relinquishing
the
    // ownership of the IRP. You cannot touch the IRP after this.
    //
    return STATUS_CONTINUE_COMPLETION ;
}


NTSTATUS
CompletionRoutine_42 (
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    //
    // Because you are stopping the completion of the IRP by returning
the
    // following status, you must complete the IRP later.
    //
    return STATUS_MORE_PROCESSING_REQUIRED ;
}
```

back to the top

Scenario 5: Complete the IRP in the dispatch routine

This scenario shows how to complete an IRP in the dispatch routine.

Important When you complete an IRP in the dispatch routine, the return status of the dispatch routine should match the status of the value that is set in the IoStatus block of the IRP (Irp->IoStatus.Status).

```
NTSTATUS
DispatchRoutine_5(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    //
    // <-- Process the IRP here.
    //
    Irp->IoStatus.Status = STATUS_XXX;
    Irp->IoStatus.Information = YYY;
    IoCompletRequest(Irp, IO_NO_INCREMENT);
    return STATUS_XXX;
}
```

back to the top

REFERENCES

Part 2 of this subject is contained in the following Knowledge Base article:

326315 Different ways of handling IRPs - cheat sheet (part 2 of 2)

- Walter Oney. Programming Windows Driver Model, Second Edition, Chapter 5.

back to the top

Properties

Article ID: 320275 - Last Review: 07/19/2005 21:07:00 - Revision: 5.0

Applies to
Microsoft Win32 Device Driver Kit for Windows 2000

Microsoft Windows XP Driver Development Kit

Microsoft Windows Server 2003 Driver Development Kit

Microsoft Windows NT 4.0

Microsoft Windows 2000 Standard Edition

the operating system: Microsoft Windows XP

Keywords:
kbinfo kbkmode kbwdm KB320275

Support

Account support

Supported products list

Product support lifecycle

Security

Virus and security

Safety & Security Center

Download Security Essentials

Malicious Software Removal Tool

Contact Us

Report a support scam

Disability Answer Desk

Locate Microsoft addresses worldwide

English (United States)

Terms of use Privacy & cookies Trademarks