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

This article was previously published under Q326315

This article is part 2 of the following article in the Microsoft Knowledge Base:

320275 Different ways of handling IRPs - cheat sheet (part 1 of 2)

Creating IRPs and sending them to another driver

INTRODUCTION

Before you examine the scenarios, you must understand the differences between a driver-created synchronous input/output request packet (IRP) and an asynchronous request.

Synchronous (Threaded) IRP	**Asynchronous (Non-Threaded) IRP**
Created by using **IoBuildSynchronousFsdRequest** or **IoBuildDeviceIoControlRequest**.	Created by using **IoBuildAsynchronousFsdRequest** or **IoAllocateIrp**. This is meant for driver to driver communication.
Thread must wait for the IRP to complete.	Thread does not have to wait for the IRP to complete.
Associated with the thread that created it, hence, the name threaded IRPs. Therefore, if the thread exits, the I/O manager cancels the IRP.	Not associated with the thread that created it.
Cannot be created in an arbitrary thread context.	Can be created in an arbitrary thread context because the thread does not wait for the IRP to complete.
The I/O manager does the post completions to free the buffer that is associated with the IRP.	The I/O manager cannot do the cleanup. The driver must provide a completion routine and free the buffers that are associated with the IRP.
Must be sent at IRQL level equal to	Can be sent at IRQL less than or equal to

PASSIVE_LEVEL.	DISPATCH_LEVEL if the Dispatch routine of the target driver can handle the request at DISPATCH_LEVEL.

MORE INFORMATION

Scenario 6: Send a synchronous device-control request (IRP_MJ_INTERNAL_DEVICE_CONTROL/IRP_MJ_DEVICE_CONTROL) by using IoBuildDeviceIoControlRequest

The following code shows how to use the **IoBuildDeviceIoControlRequest** request to make a synchronous IOCTL request.

```
NTSTATUS
MakeSynchronousIoctl(
    IN PDEVICE_OBJECT    TopOfDeviceStack,
    IN ULONG         IoctlControlCode,
    PVOID             InputBuffer,
    ULONG             InputBufferLength,
    PVOID             OutputBuffer,
    ULONG             OutputBufferLength
    )
/*++

Arguments:

    TopOfDeviceStack-

    IoctlControlCode              - Value of the IOCTL request

    InputBuffer        - Buffer to be sent to the TopOfDeviceStack

    InputBufferLength  - Size of buffer to be sent to the TopOfDeviceSt
ack

    OutputBuffer       - Buffer for received data from the TopOfDeviceS
tack

    OutputBufferLength - Size of receive buffer from the TopOfDeviceSta
```

```
ck

Return Value:

    NT status code

--*/
{
    KEVENT              event;
    PIRP                irp;
    IO_STATUS_BLOCK     ioStatus;
    NTSTATUS status;

    //
    // Creating Device control IRP and send it to the another
    // driver without setting a completion routine.
    //

    KeInitializeEvent(&event, NotificationEvent, FALSE);

    irp = IoBuildDeviceIoControlRequest (
                            IoctlControlCode,
                            TopOfDeviceStack,
                            InputBuffer,
                            InputBufferLength,
                            OutputBuffer,
                            OutputBufferLength,
                            FALSE, // External
                            &event,
                            &ioStatus);

    if (NULL == irp) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }


    status = IoCallDriver(TopOfDeviceStack, irp);

    if (status == STATUS_PENDING) {
        //
        // You must wait here for the IRP to be completed because:
        // 1) The IoBuildDeviceIoControlRequest associates the IRP with
the
        //     thread and if the thread exits for any reason, it would
cause the IRP
        //     to be canceled.
        // 2) The Event and IoStatus block memory is from the stack and
we
        //     cannot go out of scope.
```

```
        // This event will be signaled by the I/O manager when the
        // IRP is completed.
        //
        status = KeWaitForSingleObject(
                    &event,
                    Executive, // wait reason
                    KernelMode, // To prevent stack from being paged o
ut.
                    FALSE,     // You are not alertable
                    NULL);     // No time out !!!!

        status = ioStatus.Status;
    }

    return status;
}
```

Scenario 7: Send a synchronous device-control (IOCTL) request and cancel it if not completed in a certain time period

This scenario is similar to the previous scenario except that instead of waiting indefinitely for the request to complete, it waits for some user-specified time and safely cancels the IOCTL request if the wait times out.

```
typedef enum {

   IRPLOCK_CANCELABLE,
   IRPLOCK_CANCEL_STARTED,
   IRPLOCK_CANCEL_COMPLETE,
   IRPLOCK_COMPLETED

} IRPLOCK;
//
// An IRPLOCK allows for safe cancellation. The idea is to protect the
IRP
// while the canceller is calling IoCancelIrp. This is done by wrapping
the
// call in InterlockedExchange(s). The roles are as follows:
//
// Initiator/completion: Cancelable --> IoCallDriver() --> Completed
// Canceller: CancelStarted --> IoCancelIrp() --> CancelCompleted
//
// No cancellation:
```

```
//   Cancelable-->Completed
//
// Cancellation, IoCancelIrp returns before completion:
//   Cancelable --> CancelStarted --> CancelCompleted --> Completed
//
// Canceled after completion:
//   Cancelable --> Completed -> CancelStarted
//
// Cancellation, IRP completed during call to IoCancelIrp():
//   Cancelable --> CancelStarted -> Completed --> CancelCompleted
//
//  The transition from CancelStarted to Completed tells the completer
to block
//  postprocessing (IRP ownership is transferred to the canceller). Sim
ilarly,
//  the canceller learns it owns IRP postprocessing (free, completion,
etc)
//  during a Completed->CancelCompleted transition.
//


NTSTATUS
MakeSynchronousIoctlWithTimeOut(
    IN PDEVICE_OBJECT    TopOfDeviceStack,
    IN ULONG         IoctlControlCode,
    PVOID             InputBuffer,
    ULONG             InputBufferLength,
    PVOID             OutputBuffer,
    ULONG             OutputBufferLength,
    IN  ULONG               Milliseconds
    )
/*++

Arguments:

    TopOfDeviceStack   -

    IoctlControlCode   - Value of the IOCTL request.

    InputBuffer        - Buffer to be sent to the TopOfDeviceStack.

    InputBufferLength  - Size of buffer to be sent to the TopOfDeviceSt
ack.

    OutputBuffer       - Buffer for received data from the TopOfDeviceS
tack.

    OutputBufferLength - Size of receive buffer from the TopOfDeviceSta
ck.
```

```
    Milliseconds       - Timeout value in Milliseconds

Return Value:

    NT status code

--*/
{
    NTSTATUS status;
    PIRP irp;
    KEVENT event;
    IO_STATUS_BLOCK ioStatus;
    LARGE_INTEGER dueTime;
    IRPLOCK lock;

    KeInitializeEvent(&event, NotificationEvent, FALSE);

    irp = IoBuildDeviceIoControlRequest (
                    IoctlControlCode,
                    TopOfDeviceStack,
                    InputBuffer,
                    InputBufferLength,
                    OutputBuffer,
                    OutputBufferLength,
                    FALSE, // External ioctl
                    &event,
                    &ioStatus);



    if (irp == NULL) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    lock = IRPLOCK_CANCELABLE;

    IoSetCompletionRoutine(
                    irp,
                    MakeSynchronousIoctlWithTimeOutCompletion,
                    &lock,
                    TRUE,
                    TRUE,
                    TRUE
                    );

    status = IoCallDriver(TopOfDeviceStack, irp);

    if (status == STATUS_PENDING) {
```

```
        dueTime.QuadPart = -10000 * Milliseconds;

        status = KeWaitForSingleObject(
                            &event,
                            Executive,
                            KernelMode,
                            FALSE,
                            &dueTime
                            );

        if (status == STATUS_TIMEOUT) {

            if (InterlockedExchange((PVOID)&lock, IRPLOCK_CANCEL_STARTE
D) == IRPLOCK_CANCELABLE) {

                //
                // You got it to the IRP before it was completed. You c
an cancel
                // the IRP without fear of losing it, because the compl
etion routine
                // does not let go of the IRP until you allow it.
                //
                IoCancelIrp(irp);

                //
                // Release the completion routine. If it already got th
ere,
                // then you need to complete it yourself. Otherwise, yo
u got
                // through IoCancelIrp before the IRP completed entirel
y.
                //
                if (InterlockedExchange(&lock, IRPLOCK_CANCEL_COMPLETE)
== IRPLOCK_COMPLETED) {
                    IoCompleteRequest(irp, IO_NO_INCREMENT);
                }
            }

            KeWaitForSingleObject(&event, Executive, KernelMode, FALSE,
NULL);

            ioStatus.Status = status; // Return STATUS_TIMEOUT

        } else {

            status = ioStatus.Status;
        }
    }
```

```
    return status;
}

NTSTATUS
MakeSynchronousIoctlWithTimeOutCompletion(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    PLONG lock;

    lock = (PLONG) Context;

    if (InterlockedExchange(lock, IRPLOCK_COMPLETED) == IRPLOCK_CANCE
L_STARTED) {
        //
        // Main line code has got the control of the IRP. It will
        // now take the responsibility of completing the IRP.
        // Therefore...
        return STATUS_MORE_PROCESSING_REQUIRED;
    }

    return STATUS_CONTINUE_COMPLETION ;
}
```

Scenario 8: Send a synchronous non-IOCTL request by using IoBuildSynchronousFsdRequest - completion routine returns STATUS_CONTINUE_COMPLETION

The following code shows how to make a synchronous non-IOCTL request by using **IoBuildSynchronousFsdRequest**. The technique shown here is similar to scenario 6.

```
NTSTATUS
MakeSynchronousNonIoctlRequest (
    PDEVICE_OBJECT   TopOfDeviceStack,
    PVOID               WriteBuffer,
    ULONG               NumBytes
    )
/*++
```

```
Arguments:

    TopOfDeviceStack -

    WriteBuffer       - Buffer to be sent to the TopOfDeviceStack.

    NumBytes  - Size of buffer to be sent to the TopOfDeviceStack.

Return Value:

    NT status code


--*/
{
    NTSTATUS        status;
    PIRP            irp;
    LARGE_INTEGER   startingOffset;
    KEVENT          event;
    IO_STATUS_BLOCK     ioStatus;
    PVOID context;

    startingOffset.QuadPart = (LONGLONG) 0;
    //
    // Allocate memory for any context information to be passed
    // to the completion routine.
    //
    context = ExAllocatePoolWithTag(NonPagedPool, sizeof(ULONG), 'ITa
g');
    if(!context) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    KeInitializeEvent(&event,  NotificationEvent,   FALSE);

    irp = IoBuildSynchronousFsdRequest(
                IRP_MJ_WRITE,
                TopOfDeviceStack,
                WriteBuffer,
                NumBytes,


                &startingOffset, // Optional
                &event,
                &ioStatus
                );

    if (NULL == irp) {
        ExFreePool(context);
```

```
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    IoSetCompletionRoutine(irp,
                  MakeSynchronousNonIoctlRequestCompletion,
                  context,
                  TRUE,
                  TRUE,
                  TRUE);

    status = IoCallDriver(TopOfDeviceStack, irp);

    if (status == STATUS_PENDING) {

       status = KeWaitForSingleObject(
                           &event,
                           Executive,
                           KernelMode,
                           FALSE, // Not alertable
                           NULL);
        status = ioStatus.Status;
    }

    return status;
}
NTSTATUS
MakeSynchronousNonIoctlRequestCompletion(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    if (Context) {
        ExFreePool(Context);
    }
    return STATUS_CONTINUE_COMPLETION ;

}
```

Scenario 9: Send a synchronous non-IOCTL request by using IoBuildSynchronousFsdRequest - completion routine returns STATUS_MORE_PROCESSING_REQUIRED

The only difference between this scenario and scenario 8 is that the completion routine returns STATUS_MORE_PROCESSING_REQUIRED.

```
NTSTATUS MakeSynchronousNonIoctlRequest2(
    PDEVICE_OBJECT TopOfDeviceStack,
    PVOID WriteBuffer,
    ULONG NumBytes
    )
/*++ Arguments:
    TopOfDeviceStack

    WriteBuffer     - Buffer to be sent to the TopOfDeviceStack.

    NumBytes        - Size of buffer to be sent to the TopOfDeviceStac
k.

Return Value:
    NT status code
--*/
{
    NTSTATUS        status;
    PIRP            irp;
    LARGE_INTEGER   startingOffset;
    KEVENT          event;
    IO_STATUS_BLOCK ioStatus;
    BOOLEAN         isSynchronous = TRUE;

    startingOffset.QuadPart = (LONGLONG) 0;
    KeInitializeEvent(&event, NotificationEvent, FALSE);
    irp = IoBuildSynchronousFsdRequest(
                IRP_MJ_WRITE,
                TopOfDeviceStack,
                WriteBuffer,
                NumBytes,
                &startingOffset, // Optional
                &event,
                &ioStatus
                );

    if (NULL == irp) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    IoSetCompletionRoutine(irp,
                MakeSynchronousNonIoctlRequestCompletion2,
                (PVOID)&event,
                TRUE,
                TRUE,
                TRUE);
```

```
    status = IoCallDriver(TopOfDeviceStack, irp);

    if (status == STATUS_PENDING) {

        KeWaitForSingleObject(&event,
                              Executive,
                              KernelMode,
                              FALSE, // Not alertable
                              NULL);
        status = irp->IoStatus.Status;
        isSynchronous = FALSE;
    }

    //
    // Because you have stopped the completion of the IRP, you must
    // complete here and wait for it to be completed by waiting
    // on the same event again, which will be signaled by the I/O
    // manager.
    // NOTE: you cannot queue the IRP for
    // reuse by calling IoReuseIrp because it does not break the
    // association of this IRP with the current thread.
    //

    KeClearEvent(&event);
    IoCompleteRequest(irp, IO_NO_INCREMENT);

    //
    // We must wait here to prevent the event from going out of scope.
    // I/O manager will signal the event and copy the status to our
    // IoStatus block for synchronous IRPs only if the return status is
not
    // an error. For asynchronous IRPs, the above mentioned copy operat
ion
    // takes place regardless of the status value.
    //

    if (!(NT_ERROR(status) && isSynchronous)) {
        KeWaitForSingleObject(&event,
                              Executive,
                              KernelMode,
                              FALSE, // Not alertable
                              NULL);
    }
    return status;
}

NTSTATUS MakeSynchronousNonIoctlRequestCompletion2(
    IN PDEVICE_OBJECT   DeviceObject,
```

```
    IN PIRP             Irp,
    IN PVOID            Context )
{
    if (Irp->PendingReturned) {
        KeSetEvent ((PKEVENT) Context, IO_NO_INCREMENT, FALSE);
    }
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Scenario 10: Send an asynchronous request by using IoBuildAsynchronousFsdRequest

This scenario shows how to make an asynchronous request by using the **IoBuildAsynchronousFsdRequest** function.

In an asynchronous request, the thread that made the request does not have to wait for the IRP to complete. The IRP can be created in an arbitrary thread context because the IRP is not associated with the thread. You must provide a completion routine and release the buffers and IRP in the completion routine if you do not intend to reuse the IRP. This is because the I/O manager cannot do post-completion cleanup of driver-created asynchronous IRPs (created with **IoBuildAsynchronousFsdRequest** and **IoAllocateIrp**).

```
NTSTATUS
MakeAsynchronousRequest (
    PDEVICE_OBJECT   TopOfDeviceStack,
    PVOID               WriteBuffer,
    ULONG               NumBytes
    )
/*++
Arguments:

    TopOfDeviceStack -

    WriteBuffer       - Buffer to be sent to the TopOfDeviceStack.

    NumBytes  - Size of buffer to be sent to the TopOfDeviceStack.

--*/
{
    NTSTATUS        status;
    PIRP            irp;
    LARGE_INTEGER   startingOffset;
    PIO_STACK_LOCATION  nextStack;
```

```
    PVOID context;

    startingOffset.QuadPart = (LONGLONG) 0;

    irp = IoBuildAsynchronousFsdRequest(
                IRP_MJ_WRITE,
                TopOfDeviceStack,
                WriteBuffer,
                NumBytes,
                &startingOffset, // Optional
                NULL
                );

    if (NULL == irp) {

        return STATUS_INSUFFICIENT_RESOURCES;
    }

    //
    // Allocate memory for context structure to be passed to the comple
tion routine.
    //
    context = ExAllocatePoolWithTag(NonPagedPool, sizeof(ULONG_PTR), 'I
Tag');
    if (NULL == context) {
        IoFreeIrp(irp);
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    IoSetCompletionRoutine(irp,
                   MakeAsynchronousRequestCompletion,
                   context,
                   TRUE,
                   TRUE,
                   TRUE);
    //
    // If you want to change any value in the IRP stack, you must
    // first obtain the stack location by calling IoGetNextIrpStackLoca
tion.
    // This is the location that is initialized by the IoBuildxxx reque
sts and
    // is the one that the target device driver is going to view.
    //
    nextStack = IoGetNextIrpStackLocation(irp);
    //
    // Change the MajorFunction code to something appropriate.
    //
    nextStack->MajorFunction = IRP_MJ_SCSI;
```

```
    (void) IoCallDriver(TopOfDeviceStack, irp);

    return STATUS_SUCCESS;
}
NTSTATUS
MakeAsynchronousRequestCompletion(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    PMDL mdl, nextMdl;

    //
    // If the target device object is set up to do buffered i/o
    // (TopOfDeviceStack->Flags and DO_BUFFERED_IO), then
    // IoBuildAsynchronousFsdRequest request allocates a system buffer
    // for read and write operation. If you stop the completion of the
IRP
    // here, you must free that buffer.
    //

    if(Irp->AssociatedIrp.SystemBuffer && (Irp->Flags & IRP_DEALLOCAT
E_BUFFER) ) {
            ExFreePool(Irp->AssociatedIrp.SystemBuffer);
    }

    //
    // If the target device object is set up do direct i/o (DO_DIRECT_I
O), then
    // IoBuildAsynchronousFsdRequest creates an MDL to describe the buf
fer
    // and locks the pages. If you stop the completion of the IRP, you
must unlock
    // the pages and free the MDL.
    //

    else if (Irp->MdlAddress != NULL) {
        for (mdl = Irp->MdlAddress; mdl != NULL; mdl = nextMdl) {
            nextMdl = mdl->Next;
            MmUnlockPages( mdl ); IoFreeMdl( mdl ); // This function wi
ll also unmap pages.
        }
        Irp->MdlAddress = NULL;
    }

    if(Context) {
        ExFreePool(Context);
    }
```

```
    //
    // If you intend to queue the IRP and reuse it for another request,
    // make sure you call IoReuseIrp(Irp, STATUS_SUCCESS) before you re
use.
    //
    IoFreeIrp(Irp);

    //
    // NOTE: this is the only status that you can return for driver-cre
ated asynchronous IRPs.
    //
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Scenario 11: Send an asynchronous request by using IoAllocateIrp

This scenario is similar to the previous scenario except that instead of using **IoBuildAsynchronousFsdRequest**, this scenario uses **IoAllocateIrp** function to create the IRP.

```
NTSTATUS
MakeAsynchronousRequest2(
    PDEVICE_OBJECT   TopOfDeviceStack,
    PVOID               WriteBuffer,
    ULONG               NumBytes
    )
/*++
Arguments:

    TopOfDeviceStack -

    WriteBuffer       - Buffer to be sent to the TopOfDeviceStack.

    NumBytes  - Size of buffer to be sent to the TopOfDeviceStack.

--*/
{
    NTSTATUS        status;
    PIRP            irp;
```

```
    LARGE_INTEGER   startingOffset;
    KEVENT          event;
    PIO_STACK_LOCATION  nextStack;

    startingOffset.QuadPart = (LONGLONG) 0;

    //
    // Start by allocating the IRP for this request.  Do not charge quo
ta
    // to the current process for this IRP.
    //

    irp = IoAllocateIrp( TopOfDeviceStack->StackSize, FALSE );
    if (NULL == irp) {

        return STATUS_INSUFFICIENT_RESOURCES;
    }

     //
    // Obtain a pointer to the stack location of the first driver that
will be
    // invoked.  This is where the function codes and the parameters ar
e set.
    //

    nextStack = IoGetNextIrpStackLocation( irp );
    nextStack->MajorFunction = IRP_MJ_WRITE;
    nextStack->Parameters.Write.Length = NumBytes;
    nextStack->Parameters.Write.ByteOffset= startingOffset;


    if(TopOfDeviceStack->Flags & DO_BUFFERED_IO) {

        irp->AssociatedIrp.SystemBuffer = WriteBuffer;
        irp->MdlAddress = NULL;

    } else if (TopOfDeviceStack->Flags & DO_DIRECT_IO) {
        //
        // The target device supports direct I/O operations.  Allocate
        // an MDL large enough to map the buffer and lock the pages int
o
        // memory.
        //
        irp->MdlAddress = IoAllocateMdl( WriteBuffer,
                                         NumBytes,
                                         FALSE,
                                         FALSE,
                                         (PIRP) NULL );
        if (irp->MdlAddress == NULL) {
```

```
            IoFreeIrp( irp );
            return STATUS_INSUFFICIENT_RESOURCES;
        }

        try {

            MmProbeAndLockPages( irp->MdlAddress,
                                 KernelMode,
                                 (LOCK_OPERATION) (nextStack->MajorFunc
tion == IRP_MJ_WRITE ? IoReadAccess : IoWriteAccess) );

        } except(EXCEPTION_EXECUTE_HANDLER) {

              if (irp->MdlAddress != NULL) {
                  IoFreeMdl( irp->MdlAddress );
              }
              IoFreeIrp( irp );
              return  GetExceptionCode();

        }
    }

    IoSetCompletionRoutine(irp,
                   MakeAsynchronousRequestCompletion2,
                   NULL,
                   TRUE,
                   TRUE,
                   TRUE);

    (void) IoCallDriver(TargetDeviceObject, irp);

    return STATUS_SUCCESS;
}

NTSTATUS
MakeAsynchronousRequestCompletion2(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    PMDL mdl, nextMdl;

    //
    // Free any associated MDL.
    //

    if (Irp->MdlAddress != NULL) {
        for (mdl = Irp->MdlAddress; mdl != NULL; mdl = nextMdl) {
```

```
            nextMdl = mdl->Next;
            MmUnlockPages( mdl ); IoFreeMdl( mdl ); // This function wi
ll also unmap pages.
        }
        Irp->MdlAddress = NULL;
    }

    //
    // If you intend to queue the IRP and reuse it for another request,
    // make sure you call IoReuseIrp(Irp, STATUS_SUCCESS) before you re
use.
    //

    IoFreeIrp(Irp);

    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Scenario 12: Send an asynchronous request and cancel it in a different thread

This scenario shows you how you can send one request at a time to a lower driver without waiting for the request to complete, and you can also cancel the request at any time from another thread.

You can remember the IRP and other variables to do this work in a device extension or in a context structure global to the device as shown below. The state of the IRP is tracked with an IRPLOCK variable in the device extension. The IrpEvent is used to make sure that the IRP is fully completed (or freed) before making the next request.

This event is also useful when you handle IRP_MN_REMOVE_DEVICE and IRP_MN_STOP_DEVICE PNP requests where you have to make sure that there are no pending IRPs before you complete these requests. This event works best when you initialize it as a synchronization event in AddDevice or in some other initialization routine.

```
typedef struct _DEVICE_EXTENSION{
    ..
    PDEVICE_OBJECT TopOfDeviceStack;
    PIRP PendingIrp;
    IRPLOCK IrpLock; // You need this to track the state of the IRP.
    KEVENT IrpEvent; // You need this to synchronize various threads.
    ..
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

```
InitializeDeviceExtension( PDEVICE_EXTENSION  DeviceExtension)
{
    KeInitializeEvent(&DeviceExtension->IrpEvent, SynchronizationEvent,
TRUE);
}

NTSTATUS
MakeASynchronousRequest3(
    PDEVICE_EXTENSION  DeviceExtension,
    PVOID               WriteBuffer,
    ULONG               NumBytes
    )
/*++
Arguments:

    DeviceExtension -

    WriteBuffer       - Buffer to be sent to the TargetDeviceObject.

    NumBytes  - Size of buffer to be sent to the TargetDeviceObject.

--*/
{
    NTSTATUS        status;
    PIRP            irp;
    LARGE_INTEGER   startingOffset;
    PIO_STACK_LOCATION  nextStack;

    //
    // Wait on the event to make sure that PendingIrp
    // field is free to be used for the next request. If you do
    // call this function in the context of the user thread,
    // make sure to call KeEnterCriticialRegion before the wait to prot
ect
    // the thread from getting suspended while holding a lock.
    //
    KeWaitForSingleObject( &DeviceExtension->IrpEvent,
                           Executive,
                           KernelMode,
                           FALSE,
                           NULL );
```

```
    startingOffset.QuadPart = (LONGLONG) 0;
    //
    // If the IRP is used for the same purpose every time, you can just
create the IRP in the
    // Initialization routine one time and reuse it by calling IoReuseI
rp.
    // The only thing that you have to do in the routines in this artic
le
    // is remove the lines that call IoFreeIrp and set the PendingIrp
    // field to NULL. If you do so, make sure that you free the IRP
    // in the PNP remove handler.
    //
    irp = IoBuildAsynchronousFsdRequest(
                IRP_MJ_WRITE,
                DeviceExtension->TopOfDeviceStack,
                WriteBuffer,
                NumBytes,
                &startingOffset, // Optional
                NULL
                );

    if (NULL == irp) {

        return STATUS_INSUFFICIENT_RESOURCES;
    }

    //
    // Initialize the fields relevant fields in the DeviceExtension
    //
    DeviceExtension->PendingIrp = irp;
    DeviceExtension->IrpLock = IRPLOCK_CANCELABLE;

    IoSetCompletionRoutine(irp,
                   MakeASynchronousRequestCompletion3,
                   DeviceExtension,
                   TRUE,
                   TRUE,
                   TRUE);
    //
    // If you want to change any value in the IRP stack, you must
    // first obtain the stack location by calling IoGetNextIrpStackLoca
tion.
    // This is the location that is initialized by the IoBuildxxx reque
sts and
    // is the one that the target device driver is going to view.
    //

    nextStack = IoGetNextIrpStackLocation(irp);
```

```
    //
    // You could change the MajorFunction code to something appropriat
e.
    //
    nextStack->MajorFunction = IRP_MJ_SCSI;

    (void) IoCallDriver(DeviceExtension->TopOfDeviceStack, irp);

    return STATUS_SUCCESS;
}

NTSTATUS
MakeASynchronousRequestCompletion3(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    PMDL mdl, nextMdl;
    PDEVICE_EXTENSION deviceExtension = Context;

    //
    // If the target device object is set up to do buffered i/o
    // (TargetDeviceObject->Flags & DO_BUFFERED_IO), then
    // IoBuildAsynchronousFsdRequest request allocates a system buffer
    // for read and write operation. If you stop the completion of the
IRP
    // here, you must free that buffer.
    //

    if(Irp->AssociatedIrp.SystemBuffer && (Irp->Flags & IRP_DEALLOCAT
E_BUFFER) ) {
            ExFreePool(Irp->AssociatedIrp.SystemBuffer);
    }

    //
    // If the target device object is set up to do direct i/o (DO_DIREC
T_IO), then
    // IoBuildAsynchronousFsdRequest creates an MDL to describe the buf
fer
    // and locks the pages. If you stop the completion of the IRP, you
must unlock
    // the pages and free the MDL.
    //

    if (Irp->MdlAddress != NULL) {
        for (mdl = Irp->MdlAddress; mdl != NULL; mdl = nextMdl) {
            nextMdl = mdl->Next;
            MmUnlockPages( mdl ); IoFreeMdl( mdl ); // This function wi
```

```
ll also unmap pages.
        }
        Irp->MdlAddress = NULL;
    }

    if (InterlockedExchange((PVOID)&deviceExtension->IrpLock, IRPLOCK_C
OMPLETED)
                    == IRPLOCK_CANCEL_STARTED) {
        //
        // Main line code has got the control of the IRP. It will
        // now take the responsibility of freeing the IRP.
        // Therefore...
        return STATUS_MORE_PROCESSING_REQUIRED;
    }

    //
    // If you intend to queue the IRP and reuse it for another request,
make
    // sure you call IoReuseIrp(Irp, STATUS_SUCCESS) before you reuse.
    //
    IoFreeIrp(Irp);
    deviceExtension->PendingIrp = NULL; // if freed
    //
    // Signal the event so that the next thread in the waiting list
    // can send the next request.
    //
    KeSetEvent (&deviceExtension->IrpEvent, IO_NO_INCREMENT, FALSE);

    return STATUS_MORE_PROCESSING_REQUIRED;
}

VOID
CancelPendingIrp(
    PDEVICE_EXTENSION DeviceExtension
    )
/*++
    This function tries to cancel the PendingIrp if it is not already c
ompleted.
    Note that the IRP may not be completed and freed when the
    function returns. Therefore, if you are calling this from your PNP
Remove device handle,
    you must wait on the IrpEvent to make sure the IRP is indeed comple
ted
    before successfully completing the remove request and allowing the
driver to unload.
--*/
{
     if (InterlockedExchange((PVOID)&DeviceExtension->IrpLock, IRPLOC
K_CANCEL_STARTED) == IRPLOCK_CANCELABLE) {
```

```
        //
        // You got it to the IRP before it was completed. You can cance
l
        // the IRP without fear of losing it, as the completion routine
        // will not let go of the IRP until you say so.
        //
        IoCancelIrp(DeviceExtension->PendingIrp);
        //
        // Release the completion routine. If it already got there,
        // then you need to free it yourself. Otherwise, you got
        // through IoCancelIrp before the IRP completed entirely.
        //
        if (InterlockedExchange((PVOID)&DeviceExtension->IrpLock, IRPLO
CK_CANCEL_COMPLETE) == IRPLOCK_COMPLETED) {
            IoFreeIrp(DeviceExtension->PendingIrp);
            DeviceExtension->PendingIrp = NULL;
            KeSetEvent(&DeviceExtension->IrpEvent, IO_NO_INCREMENT, FAL
SE);
        }

     }

    return ;
}
```

Note This is a "FAST PUBLISH" article created directly from within the Microsoft support organization. The information contained herein is provided as-is in response to emerging issues. As a result of the speed in making it available, the materials may include typographical errors and may be revised at any time without notice. See Terms of Use for other considerations.

Properties

Article ID: 326315 - Last Review: 06/04/2012 18:25:00 - Revision: 11.0

Applies to
Microsoft Windows XP Driver Development Kit

Keywords:
kbinfo kbkmode kbwdm KB326315

English (United States)