The NT Insider

The Truth About Cancel - IRP Cancel Operations (Part II)

(By: The NT Insider, Vol 5, Issue 2, Mar-Apr 1998 | Published: 15-Apr-98| Modified: 20-Aug-02)

Click Here to Download: Code Associated With This Article Zip Archive, 797KB

The first part of this article introduced the IRP Cancel mechanism in the Windows NT I/O Subsystem. We looked at three code segments from a sample driver that performed:

- The Dispatch processing operations that set up a driver and an IRP for subsequent Cancel processing;
- A Cancel routine;
- Completion processing that removes a driver from Cancel processing of an IRP.

The NT I/O Manager was described as having a Three Phase Cancel processing state machine. Ignoring explicit IRP Cancellation, Cancel processing is initiated by thread termination. In Phase One the I/O Manager runs the Thread IRP list, calling the cancel routine set in the IRP for each IRP on the list. Phase Two starts as soon as the IRP list has been traversed. The I/O Manager executes a timed wait loop that terminates when the thread **IrpList** is empty or five minutes has expired. The intent of Phase Two is allow IRPs that were, for example, in progress on a physical device, to complete. Finally, Phase Three is reached if Phase Two completes due to a timeout condition. The remaining IRPs on the terminating threads **IrpList** are removed from this list, but the IRPs are not freed. Rather the IRPs and their associated resources are retained by the I/O Manager under the optimistic assumption that they will eventually be released back into the system. The thread and perhaps its parent process are then free to continue termination processing.

This article will examine some of the assumptions contained within the code samples provided in the first article.

Setting a Cancel Routine

Previously we asserted that the correct processing steps to support Cancel operations in a dispatch routine were:

1. Mark the IRP pending;
2. Serialize access to your drivers internal queue through a lock;
3. Set the IRP cancel routine (**IoSetCancelRoutine**(**Irp**, your **CancelRoutine**));
4. Queue the IRP;
5. Release your lock.

Wow, someone actually read part one of this article!

Mark Libucha writes:

> *"In your **CancelReadWrite(...)** example, you set the Cancel routine before you queue the IRP. I believe this creates an unnecessary window that could cause this IRP to be a problem IRP in your Phase Two (it would need the 5 minute timeout). Specifically, if the system calls the Cancel routine after your **IoSetCancelRoutine(...)** call and before your*
>
> ***InsertTailList(...)** call, the Cancel routine will not find the IRP. Shouldn't the order of these two calls be reversed?"*

An excellent question. The DDK is quite explicit:

> *"Any driver routine that passes IRPs on for further processing by other driver routines when an IRP might be held in a cancelable state must call **IoSetCancelRoutine(?)** to set its entry point for the Cancel routine in the IRP. Only then can that driver routine call any support routine that causes the IRP to be held in a cancelable state, such as **IoStartPacket(?)**, **IoAllocateController(?)**, or an **ExInterlockedInsert..List(?)** routine."*
>
> *NT DDK 4.0 Design Guide Section 12.4 Points To Consider Holding Cancelable IRPs.*

This explicit ordering is repeated several times in the documentation. The DDK could of course be wrong, and repetition does not increase the truth-value of a statement (although it certainly increases the tendency for people to believe that something is true.) However, it is rather rare for the DDK to be so explicit about something, *and* be

wrong, at least in my experience.

So what is Mr. Libucha?s concern? He is worried that a driver executes the sequence **IoMarkIrpPending(?)**, **IoSetCancelRoutine(?)**, "queue the IRP" while concurrently the thread that issued the IRP terminates, causing the I/O Manager to enter Phase One Cancel processing. His concern is the consequences of the various possible orderings of the I/O Manager?s call to **IoCancelIrp(?)** with respect to the three operations performed by the driver (**pend**. **IoSetCancelRoutine**, **Queue**). Actually we can ignore the **IoMarkIrpPending(?)** operation. Cancel is not going to occur if the IRP is not pending, so it is really the operations (**IoSetCancelRoutine, Queue**) and the call from the I/O Manager to **IoCancelIrp(?)** that are the issue.

When I look at concurrency problems I like to simply line up two sets of operations, representing two threads of execution, and look at the consequences of various orderings of execution between the threads.

In our case we have one thread executing (**IoSetCancelRoutine, Queue**) while another thread executes (**IoCancelIrp**). To simplify things, assume that each of these three operations is in itself atomic and independent of the other operations. Further assume that a single IRP is present in the system, associated with our terminating thread.

The following orderings could occur:

1. {**IoCancelIrp**} could precede { **IoSetCancel Routine, Queue**}, so the order of operations is {**IoCancelIrp, IoSetCancelRoutine, Queue**}
2. {**IoCancelIrp**} could follow { **IoSetCancel Routine, Queue**}, so the order of operations is {**IoSetCancelRoutine, Queue, IoCancelIrp**}
3. {**IoCancelIrp**} could interleave { **IoSetCancel Routine, Queue**}, so the order of operations is {**IoSetCancelRoutine, IoCancelIrp, Queue**}

Ordering 1 can occur if the IRP is already Pending, the issuing thread terminates, and the I/O Manager runs the thread?s pending **IrpList** calling **IoCancelIrp(?)** for each IRP. In this case the **Irp->CancelRoutine** should be null so our drivers Cancel routine will not run. *Not an optimal result*.

Ordering 2 is the normal expected ordering of events. We set our Cancel routine and queue the IRP, later the IRP is canceled and our Cancel routine runs.

Ordering 3 is the case that Mark is concerned about. We set the Cancel routine in the IRP, but before we get a chance to actually queue the IRP, thread termination occurs and the I/O Manager calls **IoCancelIrp(?)**. Our driver?s Cancel routine runs, we don?t find the offending IRP on our queue (we haven?t queued it yet), and now we have to wait the obligatory five minute stupid driver penalty.

Well not quite, our initial assumption that each of these operations is atomic and independent is wrong. What saves us is the fact that both the sequence of operations {**IoSetCancelRoutine, Queue**} and our driver?s Cancel routine acquire our drivers private queue spinlock. While the I/O Manager can call our driver?s Cancel routine in-between {**IoSetCancelRoutine, Queue**}, the driver?s Cancel routine cannot execute anything other than the **KeAcquire Spinlock(?)** statement until {**IoSetCancelRoutine, Queue**} completes with the IRP in question queued. As long as your driver is coded correctly, effectively **IoCancelIrp(?)** cannot interleave the {**IoSetCancelRoutine, Queue**} operations.

Mark?s suggestion is to use an alternate ordering {**Queue, IoSetCancelRoutine**}. Given the correct serialization in the driver sample code, these two orderings don?t make any difference.

Why is the DDK so explicit in requiring the ordering {**IoSetCancelRoutine, Queue**}? My opinion is that the concern in the DDK is that an IRP is queued, partially completed, then the driver calls **IoSetCancelRoutine(?)**, then the issuing thread terminates. In Phase One Cancel processing the IRP is re-completed and the system crashes. As long as the {**IoSetCancelRoutine, Queue**} operations, the drivers Cancel routine(s) and the drivers completion routines use spinlocks correctly to serialize their operations, I don?t think either order is a problem.

There is still a window as described in Ordering One, and it is not closed by any ordering of **Queue** and **IoSetCancelRoutine(?)**. The window is that an IRP will be queued after Cancel Phase One has run.

Consider the case where Phase One runs before our driver does a {**IoSetCancelRoutine, Queue**} sequence. Clearly the IRP will be on our queue, Phase One has already run and will not run again, and until five minutes elapse or the IRP is completed, thread termination will be suspended. Changing the ordering to {**Queue,**

IoSetCancelRoutine} doesn?t alter this fact.

Does it matter? In most cases it doesn?t. In most cases IRPs are queued and then quickly processed to completion. If they happen to fall through the Cancel processing of Phase One, they get completed normally and the thread terminates. It is the case where an IRP is held for extended periods of time that causes trouble, as such an IRP can cause its issuing thread and that thread?s containing process to hang for up to five minutes while terminating.

(Actually there is another truly unpleasant but highly marginal sense in which it matters as well, but I?ll defer that discussion to the end of this article).

Is there a solution?

Mark Libucha continues:

> *"A related question is, before you place the IRP on the queue, why don't you check to see if the IRP has already been cancelled? If it has already been cancelled, the Cancel routine will never be called for this IRP."*

This is indeed a way around the problem, however testing the **Irp->Cancel** flag only makes sense if you also hold the global Cancel spinlock before the test and until after the IRP is queued. Otherwise you might as well look at any random memory location and decide based on the value of the 6th bit.

We can close the window that allows an IRP to be queued after Phase One Cancel Processing has run by holding the global Cancel spinlock while queuing IRPs. Note however that if every device driver adopts this strategy then for every asynchronous I/O request in the system, every time a driver queues an IRP, the system is going to serialize over the global Cancel spinlock. You might as well replace that spiffy quad-processor system you just bought with an El Cheapo x86 single processor system.

Let?s go back to an earlier assertion. I said that,

> *"In most cases IRPs are queued and then quickly processed to completion."*

A possible solution then is for a driver to take the drastic step of holding the global Cancel spinlock only if the driver knows that a specific IRP is likely to pend on the driver?s internal queue for extended periods of time.

What is an extended period of time? Certainly, anything greater than five minutes qualifies. Otherwise, this is simply a judgement call in your driver.

(If all or almost all IRPs are quickly processed to completion it is arguable that a driver writer could simply ignore Cancel processing entirely, allowing for the occasional annoying time-out on process termination. I?m not going to argue that position, but it does have some merit.)

I think that the window on queuing an IRP after Phase One is very small, and that the mechanisms outlined in part one of this article will approach a tolerable level of correctness, and are easy to implement. So don?t give up because you can?t catch every IRP, and don?t settle for a huge performance bottleneck because you are determined to never have some system pop-up dialog complaining that your driver hasn?t managed to complete all of its I/O Requests.

Enough already of picking on our readers.

Data Corruption

Here is a surprising twist on Cancel processing: a terminating application or thread can corrupt data that is being processed by a driver, and short of always using **METHOD_BUFFERED**, there isn?t anything you can do about it.

I fell over this problem writing a test application and driver to explore Cancel processing inside the NT operating system.

The Developer Studio project for this application and driver are available on our website, (http://www.osr.com) so if you don?t believe my reported results, you can go and try this yourself at home. **But please kids, always have a responsible adult to supervise your experiments, and remember to always wear safety goggles.**

Consider the case of a multithreaded C++ application. One thread issues some number of asynchronous I/O

requests, and then ooops, it has an exception. Luckily for the application it has nice exception handling code that does nice C++ things such as calling all of the destructors for all of the stack based objects created by the thread. These destructors do all sorts of clever C++ destructor like things. They deallocate data structures, for example. Some of the data structures deallocated might include the data buffers used to send data to a driver. As part of the deallocation process, a clever C++ destructor might overwrite such buffers ? clearing them to zero for example ? to better detect stale pointer references. This could happen, honest.

Now suppose your driver uses **METHOD_DIRECT** to do data transfer between applications and drivers. The good thing about **METHOD_DIRECT** is that the driver has direct access to the users original data buffer so there is no intermediate copy operation. The fairly hideous Cancel implication is that a terminating thread might very well overwrite data buffers that are currently in progress on your device. Furthermore, there is absolutely nothing you can do about this.

It turns out that my sample application can readily demonstrate data corruption. It can demonstrate many other things too (including the fact that I?m never going to be a GUI programmer). So perhaps this would be a good time to introduce the application.

A Developer Studio Project?

I know that the rumor is that at OSR we do all of our coding using Edit (or is it the editor that nobody ever uses that comes with the SDK?). Moreover, that we never write applications and that we don?t know a foundation class from a java bean. Nevertheless, the sample code is packaged up as a Developer Studio workspace containing a pair of Developer Studio projects, one for an application and one for a driver.

App, the Test Application

The test application is a dialogue based MFC application (shown in *Figure 1*).

It does three things:

1. It loads, starts, stops, and unloads the Cancel driver.
2. It sets the operational mode of the Cancel driver
3. It generates I/O to the Cancel driver and provokes cancellation of the generated I/O.

The test application looks like this:

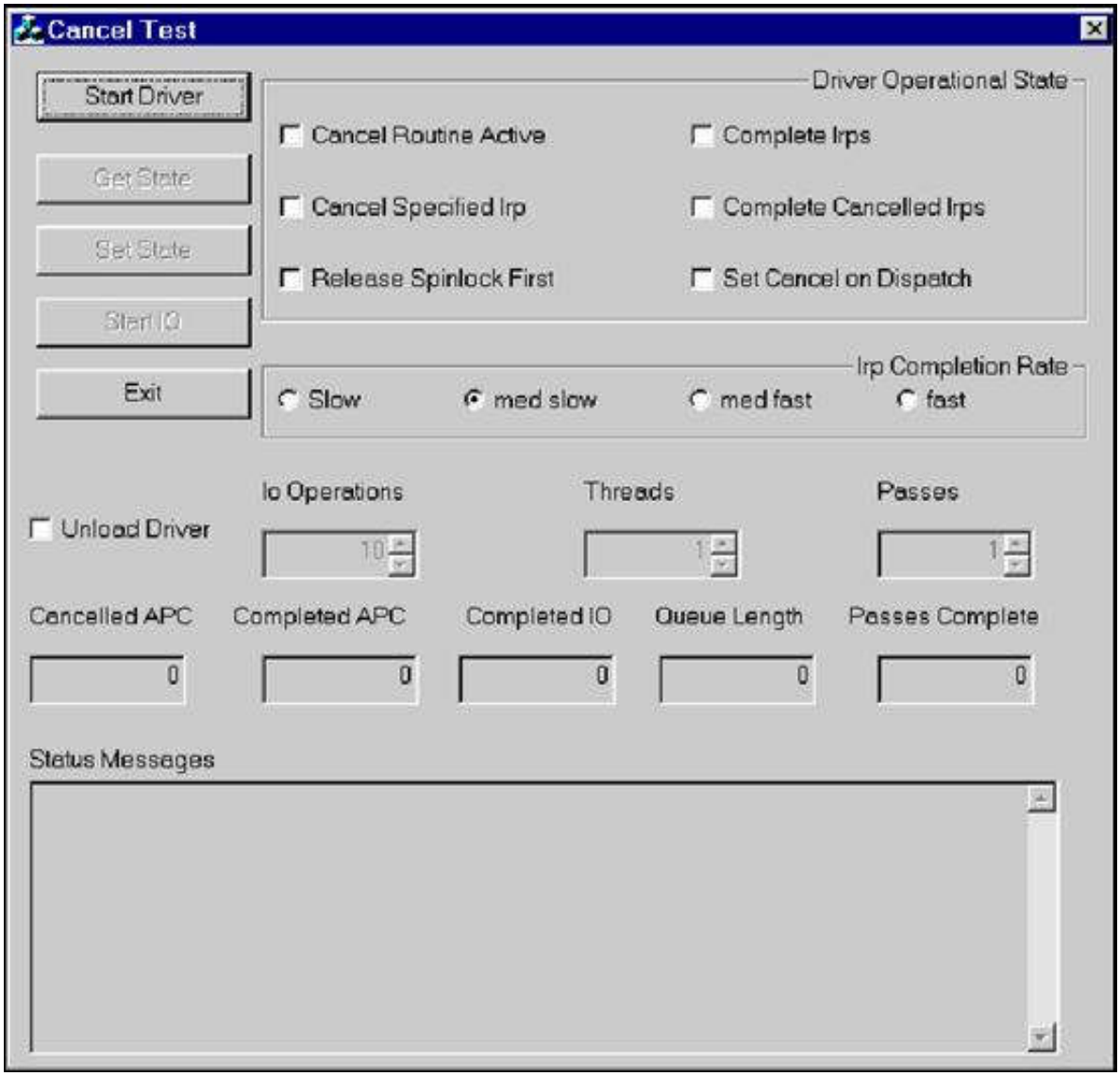

Figure 1 ? Test App

The application has a ServiceControl interface for managing driver operational state. (As in load the driver, start the driver, stop the driver, unload the driver). Users access this interface through the Start Driver button (which of course renames itself to Stop Driver as appropriate).

Driver loading and unloading is accomplished by simply creating and then deleting a ServiceControl object. The source code for ServiceControl is included if you want to know the gory details of exactly how this works. Creating a ServiceControl object also produces a FileObject handle that can be used to target I/O requests to the device implemented by the driver. Deleting the ServiceControl object closes the FileObject, and optionally stops and unloads the driver.

If you are running multiple instances of the test application you should *not* select the Unload Driver option, as this can leave the test driver in a state where it is inaccessible but only a system reboot will actually remove it from the system.

App uses DeviceIoControl operations to set the operational state of the Cancel device based on user input to the dialogue. The set of check boxes grouped under the Driver Operational State label control the way the driver handles I/O cancellation. This is the basic mechanism used to explore various issues with Cancel processing. The runtime behavior of the driver can be modified to do all sorts of stupid things with cancelled IRPs.

App generates a user configurable number of I/O requests to the Cancel device by creating a user configurable set of worker threads that issue asynchronous WriteFile requests to the Cancel device. Once a worker thread has completed its task of issuing WriteFile requests, it sends a message back to the main dialogue thread and then terminates. The worker thread termination triggers IRP cancellation for all I/O requests in progress issued by that

thread. User input into the three editable windows (labeled Io Operations, Threads, and Passes) controls the volume of I/O activity targeted at the test driver.

The Cancel Driver

The driver has eight major functions, a DriverEntry routine, four dispatch entry routines that handle five IRP_MJ function codes, an unload routine, a DPC routine, and a Cancel routine.

DriverEntry(?) of course is the driver initialization routine called once when the driver is loaded. It sets up the initial sate of the driver, creates a device object, symbolic links, etc. The routine is not either complicated or unusual, but it is perhaps worthwhile to look at the code.

The DeviceExtension structure created by the Cancel driver for its single DeviceObject contains the operational state information for the device. This structure is entirely of our own design, and has been constructed so that an application can modify the runtime behavior of the driver with respect to Cancel processing in the driver.

```
typedef struct {

    LIST_ENTRY      irpList;          // our private IRP queue
    ULONG           queueLength;      // how big is it?
    KSPIN_LOCK      lock;             // lock covers this structure;
    KTIMER          timer;            // timer for completing IRPs
    KDPC            dpc;              // timer dpc object
    PDEVICE_OBJECT  device;           // backpointer to the device object
    ULONG           state;            // operational state of the device
    ULONG           speed;            // canonical representation of DPC speed
    LARGE_INTEGER   CancelDpcInterval; // how fast we run the DPC
    BOOLEAN         takeFromFront;    // complete io from head or tail of queue?
    ULONG           tagfield;         // structure tag

} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

#define DEVICE_EXTENSION_TAG 0x73A00101
```

Figure 2 ? Device Extension

The **irpList** field is the device internal queue that this driver will manage, and from which the driver will perform Cancel processing.

The **lock** field is the spinlock that covers access to the Device Extension data structure.

The Cancel driver simulates interrupt activity using a timer DPC. The timer DPC is controlled by the **timer**, **dpc** and **CancelDpcInterval** fields in the Device Extension.

How the Cancel driver performs IRP processing is controlled by the **state** field. As we shall see, the application can modify this field, changing the runtime behavior of the driver. The include file **cancel.h** defines the bits in the state field. These fields (see *Figure 3*) can be viewed and modified by the test application.

```
//
// operational state bits for device extension state field
//
#define CANCEL_ON           0x00000001 // turns on or off doing anything in cancel routine
#define CANCEL_IRP          0x00000002 // cancel the input Irp or the first cancel'd irp?
#define COMPLETE_ON         0x00000004 // do anything in timer dpc?
#define COMPLETE_CANCELED   0x00000008 // if Irp->Cancel complete the Irp anyway?
#define EARLY_RELEASE       0x00000010 // release cancel spinlock at top of cancel routine
#define SET_CANCEL          0x00000020 // set the cancel routine on dispatch
#define VALID_FLAGS         0x0000003F
```

Figure 3 ? Driver State Bit Fields

As a matter of good engineering practice, I always put a **tag** field in data structures I define. The tag field is a field within a data structure with a known constant value that can be used to verify a pointer to such a structure with a high degree of confidence. In this case, the device extension for the Cancel driver has a **tagfield**, which should always have the value **DEVICE_EXTENSION_TAG**.

The **CancelOpen(?)** and **CancelClose(?)** dispatch routines do nothing other than complete their IRPs with **STATUS_SUCCESS**. They could in fact be combined into a single **CancelOpenClose(?)** dispatch routine.

The **CancelReadWrite(?)** dispatch routine accepts **IRP_MJ_READ** and **IRP_MJ_WRITE** requests. It processes both requests in exactly the same manner: it simply places the request on the internal queue for the device, marks the IRP pending, sets up the Cancel routine for the IRP and returns **STATUS_PENDING** to the caller. Consequently, a single dispatch routine processes both **IRP_MJ_READ** IRPs and **IRP_MJ_WRITE IRP**s.

This dispatch routine introduces one of our test insertion points. Our test application can turn the **SET_CANCEL** bit on or off in the driver?s Device Extension State field. If this bit is off, this routine does *not* set the **CancelRoutine** in the IRP.

CancelReadWrite(?) is also responsible for scheduling the deferred processing routine, **CancelTimerDpc(?),** if it queues an IRP to an empty queue. The timer DPC routine simulates interrupt based IRP completion processing.

CancelIoctl(?) provides an interface that allows our test application to look at and modify the operational state of the driver.

CancelUnload(?) supports the NT Unload functionality.

The **CancelTimerDpc(?)** function simulates a DpcForIsr routine in a lowest level driver that completes IRPs based on external interrupts.

CancelTimerDpc(?) is called at **DISPATCH_LEVEL** by the NT kernel. It simulates interrupt based I/O processing by completing one of the internally queued IRPs for the *Cancel* driver. To simulate a truly asynchronous device, the routine does not complete IRPs in order, but instead alternates completing IRPs from either the head or the tail of the internal queue. Because we want our driver to have IRPs to cancel, **CancelTimerDpc(?)** does not complete more than one IRP; instead, it reschedules itself if there are more IRPs left on the internal queue.

Finally, **CancelCancel(?)** is the device CancelRoutine callback function.

CancelReadWrite(?) sets the **CancelRoutine** field of each IRP that it queues to the device internal queue to point to **CancelCancel(?)**. When the I/O Manager decides to start canceling IRPs, it calls **CancelCancel(?)** once for each IRP that needs to be cancelled and that has its **CancelRoutine** field set to the address of **CancelCancel(?)**.

This function can be configured to operate in two ways, depending on the value of the **CANCEL_IRP** bit in the DeviceExtension **state** field. If **CANCEL_IRP** is set, **CancelCancel(?)** will search its internal IRP queue for the IRP passed in as the **Irp** parameter, and only complete this IRP if it is found in the internal queue and has the IRP **Cancel** flag set. If **CANCEL_IRP** is not set, **CancelCancel(?)** ignores the **Irp** parameter and instead finds the first IRP in the internal queue that has the IRP **Cancel** flag set and completes this IRP rather than the input parameter **Irp**. This latter behavior is expected to be the correct algorithm for processing IRPs in the Cancel callback function of a driver that uses internal queuing.

The **CancelCancel(?)** function also examines the **EARLY_RELEASE** bit in the DeviceExtension state field. If **EARLY_RELEASE** is set the **CancelCancel(?)** function calls **IoReleaseCancelSpinLock(?)** after saving the value of **Irp->CancelIrql** but before acquiring the spinlock for the DeviceExtension. If the bit is not set, then the function holds the global Cancel spinlock while acquiring the DeviceExtension spinlock, and does not release either spinlock until after it has searched the internal queue for an IRP to cancel.

Finally, **CancelCancel(?)** does not cancel *any* IRPs if the **CANCEL_ON** bit is not set in the DeviceExtension state field. This allows us to examine the behavior of the system when a device driver queues IRPs internally but does not support the systems IRP Cancellation facility.

Installing App and Cancel

To install **App.exe** and **Cancel.sys** on your test system, simply create a folder on the test system and copy **App.exe** and **Cancel.sys** to that folder. There are of course Debug and Release versions of **App.exe**, and checked and free versions of **Cancel.sys**.

Both **App.exe** and **Cancel.sys** must be in the same folder. **App.exe** has to load **Cancel.sys**, and simply looks for it in the current working directory.

My test system runs NT4.0SP3.The driver was tested on both the checked and free builds of the operating system. This software was also tested on a quad-processor NT system running NT4.0SP0. The version available was built using VC5.0 and therefore must be debugged with a debugger that understands the VC5.0 CodeView formats.

Running a Test

To run a test, do the following:

1. Using the explorer shell double click on the copy of **App.exe** in the folder you created on the test system.
2. After the application starts up click the Start Driver button.
3. Change any of the operational parameters you want.
4. Press the Start IO Button.

When you tire of this, press the exit button and **App.exe** should terminate.

The basic operational scenario is that when you press Start I/O, the application creates one or more threads, each of which issues some number of asynchronous I/O requests to the Cancel driver. As each thread completes its task of issuing asynchronous I/O requests, that thread terminates, provoking Cancel processing in the I/O subsystem.

Demonstrating Data Corruption

OK lets get to the fun part. Fire up *two* instances of the test application. One instance is going to queue a small number of I/O Requests to the Cancel driver, however we are going to configure our Cancel driver to *NOT* cancel these I/O requests. Instead we are going to simulate a device driver that holds onto I/O requests for an extended period of time. The second instance of our test app is used to actually cause the Cancel driver to complete the queued I/O requests *after the first instance of the test application has terminated.*

In order to detect data corruption the test application writes a predictable pattern into the data buffers that it sends to the driver. The Cancel device driver, in turn, examines these data buffers looking for the pattern when it decides to complete an I/O request. If the pattern isn?t found it simply notes that fact by writing a message to the debug monitor console.

Steps required to demonstrate data corruption:

1. Setup windbg for kernel debugging and make sure it is functional and connected to your test system
2. Install the test application on the target system *but be sure that you copy **App.exe** from the Debug directory of the source distribution, not the Release directory*.
3. Start two instances of the test application (that is the version on the Debug directory of the , we will refer to them as **left** and **right**. (And I suggest that you put **left** on the *left* side of your display.)
4. On **left** start the Cancel driver by pushing the Start Driver button. Do the same on **right.**
5. On **left,** turn off the Cancel Routine Active and Complete IRPs check boxes, and then press the Set State button. This step disables Cancel processing in the driver and also defers any completion of I/O requests.
6. On **right** press the Get State button. You should observe that both the Cancel Routine Active and Complete IRPs check boxes are off now. At this point both boxes should look as in *Figure 4 Data Corruption Setup*.
7. Now set up a small number if I/O request in **left** by setting the Io Operations window to 5. Leave the other numeric windows alone.
8. In **left** push the Start IO button. The Status window should display completion status data.
9. Exit the **left** application.
10. Now over in **right** set the Cancel Routine Active and Complete IRPs check boxes back on and press the Set State button. This will cause the Cancel driver to complete the I/O Requests from **left**.
11. Observe the debug output from Windbg. It should complain about corrupted data.

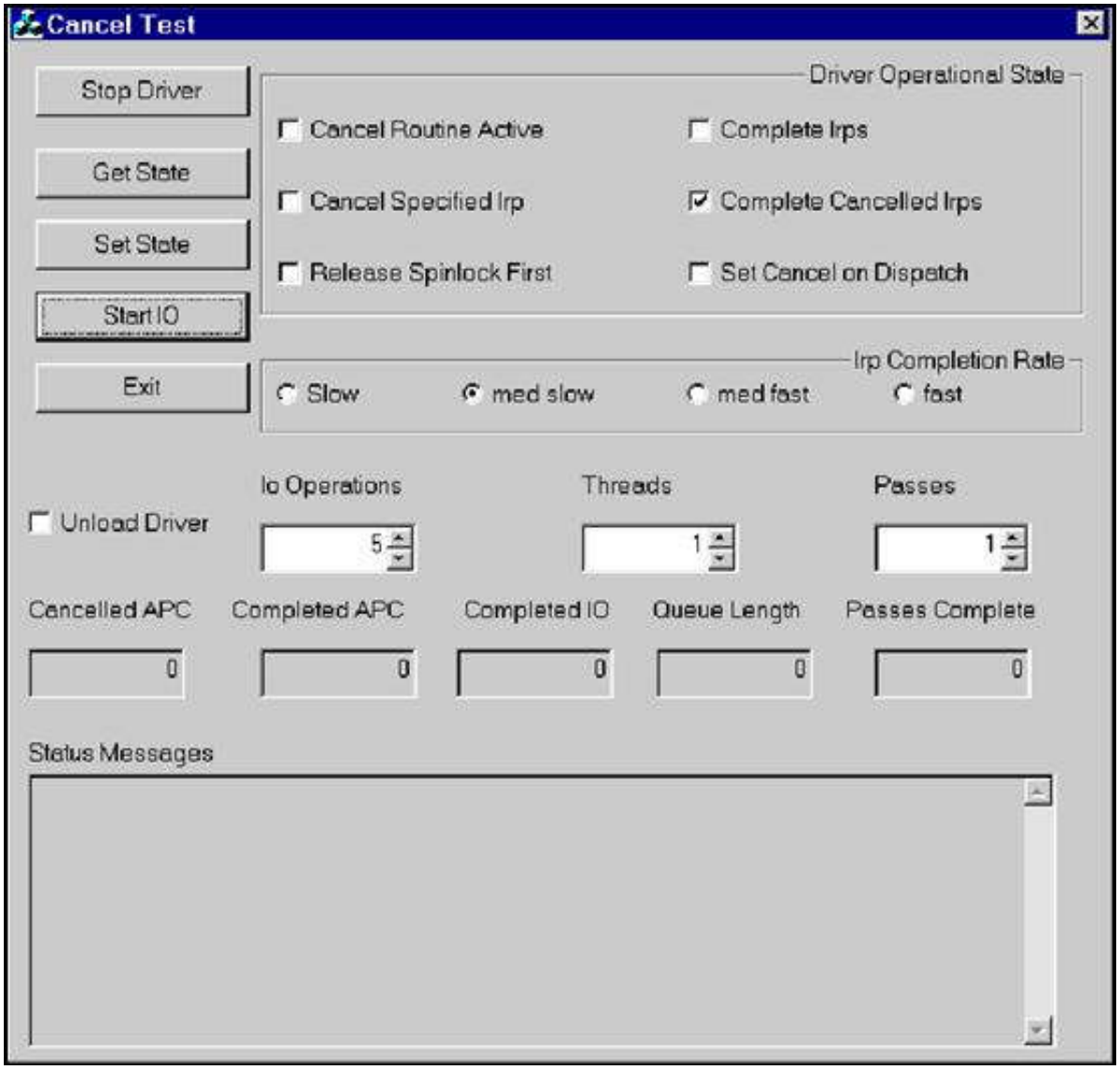

Figure 4 ? Data Corruption Setup

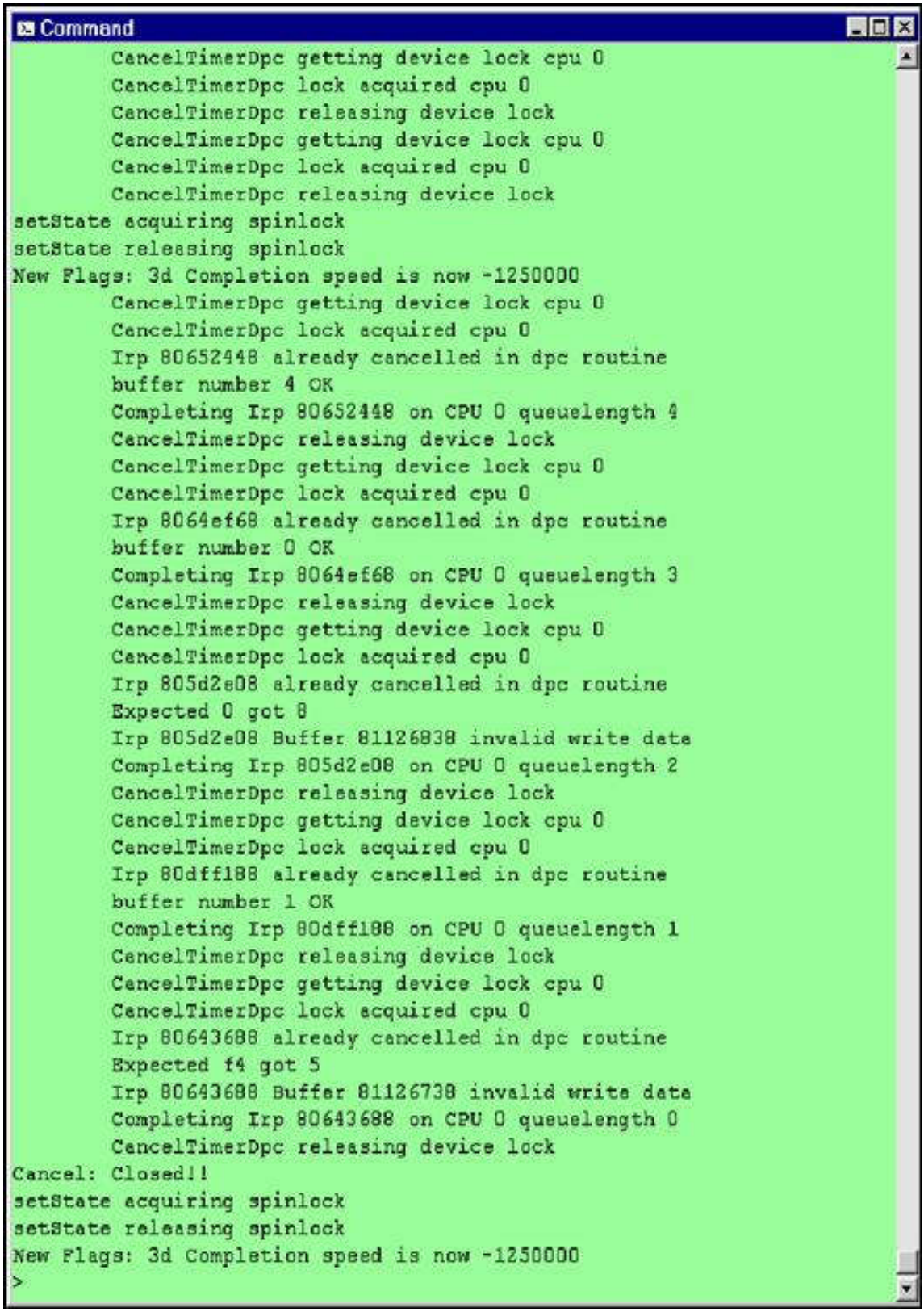

Figure 5 ? Windbg Output

In the windbg output shown in *Figure 5* the Cancel driver examines each IRP as it completes the IRP. It records the fact that the **Irp->Cancel** flag is set, and then tests the data for integrity. Of the five data buffers we queued to the driver, two of them failed the test. Buffers 4, 0, and 1 were OK, but buffers 2 and 3 contained unexpected data.

Analysis

What is going on here?

In the step by step procedure, step 2 required you to install the *Debug* version of the test application. Oddly

enough, the results are *not* reproducible using the *Release* version of the application. The obvious conclusion is that the data corruption is an artifact of the debug processing the Debug versions of the MFC class libraries.

Corruption only occurs if a process is actually terminated, not on simple thread termination. You can use the NT Task Manager to verify that even though the user interface vanishes, the process image for **App.exe** remains in the system until either the 5 minute Phase Two timeout or all of the associated IRPs are completed.

Without spending too much time poking through the MFC process classes and memory managers, clearly what is going on is that as part of the process object destructor, the debug version of MFC is overwriting some of the thread local storage used for sending data to the Cancel driver. As the Cancel driver uses **METHOD_DIRECT** for data transfers, this has the potential for modifying the contents of the data buffers that the driver is processing.

Actually the SDK warns you that this is indeed a problem:

> "This buffer must remain valid for the duration of the write operation. The caller must not use this buffer until the write operation is completed. "
>
> *WriteFileEx Platform SDK.*

What is not explicit is that abnormal thread termination can result in device drivers writing corrupted data out to peripheral devices.

Before you start uninstalling your operating system, you should keep in mind that we are talking about a very small window. The I/O has to be asynchronous. The I/O has to use **METHOD_DIRECT** (or perhaps **METHOD_NEITHER**). Phase One Cancel Processing has to miss an I/O request in progress on a device. The termination process for a thread or process has to overwrite the data buffers.