Portable Systems Group

Windows NT Driver Model Specification

Author: *Darryl E. Havens*

Revision 1.2, July 20, 1990

1. Introduction

The **Windows NT** I/O system provides system programmers the features necessary to write their own device drivers for those devices that **Windows NT** does not support as part of its regular SDK. The driver interface is designed to allow these programmers to write all device drivers in a high-level language. **Windows NT** provides all of the necessary include files to write these drivers in C.

This specification describes the basic flow of control of an I/O request from the requestor's call, through the I/O system, through the device driver, and back through the I/O system to the requestor. It does not attempt to exhaustively enumerate all of the error conditions nor does it attempt to specify how every type of device is to be dealt with in this design.

This specification also describes the basic driver model, how file systems and network systems fit into that model, and then describes the data structures and I/O APIs used to support the model.

For background information about the I/O system API used by code external to the I/O system, please see the *Windows NT I/O System Specification*.

2. Overview

This section presents an overview of the sequence of operations that take place when an I/O operation is requested.

When a user invokes the **NtCreateFile** or the **NtOpenFile** service, the system attempts to translate the name of the supplied file specification that is to be accessed. If the name successfully translates to a device object, then the system passes the remainder of the file specification, if any, to the parse routine for the object.

All I/O services begin by performing the following operations, except as noted. Arguments and addresses are captured as appropriate.

- o - The caller's arguments are probed for read access by the previous mode.

- o - The file handle is translated, referenced, and checked for validity. If the handle is valid, then it is set to the Not-Signaled state. This obviously does not occur on an open or a create operation.

o - The event object handle, if specified, is translated, referenced, and set to the Not-Signaled state, if valid.

o - The caller's I/O status block is alignment checked and probed for write access by the previous mode.

o - The caller's buffers, if any, are probed for the appropriate access by the previous mode. This only occurs at this point if it is known that the I/O being performed is *buffered I/O*[1].

o - On an open or create call, the file name is parsed to determine the device for which the operation is destined.

o - All other parameters specific to device-independent services are checked for accessibility and validity.

Once the above checks have been made and it is determined that the caller's device-independent parameters are valid, an *I/O Request Packet* **(IRP)** is allocated and the parameters are marshalled into it. Some parameters, such as the length of a buffer, are simply copied. Others, such as the handle for an event, are put into the IRP as pointers to objects rather than as handles to them. Also at this point, any user buffers that need to be locked into memory are probed and locked. This causes a *Memory Descriptor List* **(MDL)** to be built that describes the pages that are locked. The address of the MDL is also stored in the IRP.

The IRP is then handed to the driver's entry point according to the major function code of the request. This routine is given a pointer to one of its device objects (the one that the request is for) and a pointer to the IRP. It is the driver's responsibility to validate the remainder of the parameters and then, if valid, to start the I/O operation. Drivers can use the I/O system routine, **IoStartPacket**, to pass the packet to the start I/O routine. This function gives the IRP directly to the driver's start I/O routine if the device is not busy and sets the driver's busy flag. If the device is already busy, then it simply queues the packet to the driver's request queue. All synchronization of the queue is handled in this routine via the use of a *device queue*, a kernel-provided object designed just for this purpose.

---

[1] *Buffered I/O* refers to I/O being performed to an intermediary buffer. *Direct I/O* refers to I/O being performed directly on the original buffer.

Regardless of whether the I/O actually gets started, the packet simply gets queued for later processing, or an error of some kind occurred, the driver's major function routine returns to the I/O system indicating whether or not everything up to this point has been successful. The I/O system then returns to complete the original user call. If the operation was successfully started or queued, then it is up to the caller to synchronize itself with the completion of the I/O operation, unless one of the synchronous options has been specified.

If an error does occur and the I/O operation was neither started nor queued for later processing by the driver, then the operation is considered to be in error and will never be "completed". That is, an error status code is returned to the caller indicating that the operation failed. The file object will not be set to the Signaled state, nor will the event, if one was specified. The APC routine is not executed and the state of the I/O status block is undefined.

The start I/O routine must synchronize access to the device so that the *Interrupt Service Routine* **(ISR)** cannot access the device at the same time as the start I/O routine. Three items of interest must be considered. The start I/O routine must synchronize with the following:

o - Interrupts occurring on the current processor

o - The same start I/O routine executing on another processor

o - Access to the device with power failures

These synchronization issues will be discussed in detail later. Note that this type of synchronization is particularly interesting when dealing with controllers that service multiple devices. It is possible to write a device driver that depends on the state of a busy flag to synchronize access to the device, provided that no unsolicited input interrupt can be taken from the device. That is, for devices of this type, the start I/O routine will never be invoked when an interrupt can occur because the busy flag would already be set. Of course, synchronization with powerfail interrupts must still be dealt with by the driver.

Once the I/O operation completes on the device, the device requests an interrupt. The interrupt will be taken when the processor's *Interrupt Request Level* **(IRQL)** is at a lower level than that of the requesting interrupt. The interrupt dispatcher then uses the

interrupt object, created when the driver was initialized, to invoke the device driver's ISR. The ISR is executed at the same priority level that the device interrupted. This means that no other devices at the same interrupt request level can be serviced until the current interrupt servicing has been completed.

Device drivers are written to perform as little work as possible in their interrupt service routines. Notice also that access to the device needs to be synchronized with the remainder of the driver. Because **Windows NT** supports multiprocessor systems, other parts of the device driver could be accessing the device or some common data base from another processor. (They could not be doing so on the current processor because part of synchronizing with the ISR is raising the current IRQL to the level that the device interrupts, and this blocks the device from interrupting on the current processor.) A per-device spin lock is used to perform this synchronization among the processors. The steps a device driver takes in order to synchronize with the interrupt service routine are as follows:

1. Save the current IRQL.

2. Raise the IRQL to the device interrupt priority level.

3. Obtain the spin lock for the device.

4. Raise IRQL to block powerfail interrupts and check for power failure. If a failure has occurred, do not perform the next step.

5. Manipulate device registers.

6. Release spin lock.

7. Lower IRQL back to saved IRQL.

Notice that many drivers will simply raise the IRQL to block power failure interrupts during step 2, hence saving extra time by not setting the IRQL twice. The main reason that drivers might wish to use the above steps as written is that they might want to do more than simply manipulate device registers at raised IRQL. It is much better to do the work between steps 3 and 4 at device IRQL than it is at powerfail IRQL.

Programmers developing device drivers for **Windows NT** need not concern themselves with the particulars of how to synchronize access between the ISR and other parts of the device driver. The kernel provides a synchronization mechanism explicitly designed to aid in writing device drivers. When the device driver is being initialized, it creates an entity called an *interrupt object*. An interrupt object allows the device driver to describe to the system what IRQL its ISR should be associated with. This object can then be used in calls to another kernel-provided routine, **KeSynchronizeExecution**. This routine provides the above access synchronization, except for synchronization with power recovery interrupts.

Power recovery is handled through the use of two other kernel-provided objects, the *power status object* and the *power notify object*. The former object provides drivers with the ability to specify a Boolean variable that should be set in the event that the power has failed and then come back on with the contents of dynamic memory preserved. This allows drivers to provide the synchronization in step 4 above. The latter object provides drivers with the ability to register a routine that should be invoked should a power recovery occur. This gives the driver a chance to reinitialize its device, handle any problems that may have occurred if an operation was currently in progress, perform cleanup operations, etc.

Once the driver ISR has completed its interrupt service processing, most of the time it will need to do more work at a lower priority level, such as start another I/O operation or "complete" the current operation. This can be done by requesting the execution of a *Deferred Procedure Call* **(DPC)**. The DPC queue, a kernel-provided mechanism, allows the device driver to request the execution of a routine at a later time at a lower IRQL. The I/O system provides a set of routines that allow the device driver to use this mechanism. The following steps provide this functionality to the driver:

o - The device driver, in its initialization routine, initializes the DPC in its device object to specify the address of the routine that is to be executed when the DPC is requested. This is done through the use of the **IoInitializeDpcRequest** routine.

o - When the interrupt service routine wishes to request that a DPC routine be executed later, it invokes the **IoRequestDpc** routine. This routine uses the kernel-provided routine to insert the DPC into the system's DPC queue.

o - The DPC routine is executed after other higher level interrupts have been dismissed and the DPC queue is being processed. This may happen on any processor, including a different processor than the one on which the original interrupt occurred.

When the DPC queue interrupt occurs, at DISPATCH_LEVEL, the system-provided ISR examines the DPC queue to determine if there is any work to be performed. If so, then it removes a DPC entry from the queue and processes it. Processing consists of invoking the specified routine with a pointer to the DPC entry itself as well as the parameters specified in the call to **IoRequestDpc**.

It is at DISPATCH_LEVEL IRQL that the driver performs the majority of its work. It is here that the driver determines whether any errors have occurred, performs its error logging if needed, cleans up its context for the operation and determines whether there are more operations to do and starts them. All work that does not absolutely have to be performed in the ISR at device IRQL is done in this DPC routine.

The system provides a routine, **IoStartNextPacket**, that removes the next packet from the front of the device queue and returns a pointer to it to the caller, making the next IRP in the queue the "current" packet. If the device queue was empty, then the function returns a null pointer. This function assumes that the device busy flag is already set and will actually bugcheck the system if it is not set. If there was no other packet in the queue, then the busy flag is cleared.

Once the driver has completed its own processing, it then invokes the system routine, **IoCompleteRequest**, to complete the I/O operation. I/O completion, in general, consists of the following operations:

o - Unlock any buffers that were locked down for DMA I/O.

o - Copy any buffered data from system buffers into the user's buffer.

o - Copy the status and state information from the IRP into the user's I/O status block.

o - If an event was specified, set it to the Signaled state and dereference it.

- o - If no event was specified, set the file object to the Signaled state.

- o - Dereference the file object.

- o - If an APC was requested, queue it to the target process.

Copying the data and the I/O status block information must be performed in the context of the user's process so that its address space is accessible. Of course, signaling the event cannot occur until these two operations have completed since doing so causes a race condition.

The most important operation to complete as soon as possible is to unlock any buffers that have been locked into memory. Therefore, the first operation that the completion routine performs is to unlock the caller's buffers. This is done by invoking a memory management routine that unlocks the pages.

Once any buffers that need to be unlocked have been unlocked the completion routine queues a special kernel mode APC to the target process. When this APC executes, it finishes the tasks to be performed. It copies any buffered data that must be copied and copies the I/O status information into the requestor's I/O status block. It also sets either the file object or the event to the Signaled state and dereferences them, and queues the caller's APC, if one was specified. If the caller did not specify an APC then the I/O request packet is deallocated.

3. Driver Model Description

Device drivers in **Windows NT** are loaded either at system initialization or dynamically using a special device driver loader program. This loader is executed from a directory that can be protected from non-privileged but malicious users who might try to load bogus code into the privileged part of the system, thereby compromising the system. The device driver image files themselves are also loaded, by default, from a protected directory on the boot device so that non-privileged users cannot overwrite them.

The image file format for a device driver is no different than other normal programs in the system. It is an executable program with a transfer address.

The components of a driver that the system is most interested in are as follows:

o - The *initialization routine*. This routine is invoked once when the driver is loaded. It is responsible for any initialization that the driver must perform including its own data, the device, the controller, etc.

This routine creates the objects (see the discussion below) that the driver needs in order to be used by other drivers in the system, by user application programs, etc.

This routine is specified as the transfer address of the device driver.

This routine is also responsible for filling in the addresses of the other routines in the driver object. This allows the I/O system to locate the various entry points for the driver.

o - The *major function routines*. These routines specify the entry points in the driver for each of the major function codes that can be specified in an I/O request packet. They are invoked when the driver is called with an IRP. It is their responsibility to perform any parameter checking, etc. If the state of the packet is acceptable, then the routine starts the I/O operation.

o - The *start I/O routine*. This routine is invoked to actually start the I/O operation on a device. It is invoked either because the device is not busy and a request is ready to be performed, or because the current request is finished and now the next request can be started. Its responsibility is to actually start the request on the device.

o - The *unload routine*. This routine is responsible for cleaning up any data structures that the driver has, deallocating any pool, and closing any objects that it has opened. The system then frees the driver's code and data space, and marks it as gone from the system.

o - The *cancel routines*. These routines are invoked when a packet is to be canceled, and the packet is in such a state that simply setting its cancel flag will not cause the packet to be examined by the device driver. There is a cancel entry point that cooresponds to each state that a packet can be in. When a packet is marked for cancellation, the appropriate cancel routine is invoked to cancel the request.

As can be seen from this initial overview, device drivers do not really have a full context in the sense that a thread has a context. That is, they do not have their own address space or set of registers. A driver's context is the objects that it owns and the IRPs that it has access to via its device queue.

Device drivers execute in one of four different contexts depending on what part of the driver is executing and why. These contexts are as follows:

1. In the context of the user client thread. User threads request I/O by invoking the user APIs described in the *Windows NT I/O System Specification*. These routines validate the I/O operation, including probing the device-independent parameters and copying them into the IRP. The routines then invoke the device driver at its various function entry points to check the device-dependent parameters and to start the packet. If the device is already busy performing some operation, then the request may simply be queued to the driver.

   The driver receives the request from the system in the form of an IRP. This packet describes the user's parameters. The driver may then use the I/O subroutines at this point to aid in getting the operation started. These routines are discussed in a later section of this document.

2. In the context of the ISR. When a device interrupts the processor, the processor executes a system routine that invokes the driver's interrupt service routine. In some machine architectures this may happen directly. In others, it may take place through a common interrupt dispatcher that gains control and then passes control to the appropriate device driver(s). The kernel sets up the necessary data structures for the given architecture so the correct control flow takes place.

   The ISR must synchronize itself with the device driver start I/O routine. This is done through the use of the interrupt object and a routine, provided by the kernel, **KeSynchronizeExecution**. This synchronization object is necessary so that the driver's start I/O and interrupt service routines do not attempt to access the device at the same time.

3. In the context of the DPC interrupt. Once an ISR has completed the absolutely minimal amount of work required to satisfy the device interrupt, it requests that it be able to perform the remainder of the work needed to process the interrupt at a lower IRQL. This is done by requesting that it be invoked at its DPC routine at DISPATCH_LEVEL.

The kernel's DISPATCH_LEVEL interrupt service routine scans the queue of DPCs to be executed and invokes them in order. The driver's DPC routine should perform the majority of its work servicing the device in this routine. It should deal with checking for errors, making error log entries if appropriate, setting the device to a known state if needed, completing any handshaking required by the device, etc.

This routine is also responsible for starting the next I/O operation on the device if there are any in the device queue. This can be done by invoking the **IoStartNextPacket** function. This function checks, in a synchronized manner, for another IRP in the device queue. If one is found, the routine dequeues it and returns the address of the packet. The device driver can then perform whatever actions are necessary to get the operation started on the device.

The DPC routine is also responsible for completing the current I/O request. This is done by invoking the **IoCompleteRequest** function. This function actually completes the request by copying data, unlocking buffers, setting events, etc.

The DPC routine is probably the most important and certainly the most complicated part of a simple device driver. It is here where most of the time in the driver will be spent.

4. In the context of the "driver process". Some drivers may actually have a full driver process associated with them. Drivers that have associated processes, as will be seen more clearly later, actually have two parts: 1) The standard driver part described above, and 2) a process that has one or more threads, each complete with a virtual address space, general register set, etc. The **Windows NT** file system and network service drivers are built using this model.

Drivers deal primarily with several different system objects that are specific to drivers. These objects are as follows:

o - **Driver object.** Driver objects are created by the driver loader when the driver is initially loaded into the system. They are used by the system to determine where the entry points are for the driver, as well as for locating all of the controller and device objects that the driver is servicing. It also keeps track of where the code for the driver is loaded, its size, etc.

Device driver writers themselves do not generally manipulate the driver object itself. However, the object is used indirectly to locate entry points, etc., when calling between drivers or when the I/O system is calling the driver.

o - **Device object.** A device object is created by a driver for each device or device partition that the driver is to service. The collection of all of these objects, therefore, describes all of the devices in the system. Device objects contain information to allow the I/O system to manage I/O operations on the device, device characteristics, queue headers, etc. They also contain a device-specific area that the device driver can define. This allows the driver to maintain device-dependent context besides the IRPs with which it deals.

o - **Device queue.** A device queue is a kernel-provided object that allows the driver to synchronize a list of operations that need to be performed. Briefly, it provides the driver with a queue of IRPs to be executed, queue and dequeue routines, and a busy flag. Access to these items is synchronized by the kernel.

Device drivers do not normally manipulate the device queue directly; rather, they use the I/O system-provided routines to perform this work for them.

o - **Interrupt object.** Interrupt objects are created by the driver when associating itself with an interrupt vector, so that its interrupt service routine can be executed when the device interrupts. This is known as **connecting** to an interrupt. Interrupt objects, which are provided by the kernel, also provide the driver with a synchronization mechanism so that the driver can synchronize the execution of various critical sections of code with the driver's interrupt service routine.

o - **Controller object.** When the driver's initialization routine is invoked, it can create a controller object to represent the physical controller for its devices.

Controller objects are created by device drivers to allow synchronization of requests bound for devices through the controller to which the devices are physically attached.

o - **Adapter object.** Adapter objects are created by the I/O system when the system is initialized. An adapter object represents the system's mapping hardware. This object allows mapping registers to be allocated and set up so that scatter/gather operations can be performed to and from DMA devices. Device drivers use the adapter object in I/O system calls to allocate and initialize these mapping registers. In some systems, a device driver, via its device object, may be placed into a queue awaiting enough contiguous mapping registers to perform its transfer.

This object is also used to represent the channels on buses on some systems. The I/O system uses the adapter object to synchronize access to the channels by queueing the requesting device object to the adapter object.

Windows NT takes a "layered" approach to its driver design. That is, one driver may be layered on top of another device driver. This allows functionality to be added to the I/O system in such a way that device drivers themselves can be smaller and common code can be used for various types of devices.

For example, a device driver writer might want to implement a SCSI driver in one of two ways:

o - *The layered approach.* Using this approach, the writer would write a "port" driver and a "class" driver. The port driver would handle managing the controller itself and determining which device could be serviced based on requests being sent to it. The requests, of course, would be given to it in the form of an IRP, as for any driver.

The class driver would handle the devices themselves. For example, one class driver might drive the hard disks that are attached to the SCSI controller. Another class driver might handle a floppy disk, or a tape drive.

This approach is used in **Windows NT** to implement some of its drivers, including such drivers as the various file system drivers and network drivers. Requests, for

example, can be given to a file system driver that modifies the request, and then gives it to the device driver itself. This is analogous to having the file system driver be a class driver and the device driver being a port driver.

o - *The controller approach.* This approach allows the driver writer to use the controller object to provide synchronization rather than have a separate device object and device driver to do this. For many simple controllers and devices, this approach works best.

When a device driver which is using this approach wishes to start an I/O operation, it must allocate the controller object, start the I/O operation and then either deallocate the controller object if it is no longer needed (for example, a disk seek), or keep the object allocated until the operation is complete. When the operation is complete, such a driver deallocates the controller so that the next operation can be started.

Which of these two approaches a device driver developer uses is discretionary. Some more complex operations, such as a file system, clearly require something more than a controller object. A SCSI driver could be done using either approach with a fair amount of efficiency.

3.1. Time-Out Handling

Some device drivers must be concerned with operations timing out, whether because the device itself has timed out or because the user has asked that an operation be completed in a certain amount of time and it did not finish within the allocated time.

The **Windows NT** kernel provides driver writers with a means to timing operations. It does this through an object termed a *timer object*. A timer object can be initialized by calling the **KeInitializeTimer** kernel function. This routine initializes a timer object that can then be set to an expiration time using the **KeSetTimer** function. Expiration times may either be expressed in an absolute or a delta time format. The set timer routine also optionally accepts a *Deferred Procedure Call* **(DPC)** object address. This allows the specified routine to be executed when the timer expires. Finally, the kernel provides the **KeCancelTimer** function that allows a timer to be canceled.

The I/O system provides a simple interface to the structures and routines provided by the kernel. The interface allows the device driver to specify the address of a routine to

be executed once every second. This routine is provided with a context pointer that it uses to access its own data structures. One of these structures can contain a counter that is decremented each time the routine is invoked. If the counter is ever decremented to zero, then the operation is considered to have timed out. Whenever a new operation is started on the device, the counter can be set to the number of seconds that the operation has before it should time-out. If no operation is in progress, then the counter can be set to some predetermined value, such as minus one. This allows the timer routine to determine that no operation is being performed and it should not modify the counter.

The I/O system provides a device driver object with a built-in timer with the above functionality. The routines used to support this interface are as follows:

o - The **IoInitializeTimer** routine. This function accepts the address of the routine that is to be executed and the address of a device object. The **IoInitializeTimer** routine initializes the device object's timer for use when it is started.

o - The **IoStartTimer** routine. This routine starts the timer; that is, it causes the routine specified in the above call to be invoked once every second. The driver-specified routine will be invoked with the context parameter that it specified in the call to the **IoInitializeTimer** routine.

o - The **IoStopTimer** routine. This routine stops the timer; that is, it will stop invoking the driver's timer routine once every second. The **IoStopTimer** routine is generally only invoked if the driver is being unloaded from the system.

3.2. Power Recovery

Drivers must also deal with the possibility of power failures and recoveries. When a power failure interrupt occurs, the kernel saves enough state so that when the power is restored the system can be restarted from wherever it was executing when the power failed. Special considerations must be made for device drivers because of the hardware associated with them.

The concerns here are three-fold:

1. Device drivers must be given the chance to reinitialize the devices that they are servicing. This allows the devices to be put back into a well known state so that operations may once again be requested. Controllers also need to be reinitialized and placed into well known states.

2. Device drivers must deal with operations that were currently in progress when power was lost. For most devices, this probably means restarting the I/O request once the device has been reinitialized.

3. Device drivers must deal with synchronization of power recovery interrupts and of executing code that touches the device itself. That is, the device driver should ensure that once it is ready to actually tell the device to begin a transfer, it does this atomically. This means locking out power failure interrupts for a short period of time. If a power failure interrupt occurs between the time that the device driver synchronized itself with its start I/O routine (at device interrupt request level - DIRQL) and the time that it was ready to begin writing device registers, then it would need to abort the operation and not tell the device to perform it. This is because if the power failed while writing device registers and then came back up, the device could perform an erroneous operation if the transfer were restarted.

Likewise, the device driver needs to synchronize with power failures occurring while the device driver is in its device interrupt service routine or in a DPC routine that touches the device registers. In both of these cases the driver needs to determine whether or not a power failure has occurred so that it does not inadvertently "complete" an I/O request based on unpredictable device register contents.

For these reasons, the **Windows NT** kernel provides device drivers with two different objects that can be used to deal with power failures. These two objects are as follows:

o - **Power Notify Object.** This object allows the driver to establish a DPC that is to be invoked whenever the system's power is restored. This allows the device driver to be asynchronously notified when the power has been restored by having the system call one of the driver's routines. This routine should deal with the first two concerns above. It should handle the device/controller reinitialization and it should restart or fail the "current" request.

The kernel provides a routine to initialize a power notification object **(KeInitializePowerNotify)**, a routine to insert the power notification object into a queue so that the associated routine can be invoked **(KeInsertQueuePowerNotify)**, and a routine to remove the object from a queue **(KeRemovePowerNotify)**.

o - **Power Status Object.** This object and its associated functions are provided to deal with the third concern. This object provides the driver with a flag that can be tested to determine whether or not the power has failed. The flag is set to TRUE if the power has failed and recovered; otherwise it is set to FALSE.

The kernel provides routines to initialize a power status object **(KeInitializePowerStatus)**, a routine to register the power status object **(KeInsertPowerStatus)**, and a routine to unregister the power status object **(KeRemovePowerStatus)**.

The model for recovering from a power fail interrupt is to use the power notification object and two power status objects. The two status objects should be used as follows:

1. The first power status object should be used to indicate that the power has failed. This power status object will hereafter be termed the *power failed* object.

2. The second power status object should be used to indicate whether or not the power notification routine has finished its processing of the power recovery. This power status object will hereafter be termed the *power recovery routine not done* object.

When the power fails, both of the power status objects will be set to TRUE. This indicates to the driver that the power has failed and the power recovery routine has not finished its processing. At well-defined points in the driver, both of these status objects should be checked to determined whether either is TRUE. If **either** of the objects is TRUE, then the driver should not continue processing of requests. That is, the driver should not allow new requests to be processed. For example, the driver's start I/O routine, interrupt service routine, and dispatch routines should check to ensure that no power failure processing is occurring in the driver. This is done by raising the current

IRQL to POWER_LEVEL and check the logical OR of the two status objects. If either is TRUE, then processing should not continue for the device.

The power notification routine performs four functions:

1. The first step is to set the power failed object to FALSE. This indicates that the driver is aware that the power has failed and recovered.

2. The second step is to perform any device initialization that is necessary to allow normal operations to continue on the device.

3. The third step is to set the power recovery routine not done object to FALSE. This indicates that all power recovery for the device has been completed and operations can be continued normally.

4. The final step is to restart the current packet for the device, if one was active. The *CurrentIrp* field of the device object is a pointer to the packet that was currently being processed by the driver.

3.3. Canceling I/O

I/O operations can be canceled in one of two ways:

1. All of the I/O for a thread can be canceled by invoking the **IoCancelThreadIo** subroutine. This routine scans the IRP list for a thread and cancels the I/O operation represented by each IRP in the list. This function is useful in thread rundown and termination. It is invoked from kernel mode in the executive and is not available to general users.

2. All pending operations issued by the calling thread for a file handle can be canceled by invoking the **NtCancelIoFile** service. This service performs the same operation as above except that only the thread I/O for the file associated with the specified file handle is canceled. All other I/O is unaffected.

\\ *This functionality in the I/O system is currently being redesigned. This design will be complete and implemented for the next version of this specification.* \\

3.4. Driver Layering

As mentioned earlier, drivers in the **Windows NT** I/O system may be layered. This allows one driver to communicate with another driver by simply calling it and passing it a pointer to an IRP. This feature allows the system programmer to add functionality to the system in many broadly or narrowly focused ways.

Consider, for example, a file system and a simple disk driver. The file system is represented by a file system driver and the disk driver is represented by a device driver. Both have device objects which are named so that they can be referenced from outside themselves. The file system sees the disk that the device driver presents as a stream of 512-byte blocks that are referred to by numbers, 0 through *n*.

In this simple case, when the user attempts to read part of a file from the file system, the file system accepts the request and changes it into a request to the disk driver for whatever blocks on the disk that need to be read.

IRPs are set up to handle exactly this type of layering. Each packet contains a fixed portion that contains information about the original request, which thread the request belongs to, event pointers, etc. The remainder of the packet is an array of structures that is treated as a "stack". That is, each layer in the chain of drivers owns one of the structures in the array. In our example then, the disk driver would own "stack location" number one and the file system would own "stack location" number two.

These stack locations contain the following information:

- o - A major and minor function code. These function codes are used to tell the driver what type of operation to perform and how to interpret the remainder of the information in the structure.

- o - Parameters. These are the parameters that the driver uses, based on the function codes, to perform the specified operation. Up to four separate 32-bit longwords of arguments can be passed, as well as an 8-bit flags byte.

- o - I/O system information. There is also some information that is maintained by the I/O system that describes information about the driver.

As each layered driver is invoked down the chain, the "current stack location" is adjusted so that it points to the proper structure for the next driver. As each driver is invoked, it may decide that it would like to be invoked on certain conditions once the I/O

operation is complete. For example, the file system performing the read on behalf of the user may wish to know if an error occurred during the read operation so that it knows that there is a potentially bad block in the file.

A driver can register a "completion routine" through the use of the **IoSetCompletionRoutine** function. This function accepts the following parameters:

o - A pointer to the IRP that is currently being worked on.

o - The address of a routine to be invoked when all of the layers below the current driver have completed the request.

o - A context parameter which can be used by the driver for whatever extra information it requires.

o - Three flags which indicate whether the completion routine is to be invoked if the operation is successful, if it completes with an error, or if the operation is being canceled.

Once the drivers below the current driver, in our example the disk driver, completes the operation, the file system's completion routine is invoked, if one was specified. It is invoked with the following parameters:

o - A pointer to the driver's device object, which contains the information about the volume or device on which the I/O operation is taking place.

o - A pointer to the IRP. The current stack location in the packet is the one that belongs to the current driver.

o - The context pointer that was passed in the above function call.

If the driver does not wish to be invoked at all, then it need not call this I/O function.

Layering of drivers also yields the ability for the system programmer to insert layers of drivers between each other to provide different functionality to the system. In the previous scenario, for example, a driver can be inserted between the file system and the disk driver which interfaces to many disks rather than to a single disk. It could then present all of the disks to the file system as a single large disk with lots of blocks, still referenced as 0 to *n*. Note that since this driver is completely file-system-independent, it could be used with any file system type. So, this single driver could be used as an

enhancement that provided tightly coupled volume sets to several different file systems without ever touching the sources of the file system driver.

4. File System Description

File systems in **Windows NT** are executed as layered drivers with the special exception that they have an actual process associated with them. These processes are referred to as *file system processes*, or **FSPs**. Having these processes gives file systems the ability to not only be able to execute in the context of a requesting thread, or to execute without concern as to what thread they are executing in, but it also gives them the ability to perform such operations as waiting on an object without causing the client thread to wait.

File system processes are like most other user processes in the system except that they run entirely in kernel mode. These types of processes are referred to as *kernel processes* in **Windows NT**. They can make direct calls to I/O functions and to their own file system drivers **(FSDs)**. They can also allocate system pool, etc.

The communication between an FSD and an FSP occurs through a *communication region* that is set up by the FSD during its initialization. This communication region contains a queue, a spin lock to protect the queue, and an event. The spin lock and the queue are used in conjuction to provide the functionality of an interlocked queue. The event, generally autoclearing, is used to notify the FSP when an item has been placed into the queue.

The queue provides a location for the FSD to place IRPs that need more processing than the FSD itself is capable of performing in its limited context. Other shared data used between the FSD and the FSP may be contained in this region. How it is structured is up to the file system writer. The communication region need not be statically allocated within the image. It can be allocated as part of the device extension when the device object is created.

To synchronize access to the queue, the FSD performs the following steps:

- o - Inserts the IRP into the queue using the **ExInterlockedInsertTailList** function.

- o - Sets the event to the Signaled state.

The FSP synchronizes access to the queue by performing the following steps:

o - Wait for the event to be set to the Signaled state and reset it if it is not an autoclearing type event.

o - Remove the entry from the head of the queue using the **ExInterlockedRemoveHeadList** function.

o - Loop, removing entries from the head of the queue until there are no more entries remaining.

It should be noted that it is possible for the FSP to be awakened for an entry in the queue that it has already removed. Therefore, it must be able to handle the situation where there are no entries in the queue to be processed.

Using this queueing method allows the FSD to pass information to the FSP. FSPs, on the other hand, can communicate with FSDs by simply invoking routines in the FSD. Because the FSD is thread-context-independent, this is basically no different than the FSD being invoked by the system to perform a request on behalf of a user thread.

The two parts of the file system, the FSD and the FSP, both reside in the same image file on the disk. When this image is loaded, the loader maps the entire image into system space. The FSD, in its initialization, sets up the data structures and communication region that it will use to communicate with the FSP and creates the FSP.

Both parts of the file system share data in the communication region described above. Also in the communication region, or perhaps pointed to by it, are the data structures that allow the FSD to determine which files have been opened, what data is in the cache, etc.

Among this shared data is also a description of how blocks in files are mapped to blocks on the disk itself. The data structure supported by the **Windows NT** I/O system to describe this is termed a *Map Control Block* **(MCB)**. Not all file systems need to use MCBs, as the on-disk structure may already be well structured enough to describe the mapped blocks efficiently.

An MCB describes the extents of a file in an array of structures composed of the following two longwords of information:

o - Virtual Block Number (VBN). The starting block within the file that the extent represents.

o - Logical Block Number (LBN). The starting block on the disk where the extent resides.

A header on the data structure describes the maximum number of entries in the MCB, the current number of valid entries in the MCB, and the block number of the extent in the file which represents the end of the file. It should be noted that this structure also works well for describing sparse files and can be searched with a binary search.

If the FSD attempts to translate a file block number into a disk block number and the entry required to perform the mapping is not currently in the MCB, then the FSD must have the FSP perform a *window turn* operation on the MCB. That is, the FSP must adjust the entries in the MCB so that the file extent to be mapped is actually described in the MCB entries. The FSP is needed for this operation because the amount of shared data necessary to allow the FSD to perform this operation would be too large for some file systems. The FSP is also needed so that no implicit wait operations are required in the context of the client thread.

\\ *The set of interface routines for providing this functionality has yet to be designed. Since it is required fairly soon in the implementation of the project, it will be provided in the next revision of this document.* \\

4.1. IFS Design

Multiple file systems may be active in **Windows NT** at any given time. These file systems might be servicing multiple devices or they might be servicing different partitions on the same device.

When a file system is loaded, it registers itself as a resident file system that is eligible to be invoked when a volume is to be "mounted". This is accomplished by creating a device object whose type is *FILE_DEVICE_FILE_SYSTEM* and invoking the

IoRegisterFileSystem function with the returned device object. This function inserts the device object onto the list of active file systems.

The loader sees a file system driver the same as it sees a device driver, as a simple program. The transfer address of a file system driver is its initialization routine.

The entry points that the I/O system is interested in for a file system driver are as follows:

o - *Initialization routine*. This FSD routine initializes the file system's data structures, allocates and initializes the communication region, creates a device object for the file system, creates the FSP with at least one thread executing, and registers itself as a file system.

When the FSP is created, it is passed the address of the device object. Its responsibility is to initialize itself and set up any other structures that may be used between the two components, if any.

The FSD routine must also fill in the driver object routine address fields with its entry points so its routines can be located.

o - *Major function routines*. These routines correspond to the major function codes in an IRP. They are selectively given control when the IRP is handed to the driver. The file system supplies a routine entry point for each of the major functions that it implements. All others are defaulted by the I/O system to a routine that returns an error code indicating that the request is not implemented by the driver.

The major function routines are responsible for validating the parameters in the I/O request packet and determining what should be done to perform the request. This might mean copying data from the file system's cache, or performing a window turn on the file, etc. If more processing is required, then the FSD might give the packet to its FSP.

o - *Unload routine*. This routine is invoked by the system when the file system is being unloaded. Its responsibilities are to let the FSP know that it should clean up and exit, and then to clean up any data structures the FSD has, deallocate its

communication region, unregister itself as an active file system, and close its objects. The system then frees the file system's code and data space and marks it as gone from the system.

o - The *cancel routines*. These routines are invoked when a packet is to be canceled and the packet is in a state such that simply setting its cancel flag will not cause the packet to be examined by the driver. There is a cancel entry point that cooresponds to each state that a packet can be in. When a packet is marked for cancellation, the appropriate cancel routine is invoked to cancel the request.

Once the media in a device has been recognized by an active file system, a record is kept of which file system owns the media. When a request for that device is issued, the system first gives the request to the file system for processing. This file system can then give the request to the device driver, if required.

The structure that is used to keep track of which file system is currently servicing a device is called a *volume parameter block* **(VPB)**. This structure contains the volume label and serial number that the file system returned when the volume was mounted. If a request is made for a volume that is no longer in the drive, then the system hard error routine will be invoked to request that the user place the appropriate media back into the drive so the operation can continue.

A file system may request that it be permitted to perform post-processing work in the FSD after the device driver has completed the I/O operation. This is accomplished by using the its stack location in the I/O request packet, as described in an earlier section of this document on *Driver Layering*.

4.2. Mapped File I/O

\\ *This section is somewhat out-of-date. The current revision, 1.1, has cleaned up the obsolete statements made in previous versions which were very old. As the design evolves, this section will be updated to reflect the latest thinking on this area.* \\

It is possible in **Windows NT** for users to perform I/O to a file by simply reading and writing memory. A set of library routines are provided to support the interfaces that set this up for the caller. Mapped I/O can be set up by calling the following functions:

o - **NtCreateFile(**FH, ...**);**

o - **NtCreateSection(**SH, ..., FH**);**

o - **NtMapViewOfSection(**SH, ...**);**

Once the file has been created or opened (via **NtCreateFile** or **NtOpenFile**), a section is created to the file. The user then creates the section for the file by invoking the **NtCreateSection** service. When this service is invoked, it must be invoked with the maximum size that the file will ever grow to before the section is closed.

> \\ *This will need to be fixed. The section must be able to grow if people are to realistically use sections for files. An editor, for example, cannot determine how large a file will grow when the user initially opens it.* \\

The section is then mapped using the **NtMapViewOfSection** service. This service allows the user access to the file actually backing up the section. Reads and writes to the file occur implicitly by simply reading or writing the memory that is mapped by the view to the section.

Likewise, the user may still use the file handle returned from the **NtCreateFile** (or **NtOpenFile**) service to perform other operations on the file such as reading and writing it. In this case, the file system simply maps a view to the section for the file into the system virtual address space and then reads or writes the data from/to its memory.

There is a design note, **Windows NT Mapped I/O Design Note**, that further discusses this mapped I/O model and other models that were considered. See this document if there are questions regarding mapped I/O.

4.3. File Caching

Caching in the file system is provided through the *Cache Manager*. This component of the system provides file system drivers with the ability to map files into the system virtual address space so that they can be managed by the file system through the use of the memory manager. More information on the Cache Manager can be found in the *Windows NT Cache Design Note*.

The amount of I/O system-specific support required to provide this functionality to file systems is minimal. In fact, the only feature required is the addition of *stream file objects*. Stream file objects allow a file system to represent various parts of an on-disk structure that are not in proper files as files in the I/O system. That is, a file is cached through the memory manager by creating a section that is backed by the file itself. Once the section is created, the file system simply needs to map a view to the part of the file that it requires access to, and then touch that virtual address space. This causes a pagefault to occur for the mapped file. The memory manager then performs a read or write operation to the file based on the type of access the file system made to the memory location.

In order to allow EAs or ACLs for files to be cached, the file system needs a way to describe the portions of the disk structure that contain this data. Since this data may not be in the file itself, a virtual file concept is used to describe and map the data as if it were part of a normal file. Stream file objects are used to represent these abstractions.

A stream file object can be created to represent any part of the on-disk structure that the file system needs access to through the cache. Creating a stream file object is accomplished by calling the **IoCreateStreamFile** function. This function creates a stream file and returns a pointer to it. The pointer can then be used by the file system to create a memory section as it would for a normal file.

4.4. Splitting Transfers

Sometimes a user requests a file transfer that spans multiple extents in the file. The FSD can determine this by attempting to map the offset in the file via the MCB for the file. When this occurs, the file system needs some way of splitting the request into several different transfers. This section describes the three options that a file system writer has available.

All of the following models begin with a request (IRP), being passed to the appropriate major function routine in the FSD. Once this routine recognizes that the I/O request requires multiple transfer requests to the device, it uses one of the following models. Each model is described using a scenario where the operation is a read and there is no caching involved in the transaction to keep it as simple as possible. Other scenarios may be extrapolated from the models without much effort, so they are not exhaustively elaborated here.

4.4.1. FSP Model

The FSP model uses the FSP to split a transfer into multiple device requests. The FSP splits the transfer by performing the following steps:

o - The FSD places the IRP into the interlocked work queue of the shared communication region and sets the event associated with the queue to the Signaled state for the FSP.

o - The FSP's wait on the event is now satisfied so it removes packets from the interlocked work queue.

o - It determines the number of individual requests that are required to satisfy the original request and fills in a counter location in the original IRP with that number.

o - For each individual contiguous transfer from the device, the FSP performs the following steps:

- Allocates an IRP,

- Fills in the IRP with the information to describe the partial request,

- Marks the IRP as an associated IRP using the **IoMakeAssociatedIrp** function,

- Fills in the starting LBN for this extent,

- Fills in the length of this extent,

- Allocates and initializes an MDL for the next part of the requestor's buffer using the **IoAllocateMdl** and **IoBuildPartialMdl** functions.

- Calls the appropriate device driver to perform the function using the **IoCallDriver** function.

Each time a requested transfer completes, the device driver completes the IRP which causes the system's I/O completion routine to be invoked. This routine sees that the IRP is an associated IRP and decrements the counter in the original IRP. The final I/O status is also formed from each I/O request packet.

Once all of the IRPs are complete, denoted by the original counter going to zero, the completion code "completes" the original request. Notice that until the original request is actually completed, the completion code need not context switch back to the original requestor. Likewise, no context switches to the FSP are required.

This model causes the various pieces of the transfer to be processed in a quasi-parallel fashion because the FSP can queue multiple transfer requests to the device driver. This allows the device driver to process the packets more quickly.

A variation on this model is to use an algorithm where the FSP simply queues one request for each extent, one at a time to the driver. The FSP then requests that it be notified explicitly when the request completes. This allows the FSP to keep state information about the original IRP rather than setting the associated IRP pointer in the new IRPs that it queued.

This variation would work best if the FSP used APC routines in its own threads to synchronize the I/O operations, thereby using fewer system resources by not dedicating a thread to each request.

4.4.2. FSD Parallel Model

This model is similar to the FSP model except that, rather than have the FSD context switch to the FSP to let it allocate the IRPs and queue them to the device driver, the FSD would perform all of the steps itself. It would perform the allocation and initialization of the IRPs while still executing in the context of the user's thread.

While this model works well for some small number of situations, it further complicates the FSD code and causes more state information to be kept in the FSD.

This model is still considered quasi-parallel however, because it allows the file system to queue multiple requests to the device driver without having to wait for any one operation to complete.

4.4.3. FSD Serial Model

This model is similar to the FSD model except that rather than have the FSD allocate, initialize, and queue multiple request packets, it simply allows the FSD to reuse the original request packet over and over again until the entire transfer request has been satisfied.

This is done by storing the original request information in the packet as context information and then changing the MDL, length, and starting LBNs according to the next extent of the file.

The stack location completion routine for the FSD in the IRP can be filled in so that it gets a chance to execute when the request has been completed by the device driver. When this routine is invoked, the FSD sees that this is a multiple-transfer IRP, fills in the next request information, and gives it back to the device driver again. This continues until the request has been completed.

Once the entire original request has been satisfied, the FSD calls the normal completion code to complete the operation.

This model has the advantage that it requires the least amount of processing overhead. No IRPs are being allocated and initialized from scratch. On the other hand it is potentially slower because it is serialized. If the I/O were being done to multiple spindles, such as a striped file, this method could actually lower the throughput of the I/O system even though the amount of processing overhead is less.

4.5. Mounting and Volume Verification

Windows NT supports multiple file systems running at the same time. This imposes three basic requirements on **Windows NT**:

o - Automatic media recognition. When the media in a device is first accessed, the system must be able to determine which file system is supposed to deal with that media. That is, when a user makes a request such as an open, read, write, etc., operation on a device that needs the support of a file system, the system must be able to determine which file system is to handle the request.

o - Supporting removable media. Because the system supports removable media, such as floppy disks, it must also be able to dynamically change its idea about which file system is currently supporting the device.

For example, if the user switches the media in a floppy drive, the system must be able to determine whether the on-disk file structure on the new media is the same as on the previous media. If it is not, then **Windows NT** must determine which file system understands the new media and associate the new file system with that media.

o - Supporting multiple partitions. Finally, the system must be able to support multiple partitions on hard drives. This means that it must also be able to support cases when each partition is a different on-disk structure.

These three requirements mean that when a device is first accessed or when a device is being accessed whose media may have changed, the system must locate the appropriate file system for the device. This is done by keeping track of the media that the system thinks is currently in the drive. The structure used to do this is a called a *volume parameter block* **(VPB)**. This structure keeps the volume label and the volume serial number of the media. It also keeps track of the file system's device object associated with the volume.

Initially, the VPB associated with a drive is blank; that is, it has no name and a zero (an invalid value) for its serial number. When a volume is first touched, the system realizes that the information in the VPB is invalid and begins invoking registered file systems to determine which one recognizes the structure on the media. It does this by passing the FSD an IRP with a function of "mount".

The file system generally performs the following steps to determine whether or not it should successfully complete the mount request, thereby "owning" the volume on the device:

o - It begins by giving the IRP to a thread in the FSP which can perform I/O to the device.

o - The FSP reads whatever blocks are required to determine whether or not it recognizes the on-disk structure.

o - It then either completes the request with an error if the structure was unrecognizable, or it continues.

o - If the FSP continues, the FSP creates a device object which will be used to hold its context for this volume and initializes the file system-specific data structures in the newly created device object.

o - The FSP then fills in the device object field of the VPB with the address of the newly created device object.

o - The FSP then creates a thread to handle this volume, passing it the address of the newly created device object.

o - Finally, the FSP completes the original mount I/O request packet with a success status, indicating that the media format was recognized.

If the file system does not recognize the structure on the media as its own, then the system then continues through the list of registered FSD's looking for a file system that recognizes the media. If none is found, then the RAW file system, the last in the list, takes over the media and treats it as non-formatted media. This file system provides a way of reading and writing the device. It recognizes all media.

Upon completion of a mount, the I/O system mount code checks the status of the operation, and if it is successful, it associates the VPB with the device. This is accomplished by having a pointer in the device object that contains the address of the VPB. The VPB currently being pointed to represents the system's idea of the file system structure in the drive.

5. Network Service Description

\\ *This section is currently under design review in the Windows NT Network Group. The current thinking is that there will be four to five layers: Redirectors, Transports, Networks, Datalinks, and Physical links. Each layer will be able to be invoked from the layer above it*

or directly by the user with no differences. There can be any number of drivers at any layer. Further, some layers may be subsumed by another layer.

This section will also discuss how names are resolved.

More information on Networks will be available in the next release of this specification. \\

6. I/O Completion

I/O completion consists of a routine which drivers invoke and a special kernel APC routine internal to the I/O system. This section presents how each is used and the functions that each performs.

Once a driver has finished all of its processing for an I/O request, it invokes the **IoCompleteRequest** function to actually complete the I/O request. The device driver normally invokes this routine when it is at DISPATCH_LEVEL in its DPC queue routine.

IoCompleteRequest begins by checking the IRP stack to determine if there are any other drivers that need to be notified that the I/O completed. If there are, then it invokes each driver's completion routine if one was specified in the stack. The function determines whether or not there are more drivers that need to be notified by comparing the count of stack entries to the current entry number. If they are different, then there is another driver to be notified.

The routine determines whether or not to invoke the driver's completion routine based on whether the cancel I/O flag is set, the success or failure of the status code in the I/O status block of the IRP, and whether the routine is to be called for each case depending on what was specified in the call to **IoSetCompletionRoutine**.

The algorithm that I/O completion uses to determine whether the routine should be invoked is as follows:

```
if (Irp.CancelIo AND InvokeOnCancel)
                        OR
  ((NT_SUCCESS(Irp.IoStatus.Status)) AND InvokeOnSuccess)
                        OR
  ((!NT_SUCCESS(Irp.IoStatus.Status)) AND InvokeOnError)
```

```
/* invoke the completion routine */
```

A driver may not wish to be invoked for any of the above reasons. If this is the case, then it need not take any action since the I/O system guarantees that the flags are zeroed in the stack when the driver is initially invoked.

Once all of the device drivers have been invoked, the I/O completion routine checks to see if any pages were locked into memory for this operation. This is done by querying the *MdlAddress* field of the IRP. If any MDLs are associated with the IRP the I/O system walks the list and unlocks all of the pages associated with each of the buffers described by the MDLs.

Once any pages that were locked are unlocked a special kernel mode APC is initialized and queued to the target thread using the kernel APC interface routines. If the thread is in an appropriate state, then it will be scheduled to execute. The address of the routine to be executed as the special kernel APC is part of the I/O completion code.

Once the special kernel mode APC begins execution in the context of the target thread, it performs the following steps to complete the I/O request:

o - All buffered data, if any, is copied from the system space buffers into the user's buffers. If the system space buffers were temporary buffers allocated to transfer this information, then they are deallocated.

Buffered data here also refers to any data that is OUT in any API, with the exception of the I/O status block. That is, any data in the output buffer interface such as the **NtQueryDirectoryFile** service, for example, is copied into the user's buffers.

o - Any MDLs used to describe the user's buffers are deallocated.

o - The I/O status information in the IRP is copied to the user's I/O status block.

o - The file object is set to the Signaled state, if no event was specified, and it is dereferenced.

o - The specified event for this request, if any, is set to the Signaled state and dereferenced.

o - If no APC was requested by the I/O initiator, then the IRP is deallocated.

o - If an APC was requested, an APC object is initialized and queued to the thread. The IRP is deallocated once the APC has been removed from the thread queue by the kernel.

Some system features cause special considerations in how I/O completion works. IRPs that have been linked together by a file system, for example, are handled differently to gain better performance and because the file system isn't generally interested in the completion notification unless the operation fails. Many times it is only concerned when all of the operations have completed and sometimes even then it is not interested directly.

Paging I/O also requires special consideration during I/O completion. Such considerations include the following:

o - No pagefaults may occur on any path of pagefault code during I/O completion.

o - No special kernel mode APCs may be taken in the context of the target thread because this could cause another pagefault to occur. Moreover, APCs are blocked anyway. Therefore, the general mechanism for completion cannot be used.

o - Caching in the file system is disabled for paging I/O.

o - No APC is delivered to the target process to "complete" the paging I/O request. This can all be done without actually entering the thread's context. The pager's I/O status block is in non-paged memory in system space and will never incur a pagefault. It is always visible regardless of what thread context the system happens to be in when the I/O completion code is executing.

o - The pages that were being paged in do not need to be unlocked by the I/O system since they will be unlocked by the pager.

7. Error Logging and Handling

Windows NT provides both special error handling for users, and an error logging facility for drivers. This section explains how both work and how they should be used from the driver's point of view.

7.1. Error Logging Facility

Error logging in **Windows NT** is supported by the following components:

o - I/O support routines. These routines provide drivers with an interface to the error process. Communication with this process occurs through the use of datagrams sent to its port that contain the information that the driver would like to write to the error log file. These routines allow the driver to allocate a datagram, fill it in, and send it to the error log process's port.

o - The I/O system thread. This thread removes error log buffer entries from the pending queue and sends them to the error log process as datagrams. It then frees the entries back into the buffer pool. This thread uses the standard interlocked work queue and event method of synchronization used by FSD/FSP drivers.

o - The error log process. This process maintains a port to which error log datagrams can be sent. It is the responsibility of this process to take the datagrams and write their contents to the error log file. This process is also responsible for maintaining the file itself. An old file can be opened, an old file can be closed and a new one created, or a new file may be created. Finally, it is the responsibility of this process to write time stamps to the file. These are written in such a way that if no actual error log entries are written between time stamps, then only one time stamp is entered in the file. This saves disk space by minimizing the amount of data actually written to the file.

o - The error format utility **(EFU)**. The EFU has the responsibility of reading error entries out of the error log file and displaying the contents in a form that is understandable by service personnel.

Because there are many different types of devices in the system and some will be supported by device drivers other than the standard drivers that are part of the **Windows NT** operating system, device driver writers can write error log buffer translation routines that the EFU can invoke when an entry is found for the specified device.

Each error log entry contains a header that specifies the type of entry being formatted. This header also contains a device type identifier field that is filled in by the driver through the I/O support routines. This is a unique name that is declared by the device driver. When the EFU discovers the entry, it invokes the entry in the DLL that corresponds to the entry to format it. All of the images for formatting the entries are contained in the *\error* directory in the **Windows NT** directory tree. The names of the images are the same as the device type identifier field. That is, a ".DLL" is appended to the device type identifier field and that dynalink library is invoked to format the entry.

The EFU passes the DLL routine a pointer to the entry to be formatted as well as the address of a routine that can be invoked to output whatever information needs to be output for formatting purposes.

The name of the entry point in the DLL for the EFU to invoke is **FormatEntry**.

The I/O support routines related to error logging are as follows:

o - **IoAllocateErrorLogEntry** - This routine allocates an error log entry for the device driver. The header information is automatically filled in by the routine. The driver can then place the error information into the entry. If there are no available error log entries in the system, then a null pointer is returned.

o - **IoWriteErrorLogEntry** - This routine sends the specified error log entry to the error log process to be written to the error log file.

\\ *There also needs to be an API like NtCloseErrorLogFile that users with the appropriate privilege can call to have the error log process close the current error log file and optionally open a new one.* \\

7.2. Error Ports

Users may wish to be notified when a device operation is going to fail because the wrong disk is in a drive or because of some other error that requires intervention on behalf of the user. The OS/2 subsystem also requires this functionality in order to emulate the "fail on error" flag.

Windows NT provides this type of functionality by giving the user the option of providing an "error port" on a create or open to a file or device. This port allows the I/O subsystem to RPC to the user when an error occurs to let him determine what he would like to do about it. That is, if the wrong volume is in a device, for example, an RPC message is sent to the user's port in order to determine what he'd like to do about the situation. The user may return one of the following:

o - Retry - Retry the operation. In this case, the caller presumably communicated the problem to the user and asked that the correct volume be placed in the device. The operation is simply retried by sending the IRP back through the system.

o - Abort - Abort the operation. In this case, the I/O request is simply aborted and the operation is completed with the appropriate error status.

The I/O system uses this common model with the system hard error thread. This thread has a globally known port that receives an RPC when an error such as the above is encountered. The hard error routine does exactly what was described above in order to have the situation resolved.

8. Terminal I/O Considerations

This section addresses those special requirements needed by terminal devices, especially dealing with unsolicited input and conventions used by subsystems to perform terminal input operations that they need to support.

8.1. Unsolicited Input

The operating system emulation subsystems, the Session Manager (SM), and the Terminal Logon Process need to be able to be notified when an unsolicited input occurs on a terminal which, by definition, is not logged into the system. To do this, there is an

NtDeviceIoControlFile service that allows a user with sufficient privilege to specify a port to be given a message when an unsolicited input occurs on a terminal.

The Terminal Logon Process can therefore open the terminal, issue this service and then close the terminal, and not keep any more state until something happens on the terminal.

When someone enters a termination or interrupt character on the terminal, the terminal driver will use the **IoSendMessage** routine to send a message to the specified port. This routine allocates and queues a kernel mode APC to the I/O completion thread. The routine specified as the APC will be a routine in the I/O system that translates the name of the port and sends a datagram message to it specifying the name of the terminal device on which the unsolicited input occurred. The type of the message sent is **PORT_TERMINAL_INPUT**.

The Terminal Logon Process can then open the terminal and begin performing the standard login sequence.

8.2. Subsystem Input

The model used for terminal input by subsystems that are emulating different APIs varies, but the following recommendations help system performance and, in some cases, provide the subsystem with an easier task:

o - Subsystems should attempt to perform all terminal input operations on buffers that are long enough to contain the average terminal line size.

o - Subsystems should avoid performing single character I/O to terminals or other devices when possible.

o - Subsystems may wish to have one dedicated thread that performs terminal input. This thread can process the data by placing it into internal queues and emulating the PM keyboard model of I/O. This allows the subsystem to have one interface to these types of devices.

o - Subsystems should make use of the device I/O control service to set the termination characters for a device to those characters generally recognized by

the API they are trying to emulate. They should also include those characters that are considered out-of-band characters for their API. This allows the subsystem to scan characters written to its buffer from the terminal driver and determine when these character have been seen.

9. I/O Data Structures and Objects

This section gives a brief overview of the data structures and objects used in the **Windows NT** I/O system. It describes what the data structures are, how they are used, and in some cases a description of the major fields in the data structures.

All of the data structures in this section contain a type field which is used by the I/O system for robustness. This allows the system to check to ensure that a pointer really points to the type of structure to which it is supposed to point.

9.1. I/O Request Packet Description

The I/O Request Packet (IRP) is the primary data structure used in the I/O system to pass information from system services to drivers, from drivers to other drivers, and from drivers back to the I/O system. IRPs are always allocated from some part of nonpaged memory since they are sometimes accessed at raised IRQL.

An IRP is allocated with an array of structures that are associated with each of the drivers in the I/O system hierarchy required to complete the specific I/O request. Each of these structures, called "stack locations", contain function and parameter information for each driver. (For more information see the section in this document on *Driver Layering*.)

The primary fields of an IRP include the following:

o - File object. This field points to the file object that the request is being performed on.

o - MDL. This field points to the MDL(s) associated with the I/O operation. The MDL(s) describe the buffer or buffers being used for the operation in terms of both their virtual addresses and physical page numbers. MDLs also describe the length of the buffer.

o - User service-independent parameters. These fields contain information about the standard parameters to I/O system services. An example is a field that contains a pointer to the referenced event object that the user specified in a service call.

o - Thread. This field contains a pointer to the thread that originally requested the I/O operation.

o - I/O status. This field contains the final status of the operation. It is copied into the user's I/O status block variable when the operation is complete.

o - Flags. This field describes to the various drivers and I/O system subroutines the type of operation that is represented by the IRP. For example, *IRP_INPUT_OPERATION* is a flag that indicates that this is an input operation and buffer copying during I/O completion needs to take place between a system allocated buffer and the user's buffer.

o - IRP stack location management. These fields keep track of which stack location in the IRP stack is the current location and how many locations are in the stack.

o - Device queue entry. This structure is used to queue to IRP to the device object device queue.

o - APC entry. This structure is used to allow the IRP to be used as an APC. It may either represent the special kernel APC used in I/O completion or the caller's APC routine.

9.2. Volume Parameter Block

A Volume Parameter Block (VPB) is used to keep track of the volume in a specific device. It also associates the volume with a specific file system that is currently managing the volume.

The major fields of a VPB are as follows:

o - Volume label. This field stores the name of the volume. This is useful when a removable volume is accessed but the media has been taken out of the drive. It gives the system a way to refer to the volume that the user can understand.

o - Volume serial number. This field stores the serial number of the volume. This is useful when two or more volumes mounted in the system have the same name. The serial number uniquely identifies the volume. A serial number of zero is invalid.

o - Real device object. This field is a pointer to the device object of the physical device itself on which the media is currently mounted.

o - Device object. This field is a pointer to the file system's device object for the volume. It is set by the file system after a successful mount operation.

9.3. File Object

A file object is the object used to represent files in **Windows NT**. These objects are created by the object management parse routine for device objects in the I/O system when a file is being opened or created. They represent the actual file itself.

The major fields of a file object are as follows:

o - Device object. This field is a pointer to the device object on which the file resides. This field is interrogated by the I/O system to determine which driver should be invoked when the file is being accessed through I/O system services. If the device object has a VPB associated with it, then the device driver for the VPB's file system device object is invoked rather than the real device's driver.

o - VPB. This field is a pointer to the Volume Parameter Block for the volume that the file resides on, if any.

o - File system context. These fields are reserved for use by the file system. They are undefined for I/O system use.

o - File name. This field contains the volume-relative name of the file.

o - Synchronization objects. These fields are used to control caller synchronization to the file. They allow the caller to wait on the file handle, for example.

9.4. Driver Object

A driver object is used to represent the driver code and data. It is used to keep track of the entry points for the driver as well as where the driver is currently loaded. Driver objects are used by the I/O system and the Configuration Manager.

The major fields of a driver object are as follows:

o - Entry points. These fields keep track of the routine entry points in the driver.

o - Device object. This field is a list head of all of the device objects that are being serviced by this device driver. This list represents the number of reasons why the driver cannot be unloaded from the system. The driver cannot be unloaded from the system until all of its device object have been deleted.

o - Driver object. This field is a pointer to the next driver object in the system. It is used to link all of the drivers in the system together.

o - Driver base. This field contains the base system virtual address of the driver itself. It is used when the driver is being unloaded.

o - Driver size. This field contains the size of the driver. It is used when the driver is being unloaded.

9.5. Device Object

A device object is a permanent object used to represent a physical, logical, or virtual device. The collection of all device objects represents all known devices in the system. Device objects are used by system services and drivers.

The major fields of a device object are as follows:

o - Reference count. This field represents the number of reasons why this particular device object cannot be deleted.

o - Driver object. This field points to the driver object for this device. It is used by the I/O system to locate the entry points to the driver so that it may be invoked by system services and other drivers.

o - Device objects. These fields link this device object to other device objects for this driver and for devices attached to this device.

o - Current IRP. This field is a pointer to the IRP that this driver is currently working on if it is busy.

o - Timer information. These fields are used by the I/O system to implement the timers for the driver.

o - Device extension. This field is a pointer to the driver-specific extension to the device object.

o - Device type. This field specifies the type of device that the object represents.

o - Device queue. This structure is used to allow IRPs to be queued to the device using kernel synchronization and worked on.

9.6. Controller Object

A controller object represents a hardware controller. It is used by drivers to synchronize access to the controller by various devices.

The major fields of a controller object are as follows:

o - Wait queue. This structure allows device objects to be queued and dequeued to/from the controller using kernel-provided synchronization.

9.7. Adapter Object

An adapter object represents a hardware bus adapter or DMA controller. It is used to synchronize access to the hardware.

The major fields of an adapter object are as follows:

o - Channel information. These fields describe the channels on the adapter.

o - Map register information. These fields describe the map registers in the adapter.

o - Map register allocation control. These fields keep track of the allocation of map registers in the adapter.

o - Wait queue. This structure allows device objects to be queued and dequeued to/from the adapter using kernel-provided synchronization.

10. I/O System APIs

The APIs described in this section are used by various components of the I/O system. Some are used by executive services, some are used by file systems, some are used by I/O subroutines, and some are used by drivers. All of the functions must be called from kernel mode.

Some of these functions are implemented as separate procedures, some as inline routines, and some as C language macros.

This section describes the following APIs:

IoAbortInvalidRequest - Abort an invalid I/O Request Packet.
IoAllocateAdapterChannel - Allocate adapter channel and execute routine.
IoAllocateErrorLogEntry - Allocate error log entry.
IoAllocateIrp - Allocate I/O Request Packet.
IoAllocateMdl - Allocate a Memory Descriptor List.
IoAsynchronousPageWrite - Write page data to the paging file asynchronously.
IoAttachDeviceByName - Attach two device objects.
IoBuildAsynchronousFsdRequest - Build asynchronous I/O Request Packet for an FSD.
IoBuildFspRequest - Build I/O Request Packet for an FSP.
IoBuildPartialMdl - Build a partial Memory Descriptor List.
IoBuildSynchronousFsdRequest - Build synchronous I/O Request Packet for an FSD.

IoCallDriver - Invokes a driver at its major function entry point.
IoCancelThreadIo - Cancel all I/O for a thread.
IoCheckDesiredAccess - Check desired access against granted access.
IoCheckFunctionAccess - Check function access against granted access.
IoCheckShareAccess - Check shared access request to a file.
IoCreateController - Create a controller object.
IoCompleteRequest - Complete an I/O request.
IoCreateDevice - Create a device object.
IoCreateFile - Create/open a file.
IoCreateStreamFile - Create a stream file object.
IoDeallocateAdapterChannel - Deallocate an adapter channel
IoDeallocateController - Deallocate a controller.
IoDeallocateIrp - Deallocate an I/O Request Packet.
IoDeallocateMdl - Deallocate a Memory Descriptor List.
IoDeleteController - Delete a controller object.
IoDeleteDevice - Delete a device object.
IoDeregisterFileSystem - Deregister driver as an active file system.
IoDetachDevice - Detach two device objects.
IoFlushAdapterBuffers - Flush adapter buffers to memory or device.
IoGetAttachedDevice - Get pointer to highest level attached device.
IoGetCurrentIrpStackLocation - Get pointer to IRP stack location.
IoGetNextIrpStackLocation - Get pointer to next IRP stack location.
IoGetRelatedDeviceObject - Get device object related to specified file object.
IoGetRequestorProcess - Get process of I/O requestor.
IoInitializeDpcRequest - Initialize a DPC for later posting.
IoInitializeTimer - Initialize a one-second timer.
IoIsOperationSynchronous - Determine whether an I/O operation is synchronous.
IoMakeAssociatedIrp - Allocate and initialize an associated IRP.
IoMapTransfer - Map an I/O transfer in DMA controller.
IoPageRead - Build a page read request packet for the pager.
IoQueryInformation - Query information about a file.
IoRegisterFileSystem - Register driver as an active file system.
IoRemoveShareAccess - Remove the share access information when a file is closed.
IoRequestDpc - Request a DPC routine execution.
IoSendMessage - Send terminal input message to port.

IoSetCompletionRoutine - Set completion routine and context in IRP stack.
IoSetShareAccess - Set share access information for a file open.
IoStartNextPacket - Start the next I/O Request Packet, if any.
IoStartPacket - Start the current I/O Request Packet if device not busy.
IoStartTimer - Start a one-second timer.
IoStopTimer - Stop a one-second timer.
IoSynchronousPageWrite - Write page data to the paging file synchronously.
IoUpdateShareAccess - Update share access information for a file.
IoWriteErrorLogEntry - Queue error log entry to be written to log file.

10.1. IoAbortInvalidRequest

A driver may abort an invalid I/O Request Packet using the **IoAbortInvalidRequest** function:

VOID
IoAbortInvalidRequest(
IN PIRP *Irp*
);

Parameters:

Irp - A pointer to the IRP that represents the I/O request.

The **IoAbortInvalidRequest** function is invoked to abort a request represented by an IRP when the packet specifies an invalid operation. For example, this routine is invoked when an attempt is made to read from a nonexistent sector on a disk device. This function causes the entire operation to be aborted. That is, the initiator's I/O operation is not completed normally, if the request has not already been successfully queued at a driver level above the current driver that invoked the function.

This function must be invoked at DISPATCH_LEVEL.

Once this function has been invoked the IRP is no longer accessible to the driver that made the call.

10.2. IoAllocateAdapterChannel

A driver may allocate an adapter object channel and cause its execution routine to be invoked using the **IoAllocateAdapterChannel** function:

VOID
IoAllocateAdapterChannel(
IN PADAPTER_OBJECT *AdapterObject,*
IN PDEVICE_OBJECT *DeviceObject,*
IN ULONG *NumberOfMapRegisters,*
IN PDRIVER_CONTROL *ExecutionRoutine,*
IN PVOID *Context*
);

Parameters:

AdapterObject - A pointer to the adapter object that represents the adapter channel to be allocated.

DeviceObject - A pointer to the device object that represents the device on which the I/O is to be performed.

NumberOfMapRegisters - Supplies the number of map registers to be allocated. If zero, only the adapter is allocated and no map registers are allocated.

ExecutionRoutine - The address of a routine to be executed once the adapter channel has successfully been allocated.

Context - A context parameter to pass to the *ExecutionRoutine* when it is invoked.

The routine specified by the *ExecutionRoutine* parameters has the following type definition:

typedef
IO_ALLOCATION_ACTION
(*PDRIVER_CONTROL) (
IN PDEVICE_OBJECT *DeviceObject,*
IN PIRP *Irp,*

IN PVOID *MapRegisterBase,*
IN PVOID *Context*
);

Parameters:

DeviceObject - A pointer to the device object specified as the *DeviceObject* parameter in the call to the **IoAllocateAdapterChannel** function.

Irp - A pointer to the current I/O Request Packet that the device is working on.

MapRegisterBase - A pointer to the base address of the first map register allocated. If no map registers were allocated, then the pointer's value is null.

Context - A pointer to be used as context for the routine. This value of this parameter is the same as the *Context* parameter in the call to the **IoAllocateAdapterChannel** function.

The **IoAllocateAdapterChannel** function allocates an adapter object for the channel specified by the *AdapterObject*, waiting if necessary. If the caller requested a set of map registers, then the number of registers are also allocated. The *DeviceObject* is potentially queued to the adapter object wait queue if the channel is busy, or if the number of requested map registers cannot be immediately granted.

Once the adapter channel, and potentially the map registers, have been allocated, the driver's *ExecutionRoutine* is invoked.

The execution routine may return a value that indicates whether or not the channel and/or map registers are to remain allocated to the device. If they are, then the driver must subsequently invoke the appropriate routine to explicitly deallocate them.

This function is used by device drivers that service devices that are referenced through a bus adapter or a DMA controller.

10.3. IoAllocateController

A driver may allocate a controller object and cause its execution routine to be invoked using the **IoAllocateController** function:

VOID
IoAllocateController(
IN PCONTROLLER_OBJECT *ControllerObject,*
IN PDEVICE_OBJECT *DeviceObject,*
IN PDRIVER_CONTROL *ExecutionRoutine,*
IN PVOID *Context*
);

Parameters:

ControllerObject - A pointer to the controller object that represents the physical device controller to be allocated.

DeviceObject - A pointer to the device object that represents the device on which the I/O is to be performed.

ExecutionRoutine - The address of a routine to be executed once the controller has successfully been allocated.

Context - A context parameter to pass to the *ExecutionRoutine* when it is invoked.

The routine specified by the *ExecutionRoutine* parameters has the following type definition:

typedef
IO_ALLOCATION_ACTION
(*PDRIVER_CONTROL) (
IN PDEVICE_OBJECT *DeviceObject,*
IN PIRP *Irp,*
IN PVOID *MapRegisterBase,*
IN PVOID *Context*
);

Parameters:

DeviceObject - A pointer to the device object specified as the *DeviceObject* parameter in the call to the **IoAllocateController** function.

Irp - A pointer to the current I/O Request Packet that the device is worked on.

MapRegisterBase - A reserved pointer that should be set to null.

Context - A pointer to be used as context for the routine. This value of this parameter is the same as the *Context* parameter in the call to the **IoAllocateController** function.

The **IoAllocateController** function allocates the controller specified by the *ControllerObject* parameter. The *DeviceObject* is potentially queued to the controller object wait queue if the controller is busy.

Once the controller has been allocated, the driver's *ExecutionRoutine* is invoked.

The execution routine may return a value that indicates whether or not the controller is to remain allocated to the device. If so, then the driver must subsequently invoke the appropriate routine to explicitly deallocate it.

This function is used by device drivers that service devices that are referenced through a physical device controller.

10.4. IoAllocateErrorLogEntry

An error log entry buffer may be allocated using the **IoAllocateErrorLogEntry** function:

PVOID
IoAllocateErrorLogEntry(
IN PDEVICE_OBJECT *DeviceObject,*
IN UCHAR *EntrySize*
);

Parameters:

DeviceObject - A pointer to the device object to be associated with the error log entry.

EntrySize - The size of the buffer to be allocated. This value must be less than the maximum number of bytes in an entry buffer as specified by ERROR_LOG_MAXIMUM_SIZE.

The **IoAllocateErrorLogEntry** function allocates an error log entry buffer and returns a pointer to it. The size of the buffer allocated may be specified by the *EntrySize* parameter. This parameter specifies the size in bytes and must be less than the maximum size of an error log buffer.

The device driver may then fill in the data buffer and use the **IoWriteErrorLogEntry** function to post the entry to the error log thread. The device driver may treat the data as anything it likes provided that it does not overflow the size of the buffer.

10.5. IoAllocateIrp

An I/O Request Packet may be allocated and initialized using the **IoAllocateIrp** function:

PIRP
IoAllocateIrp(
IN CCHAR *StackSize,*
IN BOOLEAN *ChargeQuota*
);

Parameters:

StackSize - Specifies the number of stack locations needed in the IRP. This value should equal the number of layers in the chain of layered drivers servicing this request.

ChargeQuota - Specifies whether the current thread should be charged quota for the pool memory used to allocate the IRP. This flag should only be specified by system services. File system processes should not charge quota for associated IRPs used to implement a function.

The **IoAllocateIrp** function allocates and initializes an I/O Request Packet (IRP). It allocates the packet so that it contains *StackSize* stack locations at the end of the packet for use in layering drivers.

10.6. IoAllocateMdl

An MDL (Memory Descriptor List) may be allocated and initialized using the **IoAllocateMdl** function:

PMDL
IoAllocateMdl(
IN PVOID *VirtualAddress,*
IN ULONG *Length,*
IN BOOLEAN *SecondaryBuffer,*
IN BOOLEAN *ChargeQuota,*
IN OUT PIRP *Irp* **OPTIONAL**
);

Parameters:

VirtualAddress - Specifies the base virtual address of the buffer that the MDL is to describe.

Length - Specifies the length of the buffer starting at *VirtualAddress* that the MDL is to describe, in bytes.

SecondaryBuffer - Indicates whether this buffer is a primary or secondary buffer. This determines how the MDL will be linked into the IRP, if specified. All buffers except for the first buffer described by an MDL in an IRP are considered secondary buffers.

ChargeQuota - Indicates whether quota should be charged to the current thread for the non-paged pool that is allocated to contain the MDL.

Irp - Optionally specifies a pointer to an IRP that the MDL is to be associated with. If this parameter is specified, then the MDL is linked into the IRP's MDL list according to the value of *SecondaryBuffer*.

The **IoAllocateMdl** function allocates and initializes a Memory Descriptor List. The MDL is allocated in such a way that it can later be used to map the buffer; that is, there is enough storage to contain the Page Frame Numbers (PFNs) that map the buffer into

physical memory. The PFNs themselves are not initialized. The MDL header is initialized to describe the specified buffer.

This function is used by the I/O system to map the caller's buffers. It is also used by any device driver that needs to break a buffer into parts, each mapped by an MDL, or to map a separate buffer. Mapping a complete buffer may be used to when a driver is given a pointer to the caller's buffer and needs to lock it, or when it needs to lock a buffer that the driver has allocated.

10.7. IoAsynchronousPageWrite

The modified page writer can asynchronously write pages of data to the paging file or to a mapped file using the **IoAsynchronousPageWrite** function:

NTSTATUS
IoAsynchronousPageWrite(
    IN PFILE_OBJECT *FileObject,*
    IN PMDL *MemoryDescriptorList,*
    IN PLARGE_INTEGER *StartingOffset,*
    IN PIO_APC_ROUTINE *ApcRoutine,*
    IN PVOID *ApcContext,*
    OUT PIO_STATUS_BLOCK *IoStatusBlock*
    );

Parameters:

FileObject - A pointer to a referenced file object representing the file to write.

MemoryDescriptorList - A Memory Descriptor List (MDL) that describes the locked-down buffer containing the data to write to the file.

StartingOffset - The starting byte offset where the write operation is to begin.

ApcRoutine - The address of an APC routine that is to be executed once the I/O operation is complete. This is the only valid synchronization technique for this type of request.

ApcContext - A value that will be given to the caller's *ApcRoutine* when it is invoked.

IoStatusBlock - A variable to receive the final completion status and information about the write operation. The number of bytes actually written is returned in the *Information* field.

The **IoAsynchronousPageWrite** function gives the **Windows NT** Modified Page Writer a quick way of building and starting an I/O request to write data to a file asynchronously. This allows I/O completion to be short circuited for paging I/O.

The function writes the number of bytes specified by the MDL from the buffer described by the MDL, beginning at the *StartingOffset* within the file. The *ApcRoutine* is invoked once the operation has completed.

This function is only invoked by the **Windows NT** Modified Page Writer.

10.8. IoAttachDeviceByName

A device object may be attached to another device object to allow association between the two using the **IoAttachDevice** function:

NTSTATUS
IoAttachDevice(
IN PDEVICE_OBJECT *SourceDevice,*
IN PSTRING *TargetDevice*
);

Parameters:

SourceDevice - A pointer to the device object that should be attached. This device object must belong to the calling driver.

TargetDevice - The name of the device that the *SourceDevice* should be attached to for servicing.

The **IoAttachDevice** function allows a device driver to attach a device object to another device object. This association allows operations given to the lower level device driver to be given to the device driver for the *SourceDevice* device object. It also establishes the

layering between drivers so that the same IRP may be used by all layers. If the device has already been attached to, then this function returns an appropriate error status.

This service is used by intermediate device drivers. It allows a driver to attach a device object to another device in such a way that any requests being made to the original device will now be given to the intermediate device driver's device object. For example, a file system normally interfaces directly to a disk device driver. This function allows an intermediate driver, such as a striper driver, to attach itself to the disk driver's device object such that when the file system attempts to communicate with the disk driver via its device object, the request will be routed to the intermediate device driver first.

10.9. IoBuildAsynchronousFsdRequest

A file system driver may build an asynchronous I/O Request Packet for use in performing a read or a write to another device driver using the **IoBuildAsynchronousFsdRequest** function:

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

Parameters:

MajorFunction - The function that the FSD is requesting the lower level driver to perform. This value must be one of *IRP_MJ_READ* or *IRP_MJ_WRITE*.

DeviceObject - A pointer to the device object that represents the target of the read or the write operation. The device driver for this device will be invoked with the IRP that this function builds.

Buffer - A pointer to a buffer that, on a write contains the data to write, or, on a read is to receive the data read.

Length - Specifies the length, in bytes, of the data to be read or written. (For many devices, such as disks, this value must be an integral of 512.)

StartingBlock - Specifies the byte offset that the transfer is to begin. (For many devices, such as disks, this value must specify a sector boundary.)

IoStatusBlock - A variable to receive the final status and information from the operation. The final status will be written to the *Status* field and the number of bytes read or written will be contained in the *Information* field of this variable once the operation has completed. This variable must be in a nonpageable page.

The **IoBuildAsynchronousFsdRequest** function builds an I/O Request Packet (IRP) that can be given to a device driver to perform an asynchronous read or a write operation. The IRP contains only enough information to get the operation started and to complete to the FSD. No other context information is kept track of since the request is context-independent.

It is up to the FSD to determine when the I/O has completed, if it is interested. This can be done by setting a completion routine in the returned IRP using the **IoSetCompletionRoutine** function.

This function is used by file system drivers operating in a thread-context-independent manner to issue I/O requests.

10.10. IoBuildFspRequest

A file system process may build an I/O Request Packet for use in performing a read or a write to another device driver using the **IoBuildFspRequest** function:

PIRP
IoBuildFspRequest(
IN ULONG *MajorFunction,*
IN PDEVICE_OBJECT *DeviceObject,*
IN OUT PVOID *Buffer,*
IN ULONG *Length,*
IN PLARGE_INTEGER *StartingOffset,*

IN PKEVENT *Event* **OPTIONAL,**
IN PIO_APC_ROUTINE *ApcRoutine* **OPTIONAL,**
IN PVOID *ApcContext* **OPTIONAL,**
OUT PIO_STATUS_BLOCK *IoStatusBlock*
);

Parameters:

MajorFunction - The function that the FSP is requesting the target device to perform. This value must be one of *IRP_MJ_READ* or *IRP_MJ_WRITE*.

DeviceObject - A pointer to the device object that represents the target of the read or the write operation. The device driver for this device will be invoked with the IRP that this function builds.

Buffer - A pointer to a buffer that contains the data to write, or is the location that is to receive the data read.

Length - Specifies the length, in bytes, of the data to be read or written. (For many devices, such as disks, this value must be an integral of 512.)

StartingOffset - Specifies the byte offset at which the transfer is to begin. (For many devices, such as disks, this value must be the start of a sector boundary.)

Event - An optional pointer to a kernel event that should be set to the Signaled state once the operation completes.

ApcRoutine - Optionally specifies the address of an APC routine that should be executed upon completion of the request. This routine is invoked with the address of the I/O status block and the *ApcContext* value as its parameters.

ApcContext - Optionally specifies a context parameter that should be passed to the *ApcRoutine* when it is invoked upon completion of the I/O operation.

IoStatusBlock - A variable to receive the final status and information from the operation. The final status will be written to the *Status* field and the

number of bytes read or written will be contained in the *Information* field of this variable once the operation has completed.

The **IoBuildFspRequest** function builds an I/O Request Packet (IRP) that can be given to a device driver to perform a read or write operation. The IRP closely resembles an IRP that would be built by the general **NtReadFile** or **NtWriteFile** services. This packet can be modified by the FSP before the FSP invokes the device driver with the request, if necessary.

Completion of the I/O request occurs as for normal I/O requests, except that this packet refers to a kernel event, whereas a normal packet refers to an event object.

This function is used by file system processes (FSPs) operating in their own context and issuing I/O operations on behalf of I/O system users.

10.11. IoBuildPartialMdl

A Memory Descriptor List (MDL) may be built to describe part of a buffer described by another MDL (i.e., "master") by using the **IoBuildPartialMdl** function:

VOID
IoBuildPartialMdl(
IN PMDL *SourceMdl,*
IN OUT PMDL *TargetMdl,*
IN PVOID *VirtualAddress,*
IN ULONG *Length,*
);

Parameters:

SourceMdl - A pointer to the MDL that describes the original buffer, a subset of which is to be mapped by this function.

TargetMdl - A pointer to an MDL to be filled in that describes the desired subset of the buffer specified by the *SourceMdl* parameter. The MDL must be large enough to contain the PFNs required to map the subset buffer.

VirtualAddress - Specifies the base virtual address of the *TargetMdl* buffer to be mapped; this value must be contained within the buffer mapped by the *SourceMdl*.

Length - Specifies the length in bytes to be mapped by the *TargetMdl*; this value in combination with that of *VirtualAddress* must specify a buffer that is a proper subset of the *SourceMdl* buffer. If specified as zero, then the function maps the remainder of the *SourceMdl* buffer, starting at *VirtualAddress*, as the *TargetMdl*.

The **IoBuildPartialMdl** function maps a subset of a buffer that is currently mapped by the *SourceMdl*. The *VirtualAddress* and *Length* parameters describe the mapped subset. The descriptor to map the specified subset is written to the *TargetMdl*. If a length of zero is specified, then the remainder of the specified buffer is mapped starting at *VirtualAddress*. See section 4.4, *Splitting Transfers*, for an overview of how this function is used.

10.12. IoBuildSynchronousFsdRequest

A file system driver may build a synchronous I/O Request Packet for use in performing a read or a write to another device driver using the **IoBuildSynchronousFsdRequest** function:

PIRP
IoBuildSynchronousFsdRequest(
IN ULONG *MajorFunction,*
IN PDEVICE_OBJECT *DeviceObject,*
IN OUT PVOID *Buffer,*
IN ULONG *Length,*
IN PLARGE_INTEGER *StartingOffset,*
IN PKEVENT *Event,*
OUT PIO_STATUS_BLOCK *IoStatusBlock*
);

Parameters:

MajorFunction - The function that the FSD is requesting the lower level driver to perform. This value must be one of *IRP_MJ_READ* or *IRP_MJ_WRITE*.

DeviceObject - A pointer to the device object that represents the target of the read or the write operation. The driver for this device will be invoked with the IRP that this function builds.

Buffer - A pointer to a buffer that contains the data to write, or is the location that is to receive the data read.

Length - Specifies the length, in bytes, of the data to be read or written. (For many devices, such as disks, this value must be an integral of 512.)

StartingOffset - Specifies the byte offset that the transfer is to begin. (For many devices, such as disks, this value must specify the start of a sector boundary.)

Event - A pointer to a kernel event that is to be set to the Signaled state once the operation completes.

IoStatusBlock - A variable to receive the final status and information from the operation. The final status will be written to the *Status* field and the number of bytes read or written will be contained in the *Information* field of this variable once the operation has completed. This variable must be in a nonpageable page.

The **IoBuildSynchronousFsdRequest** function builds an I/O Request Packet (IRP) that can be given to a device driver to perform a synchronous read or write operation. The IRP contains only enough information to get the operation started and to complete to the FSD.

It is up to the FSD to determine when the I/O has completed by waiting on the *Event*. It should be noted that performing the wait operation causes the current thread to wait. Therefore this operation should be used during initialization of the driver or when the I/O being performed is synchronous.

10.13. IoCallDriver

The I/O system or a layered driver may invoke a driver at its major function entry using the **IoCallDriver** function:

NTSTATUS
IoCallDriver(
IN PDEVICE_OBJECT *DeviceObject,*
IN OUT PIRP *Irp*
);

Parameters:

DeviceObject - A pointer to the device object upon which the I/O request is to be performed. The *Irp* is given to the driver that is servicing the device.

Irp - A pointer to the I/O Request Packet that represents the request to be performed.

The **IoCallDriver** function invokes a driver's major function routine according to the IRP function code. The driver that is called is the one that is servicing the device specified by *DeviceObject*. The IRP specified by *Irp* is passed to the driver at its appropriate entry point.

This function is used by I/O system services and layered drivers.

Once this function has been invoked the IRP is no longer accessible to the driver.

10.14. IoCancelThreadIo

All pending I/O operations for a given thread may be canceled by using the **IoCancelThreadIo** function:

VOID
IoCancelThreadIo(
IN PETHREAD *Tcb*
);

Parameters:

Tcb - A pointer to the Thread Control Block for the thread whose pending I/O operations should be canceled.

The **IoCancelThreadIo** function chases all of the pending I/O requests for the specified thread and cancels each one. Some requests may already be in a state of being completed and these are allowed to complete. Pending operations that are not in a state of being completed are canceled if they can be. Most of the operations that would normally occur during I/O completion, such as unlocking buffers and dereferencing events, are performed so that thread rundown occurs properly.

This function is used by the executive during thread rundown.

10.15. IoCheckDesiredAccess

A driver can check whether a desired access is permitted to a file using the **IoCheckDesiredAccess** function:

NTSTATUS
IoCheckDesiredAccess(
IN OUT PACCESS_MASK *DesiredAccess,*
IN ACCESS_MASK *GrantedAccess*
);

Parameters:

DesiredAccess - Supplies a pointer to the access mask that represents the type of access to the file that is desired.

GrantedAccess - Supplies the access mask that represents the access already granted to the file.

The **IoCheckDesiredAccess** function checks whether or not the caller has the desired access rights to a file based on the current granted access mask. If not, then an access denied error status is returned. Otherwise, a successful status is returned and the *DesiredAccess* variable is overwritten with an expanded representation of the actual access desired. That is, all generic accesses to the file are expanded to individual access bits.

This function is used as a security filter by kernel mode components that are accessing files for user mode clients. Since no access checks are made for kernel mode requests,

a server system needs a way of determining whether or not its client has the appropriate access to perform a given function on the file. The **IoCheckDesiredAccess** routine is used to provide this functionality.

10.16. IoCheckFunctionAccess

A driver can check whether an operation is permitted to a file using the **IoCheckFunctionAccess** function:

NTSTATUS
IoCheckFunctionAccess(
    IN ACCESS_MASK *GrantedAccess,*
    IN UCHAR *MajorFunction,*
    IN UCHAR *MinorFunction,*
    IN PFILE_INFORMATION_CLASS *FileInformationClass* **OPTIONAL,**
    IN PFS_INFORMATION_CLASS *FsInformationClass* **OPTIONAL**
    );

Parameters:

GrantedAccess - Supplies the access mask that represents the access granted to the file.

MajorFunction - Supplies the IRP major function code that represents the operation to be performed on the file.

MinorFunction - Supplies the IRP minor function code that represents the operation to be performed on the file, if any.

FileInformationClass - Supplies the file information class for the set or query operation to be performed on the file. This parameter is optional if the function code does not specify a set or query operation.

FsInformationClass - Supplies the file system information class for the set or query volume operation to be performed. This parameter is optional if the function code does not specify a set or query volume operation.

The **IoCheckFunctionAccess** function checks whether or not the caller has the access rights to a file to perform a specific function given his granted access. If not, then an access denied error status is returned. Otherwise, a successful status is returned.

This function is used as a security filter by kernel mode components that are accessing files for user mode clients. Since no access checks are made for kernel mode requests, a server system needs a way of determining whether or not its client has the appropriate access to perform a given function on the file. The **IoCheckFunctionAccess** routine is used to provide this functionality.

10.17. IoCheckShareAccess

A file system may check whether shared access is permitted to a file using the **IoCheckShareAccess** function:

NTSTATUS
IoCheckShareAccess(
IN ACCESS_MASK *DesiredAccess,*
IN ULONG *DesiredShareAccess,*
IN OUT PFILE_OBJECT *FileObject,*
IN OUT PSHARE_ACCESS *ShareAccess*
IN BOOLEAN *Update*
);

Parameters:

DesiredAccess - Supplies the types of access that the current open request would like to the file. This parameter is generally the same *DesiredAccess* parameter given to the file system by the I/O system when the open request is made via the create file IRP.

DesiredShareAccess - Supplies the types of shared access that the current open request would like to the file. This parameter is generally the same *ShareAccess* parameter given to the file system by the I/O system when the open request is made via the create file IRP.

FileObject - A pointer to the file object for the current open request.

ShareAccess - A pointer to the common share access data structure associated with the file being opened. This structure is treated as an opaque type by drivers.

Update - Supplies a BOOLEAN value indicating whether the share access information for the file is to be updated if the open request is permitted.

The **IoCheckShareAccess** function checks a file open request to determine if the types of desired and shared accesses specified are compatible with the way in which the file is currently being accessed by other opens of the file.

File systems maintain state about files through structures called File Control Blocks **(FCBs)**. The **SHARE_ACCESS** is a structure that describes how the file is currently accessed by all opens. It is contained in the FCB as part of the open file state.

If the requestor's access to the file is compatible with the way in which the file is currently open, a status of *STATUS_SUCCESS* is returned and the **SHARE_ACCESS** information for the file is updated according to the *Update* parameter. If the file request is denied because of a file sharing violation, then a status of *STATUS_SHARING_VIOLATION* is returned.

This function is used by file systems, after ensuring that the requestor has access to the file, to determine whether or not the open request can be satisfied according to the **Windows NT** file sharing semantics.

10.18. IoCompleteRequest

The processing for an I/O request may be declared complete using the **IoCompleteRequest** function:

VOID
IoCompleteRequest(
IN PIRP *Irp,*
IN CCHAR *PriorityBoost*
);

Parameters:

Irp - A pointer to the I/O Request Packet that represents the I/O operation to be completed.

PriorityBoost - Specifies the amount of priority boost the requesting thread should be given when the special kernel APC is queued to it for I/O completion.

The **IoCompleteRequest** function "completes" an I/O operation for the request packet that represents it. Completing an operation involves notifying all drivers in the IRP stack that the I/O operation has completed, provided that they would like to know (that is, they invoked the **IoSetCompletionRoutine** function to set the address of a completion routine). It also involves unlocking the caller's buffers, posting the event if one was specified, queueing the APC to the requesting thread if one was specified, as well as other operations.

This function must be invoked at DISPATCH_LEVEL.

This function is used by drivers to indicate that the I/O operation has completed and the driver is finished with its processing.

10.19. IoCreateController

A driver can create a controller object using the **IoCreateController** function:

```
PDEVICE_OBJECT
IoCreateController(
    );
```

Parameters:

None.

The **IoCreateController** function allocates and initializes a controller object for use by a driver. The controller object is used to synchronize access to various hardware devices connected to the controller hardware. For more information on controller objects and how they are used see the *Driver Model Description* section of this document.

10.20. IoCreateDevice

A driver can create a device object using the **IoCreateDevice** function:

PDEVICE_OBJECT
IoCreateDevice(
IN PDRIVER_OBJECT *DriverObject,*
IN ULONG *DeviceExtension,*
IN PSTRING *DeviceName* **OPTIONAL,**
IN DEVICETYPE *DeviceType,*
IN BOOLEAN *Exclusive*
);

Parameters:

DriverObject - Supplies a pointer to the driver object created by the I/O system when the driver was loaded. This object indicates which driver is to be associated with the device object being created.

DeviceExtension - The size, in bytes, of any extension that should be allocated beyond the end of the I/O system's notion of the device object. This part of the device object can be used by the device driver to contain context information or a communication region for use in communicating between an FSD and an FSP.

DeviceName - An optional pointer to a string that describes the name of the device that the device object represents. This name is associated with the device object and inserted in the object directory hierarchy.

DeviceType - The type of the device that the device object represents. The values for this parameter must have the same representation and meaning that they have in the **NtQueryInformationFile** system service.

Exclusive - Indicates that this device is created as an exclusive device; that is, once the device object is "opened" by one process, no other processes may open the device.

The **IoCreateDevice** function allocates and initializes a device object for use by a driver. The object is a permanent object, is exclusive if the *Exclusive* parameter is TRUE, and is "owned" by the device driver associated with the *DriverObject* parameter. The device object is also linked into the I/O database in such a way that if the driver is unloaded, all of its device objects can be found.

Device objects for disks, tapes, CD ROMs, and RAM disks are given a Volume Parameter Block (VPB) that is initialized to indicate that the volume has never been mounted on the device.

A device driver may use the *DeviceExtension* parameter to cause storage to be allocated at the end of the device object. This storage, located by the *DeviceExtension* field in the device object, may be used by the device driver to keep device-specific context information.

This function must be invoked by each driver to create one or more device objects; otherwise the driver cannot be located by the I/O system.

10.21. IoCreateFile

The I/O system or a driver can create or open a file using the **IoCreateFile** function:

NTSTATUS
IoCreateFile(
    OUT PHANDLE *FileHandle,*
    IN ACCESS_MASK *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes,*
    OUT PIO_STATUS_BLOCK *IoStatusBlock,*
    IN PLARGE_INTEGER *AllocationSize* **OPTIONAL,**
    IN ULONG *FileAttributes,*
    IN ULONG *ShareAccess,*
    IN HANDLE *ErrorPort* **OPTIONAL,**
    IN ULONG *Disposition,*
    IN ULONG *Options,*
    IN PVOID *EaBuffer* **OPTIONAL,**
    IN ULONG *EaLength,*
    IN BOOLEAN *ForceAccessCheck,*
    IN BOOLEAN *PagingFileOpen*

);

Parameters:

FileHandle - A variable to receive the handle to the file.

DesiredAccess - Specifies the type of access that the caller requires to the file. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

ObjectAttributes - A pointer to a structure that specifies the name of the file, a root directory, a security descriptor, a quality of service descriptor, and a set of file object attributes flags. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

IoStatusBlock - A variable to receive the final completion status and information about the operation. The actual action taken by the system is written to the *Information* field of this variable.

AllocationSize - Optionally specifies the initial allocation size of the file in bytes. The size has no effect unless the file is created, overwritten, or superseded.

FileAttributes - Specifies the file attributes for the file. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

ShareAccess - Specifies the type of share access that the caller would like to the file. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

ErrorPort - Optionally specifies a handle to an open port to be RPC'd to if an error such as "the wrong volume is in the drive" occurs. If a handle is specified, the caller must have **PORT_READ** and **PORT_WRITE** access to the port.

CreateDisposition - Specifies the actions to be taken if the file does or does not already exist. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

CreateOptions - Specifies the options that should be used when creating or opening the file. See the **NtCreateFile** description in the *Windows NT I/O System Specification* for more information.

EaBuffer - Optionally specifies a list of EAs that should be set on the file if it is created. This is done as an atomic operation. That is, if an error occurs setting the EAs on the file, then the file will not be created.

EaLength - Supplies the length of the *EaBuffer*. If no buffer is supplied then this value should be zero.

ForceAccessCheck - Indicates whether access checking should be forced even though the caller's previous mode is kernel. If TRUE, then access checking will be performed.

PagingFileOpen - Indicates whether a paging file is being opened by the Modified Page Writer. Special handling is performed in some file systems when a paging file is being opened.

The **IoCreateFile** function is used by the I/O system to implement the **NtCreateFile** and **NtOpenFile** system services. It is also used by those kernel components that require special processing such as forcing access checks, or opening a paging file.

Each of the parameters to this function are syntactically and semantically the same as those specified in the **NtCreateFile** system service. The only differences between this function and the system service are the final two parameters.

10.22. IoCreateStreamFile

A file system can create a stream file object using the **IoCreateStreamFile** function:

```
PFILE_OBJECT
IoCreateStreamFile(
    IN PFILE_OBJECT FileObject OPTIONAL,
    IN PDEVICE_OBJECT DeviceObject OPTIONAL
    );
```

Parameters:

FileObject - A pointer to a file object that the stream file object is to be modeled after. This parameter is optional if the *DeviceObject* parameter is specified.

DeviceObject - A pointer to device object representing the physical device on which the stream file is being opened. This parameter is optional if the *FileObject* parameter is specified.

The **IoCreateStreamFile** function creates a stream file object that can be used to cache a stream of a file other than the data of the file. This function is used by file systems to represent those parts of the on-disk structure that are not included in proper files.

10.23. IoDeallocateAdapterChannel

A driver can explicitly deallocate an adapter channel using the **IoDeallocateAdapterChannel** function:

VOID
IoDeallocateAdapterChannel(
IN PADAPTER_CHANNEL *AdapterObject*
);

Parameters:

AdapterObject - A pointer to the adapter object representing the adapter channel to be deallocated.

The **IoDeallocateAdapterChannel** function frees an adapter channel that was previously allocated using the **IoAllocateAdapterChannel** function. If any map registers were allocated, then they are deallocated as well.

This function is used by a device driver to allocate an adapter channel and/or map registers because the device it services is referenced through a bus adapter or a DMA controller.

10.24. IoDeallocateController

A driver may explicitly deallocate a controller object using the **IoDeallocateController** function:

VOID
IoDeallocateController(
    IN PCONTROLLER_OBJECT *ControllerObject*
    );

Parameters:

ControllerObject - A pointer to the controller object that is to be deallocated.

The **IoDeallocateController** function frees a controller object that was previously allocated using the **IoAllocateController** function.

This function is used by a device driver to allocate a controller because the device it services is referenced through a device controller.

10.25. IoDeallocateIrp

An I/O Request Packet may be deallocated using the **IoDeallocateIrp** function:

VOID
IoDeallocateIrp(
    IN PIRP *Irp*
    );

Parameters:

Irp - A pointer to the I/O Request Packet to be deallocated.

The **IoDeallocateIrp** function deallocates the specified IRP.

This function is used by the I/O system to deallocate IRPs once all of the processing for the request has been completed. It is also possible, in some cases, for a file system

process to use this service to dispose of associated IRPs that the FSP created to implement a request.

10.26. IoDeleteController

A controller object can be deleted using the **IoDeleteController** function:

VOID
IoDeleteController(
    IN PCONTROLLER_OBJECT *Controller*
    );

Parameters:

Controller - A pointer to the controller object to delete.

The **IoDeleteController** function deletes a controller object. This function is invoked when a device driver is unloading. It is an error to attempt to delete a controller object if it is owned by a device object or if a device object is currently waiting to allocate the controller.

10.27. IoDeallocateMdl

A Memory Descriptor List (MDL) may be deallocated using the **IoDeallocateMdl** function:

VOID
IoDeallocateMdl(
    IN PMDL *Mdl*
    );

Parameters:

Mdl - A pointer to the MDL to be deallocated.

The **IoDeallocateMdl** function deallocates an MDL that was previously allocated through the **IoAllocateMdl** function. The function frees the storage for the MDL back to the MDL pool from which it was allocated.

This function is used by the I/O system completion code as well as by any drivers that perform their own local buffer management.

10.28. IoDeleteDevice

A driver can delete a device object using the **IoDeleteDevice** function:

VOID
IoDeleteDevice(
    IN PDEVICE_OBJECT *DeviceObject*
    );

Parameters:

DeviceObject - A pointer to the device object that is to be deleted.

The **IoDeleteDevice** function marks a device object for deletion as soon as its reference count is decremented to zero. No other references may be established to the object once it is marked for deletion; it is treated as if the object does not exist.

This function is invoked by a device driver to delete its device objects when the driver is being unloaded.

10.29. IoDeregisterFileSystem

A file system may deregister itself as an active file system using the **IoDeregisterFileSystem** function:

VOID
IoDeregisterFileSystem(
    IN OUT PDEVICE_OBJECT *DeviceObject*
    );

Parameters:

DeviceObject - A pointer to the device object for the file system.

The **IoDeregisterFileSystem** function causes the file system specified by the *DeviceObject* to be deregistered as an active file system. This is done by simply removing the specified device object from the list of active file systems.

This function is invoked by a registered file system (see **IoRegisterFileSystem**) when the driver for the file system is being unloaded.

10.30. IoDetachDevice

A driver may use the **IoDetachDevice** function to detach one device object from another device object:

VOID
IoDetachDevice(
    IN OUT PDEVICE_OBJECT *TargetDevice*
    );

Parameters:

TargetDevice - A pointer to the target device object that is to be detached from by the device that is servicing it.

The **IoDetachDevice** function detaches the device that is currently attached to the specified *TargetDevice*. This function disassociates two devices previous attached to each other with the **IoAttachDevice** function.

This function is invoked by intermediate drivers when they are unloading or when they have been told to stop servicing a device.

10.31. IoFlushAdapterBuffers

A driver can flush the buffers of an I/O adapter using the **IoFlushAdapterBuffers** function:

VOID
IoFlushAdapterBuffers(
    IN PADAPTER_OBJECT *AdapterObject*
    );

Parameters:

AdapterObject - A pointer to the adapter object representing the adapter whose buffers are to be flushed.

The **IoFlushAdapterBuffers** function is used to flush any remaining data in the I/O adapter's buffers. This function must be invoked at the end of each data transfer by all device drivers that deal with devices attached to the adapter.

10.32. IoGetAttachedDevice

The I/O system or a driver may obtain a pointer to the highest level device attached to a specific device object using the **IoGetAttachedDevice** function:

PDEVICE_OBJECT
IoGetAttachedDevice(
IN PDEVICE_OBJECT *DeviceObject*
);

Parameters:

DeviceObject - A pointer to the device for which the highest level attached device object is to be returned.

The **IoGetAttachedDevice** function returns the highest level device attached to a specified device object. That is, it follows all of the links of the devices that are attached to the specified device. This allows the I/O system or a driver to pass a request to the highest level driver associated with a device. If no devices are attached to the specified device object, then a pointer to the specified device object is returned.

This function is invoked by the I/O system and drivers to determine what driver an IRP should be passed to.

10.33. IoGetCurrentIrpStackLocation

A driver can obtain a pointer to the current stack location in an I/O Request Packet (IRP) using the **IoGetCurrentIrpStackLocation** function:

PIO_STACK_LOCATION
IoGetCurrentIrpStackLocation(
IN PIRP *Irp*
);

Parameters:

Irp - A pointer to the I/O Request Packet that contains the stack location whose address is to be returned.

The **IoGetCurrentIrpStackLocation** function returns a pointer to the current stack location in the specified IRP. This location contains the function codes, parameters, and I/O system information that describe the operation being requested to the driver.

This function is invoked by all device drivers to determine the operation to perform, as well as to determine the parameters, if any, that have been specified for the operation.

10.34. IoGetNextIrpStackLocation

A driver may obtain a pointer to the next stack location in an I/O Request Packet (IRP) using the **IoGetNextIrpStackLocation** function:

PIO_STACK_LOCATION
IoGetNextIrpStackLocation(
IN PIRP *Irp*
);

Parameters:

Irp - A pointer to the I/O Request Packet that contains the stack location whose address is to be returned.

The **IoGetNextIrpStackLocation** function returns a pointer to the next stack location in the specified IRP. This allows the current device driver to pass parameter and function code information to the next level driver using the same packet with which the current driver was invoked.

This function is invoked by all device drivers that pass IRPs to lower level drivers to pass function and parameter information. Note that even if the parameters are exactly the same, they must still be placed into the next driver's stack location since it will automatically look for them there.

10.35. IoGetRelatedDeviceObject

The device object referred to by a file object can be obtained using the **IoGetRelatedDeviceObject** function:

PDEVICE_OBJECT
IoGetRelatedDeviceObject(
    IN PFILE_OBJECT *FileObject*
    );

Parameters:

FileObject - A pointer to the file object whose related device object is returned.

The **IoGetRelatedDeviceObject** function returns a pointer to the device object that a file object refers to after all device object links have been chased.

This function is used by the I/O system and by any device driver that needs to determine the highest level device object to which a file object refers.

10.36. IoGetRequestorProcess

A driver may obtain a pointer to the process that originally made an I/O request using the **IoGetRequestorProcess** function:

PEPROCESS
IoGetRequestorProcess(
    IN PIRP *Irp*
    );

Parameters:

Irp - A pointer to the I/O Request Packet representing the request whose originator process is returned.

The **IoGetRequestorProcess** function allows a driver to obtain a pointer to the process data structure for the originator of a specified I/O request. This function is useful to file systems in keeping track of which processes own locks.

This function is used by file systems to keep track of lock owners and processes that have associated events with pipes.

10.37. IoInitializeDpcRequest

A device driver may initialize its device object's DPC using the **IoInitializeDpcRequest** function:

VOID
IoInitializeDpcRequest(
IN PDEVICE_OBJECT *DeviceObject,*
IN PIO_DPC_ROUTINE *DpcRoutine*
);

Parameters:

DeviceObject - A pointer to the device object that contains the DPC entry that is to be initialized.

DpcRoutine - The address of a routine that is to be invoked at DISPATCH_LEVEL when the Deferred Procedure Call entry is removed from the DPC queue by the kernel.

The routine specified by the *DpcRoutine* parameter has the following type definition:

typedef
VOID
(*PIO_DPC_ROUTINE) (
IN PKDPC *Dpc,*
IN PDEVICE_OBJECT *DeviceObject,*

IN PIRP *Irp,*
IN PVOID *Context*
);

Parameters:

Dpc - A pointer to the kernel DPC used to represent the call to this procedure. This parameter is ignored by device drivers.

DeviceObject - A pointer to the device object whose request needs servicing. This is the same device object as specified in the **IoInitializeDpcRequest** and **IoRequestDpc** I/O system calls.

Irp - A pointer to the I/O Request Packet that needs to be serviced. This is generally the reason that the DPC was requested in the first place.

Context - A pointer to whatever context is required by the device driver.

The **IoInitializeDpcRequest** function is used by the device driver's initialization routine to initialize the DPC in the device driver's device object so that the DPC can be used later to submit DPC requests. This allows the driver's interrupt service routine to request a DPC through the **IoRequestDpc** interface without having to initialize the DPC at device IRQL.

10.38. IoInitializeTimer

A device driver timer may be initialized using the **IoInitializeTimer** function:

VOID
IoInitializeTimer(
IN PDEVICE_OBJECT *DeviceObject,*
IN PIO_TIMER_ROUTINE *TimerRoutine*
);

Parameters:

DeviceObject - A pointer to the device object that contains the timer to be used.

TimerRoutine - Specifies the timer routine that is to be invoked once every second with a pointer to the counter associated with the device object.

The routine specified by the *TimerRoutine* parameter has the following type definition:

typedef
VOID
(*PIO_TIMER_ROUTINE) (
IN PDEVICE_OBJECT *DeviceObject,*
IN PLONG *TimerCounter*
);

Parameters:

DeviceObject - A pointer to the device object with which the timer counter is associated.

TimerCounter - A pointer to the timer counter associated with the device object.

The **IoInitializeTimer** function sets up a timer that expires once every second. Each time the timer expires, the system invokes the routine specified by the *TimerRoutine* parameter. The timer is actually started using the **IoStartTimer** function.

10.39. IoIsOperationSynchronous

A driver can determine whether an I/O operation is synchronous using the **IoIsOperationSynchronous** function:

BOOLEAN
IoIsOperationSynchronous(
IN PIRP *Irp*
);

Parameters:

Irp - Pointer to the I/O Request Packet for the operation to be checked.

The **IoIsOperationSynchronous** function checks whether the I/O request represented by the specified IRP is synchronous and returns a BOOLEAN value of TRUE if it is synchronous.

This function is used by drivers to determine whether an operation is synchronous and therefore whether or not the requestor's thread may be used by the driver to perform the I/O operation.

10.40. IoMakeAssociatedIrp

An associated I/O Request Packet can be allocated and initialized using the **IoMakeAssociatedIrp** function:

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

Parameters:

Irp - Pointer to the master I/O Request Packet with which the new packet should be associated.

StackSize - Specifies the number of stack locations needed in the IRP. This value should equal the number of layers in the chain of layered drivers servicing this request.

The **IoMakeAssociatedIrp** allocates and initializes an I/O Request Packet and associates it with a master packet. The count in the master packet should already have been set by the caller to the number of packets to be associated with it. The number of stack locations to be allocated for the associated IRP is specified by the *StackSize* parameter.

10.41. IoMapTransfer

A DMA I/O transfer may be mapped through an adapter or DMA controller using the **IoMapTransfer** function:

VOID
IoMapTransfer(
IN PADAPTER_OBJECT *AdapterObject,*
IN PMDL *Mdl,*
IN PVOID *MapRegisterBase,*
IN PVOID *CurrentVa,*
IN ULONG *Length,*
IN BOOLEAN *WriteToDevice*
);

Parameters:

AdapterObject - A pointer to the adapter object representing the adapter or DMA controller where the map registers reside.

Mdl - A pointer to a Memory Descriptor List (MDL) that maps the locked-down buffer to/from which the I/O is to take place.

MapRegisterBase - A pointer to the base of the map registers in the adapter or DMA controller. This value is passed to the driver's *ExecutionRoutine* when the adapter object and map registers have been allocated by the **IoAllocateAdapterChannel** function.

CurrentVa - A pointer to the current virtual address in the buffer described by the *Mdl* where the I/O operation is to take place.

Length - Supplies the length of the transfer to map.

WriteToDevice - Supplies a BOOLEAN value that indicates that the direction of the data transfer is to the device.

The **IoMapTransfer** function loads the appropriate map registers in the adapter or DMA controller to cause the I/O operation to map to the appropriate memory locations described by the *Mdl, CurrentVa,* and *Length* parameters.

This function is used by device drivers to map DMA I/O to the appropriate memory buffer.

10.42. IoPageRead

The pager can read pages of data from the paging file or from a mapped file using the **IoPageRead** function:

NTSTATUS
IoPageRead(
IN PFILE_OBJECT *FileObject,*
IN PMDL *MemoryDescriptorList,*
IN PLARGE_INTEGER *StartingOffset,*
IN PKEVENT *Event,*
OUT PIO_STATUS_BLOCK *IoStatusBlock*
);

Parameters:

FileObject - A pointer to a referenced file object representing the file to be read.

MemoryDescriptorList - A Memory Descriptor List (MDL) that describes the locked-down buffer into which data from the file is to be read.

StartingOffset - The starting byte offset within the specified file where the read operation is to begin.

Event - A pointer to a kernel event that should be set to the Signaled state once the I/O operation is complete. This is the only valid synchronization technique for this type of request.

IoStatusBlock - A variable to receive the final completion status and information for the read operation. The number of bytes actually read is returned in the *Information* field.

This variable must be locked into memory so that it cannot move until the I/O is complete.

The **IoPageRead** function gives the **Windows NT** Pager a quick way of building and starting an I/O request to read data from a file. This allows I/O completion to be short circuited so that no APCs or pagefaults occur while attempting to complete a page read operation.

The function reads the number of bytes specified by the MDL into the buffer described by the MDL, beginning at the *StartingBlock* within the file. The *Event* is set to the Signaled state once the operation has completed.

This function is only invoked by the **Windows NT** Pager.

10.43. IoQueryInformation

Information about a file object may be obtained using the **IoQueryInformation** function:

NTSTATUS
IoQueryInformation(
    IN PFILE_OBJECT *FileObject,*
    IN FILE_INFORMATION_CLASS *FileInformationClass,*
    IN ULONG *Length,*
    OUT PVOID *FileInformation,*
    OUT PULONG *ReturnedLength*
    );

Parameters:

FileObject - A pointer to the file object for which information is to be returned.

FileInformationClass - The file information class of the type of information that is to be returned.

Length - The length of the *FileInformation* buffer, in bytes.

FileInformation - A buffer to receive the returned information about the file.

ReturnedLength - A variable to receive the length of the information returned in the *FileInformation* buffer.

The **IoQueryInformation** function returns information about a file according to the type of information requested. The types of information that can be requested are defined by the **FILE_INFORMATION_CLASS** data type.

This function performs the same basic function as the **NtQueryInformationFile** system service, but uses a pointer to a file object rather than a handle interface. It must be invoked from kernel mode. It is also used by the **NtQueryObject** object system service to obtain the size of an ACL for a file.

10.44. IoRegisterFileSystem

A file system driver may register itself as an active file system using the **IoRegisterFileSystem** function:

VOID
IoRegisterFileSystem(
IN OUT PDEVICE_OBJECT *DeviceObject*
);

Parameters:

DeviceObject - A pointer to the device object that represents the file system.

The **IoRegisterFileSystem** function registers a driver as an active file system. This is accomplished by placing the file system's *DeviceObject* into a LIFO-ordered list of file systems to be searched when a file system is needed to service a device.

This function is invoked by each file system driver. The file systems are placed in various queues depending on the type of file system. For example, disk file systems are placed in a queue that is searched whenever a disk media is to be automatically mounted. Each disk file system driver is queried in turn to determine whether or not the driver recognizes the on-disk structure.

10.45. IoRemoveShareAccess

A file's share access information may be removed for a given open instance using the **IoRemoveShareAccess** function:

VOID
IoRemoveShareAccess(
    IN PFILE_OBJECT *FileObject,*
    IN OUT PSHARE_ACCESS *ShareAccess*
    );

Parameters:

FileObject - A pointer to the file object for the current open request.

ShareAccess - A pointer to the common share access data structure associated with the file being closed.

The **IoRemoveShareAccess** function removes the share access information for the file being closed as described by the *FileObject* parameter. This updates how the file is currently being accessed.

When a file is being closed, the file system uses the **IoRemoveShareAccess** function to update how the file is currently opened. This function updates the **SHARE_ACCESS** structure according to how the file was opened by the specified file object. It is unnecessary to call this function for the last close request on the file. That is, if the file is only accessed through the single, specified file object, the file system need not invoke this function to update the share access information for that file.

10.46. IoRequestDpc

A DPC routine can be queued by using the **IoRequestDpc** function:

VOID
IoRequestDpc(
    IN PDEVICE_OBJECT *DeviceObject,*
    IN PIRP *Irp,*
    IN PVOID *Context*

);

Parameters:

DeviceObject - A pointer to the device object that represents the device for which I/O was requested.

Irp - A pointer to an I/O Request Packet that the DPC routine is to service. This pointer is passed to the driver's DPC routine as one of its arguments.

Context - Supplies a pointer to whatever context the interrupt service routine would like to pass to the DPC routine. This pointer is passed to the driver's DPC routine as one of its arguments.

The **IoRequestDpc** function requests that the DPC routine associated with the *DeviceObject* be invoked at DISPATCH_LEVEL and passed a pointer to the *DeviceObject*, a pointer to the *Irp*, and the *Context* parameter. The device object's DPC entry must have been initialized using the **IoInitializeDpcRequest** I/O function.

10.47. IoSendMessage

A terminal driver may send an "unsolicited input" message to a message port using the **IoSendMessage** function:

VOID
IoSendMessage(
IN PSTRING *DestinationPort,*
IN PSTRING *TerminalName*
);

Parameters:

DestinationPort - The name of the port to which the message is sent.

TerminalName - The name of the terminal on which the unsolicited input occurred.

The **IoSendMessage** function allocates a datagram and sends it to the *DestinationPort*. The datagram contains a message that indicates that unsolicited input occurred on the terminal specified by the *TerminalName* parameter.

This function is invoked by terminal drivers when unsolicited input is encountered to give the system notification that perhaps a user is attempting to log on.

10.48. IoSetCompletionRoutine

A driver may set a completion routine address and a context parameter in an I/O Request Packet (IRP) stack location using the **IoSetCompletionRoutine** function:

VOID
IoSetCompletionRoutine(
IN PIRP *Irp,*
IN PIO_COMPLETION_ROUTINE *CompletionRoutine,*
IN PVOID *Context,*
IN BOOLEAN *InvokeOnSuccess,*
IN BOOLEAN *InvokeOnError,*
IN BOOLEAN *InvokeOnCancel*
);

Parameters:

Irp - A pointer to the IRP that contains the stack location in which the completion routine is set.

CompletionRoutine - The address of a completion routine to be executed upon completion of the I/O request.

Context - A context parameter that is passed to the completion routine. The driver can use this parameter for any context that it needs.

InvokeOnSuccess - Indicates that the completion routine is to be invoked if the I/O operation completes successfully.

InvokeOnError - Indicates that the completion routine is to be invoked if the I/O operation completed with an error.

InvokeOnCancel - Indicates that the completion routine is to be invoked if the I/O operation is being canceled.

The routine specified by the *CompletionRoutine* has the following type definition:

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

Parameters:

DeviceObject - Supplies a pointer to the device driver's device object.

Irp - Supplies a pointer to the I/O Request Packet.

Context - Supplies the value of the *Context* parameter specified in the call to the **IoSetCompletionRoutine** function.

The **IoSetCompletionRoutine** stores the address of the completion routine into the current IRP so that when the operation completes the driver can be invoked. If the driver does not wish to be invoked for any of the above reasons, then it need not perform any function calls.

This function is invoked by any layered driver that wishes to be notified when an I/O operation that it has passed to a lower level driver completes.

10.49. IoSetShareAccess

A file's share access information may be set using the **IoSetShareAccess** function:

VOID
IoSetShareAccess(
IN ACCESS_MASK *DesiredAccess,*

IN ULONG *DesiredShareAccess,*
IN OUT PFILE_OBJECT *FileObject,*
OUT PSHARE_ACCESS *ShareAccess*
);

Parameters:

DesiredAccess - Supplies the types of access that the current open request would like to the file. This is the same desired access parameter given to the file system by the I/O system when the open request is made.

DesiredShareAccess - Supplies the types of shared access that the current open request would like to the file. This is the same shared access parameter given to the file system by the I/O system when the open request is made.

FileObject - A pointer to the file object for the current open request.

ShareAccess - A pointer to the common share access data structure associated with the file being opened.

The **IoSetShareAccess** function sets the initial share access state for a file when it is opened or created for the first time. After the initial file open/create, other requests may be checked against the share access for the file using the **IoCheckSharedAccess** function.

File systems maintain state about files through structures called File Control Blocks **(FCBs)**. The **SHARE_ACCESS** is a structure that describes how the file is currently accessed by all opens. It is contained in the FCB as part of the open file state for the file. The **SHARE_ACCESS** data structure itself should be treated as an opaque data type by file systems and drivers. That is, its contents should only be accessed through the I/O system functions. This allows the structure to change from release to release without having to modify driver source code.

It should be noted that this function provides no synchronization with other updates to the **SHARE_ACCESS** structure. The file system should lock access to the structure by locking its FCB.

This function is used by file systems to set the initial share access for a file.

10.50. IoStartNextPacket

The next packet queued to a driver can be started using the **IoStartNextPacket** function:

VOID
IoStartNextPacket(
    IN PDEVICE_OBJECT *DeviceObject*
    );

Parameters:

DeviceObject - A pointer to the device object that contains the device queue for the device on which the I/O request is performed.

The **IoStartNextPacket** function checks the device queue in the specified *DeviceObject* for an IRP and, if one is found, it is dequeued and passed to the driver's start I/O routine with a pointer to the IRP and a pointer to the *DeviceObject*.

This function is generally invoked by a driver after the processing of the current IRP has been completed by the driver. This allows the driver to start the next operation that is pending for the device.

10.51. IoStartPacket

An I/O request can be started on a device using the **IoStartPacket** function:

VOID
IoStartPacket(
    IN PDEVICE_OBJECT *DeviceObject,*
    IN PIRP *Irp,*
    IN PULONG *Key* **OPTIONAL**
    );

Parameters:

DeviceObject - A pointer to the device object for the device on which the request is to be performed.

Irp - A pointer to the I/O Request Packet to be started on the specified device, provided that the device is not already busy.

Key - An optional key value that specifies where in the pending IRP list the *Irp* should be queued if the device is already busy.

The **IoStartPacket** function checks the device queue in the specified *DeviceObject* and either starts the request by passing it to the driver's start I/O routine or queues it to the device's work queue for later processing. If the *Irp* is queued to the device's work queue, then a *Key* may optionally be specified that indicates where in the pending list the request is queued.

For more information on how a packet is actually queued to a device queue, see the **Windows NT Kernel Specification**. In particular, refer to the section on Device Queue Objects.

This function is used by a driver's major function routine to start an operation on a device or to have it queued if the device is already busy.

10.52. IoStartTimer

An initialized one second timer can be started by using the **IoStartTimer** function:

VOID
IoStartTimer(
IN PDEVICE_OBJECT *DeviceObject*
);

Parameters:

DeviceObject - Specifies the device object whose timer is to be started. The timer must have been initialized using the **IoInitializeTimer** function.

The **IoStartTimer** function starts the timer that was previously initialized by **IoInitializeTimer**. Once the timer has been started, the timer routine is invoked once every second.

This function is used by drivers to time out operations. The timer counter should be initialized either to a negative count if no timed operation is being performed, or to the number of seconds that the operation has to complete if a timed operation is being performed. If the timer routine decrements the counter and it becomes zero, then the operation did not complete in time. If the counter is a negative number, then the routine should not modify it.

10.53. IoStopTimer

A one-second timer can be stopped using the **IoStopTimer** function:

VOID
IoStopTimer(
    IN PDEVICE_OBJECT *DeviceObject*
    );

Parameters:

DeviceObject - Specifies the device object whose timer is to be stopped.

The **IoStopTimer** function stops the timer that was previously started by the **IoStartTimer** function. While the timer is then canceled and will not expire again, it is possible for the timer routine to be invoked one more time after the call to this routine has been completed. This is because the timer could have expired and been placed into the queue during a window that cannot be canceled. Furthermore, the timer could have expired on another processor at the same time that the call to this routine was being made.

This function is used by drivers to cancel one second timers. This function is generally only invoked when the driver is being unloaded from the system.

10.54. IoSynchronousPageWrite

The pager can synchronously write pages of data to the paging file or to a mapped file using the **IoSynchronousPageWrite** function:

NTSTATUS
IoSynchronousPageWrite(
IN PFILE_OBJECT *FileObject,*
IN PMDL *MemoryDescriptorList,*
IN PLARGE_INTEGER *StartingOffset,*
IN PKEVENT *Event,*
OUT PIO_STATUS_BLOCK *IoStatusBlock*
);

Parameters:

FileObject - A pointer to a referenced file object representing the file to write.

MemoryDescriptorList - A Memory Descriptor List (MDL) that describes the locked-down buffer containing the data to write to the file.

StartingOffset - The starting byte offset within the specified file where the write operation is to begin.

Event - Supplies a pointer to a kernel event that to set to the signaled state once the write is complete.

IoStatusBlock - A variable to receive the final completion status and information about the write operation. The number of bytes actually written is returned in the *Information* field.

The **IoSynchronousPageWrite** function gives the memory manager a quick way of building and starting an I/O request to write data to a file. This allows I/O completion to be short circuited for paging I/O.

The function writes the number of bytes specified by the MDL from the buffer described by the MDL, beginning at the *StartingOffset* within the file. The *Event* is set to the signaled state once the operation has completed.

This function is only invoked by the memory manager.

10.55. IoUpdateShareAccess

A file system can update the share access for a file using the **IoUpdateShareAccess** function:

VOID
IoUpdateShareAccess(
    IN OUT PFILE_OBJECT *FileObject,*
    IN OUT PSHARE_ACCCESS *ShareAccess*
    );

Parameters:

FileObject - A pointer to the file object for the current open request.

ShareAccess - A pointer to the common share access data structure associated with the file being opened. This structure is treated as an opaque type by drivers.

The **IoUpdateShareAccess** function updates the *ShareAccess* according to the types of access being requested for the current open request. This function may only be invoked if a previous call to **IoCheckShareAccess** succeeded.

This function simply updates the **SHARE_ACCESS** structure maintained for files. It performs the same update functionality as the **IoCheckShareAccess** function but does not perform the check access functionality.

10.56. IoWriteErrorLogEntry

An error log entry buffer may be written to the error log queue using the **IoWriteErrorLogEntry** function:

VOID
IoWriteErrorLogEntry(
    IN OUT PVOID *ErrorLogEntry*

);

Parameters:

ErrorLogEntry - A pointer to the error log entry buffer that contains the entry data. This entry must have been allocated using the **IoAllocateErrorLogEntry** function.

The **IoWriteErrorLogEntry** queues the specified error log entry buffer to the error log thread's database so that the thread can send it to the error log process. The error log process is actually responsible for writing the entry out to the error log file.

This function is used by drivers to post an error log entry.

11. I/O System Folklore

The following sections describe those features of the I/O system that are not fully described in any other documentation. These sections further describe how the I/O system works and how driver writers can use this knowledge and I/O system features to develop robust, high-performance drivers.

11.1. Rules for Completing an I/O Request

A driver needs to complete an I/O request in one of two different situations:

o - The request packet was in error and will not be processed at all. For example, a parameter was invalid for the specified function.

o - The request packet parameters are correct, so the I/O request will be processed.

In the first case, the packet is given to the driver at the appropriate dispatch entry point for the function code in the IRP stack location. For a file system driver, this means that the packet is given to the FSD. Since the packet is in error, there is no reason to return a pending status and then asynchronously complete the request at a later time. The driver dispatch routine can immediately determine that this is the case, so the packet should be aborted as follows:

o - Set the error status in the IRP by writing to the *IoStatus.Status* field of the packet.

o - Raise IRQL to DISPATCH_LEVEL, saving the old IRQL.

o - Invoke the **IoAbortInvalidRequest** function.

o - Lower IRQL to the previous IRQL returned from the raise operation.

o - Return to the caller of the dispatch routine with the same status written to the status field of the IRP.

This sequence causes the I/O system to return the error to the original caller of the system service. The I/O system will not set the caller's file handle or optional event to the Signaled state, and it will not write the caller's I/O status block. This is the definition of an I/O request that was in error. Notice that the driver must still write the status block in the IRP even though it will not be written to the caller's I/O status block. This must be done in case there is a layered driver above the current driver.

In the second case, all of the parameters for the specified function are correct. This means that the I/O request will be processed.

Once the driver has determined that the I/O request is to be completed, it has one of two options:

o - Process the request and complete it immediately.

o - Queue the request to be performed at a later time and return a pending status.

If the request can be immediately processed without causing the current thread to wait, then the driver should do so. An example of such a situation is when the caller has requested information about a file and the information is in memory. The driver can simply place the information into the buffer and complete the request. Notice that an optimization for the user can be made here if the driver returns a status of *STATUS_SUCCESS*. This means that the request was not only successful, but it is actually

complete at this point. That is, the file object or event has been set to the Signaled state, the I/O status block has been written, etc.

If the packet is to be queued and completed at a later time, then the driver should queue the packet and return *STATUS_PENDING*.

Once the request completes, then the driver should invoke the normal completion function. In either of the preceding cases, the sequence that the driver uses to complete the request is as follows:

o - Set the appropriate status in the IRP *IoStatus.Status* field.

o - Raise IRQL to DISPATCH_LEVEL, saving the old IRQL.

o - Invoke the **IoCompleteRequest** function.

o - Lower IRQL to the previous IRQL returned from the raise operation.

o - Return to the caller of the dispatch routine with the same status written to the status field of the IRP.

Notice that if the request is queued and processed later it may still incur an error. The **IoCompleteRequest** sequence should still be used by the driver to complete the I/O request.

11.2. Accessing Another Driver

The **Windows NT** I/O system allows drivers to be layered so that one driver may communicate directly with another driver. The upper-level driver may do this either by reusing the same IRP or by passing a new, separate IRP.

In either case, a driver may invoke another driver by using the **IoCallDriver** function. This function takes two parameters:

o - A pointer to the device object for the device upon which the request is to be performed

o - A pointer to the I/O Request Packet itself

To obtain a pointer to the device object the upper level driver must first open the device. This can be done by simply invoking the **NtOpenFile** system service. This service returns a handle to a file object that represents a connection to the device. The file object can then be referenced by invoking the **ObReferenceObjectByHandle** function. This function writes the address of the file object as one of its output parameters. The file object itself contains a pointer to the device object for the device that was opened. This pointer can now be used to reference the device in **IoCallDriver** function calls.

If the driver is either unloading or it is told to close the device through a configuration control function packet, it should perform the following steps:

o - Dereference the file object pointer by invoking the **ObDereferenceObject** function.

o - Close the handle to the file object by invoking the **NtClose** system service.

These steps will cause the appropriate reference counts to be decremented so that the device can be removed from the system if necessary.

11.3. Generating Packets

Most of the time, a driver that would like to communicate with another driver can simply reuse the I/O Request Packet (IRP) that it is given by simply using the next stack location in the IRP. It can do this by invoking the **IoGetNextIrpStackLocation** function to get a pointer to the next stack location in the IRP. The driver can then fill in the function and parameter fields and pass the packet to the driver by using the **IoCallDriver** function.

However, there are times when this is insufficient and the driver must allocate a different IRP to pass to the next driver. This happens when the driver implements a request by splitting it into several different parallel requests. There are several different routines that can be used to aid in this situation:

o - **IoAllocateIrp** - This routine simply allocates and initializes an IRP. It is then up to the calling driver to fill in the appropriate header locations as well as the stack location to tell the target driver the function that is to be performed.

Obtaining a pointer to the appropriate stack location can be done using the **IoGetNextIrpStackLocation**.

Allocation of the packet is done using the appropriate IRP lookaside list if there are packets available in the system lists. The *StackSize* parameter required by the **IoAllocateIrp** function can be obtained from the *StackSize* field of the target device object.

o - **IoBuildSynchronousFsdRequest** - This function can be used to build a packet that is adequate to request that a target driver perform either a read or a write operation. However, the packet that this function builds is synchronized by an event specified as one of its parameters, so the current thread will have to wait for the event to be set to the Signaled state in order to synchronize the completion of the request. Therefore, it is recommended that these types of packets only be used in cases where the requesting thread is performing a synchronous I/O function.

o - **IoBuildAsynchronousFsdRequest** - This function can be used to build a packet that is adequate to request that a driver perform either a read or a write operation. It is up to the driver that builds the packet to synchronize the completion of the packet by specifying a completion routine for itself. This is done using the **IoSetCompletionRoutine** function. That is, once the packet is built, the driver sets the address of its completion routine before giving it to the next driver. In this way, the driver is notified when the request packet has completed.

This function is generally used by the FSD part of drivers because the specified completion routine is executed in a thread-independent context.

o - **IoBuildFspRequest** - This function can be used to build a packet that is adequate to request that a driver perform either a read or a write operation. It is generally used by the FSP part of a driver because synchronization of the packet is performed through either a kernel event or an APC routine. The FSP must supply one of these two parameters as its synchronization mechanism or it will not be able to determine when the packet is complete.

\\ *There will certainly be other routines that can be used to generate packets that will be added before the system ships. These routines will be added as needed. It is not recommended that drivers actually use the* **IoAllocateIrp** *interface and then generate everything by hand, however there is no better way to do this today unless one of the other routines provides the exact functionality needed by the driver.*\\

11.4. Direct vs. Buffered vs. Neither I/O

The **Windows NT** I/O system provides drivers with a choice of three different methods for implementing I/O operations. These are as follows:

1. Direct I/O - Direct I/O refers to the capability to perform I/O directly into the caller's buffer. That is, the I/O system will set up the necessary data structures to allow the I/O operation to be performed directly into the caller's buffer. The driver writer specifies that this type of I/O is desired by setting the *DO_DIRECT_IO* flag in the *Flags* field of the device object.

   If this flag is set, the I/O system performs the following operations before passing the IRP to the driver:

   - o - The caller's buffer is probed for the appropriate access according to whether the request being performed is a read or a write operation.

   - o - The caller's buffer will be locked into memory so that the physical memory backing the buffer cannot be reused for some other operation.

   - o - An MDL will be built that describes the user's buffer.

   - o - The *MdlAddress* field of the IRP will be set to point to the MDL that was built.

   A driver might do direct I/O for two different reasons. It will use this type of I/O if a device that it is servicing performs DMA I/O. The MDL can be used in a call to the **IoMapTransfer** function to map the caller's buffer so that when the DMA controller reads from or writes to memory the appropriate locations will be read or written.

A driver might also use direct I/O if it needs to gain direct access to the caller's buffer but will not be executing in the context of the caller. For example, an FSP thread might need to copy data directly into the caller's buffer but, by definition, does not have direct access to it. The **MmMapLockedPages** function must be used to map the caller's buffer into the FSP thread's virtual address space by passing it a pointer to the MDL. This temporarily allows direct access to those physical pages backing the caller's buffer. Once the copy operation has completed, the **MmUnmapLockedPages** function can be used to unmap the caller's buffer.

2. Buffered I/O - Buffered I/O refers to the capability to perform I/O operations to an intermediate buffer that contains a copy of the data from the caller (write operation) or a copy of the data that is to be copied back to the caller's buffer (read operation) when the request is complete.

This type of I/O is generally used when a device that is being serviced cannot perform DMA I/O directly into a buffer. It is also used when keeping the caller's buffer locked for an extended period could cause system resources to be depleted.

A driver may specify that it performs buffered I/O by setting the *DO_BUFFERED_IO* flag in the *Flags* field of its device object. When this flag is set, the I/O system performs the following operations to set up the caller's buffer before passing the IRP to the driver:

o - The caller's buffer is probed for the appropriate access according to whether the request being performed is a read or a write operation.

o - A sufficiently large buffer is allocated from system non-paged pool to handle all of the data being read or written.

o - If the operation is a write, the data in the caller's buffer is copied into the allocated system buffer. If the operation is a read, the *IRP_INPUT_OPERATION* flag is set in the IRP flags field so that the contents of the system buffer will be copied into the caller's buffer after the operation has completed. The *IRP_DEALLOCATE_BUFFER* flag is also set in the IRP flags

field so that the buffer will be deallocated after the copy operation is complete.

o - The *AssociatedIrp.SystemBuffer* field of the IRP is set to point to the allocated system buffer.

Once the operation completes, the system buffer is automatically deallocated by the I/O system.

3. Neither I/O - Under certain circumstances, a driver might postpone specifying either direct I/O or buffered I/O until it has a chance to determine which type of operation is appropriate, based on whether data is immediately available or whether the data must be obtained from elsewhere asynchronously. For this case, the driver sets neither of the flags in the device object *Flags* field. It is then up to the driver to perform the necessary steps to allow either direct or buffered I/O to be performed. Notice that for some cases, the driver may not have to perform either type of I/O operation and can simply copy the data directly into the user's buffer.

If this type of I/O operation is specified by the device object, the I/O system performs the following steps before passing the IRP to the driver:

o - The caller's buffer is probed for the appropriate access according to whether the request being performed is a read or a write operation.

o - A pointer to the caller's buffer is passed in the *UserBuffer* field of the IRP.

If the driver determines that the data is immediately available and wants to copy it directly into the caller's buffer, then it does so by using an exception handler around the code that performs the copy. This is done to catch access violations that occur when another thread executing in the same process changes the virtual address space described by the caller's buffer. This may also occur because of a kernel APC being executed in the context of the current thread.

If the data is not immediately available, then the driver may wish to perform either direct or buffered I/O. It must perform the same steps that the I/O system

does to setup the appropriate structures so that the I/O can be completed normally. Any deviation from the exact setup can cause the system to crash.

It is also possible for a device driver to specify a preallocated driver buffer that contains data to be copied into the caller's buffer. This can be accomplished by performing the following steps:

o - Specify Neither I/O in the device object *Flags* field, setting neither of the other two device object flags.

o - Set the *AssociatedIrp.SystemBuffer* field to point to the preallocated driver buffer.

o - Set the *IRP_INPUT_OPERATION* flag in the IRP flags field, but not set the *IRP_DEALLOCATE_BUFFER* flag in the IRP flags field.

These three types of I/O are used for all **NtReadFile** and **NtWriteFile** operations. Most other NT API services use buffered I/O almost exclusively, except for the **NtDeviceIoControlFile** and **NtFsControlFile** system services. These services pass their buffers according to the *method bits* in the I/O control code. More detailed information is contained in the *Windows NT IRP Language Definition* specification.

11.5. Building Virtually Discontiguous Buffers

There are currently no **Windows NT** APIs in the I/O system that allow callers to provide more than one input or output buffer. This keeps the design of the I/O system as simple as possible and causes I/O completion to execute more quickly. Since no complex I/O user buffer state is required, there is at most only one copy operation that takes place during I/O completion.

However, layered drivers, such as network drivers, may need to provide each other with more than one virtually discontiguous buffer. A transport driver may wish to add a transport header to the front of a user data buffer. A datalink driver may wish to put another header in front of the transport's header, and so on. Rather than having each driver allocate a buffer, place its data into the buffer and then copy all of the previous data after its own data, it is much more efficient to simply insert a virtual buffer descriptor in front of the current data descriptor.

The data structures that represent these virtual buffer descriptors in **Windows NT** are *Memory Descriptor Lists* **(MDLs)**. Each MDL describes the physical pages that make up a single virtually contiguous buffer. By chaining MDLs through the structure's *Next* pointer, virtually discontiguous buffers may be specified in different driver layers.

During I/O completion, if the drivers do not run down the MDLs, then the I/O system will provide this functionality automatically. That is, all MDLs chained together from the IRP's *MdlAddress* field will automatically be deallocated and the pages described by those MDLs will be unlocked. If the driver that specified the MDL needs to perform its own buffer management, then it should deal with this by unlinking its MDL from the chain in its I/O completion routine.

11.6. I/O Services Synchronization

The **Windows NT** I/O system provides many services to users. Among these services are those that may be completed synchronously as well as those that may be completed asynchronously. This section explains how the I/O system services actually work to provide these capabilities to users, even though the I/O system's design is based on an asynchronous model. This section provides background information to enable driver writers to better understand the environment in which their driver is executing.

The I/O system services that complete asynchronously may either be invoked as asynchronous system services, or they may be invoked to execute synchronously. (The latter is the case when the file that the services are operating on was opened with one of the *FILE_SYNCHRONOUS_IO* options.) For those services that are asynchronous and are invoked to complete as such, no special processing is required. Once the packet is given to the driver, the I/O system simply returns to the caller.

However, a user may open a file using one of the synchronous I/O options. For this case, all services, synchronous and asynchronous, perform the steps outlined below to synchronize access to the file. Note that these options also cause all I/O operations on the file to be serialized.

o - The parameters for the service are probed, captured, and validated.

o - The file object is referenced by calling the object manager with the caller's file handle.

o - The semaphore associated with the file object is then waited on in either an alertable or non-alertable manner, depending on which synchronous I/O option was used in the open or create call.

o - The service calls the driver normally and then waits for the file object itself to be set to the Signaled state. This is a special case where the I/O completion routines will set the file object to the Signaled state along with an event, if one was specified. The other special operation performed is to copy the I/O status into the file object itself so that the service can return it to the caller without having to touch the caller's I/O status block. This makes special error recovery code unnecessary when the address space for the I/O status block has been deleted while the driver was servicing the request.

o - The semaphore is released and the service returns to the caller.

Some services that are synchronous must also deal with the asynchronous I/O system, even when the user did not open the file specifying one of the synchronous I/O options. These services perform the following steps to make it appear as if the I/O system is synchronous:

o - The parameters for the service are probed, captured, and validated.

o - The file object is referenced by calling the object manager with the caller's handle.

o - A local kernel event variable is initialized to the Not-Signaled state and used as the user-specified event in the IRP. A local I/O status block variable is used as the user-specified I/O status block and its address is placed in the IRP.

o - The *IRP_SYNCHRONOUS_API* flag is set in the flags field of the IRP. This flag informs the I/O completion code that the event is a kernel event, rather than a normal user object system event, and should not, therefore, be dereferenced during completion.

o - The service calls the driver.

o - If the return status from the driver is *STATUS_PENDING*, then the service waits for the local kernel event to be set to the Signaled state.

o - The local I/O status block contents are copied to the caller's I/O status block, and the status field is returned as the final status from the service.

12. Revision History

Original Draft 1.0, March 21, 1989

Changes from I/O specification revision 1.2, from which this draft was spawned:

- Redo IRP stacks; single routine; parameters.
- Add new I/O APIs.
- Fill in sections on APIs and data structures.

Revision 1.1, February 12, 1990

- Brought spec up to current design level.
- Fixed lots of typos and grammatical errors.
- Removed **IoAllocateAdapterAndChannel**.
- Removed **IoIsRequestComplete**.
- Added **IoAbortInvalidRequest**.
- Added **IoAllocateAdapterChannel**.
- Added **IoAllocateController**.
- Added **IoDeallocateAdapterChannel**.
- Added **IoDeallocateController**.
- Added **IoDeallocateMdl**.
- Added **IoGetRelatedDeviceObject**.
- Added **IoGetRequestorProcess**.
- Added **IoMapTransfer**.
- Added **IoGetAttachedDevice**.
- Fixed IN, OUT, and IN OUT in IoXxx calls.
- Added I/O system folklore section.

Revision 1.2, July 20, 1990

- Add device object pointer to timer routine.
- Updated share access manipulation routines to reflect latest design.
- Added **IoDeallocateIrpAtDispatchLevel**.
- Added **IoFlushAdapterBuffers**.
- Added **IoIsOperationSynchronous**.
- Added **IoCreateStreamFile**.
- Replaced **IoPageWrite** with new routines.

- Added description of stream file objects.
- Updated cancel description to reflect latest design.
- Updated timer descriptions to match latest design.
- Updated description of I/O completion w/o using PFN mutex.

Revision 1.3, xxxx ??, 1990

- Update description of unlocking pages during completion
- Document **IoIsOperationSynchronous**.
- Remove **IoDeallocateIrpAtDispatchLevel**.
- Modify **IoAsynchronousPageWrite** to use 64-bit offset.
- Modify **IoBuildAsynchronousFsdRequest** to use 64-bit offset.
- Modify **IoBuildFspRequest** to use 64-bit offset.
- Modify **IoBuildSynchronousFsdRequest** to use 64-bit offset.
- Modify **IoPageRead** to use 64-bit offset.
- Modify **IoSynchronousPageWrite** to use 64-bit offset.
- Removed **IoQueryAcl** function.
- Removed **IoSetAcl** function.
- Remove access parameters from **IoUpdateShareAccess**.
- Changed references from SUCCESS to NT_SUCCESS.