Using Fast I/O

In this article we will provide the rationale for fast I/O, a brief description of the various fast I/O calls, and conclude with some suggestions on how to use this interface to improve performance for *your* project.

The standard dogma for Windows NT kernel mode development is that NT uses the *I/O Request Packet* (IRP) as the basis of communications with drivers – the advantage of the I/O Request packet is that it encapsulates the context needed for a particular operation and allows abstraction of the driver away from many of the details common to drivers.

While this approach is very general and extensible – allowing for the cleanly layered Windows NT device architecture, there is a fair amount of overhead involved for operations where the request can be satisfied quickly. In that instance, the overhead associated with creating an IRP can dominate the cost of the entire operation – slowing the system down in performance critical areas. Because of this, the NT team introduced the concept of *fast I/O.* This approach to I/O is used by some *file system drivers*, such as NTFS, HPFS, FAT, and CDFS as well as the AFD *transport driver* which is used by WinSock.

Any driver can register a set of fast I/O entry points but their use is often extremely limited – requiring that just the right conditions be met before a particular fast I/O entry point will even be called. For example, both the *read* and *write* fast I/O entry points are only called when information about the particular file is being maintained by the *cache manager* in Windows NT. We will describe these restrictions later in this article.

Of course the most frustrating aspect of fast I/O for Windows NT is the total lack of available documentation – even the file system development kit does not provide a description of how fast I/O works or how to use it.

Rationale

The rationale for providing Fast I/O seems to be one of convenience – many I/O operations are repeatedly performed against the same data. For example, like most modern operating systems, Windows NT integrates file system caching with the virtual memory system – using the system memory as a giant cache for file system information. Such systems are extremely efficient and boost both the actual and perceived performance of the system.

A second reason for this integration is Windows NT's support for memory-mapped files. Supporting both read/write access *and* memory mapped file access to the same data, requires either a high cost (and hence low performance) cache consistency scheme or the scheme used by NT – where all data is stored in the virtual memory system. This ensures that data is always consistent – even between two programs accessing the same data using different access techniques.

This tight integration means that both read and write operations can often be satisfied from this cached data. In the search for performance, then, this fact can be exploited for read and write by simply calling a specialized routine which moves data from the VM cache into the user's memory, or vice versa. This eliminates the need to allocate an I/O request packet since the operation can be satisfied synchronously and need not call into lower drivers. It is *this* fundamental model which is realized by the fast I/O operations.

Once these fast I/O entry points have been created to provide added performance for read and write, it is only a small additional step to adding other commonly performed operations to the list as well – and over time, this list has grown to its current (as of NT 3.51) list of thirteen entry points. As we describe each of these entry points, it will become obvious that most of them really *are* geared specifically towards

supporting file systems and fast I/O. These entries are contained in the **FAST_IO_DISPATCH** structure as described in *ntddk.h*. The first element of this structure describes the *size* of the structure, providing a straight-forward mechanism for adding new elements to this data structure in the future in a compatible fashion.

The I/O Manager and Fast I/O

The I/O Manager is responsible for calling the fast I/O entry points, as necessary. The model it uses is actually straight-forward for almost all the calls – the fast I/O entry points return either **TRUE** or **FALSE** indicating whether or not the fast I/O operation was completed. If it was *not* completed, or if fast I/O wasn't available, an IRP is created and sent to the top level driver. The last three entry points in the **FAST_IO_DISPATCH** structure actually do not fit this model, however. Rather they provide slightly different services for the I/O manager. We will discuss them later in this article.

FastIoCheckIfPossible

The first *call* in the **FAST_IO_DISPATCH** structure is *only* used by file systems as part of the common file system runtime library (the *FsRtl* routines).. The prototype for this routine is:

```
typedef BOOLEAN (*PFAST_IO_CHECK_IF_POSSIBLE) (
    IN struct _FILE_OBJECT *FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN ULONG Length,
    IN BOOLEAN Wait,
    IN ULONG LockKey,
    IN BOOLEAN CheckForReadOperation,
    OUT PIO_STATUS_BLOCK IoStatus,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

This routine is *only* used for read and write operations. As such, it is called by the generic fast I/O routines provided in the FsRtl library to allow the specific file system to ascertain if a read or write (depending upon the value of the Boolean *CheckForReadOperation* parameter) can be satisfied from the file cache, while using a generic implementation of managing the file system cache. Note that the parameters to this call are extremely similar to those for the read and write fast I/O entry points – except that this routine does not have any data buffer associated with it.

FastIoRead and FastIoWrite

This routine is called by the I/O manager whenever a read request is made for a file which has valid cached data associated with it. The prototype for this routine is:

```
typedef BOOLEAN (*PFAST_IO_READ) (
    IN struct _FILE_OBJECT *FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN ULONG Length,
    IN BOOLEAN Wait,
    IN ULONG LockKey,
    OUT PVOID Buffer,
    OUT PIO_STATUS_BLOCK IoStatus,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

As we noted previously, the basic parameters to this call are extremely similar to those used by *FastIoCheckIfPossible*, with this call having the necessary data buffer. All of these entry points are guaranteed that the passed in parameters have been validated so that, for instance, the buffer pointer is valid and available for reading in the calling thread's context.

The fast I/O routine can do one of two things: it can complete the operation, set the *IoStatus* field to indicate the result codes for the operation and return **TRUE** to the I/O manager. If that is the case, the I/O manager will complete the I/O operation. Alternatively, the routine can return **FALSE**, in which case the I/O manager will simply create an IRP and call the standard dispatch entry point.

Note that returning **TRUE** doesn't always guarantee the data has been transferred. For example, a read which *starts* past the end of file causes the *IoStatus.Results* field to be set to **STATUS_END_OF_FILE**, with no data copied. A read which crosses over the end of file will cause a **TRUE** to be returned, again with **STATUS_END_OF_FILE** set in the *Results* field, but this time with all remaining data in the file copied to the buffer.

Similarly, returning **FALSE** doesn't always guarantee that some data has not been transferred. While less likely, it is possible for some data to be successfully copied but then to experience an I/O error, or to have the memory of the buffer become inaccessible.

In either case a number of secondary effects can occur. For example, while reading from the cache, it is possible that some of the data being read is not currently resident. This will result in a page fault which will result in a call back into the file system to satisfy the page fault.

The *only* difference for the write case is that the *Buffer* argument is **IN** rather than **OUT**. The basic processing model is very similar. Of course, some error conditions are different – the media could be full, new pages may need to be allocated, etc.

FastIoQueryBasicInfo and FastIoQueryStandardInfo

These operations provide support for the standard *NtQueryInformationFile* API operation, while *FastIoQueryBasicInfo* is also used for certain operations associated with *NtCreateFile*. The *basic* information includes information about when the file was created, when it was last accessed, when it was last modified, and any special attributes of the file – such as being a hidden file, a directory, or other appropriate attributes. The *standard* information includes information about the allocation size being used for the file, the current size of the file, the number of hard links to the file, an indication if deletion has been requested for the file, and an indicator if this particular file is a directory.

Because this information is often cached in memory, it is a perfect candidate for fast I/O operations. Indeed, many standard utilities probe this information on a routine basis, which means that enhancing the performance of these operations will substantially increased the perceived performance of utilities, such as the *File Manager (winfile.exe)*.

The two fast I/O routines have the same interface:

```
typedef BOOLEAN (*PFAST_IO_QUERY_BASIC_INFO) (
    IN struct _FILE_OBJECT *FileObject,
    IN BOOLEAN Wait,
    OUT PFILE_BASIC_INFORMATION Buffer,
    OUT PIO_STATUS_BLOCK IoStatus,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *Wait* parameter indicates if the caller is willing to block waiting for the requisite information. If this is set to **FALSE** then the call must either complete *without* waiting, or must return **FALSE** – in which case a normal IRP can be created, including the necessary context for the full operation. Interestingly enough, it appears that in NT 3.51 neither of these entry points are called with *Wait* set to **FALSE**. Of course, this might change in future versions of NT.

Once invoked, these routines typically look at information they stored for the file when the *FileObject* was first opened. This information can also be reconstructed "on the fly" – for example, the last access time is set to the current system time by some file systems drivers. Of course, setting these values is determined by the file system implementation.

FastIoLock, FastIoUnlockSingle, FastIoUnlockAll, and FastIoUnlockAllByKey

These entry points are used to control *lock state* associated with a particular file. The *locking* being controlled by these calls is *byte range* locking against a single file. Thus, more than one byte range of a file can be locked. While not required, standard NT file systems utilize the services of the file system runtime package (the *FsRtl* routines) which provides a common code base for verifying that a requested lock can be granted and storing information about the lock ranges presently held on the file. Lock state can be controlled via the NT API calls *NtLockFile* as well as *NtUnlockFile*.

Locks in Windows NT are one of two types – either *exclusive* locks, meaning the locked byte range is locked for modification, or *shared* locks, meaning the locked byte range is locked for reads. Multiple shared locks can be granted against overlapping byte ranges, and each is then stored until it is later released. Various pieces of information are stored about *each* lock so it can be acquired for rapid access later.

The interface for *FastIoLock* is:

```
typedef BOOLEAN (*PFAST_IO_LOCK) (
    IN struct _FILE_OBJECT *FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN PLARGE_INTEGER Length,
    PEPROCESS ProcessId,
    ULONG Key,
    BOOLEAN FailImmediately,
    BOOLEAN ExclusiveLock,
    OUT PIO_STATUS_BLOCK IoStatus,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

Thus, the *FileOffest* and *Length* parameters correspond to the byte range being locked by the caller. The *ProcessId* provides information to identify the process which took out the lock – allowing cleanup later, for instance, when that process exits. The *Key* parameter provides an opaque value which can be used to associate multiple locks together for rapid access via the *FastIoUnlockAllByKey* call, for example. *FailImmediately* indicates if the call should *block* until the lock is available, or should immediately return failure. For the *FsRtl* routine, *FailImmediately* is ignored – if the lock is not available, **FALSE** is returned to the caller. The *ExclusiveLock* parameter indicates if this lock request is for exclusive (write) access or shared (read) access.

The *FastUnlockSingle* routine is used to release the byte range locking on a portion of the file. The prototype for this call is:

```
typedef BOOLEAN (*PFAST_IO_UNLOCK_SINGLE) (
    IN struct _FILE_OBJECT *FileObject,
```

```
    IN PLARGE_INTEGER FileOffset,
    IN PLARGE_INTEGER Length,
    PEPROCESS ProcessId,
    ULONG Key,
    OUT PIO_STATUS_BLOCK IoStatus,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

For most file systems, unless the file has associated Op Locks, this operation always returns **TRUE**, since even if the byte lock range is not accessible the operation has been completed (albeit with an error status). Since making the same call with an IRP would generate the same result, processing is completed at this stage.

For this unlock operation to succeed, the *FileOffset*, *Length*, *ProcessId*, and *Key* must all match an existing byte range lock. Otherwise, the operation should complete with the error **STATUS_RANGE_NOT_LOCKED** set in the *IoStatus* block which will then be returned to the caller. The *FastIoUnlockAll* routine is used to release *all* byte range locks associated with a particular file being held by a particular process. The prototype for this function is:

```
typedef BOOLEAN (*PFAST_IO_UNLOCK_ALL) (
    IN struct _FILE_OBJECT *FileObject,
    PEPROCESS ProcessId,
    OUT PIO_STATUS_BLOCK IoStatus,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

In this case, the fast I/O routine searches the available list of locks for the particular file and deletes *any* lock, whether exclusive or share, which is associated with the given *ProcessId*. This is used by the system when *NtCloseFile* is called, either because a program closed it or because a process was deleted.

The *FastIoUnlockAllByKey* operation is used to delete a set of byte range locks which have been logically associated by the *caller* using a particular key value. The prototype for this routine is:

```
typedef BOOLEAN (*PFAST_IO_UNLOCK_ALL_BY_KEY) (
    IN struct _FILE_OBJECT *FileObject,
    PVOID ProcessId,
    ULONG Key,
    OUT PIO_STATUS_BLOCK IoStatus,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

This call is provided for the benefit of *file servers* such as SRV. The I/O Manager in NT 3.51 does not appear to make any use of this call. This key allows a file server to associate a file lock with some remote client. Since it may represent many such remote clients, the use of the *ProcessId* alone is not sufficient. Similarly, since there are multiple file servers, the use of the *Key* alone might cause the erroneous release of file locks for some other file server. Combining the two ensures correct operation and allows for remote system locking.

FastIoDeviceControl

This entry point is used for supporting the native *NtDeviceIoControlFile* call which is used essentially to implement private communications channels to kernel resident drivers. As with the other Fast I/O routines, this routine returns **TRUE** if the operation was completed, and **FALSE** otherwise. For the FALSE case, the I/O Manager creates an IRP and calls the dispatch entry point for the driver in question.

The prototype for *FastIoDeviceControl* is:

```
typedef BOOLEAN (*PFAST_IO_DEVICE_CONTROL) (
    IN struct _FILE_OBJECT *FileObject,
    IN BOOLEAN Wait,
    IN PVOID InputBuffer OPTIONAL,
    IN ULONG InputBufferLength,
    OUT PVOID OutputBuffer OPTIONAL,
    IN ULONG OutputBufferLength,
    IN ULONG IoControlCode,
    OUT PIO_STATUS_BLOCK IoStatus,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

Wait indicates if the caller is willing to wait for the operation to complete, although the Fast I/O routine may return **FALSE** even if *Wait* is set to **TRUE**. The *InputBuffer* points to an optional and potentially validated caller provided buffer which is considered "optional" from the perspective of this interface, although the implementation of the Fast I/O routine can return **TRUE** and indicate an error (e.g., **STATUS_INVALID_PARAMETER**) if the *InputBuffer* is missing or of the wrong length. The *OutputBuffer* points to memory which *may* be validated, depending upon the access type for the *IoControlCode* value. If *IoControlCode* indicates an access type of 3, the *OutputBuffer* is not validated and the Fast I/O routine is responsible for its validation. Otherwise, the buffer is validated prior to calling the Fast I/O routine. This approach is identical to that used for the normal **IRP_MJ_DEVICE_CONTROL** dispatch entry point.

The implementation of this Fast I/O routine is dependent entirely upon the *IoControlCode* value passed to the *FastIoDeviceControl* routine. Typically, this entry point is associated with those kernel mode drivers which implement a rich private communications interface. Thus, none of the NT native file systems actually implement this entry point, but the WinSock support driver *AFD* uses this interface heavily for communications to the underlying transport drivers.

Thus, the actual model for these communications is dependent entirely upon the driver implementation of the Fast I/O routines.

AcquireFileForNtCreateSection and ReleaseFileForNtCreateSection

These two routines do not fit the pattern established by the other entries in the Fast I/O dispatch table. Rather, they appear to have been used to resolve certain locking issues surrounding the HPFS file system (as of the four native NT file systems, only HPFS actually *uses* these entries). The prototype for these two calls is identical and quite simple:

```
typedef VOID (*PFAST_IO_ACQUIRE_FILE) (
    IN struct _FILE_OBJECT *FileObject
    );
```

The *AcquireFileForNtCreateSection* call is made prior to mapping in pages from a file stored within the file system to ensure that any driver-specific locking has been done. The *ReleaseFileForNtCreateSection* call is made to release the locks acquired earlier for the file mapping operation.

As it turns out, if these entry points are not provided, the I/O Manager utilizes a default mechanism for ensuring proper synchronization for the file mapping operations.

FastIoDetachDevice

This last routine is a most curious one. The **ntddk.h** file does provide a hint as to the purpose of this call – namely, it is called when a device object is about to be deleted. We found this call is *extremely* useful when developing filter drivers for *file systems*, since the device objects of removable file systems are destroyed whenever the underlying media is changed. Sometimes this occurs immediately, but it can occur at almost any time *after* the media has been removed, depending upon what portions of the system are still caching information.

The prototype of this call is:

```
typedef VOID (*PFAST_IO_DETACH_DEVICE) (
    IN struct _DEVICE_OBJECT *SourceDevice,
    IN struct _DEVICE_OBJECT *TargetDevice
    );
```

In our experience, a filter driver for removable media file systems *must* be able to handle this call as the system will halt otherwise.

FastIoQueryNetworkOpenInfo

A common operation for the LanManager (CIFS) file server is to open a file and retrieve both its standard and basic attributes. This information is then combined and sent to the remote client (the LanManager/CIFS redirector) so that it can be presented to the remote application programs.

Prior to NT 4.0, SRV was required to pass an IRP to the underlying file system multiple times to extract the same information. Beginning with NT 4.0 a new information type, identified by the *FileNetworkOpenInformation* file information type, was added to the file systems interface so that a network file server, such as SRV, could extract this information in a single operation. To further speed this process, a corresponding fast I/O entry point was also added. The prototype for this is:

```
typedef BOOLEAN (*PFAST_IO_QUERY_NETWORK_OPEN_INFO) (
    IN struct _FILE_OBJECT *FileObject,
    IN BOOLEAN Wait,
    OUT struct _FILE_NETWORK_OPEN_INFORMATION *Buffer,
    OUT struct _IO_STATUS_BLOCK *IoStatus,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

As with the other fast I/O routines, the *FileObject* represents the file on which the caller is acting. The *Wait* parameter indicates if the caller is willing to block. If *Wait* is **FALSE** this routine should not block. The caller can then use the standard IRP path to obtain this information from the file system. The *Buffer* argument points to the location where the data should be copied, and the *IoStatus* block points to the standard I/O status information. Finally, the *DeviceObject* represents the file system instance being queried.

Since this is used by SRV, this is only implemented by physical file systems.

FastIoAcquireForModWrite

This call was added to allow an alternate mechanism for locking the file prior to the *Modified Page Writer* in the memory manager actually performing I/O. This function is entirely optional - but if you do *not* implement it, the File System Runtime Library will us the **ERESOURCE** pointers in the common header

of the file object to ensure correct synchronization (so your file system *must* be using the standard NT locking model.)

```
typedef NTSTATUS (*PFAST_IO_ACQUIRE_FOR_MOD_WRITE) (
    IN struct _FILE_OBJECT *FileObject,
    IN PLARGE_INTEGER EndingOffset,
    OUT struct _ERESOURCE **ResourceToRelease,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *FileObject* identifies the specific file to be locked, the *EndingOffset* indicates the highest byte in the file that the Modified Page Writer is going to write. You can use this information in your file system to synchronize anyone attempting to truncate the file during this operation. The *ResourceToRelease* is an optional parameter your FSD can return. If you return *anything except* a **PERESOURCE** you must also implement the complimentary entry point (*FastIoReleaseForModWrite*) which is described later in this article. The *DeviceObject* represents the file system instance.

FastIoMdlRead

This call was added to allow optimal performance for SRV (or other kernel resident services.) Prior versions of Windows NT allowed SRV to retrieve MDLs into the cache directly, but only when using the IRP path. This new entry point now provides a direct function call method for SRV to achieve the same results.

```
typedef BOOLEAN (*PFAST_IO_MDL_READ) (
    IN struct _FILE_OBJECT *FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN ULONG Length,
    IN ULONG LockKey,
    OUT PMDL *MdlChain,
    OUT PIO_STATUS_BLOCK IoStatus,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *FileObject* uniquely identifies the file being read, while the *FileOffset* and *Length* arguments identify the region of the file being read. The *LockKey* is used to perform the correct mandatory byte range locking checks, as this read is an "application level" read. The *MdlChain* is one (or more) MDLs describing the cache buffer. The *IoStatus* block indicates the completion status of the read operation (since this may require the data be read from disk.) The *DeviceObject* represents the file system instance.

Typically, this is implemented by file systems either using support routines from the FsRtl package or by calling CcMdlRead direcly.

FastIoMdlReadComplete

This call is used by SRV (or other kernel-resident services) to release an MDL previously acquired via a call to *FastIoMdlRead*.

```
typedef BOOLEAN (*PFAST_IO_MDL_READ_COMPLETE) (
    IN struct _FILE_OBJECT *FileObject,
    IN PMDL MdlChain,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *FileObject* identifies the file for which the MDL is being released. The *MdlChain* is the chain returned from a previous call to *FastIoMdlRead*. The *DeviceObject* represents the file system instance.

Typically, this is implemented by file systems either using support routines from the FsRtl package or by calling CcMdlReadComplete.

File System Filter Driver Writers beware! CcMdlReadComplete calls this entry point and ignores the return value (as of NT 4.0 SP3) which will cause memory loss when filtering any file system which uses CcMdlReadComplete (and both FAT and NTFS work in this fashion.)

FastIoPrepareMdlWrite

This entry point is used by SRV (or other kernel-resident services) to obtain a pointer to a range of memory within the cache which can be written to directly. This avoids the "memory copy" limitations inherent in providing buffers (as application programs do.)

```
typedef BOOLEAN (*PFAST_IO_PREPARE_MDL_WRITE) (
    IN struct _FILE_OBJECT *FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN ULONG Length,
    IN ULONG LockKey,
    OUT PMDL *MdlChain,
    OUT PIO_STATUS_BLOCK IoStatus,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *FileObject* uniquely identifies the file being written, while the *FileOffset* and *Length* arguments identify the region of the file being written. The *LockKey* is used to perform the correct mandatory byte range locking checks, as this write is an "application level" write. The *MdlChain* is one (or more) MDLs describing the cache buffer. The *IoStatus* block indicates the completion status of the write operation (since this may require that some data be read from disk.) The *DeviceObject* represents the file system instance.

Typically, this is implemented by file systems either using support routines from the FsRtl package or by calling CcMdlWrite direcly.

FastIoMdlWriteComplete

This call is used by SRV (or other kernel-resident services) to release an MDL previously acquired via a call to *FastIoMdlRead*.

```
typedef BOOLEAN (*PFAST_IO_MDL_WRITE_COMPLETE) (
    IN struct _FILE_OBJECT *FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN PMDL MdlChain,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *FileObject* identifies the file for which the MDL is being released. The *MdlChain* is the chain returned from a previous call to *FastIoMdlRead*. The *DeviceObject* represents the file system instance.

Typically, this is implemented by file systems either using support routines from the FsRtl package or by calling CcMdlReadComplete.

File System Filter Driver Writers beware! CcMdlWriteComplete calls this entry point and ignores the return value (as of NT 4.0 SP3) which will cause data and memory loss when filtering any file system which uses CcMdlReadComplete (and both FAT and NTFS work in this fashion at present.)

FastIoReadCompressed

This call was added to allow SRV (or other kernel-resident services) to fetch data in compressed format from the underlying file system. This can only be used for files that are compressed using the standard Windows NT compression libraries (such as "LZW1".) This routine can be used to either copy the data to a *buffer* provided by the caller, or it can return the data in a fashion described as an MDL.

```
typedef BOOLEAN (*PFAST_IO_READ_COMPRESSED) (
    IN struct _FILE_OBJECT *FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN ULONG Length,
    IN ULONG LockKey,
    OUT PVOID Buffer,
    OUT PMDL *MdlChain,
    OUT PIO_STATUS_BLOCK IoStatus,
    OUT struct _COMPRESSED_DATA_INFO *CompressedDataInfo,
    IN ULONG CompressedDataInfoLength,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *FileObject* uniquely identifies the file being read, while the *FileOffset* and *Length* arguments identify the region of the file being read. The *LockKey* is used to perform the correct mandatory byte range locking checks, as this read is an "application level" read. The *Buffer* argument is an (optional) buffer into which the compressed data is to be copied. The *MdlChain* (also optional) is a pointer which will be set to point to one (or more) MDLs describing the cache buffer. The *IoStatus* block indicates the completion status of the read operation (since this may require the data be read from disk.) The *CompressedDataInfo* and *CompressedDataInfoLength* identify the buffer provided where the compression information for the block being read is copied. The *DeviceObject* represents the file system instance.

FastIoWriteCompressed

This call was added to allow SRV (or other kernel-resident services) to store data in compressed format in the underlying file system. This can only be used for files that are compressed using the standard Windows NT compression libraries (such as "LZW1".)

```
typedef BOOLEAN (*PFAST_IO_WRITE_COMPRESSED) (
    IN struct _FILE_OBJECT *FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN ULONG Length,
    IN ULONG LockKey,
    IN PVOID Buffer,
    OUT PMDL *MdlChain,
    OUT PIO_STATUS_BLOCK IoStatus,
    IN struct _COMPRESSED_DATA_INFO *CompressedDataInfo,
    IN ULONG CompressedDataInfoLength,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *FileObject* uniquely identifies the file being written, while the *FileOffset* and *Length* arguments identify the region of the file being modified. The *LockKey* is used to perform the correct mandatory byte

range locking checks, as this write is an "application level" write. The *MdlChain* is a pointer which will be set to point to one (or more) MDLs describing the cache buffer. The *IoStatus* block indicates the completion status of the read operation (since this may require the data be read from disk.) The *CompressedDataInfo* and *CompressedDataInfoLength* identify the buffer provided where the compression information for the block being read is copied. The *DeviceObject* represents the file system instance.

FastIoMdlReadCompleteCompressed

This call is used by SRV (or other kernel-resident services) to release an MDL previously acquired via a call to *FastIoMdlReadCompressed*.

```
typedef BOOLEAN (*PFAST_IO_MDL_READ_COMPLETE_COMPRESSED) (
    IN struct _FILE_OBJECT *FileObject,
    IN PMDL MdlChain,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *FileObject* identifies the file for which the MDL is being released. The *MdlChain* is the chain returned from a previous call to *FastIoMdlReadCompressed*. The *DeviceObject* represents the file system instance.

FastIoMdlWriteCompleteCompressed

This call is used by SRV (or other kernel-resident services) to release an MDL previously acquired via a call to *FastIoMdlWriteCompressed*.

```
typedef BOOLEAN (*PFAST_IO_MDL_WRITE_COMPLETE_COMPRESSED) (
    IN struct _FILE_OBJECT *FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN PMDL MdlChain,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *FileObject* identifies the file for which the MDL is being released. The *MdlChain* is the chain returned from a previous call to *FastIoMdlWriteCompressed*. The *DeviceObject* represents the file system instance.

FastIoQueryOpen

This routine is used by SRV (or other kernel-resident services) to open a file, retrieve its "network information" and close the file - all in a single operation. The *Irp* passed into this routine is an **IRP_MJ_CREATE** request with the necessary information for the underlying file system to actually retrieve the files information.

```
typedef BOOLEAN (*PFAST_IO_QUERY_OPEN) (
    IN struct _IRP *Irp,
    OUT PFILE_NETWORK_OPEN_INFORMATION NetworkInformation,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *Irp* argument identifies the file being opened (it is a fully constructed **IRP_MJ_CREATE** request.) The *NetworkInformation* argument points to a buffer where the file system should store the requisite information. The *DeviceObject* represents the file system instance.

FastIoReleaseForModWrite

This routine is used by the FsRtl package to release any resources previously acquired by a call to *FastIoAcquireForModWrite* (described previously.)

```
typedef NTSTATUS (*PFAST_IO_RELEASE_FOR_MOD_WRITE) (
    IN struct _FILE_OBJECT *FileObject,
    IN struct _ERESOURCE *ResourceToRelease,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

If *FastIoReleaseForModWrite* returns a **PERESOURCE** it is not necessary for a file system to implement this entry point. However, if the file system is utilizing a more complex synchronization model, or wishes for some other reason to be called to release the file, this routine can be implemented.

The *FileObject* identifies the specific file being released. The *ResourceToRelease* corresponds to the value returned from a previous call to *FastIoAcquireForModWrite*. The *DeviceObject* represents the file system instance.

FastIoAcquireForCcFlush

This routine is used by the FsRtl package to acquire any file system resources necessary prior to the *Lazy Writer* in the cache manager writing dirty data back to the file system. If this entry point is not implemented by the file system, the FsRtl routine uses the information in the common header (*FileObject->FsContext*) to lock the **PERESOURCE**s to which it points.

```
typedef NTSTATUS (*PFAST_IO_ACQUIRE_FOR_CCFLUSH) (
    IN struct _FILE_OBJECT *FileObject,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *FileObject* represents the specific file being flushed. The *DeviceObject* represents the file system instance.

FastIoReleaseForCcFlush

This routine is used by the FsRtl package to release any file system resources acquired via a previous call to *FastIoAcquireForCcFlush*. If this entry point is not implemented by the file system, the FsRtl routine uses the information in the common header (*FileObject->FsContext)* to unlock the **PERESOURCE**s to which it points.

```
typedef NTSTATUS (*PFAST_IO_RELEASE_FOR_CCFLUSH) (
    IN struct _FILE_OBJECT *FileObject,
    IN struct _DEVICE_OBJECT *DeviceObject
    );
```

The *FileObject* represents the specific file being flushed. The *DeviceObject* represents the file system instance.

Uses for Fast I/O

For most Windows NT drivers, there will never be *any* need to use these fast I/O entry points. The I/O manager does not require them and will create IRPs for most of these I/O operations (with *AcquireFileForNtCreateSection*, *ReleaseFileForNtCreateSection*, and *FastIoDetachDevice* being the exceptions). However, for file system drivers, these calls are an important part of enhancing the performance of the file system.

Similarly, for anyone attempting to develop their *own* TDI, they could use a driver much like AFD to provide the interface (via the IOCTL control channel) between user-mode access packages and the kernel mode TDI interface. Again, while this is not *required* for correct performance, it does seem to be a feature which could be added when enhancing performance of the system.

A third category of drivers would be those kernel mode drivers which combine some of the features of file systems and network communications. For example, **NPFS** the Named Pipe File System, uses fast I/O to provide a performance tuned communications channel.

Finally, the fourth use we have found for Fast I/O is in the development of file system filter drivers – where they are virtually *required*. As it turns out, if the underlying device has a particular Fast I/O entry point, the *filtering driver* must also have that Fast I/O entry point – otherwise it will cause the system to crash under certain circumstances.