Chapter 6: I/O Operations

This chapter discusses the routines a device driver can use to perform Input/Output Operations. Also included are sections discussing the driver's I/O Poll, Interrupt Service, and Timeout routines.

I/O Requests

When NetWare has an I/O request for a specific device, NetWare calls the driver's I/O Poll routine and passes a device handle and a pointer to an I/O Request structure. This call is made once and only once for each I/O Request. However, there can be cases where the call is canceled because the device driver has already obtained the request. The I/O Request structure defines the I/O request. Figure 6-1 shows the fields defined in an I/O Request structure.

```
IORequestStruct     struc
    DriverLink          dd  ?
    DiskHandle          dd  ?
    CompletionCode      dw  ?
    Function            db  ?
    Parameter1          db  ?
    Parameter2          dd  ?
    Parameter3          dd  ?
IORequestStruct     ends
```

```
typedef struct IORequestStructure {
        struct IORequestStructure *DriverLink;
        DiskStruct *DiskHandle;
        WORD CompletionCode;
        BYTE Function;
        BYTE Parameter1;
        LONG Parameter2;
        LONG Parameter3;
        } IORequestStruct;
```

Figure 6-1 An I/O Request Structure

Each parameter in the I/O Request structure is defined below:

DriverLink — In NetWare v3.1x and v4.xx, this field will be either zeroed or will contain the request handle of an adjacent request that the OS determines to be contiguous with this request. This allows the driver to get multiple contiguous requests and to combine them into single larger requests which if supported by the device will enhance performance. Be aware that application NLMs that bypass the OS cache by making direct I/O requests may invalidate this field. Always verify the continuity of the requests. The driver may choose to ignore this field or use it for another purpose. In NetWare versions prior to v3.11, the DriverLink field is undefined.

DiskHandle — This parameter specifies the target device. This is the same value as that returned by the NetWare routine AddDiskDevice. The field is not valid until after the IOCTL has been acquired using a *GetRequest*.

CompletionCode — The driver fills in this parameter before returning the I/O Request structure to NetWare. See the I/O completion codes section below for a description of the possible status codes which may be returned by the driver. This field is also available for use prior to the driver placing completion status in it and calling PutRequest to post completion of the request.

Function — This parameter specifies the function requested. See the section on function codes below.

Parameter1 — This parameter normally specifies the number of sectors to transfer, but has other meanings for functions other than read or write (see function descriptions).

Parameter2 — This parameter normally specifies the starting sector to be transferred, but has other meanings for functions other than read or write (see function descriptions).

Parameter3 — This parameter normally specifies the server buffer address to or from which data will be transferred, but has other meanings for functions other than read or write (see function descriptions).

I/O Request Return Status

Drivers use the CompletionCode field in the IORequestStructure to 1) return an I/O request completion status to the OS and 2) return a completion or device status to the calling application. The status returned to the application is a two byte code. The status returned to the OS is embedded in the low-order byte of the status returned to the application. The general set of status codes and their definitions are listed below. Valid codes for individual I/O requests are listed in their specific definitions.

I/O Request Completion Status returned to the OS (low-order byte)

No Error	xx00h
Corrected Media Error	xx01h
Media Error	xx02h
Non-Media Error (fatal)	xx03h
No Action by OS	xx04h - xxFFh

Completion/Device Status returned to the calling application

No Error	0000h
Corrected Media Error	0001h
Media Error	0002h
Non-Media Error (fatal)	0003h
Device Not Active	0004h
Not Supported By Device	0008h
EOT (fatal)	0203h
EOT (non-fatal)	0209h
EOF (non-fatal)	0309h
End Of Partition (non-fatal)	0409h
Early Warning Area (no error)	0500h
Early Warning Area (corrected)	0501h
Early Warning Area (non-fatal)	0509h
Media Change (fatal)	0603h
No Media Present (fatal)	0703h
Media Write Protected (non-fatal)	0809h
Set Marks Detected (non-fatal)	0909h
Blank Media (non-fatal)	0A09h
Unformatted Media (non-fatal)	0B09h
Device Off-Line (non-fatal)	0C09h
Media Previously Written (non-fatal)	0D09h
Abort - Prior State (non-fatal)	0E09h
Driver Custom Status	E000h - FE00h

Figure 6-2 I/O Request Return Status

I/O Request Completion Status returned to the OS:
(low-order byte)

No Error	The request was completed successfully (with no media correction). The OS continues normally.
Corrected Media Error	The request was completed successfully and the data is valid, but a correction was made for a media error which was detected. The OS will HotFix the problem sectors.
Media Error	The request was not completed successfully. The OS will HotFix the failing sectors.
Non-Media Error (fatal)	The request was not completed successfully. A cause other than a data error has occurred. The OS will de-activate the device by issuing a "Deactivate Device" IOCTL. The driver should not service any further I/O requests (get them from the OS using *GetRequest*) until the device has been deactivated. Requests subsequent to the deactivation should be returned with a "Device Not Active" completion code.
No Action by OS	The OS takes no action except to return an error code back to the calling application.

Completion/Device Status returned to the calling application:

No Error	The request was completed successfully with no device exception status.
Corrected Media Error	The request completed successfully; however, a correction was made for a detected media error. The OS will HotFix the problem sectors.
Media Error	The request was not completed successfully.
Non-Media Error (fatal)	The request failed fatally because an undetermined device error, not a media error has occurred.
Device Not Active	The request failed non-fatally. The device has been de-activated and is no longer functional. Previously, either the driver has determined that the device is not able to function properly and has requested that it be de-activated, or the device has been de-activated by a user or OS request.
Not Supported by Device	The request failed. The Device does not support the requested function.
EOT (fatal)	The request failed fatally. The End of Media (EOT) was encountered during the operation.
EOT (non-fatal)	The request failed non-fatally. The End Of Media (EOT) was

encountered during the operation.

EOF (non-fatal) — The request failed non-fatally. The End Of File (EOF Tape Mark) was encountered during the operation.

End Of Partition (non-fatal) — The request failed non-fatally. The End of Partition was encountered.

Early Warning Area (no error) — The request completed successfully. The early warning sense area was encountered during the requested operation.

Early Warning Area (corrected) — The request completed successfully with a corrected media area. The early warning sense area was encountered during the requested operation.

Early Warning Area (non-fatal) — The request failed non-fatally. The operation was attempted in the early warning sense area of the media.

Media Change (fatal) — The request failed fatally. The media has been changed since the last operation.

Media Write Protected (non-fatal) — The request failed non-fatally. The media is write-protected.

Set Marks Detected (non-fatal) — The request failed non-fatally. A mark set previously was detected.

Blank Media (non-fatal) — The request failed non-fatally. The media is blank - no data.

Unformatted Media (non-fatal) — The request failed non-fatally. The media requires formatting but has not been formatted.

Device Off-Line (non-fatal) — The request failed non-fatally. The device is off-line or not ready and requires intervention.

Media Previously Written (non-fatal) — The request failed non-fatally. A WORM device has had the requested area written previously.

Abort - Prior State (non-fatal) — The request failed non-fatally. A previously reported condition (End of Media, etc.) exists which requires subsequent requests (including this one) to be aborted.

Driver Custom Status — These codes are available for drivers to use to return special or custom status to associated NLMs. Use of these codes will prevent the driver from working with other NLMs which are not aware of the custom codes designated by the driver.

Standard I/O Functions

This section contains a summary of standard requests, along with their functional descriptions, command codes, and possible return codes. The following apply to all drivers with regard to function processing and posted completions:

A) Drivers should respond to all unsupported I/O function requests with a status code of Not Supported by Device (0008h).

B) The CompletionCode field in the IORequestStructure (see Figure 6-2) 1) returns an I/O request completion status to the OS and 2) returns a completion or device status to the calling application. The status returned to the application is a two byte code. The status returned to OS is embedded in the low-order byte of the status returned to the application. The valid return codes are I/O request dependent and are described in each request definition.

Standard I/O Functions

Random Read	00h
Random Write	01h
Random Write Once	02h
Sequential Read	03h
Sequential Write	04h
Reset End Of Media Status	05h
Single File Mark(s)	06h
Write single file mark(s)	
Space forward single file mark(s)	
Space backwards single file mark(s)	
ConsecutiveFileMarks	07h
Write Consecutive File Marks	
Space Forward until consecutive file marks	
Space Backwards until consecutive file marks	
SingleSetMark(s)	08h
Write single set mark(s)	
Space forward single set mark(s)	
Space backwards single set mark(s)	
ConsecutiveSetMarks	09h
Write consecutive file marks	
Space forward consecutive set marks	
Space backwards consecutive set marks	
Locate/Space Relative Data Block(s)	0Ah
Space forward data blocks	
Space backwards data blocks	
Locate/Space Absolute Data Block(s)	0Bh
Return absolute position	
Goto absolute position	
SequentialPartitionOperations	0Ch
Partition the media	
Select partition	
Return number of partitions	
Return partition size	
Return max number of possible partitions	
Physical Media Operations	0Dh
Quick erase partition	
Rewind partition	
Goto end of partition	
Security erase partition	
Retention media	
Random Erase	0Eh
Reserved	0Fh–3Fh

RandomRead (00h)

Performs random reads on the media in elevator sorted order

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 0.
BYTE Parameter1	This parameter specifies the number of sectors to be read.
LONG Parameter2	This parameter specifies the logical sector number from which to begin the read.
LONG Parameter3	This parameter points to the buffer where the data is to be received.

Possible completion codes:

0000h	Operation completed with no device exception status
0001h	Operation completed, corrected media error
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0209h	Operation failure (non-fatal), found end of media (EOT)
0409h	Operation failure (non-fatal), found end of partition
0603h	Operation failed (fatal), media change has occurred
0A09h	Operation failure (non-fatal), blank media (not previously written)
0B09h	Operation failure (non-fatal), unformatted media

Description:

The driver is requested to read one or more sectors accessed randomly (for devices capable of random positioning) into the indicated buffer. The driver is responsible for error retry for both data and functional errors. If the driver does not retry an operation, the OS will take action on the reported error without further retries (for example, a media error or corrected media error completion status reported by the driver will cause the associated blocks to be HotFixed or spared. HotFix will also attempt to acquire the data from a mirror if one exists). A non-media error completion status (03h) reported by the driver will cause the device to be de-activated.

RandomWrite (01h)

Performs random writes on the media in elevator sorted order

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 1.
BYTE Parameter1	This parameter specifies the number of sectors to be written.
LONG Parameter2	This parameter specifies the starting logical sector number to be written.
LONG Parameter3	This parameter points to the buffer from which the data is to be written.

Possible completion codes:

0000h	Operation completed with no device exception status
0001h	Operation completed, corrected media error
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0203h	Operation failed (fatal), found end of media (EOT)
0209h	Operation failed (non-fatal), found end of media (EOT)
0603h	Operation failed (fatal), media change has occurred
0809h	Operation failed (non-fatal), device write protected
0B09h	Operation failed (non-fatal), unformatted media
0C09h	Operation failed (non-fatal), device off-line

Description:

The driver is requested to write one or more sectors accessed randomly (for devices capable of random positioning) from the indicated buffer. The driver is responsible for error retry for both data and functional errors. HotFix will recover from a write operation for which the driver has returned a completion status of correctable or non-correctable media error by flagging the block or blocks as bad and writing the data to an alternate location. Non-media completion error status (03h) reported by the driver will cause the device to be de-activated.

RandomWriteOnce (02h)

Performs a random write on Write Once Read Many (WORM) media

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 2.
BYTE Parameter1	This parameter specifies the number of sectors to be written.
LONG Parameter2	This parameter specifies the starting logical sector number to be written.
LONG Parameter3	This parameter points to the buffer from which the data is to be written.

Possible completion codes:

0000h	Operation completed with no device exception status
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0209h	Operation failed (non-fatal), found end of media (EOT)
0603h	Operation failed (fatal), media change has occurred
0809h	Operation failed (non-fatal), device write protected
0A09h	Operation failed (non-fatal), blank media (not previously written)
0B09h	Operation failed (non-fatal), unformatted media
0C09h	Operation failed (non-fatal), device off-line
0D09h	Operation failed (non-fatal), media previously written

Description:

This function is identical to the RandomWrite except that it is specifically designated for a WORM device.

SequentialRead (03h)

Reads the next sector(s) on sequential media

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 3.
BYTE Parameter1	Not used.
LONG Parameter2	This parameter specifies the number of sectors to be read. If the driver does not read all of the requested sectors, it must update this field to indicate the number of sectors which were read.
LONG Parameter3	This parameter points to the buffer where the data is to be received.

Possible completion codes:

0000h	Operation completed with no device exception status
0500h	Operation completed, detected Early Warning Area
0001h	Operation completed, corrected media error
0501h	Operation completed, corrected media error and detected Early Warning Area
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0209h	Operation failed (non-fatal), found end of media (EOT)
0309h	Operation failed (non-fatal), found end of file (EOF)
0409h	Operation failed (non-fatal), found end of partition
0509h	Operation failed (non-fatal), already in Early Warning Area
0603h	Operation failed (fatal), media change has occurred
0909h	Operation failed (non-fatal), found set marks
0A09h	Operation failed (non-fatal), blank media (not previously written)
0B09h	Operation failed (non-fatal), unformatted media
0C09h	Operation failed (non-fatal), device off-line
0E09h	Operation failed (non-fatal), Abort caused by prior driver state

Description:

This function is similar to the RandomRead function, but is issued to devices which support sequential positioning. The high-order bit of the function code must be set to indicate that the function is not to be sorted in sector number order (elevator queues), but is to be issued in the same sequence that it is requested. This function must be initiated from a device level to have the device exception status returned, as required for sequential devices.

SequentialWrite (04h)

Writes the indicated sector(s) at the current position on sequential media

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 4.
BYTE Parameter1	Not used.
LONG Parameter2	This parameter specifies the number of sectors to write. If the driver could not write the entire amount, it should update this field with the number of sectors actually written.
LONG Parameter3	This parameter points to the buffer from which the data is to be written.

Possible completion codes:

0000h	Operation completed with no device exception status
0500h	Operation completed, detected Early Warning Area
0001h	Operation completed, corrected media error
0501h	Operation completed, corrected media error and detected Early Warning Area
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0209h	Operation failed (non-fatal), found end of media (EOT)
0409h	Operation failed (non-fatal), found end of partition
0509h	Operation failed (non-fatal), already in Early Warning Area
0603h	Operation failed (fatal), media change has occurred
0809h	Operation failed (non-fatal), device write protected
0A09h	Operation failed (non-fatal), blank media (not previously written)
0B09h	Operation failed (non-fatal), unformatted media
0C09h	Operation failed (non-fatal), device off-line
0E09h	Operation failed (non-fatal), Abort caused by prior driver state

Description:

This function is similar to the RandomWrite function, but is issued to devices which support sequential positioning. The high-order bit of the function code must be set to indicate that the functions are not to be sorted in sector number order (elevator queues), but are to be issued in the same sequence that they are requested. This function must be initiated from a device level to have the device exception status returned, as required for sequential devices.

ResetEndOfMediaStatus (05h)

Resets a device's EndOFMediaStatus flag to allow a subsequent write in the Early Warning Area

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 5.
BYTE Parameter1	Not used.
LONG Parameter2	Not used.
LONG Parameter3	Not used.

Possible completion codes:

0000h	Operation completed with no device exception status
0008h	Operation failed (non-fatal), function not supported by device

Description:

This function is used to indicate to a driver supporting sequential media that the driver's early warning flag (indicating that the device has entered the early-warning area) should be reset, so that a subsequent function can be performed in the early-warning area of the media. This serves as an override for the next requested function (drivers should fail all requests with 0E09h status upon having detected the early warning area at the conclusion of an operation, until receiving this function request). This allows drivers to be designed to handle multiple requests queued for the device when an early warning condition is detected. This function must be initiated from a device level to have the device exception status returned, as required for sequential devices.

SingleFileMark(s) (06h)

Writes or spaces over single file marks

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 6.
BYTE Parameter1	This parameter indicates the operation to be performed: 0 write file mark 1 space forward 2 space backwards
LONG Parameter2	This parameter indicates the number of file marks associated with the operations. (i.e. the number of file marks to be written or spaced over). This field should be updated by the driver on a space operation to indicate the number of file marks found.
LONG Parameter3	Not used.

Possible completion codes:

0000h	Operation completed with no device exception status
0500h	Operation completed, detected Early Warning Area
0001h	Operation completed, corrected media error
0501h	Operation completed, corrected media error and detected Early Warning Area
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0209h	Operation failed (non-fatal), end of media (EOT)
0409h	Operation failed (non-fatal), end of partition
0509h	Operation failed (non-fatal), already in Early Warning Area
0603h	Operation failed (fatal), media change has occurred
0809h	Operation failed (non-fatal), device write protected
0909h	Operation failed (non-fatal), found set marks
0A09h	Operation failed (non-fatal), found blank media (not previously written)
0B09h	Operation failed (non-fatal), unformatted media
0C09h	Operation failed (non-fatal), device off-line
0E09h	Operation failed (non-fatal), Abort caused by prior driver state

Description:

These functions provide the ability to perform forward and backward single or multiple (but not required to be consecutive) file marks for sequential media. This function must be initiated from a device level to have the device exception status returned, as required for sequential devices.

ConsecutiveFileMark(s) (07h)

Writes or spaces over consecutive file marks.

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 7.
BYTE Parameter1	This parameter indicates the operation to be performed: 0 write file mark. 1 space forward for consecutive file marks 2 space backward for consecutive file marks
LONG Parameter2	This parameter specifies the number of file marks associated with the operation. (i.e. the number of consecutive file marks to write or the number of consecutive file marks to space over.) This field should be updated by the driver on a space operation to indicate the number of file marks found.
LONG Parameter3	Not used.

Possible completion codes:

0000h	Operation completed with no device exception status
0500h	Operation completed, detected Early Warning Area
0001h	Operation completed, corrected media error
0501h	Operation completed, corrected media error and detected Early Warning Area
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0209h	Operation failed (non-fatal), found end of media (EOT)
0409h	Operation failed (non-fatal), found end of partition
0509h	Operation failed (non-fatal), already in Early Warning Area
0603h	Operation failed (fatal), media change has occurred
0809h	Operation failed (non-fatal), device write protected
0909h	Operation failed (non-fatal), found set marks
0A09h	Operation failed (non-fatal), blank media (not previously written)
0B09h	Operation failed (non-fatal), unformatted media
0C09h	Operation failed (non-fatal), device off-line
0E09h	Operation failed (non-fatal), Abort caused by prior driver state

Description:

These functions provide ability to perform forward and backspace multiple (required to be consecutive) file marks for sequential media. This function must be initiated from a device level to have the device exception status returned, as required for sequential devices.

SingleSetMark(s) (08h)

Writes or spaces over single Set Marks

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 8.
BYTE Parameter1	This parameter indicates the operation to be performed: 0 write set mark 1 space forward set marks 2 space backward set marks
LONG Parameter2	The number of set marks associated with the operation. (i.e the number of set marks to be written or the number of set marks to be spaced over.) This field should be updated by the driver to indicate the number of set marks actually spaced over in the operation.
LONG Parameter3	Not used.

Possible completion codes:

0000h	Operation completed with no device exception status
0500h	Operation completed, detected Early Warning Area
0001h	Operation completed, corrected media error
0501h	Operation completed, corrected media error and detected Early Warning Area
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0209h	Operation failed (non-fatal), end of media (EOT)
0409h	Operation failed (non-fatal), end of partition
0509h	Operation failed (non-fatal), already in Early Warning Area
0603h	Operation failed (fatal), media change has occurred
0809h	Operation failed (non-fatal), device write protected
0A09h	Operation failed (non-fatal), found blank media (not previously written)
0B09h	Operation failed (non-fatal), unformatted media
0C09h	Operation failed (non-fatal), device off-line
0E09h	Operation failed (non-fatal), Abort caused by prior driver state

Description:

These functions provide ability to perform forward and backspace single and multiple (but not required to be consecutive) set marks for sequential media. This function must be initiated from a device level to have the device exception status returned, as required for sequential devices.

ConsecutiveSetMark(s) (09h)

Write or spaces over consecutive set marks

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 9.
BYTE Parameter1	This parameter indicates the operation to be performed: 0 write consecutive set marks 1 space forwards consecutive set marks 2 space backwards consecutive set marks
LONG Parameter2	The number of consecutive set marks associated with the operation. (i.e. the number of consecutive set marks to be written or spaced over.) This field should be updated by the driver to indicate the number of set marks actually spaced over in the operation.
LONG Parameter3	Not used.

Possible completion codes:

0000h	Operation completed with no device exception status
0500h	Operation completed, detected Early Warning Area
0001h	Operation completed, corrected media error
0501h	Operation completed, corrected media error and detected Early Warning Area
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0209h	Operation failed (non-fatal), found end of media (EOT)
0409h	Operation failed (non-fatal), found end of partition
0509h	Operation failed (non-fatal), already in Early Warning Area
0603h	Operation failed (fatal), media change has occurred
0809h	Operation failed (non-fatal), device write protected
0A09h	Operation failed (non-fatal), blank media (not previously written)
0B09h	Operation failed (non-fatal), unformatted media
0C09h	Operation failed (non-fatal), device off-line
0E09h	Operation failed (non-fatal), Abort caused by prior driver state

Description:

These functions provide ability to perform forward and backspace multiple (required to be consecutive) set marks for sequential media. This function must be initiated from a device level to have the device exception status returned, as required for sequential devices.

SpaceRelativeDataBlock(s) (0Ah)

Spaces over data blocks relative to the current sequential media position

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 0A (hex).
BYTE Parameter1	This parameter indicates the operation to be performed: 1 space forward data blocks 2 space backward data blocks
LONG Parameter2	The number of data blocks to be spaced over. This field should be updated by the driver to indicate the number of blocks actually spaced over in the operation.
LONG Parameter3	Not used.

Possible completion codes:

0000h	Operation completed with no device exception status
0500h	Operation completed, detected Early Warning Area
0001h	Operation completed, corrected media error
0501h	Operation completed, corrected media error and detected Early Warning Area
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0209h	Operation failed (non-fatal), found end of media (EOT)
0309h	Operation failed (non-fatal), found end of file (EOF)
0409h	Operation failed (non-fatal), found end of partition
0509h	Operation failed (non-fatal), already in Early Warning Area
0603h	Operation failed (fatal), media change has occurred
0909h	Operation failed (non-fatal), found set marks
0A09h	Operation failed (non-fatal), found blank media (not previously written)
0B09h	Operation failed (non-fatal), found unformatted media
0C09h	Operation failed (non-fatal), found device off-line
0E09h	Operation failed (non-fatal), Abort caused by prior driver state

Description:

These functions provide ability to forward or backspace over single or multiple data blocks for sequential media. The operation must be terminated with appropriate status if a file mark, set mark, beginning, or end of media is encountered. If partitioned media is mounted, this function must provide operations relative to the selected partition. This function must be initiated from a device level to have the device exception status returned, as required for sequential devices.

AbsoluteDataBlock (0Bh)

Locates or positions the media to data block location; the absolute location need not be known by the application, the application only receives a token that represents the absolute location to the driver.

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 0B (hex).
BYTE Parameter1	This parameter indicates the operation to be performed: 0 return absolute position 1 goto absolute position
LONG Parameter2	The size of the buffer for the absolute position information; the size value can be obtained from the ReturnTapeDeviceInfo IOCTL.
LONG Parameter3	This parameter points to the buffer where the position token is either passed or returned.

Possible completion codes:

0000h	Operation completed with no device exception status
0500h	Operation completed, detected Early Warning Area
0001h	Operation completed, corrected media error
0501h	Operation completed, corrected media error and detected Early Warning Area
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0209h	Operation failed (non-fatal), found end of media (EOT)
0309h	Operation failed (non-fatal), found end of file (EOF)
0409h	Operation failed (non-fatal), found end of partition
0603h	Operation failed (fatal), media change has occurred
0909h	Operation failed (non-fatal), found set marks
0A09h	Operation failed (non-fatal), found blank media (not previously written)
0B09h	Operation failed (non-fatal), found unformatted media
0C09h	Operation failed (non-fatal), found device off-line
0E09h	Operation failed (non-fatal), Abort caused by prior driver state

Description:

These functions provide applications the tool to position media to a specific absolute block address, or to determine the absolute block address where the media is currently positioned. If partitioned media is mounted this function must provide operations relative to the selected partition (absolute block addresses are absolute block addresses within the current partition). The format of this information only needs to be known by the driver. Applications should use this function as a book marker (or token). The application first asks for the token, and then gives it back when requesting the driver to reposition media to that location.

SequentialPartitionOperations (0Ch)

Provides partition operations on partitioned sequential devices

Parameters:

struct IORequestStructure *DriverLink — Contains a link to a contiguous request or NULL.

DiskStruct *DiskHandle — Contains the device handle returned by *AddDiskDevice*.

WORD CompletionCode — The driver fills this field with a completion status.

BYTE Function — Contains a value of 0C (hex).

BYTE Parameter1 — This parameter indicates the operation to be performed:

- 0 partition the media
- 1 select partition (advance tape to selected partition)
- 2 return number of partitions
- 3 return partition size
- 4 return maximum number of partitions that can be defined

LONG Parameter2 — This parameter's usage is dependent on Parameter1.

- 0 number of partitions to be created.
- 1 partition number (zero relative) of the partition to be selected.
- 2 not used
- 3 partition number (zero relative) of the partition size to be returned.
- 4 not used

LONG Parameter3 — This parameter's usage is dependent on Parameter1.

- 0 handle to an array of LONGs. Each array entry describes the explicit size (in an exponent - mantissa format) of the corresponding partition on the media. The most significant byte contains a value indicating the desired unit size on the partition as follows:
  - 0 byte
  - 1 kilobyte (1024 bytes)
  - 2 megabyte (1024 kilobytes)

  The least significant three bytes contain the number of units. If the last partition is to consist of the remaining space on the media of unknown size, a -1 (0xFFFF) should be placed in the corresponding array entry.
- 1 Not used.
- 2 handle to a LONG that contains the number of partitions on the media.
- 3 handle to a LONG that contains the explicit size (in the above described exponent - mantissa format) of the partition designated in Parameter 2.
- 4 handle to a LONG that contains the maximum number of partitions that can be defined on the tape.

Possible completion codes:

0000h	Operation completed with no device exception status
0500h	Operation completed, detected Early Warning Area
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0209h	Operation failed (non-fatal), found end of media (EOT)
0309h	Operation failed (non-fatal), found end of file (EOF)
0409h	Operation failed (non-fatal), found end of partition
0509h	Operation failed (non-fatal), already in Early Warning Area
0603h	Operation failed (fatal), media change has occurred
0809h	Operation failed (non-fatal), device write protected
0909h	Operation failed (non-fatal), found set marks
0A09h	Operation failed (non-fatal), found blank media (not previously written)
0B09h	Operation failed (non-fatal), found unformatted media
0C09h	Operation failed (non-fatal), found device off-line
0E09h	Operation failed (non-fatal), Abort caused by prior driver state

Description:

These functions provide miscellaneous operations required with sequential media which supports partitions. This function must be initiated from a device level to have the device exception status returned, as required for sequential devices.

PhysicalMediaOperations (0Dh)

Positions media and/or provides special media operations

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 0D (hex).
BYTE Parameter1	This parameter indicates the operation to be performed: 0 quick erase partition 1 rewind partition 2 go to end of partition 3 security erase partition 4 retention media
LONG Parameter2	Not used.
LONG Parameter3	Not used.

Possible completion codes:

0000h	Operation completed with no device exception status
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0603h	Operation failed (fatal), media change has occurred
0809h	Operation failed (non-fatal), device write protected
0C09h	Operation failed (non-fatal), device off-line

Description:

These functions provide miscellaneous operations for sequential media which may be required, depending upon the media type. This function must be initiated from a device level to have the device exception status returned, as required for sequential devices.

RandomErase (0Eh)

Performs random erasures on the media in elevator sorted order (used with magneto-opticals)

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 0E (hex).
BYTE Parameter1	This parameter specifies the number of sectors to be erased.
LONG Parameter2	This parameter specifies the starting logical sector number to be erased.
LONG Parameter3	Not used.

Possible completion codes:

0000h	Operation completed with no device exception status
0001h	Operation completed, corrected media error
0002h	Operation failed, uncorrectable media error
0003h	Operation failed (fatal), device failure cause is undetermined
0004h	Operation failed (non-fatal), device is not active
0008h	Operation failed (non-fatal), function not supported by device
0203h	Operation failed (fatal), found end of media (EOT)
0209h	Operation failed (non-fatal), found end of media (EOT)
0603h	Operation failed (fatal), media change has occurred
0809h	Operation failed (non-fatal), device write protected
0B09h	Operation failed (non-fatal), unformatted media
0C09h	Operation failed (non-fatal), device off-line

Description:

The driver is requested to erase one or more sectors accessed randomly (for devices capable of random positioning). The driver is responsible for error retry for both data and functional errors. Non-media completion error status (03h) reported by the driver will cause the device to be de-activated.

Reserved (0Fh - 3Fh)

These codes are reserved

Parameters:

struct IORequestStructure *DriverLink	Contains a link to a contiguous request or NULL.
DiskStruct *DiskHandle	Contains the device handle returned by *AddDiskDevice*.
WORD CompletionCode	The driver fills this field with a completion status.
BYTE Function	Contains a value of 0F - 3F (hex).
BYTE Parameter1	Not used.
LONG Parameter2	Not used.
LONG Parameter3	Not used.

Possible completion codes:

0008h	Operation failed (non-fatal), function not supported by device

IOPoll

NetWare passes the IOPoll routine a device handle and a pointer to a I/O Request structure when it is called. The device handle indicates to the driver which device should perform the request indicated by the request structure handle. The driver may either use the pointer specified and acquire the associated IORequest structure by issuing a GetRequest call, or the driver may elect to acquire the next queued request from the device queues by calling GetRequest providing a zero as a request handle. However, the driver **can not** use the pointer to actually service the request without first "acquiring" the request by making a call to GetRequest.

The GetRequest routine is passed a device handle and a pointer to an IORequest structure and returns a pointer to an IORequest structure (or zero if none are available). The syntax for GetRequest is shown in Figure 6-3. If the driver is required to retrieve a particular request, the driver must pass a pointer to the request and GetRequest will return the same pointer to the driver. On the other hand, if the driver simply wants whichever request is next, it passes a null pointer in the IORequest parameter, and GetRequest returns a pointer to the next request. Most I/O Poll routines should use the latter method to allow the requests to be serviced in sorted order from the elevator queues.

```
IORequestStruct *GetRequest(
        DiskStruct *DiskHandle,
        IORequestStruct *NextRequest)
```

Figure 6-3 GetRequest Syntax

Once a driver obtains the pointer to the I/O Request structure, the driver should attempt the I/O request. When the request is completed, the driver must fill in the completion code field in the I/O Request structure, then return the structure to NetWare by calling PutRequest (the IORequest and DiskStructure handle are again required).

IOPoll may also be responsible for maintaining the driver's thread of execution either by looping through requests or making a call to itself.

The driver IOPoll routine must accomplish the following:

1) Preserve EBP, EBX, ESI, and EDI on the stack

2) Determine if the request can be issued at this time (the adapter may be busy with prior request and not be able to accept more requests, or the device requested may be busy, and the adapter unable to queue requests).

   It is also possible that the driver may need to service an IOCTL request at this time, and cannot service another *IORequest* until the IOCTL is completed.

3) If the request can not be issued now, the driver may need to record that a request be queued for later service. The *IOPoll* routine may restore the registers saved in step 1 and return to the caller at this time, or examine other devices or cards for possible requests to be started, prior to returning to the caller.

4) If the request can be issued, the request must be obtained by a call to *GetRequest.* Typically drivers will need to get the next request from the elevators in sorted order, in which case the call to *GetRequest* will pass a zero instead of the *IORequestStructure* pointer (request handle). It should be noted that the call to *GetRequest* may be unsuccessful and the request handle returned will be zero. If this occurs, the specific request or next request was not obtained (possibly due to being already taken by a mirror drive). If the call was made to get the next request, no requests are on the elevator for this device, in which case the registers must be restored and return made to the caller (unless, of course, the driver must examine other devices and cards for work to start).

5) For 16-bit Host Adapters using bus-mastering or DMA functionality on servers with more than 16 megabytes of memory, the driver must determine if the physical address of any or all of the request buffer exceeds the 16 meg boundary. If so, the driver must use a buffer (previously allocated using *AllocBufferBelow16Meg*) guaranteed to be below 16 meg. A write function requires the data to be copied into and issued from the special buffer. A read fuctions requires the driver to issue the read to the special buffer, then, upon completion, copy the data read to the actual request buffer (see Appendix G).

6) Initiate the request.

7) See if any other request may be started for other devices or cards.

8) Restore the saved registers and return to the IOPoll caller.

I/O Poll

The I/O Poll routine services I/O requests.

Syntax

```
void IOPoll(
    DiskStruct *DiskHandle,
    IORequestStruct *IORequest)
```

Return Values None

Parameters

DiskHandle	This parameter passes a handle for the target hard disk. NetWare returns this value to the driver's Initialization routine. The value serves two purposes. First, the driver can use the value as a handle for calls to GetRequest and PutRequest. Second, the handle is actually a pointer to a Disk structure. (The Disk structure was established when the driver's initialization routine called AddDiskDevice.)
IORequest	This parameter passes a pointer to an IORequest structure.

Remarks **The name of the I/O Poll routine is arbitrary**. A disk driver's Initialization routine passes the address of the I/O Poll routine to NetWare when the Initialization routine calls the NetWare routine AddDiskDevice.

I/O Polling Outline

(This routine assumes a single drive.)

```
IOPoll(DiskStruct *DiskHandle, IORequestStruct *IORequest)
{

    IORequestStruct      *Request;

    if (DiskHandle->status == Busy)
    {
        ++RequestCount;          //Reminder that this request need to be
        return;                  //serviced later
    }

    while (Request = GetRequest(DiskHandle, 0))
    {

        if ( Request->DiskHandle is deactivated )
        {
            Request->CompletionCode = 04;
            PutRequest(Request->DiskHandle, Request)
        }
        else
        {
            switch ( Request->Function )
            {
                case READ:
                    DriverReadRoutine(Request);         //   this   routine   issues
                    commands
                    break;                          // to read from the drive.

                case WRITE:
                    DriverWriteRoutine(Request);           //  this  routine  issues
                    commands
                    break;                          // to write to the drive.

                case OTHERFUNCTION:
                    DriverOtherFunctionRoutine(Request);   //  test  here  for  other
functions
                                                    // besides read and write.

                    .
                    .
                    .
                    .
                default:
                    break;
            }

            if (DiskHandle->status = Busy)              // status is set to Busy if
            READ,
                                                    // WRITE or other functions have

                return;                                 // not completed.

            Request->CompletionCode = DiskHandle->CompletionStatus;

            PutRequest(TargetDisk, Request);               //    PutRequests    will
            usually be issued
        }                                           // from the ISR routine.
    }
```

Timeout Routine

A driver should never indefinitely attempt to service a read, write, or other host adapter action. To avoid a "forever-in-error" condition, a driver must schedule an asynchronous event. The asynchronous event will rescue the system in case an error occurs from which the driver cannot otherwise recover. This asynchronous event is referred to as the **timeout routine**. If the driver cannot complete the I/O within the allotted time, the timeout routine gets control and allows the driver to recover.

The timeout routine typically checks each adapter for active requests or operations which have exceeded the maximum time allowed for completion. If the time exceeds the driver-established limit, the timeout routine takes over and restarts the request or aborts and posts completion to the failed request. It is possible to utilize many AES events, but since the number of AES event threads are limited, doing so in a blocking environment runs the risk of creating a dead-lock condition that will hang the server. A single AES routine can be used to monitor all active devices. Each device is assigned a variable set to a timeout value. The variables are decremented each time the AES routine is called, and when they reach a terminal value appropriate action may be taken.

The completion of a request may often be delayed by other system events, including other requests which must be retried, so time-out limits should be set generously to allow for system interaction. An appropriate time-out value generally will allow ample time for a few retries yet terminate before the user becomes impatient.

If it is necessary to de-activate a device because of a non-media error caused by a time-out, the affected device will likely be inactive for a long time, which can cause the user a significant delay. The granularity, or interval between timeout routine scans, should be large, typically once a second, so that the processor spends a little of its processing resource protecting itself from events or abnormalities which happen very infrequently, if at all.

Another common problem occurs when a driver times out a specific function but fails to physically abort the function in the host adapter. The request may be completed later, or be timed out internally by the host adapter, after which the host adapter can post an interrupt (or perform a DMA transfer or Bus Master DMA transfer to host memory into what used to be the proper request buffer, which may now belong to some other critical server function) indicating completion. The driver must detect an erroneous interrupt and must not attempt to post completion for a request which has already been posted complete (which may abend the server.) When a driver determines that an event has exceeded its limit, the driver is responsible for terminating any activity which may be harmful to the integrity of the OS.

Timeout routines are scheduled by calling the NetWare routine *ScheduleNoSleepAESProcessEvent*. This routine requires a pointer to the AES Event structure shown in Figure 2-6. The timeout event execution is scheduled initially by the driver during its Initialize Driver routine. To schedule the timeout event, the driver calls *ScheduleNoSleepAESProcessEvent* and passes an AES Event structure specifying the delay interval (in 1/18.2 second ticks) and the Timeout routine. The timeout event must continue to reschedule itself by calling *ScheduleNoSleepAESProcessEvent* (see chapter 7). For example, if the driver specifies a value of 180 in the AESWakeUpDelayInterval, the NetWare schedules a call to the Timeout routine approximately every ten seconds.

AES timers with the NoSleep option are so named because they are not allowed to put the current thread of execution to sleep (by making a call to a blocking routine, etc.). If it is necessary for the timer scan routine to make blocking calls, the driver must specify an AES timer with the Sleep option (allowed to put the current process to sleep) by scheduling the event with *ScheduleSleepAESProcessEvent*.

If the timeout AES structure is maintained locally, the driver is free to add fields to the structure that could help pinpoint problems.

Please note that the AESStructure should be allocated at initialization time and must have a valid resource tag placed in it prior to any call.

To remove the pointer to an AES Event structure from the OS and eliminate a Timeout routine, the driver must cancel the event by calling the *CancelNoSleepAESProcessEvent* or *CancelSleepAESProcessEvent* OS support routines.

TimeOut

The TimeOut routine monitors the drive status and handles error recovery.

Syntax

```
LONG TimeOut(
    AESEventStruct  *AESHandle )
```

Return Values None

Parameters AESHandle Passes a pointer to an AES event structure.

Remarks **The name of the TimeOut routine is arbitrary.** The routine is scheduled and its address passed to NetWare using the ScheduleNoSleepAESProcessEvent or the ScheduleSleepAESProcessEvent API. Interrupts are enabled upon entry into this routine and should be enabled on exit as well.

Interrupt Service Routine

The driver's Interrupt Service Routine (ISR) is called by NetWare in response to a hardware interrupt generated by a specific adapter board. The ISR routine must provide logic to service interrupts for both normal I/O requests and IOCTL requests. The driver must provide a strategy for determining what particular request caused the interrupt. The driver's *CardStructure* and *DiskStructure* are available for this purpose.

The driver acquires or allocates an interrupt by making a call to the OS support routine SetHardwareInterrupt (see chapter 7). This routine requires that the driver indicate whether the interrupt is shared, and if so, whether it should be placed at the first or the last of a shared interrupt chain. The OS places its own ISR routine address in the OS Interrupt Descriptor Table (IDT), saving the drivers ISR address in a table. The OS ISR services the interrupt, saves all registers (including all segment registers), clears the direction flag, sets up necessary segment registers, and calls the driver ISR found in the associated table entry.

The driver's *RemoveDriver* routine returns or de-allocates the interrupt by calling *ClearHardwareInterrupt,* which removes the its ISR routine from the list of those to be called by the OS. All interrupts are serviced initially by the OS ISR using NEAR procedure call. It is necessary for the driver on exit to reset the interrupt in the host adapter and issue any required EOI(s) by calling *CDoEndOfInterrupt*. The ISR must do a RET (near) to return to the OS ISR. Driver ISRs must not issue an IRETD instruction because that is done by the OS ISR upon return from the driver ISR. Performing an IRETD from the driver ISR will normally cause a General Protection Interrupt (GPI).

Required ISR Functional Flow

The driver ISR is required to perform the following steps:

1) Determine the adapter that caused the interrupt and obtain a pointer to either the adapter's IOConfigurationStructure or CardStructure for subsequent I/O instructions to the adapter (typically this is accomplished by registering ISR separate entry points for each adapter, each setting a structure address in a register such as EBP or EBX for the associated IOConfigurationStructure or CardStructure).

2) Save the interrupting adapter status.

3) Optionally mask the adapter interrupt, or the interrupt level in the programmable interrupt controller (PIC). This step is **required** if the ISR is going to service subsequent interrupts while still in the ISR. Some host adapters remove interrupt requests before they are serviced, or otherwise glitch the interrupt request line. If this step is omitted and the above is attempted, some 486 processors may generate a "Lost Interrupt" indication. The problem can be eliminated by preventing the PICs from recording any interrupt requests which will be reset by software or removed by the host adapter without allowing the processor to vector. Masking the interrupt level in the PIC accomplishes this. Drivers can still determine if a subsequent interrupt has become pending in the host adapter (see chapter 7, CCheckHardwareInterrupt).

4) Reset the interrupt request line in the adapter card requesting the interrupt. Failure to reset the adapter interrupt request prior to issuing the EOI(s) may result in a subsequent false or extra interrupt.

5) Reset any software timer for the event which has just completed.

6) The driver ISR should now perform any actions required to service the interrupt, provided that the ISR

remains non-blocking. The ISR may now enable interrupts if desired provided that no EOI(s) have been issued.

7) Reset the system interrupt controller(s) by issuing CDoEndOfInterrupt EOI calls. Previously, with NetWare v3.1x, there were several ways to handle "end of interrupts." It is now mandatory that you only use *CDoEndofInterrupt* to reset system interrupt controllers.

8) Read completions for adapters using shared memory may require data to be moved from the shared memory into the request buffer.

   Read completions for adapters without shared memory, DMA, or Bus-Master DMA, the ISR may need to use string I/O to acquire the data from the adapter.

   Read completions for DMA and Bus-master DMA adapters will typically have already placed the data in the request buffer. Drivers supporting 16-bit Host Adapters on machines with more than 16 megabytes must move the data from a special buffer previously allocated below 16 megabytes to the actual request buffer for requests with real absolute memory addresses above 16 meg.

9) Examination of the adapter completion status must be done to determine if the request completed successfully. Retries may need to be initiated at this point for requests completed with errors.

10) Completion of the request (either good or retry limit expired) must be indicated to the OS by placing the completion status in the associated request structure and making a call to PutRequest (or PutIOCTL if the function completed was an IOCTL request). The driver ISR must be able to function correctly with the PutRequest routine opening an **interrupt enable window** during its execution.

11) The ISR may now need to determine if any further I/O or IOCTL requests are pending for the device (or card), and if one is found, issue a GetRequest (or GetIOCTL), set up the function, and start the operation with the adapter card (the same steps as in IOPoll).

12) Optionally, the ISR may now see if an additional or subsequent interrupt is pending for this interrupt level. If the interrupt level was masked in the PIC in step 3, the ISR may call CCheckHardwareInterrupt to determine if a further interrupt is pending. The subsequent interrupt will not occur if the interrupt was masked and is serviced and reset prior to un-masking the PIC in step 13.

13) The ISR must un-mask (enable) the adapter or PIC if masked in step 3. If the PIC interrupt level was masked, the driver can unmask it by calling *CEnableHardwareInterrupt*.

14) The ISR returns to the caller (the system ISR). The return must be accomplished using a NEAR return or RET instruction with **processor interrupts disabled** (under no circumstances may the driver ISR execute an IRETD instruction).

Disabling Interrupts & Routine Duration

It is undesirable to have a driver keep interrupts disabled for any long period. This can cause lost data on unbuffered devices, and can cause delays in I/O on devices with interrupts pending. The ability of a driver to enable interrupts is determined by the driver design and also the host adapter's limitations. A Driver should be designed to minimize the duration of execution with interrupts disabled. The following guidelines should be kept in mind:

A) Sometimes a driver can accomplish functions required in an ISR very quickly. If it is possible to service the device very quickly and return, then it is easier to leave interrupts disabled for the duration of the ISR. This would allow the ISR to finish any interrupt-related activities, rather than spend the additional time to preserve registers to allow other interrupts.

B) If the interrupt cannot be serviced quickly (a few hundred instructions), and the driver can be constructed to run with interrupts enabled after initial housekeeping, it is desirable to enable interrupts after initial housekeeping.

C) Drivers should be designed to enable interrupts while performing long block moves, where possible. Devices or host adapters which require the processor to run with interrupts disabled for longer than a few hundred instructions probably are not good choices in a high-performance NetWare system.

D) No device driver routine should run longer than 250 millisecond with out relinquishing control to other processes, either by terminating or rescheduling using *CYieldIfNeeded* (v4.xx), *CRescheduleLast* (v3.1x), or *DelayMyself*.

Shared Interrupts

A driver may support shared interrupts if they are also supported by the processor bus and host adapters. Interrupts may not be shared on a bus unless the bus is operating in level-triggered mode. The Micro Channel Architecture Bus (MCA) always uses level-triggered interrupts, and can support shared interrupts provided the host adapters are also designed to support shared interrupts. The PC/AT bus normally uses edge-triggered interrupts, and thus will not support true shared interrupts. The EISA Bus can be set to operate in either edge-triggered or level-triggered mode, and could thus support shared interrupts (if the BIOS sets and the host adapters support level-triggered mode).

The driver initialize procedure must indicate that the adapters are sharing interrupts in the CFLAGS field of the adapter *IOConfigurationStructure*, and the share indicator must have been set when calling *SetHardwareInterrupt*. A driver ISR which supports shared interrupts is very similar to a driver ISR which does not support shared interrupts, but must additionally provide the following functions in the ISR code:

A) The driver shared interrupt ISR code must determine if any of its host adapters associated with the interrupt are requesting an interrupt. If none of the associated adapters are requesting an interrupt, the ISR must return immediately to the driver ISR caller (the OS ISR) with EAX non zero and the zero flag reset (non-zero status). This is accomplished by executing the following (do not call *CDoEndOfInterrupt*):

```
int Code = 0;          // define Code
                                          or      al, 01h     ;reset zero flag
return (Code | 1);     // reset zero flag, set return       ret                 ;return to OS ISR code
                       value
```

If one of the associated adapters is **currently** requesting an interrupt, the ISR should service the request. Upon completion of the ISR code, the driver ISR should return to the OS ISR (the driver ISR caller) with EAX zeroed and the zero flag set (zero status). This is accomplished by executing the following:

```
int Code = 0;          // define Code
                                          xor eax, eax    ;set zero flag, clear eax
return (Code | 0);     // set zero flag, clear return       ret                 ;return to OS ISR code
                       code
```

Make sure to call *CDoEndOfInterrupt* before returning to the calling ISR.

B) Additional ISR logic may be required to protect the driver if interrupts are enabled in the ISR code and subsequent interrupts occur on the same interrupt level.

NOTE: The OS ISR performs the same save and restore sequence for both shared and non-shared interrupts.

Interrupt Service Routine (ISR) - Sharable

The ISR services the HBA/controller interrupts.

Syntax LONG ISR( )

Return Values

0	The interrupt was serviced successfully. The zero flag must be set (zero status).
non-zero	The interrupt was not serviced. The zero flag must be reset (non-zero status).

Parameters None

Remarks **The name of the ISR is arbitrary**. The driver should register the ISR and pass its address to the OS using the *SetHardwareInterrupt* API. Interrupts are disabled upon entry and exit.

Interrupt Service Routine (ISR) - Non-Sharable

The ISR services the HBA/controller interrupts.

Syntax void ISR( )

Return Values None

Parameters None

Remarks **The name of the ISR is arbitrary**. The driver should register the ISR and pass its address to the OS using the *SetHardwareInterrupt* API. Interrupts are disabled upon entry and exit.

Driver ISR Algorithm

```
DriverISR()
{
    Request = I/O or IOCTL that caused the interrupt;

    if (the interrupt is spurious)
        return(0);

    else if (interrupt is an end of I/O with good completion)
    {
        Request->CompletionCode = 0;
        PutRequest(DiskHandle, Request);
    }
    else if (interrupt is an I/O request with error completion)
    {
        Request->CompletionCode = ERROR_CODE;
        PutRequest(DiskHandle, Request);
    }


    else if (interrupt was produced by an IOCTL with good completion)
    {
        Request->CompletionCode = 0;
        PutIOCTL(CardHandle, Request);
    }

    else if (interrupt was produced by an IOCTL with error completion)
    {
        Request->CompletionCode = ERROR_CODE;
        PutIOCTL(CardHandle, Request);
    }
}
```

Read After Write Verification

Read-After-Write Verify is a data integrity feature of NetWare which disk drivers may support. The OS has an overall Read-After-Write Verify Mode flag which indicates the default verify mode for all disks. When normal storage devices are registered with the OS, the system Read-After-Write Verify mode should be indicated as the current device Read-After-Write Verify mode. The current global system Read-After-Write Verify Status can be determined by making a call to the system support function ***GetReadAfterWriteVerifyStatus*** (see chapter 7).

Many sequential tape devices have read heads positioned immediately after the write heads, and include the logic necessary to verify data blocks as they are written (on the fly). Disk storage devices do not have this ability, and must wait for at least one rotation of the media to perform a verify operation. Read-After-Write verification may be performed by the drive, the drive controller or host adapter, or by the driver (software). Although grown media defects are not frequent, one of these forms of Read-After-Write Verification must be used to immediately detect loss of data due to a grown defect.

Drivers must support the Device Verify Mode IOCTL (see chapter 5), even if Read-After-Write Verification is not supported. This IOCTL allows the OS or an NLM to set the current device Read-After-Write Verify mode or to determine its current setting. If this IOCTL is used to set a new mode, the mode will override the default system mode initially set when the device was registered. If a driver or device does not support the new mode indicated in the IOCTL, the driver must return the actual mode the driver has substituted for the device.

It is also recommended that very limited error recovery be allowed on ReadAfterWriteVerify operations so that sporadic failures may be HotFixed.

NOTE: The nature of the cache implementation in the OS and the disk driver interface is such that it is possible for the OS to change the contents of a cache buffer after a write request has been taken by the driver using a *GetRequest*, since the write request buffer is actually the active cache buffer. It is possible that the driver may have already initiated the actual write operation when this modification takes place. This will not present an OS integrity problem since another write request will be issued after the buffer is modified. If the driver were to rely on the contents of the original cache buffer for Read-After-Write verify operations, however, the driver may get erroneous indications of verify failures. Novell recommends that the driver retry all requests that have failed verification by rewriting them to the device and then repeating the verification. Since the above-described scenario occurs infrequently, the repetition of the erroneously diagnosed requests will not cause any significant loss in system thoughput. As another option, the driver can copy the request buffer and retain the copy for verification comparison. However, this will cause a significant loss in system thoughput. It is generally faster for the host adapter or driver to accomplish verification in hardware (with write-verify, etc) where possible.

HotFix

HotFix is the OS procedure which provides dynamic bad-sector re-allocation and recovery for normal read/write disk devices (not tape, CD-ROM, nor WORM devices), where possible. The OS will HotFix all failing read or write requests.

HotFix only handles requests with MEDIA ERROR and CORRECTED MEDIA ERROR completion status. When a media error is detected, HotFix ignores the original failing request, generates multiple new single-sector requests to replace the original multi-sector request, then puts the multi-sector request back together before returning the completed request to the caller. If a single-sector request completes successfully, the block is not HotFixed, but if the single-sector request has bad status returned, the sector is HotFixed.

This allows HotFix to isolate the specific sector(s) which have a problem. The failing sectors are HotFixed by placing them in the BadBlockTable, assigning an alternate sector from the HotFix area to be substituted for the original bad sector in all subsequent I/O requests. The HotFix then retries the failed operation if it was a Write, so that the new alternate sector contains valid data (if the operation was a Read, HotFix attempts to obtain valid data from a mirrored device, after which the good data will be written to the new sector). In order for HotFix to work, a driver which supports HotFix must provide single 512 byte sector operations.

Sequential Device Support Considerations

Support of sequential devices requires consideration to several issues which are not necessary to address with drivers supporting random access devices only. These issues are discussed briefly below:

Streaming

Streaming sequential devices which are not kept streaming present a serious performance problem, since typical re-position times transform the average time to write a block from a few milliseconds to as much as several seconds. This can dramatically increase backup and restore times (also **decrease the life** of streaming devices, since they are manufactured with light-duty motors not designed to handle constant starting, stopping, and reversing directions, to allow the drives to be smaller and less expensive).

Keeping a device streaming requires architectural considerations in the design of the application, the I/O subsystem, and the driver. Streaming tape devices are designed so that tape motion continues at the same speed after reading or writing a block (gap sizes between blocks are generally more relaxed), assuming that another read or write will immediately follow. Start-stop sequential devices usually ramp down speed immediately after reading or writing a block (in the gap area). If the driver fails to issue another read or write command within the allowed time, a streaming drive will have gone long past the point on the media where it would have begun the next block. To recover the streaming device must ramp down the drive, re-position the media back to a position mid-way in the area where a normal gap would have been placed, then start the drive forward again while writing (or reading) the remainder of the gap, then writing (or reading) the actual data block. The keys to keeping a device streaming are:

A) The OS must not have long periods where interrupts are disabled, or where process level execution of the application is suspended.

B) The OS I/O subsystem must facilitate multiple queued I/O requests to the streaming device.

C) The driver must be designed to **always** have at least the next operation immediately available while the current operation is being completed. Also, upon completion of an operation, the next operation should be started prior to posting completion for the operation just completed. This helps insure that the next operation is initiated within the device re-instruct time limit so that the device will not time-out and reposition the media. The driver should also attempt to issue the next command to the device before allowing interrupts (allowing other events to delay it from issuing the necessary command to facilitate streaming). This implies that a drivers ISR should be very efficiently coded to avoid executing with interrupts disabled for any long period.

D) Applications must be designed to handle multiple outstanding requests. This means that the application must acquire new data to be written to the streaming sequential device while the streaming device is writing the previous block or blocks (backup), or must be saving data just read while the streaming device is reading a subsequent block (restore). If the data must be analyzed as it is read or written, more operations must be queued to compensate for the delays.

   Any frequent delays in obtaining data from or saving data to files must be compensated for by additional pre-processing and queuing of commands to the streaming device. Obviously, applications which meet this requirement **must not use blocking calls** to either the streamer or to the file system. Typically queuing 10 requests should keep a device streaming. This amount will vary depending on the size of the requests and the OS overhead required to input or output file data. Provided that enough requests are queued to keep the device streaming, no benefit is derived from having a large number of requests queued for the streaming device. An extremely large number of queued requests allocates more of the OS memory resource for an extended period, while not providing any increase in the speed of the streamer operation.

If all of the above requirements are met, the device should stream consistently.

Early Warning, End of Media, and Error handling

Special handling of unusual conditions are required by both the driver and calling applications when requests are queued to a sequential device. Logical problems arise if a driver is designed so that it reports exceptions to the caller and requires the caller to take remedial actions or requests, since additional requests are typically queued preceding any remedial requests the caller may issue. The driver must refuse (post requests with status of 0E09h) all further requests until the queued requests are all flushed (the caller must also keep track of the unsatisfied requests so that they may be issued again). After all requests have been flushed the caller must request the driver to reset its indicator (by calling with a ResetEndOfMediaStatus function) which it set indicating that future requests must be refused. At this point the caller can issue requests to perform any required remedial action. Upon completion, the caller may again begin to queue multiple requests for the device.

A driver that detects an Early Warning, End of Media, or End of Partition status during an operation should finish the operation, then post successful completion, indicating that an Early Warning device exception has occurred. At this time the driver should set an internal flag which it should check before accepting any further requests. The driver should refuse any further requests and return outstanding requests as "not completed" (by calling PutRequest indicating status of 0E09h, representing a non-fatal error caused by a prior driver state) when they are examined prior to service and prior to setting the internal flag.

Media Error Handling

Sequential devices must be called from the Device level of the Media Manager Interface for versions 3.12 and 4.xx. The Device Applications Interface (DAI) may still be in use for some developers writing to v3.11 and preceding versions. This eliminates any HotFix support or other normal system error handling. The driver, host adapter/controller, and the device are responsible to perform any retries for media errors, including any needed erase gap or media motion operations. Either the driver or the device **must** perform retries. Since many applications do not have any way of recovering from sequential media errors, **the driver assumes complete responsibility for reasonable error recovery** operation.