Chapter 7: NetWare Driver Support Routines

This chapter describes the following NetWare v3.1x and v4.xx support routines that are available to file server device drivers. The routines marked as 'NetWare v3.1x Only' are emmulated in NetWare v4.xx but will be eliminated in succeeding versions. The routines marked as 'NetWare v4.xx Only' are not available in NetWare versions 3.1x.

- AddDiskDevice
- AddDiskSystem
- AlertDevice
- Alloc
- AllocateResourceTag
- AllocBufferBelow16Meg
- * AllocSemiPermMemory
- CAdjustRealModeInterruptMask
- CancelNoSleepAESProcessEvent
- CancelSleepAESProcessEvent
- CCheckHardwareInterrupt
- CDisableHardwareInterrupt
- CDoEndOfInterrupt
- CEnableHardwareInterrupt
- CheckDiskCard
- * CheckDiskDevice
- ClearHardwareInterrupt
- CPSemaphore
- * CRescheduleLast
- CUnAdjustRealModeInterruptMask
- CVSemaphore
- ** CYieldIfNeeded
- ** CYieldWithDelay
- DelayMyself
- DeleteDiskDevice
- DeleteDiskSystem
- DeRegisterHardwareOptions
- DoRealModeInterrupt
- EnterDebugger
- Free
- FreeBufferBelow16Meg
- * FreeSemiPermMemory
- GetCurrentTime
- GetHardwareBusType
- GetIOCTL
- GetReadAfterWriteVerifyStatus
- GetRealModeWorkSpace
- GetRequest
- GetSectorsPerCacheBuffer
- MapAbsoluteAddressToCodeOffset
- MapAbsoluteAddressToDataOffset
- MapCodeOffsetToAbsoluteAddress
- MapDataOffsetToAbsoluteAddress
- ** NetWareAlert
- OutputToScreen
- ParseDriverParameters
- PutIOCTL
- PutRequest
- * QueueSystemAlert
- ** ReadPhysicalMemory
- RegisterForEventNotification
- RegisterHardwareOptions
- RemoveDiskDevice
- ScheduleNoSleepAESProcessEvent
- ScheduleSleepAESProcessEvent
- SetHardwareInterrupt
- UnRegisterEventNotification

* NetWare v3.1x Only
** NetWare v4.xx Only

Definitions:

The following API descriptions contain important terms that must be understood to design a driver to work properly with NetWare. Please note the following descriptive terms:

Blocking	-	Indicates the routine may cause the current thread of execution (NetWare process) to be suspended or "blocked" until the requested function is completed (or calls other blocking system routines). At no time can a driver Interrupt Service Routine (ISR) make a call to a blocking routine.
Non-blocking	-	Indicates the routine will return immediately, without causing the current thread or process to be suspended.
Interrupts Disabled	-	Indicates that interrupts must be disabled before calling the routine. This means that no processor interrupts excepting Non-maskable interrupts can occur. This state is often required to maintain system and driver integrity.
Process Level	-	Indicates the level of execution of NetWare v3.1x/v4.xx processes or scheduled tasks. NLMs normally execute at process level. Also, the loader and command processor execute at process level.
Interrupt Level	-	Indicates execution caused by a processor interrupt, in which case the current OS process is unknown. The ISR executes as the current process, and must never make blocking calls, etc.

Please note the following guidelines:

- All routines shown as "blocking" may only be called from blocking process level.
- All routines shown as "non-blocking" may be called from both blocking and non-blocking levels (see chapter 1).
- Other required calling environments are indicated in the **Requirements:** entry for each routine.
- The v3.1x, v3.1x & v4.xx or v4.xx designation indicates the Netware version in which the API is supported.

AddDiskDevice

(Blocking) v3.1x & v4.xx

Allocates DiskStructure and registers device with OS

Syntax:

```
DiskStruct *AddDiskDevice(
        BYTE *DeviceName,
        void (*IOPollRoutine)(
            DiskStruct *DiskHandle, IORequestStruct *IORequest),
        LONG TotalSize,
        LONG DriveSizes,
        LONG DriveParameters,
        LONG DriveID,
        CardStruct *CardHandle,
        LONG DiskStructureSize);
```

Return Value: Returns a handle to a DiskStructure, or 0 if unsuccessful

Requirements: Must be called from blocking process level only.

Parameters:

DeviceName — Pointer to a 32-byte ASCII string; byte 0 = length, bytes 1-31 = name of device which describes the physical device. (Exclude the length byte and the NULL character from the string length count.)

IOPollRoutine — Pointer to the driver's IOPoll routine for the device. The device driver must be able to receive a call to the IOPoll routine at any time upon exit from the *AddDiskDevice* routine.

TotalSize — The useable sector capacity of the physical device or media in the device. (The sector size is as reported in the **SectorSize** field.) For writeable media this value should be rounded down to a cylinder boundary (using the device geometry as reported below), since all partitions must begin and end on cylinder boundaries. For read-only media (CDROM) this value should be reported with no modifications. For sequencial access devices, if the capacity is unknown, this field should be set to a -2.

DriveSizes — Information about the drive size. It includes the following bytes:

```
db    AccessFlags    (lsb)
db    DriveType
db    BlockSize
db    SectorSize     (msb)
```

AddDiskDevice (continued)

AccessFlags indicates special device or access characteristics to be used with the device:

RemovableDevice	01h
ReadOnlyDevice	02h
WriteSequential	04h
ChangerDevice	10h *
MagazineDevice	20h *

* v3.12 & v4.xx only

RemovableDevice indicates that device media may be removed and replaced with other media. Device characteristics may be changed by insertion of new media, such as BlockSize, SectorCount, HeadCount, and CylinderCount, as well as other AccessFlags. The RemovableDevice access flag may not be changed after a device has been registered with the OS.

ReadOnlyDevice indicates to the OS that write operations should not be issued to the device. A valid Netware volume may be written, dismounted, registered as write-protected, then mounted again.

Write Sequential indicates to the OS that I/O requests to the device should be sent in sequential order.

The **ChangerDevice** access flag indicates that a Read/Write device associated with an autochanger is being added to the system. If this flag is set, the NetWare 4.xx or 3.12 OS will subsequently issue the appropriate IOCTLs in order to obtain the autochanger configuration.

The **MagazineDevice** access flag indicates that a Read/Write device associated with a magazine is being added to the system. If this flag is set, the NetWare 4.xx or 3.12 OS will subsequently issue the appropriate IOCTLs in order to obtain the magazine configuration.

AddDiskDevice (continued)

The **DriveType** is defined as follows:

0 Hard Disk
1 CD-ROM Device *
2 WORM Device *
3 Tape Device *
4 Magneto-Optical (MO) Device

* NetWare volumes are not **currently** supported on these device types. The types are provided to allow application software means to identify these devices and exploit their function.

BlockSize is the driver maximum I/O request size:

0 - 1 sector	4 - 16 sectors
1 - 2 sectors	5 - 32 sectors
2 - 4 sectors	6 - 64 sectors
3 - 8 sectors	7 - 128 sectors

SectorSize: The value inserted for **SectorSize** is actually a shift factor. The shift factor is used as the exponent in the following formula:

$512 * 2^{(sectorSize)}$ = Actual Sector Size

where **SectorSize** >= 0. *There must be a value declared for SectorSize*. Currently, this must be a value of 0 which calculates to a sector size of 512. The NetWare File System only supports a sector size of 512 bytes. All requests generated by the NetWare File System will be in sectors of that size. Drivers that support devices with native sector sizes other than 512 are required to translate these requests into the proper format.

AddDiskDevice (continued)

DriveParameters — Includes the following drive parameter fields (ignored for devices indicated as removable):

```
db    SectorCount     (lsb)
db    HeadCount
dw    CylinderCount   (msw)
```

SectorCount is the number of sectors per track on the device. **HeadCount** is the number of heads on the device.
CylinderCount is the number of cylinders on the device.
For writeable media the SectorCount and HeadCount parameters are used by the partition editor to determine the partition boundaries and are required to match the geometry of other partitions on the drive. For read-only media, if the device capacity does not fall on a cylinder boundary, the count should incremented to include the partial cylinder. (See TotalSize.)

DriveID — Drive identification. It includes the following fields:

```
db    ControllerNumber   (lsb)
db    DriveNumber
db    CardNumber
db    DriverID           (msb)
```

ControllerNumber is the device target address (SCSI id.) or equivalent.
DriveNumber is the device Logical Unit Number (LUN) or equivalent. If the ControllerNumber and DriveNumber reference the same object (i.e. SCSI devices with integrated drive electronics) this number is zero.
CardNumber is the host adapter card number. This number is optionally assigned by the system administrator and is passed to the driver at load time though a command line parameter (CARD=xx).
DriverID is the Novell-assigned driver number (obtained through Novell Labs IMSP.)

CardHandle — The card handle AddDiskSystem returned for the adapter on which the device resides.

DiskStructureSize — Size of the required device structure AddDiskDevice will allocate and zero fill. AddDiskDevice returns a pointer to this structure. This structure must be allocated even if the size is specified as 0 bytes, as the pointer is required for many calls.

AddDiskDevice (continued)

Example:

```
push    SIZE DiskStruct         ;allocate a disk structure
push    CardHandle              ;card handle
push    DriveId                 ;
push    DriveParameters         ;
push    DriveSizes              ;
push    TotalSize               ;
push    OFFSET IOPollRoutine    ;IOPoll entry point
push    OFFSET DeviceName       ;description text for device
call    AddDiskDevice           ;register with the OS
lea     esp, [esp + (8*4)]      ;adjust stack ptr
```

Description: AddDiskDevice creates a system device structure to provide NetWare information for the device specified. AddDiskDevice is called by the driver to register each un-registered device found during the driver's ScanForDevices procedure (devices which support removable media must be registered by the driver even if no media is currently present, as the device thus defined will not be active when it fails a subsequent mount request. The device may be activated later when media is present).

AddDiskDevice allocates and returns a pointer to a DiskStructure for driver use (driver determined size). The pointer serves both as a device handle for calls to AlertDevice, RemoveDiskDevice, DeleteDiskDevice, GetRequest, and PutRequest routines, and as a pointer to reference the DiskStructure.

See Also: AlertDevice, DeleteDiskDevice, RemoveDiskDevice, ScanForDevices, ReturnDeviceStatus IOCTL, I/O Function Codes

AddDiskSystem (Blocking) v3.1x & v4.xx

Allocates Card Structure and registers adapter with OS

Syntax:

```
CardStruct *AddDiskSystem(
        LONG NLMHandle,
        IOConfigStruct *IOConfig,
        void (*IOCTLPollRoutine)(
            CardStruct *CardHandle, IOCTLRequestStruct *IOCTLRequest),
        void (*ScanForDevices)(CardStruct *CardHandle),
        void (DeleteDevice)(DiskStruct *DiskHandle),
        LONG NovellNumber,
        LONG DriverResourceTag,
        LONG CardStructureSize);
```

Return Value: Returns a pointer to a Card structure, or 0 if unsuccessful

Requirements: Must be called from blocking process level only.

Parameters:

NLMHandle	The handle NetWare passed on the stack to the driver initialization routine.
IOConfig	The corresponding adapter board's IOConfiguration structure pointer.
IOCTLPollRoutine	The driver's IOCTL Poll routine entry point. The device driver must be able to receive a call to the IOCTLPoll routine at any time upon exit from the *AddDiskDevice* routine.
ScanForDevices	The driver's ScanForDevices routine entry point. The device driver must be able to receive a call to the ScanForDevices routine at any time upon exit from the *AddDiskDevice* routine.
DeleteDevice	v3.11 only - The entry point to the driver's DeleteDevice routine. For all other versions (v3.12 and v4.xx), this parameter should be initialized to a NULL (0).
NovellNumber	The number assigned for this driver by Novell.
DriverResourceTag	Resource tag allocated by driver with the "Driver Signature".
CardStructureSize	Driver-defined Card structure size, to be allocated by AddDiskSystem (zero not used by driver).

AddDiskSystem (continued)

Example:

```
push    SIZE CardStruct                 ;structure size to allocate
push    DriverResourceTag               ;identify owner of this resource
push    NovellNumber                    ;Novell assigned driver number
push    0                               ;Reserved0
push    OFFSET ScanForDevices           ;driver scan/add routine
push    OFFSET IOCTLPollRoutine         ;driver's IOCTL entry point
push    OFFSET IOConfig                 ;handle to IOConfiguration structure
push    NLMHandle                       ;passed at driver initialization.
call    AddDiskSystem                   ;register card with OS
lea     esp, [esp + (8*4)]              ;adjust stack pointer
```

Description: A device driver's Initialization routine calls this routine to register an adapter board with NetWare. AddDiskSystem creates a structure inside the NetWare Operating System to retain information about the specified adapter board. AddDiskSystem also allocates memory for a driver-defined local Card structure and passes a pointer back to the driver.

The pointer value serves two purposes. First, the driver uses the pointer as a card handle when calling CheckDiskCard, GetIOCTL, and PutIOCTL, AddDiskDevice, and DeleteDiskSystem. Second, the pointer is used to reference the card structure, which AddDiskSystem created, where the driver may store data for the corresponding adapter card.

See Also: DriverInitialization, DriverCheck, DriverUnload, DeleteDiskSystem, CheckDiskCard, DeleteDevice, ScanForDevices, ReturnDeviceStatus IOCTL

AlertDevice

(Non-blocking) v3.1x & v4.xx

Notifies Operating System of a device condition change

Syntax:

```
void AlertDevice(
        DiskStruct *DiskHandle,
        LONG MessageBit);
```

Return Value: None

Requirements: Interrupts disabled.

Parameters: DiskHandle Handle returned by AddDiskDevice for device.

MessageBit A **single** bit value indicating the device condition or cause of the AlertDevice call, defined as follows:

	hex	binary	
	01	0000 0001	**Device Failed** - a device has failed and is no longer active. The OS will deactivate the device, clear all pending I/O requests it owns and issue a deactivate IOCTL call.
	08	0000 1000	**Media Ejected** - media not present in the device (for removables). The OS will deactivate the device, clear all pending I/O requests it owns and issue a deactivate IOCTL call.
	20	0010 0000	**Media Inserted** - informs the OS that media has been inserted in the device. The OS will send a message to all applications that have locked the device.
*	40	0100 0000	**Delete Device** - requests the device be deleted. The OS will deactivate the device, clear all pending I/O requests it owns and calls the card's DeleteDevice routine.

* v3.1x only

AlertDevice (continued)

Example:

```
push    00000001b           ;indicate device failure
push    DiskHandle          ;device handle from AddDiskDevice call
call    AlertDevice         ;tell system about device status change
lea     esp, [esp + (2*4)]  ;adjust stack pointer
```

Description: This call notifies the OS of a status change or problem with a device. In the cases when the OS responds by deactivating the device, the driver is required to post completion for any outstanding requests for the device. All requests acquired with a GetRequest call must be returned to the OS with a *Device Not Active* completion code.

See Also: DeleteDiskDevice, RemoveDiskDevice

Alloc

(Non-blocking) v3.1x & v4.xx

Allocates block of returnable memory for driver use

Syntax:

```
void *Alloc(
        LONG NumberOfBytes,
        LONG MemRTag);
```

Return Value: Pointer to the allocated memory in EAX, or 0 if unsuccessful.

Requirements: Interrupts disabled.

Parameters:

NumberOfBytes	Passes in the amount of memory in bytes to be allocated.
MemRTag	Resource tag acquired by driver for memory allocation using an "AllocSignature" resource signature.

Example:

```
push    MemRTag                 ;identify type of resource
push    NumberOfBytes           ;indicate amount of memory required
call    Alloc                   ;returns pointer to memory in eax
lea     esp, [esp + (2*4)]      ;adjust stack pointer
mov     ebp, eax                ;need for use and to return
```

Description: Alloc is used to allocate memory for any driver requirements such as IOConfiguration structures or special buffers. Alloc is passed the amount of memory to allocate and returns a pointer to the allocated memory in the EAX register. This routine is available to drivers for Initialize Driver, Mass Storage Control Interface, IOPoll, and IOCTLPoll routines. It may also be called from within an interrupt environment (ISR); however, the availability of memory will be diminished. The memory allocated is not initialized by the allocation routine, and must be initialized by the driver. The repeated allocation and deallocation of relatively small blocks of memory will tend to cause memory fragmentation. For increased system efficiency, a large block of memory can be initially allocated and maintained as a pool of smaller blocks. **Memory is always allocated on a paragraph (16 byte) boundary.**

See Also: Free, AllocateResourceTag

AllocateResourceTag (Blocking) v3.1x & v4.xx

Allocates OS resource tags for specific resource types

Syntax:

```
LONG  AllocateResourceTag(
            LONG NLMHandle,
            void *ResourceDescString,
            LONG ResourceSignature);
```

Return Value: Resource tag identifying specified entry type (0 if error).

Requirements: Must be called from blocking process level only.

Parameters:

DriverHandle — The module handle passed to the driver (NLM) when its initialization routine was called.

ResourceDescString — Pointer to a null-terminated text string describing the resource, with a maximum total length of 16 bytes, including null terminator.

Example: db 'NDCB Driver',0

ResourceSignature — A value used to identify a specific resource type. The signatures the driver must pass (indicates to the OS the kind of resource tag to allocate, consequently do not change the following equates or the OS will fail the drivers request to allocate a resource tag) to identify each resource tag type requested are defined as follows:

```
    AESProcessSignature                 equ  50534541h
    AllocSignature                      equ  54524C41h
    CacheBelow16MegMemorySignature      equ  36314243h
    EventSignature                      equ  544E5645h
    DiskDriverSignature                 equ  4B534444h
    InterruptSignature                  equ  50544E49h
    IORegistrationSignature             equ  53524F49h
*   SemiPermMemorySignature         equ  454D5053h
    TimerSignature                      equ  524D4954h
```

* v3.1x only

AllocateResourceTag (continued)

Example:

```
    cmp     LoadedOnceGoodFlag, 0       ;already allocated tags ?
    jne     GotTags                     ;yes - skip
    push    DriverSignature             ;identifies Driver resource type
    push    OFFSET rTagString           ;resource tag descriptive string
    push    NLMHandle                   ;driver module id
    call    AllocateResourceTag         ;returns a tag id in EAX
    lea     esp, [esp + (3*4)]          ;adjust stack pointer
    mov     DrvrRTag, eax               ;save our driver resource tag
    push    IOSignature                 ;identifies I/O device resource type
    push    OFFSET IORTagString         ;resource tag descriptive string
    push    NLMHandle                   ;driver module id
    call    AllocateResourceTag         ;returns a tag id in EAX
    lea     esp, [esp + (3*4)]          ;adjust stack pointer
    mov     IORtag, eax                 ;save for RegisterHardwareOptions use
    push    IntSignature                ;identifies Interrupt resource type
    push    OFFSET IntRTagString        ;resource tag descriptive string
    push    NLMHandle                   ;driver module id
    call    AllocateResourceTag         ;returns a tag id in EAX
    lea     esp, [esp + (3*4)]          ;adjust stack pointer
    mov     IntRTag, eax                ;save for SetHardwareInterrupt use
    push    MemSignature                ;identifies Memory resource type
    push    OFFSET MemRTagString        ;resource tag descriptive string
    push    NLMHandle                   ;driver module id
    call    AllocateResourceTag         ;returns a tag id in EAX
    lea     esp, [esp + (3*4)]          ;adjust stack pointer
    mov     MemRTag, eax                ;save for Alloc use
    push    MemoryBelow16MegSignature   ;identifies special memory resource tag
    push    OFFSET MemBelow16RTag       ;resource tag descriptive string
    push    NLMHandle                   ;driver module id
    call    AllocateResourceTag         ;returns a tag id in EAX
    lea     esp, [esp + (3*4)]          ;adjust stack pointer
    mov     MemBL16RTag, eax            ;save resource tag for allocate and free calls
    push    AESSignature                ;identifies AES timer resource type
    push    OFFSET AESRTagString        ;resource tag descriptive string
    push    NLMHandle                   ;driver module id
    call    AllocateResourceTag         ;returns a tag id in EAX
    lea     esp, [esp + (3*4)]          ;adjust stack pointer
    mov     AESRTag, eax                ;save for later references
    push    TmrSignature                ;identifies timer resource type
    push    OFFSET TmrRTagString        ;resource tag descriptive string
    push    moduleHandle                ;driver module id
    call    AllocateResourceTag         ;returns a tag id in EAX
    lea     esp, [esp + (3*4)]          ;adjust stack pointer
    mov     TmrTag, eax                 ;save for later reference
    mov     LoadedOnceGoodFlag,1        ;indicate done once
GotTags:
```

Description: Acquires a tracking identifier required by certain OS calls to track system resources (and recover them from NLM or Driver failure). The driver **must acquire a tag for each different type** of resource to be allocated.

See Also: Driver Initialization, Driver Unload

AllocBufferBelow16Meg (Blocking) v3.1x & v4.xx

Allocates block of returnable memory below the 16 megabyte boundary for driver use.

Syntax:

```
void *AllocBufferBelow16Meg(
        LONG RequestedSize
        LONG *ActualSize,
        LONG MemBelow16RTag);
```

Return Value: Pointer to the allocated memory in EAX, or 0 if unsuccessful.

Requirements: Interrupts disabled.

Parameters:

RequestedSize	Number or contiguous bytes requested
ActualSize	Receives the actual number of bytes allocated in the location pointed to by this parameter
MemBelow16RTag	Resource tag acquired by driver for memory allocation (with a "CacheBelow16MegMemorySignature")

Example:

```
push    MemBelow16RTag          ;identifies type of resource
push    OFFSET ActualSize       ;amount of memory acquired returned here
push    RequestedSize           ;number of bytes required supplied here
call    AllocBufferBelow16Meg   ;returns pointer to memory in eax
lea     esp, [esp + (3*4)]      ;adjust stack pointer
mov     ebp, eax                ;need for use and to return
```

Description: Use AllocBufferBelow16Meg **only** to allocate memory for drivers supporting 16-bit host adapters **in machines with more than 16 megabytes of memory** to allow the driver to do I/O operations to or from intermediate buffers below 16 megabytes, moving the data to or from the actual request buffer when above the 16 megabyte boundary. The memory returned will be one or more contiguous cache buffers. The pointer to the buffer allocated is returned in EAX (zero if none allocated). Drivers **must** call Alloc for **all** other memory allocation requirements. Memory is not initialized to zero. See Appendix G for implementation details. The repeated allocation and deallocation of relatively small blocks of memory will tend to cause memory fragmentation. For increased system efficiency, a large block of memory can be initially allocated and maintained as a pool of smaller blocks. **Memory is always allocated on a paragraph (16 byte) boundary.**

See Also: FreeBufferBelow16Meg, AllocateResourceTag

AllocSemiPermMemory (Non-blocking) v3.1x

Allocates block of returnable memory for driver use

Syntax:
```
void *AllocSemiPermMemory(
        LONG NumberOfBytes,
        LONG MemRTag);
```

Return Value: Pointer to the allocated memory in EAX, or 0 if unsuccessful.

Requirements: Interrupts disabled. May not be called from interrupt level.

Parameters:

NumberOfBytes	Passes in the amount of memory in bytes to be allocated.
MemRTag	Resource tag acquired by driver for memory allocation using an "SemiPermMemorySignature" resource signature.

Example:

```
push    MemRTag                 ;identify type of resource
push    NumberOfBytes           ;indicate amount of memory required
call    AllocSemiPermMemory     ;returns pointer to memory in eax
lea     esp, [esp + (2*4)]      ;adjust stack pointer
mov     ebp, eax                ;need for use and to return
```

Description: AllocSemiPermMemory is used to allocate memory for any driver requirements such as IOConfiguration structures or special buffers. AllocSemiPermMemory is passed the amount of memory to allocate and returns a pointer to the allocated memory in the EAX register. This routine is available to drivers for Initialize Driver, Mass Storage Control Interface, IOPoll, and IOCTLPoll routines, but may not be called from interrupt-level. The memory allocated is not initialized by the allocation routine, and must be initialized by the driver. This API will not be supported in future products and is only emulated in NetWare 4.xx. It should be replaced with the "Alloc" API. The repeated allocation and deallocation of relatively small blocks of memory will tend to cause memory fragmentation. For increased system efficiency, a large block of memory can be initially allocated and maintained as a pool of smaller blocks. **Memory is always allocated on a paragraph (16 byte) boundary.**

See Also: Alloc, Free, FreeSemiPermMemory, AllocateResourceTag

CAdjustRealModeInterruptMask (Non-blocking) v3.1x & v4.xx

Adjusts Real Mode interrupt mask for calls to DOS driver

Syntax: void CAdjustRealModeInterruptMask(
LONG IRQNumber);

Return Value: None

Requirements: Interrupts disabled.

Parameters: IRQNumber Interrupt (IRQ) Number utilized by the associated card.

Example:

```
push    IRQNumber                   ;tell OS which interrupt bit to unmask
call    CAdjustRealModeInterruptMask ;w/DOS for Real mode switch
lea     esp, [esp + 4]              ;adjust stack
```

Description: This call clears the corresponding bit in the RealModeInterruptMask. (The bit was set by a SetHardwareInterrupt call.) This mask is written to the priority interrupt controllers (PICs) when a NetWare call is made to return the processor to real mode (in order to make DOS calls.) This has the effect of unmasking the interrupt for use in real mode. Drivers that support adapter/devices also supported by DOS in conjunction with DOS drivers should make this call immediately after the SetHardwareInterrupt call. (Note: The loader uses DOS drivers to load NLMs and drivers from DOS partitions).

See Also: SetHardwareInterrupt, ClearHardwareInterrupt, CUnAdjustRealModeInterruptMask

CancelNoSleepAESProcessEvent (Non-blocking) v3.1x & v4.xx

Cancels No-Sleep AES timer event

Syntax: void CancelNoSleepAESProcessEvent(
AESEventStruct *AESEvent);

Return Value: None

Requirements: Interrupts disabled.

Parameters: AESEvent Passes a pointer to an AES structure.

Example:

```
push    OFFSET AESEvent              ;address of AES structure
call    CancelNoSleepAESProcessEvent ;no further event callbacks
lea     esp, [esp + 4]               ;adjust stack pointer
```

Description: CancelNoSleepAESProcessEvent cancels the AES event indicated by the AES structure pointer it is passed. A Remove Driver procedure must make this call for every AES No-Sleep timer the driver has used.

See Also: Driver Initialization, Driver Unload, AESEventStructure, ScheduleNoSleepAESProcessEvent

CancelSleepAESProcessEvent (Non-blocking) v3.1x & v4.xx

Cancels Sleep AES timer event

Syntax: void CancelSleepAESProcessEvent(
AESEventStruct *AESEvent);

Return Value: None

Requirements: Interrupts disabled.

Parameters: AESEvent Passes a pointer to an AES structure.

Example:

```
push    OFFSET AESEvent             ;address of AES structure
call    CancelSleepAESProcessEvent  ;no further event callbacks
lea     esp, [esp + 4]              ;adjust stack pointer
```

Description: CancelSleepAESProcessEvent cancels the AES event indicated by the AES structure pointer it is passed. A Remove Driver procedure must make this call for every AES Sleep timer the driver has used.

See Also: Driver Initialization, Driver Unload, AESEventStructure, ScheduleSleepAESProcessEvent

CCheckHardwareInterrupt (Non-blocking) v3.1x & v4.xx

Returns indication of interrupt requested for specified interrupt

Syntax: LONG CCheckHardwareInterrupt(
LONG IRQNumber);

Return Value:

zero	No interrupt request active for IRQ Number
non-zero	Interrupt requested for IRQ Number

Requirements: Interrupts disabled.

Parameters: IRQNumber Interrupt to be checked for pending request.

Example:

```
push    IRQNumber                   ;interrupt number (0-15)
call    CCheckHardwareInterrupt     ;determine if active request
lea     esp, [esp + 4]              ;adjust stack pointer
```

Description: CCheckHardwareInterrupt determines if an interrupt request is currently being made to the priority interrupt controller (PIC) assigned to the indicated interrupt number. The PIC should normally have this IRQ masked off while this call is made. (The interrupt will not be recorded by the PIC). A return value of zero indicates that the PIC has no interrupt request being made to it.

See Also: CDisableHardwareInterrupt, CEnableHardwareInterrupt, CDoEndOfInterrupt

CDisableHardwareInterrupt

(Non-blocking) v3.1x & v4.xx

Masks off indicated IRQ in associated interrupt controller

Syntax: void CDisableHardwareInterrupt(
LONG IRQNumber);

Return Value: None

Requirements: Interrupts disabled.

Parameters: IRQNumber Specifies interrupt to be masked off.

Example:

```
push    IRQNumber                   ;desired interrupt (0-15)
call    CDisableHardwareInterrupts  ;no interrupts allowed (or recorded) from level
lea     esp, [esp + 4]              ;adjust stack pointer
```

Description: CDisableHardwareInterrupt causes the corresponding interrupt in the Programmable Interrupt Controller (PIC) to be masked off so that no further interrupts are allowed or recorded by the PIC.

See Also: CEnableHardwareInterrupts, CCheckHardwareInterrupt, CDoEndOfInterrupt

CDoEndOfInterrupt

(Non-blocking) v3.1x & v4.xx

Issues required EOIs for the specified interrupt

Syntax: void CDoEndOfInterrupt(
LONG IRQNumber);

Return Value: None

Requirements: Interrupts disabled.

Parameters: IRQNumber Indicates interrupt for which EOIs are to be issued.

Example:

```
push    IRQNumber               ;desired interrupt (0 - 15)
call    CDoEndOfInterrupt       ;issue required EOIs
lea     esp, [esp + 4]          ;adjust stack pointer
```

Description: Issues End of Interrupt (EOI) command to the associated interrupt controller for the IRQ indicated. If the IRQ is assigned to a secondary PIC, an EOI will be issued to the secondary PIC, followed by a short delay for the bus, then to the primary PIC. If the IRQ is assigned to a primary PIC, an EOI will be issued to the primary PIC only.

See Also: CCheckHardwareInterrupt, CDisableHardwareInterrupt, CEnableHardwareInterrupt

CEnableHardwareInterrupt

(Non-blocking) v3.1x & v4.xx

Enables specified IRQ in associated interrupt controller

Syntax: void CEnableHardwareInterrupt(
LONG IRQNumber);

Return Value: None

Requirements: Interrupts disabled.

Parameters: IRQNumber Indicates desired hardware interrupt

Example:

```
push    IRQNumber                ;hardware interrupt to be enabled
call    CEnableHardwareInterrupt ;unmask (enable) interrupt level
lea     esp, [esp + 4]           ;adjust stack pointer
```

Description: CEnableHardwareInterrupt un-masks (enables) the indicated interrupt in the associated programmable Interrupt Controller (PIC). This allows further interrupts to be recorded or to occur.

See Also: CDisableHardwareInterrupt, CCheckHardwareInterrupt, CDoEndOfInterrupt

CheckDiskCard

(Blocking) v3.1x & v4.xx

Returns composite lock status of all devices on adapter card.

Syntax:

```
LONG  CheckDiskCard(
            CardStruct *CardHandle,
            LONG ScreenHandle);
```

Return Value: Composite (logically OR'ed) status of all card devices, as follows:

0 no devices are locked
1 at least one device is locked but has a mirror associated with a separate driver
2 at least one device is locked and doesn't have a mirror associated with a separate driver
3 same as 2 (logical 'or' of 1 and 2)

Requirements: Must be called from blocking process level only.

Parameters:

CardHandle — The handle (pointer to the card structure) of the desired adapter board returned by the AddDiskSystem API.

ScreenHandle — The screen handle passed to the driver's Check Driver routine.

Example:

```
push    ScreenHandle        ;allow console messages
push    CardHandle          ;identify CardStructure
call    CheckDiskCard       ;see if any card devices locked
lea     esp, [esp + (2*4)]  ;adjust stack pointer
or      ccode, eax          ;combine results for driver check
```

Description: CheckDiskCard returns in the EAX register the combined status of the registered devices attached to adapter corresponding to the card handle (passed as a parameter to CheckDiskCard.) It also uses the screen handle to display the status of the devices that are locked. It is the responsibility of the driver's Check Driver routine to determine the status of all registered devices on each adapter card and return the combined (OR'ed) status.

Several NetWare commands call the driver's Check Driver routine as a precautionary measure to determine if any of the driver's registered devices are locked. For example, the console command UNLOAD calls a driver's Check Driver before unloading the driver.

See Also: CheckDriver, UnloadDriver

CheckDiskDevice (Blocking) v3.1x

Returns the lock status of the storage device.

Syntax:

```
LONG  CheckDiskCard(
            CardStruct *DiskHandle,
            LONG ScreenHandle);
```

Return Value: Returns one of the following codes indicating the device status:

0 device is not locked
1 device is locked but has a mirror associated with a separate driver
2 device is locked and doesn't have a mirror associated with a separate driver

Requirements: Must be called from blocking process level only.

Parameters:

DiskHandle Handle returned by AddDiskDevice for this device.

ScreenHandle The screen handle passed to the Check Driver routine.

Example:

```
push        ScreenHandle        ;allow console messages
push        DiskHandle          ;identify DiskStructure
call        CheckDiskDevice     ;see if device locked
lea         esp, [esp + (2*4)]  ;adjust stack pointer
or          ccode, eax          ;combine results for driver check
```

Description: CheckDiskDevice returns in the EAX register the status of the registered device corresponding to the device handle (passed as a parameter to CheckDiskDevice.) It also uses the screen handle to display the status of the devices that are locked. It is the responsibility of the driver's Check Driver routine to determine the status of all registered devices on each adapter card and return the combined (OR'ed) status. This API will not be supported in future products and is only emulated in NetWare 4.xx. It should be replaced with the "CheckDiskCard" API.

Several NetWare commands call the driver's Check Driver routine as a precautionary measure to determine if any of the driver's registered devices are locked. For example, the console command UNLOAD calls a driver's Check Driver before unloading the driver.

See Also: CheckDriver, UnloadDriver

ClearHardwareInterrupt (Non-blocking) v3.1x & v4.xx

Deallocates adapter card interrupt

Syntax: void ClearHardwareInterrupt(
LONG IRQNumber,
void (*InterruptService)()); or LONG (*InterruptService)());

Return Value: None

Requirements: Interrupts disabled. May not be called from interrupt level.

Parameters: IRQNumber Passes the IRQ number of the hardware interrupt.

InterruptService Pointer to the interrupt service routine (ISR) that was assigned to the specified interrupt. The service routine returns a value in a shared interrupt configuration.

Example:

```
push    InterruptService            ;ISR address for this card
push    IRQNumber                   ;interrupt number
call    ClearHardwareInterrupt
lea     esp, [esp + (2*4)]          ;adjust stack pointer
```

Description: ClearHardwareInterrupt releases a processor hardware interrupt previously allocated by SetHardwareInterrupt for an adapter board. It also masks off the interrupt at the priority interrupt controllers (PICs) and clears the corresponding bit in the RealModeInterruptMask. In the case of shared interrupts, the masking process is performed only if the specified ISR is the only one remaining in the chain. (The other ISRs have been cleared previously.) This call must be made by a driver's Remove Driver routine for each card for which a SetHardwareInterrupt call was made previously.

See Also: SetHardwareInterrupts, CAdjustHardwareInterruptMask, CUnAjustHardwareInterruptMask, Driver ISR

CPSemaphore

(Blocking) v3.1x & v4.xx

Set a Semaphore

Syntax: void CPSemaphore(LONG WorkSpaceSemaphore);

Return Value: None

Requirements: Must be called from blocking process level only.

Parameters: WorkSpaceSemaphore handle to the semaphore

Example:

```
push    WorkSpaceSemaphore   ;load semaphore
call    CPSemaphore          ;lock workspace for our use
add esp, (1 * 4)             ;restore stack
```

Description: *CPSemaphore* is used to lock the real mode workspace when making a BIOS call. This routine is called with interrupts disabled, and interrupts remain disabled.

For more information on how to use the BIOS call, refer to Appendix F.

Do not use this call to handle critical sections local to the driver.

See Also: CVSemaphore, GetRealModeWorkSpace, Appendix F

CRescheduleLast (Blocking) v3.1x

Places the current process last in active queue (delays)

Syntax: void CRescheduleLast(void);

Return Value: None

Requirements: Must be called from blocking process level only.

Parameters: None

Example:

```
call        CRescheduleLast
; will regain control undefined time later
```

Description: This routine places the current task last on the list of active tasks to be executed. This allows other tasks to be scheduled first, keeping OS processes functioning.

See Also: CYieldIfNeeded, CYieldWithDelay, DelayMyself, AllocateResourceTag

CUnAdjustRealModeInterruptMask (Non-blocking) v3.1x & v4.xx

Readjusts Real Mode Interrupt mask

Syntax: void CUnAdjustRealModeInterruptMask(
LONG IRQNumber);

Return Value: None

Requirements: Interrupts disabled,

Parameters: IRQNumber Interrupt Number utilized by the associated card.

Example:

```
push    InterruptNumber                 ;tell OS sharing interrupt
call    CUnAdjustRealModeInterruptMask  ;w/DOS for Real mode switch
lea     esp, [esp + 4]                  ;adjust stack
```

Description: This call sets the corresponding bit in the RealModeInterruptMask. This mask is written to the priority interrupt controllers (PICs) when a NetWare call is made to return the processor to real mode (in order to make DOS calls.) This has the effect of masking the interrupt in real mode.

See Also: SetHardwareInterrupt, ClearHardwareInterrupt, CAdjustRealModeInterruptMask

CVSemaphore (Non-Blocking) v3.1x & v4.xx

Clear a Semaphore

Syntax: void CVSemaphore(LONG WorkSpaceSemaphore);

Return Value: None

Requirements: None

Parameters: WorkSpaceSemaphore handle to the semaphore

Example:

```
push    WorkSpaceSemaphore      ;pass semaphore
call    CVSemaphore             ;unlock workspace
add     esp, (1 * 4)            ;restore stack
```

Description: *CVSemaphore* clears a semaphore that was set with *CPSemaphore*. This routine returns with interrupts enabled.

Normally, *CVSemaphore* is used when the driver has finished making an EISA BIOS call so that other processes can be allowed to use the workspace (Refer to Appendix G).

See Also: CPSemaphore, Appendix F

CYieldIfNeeded (Blocking) v4.xx

Places the current process last in the run queue if other work is pending

Syntax: void CYieldIfNeeded(void);

Return Value: None

Requirements: Must be called from blocking process level only.

Parameters: None

Example:

```
call        CYieldIfNeeded      ; will regain control undefined time later if other processes
                                require run time.  Otherwise continue processing.
```

Description: This routine places the current task last on the list of active tasks to be executed only if other non-low priority tasks require run time. This increases system efficiency by not disrupting the current process until actually necessary; however, low priority threads are disabled until the process runs to completion or releases control using the *CYieldWithDelay* API.

See Also: CYieldWithDelay, CRescheduleLast, DelayMyself, AllocateResourceTag

CYieldWithDelay

(Blocking)
v4.xx

Places the current process last in the run queue (delays)

Syntax: void CYieldWithDelay(void);

Return Value: None

Requirements: Must be called from blocking process level only.

Parameters: None

Example:

```
call        CYieldWithDelay      ; will regain control undefined time later
```

Description: This routine places the current task last on the list of active tasks to be executed. This allows other tasks to be scheduled, keeping OS processes fuctioning.

See Also: CYieldIfNeeded, CRescheduleLast, DelayMyself, AllocateResourceTag

DelayMyself (Blocking) v3.1x & v4.xx

Delays current process for clock ticks specified

Syntax:

```
void  DelayMyself(
        LONG ClockTicks,
        LONG TimerResourceTag);
```

Return Value: None

Requirements: Must be called from blocking process-level only.

Parameters:

ClockTicks — Value indicating number of 1/18th second clock ticks to put this process to sleep (minimum time before return).

TimerResourceTag — Timer resource tag given to timer category when driver allocated resource tags during initialization.

Example:

```
push        TimerResourceTag     ;identify this driver
push        ClockTicks           ;time to sleep
call        DelayMyself          ;delay # ticks indicated
lea         esp, [esp + (2*4)]   ;adjust stack pointer
```

Description: Puts current running process (caller) to sleep for the designated time. Return is made following expiration of the specified number of ticks. This routine is called to prevent a process from dominating process resources and preventing other vital processes from running. It also provides a specific minimum delay before the process is re-awakened, which may be helpful for tasks where some function will not complete for at least a specified period.

See Also: CRescheduleLast, AllocateResourceTag

DeleteDiskDevice

(Blocking) v3.1x & v4.xx

Removes a device structure (DiskStructure) from OS

Syntax: void DeleteDiskDevice(
DiskStruct *DiskHandle);

Return Value: None

Requirements: Must be called from blocking process level only.

Parameters: DiskHandle Passes a handle for the target device. This is the same value returned by AddDiskDevice.

Example:

```
push    eax                 ;push device handle on stack
call    DeleteDiskDevice    ;remove the structure
lea     esp, [esp + 4]      ;adjust stack pointer
```

Description: DeleteDiskDevice completes the removal of a device. This routine must be called after RemoveDiskDevice. DeleteDiskDevice returns to NetWare the memory allocated for a device handle structure (DiskStructure) by passing the handle of the device to be deleted.

See Also: RemoveDiskDevice

DeleteDiskSystem (Blocking) v3.1x & v4.xx

Removes a Card Structure from the OS

Syntax:

```
void DeleteDiskSystem(
        CardStruct *CardHandle,
        LONG Status);
```

Return Value: None

Requirements: Must be called from blocking process level only.

Parameters:

CardHandle — Passes a handle for the card structure for the associated adapter board. AddDiskSystem returned this handle for the driver.

Status — This parameter is included in the NetWare 3.1x and 4.xx versions for capatibility reasons only. It should be **initialized to a two (2)**.

Example:

```
push    2
push    eax                     ;push CardHandle on stack
call    DeleteDiskSystem
lea     esp, [esp + (2*4)]      ;adjust stack pointer
```

Description: DeleteDiskSystem deletes a mass storage adapter board from NetWare. A driver calls this routine. DeleteDiskSystem destroys the Card Structure that AddDiskSystem created to correspond to the specified adapter board. Once DeleteDiskSystem returns, NetWare no longer knows about the specified adapter board. After DeleteDiskSystem returns, **do not** reference the memory once allocated for the AddDiskSystem call.

See Also: AddDiskSystem

DeRegisterHardwareOptions (Blocking) v3.1x & v4.xx

Releases hardware options reserved previously

Syntax: void DeRegisterHardwareOptions(
IOConfigStruct *IOConfig);

Return Value: None

Requirements: Interrupts disabled. Must be called from blocking process level only.

Parameters:

IOConfig	Passes a pointer to the adapter board's corresponding IOConfiguration structure.

Example:

```
push    eax                         ;pass IOConfig structure ptr
call    DeRegisterHardwareOptions
lea     esp, [esp + 4]              ;adjust stack pointer
```

Description: DeRegisterHardwareOptions removes previously reserved hardware options for a particular adapter board. A driver's Remove Driver routine calls this routine. DeRegisterHardwareOptions removes the hardware options specified in a adapter board's I/O Configuration structure.

See Also: RegisterHardwareOptions, ParseDriverParameters

DoRealModeInterrupt (Blocking) v3.1x & v4.xx

Perform a Dos Interrupt call

Syntax:

```
LONG DoRealModeInterrupt(
        InputParamStruct *InputParameters,
        OutputParamStruct *OutputParameters);
```

Return Value: EAX contains:

0 Successful; sets the zero flag if the interrupt vector is called
1 Fail; clears the zero flag if the interrupt vector is no longer available because DOS has been removed

Requirements: The input parameter structure must already be initialized. Must be called from blocking process level only.

Parameters:

InputParameters	pointer to a filled in InputParameterStructure that is defined below
OutputParameters	pointer to a filled in OutputParameterStructure that is defined below

Example:

```
push    OFFSET OutputParameters
push    OFFSET InputParameters
call    DoRealModeInterrupt

add     esp, 2 * 4
cmp     eax, 0
jne     IntNotValidErrorExit
```

DoRealModeInterrupt (continued)

Description: *DoRealModeInterrupt* is used to perform real mode interrupts, such as BIOS and DOS interrupts. This routine can only be called at process time, and it may enable interrupts and put the calling process to sleep.

EISA boards will need to use *DoRealModeInterrupt* to perform the INT 15h BIOS call that returns the board configuration (Refer to Appendix F). The parameter structures are defined below:

InputParameters

```
InputParamStruct    struc
    IAXRegister         dw  ?
    IBXRegister         dw  ?
    ICXRegister         dw  ?
    IDXRegister         dw  ?
    IBPRegister         dw  ?
    ISIRegister         dw  ?
    IDIRegister         dw  ?
    IDSRegister         dw  ?
    IESRegister         dw  ?
    IIntNumber          dw  ?
InputParamStruct    ends
```

```
typedef struct InputParameterStructure {
    WORD IAXRegister;
    WORD IBXRegister;
    WORD ICXRegister;
    WORD IDXRegister;
    WORD IBPRegister;
    WORD ISIRegister;
    WORD IDIRegister;
    WORD IDSRegister;
    WORD IESRegister;
    WORD IIntNumber;
} InputParamStruct;
```

OutputParameters

```
OutputParamStruct   struc
    OAXRegister         dw  ?
    OBXRegister         dw  ?
    OCXRegister         dw  ?
    ODXRegister         dw  ?
    OBPRegister         dw  ?
    OSIRegister         dw  ?
    ODIRegister         dw  ?
    ODSRegister         dw  ?
    OESRegister         dw  ?
    OFlags              dw  ?
OutputParamStruct   ends
```

```
typedef struct OutputParameterStructure {
    WORD OAXRegister;
    WORD OBXRegister;
    WORD OCXRegister;
    WORD ODXRegister;
    WORD OBPRegister;
    WORD OSIRegister;
    WORD ODIRegister;
    WORD ODSRegister;
    WORD OESRegister;
    WORD OFlags;
} OutputParamStruct;
```

See Also: GetRealModeWorkSpace, Appendix F

EnterDebugger

(Non-blocking) v3.1x & v4.xx

Enter the Debugger

Syntax: void EnterDebugger(void);

Return Value: None

Requirements: None

Parameters: None

Example:

```
call    EnterDebugger   ;C call
                -OR-
int     3               ;assembly code equivalent
```

Description: EnterDebugger stops execution of the NetWare OS and enters the internal assembly language-oriented debugger.

See Also: Appendix B

Free

(Non-blocking) v3.1x & v4.xx

Returns previously allocated memory to OS

Syntax: void Free(void *MemoryAddress);

Return Value: None

Requirements: Interrupts disabled.

Parameters:

MemoryAddress	Passes a pointer to memory to be returned to NetWare (must have been acquired previously by a call to Alloc).

Example:

```
push    eax             ;ptr to memory allocated
call    Free            ;return to system
lea     esp, [esp + 4]  ;adjust stack pointer
```

Description: Free returns memory allocated by the driver for any purpose (typically for Read-After-Write Verify buffers or to read in custom data from the custom data file). Drivers are expected to make this call as needed. Returning memory to NetWare is an essential part of cleaning up before exiting.

See Also: Alloc

FreeBufferBelow16Meg

(Non-blocking) v3.1x & v4.xx

Returns previously allocated special buffer to OS

Syntax: void FreeBufferBelow16Meg(
void *MemoryAddress);

Return Value: None

Requirements: Interrupts disabled.

Parameters: MemoryAddress Passes a pointer to memory to be returned to NetWare (which must have been acquired previously by a call to AllocBufferBelow16Meg).

Example:

```
push    eax                     ;ptr to memory previously allocated
call    FreeBufferBelow16Meg    ;return to system
lea     esp, [esp + 4]          ;adjust stack pointer
```

Description: FreeBufferBelow16Meg returns memory allocated by the driver for Bus Master or DMA I/O which was required to be below 16 Megabytes (This memory must have been acquired by a call to AllocBufferBelow16Meg). Returning memory to NetWare is an essential part of cleaning up before exiting. See Appendix G for additional details.

See Also: AllocBufferBelow16Meg, Appendix G

FreeSemiPermMemory (Non-blocking) v3.1x

Returns previously allocated memory to OS

Syntax: void FreeSemiPermMemory(void *MemoryAddress);

Return Value: None

Requirements: Interrupts disabled. May not be called from interrupt level.

Parameters: MemoryAddress Passes a pointer to memory to be returned to NetWare (must have been acquired previously by a call to AllocSemiPermMemory).

Example:

```
push    eax                 ;ptr to memory allocated
call    FreeSemiPermMemory  ;return to system
lea     esp, [esp + 4]      ;adjust stack pointer
```

Description: FreeSemiPermMemory returns memory allocated by the driver for any purpose (typically for Read-After-Write Verify buffers or to read in custom data from the custom data file). Drivers are expected to make this call as needed. Returning memory to NetWare is an essential part of cleaning up before exiting.

See Also: AllocSemiPermMemory

GetCurrentTime

(Non-blocking) v3.1x & v4.xx

Returns current time in clock ticks since loading server

Syntax: LONG GetCurrentTime(void);

Return Value: LONG number of clock ticks (1/18th second) since the server was last loaded and began execution.

Requirements: None

Parameters: None

Example:

```
call        GetCurrentTime              ;get time in ticks
mov         CurrentTimeSave, eax        ;save for driver
```

Description: This call is useful to determine the current relative time in order to determine the elapsed time for some driver-related activities, etc. The current time value less the value returned at the start of an operation is the elapsed time in 1/18th second clock ticks. It requires more than 7 years for this timer to roll over, allowing it to be used for elapsed time comparisons.

See Also: Driver Initialization, Operation time-out

GetHardwareBusType

(Non-blocking) v3.1x & v4.xx

Returns I/O bus type and bios support indicators, etc.

Syntax: LONG GetHardwareBusType(void);

Return Value:
0 - I/O bus is ISA (Industry Standard Architecture)
1 - I/O bus is MCA (Micro-Channel Architecture)
2 - I/O bus is EISA (Extended Industry Standard Architecture)

Requirements: None

Parameters: None

Example:

```
call    GetHardwareBusType
mov     IOBusType, eax              ;save bus type
```

Description: This routine returns an value indicating the processor bus type, for use by the driver. Typical application would allow a driver to support two different board types, which, once initialized, appear identical to the driver.

See Also: Driver Initialization

GetIOCTL

(Non-blocking) v3.1x & v4.xx

Returns specified or next IOCTL request handle

Syntax:

```
IOCTLRequestStruct *GetIOCTL (
        CardStruct *CardHandle,
        IOCTLRequestStruct *IOCTLRequest);
```

Return Value: Pointer to an IOCTL request structure, or zero if unsuccessful.

Requirements: Interrupts disabled.

Parameters:

CardHandle — Passes a handle for the card structure for the associated adapter. AddDiskCard returned this handle to the driver.

IOCTLRequest — Passes a pointer to an IOCTL request structure. GetIOCTL returns this same value unless the value is zero, in which case, GetIOCTL returns a pointer to the next available IOCTL request.

Example:

```
        push    eax                     ;get specific IOCTL Request
        push    edx                     ;contains card handle
        call    GetIOCTL
        lea     esp, [esp + (2*4)]      ;adjust stack pointer
        or      eax, eax                ;got one ?
        jnz     DoIOCTLRequest          ;got IOCTL request
        ..
        ; no request was pending!!
        ..
DoIOCTLRequest:
        mov     esi, eax                ;save request pointer
```

Description: A driver's IOCTL notification routine or DriverISR routine calls GetIOCTL to obtain an IOCTL request from NetWare. GetIOCTL identifies the IOCTL request by passing a card handle and a pointer to the request structure. NetWare keeps the IOCTL requests on an IOCTL queue (one per card) in the order received, until the driver requests them.

In the event that the driver is busy when it receives an IOCTL request, the request will remain on the queue until the driver retrieves it with GetIOCTL. The driver may obtain the next IOCTL request issued for a card by passing a request handle of zero, or may request a specific IOCTL request by passing the desired request handle in the call.

Drivers must notify the Operating System of completion of the IOCTL request by making a call to PutIOCTL. See Chapter 5 for complete details on IOCTL function codes, IOCTL return status, and IOCTL processing.

See Also: PutIOCTL, GetRequest, PutRequest, Chapter 5

GetIOCTL (continued)

Function	*Sub-Function*	
0	*0*	*Activate Device*
	1	*Deactivate Device*
	2	*Format*
	3	*Device Verify Mode*
	4	*Identify Device*
	5	*Return Bad-Block Info*
	6	*Return Device Status*
	7	*Logical Device Mount*
	8	*Logical Device Dismount*
	9	*Lock Device Media*
	10	*Unlock Device Media*
	11	*Eject Media*
1	*0*	*ReturnDeviceInfo (see old v3.11 func.0, subfunc.17)*
	1	*ReturnMediaInfo (see old v3.11 func.0, subfunc.18)*
	2	*SetDeviceParameters (see old v3.11 func.0, subfunc.19)*
	3	*ReturnTapeDeviceInfo*
2	*0*	*ReturnMagazineInfo*
	1	*(not assigned)*
	2	*ReturnMagazineMediaMapping*
	3	*MagazineSelectCommand*
	4	*MagazineDeselectCommand*
	5	*MagazineLoad*
	6	*MagazineUnload*
	7	*MagazineEject*
3	*0*	*ReturnChangerInfo*
	1	*ReturnChangerDeviceMapping*
	2	*ReturnChangerMediaMapping*
	3	*ChangerCommand*
4-63		*Reserved by Novell*
64-255		*IOCTLs for third party use. Assigned by Novell*

IOCTL Functions deleted from the new specification

0	*12*	*Return Changer Element count*
	13	*Return Changer Element Info*
	14	*Changer command*
	15	*Select Media*
	16	*Unselect Media*

Figure 7-1 v3.1x/v4.xx IOCTL (I/O Control) Routine Assignments

GetIOCTL (continued)

Function	Sub-Function	
0	0	Activate Device
	1	Deactivate Device
	2	Format
	3	Device Verify Mode
	4	Identify Device
	5	Return Bad-Block Info
	6	Return Device Status
	7	Logical Device Mount
	8	Logical Device Dismount
	9	Lock Device Media
	10	Unlock Device Media
	11	Eject Media
	12	Return Changer Element count *
	13	Return Changer Element Info *
	14	Changer command *
	15	Select Media *
	16	Unselect Media *
	17	ReturnDeviceInfo (see 3.1x/v4.xx func.1, subfunc.0) *
	18	ReturnMediaInfo (see 3.1x/v4.xx func.1, subfunc.1) *
	19	SetDeviceParameters (see 3.1x/v4.xx func.1, subfunc.2) *
1-63		Reserved by Novell
64-255		IOCTLs for third party use. Assigned by Novell

* These IOCTLs are defined in later versions of the 3.11 specification but are never issued by the NetWare 3.11 OS.

Figure 7-2 Old v3.11 IOCTL (I/O Control) Routine Assignments

```
typedef struct IOCTLRequestStructure
    {
        LONG        DriverLink;
        CardStruct  *CardHandle;
        WORD        CompletionCode;
        BYTE        Function;
        BYTE        SubFunction;
        LONG        IOCTLParameter;
        LONG        *IOCTLBuffer;
    }  IOCTLRequestStruct;
```

Figure 7-3 The IOCTL Request Structure

GetIOCTL (continued)

Completion/Device Status returned to the calling application

No Error	0000h
Non-Media Error	0003h
Device Not Active	0004h
Adapter Card Error	0005h
Device Parameter Error	0006h
System Parameter Error	0007h
Not Supported By Device	0008h
Device Fault	0103h
No Media Present	0703h
Media Write Protected	0803h
Magazine Not Present	0F09h
Changer Error	1009h
Changer Source Empty	1109h
Changer Destination Full	1209h
Changer Jammed	1303h
Magazine Error	1409h
Magazine Source Empty	1509h
Magazine Destination Full	1609h
Magazine Jammed	1703h
Driver Custom Status	E0xxh - FExxh
Not Supported By Driver	FFF9h

Figure 7-4 IOCTL Request Return Status

GetReadAfterWriteVerifyStatus (Non-blocking) v3.1x & v4.xx

Returns global ReadAfterWrite verify status

Syntax: LONG GetReadAfterWriteVerifyStatus(void);

Return Value: 0 - Read-After-Write Verify disabled
1 - Read-After-Write Verify enabled

Requirements: None

Parameters: None

Example:

```
call    GetReadAfterWriteVerifyStatus
mov     RAWVerifySave, eax              ;save for driver
```

Description: The value returned by this call is a server level flag which determines if Read-After-Write Verification will take place. The value should be examined by drivers when the device is registered with the Operating System. If a specific override has been issued (such as an IOCTL call) for any drive, it takes precedence over this flag for that device.

See Also: Device Verify Mode IOCTL

GetRealModeWorkSpace (Non-Blocking) v3.1x & v4.xx

Syntax:

```
void GetRealModeWorkSpace(
        LONG *WorkSpaceSemaphore,
        LONG *ProtectedModeAddressOfWorkSpace,
        WORD *RealModeSegmentOfWorkSpace,
        WORD *RealModeOffsetOfWorkSpace,
        LONG *WorkSpaceSizeInBytes);
```

Return Value: None

Requirements: None

Parameters:

WorkSpaceSemaphore	receives a handle to the operating system semaphore structure
ProtectedModeAddressOfWorkSpace	receives a 32-bit logical address of the workspace block
RealModeSegmentOfWorkSpace	receives the real mode segment of workspace from the OS
RealModeOffsetOfWorkSpace	receives the real mode offset in the workspace segment from the OS
WorkSpaceSizeInBytes	receives the size of the workspace in bytes

Example: (See example below)

Description: *GetRealModeWorkSpace* is used in conjunction with *DoRealModeInterrupt* to allow the driver access to memory in real mode.

NetWare v3.1x and v4.xx drivers run in protected mode and do not allow direct access to BIOS based information. The call *DoRealModeInterrupt* allows the driver to access the BIOS and get data from it (See Appendix F).

DoRealModeInterrupt turns on the system interrupts and executes in a critical section; therefore, semaphore routines--*CPSemaphore* and *CVSemaphore* are called in order to keep other processes out of the workspace.

The driver must provide the following storage locations for the pointers that will be passed to it during this call:

```
WorkSpaceSemaphore              dd  0
ProtectedModeAddressOfWorkSpace dd  0
RealModeSegmentOfWorkSpace      dw  0
RealModeOffsetOfWorkSpace       dw  0
WorkSpaceSizeInBytes            dd  0
```

See Also: DoRealModeInterrupt

GetRealModeWorkSpace (continued)

Example:

```
;*****************************************************************************
;* Get  realmode workspace
;*****************************************************************************

push    OFFSET WorkSpaceSizeInBytes             ;size of workspace
push    OFFSET RealModeOffsetOfWorkSpace        ;real mode offset into segment
push    OFFSET RealModeSegmentOfWorkSpace       ;real mode segment address
push    OFFSET ProtectedModeAddressOfWorkSpace  ;address in protected mode
push    OFFSET WorkSpaceSemaphore               ;semaphore

call    GetRealModeWorkSpace                ;call OS to fill in information
add esp, (5 * 4)                            ;clean up stack

;*****************************************************************************
;* Lock the workspace
;*****************************************************************************

push    WorkSpaceSemaphore                  ;load semaphore
call    CPSemaphore                         ;lock workspace for our use
add esp, (1 * 4)                            ;clean up stack

;*****************************************************************************
;* Setup and execute real mode interrupt
;*****************************************************************************

movzx   eax, RealModeSegmentOfWorkSpace     ;get WorkSpace segment
movzx   ebx, RealModeOffsetOfWorkSpace      ;get offset into segment

mov cl, SlotToReadConfiguration             ;get slot number
xor ch, ch   ;read first block

mov esi, OFFSET InputParms                  ;point to input area
mov [esi].IAXRegister, 0D801h               ;Eisa read configuration
mov [esi].ICXRegister, cx                   ;slot and data block
mov [esi].ISIRegister, bx                   ;offset of DosWorkArea
mov [esi].IDSRegister, ax                   ;segment of DosWorkArea
mov [esi].IIntNumber, 15h                   ;interrupt number

push    OFFSET OutputParameters             ;pt at output regs
push    OFFSET InputParameters              ;pt at input regs
call    DoRealModeInterrupt                 ;tell os to do it
lea esp, [esp + 2 * 4]                      ;clear up stack

cmp eax, 0   ;did the OS do the
jne IntNotValidErrorExit                        ;int correctly
cmp byte ptr OutputParmeters.OAXRegister + 1,0  ;Bios Int 15 return
jne IntNotValidErrorExit                        ;successful ?

mov esi, ProtectedModeAddressOfWorkSpace        ;load pointer to data
movzx   ecx, BYTE PTR [esi + INTERRUPTOFFSET]   ;get int if any
and cl, ISOLATEINTMASK                          ;isolate interrupt level
jecxz   NoAddInterrupt                          ;if none skip add

mov SaveInterrupt, cl                           ;save interrupt for later
;*****************************************************************************
;* Unlock interrupt
;*****************************************************************************

NoAddInterrupt:
push    WorkSpaceSemaphore                  ;pass semaphore
call    CVSemaphore                         ;unlock workspace
add esp, (1 * 4)                            ;clean up stack
```

GetRequest

(Non-blocking) v3.1x & v4.xx

Returns next or specified I/O request structure pointer

Syntax: IORequestStruct *GetRequest(
DiskStruct *DiskHandle,
IORequestStruct *IORequest);

Return Value: Pointer to an I/O request structure, or 0 if unsuccessful

Requirements: Interrupts disabled.

Parameters: DiskHandle Handle for the target device. This is the same value returned by AddDiskDevice.

IORequest Pointer to an I/O request structure. GetRequest returns this same value unless the value supplied is zero, in which case, GetRequest returns a pointer to the next available I/O request (if any).

Example:

```
push    0                   ;for next I/O request
push    edi                 ;contains Disk structure ptr
call    GetRequest          ;see if one is available
lea     esp, [esp + (2*4)]  ;adjust stack pointer
```

Description: When NetWare has an I/O request for a specific device, NetWare calls the driver's request notification (IOPoll) routine, passing a DiskStructure Handle and a pointer to an I/O Request structure. The DiskStructure Handle is a structure pointer to the device. The I/O Request structure defines the read or write request. The driver's IOPoll or Interrupt service routine must call GetRequest to obtain an I/O request from NetWare.

For more details on the request structure, function codes, and related issues, please refer to Chapter 6.

See Also: PutRequest, GetIOCTL, PutIOCTL, Chapter 6

GetRequest (continued)

Name	Code
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)	
ConsecutiveSet Marks	09h
Write consecutive file marks	
space forward until consecutive set marks	
space backwards until 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
Format to partition media	
Select partition	
Return number of partitions	
Return partition size	
Return max number of possible partitions	
Physical Media Operations	0Dh
Security erase partition	
Rewind partition	
Goto end of partition	
Random Erase	0Eh
Reserved	0Fh-3Fh

Figure 7-5 I/O Function Codes

GetRequest (continued)

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

Figure 7-6 The I/O Request Structure

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
Ignored 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
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 7-7 I/O Request Return Status

GetSectorsPerCacheBuffer (Non-blocking) v3.1x & v4.xx

Returns number of sectors in server cache buffers

Syntax: LONG GetSectorsPerCacheBuffer(void);

Return Value: An integer (8, 16, or 32) indicating the number of sectors in a system cache buffer.

Requirements: None

Parameters: None

Example:

```
call    GetSectorsPerCacheBuffer      ;get typical request size
mov     CacheSizeSave, eax            ;for driver optimization
```

Description: This routine returns to the caller the number of sectors in a server cache buffer. The value returned will be either 8 (4K), 16 (8K), or 32 (16K). This value may allow drivers which allocate buffers in SRAM to allocate the optimal buffer size, thus providing better performance.

See Also: Chapter 3

MapAbsoluteAddressToCodeOffset (Non-blocking) v3.1x & v4.xx

Converts absolute memory address to logical NetWare address

Syntax: LONG MapAbsoluteAddressToCodeOffset(
LONG AbsoluteAddress);

Return Value: Logical address where code appears

Requirements: None

Parameters: AbsoluteAddress Real 32-bit absolute hardware memory address

Example:

```
mov     eax, AbsoluteAddress            ;get real SRAM address
push    eax
call    MapAbsoluteAddressToCodeOffset
lea     esp, [esp + 4]                  ;adjust stack pointer
mov     LogicalAddressSave, eax         ;SRAM appears at this address
```

Description: This routine converts absolute hardware memory addresses to logical Netware addresses that are used by the drivers and the Operating System. This routine may be used to convert an absolute address to the logical address where it will appear in NetWare address space. **This routine may only be used with memory addresses that have previously been registered with the OS.** (Shared RAM is registered through a call to the *RegisterHardwareOptions* API and its logical address is returned to the driver in the IOConfigStructure.)

See Also: MapCodeOffsetToAbsoluteAddress

MapAbsoluteAddressToDataOffset (Non-blocking) v3.1x & v4.xx

Converts absolute memory address to logical NetWare address

Syntax: LONG MapAbsoluteAddressToDataOffset(
LONG AbsoluteAddress);

Return Value: Logical address where data appears

Requirements: None

Parameters: AbsoluteAddress Real 32-bit absolute hardware memory address

Example:

```
mov     eax, AbsoluteAddress            ;get real SRAM address
push    eax
call    MapAbsoluteAddressToDataOffset
lea     esp, [esp + 4]                  ;adjust stack pointer
mov     LogicalAddressSave, eax         ;SRAM appears at this address
```

Description: This routine converts absolute hardware memory addresses to logical Netware addresses, used by drivers and by the Operating System. This routine may be used to convert an absolute address to the logical address where it will appear in NetWare address space. **This routine may only be used with memory addresses that have previously been registered with the OS.** (Shared RAM is registered through a call to the *RegisterHardwareOptions* API and its logical address is returned to the driver in the IOConfigStructure.)

See Also: MapDataOffsetToAbsoluteAddress

MapCodeOffsetToAbsoluteAddress (Non-blocking) v3.1x & v4.xx

Converts logical NetWare address to absolute memory address

Syntax: LONG MapCodeOffsetToAbsoluteAddress(
CodeOffset);

Return Value: 32-bit real hardware memory address

Requirements: None

Parameters: CodeOffset Logical NetWare 32-bit memory address

Example:

```
mov     eax, CodeOffset             ;netware data address
push    eax                         ;pass address driver uses
call    MapCodeOffsetToAbsoluteAddress
lea     esp, [esp + 4]              ;adjust stack pointer
mov     AbsAddrsave, eax            ;bus master card needs real address
```

Description: This routine converts a logical NetWare address, used throughout NetWare, to a real hardware memory address, required to initialize DMA channels and Bus Master devices. It also validates specified hardware options. **This routine may only be used with memory addresses that have previously been registered with the OS.**

See Also: MapAbsoluteAddressToCodeOffset

MapDataOffsetToAbsoluteAddress (Non-blocking) v3.1x & v4.xx

Converts logical NetWare address to absolute memory address

Syntax: LONG MapDataOffsetToAbsoluteAddress(
DataOffset);

Return Value: 32-bit real hardware memory address

Requirements: None

Parameters: DataOffset Logical NetWare 32-bit memory address

Example:

```
mov     eax, DataOffset                 ;netware data address
push    eax                             ;pass address driver uses
call    MapDataOffsetToAbsoluteAddress
lea     esp, [esp + 4]                  ;adjust stack pointer
mov     AbsAddrsave, eax                ;bus master card needs real address
```

Description: This routine converts a logical NetWare address, used throughout NetWare, to a real hardware memory address, required to initialize DMA channels, Bus Master devices, and to validate specified hardware options. **This routine may only be used with memory addresses that have previously been registered with the OS.**

See Also: MapAbsoluteAddressToDataOffset

NetWareAlert

(Non-blocking) v4.xx

Notifies system of serious driver problem

Syntax:

```
void  NetWareAlert(
            LONG NLMHandle,
            NWAlertStruct *Alert,
            LONG ParamCount,
            args...);
```

Return Value: None

Requirements: None

Parameters:

NLMHandle	The handle NetWare passed on the stack to the driver initialization routine.
Alert	A handle to a NetWareAlert structure that holds the display, format and routing information of the message to be sent. The structure size and format is defined below.
ParamCount	The number of additional parameters to be passed as determined by the *Control String* field in NetWareAlert structure passed through the *Alert* parameter.
args...	Additional arguments to be passed. (See *ParamCount*.)

NetWareAlertStructure

```
NWAlertStruct   struc
    Reserved0               dd  ?
    AlertFlags              dd  ?
    TargetStation           dd  ?
    TargetNotificationBits  dd  ?
    AlertID                 dd  ?
    AlertLocus              dd  ?
    AlertClass              dd  ?
    AlertSeverity           dd  ?
    Reserved1               dd  ?
    Reserved2               dd  ?
    ControlString           dd  ?
    Reserved3               dd  ?
NWAlertStruct   ends
```

```
typedef struct NetWareAlertStructure {
    LONG Reserved0;
    LONG AlertFlags;
    LONG TargetStation;
    LONG TargetNotificationBits;
    LONG AlertID;
    LONG AlertLocus;
    LONG AlertClass;
    LONG AlertSeverity;
    LONG Reserved1;
    LONG Reserved2;
    BYTE *ControlString;
    LONG Reserved3;
} NWAlertStruct;
```

NetWareAlert (continued)

Each field in the NetWareAlert structure is defined below:

Reserved0 This parameter should be initialized to a NULL (0).

AlertFlags Masks the functionality of the structure. (This field is usually set to QUEUE_THIS_ALERT_MASK.)

QUEUE_THIS_ALERT_MASK	01h
ALERTID_VALID_MASK	02h
ALERT_LOCUS_VALID_MASK	04h
ALERT_EVENT_NOTIFY_ONLY_MASK	08h
ALERT_NO_EVENT_NOTIFY_MASK	10H

TargetStation Supply a zero for the console.

TargetNotificationBits Identifies destinations of notification

NOTIFY_CONNECTION_BITS	01h
NOTIFY_EVERYONE_BIT	02h
NOTIFY_ERROR_LOG_BIT	04h
NOTIFY_CONSOLE_BIT	08h

AlertID Provides error code for system log, as follows:

OK	00h
ERR_HARD_FAILURE	0FFh

AlertLocus Defines locus of error (always disks)

LOCUS_DISKS	03h

AlertClass Indicates class of error, as follows:

CLASS_UNKNOWN	00h
CLASS_TEMP_SITUATION	02h
CLASS_HARDWARE_ERROR	05h
CLASS_BAD_FORMAT	09h
CLASS_MEDIA_FAILURE	11h
CLASS_CONFIGURATION_ERROR	15h
CLASS_DISK_INFORMATION	18h

NetWareAlert (continued)

AlertSeverity	Indicates error severity, as follows: SEVERITY_INFORMATIONAL 00h SEVERITY_WARNING 01h SEVERITY_RECOVERABLE 02h SEVERITY_CRITICAL 03h SEVERITY_FATAL 04h SEVERITY_OPERATION_ABORTED 05h
Reserved1	This parameter should be initialized to a NULL (0).
Reserved2	This parameter should be initialized to a NULL (0).
ControlString	Pointer to null-terminated control string similar to that used in the sprintf function, including embedded returns, line-feeds, tabs, bells, and % specifiers (except floating-point specifiers).

Example:

```
push    0                       ;no arguments
push    Alert                   ;handle to the NetWareAlert structure
push    NLMHandle
call    NetWareAlert            ;tell system of problem
lea     esp, [esp + (3*4)]      ;adjust stack pointer
```

Description: Provides system notification of driver hardware or software problems at times **other** than during driver initialization procedure.

See Also: OutputToScreen

OutputToScreen

(Non-Blocking) v3.1x & v4.xx

Outputs message to Driver initialize screen

Syntax:

```
void  OutputToScreen(
            LONG ScreenHandle,
            BYTE *ControlString,
            args...);
```

Return Value: None

Requirements: May be called only during driver initialize procedure

Parameters:

ScreenHandle	Handle of console screen passed to driver on stack upon entry to the driver initialize procedure, becomes invalid upon return from driver initialize procedure.
ControlString	Pointer to a null-terminated ASCII control string similar to that used with sprintf, including embedded returns, line feeds, tabs, bells, and % specifiers (except floating-point specifies).
args	Arguments as indicated by the above control string.

Example:

```
push    arg                 ;if just one argument
push    esi                 ;contains ptr to string
push    ScreenHandle        ;init screen handle (init only)
call    OutputToScreen      ;may only call during init
lea     esp, [esp + (3*4)]  ;adjust stack pointer
```

Description: This routine displays a driver error message on the server console screen. Drivers should not display non-vital messages, and must limit the number of lines output to the screen for essential messages (the OS will display drives registered and their descriptive text, etc.). Drivers which display unneeded output will cause important information to scroll off the console screen. This routine is similar in function to the **sprintf** function.

See Also: Driver Initialization, NetWareAlert

ParseDriverParameters (Blocking) v3.1x & v4.xx

Parses LOAD command line, prompts, and validates parameters

Syntax:

```
LONG  ParseDriverParameters(
            struct IOConfigurationStructure *IOConfig,
            LONG Reserved0,
            AdapterOptionStruct *Options,
            LONG Reserved1,
            LONG Reserved2,
            LONG NeedBitMap,
            BYTE *CommandLine,
            LONG ScreenHandle);
```

Return Value:

0	Success
non-zero	Failure - conflict or bad command line parameters

Requirements: Must be called from blocking process level only.

Parameters:

IOConfig	Pointer to Adapter's corresponding IOConfiguration structure (must be initialized and have correct resource tag stored in it).
Reserved0	Reserved by NetWare
Options	Pointer to Adapter Options Definition Structure.
Reserved1	Reserved by NetWare
Reserved2	Reserved by NetWare
NeedBitMap	A bit map (double word value) telling ParseDriverParameters which hardware options the driver requires, as follows:

```
NeedsIOSlotBit              equ  0001h
NeedsIOPort0Bit             equ  0002h
NeedsIOLength0Bit           equ  0004h
NeedsIOPort1Bit             equ  0008h
NeedsIOLength1Bit           equ  0010h
NeedsMemoryDecode0Bit       equ  0020h
NeedsMemoryLength0Bit       equ  0040h
NeedsMemoryDecode1Bit       equ  0080h
NeedsMemoryLength1Bit       equ  0100h
NeedsInterrupt0Bit          equ  0200h
NeedsInterrupt1Bit          equ  0400h
NeedsDMA0Bit                equ  0800h
NeedsDMA1Bit                equ  1000h
```

ParseDriverParameters (continued)

CommandLine	Pointer to command line passed to the driver's Initialize routine on the stack at load time.
ScreenHandle	Handle to the driver's screen display. NetWare also passed this value to the driver's Initialize Driver routine on the stack at load time.

Example:

```
mov     eax, cardNum                    ;our adapter index
push    [esp + Parm1]                   ;init screen handle
push    [esp + Parm2]                   ;command line pointer
push    NeedsIOPort0Bit + NeedsInterrupt0Bit    ;need I/O port and interrupt
push    0                               ;frame type description
push    0                               ;LAN config limits
push    OFFSET Options                  ;card options template
push    0                               ;driver configuration
mov     ebx, IOConfigList[eax * 4]      ;get IOConfig structure from list
push    ebx                             ;IOConfig structure ptr
call    ParseDriverParameters           ;fill out our IOConfig Structure
lea     esp, [esp + (8*4)]              ;adjust stack pointer
```

Description: ParseDriverParameters fills in the IOConfigurationStructure associated with an adapter board, utilizing tables provided by the driver, the command line parameters, and operator input. This routine allows a driver's Initialization routine to accept I/O Port addresses and ranges, memory decode addresses and lengths, interrupts, and DMA addresses from the driver "load" command line. All values inputed at the commandline are treated and displayed as hex values. For example, a load command could contain the following specifications:

load sample port = 300, port length = 32, int = 3 <Enter>

In this case, the driver "SAMPLE" is being loaded. The first adapter board will occupy I/O ports 300h to 31Fh and interrupt 3.

ParseDriverParameters (continued)

ParseDriverParameters works in conjunction with another "C" NetWare routine called RegisterHardwareOptions. The following list describes how these two routines work in unison:

- As mentioned above, ParseDriverParameters looks for information about I/O Port addresses and ranges, memory decode addresses and lengths, interrupts, and/or DMA addresses depending on what the adapter board needs.

- ParseDriverParameters looks for this information in two sources: (1) the command line, and (2) the Options structure which is a hard-coded part of the driver's data segment.

- ParseDriverParameters uses a NeedBitMap to determine which hardware options the adapter board needs.

- If the NeedBitMap requires data and ParseDriverParameters cannot find the data on the command line or in the AdapterOptionsStructure table associated with the required item, ParseDriverParameters will prompt the console operator for the data, showing as a default the first entry in the table pointed at by the associated entry in the AdapterOptionsStructure.

- Using the NeedBitMap as a shopping list, ParseDriverParameters collects the necessary information from the command line and from the Options structure, fills out the IOConfiguration Structure, and returns successfully.

- RegisterHardwareOptions then uses the IOConfiguration structure to reserve the specified file server hardware options.

ParseDriverParameters (continued)

The command line keywords are:

SLOT =
PORT =
PORT LENGTH =
MEM =
MEM LENGTH =
INT =
DMA CHANNEL =

The following two keywords are valid if NeedsIOPort1Bit is set:

PORT1 =
PORT LENGTH =

The following two keywords are valid if NeedsMemoryDecode1Bit is set:

MEM1 =
MEM LENGTH =

The following keyword is valid if NeedsInterrupt1Bit is set:

INT1 =

The following keyword is valid if NeedsDma1Bit is set:

DMA CHANNEL1 =

The driver may implement additional custom keywords which it alone may recognize. The driver must then parse the command line itself (It is recommended that the driver not adjust the command line pointer, but simply allow the ParseDriverParameters routine to ignore and skip over the additional parameters).

ParseDriverParameters (continued)

The Adapter Options Structure is defined as follows:

```
AdapterOptionStruct     struc
 IOSlot             dd      ? ;MCA or EISA slot #
 IOPort0            dd      ? ;I/O port base
 IOLength0          dd      ? ;range (# ports)
 IOPort1            dd      ? ;2nd I/O port base
 IOLength1          dd      ? ;range (# ports)
 MemoryDecode0      dd      ? ;memory (SRAM/EPROM)
 MemoryLength0      dd      ? ;range (paragraphs)
 MemoryDecode1      dd      ? ;2nd memory base
 MemoryLength1      dd      ? ;range (paragraphs)
 Interrupt0         dd      ? ;Interrupt #
 Interrupt1         dd      ? ;2nd Int #
 DMA0               dd      ? ;DMA channel
 DMA1               dd      ? ;2nd DMA channel
AdapterOptionStruct     ends
```

Each entry in the above options structure is normally a pointer to a table. If the entry is zero (a zero pointer), no table exists for that entry. Each table consists of a doubleword containing the number of following table entries. Each table entry represents a valid value which may be selected from the command line. The default entry if none is specified is the first entry in the table, and subsequent entries in order of occurrence in the table.

Note: It is not valid to indicate that an entry is required by setting the associated bit in the NeedBitMap while having a zero pointer or a table with the number of entries indicated as zero.

A sample option table follows:

```
PortOptionTable:
    dd  4       ;number of port table entries
    dd  340h    ;first (default) port address
    dd  344h    ;second possible port address
    dd  320h    ;third possible port address
    dd  324h    ;last possible port address
```

A driver typically maintains one AdapterOptionsStructure, although multiple Adapter Options Structures may be used if the driver supports more than one adapter type requiring different parameters.

See Also: AdapterOptionStructure, IOConfigurationStructure, CardStructure, RegisterHardwareOptions, DeRegisterHardwareOptions

PutIOCTL

(Non-blocking) v3.1x & v4.xx

Posts IOCTL (control) request completion

Syntax:

```
LONG  PutIOCTL(
            CardStruct *CardHandle,
            IOCTLRequestStruct *IOCTLRequest);
```

Return Value:

0	Success
non-zero	Invalid Request

Requirements: Interrupts disabled. (see note below)

Parameters:

CardHandle	Passes a handle to the card structure for the associated adapter board. AddDiskCard returned this handle to the driver.
IOCTLRequest	Passes a pointer to an IOCTL request.

Example:

```
push        eax                     ;IOCTL request ptr
push        ebx                     ;CardStructure address
call        PutIOCTL
lea         esp, [esp + (2*4)]      ; adjust stack pointer
```

Description: PutIOCTL notifies NetWare of the completion of an IOCTL request. PutIOCTL may be called from the driver ISR or from the driver IOCTL request notification routine (IOCTLPoll). PutIOCTL must be called for every IOCTL request. The driver must have placed the completion status in the IOCTL request prior to making this call to post completion.

NOTE: This routine may open an interrupt window, even though it must be called with interrupts disabled and returns with interrupts disabled. For more information, see Chapter 5.

See Also: GetIOCTL, GetRequest, PutRequest, Chapter 5

PutIOCTL (continued)

Function	*Sub-Function*	
0	*0*	*Activate Device*
	1	*Deactivate Device*
	2	*Format*
	3	*Device Verify Mode*
	4	*Identify Device*
	5	*Return Bad-Block Info*
	6	*Return Device Status*
	7	*Logical Device Mount*
	8	*Logical Device Dismount*
	9	*Lock Device Media*
	10	*Unlock Device Media*
	11	*Eject Media*
1	*0*	*ReturnDeviceInfo (see old v3.11 func.0, subfunc.17)*
	1	*ReturnMediaInfo (see old v3.11 func.0, subfunc.18)*
	2	*SetDeviceParameters (see old v3.11 func.0, subfunc.19)*
	3	*ReturnTapeDeviceInfo*
2	*0*	*ReturnMagazineInfo*
	1	*(not assigned)*
	2	*ReturnMagazineMediaMapping*
	3	*MagazineSelectCommand*
	4	*MagazineDeselectCommand*
	5	*MagazineLoad*
	6	*MagazineUnload*
	7	*MagazineEject*
3	*0*	*ReturnChangerInfo*
	1	*ReturnChangerDeviceMapping*
	2	*ReturnChangerMediaMapping*
	3	*ChangerCommand*
4-63		*Reserved by Novell*
64-255		*IOCTLs for third party use. Assigned by Novell*

IOCTL Functions deleted from the new specification

0	*12*	*Return Changer Element count*
	13	*Return Changer Element Info*
	14	*Changer command*
	15	*Select Media*
	16	*Unselect Media*

Figure 7-8 v3.1x/v4.xx IOCTL (I/O Control) Routine Assignments

PutIOCTL (continued)

Function	Sub-Function	
0	0	Activate Device
	1	Deactivate Device
	2	Format
	3	Device Verify Mode
	4	Identify Device
	5	Return Bad-Block Info
	6	Return Device Status
	7	Logical Device Mount
	8	Logical Device Dismount
	9	Lock Device Media
	10	Unlock Device Media
	11	Eject Media
	12	Return Changer Element count *
	13	Return Changer Element Info *
	14	Changer command *
	15	Select Media *
	16	Unselect Media *
	17	ReturnDeviceInfo (see v3.1x/v4.xx func.1, subfunc.0) *
	18	ReturnMediaInfo (see v3.1x/v4.xx func.1, subfunc.1) *
	19	SetDeviceParameters (see v3.1x/v4.xx func.1, subfunc.2) *
1-63		Reserved by Novell
64-255		IOCTLs for third party use. Assigned by Novell

* These IOCTLs are defined in later versions of the 3.11 specification but are never issued by the NetWare 3.11 OS.

Figure 7-9 Old v3.11 IOCTL (I/O Control) Routine Assignments

```
typedef struct IOCTLRequestStructure
    {
        LONG        DriverLink;
        CardStruct  *CardHandle;
        WORD        CompletionCode;
        BYTE        Function;
        BYTE        SubFunction;
        LONG        IOCTLParameter;
        LONG        *IOCTLBuffer;
    }  IOCTLRequestStruct;
```

Figure 7-10 The IOCTL Request Structure

PutIOCTL (continued)

Completion/Device Status returned to the calling application

No Error	0000h
Non-Media Error	0003h
Device Not Active	0004h
Adapter Card Error	0005h
Device Parameter Error	0006h
System Parameter Error	0007h
Not Supported By Device	0008h
Device Fault	0103h
No Media Present	0703h
Media Write Protected	0803h
Magazine Not Present	0F09h
Changer Error	1009h
Changer Source Empty	1109h
Changer Destination Full	1209h
Changer Jammed	1303h
Magazine Error	1409h
Magazine Source Empty	1509h
Magazine Destination Full	1609h
Magazine Jammed	1703h
Driver Custom Status	E0xxh - FExxh
Not Supported By Driver	FFF9h

Figure 7-11 IOCTL Request Return Status

PutRequest

(Non-blocking) v3.1x & v4.xx

Posts I/O request completion

Syntax:

```
LONG PutRequest(
        DiskStruct *DiskHandle,
        IORequestStruct *IORequest);
```

Return Value:

0	Successful
non-zero	Invalid Request

Requirements: Interrupts disabled. (see note below)

Parameters:

DiskHandle	Passes a handle for the target device. This is the same value returned by AddDiskDevice.
IORequest	Passes a pointer to the I/O request structure to be returned to NetWare.

Example:

```
mov     [esi].SCompletionCode, 0    ;indicate good completion
push    esi                         ;ptr to I/O Request structure
push    edi                         ;contains Disk structure ptr
call    PutRequest                  ;notify OS of completion
lea     esp, [esp + (2*4)]          ;adjust stack pointer
```

Description: PutRequest notifies the Operating System that an I/O request has been completed. The completion status code must be placed in the request structure prior to making this call. Several driver routines call this routine, including a driver's Remove Driver, I/O Poll, and Interrupt Service routines.

NOTE: This routine may open an interrupt window, even though it must be called with interrupts disabled and returns with interrupts disabled. For more information, see Chapter 6.

See Also: GetRequest, GetIOCTL, PutIOCTL, Chapter 6

PutRequest (continued)

Name	*Code*
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)	
ConsecutiveSet Marks	*09h*
Write consecutive file marks	
space forward until consecutive set marks	
space backwards until 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*
Format to partition media	
Select partition	
Return number of partitions	
Return partition size	
Return max number of possible partitions	
Physical Media Operations	*0Dh*
Security erase partition	
Rewind partition	
Goto end of partition	
Random Erase	*0Eh*
Reserved	*0Fh-3Fh*

Figure 7-12 I/O Function Codes

PutRequest (continued)

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

Figure 7-13 The I/O Request Structure

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
Ignored 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
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 7-14 I/O Request Return Status

QueueSystemAlert (Non-blocking) v3.1x

Notifies system of serious driver problem

Syntax:

```
LONG QueueSystemAlert(
        LONG TargetStation,
        LONG TargetNotificationBits,
        LONG ErrorLocus,
        LONG ErrorClass,
        LONG ErrorCode,
        LONG ErrorSeverity,
        void *ControlString,
        args...);
```

Return Value: None

Requirements: None

Parameters:

TargetStation — Supply a zero for the console

TargetNotificationBits — Identifies destinations of notification

NOTIFY_CONNECTION_BITS	01h
NOTIFY_EVERYONE_BIT	02h
NOTIFY_ERROR_LOG_BIT	04h
NOTIFY_CONSOLE_BIT	08h

ErrorLocus — Defines locus of error (always disks)

LOCUS_DISKS	03h

ErrorClass — Indicates class of error, as follows:

CLASS_UNKNOWN	0
CLASS_TEMP_SITUATION	2
CLASS_HARDWARE_ERROR	5
CLASS_BAD_FORMAT	9
CLASS_MEDIA_FAILURE	11
CLASS_CONFIGURATION_ERROR	15
CLASS_DISK_INFORMATION	18

ErrorCode — Provides error code for system log, as follows:

OK	00h
ERR_HARD_FAILURE	0FFh

QueueSystemAlert (continued)

ErrorSeverity	Indicates error severity, as follows: SEVERITY_INFORMATIONAL 0 SEVERITY_WARNING 1 SEVERITY_RECOVERABLE 2 SEVERITY_CRITICAL 3 SEVERITY_FATAL 4 SEVERITY_OPERATION_ABORTED 5
ControlString	Pointer to null-terminated control string similiar to that used in the sprintf function, including embedded returns, line-feeds, tabs, bells, and simple % specifiers (excluding modifying, precision and floating-point specifiers).
args	Arguments as indicated by the above control string.

Example:

```
push    arg                             ;if single argument
push    eax                             ;ptr to control string
push    SEVERITY_CRITICAL               ;severity level
push    ERR_HARD_FAILURE                ;error code
push    CLASS_HARDWARE_ERROR            ;error class
push    LOCUS_DISKS                     ;locus of error
push    NOTIFY_CONSOLE_BIT + NOTIFY_ERROR_LOG_BIT
push    0                               ;target station
call    QueueSystemAlert                ;tell system of problem
lea     esp, [esp + (8*4)]              ;adjust stack pointer
```

Description: Provides system notification of driver hardware or software problems at times **other** than during driver initialization procedure.

See Also: OutputToScreen

ReadPhysicalMemory (Blocking) v4.xx

This routine must be used to access data stored in the DOS address space. The information is copied to a buffer allocated by the driver where it then is visible.

Syntax:

```
LONG ReadPhysicalMemory (
        BYTE  *Source,
        BYTE  *Destination,
        LONG  NumUnits,
        LONG  UnitSize);
```

Return Value:

1 (true; non-zero) Parameters were valid; transfer completed
0 (false) Transfer not completed because of bad parameters

Requirements: Must be called from blocking process level only.

Parameters:

Source	A physical address of memory below 0x100000.
Destination	Handle to a buffer allocated by the driver to hold the copied data.
NumUnits	Number of units to be read from memory.
UnitSize	Size in bytes of each unit to be read.

Description: Assumes that data passed in will not hang the machine; the physical address range must be below 0x100000; The word-sized requests must begin on word boundaries and longword request must begin on longword boundaries.

RegisterForEventNotification (Blocking) v3.1x & v4.xx

Registers a procedure to be called prior to specific system events

Syntax:

```
LONG  RegisterForEventNotification(
            LONG ResourceTag,
            LONG EventType,
            LONG Priority,
            LONG (*WarnProcedure)(
                void (*OutputRoutine)(void *ControlString, ...),
                LONG Parameter),
            void (*ReportProcedure)(
                LONG Parameter));
```

Return Value: Returns a 32 bit EventID (0 if call failed) to be used with a subsequent UnRegisterEventNotification call.

Requirements: Must be called from blocking process level only.

Parameters:

EventResourceTag — The resource tag returned by an AllocateResourceTag call during driver initialization which must have been made using the Event resource signature.

EventType — Indicates the type of event for which the caller wishes notification.

The following describes event for which notification may be received, the type of notification that can be made (Warn, Report or both), the environment of the notification call (blocking, non-blocking) and the defined use of the parameter that is passed with the call.

Type Definition	**Type Number** (in Decimal)
EVENT_VOL_SYS_MOUNT The parameter is undefined. Report Routine will be called **immediately after** vol SYS has been mounted. The Report Routine may block the thread.	0
EVENT_VOL_SYS_DISMOUNT The parameter is undefined. Both the Warn and Report Routines will be called **before** vol SYS is dismounted. The Report Routine may block the thread.	1
EVENT_ANY_VOL_MOUNT The parameter is the volume number. The Report Routine will be called **immediately after** any volume is mounted. The Report Routine may block the thread.	2

RegisterForEventNotification (continued)

EventType (contd)	EVENT_ANY_VOL_DISMOUNT The parameter is the volume number. The Warn **and** the Report Routines will be called **before** any volume is dismounted. The Report Routine may block the thread.	3
	EVENT_DOWN_SERVER The parameter is undefined. The Warn **and** Report routines will be called **before** the server is shut down. The Report Routine may block the thread.	4
	EVENT_CHANGE_TO_REAL_MODE The parameter is undefined. The Report routine will be called **before** the server changes to real mode. No blocking calls may be made by the Report Routine.	5
	EVENT_RETURN_FROM_REAL_MODE The parameter is undefined. The Report routine will be called **after** the server has returned from real mode. No blocking calls may be made by the Report Routine.	6
	EVENT_EXIT_TO_DOS The parameter is undefined. The Report routine will be called **before** the server exits to DOS. The Report Routine may block the thread.	7
	EVENT_MODULE_UNLOAD The parameter is the module handle. The Warn **and** Report routines will be called **before** a module is unloaded from the console command line. Only the Report routine will be called when a module unloads itself. The Report Routine may block the thread.	8
	EVENT_ACTIVATE_SCREEN The parameter is the Screen ID. The Report Routine is called **after** the screen becomes the active screen. The Report Routine may block the thread.	14

RegisterForEventNotification (continued)

EventType (contd)	EVENT_UPDATE_SCREEN The parameter is the Screen ID. The Report routine is called **after** a change is made to the screen image. The Report Routine may block the thread.	15
	EVENT_UPDATE_CURSOR The parameter is the Screen ID. The Report Routine is called **after** a change to the cursor position or state occurs. No blocking calls may be made by the Report Routine.	16
	EVENT_KEY_WAS_PRESSED The parameter is undefined. The Report Routine is called **after** any key on the keyboard has been pressed (including shift/alt/control). This routine is called at interrupt time. No blocking calls may be made by the Report Routine.	17
	EVENT_DEACTIVATE_SCREEN The parameter is the Screen ID. The Report Routine is called **after** the screen becomes inactive. No blocking calls may be made by the Report Routine.	18
	EVENT_OPEN_SCREEN The parameter is the Screen ID for the newly created screen. The Report Routine will be called **after** the screen is created. The Report Routine may block the thread.	20
	EVENT_CLOSE_SCREEN The parameter is the Screen ID for the screen that will be closed. The Report Routine will be called **before** the screen is closed. The Report Routine may block the thread.	21
	EVENT_MODULE_LOAD The parameter is the module handle. The Report Routine will be called **after** the module has been loaded. The Report Routine may block the thread.	27
	EVENT_GENERIC	32

RegisterForEventNotification (continued)

Priority — The priority used to call this notification procedure. Priorities are defined as follows:

Priority Definition	**Priority Number** (in Decimal)
EVENT_PRIORITY_OS	0
EVENT_PRIORITY_APPLICATION	20
EVENT_PRIORITY_DEVICE	40

WarnProcedure — A pointer to a procedure that is called when the OS makes an EventCheck call. If the warn routine does not want the event to occur, it must output a message and then return a non-zero value. Most event notification procedures are called at process level, but several are made at interrupt level (the thread may not be blocked). The above table of event types specifies which events must be checked to determine if the event allows its thread to be blocked.

ReportProcedure — A pointer to a procedure that is called when the OS makes an EventReport call. Its environment is the same as the Warn procedure indicated above.

RegisterForEventNotification (continued)

Example:

```
push    OFFSET ReportProcedure          ;report procedure
push    OFFSET WarnProcedure            ;warn procedure
push    Priority                        ;typically 40
push    EVENT_DOWN_SERVER               ;indicate event type
push    ResourceTag                     ;obtained during init
call    RegisterForEventNotification
lea     esp, [esp + (5*4)]              ;adjust stack pointer
```

Description: On some occasions a driver is required to perform some action prior to the OS terminating, switching to real mode, exiting to DOS, etc. The driver should call RegisterForEventNotification providing notification procedure pointers as indicated above. Even though the calls to register and un-register the event notification are blocking, the actual call to the event notification procedure provided by the driver is not always made from blocking process level (the environment varies with the particular event being reported).

The Warn Routine will be provided with two parameters when called. The first is the output routine which must be used to output messages (the output routine must be called with a control string and as many parameters and the control string indicates), and the second is the parameter described in each of the event types above. When the Report Routine is called it is passed a single parameter. This is the same parameter described in each of the event types above.

See Also: UnRegisterEventNotification, Driver Unload, Switch to Real Mode, Exit to DOS, AllocateResourceTag

RegisterHardwareOptions (Blocking) v3.1x & v4.xx

Reserves hardware options for an adapter card.

Syntax:

```
LONG  RegisterHardwareOptions(
            IOConfigStruct *IOConfig,
            LONG Reserved0);
```

Return Value:

0	Success
non-zero	Conflicting Option

Requirements: Must be called only from blocking process-level.

Parameters:

IOConfig	Handle to the adapter board's corresponding IOConfiguration structure. (The structure must be initialized with appropriate values, including the correct resource tag.)
Reserved0	Reserved by NetWare. A NULL (0) must be passed in this parameter.

Example:

```
;  ebx points to the IOConfig structure filled out by ParseDriverParameters
mov     eax, IORtag                 ;tag acquired for I/O registration
mov     [ebx].CRTagPointer, eax     ;put resource tag in IOConfig
push    0                           ;no driver config structure
push    ebx                         ;IOConfig structure
call    RegisterHardwareOptions
lea     esp, [esp + (2*4)]          ;adjust stack pointer
or      eax, eax                    ;error ?
jnz     InitRegisterHardwareError   ;yes - deal with it
```

Description: RegisterHardwareOptions is called by a driver's initialization routine to reserve hardware options for a particular adapter board. The driver passes RegisterHardwareOptions a IOConfigurationStructure pointer for the adapter card (to reserve the specified hardware options). If any of the hardware options are already in use, the routine returns an error code.

See Also: DeRegisterHardwareOptions, ParseDriverParameters, Driver Initialization, IOConfigurationStructure, AllocateResourceTag

RemoveDiskDevice (Blocking) v3.1x & v4.xx

Notifies applications using a device of pending device removal, prepares device for removal and deactivates device

Syntax:

```
void RemoveDiskDevice(
        DiskStruct *DiskHandle,
        LONG Status);
```

Return Value: None

Requirements: Must be called from blocking process level only.

Parameters:

DiskHandle — Passes a handle for the target device. This is the same value returned by AddDiskDevice.

Status — This parameter is included in the NetWare 3.1x and 4.xx versions for capatibility reasons only. It should be **initialized to a two (2)**.

Example:

```
push    2                   ;status
push    edi                 ;contains Disk structure ptr
call    RemoveDiskDevice
lea     esp, [esp + (2*4)]  ;adjust stack pointer
```

Description: A driver calls RemoveDiskDevice to remove a mass storage device from the file server's list of active devices. After returning from this routine, the driver then calls DeleteDiskDevice to return memory allocated for the DiskStructure. NetWare flushes all requests to the device before de-registering the device. This is done by making repeated calls to the device's IOPoll routine. (Note: **Only one IOPoll call is made per request**. Requests whose IOPoll was called previously will not be repeated.) The driver must remain ready to service further I/O requests if they are issued. RemoveDiskDevice will not return until all requests on the elevator queue have been serviced. (i.e. a GetRequest and a PutRequest has been performed on them) Once this is completed the OS issues a Deactivate IOCTL and returns.

See Also: DeleteDiskDevice

ScheduleNoSleepAESProcessEvent (Non-Blocking) v3.1x & v4.xx

Schedules an asynchronous event (non-blocking thread or process)

Syntax: void ScheduleNoSleepAESProcessEvent(
AESEventStruct *AESEvent);

Return Value: None

Requirements: Interrupts disabled.

Parameters: AESEvent Passes a pointer to an AES structure.

Example:

```
push    eax                         ;contains ptr to AES structure
call    ScheduleNoSleepAESProcessEvent
lea     esp, [esp + 4]              ;adjust stack pointer
```

Description: A driver's Initialization routine may call ScheduleNoSleepAESProcessEvent to set up a background AES (Asynchronous Event Scheduler) entry to a designated "gremlin" that will run throughout the time that the driver is loaded in file server memory. The driver must allocate the structure prior to the first call, must have placed the AES resource tag acquired at initialization into the structure, and must provide the execution interval and execution address.

A single call to this routine will cause a single entry to the defined routine, thus requiring another call to this routine at the conclusion of the routine executed if it is desired to have a subsequent exit to the routine. (See "Timeout" in Chapter 2.)

See Also: CancelNoSleepAESProcessEvent, AllocateResourceTag

ScheduleSleepAESProcessEvent (Non-Blocking) v3.1x & v4.xx

Schedules an asynchronous event (blocking thread or process)

Syntax: void ScheduleSleepAESProcessEvent(
AESEventStruct *AESEvent);

Return Value: None

Requirements: Interrupts disabled.

Parameters: AESEvent Passes a pointer to an AES structure.

Example:

```
push    eax                         ;contains ptr to AES structure
call    ScheduleSleepAESProcessEvent
lea     esp, [esp + 4]              ;adjust stack pointer
```

Description: A driver may call ScheduleSleepAESProcessEvent to set up a background AES (Asynchronous Event Scheduler) thread that will be executed at a desired interval and can be blocked or make blocking calls while executing. The driver must allocate the structure prior to the first call, must have placed the AES resource tag acquired during initialization into the structure, and must provide the execution interval and execution address. A single call to this routine will cause a single entry to the defined routine, thus requiring another call to this routine at the conclusion of the routine executed if it is desired to have a subsequent exit to the routine.

See Also: CancelSleepAESProcessEvent, AllocateResourceTag, ScheduleNoSleepAESProcessEvent, CancelNoSleepAESProcessEvent

SetHardwareInterrupt

(Non-blocking) v3.1x & v4.xx

Allocates an interrupt for an adapter card.

Syntax:

```
LONG SetHardwareInterrupt(
        LONG IRQNumber,
        void (*InterruptService)(void), or  LONG (*InterruptService)(void),
        LONG IntRTag,
        LONG ChainFlag,
        LONG ShareFlag,
        LONG *EOIFlag)
```

Return Value:

0	Success
non-zero	Conflicting options

Requirements: Interrupts disabled. May not be called from interrupt level.

Parameters:

IRQNumber — The hardware interrupt level.

InterruptService — Pointer to the interrupt service routine (ISR) that will be assigned to the specified interrupt. The service routine returns a value in a shared interrupt configuration.

IntRTag — The resource tag acquired by the driver initialization routine for interrupts.

ChainFlag — An indicator specifying whether the ISR is to be placed on the front or the back of the queue (only valid if the ShareFlag is set to a one). A value of 0 indicates placement at the front of the queue, while a value of 1 specifies placement at the rear of the queue.

ShareFlag — An indicator specifying if interrupts may be shared by the device (and driver). A value of zero specifies no sharing, and a value of 1 specifies interrupt sharing.

*EOIFlag — A pointer to a double-word. The OS uses this pointer to return a flag indicating that a second EOI is required for this interrupt (0=only one EOI required, 1=second EOI required). The function of this parameter is obsolete since all EOIs must now be handled indirectly through a call to *CDoEndOfInterrupt*. A NULL value may be substituted for the pointer.

SetHardwareInterrupt (continued)

Example:

```
mov         eax, cardNum                ;get adapter #
mov         edx, OFFSET EOITable        ;get table base
mov         ecx, eax
shl         ecx, 2                      ;create index
add         edx, ecx
push        edx                         ;extra EOI flag location
push        0                           ;share flag (0=no chain ints
                                        ;            1=chain ints)
push        0                           ;end of chain flag  (0=first,
                                        ;                    1=last)
push        IntRtag                     ;tag acquired for ints
mov         edx, DriverISRTable[eax*4]
push        edx                         ;provide ISR
mov         ebp, IOConfigTable[eax*4]   ;get IOConfig address
movzx       eax, [ebp].Interrupt0       ;get int #
push        eax                         ;pass
call        SetHardwareInterrupt        ;allocate interrupt
lea         esp, [esp + (6*4)]          ;adjust stack
```

Description: SetHardwareInterrupt allocates the specified interrupt and provides a driver ISR entry point (The OS fields the actual interrupt, saves all registers, sets up segment registers, calls the driver ISR as a near procedure, and issues the IRETD upon return). It also enables the interrupt at the priority interrupt controllers (PICs) and sets the corresponding bit in the RealModeInterruptMask.

See Also: ClearHardwareInterrupt, CAdjustRealModeInterruptMask, CUnAdjustRealModeInterruptMask, RegisterHardwareOptions, AllocateResourceTag

UnRegisterEventNotification (Blocking) v3.1x & v4.xx

Removes notification procedure from list called prior to system event occurrence

Syntax:
```
LONG UnRegisterEventNotification(
        LONG EventID);
```

Return Value:

0	Successful
-1	Invalid parameters

Requirements: Must be called from blocking process level only.

Parameters:

EventID	The 32 bit value (used to identify this notification procedure) returned by an earlier call to RegisterForEventNotification.

Example:

```
push    EventID                     ;ID from register call
call    UnRegisterEventNotification ;remove exit from list
lea     esp, [esp + 4]              ;adjust stack
```

Description: UnRegisterEventNotification removes notification procedure(s) from the list of procedures to be called by the OS prior to or following specific events in the OS. This is mandatory if a driver is being unloaded and a previous event notification was requested.

See Also: RegisterForEventNotification, Driver Unload

Support Routine Call Compatibility Summary
Device Driver Phases

Routine Name	Drivr Init	Drivr Check	Drivr Unloa	ScanF Devic	Delet Devic	Sleep Entry	NoSle Entry	IOPol Entry	IOCTL Poll	Intrp Entry
AddDiskDevice#				ONLY						
AddDiskSystem#	ONLY									
AlertDevice*						OK	OK	OK	OK	OK
Alloc*	OK			OK		OK	OK	OK	OK	
AllocateResourceTag#	ONLY!									
AllocBufferBelow16Meg#*	OK			OK		OK				
AllocSemiPermMemory*	OK			OK		OK	OK	OK	OK	
CAdjustRealModeInterruptMask*	OPT									
CancelNoSleepAESProcessEvent*	REQ		REQ	OK	OK	OK	OK	OK	OK	OK
CancelSleepAESProcessEvent*	REQ		REQ	OK	OK	OK	OK	OK	OK	OK
CCheckHardwareInterrupt*	OK		OK	OK	OK	OK	OK	OK	OK	OK
CDisableHardwareInterrupt*				OK	OK	OK	OK	OK	OK	OK
CDoEndOfInterrupt*										OK
CEnableHardwareInterrupt*			OK	OK	OK	OK	OK	OK	OK	OK
CheckDiskCard#		ONLY								
CheckDiskDevice#		ONLY								
ClearHardwareInterrupt*	REQ		REQ							
CPSemaphore#	ONLY									
CRescheduleLast#	OK		OK	OK	OK	OK				
CUnAdjustRealModeInterruptMask*	OPT		OPT							
CVSemaphore	ONLY									
CYieldIfNeeded#	OK		OK	OK	OK	OK				
CYieldWithDelay#	OK		OK	OK	OK	OK				
DelayMyself#	OK		OK	OK	OK	OK				
DeleteDiskDevice#			REQ		REQ	OK				
DeleteDiskSystem#	REQ		REQ							
DeRegisterHardwareOptions#*	REQ!		REQ!							
DoRealModeInterrupt#	ONLY									
EnterDebugger	OK		OK	OK	OK	OK				
Free*	OK		REQ	OK	OK	OK				
FreeBufferBelow16Meg*	OK		REQ	OK	OK	OK				
FreeSemiPermMemory*	OK		REQ	OK	OK	OK				
GetCurrentTime	OK		OK	OK	OK	OK	OK	OK	OK	OK
GetHardwareBusType	OPT									
GetIOCTL*			OK		OK	OK	OK	OK	OK	OK
GetReadAfterWriteVerifyStatus				REQ		OK	OK	OK	OK	OK
GetRealModeWorkSpace	ONLY									
GetRequest*			OK		OK	OK	OK	OK	OK	OK
GetSectorsPerCacheBuffer	OPT									
MapAbsoluteAddressToCodeOffset	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
MapAbsoluteAddressToDataOffset	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
MapCodeOffsetToAbsoluteAddress	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
MapDataOffsetToAbsoluteAddress	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
NetWareAlert	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
OutputToScreen#	ONLY									
ParseDriverParameters#	ONLY									
PutIOCTL*			OK		OK	OK	OK	OK	OK	OK
PutRequest*			OK		OK	OK	OK	OK	OK	OK
QueueSystemAlert	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
ReadPhysicalMemory#	OK									
RegisterForEventNotification#	ONLY									
RegisterHardwareOptions#*	ONLY									
RemoveDiskDevice#					REQ	OK				
ScheduleNoSleepAESProcessEvent*	OK			OK	OK	OK	OK	OK	OK	OK
ScheduleSleepAESProcessEvent*	OK			OK	OK	OK	OK	OK	OK	OK
SetHardwareInterrupt*	ONLY									
UnRegisterEventNotification#	OK		REQ							

LEGEND: REQ = Required here blank = Not Allowed ONLY = Allowed here only
OPT = Optional # = Blocks Thread * = Interrupts must be off here
! = Mandatory OK = Allowed here