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