Chapter 2: Device Driver Design

This chapter summarizes the components of a NetWare loadable device driver. The data structures and driver routines required to support this architecture are described below.

Data Structures

Drivers are required to create and maintain several data structures to interface with the Operating System. Most device driver's will require the following five structures:

- IOConfigurationStructure
- AdapterOptionStructure
- CardStructure
- DiskStructure
- AESEventStructure

The IOConfigurationStructure is defined by NetWare (one for each adapter card), and is required by the driver when the driver calls various NetWare support routines. The Adapter Options Structure is also defined by NetWare, and is required if the driver calls the ParseDriverParameters support routine (which fills out the options in the IOConfigurationStructure from the command line and/or interactively from the operator console). The Card and Disk structures are also required, but their contents (if any) are determined entirely by the drivers requirements. The AES structure is defined by NetWare, and is required to set up a timer for driver timeout recovery and other required functions.

The occurrence of each structure within the driver depends on the driver's hardware configuration. Each adapter board must have an associated IOConfigurationStructure and a CardStructure. Each storage device also requires a DiskStructure.

Device drivers are also required to work with two additional structures. These structures are defined by NetWare, and used by the Operating System to represent caller requests. The request structures are:

- IOCTLRequestStructure
- IORequestStructure

The IOCTLRequestStructure and IORequestStructure are defined by NetWare and are passed to the driver by the GetIOCTL and GetRequest system routines respectively, as described in Chapter 7.

Handles for the following structures are passed as parameters to some of the driver routines and NetWare driver support routines. The structures are defined in the OS and their size and configuration are transparent to the driver.

- NLMDefinitionStructure
- ScreenStructure
- SemaphoreStructure

Additional structures are optional and can be used as needed.

IOConfigurationStructure

A loadable device driver maintains an instance of the IOConfigurationStructure for each supported adapter board. The IOConfigurationStructure contains information about I/O ports, decode memory addresses and ranges, interrupts, and DMA channels. A driver uses the structure during initialization to reserve file server hardware resources (the structure may not be re-used for subsequent adapters).

Since a driver's initialization routine is typically called once for each adapter board, drivers can dynamically allocate each IOConfigurationStructure (if desired) by calling the Alloc support routine. The driver may also simply reserve the required space in its data segment for the structures.

The OS routines ParseDriverParameters, DeRegisterHardwareOptions, RegisterHardwareOptions, AddDiskSystem, and DeleteDiskSystem all require the IOConfigurationStructure. The support routine ParseDriverParameters may be called by the driver to fill out the necessary fields in the IOConfigurationStructure prior to registering it with the OS (the driver must supply a value indicating which options are required, described in Chapter 7 in the section describing "ParseDriverParameters" as the NeedBitMap). All fields of the IOConfigurationStructure must be zeroed prior to calling ParseDriverParameters, with the exceptions noted below:

```
IOConfigStruct  struc
    Reserved0           dd   ?
    Flags               dw   ?
    Slot                dw   ?
    IOPort0             dw   ?
    IOLength0           dw   ?
    IOPort1             dw   ?
    IOLength1           dw   ?
    MemoryDecode0       dd   ?
    MemoryLength0       dw   ?
    MemoryDecode1       dd   ?
    MemoryLength1       dw   ?
    Interrupt0          db   ?
    Interrupt1          db   ?
    DMAUsage0           db   ?
    DMAUsage1           db   ?
    IORTag              dd   ?
    Reserved1           dd   ?
    CmdLineOptionStr    dd   ?
    Reserved3           db   18 dup (?)
    LinearMemory0       dd   ?
    LinearMemory1       dd   ?
    Reserved4           db   8 dup (?)
IOConfigStruct  ends
```

```
typedef struct IOConfigurationStructure  {
    LONG Reserved0;
    WORD Flags;
    WORD Slot;
    WORD IOPort0;
    WORD IOLength0;
    WORD IOPort1;
    WORD IOLength1;
    LONG MemoryDecode0;
    WORD MemoryLength0;
    LONG MemoryDecode1;
    WORD MemoryLength1;
    BYTE Interrupt0;
    BYTE Interrupt1;
    BYTE DMAUsage0;
    BYTE DMAUsage1;
    LONG IORTag;
    LONG Reserved1;
    BYTE *CmdLineOptionStr;
    BYTE Reserved3[18];
    LONG LinearMemory0;
    LONG LinearMemory1;
    BYTE Reserved4[8];
    } IOConfigStruct;
```

Figure 2-1 IOConfigurationStructure for NetWare v4.xx

```
IOConfigStruct  struc
    Reserved0       dd   ?
    Flags           dw   ?
    Slot            dw   ?
    IOPort0         dw   ?
    IOLength0       dw   ?
    IOPort1         dw   ?
    IOLength1       dw   ?
    MemoryDecode0   dd   ?
    MemoryLength0   dw   ?
    MemoryDecode1   dd   ?
    MemoryLength1   dw   ?
    Interrupt0      db   ?
    Interrupt1      db   ?
    DMAUsage0       db   ?
    DMAUsage1       db   ?
    IORTag          dd   ?
    Reserved1       dd   ?
    Reserved2       dd   ?
    Reserved3       db   18 dup (?)
    Reserved4       db   16 dup (?)
IOConfigStruct  ends
```

```
typedef struct IOConfigurationStructure  {
    LONG Reserved0;
    WORD Flags;
    WORD Slot;
    WORD IOPort0;
    WORD IOLength0;
    WORD IOPort1;
    WORD IOLength1;
    LONG MemoryDecode0;
    WORD MemoryLength0;
    LONG MemoryDecode1;
    WORD MemoryLength1;
    BYTE Interrupt0;
    BYTE Interrupt1;
    BYTE DMAUsage0;
    BYTE DMAUsage1;
    LONG IORTag;
    LONG Reserved1;
    LONG Reserved2;
    BYTE Reserved3[18];
    BYTE Reserved4[16];
} IOConfigStruct;
```

Figure 2-2 IOConfigurationStructure for NetWare v3.1x

Each IOConfigurationStructure field is described below:

Reserved0 — Reserved by NetWare.

Flags — Various features or characteristics of the driver are enabled by setting the correct bits (flags) in this field as described below. This field must be set before calling ParseDriverParameters routine.

IODetached	01h

Bits 02h - 100h must be set when the corresponding hardware options (I/O device addresses, interrupts, etc.) are to be shared by all the adapter cards linked to a single driver.

IOSharablePort0Bit	02h
IOSharablePort1Bit	04h
IOSharableMem0Bit	08h
IOSharableMem1Bit	10h
IOSharableInt0Bit	20h
IOSharableInt1Bit	40h
IOSharableDMA0Bit	80h
IOSharableDMA1Bit	100h

v4.xx only - Bit 200h must be set if custom command line options pointed to by the **CmdLineOptionStr** field are to be included in the **STARTUP.NCF** file.

IOCmdLineOptionsBit 200h

v4.xx only - Bit 400h must be set if the standard hardware options defined in IOConfigurationStructure (I/O device addresses, interrupts, DMA, etc.) are **not** to be included in the **STARTUP.NCF** file. (This bit does not affect the custom command line options.) (Note: Any valid value in the **Slot** field will cause all other standard hardware options to be excluded from the **STARTUP.NCF** file.)

IONoHardwareOptionsBit 400h

Slot	For adapter boards running in PS/2 and EISA machines, this field holds the slot number where the board is installed.
IOPort0	Contains the primary base I/O port for this adapter board.
IOLength0	Contains the number of I/O ports starting at IOPort0.
IOPort1	Contains the secondary base I/O port for this adapter board.
IOLength1	Contains the number of I/O ports starting at IOPort1.
MemoryDecode0	Must contain the primary shared memory **absolute** address used by the adapter board (if any).
MemoryLength0	Contains the amount of memory (in paragraphs) that the adapter board uses starting at MemoryDecode0.
MemoryDecode1	Must contain the secondary shared memory **absolute** address used by the adapter board (if any).
MemoryLength1	Contains the amount of memory (in paragraphs) that the adapter board uses starting at MemoryDecode1.
Interrupt0	Contains the primary interrupt vector number. This field must be initialized to FFh if it is unused.
Interrupt1	Contains the secondary interrupt vector number. This field must be initialized to FFh if it is unused.
DMAUsage0	Contains the primary DMA channel used by the adapter board. This field must be initialized to FFh if it is unused.
DMAUsage1	Contains the secondary DMA channel used by the adapter board. This field must be initialized to FFh if it is unused.

IORTag — The *IORegistrationSignature* resource tag must be placed in this field before any calls can be made to ParseDriverParameters or RegisterHardwareOptions.

Reserved1 — Reserved by NetWare.

CmdLineOptionStr — v4.xx only - Pointer to linked CmdLineOptionStruct structures that contain custom command line options in string format to be included in the STARTUP.NCF. If this field is to be used, bits 9 and 10 of the Flag field need to be set appropriately. The structures have the following format.

```
CmdLineOptionStruct     struc
    Link                dd    ?
    OptionStr           dd    ?
CmdLineOptionStruct     ends
```

```
typedef struct CmdLineOptionStructure  {
        CmdLineOptionStructure   *Link;
        char *OptionStr;
        } CmdLineOptionStruct;
```

Link	Pointer to the next structure in the list.
OptionStr	NULL terminated string that holds a custom command line option to be included in the STARTUP.NCF file.

Reserved2 — v3.1x only - Reserved by NetWare.

Reserved3 — Reserved by NetWare.

LinearMemory0 — v4.xx only - The *RegisterHardwareOptions* support routine fills this field with the **linear** (logical) address translation of the absolute address in MemoryDecode0.

LinearMemory1 — v4.xx only - The *RegisterHardwareOptions* support routine fills this field with the **linear** (logical) address translation of the absolute address in MemoryDecode1.

Reserved4 — Reserved by NetWare.

AdapterOptionStructure

In order to use the NetWare support routine *ParseDriverParameters* to parse the command line, a device driver must maintain a single instance (or more if more than one adapter type is supported by the same driver) of the AdapterOptionStructure (See *ParseDriverParameters* in Chapter 7). This structure serves as a template defining the available choices for various adapter options. *ParseDriverParameters* uses this template to query the operator, validate entries, and to fill in the associated fields in the supplied IOConfigurationStructure.

```
AdapterOptionStruct  struc
    IOSlot              dd  ?
    IOPort0             dd  ?
    IOLength0           dd  ?
    IOPort1             dd  ?
    IOLength1           dd  ?
    MemoryDecode0       dd  ?
    MemoryLength0       dd  ?
    MemoryDecode1       dd  ?
    MemoryLength1       dd  ?
    Interrupt0          dd  ?
    Interrupt1          dd  ?
    DMA0                dd  ?
    DMA1                dd  ?
AdapterOptionStruct  ends
```

```
typedef struct AdapterOptionDefinitionStructure {
    LONG *IOSlot;
    LONG *IOPort0;
    LONG *IOLength0;
    LONG *IOPort1;
    LONG *IOLength1;
    LONG *MemoryDecode0;
    LONG *MemoryLength0;
    LONG *MemoryDecode1;
    LONG *MemoryLength1;
    LONG *Interrupt0;
    LONG *Interrupt1;
    LONG *DMA0;
    LONG *DMA1;
} AdapterOptionStruct;
```

Figure 2-3 Defining an Adapter Option Structure

Each field in the structure is a double-word pointer to a length-preceded list of parameters. Each list assumes the following form:

```
List    dd      n           ;number of entries          LONG List[n]
        dd      entry1      ;first (default) value
        dd      entry2
                .
                .
        dd      entryn      ;last available value
```

Figure 2-4 Defining an Options Parameter List

If entries are not used in the AdapterOptionStructure, the pointer to the associated list is allowed to be replaced by a zero value. The fields are explained in more detail in the IOConfigurationStructure section above.

CardStructure

As with the IOConfigurationStructure, a driver maintains a CardStructure for each supported adapter board. This structure is especially helpful in creating re-entrant drivers, making card information easy to access without complicating stack management. The size and content of this structure is defined by the needs of the driver.

The Card Structure is not allocated by the driver's initialization routine. Instead, the initialization routine passes NetWare the requested size of the structure when the initialization routine calls AddDiskSystem. AddDiskSystem then allocates memory for the structure, zeros it, and returns a pointer to the CardStructure back to the driver. If a driver does not require any information to be kept in this structure, it should supply a zero as the requested structure size when calling AddDiskSystem. The driver must retain the structure handle passed back by AddDiskSystem, even if the structure size requested was zero (many driver support routines require the CardStructure pointer as a parameter).

Note: Tape or other device drivers must also use the AddDiskSystem call to receive the proper information during initialization.

```
CardStruct    struc

              .
              .
  (driver defined)
              .
              .

CardStruct    ends
```

```
typedef struct CardStructure {

              .
              .
  (driver defined)
              .
              .

} CardStruct;
```

Figure 2-5 Driver-defined CardStructure

The format of the CardStructure is undefined. Drivers may utilize this structure according to their requirements. The structure should include all information required for one adapter board. An IOConfigurationStructure pointer, along with a list of registered DiskStructures, etc., are usually appropriate to keep in a CardStructure.

DiskStructure

A device driver maintains a DiskStructure for each supported storage device. A driver's disk (device) initialization routine (Scan For Devices) allocates this structure by calling AddDiskDevice and passing the requested SIZE of the DiskStructure as one of the parameters. AddDiskDevice then allocates memory for the structure, zeros it, and returns a pointer to the allocated memory. If a driver does not require this structure, it should pass a zero as the requested size, and must retain the pointer returned (required by many system support routines).

Note: Tape or other device drivers must also use the AddDiskSystem call to receive the proper information during initialization.

```
DiskStruct    struc
          .
          .
 (driver defined)
          .
          .
DiskStruct    ends
```

```
typedef struct DiskStructure {
          .
          .
 (driver defined)
          .
          .
} DiskStruct;
```

Figure 2-6 Driver-defined DiskStructure

Like the CardStructure, the format of the DiskStructure is undefined. The driver may define use and contents of the structure as required. It will at least need to contain a pointer to the associated CardStructure or IOConfigurationStructure. Normally the DiskStructure should include any information the driver needs to maintain information corresponding to one disk or mass storage device, such as retry counters, pending operations, etc.

AESEventStructure

A device driver maintains an AES structure for each separate time-out entry point desired (it is not suggested that the driver use many timers, as each active timer may require additional system overhead while processing clock interrupts). The AES structure must be initialized before calling ScheduleNoSleepAESProcessEvent. Drivers must place the resource tag acquired during the first part of the initialize driver module into the AES structure, so that the Operating System can track the resource.

```
AESEventStructure     struc
        AESReserved0          dd    ?
        AESWakeUpInterval     dd    ?
        AESReserved1          dd    ?
        AESProcessToCall      dd    ?
        AESResourceTag        dd    ?
        AESReserved2          dd    ?
AESEventStructure     ends
```

```
typedef struct AESEventStructure {
        LONG AESReserved0;
        LONG AESWakeUpInterval;
        LONG AESReserved1;
        void (*AESProcessToCall)(
              AESEventStruct *EventPtr);
        LONG AESResourceTag;
        LONG AESReserved2;
        } AESEventStruct;
```

Figure 2-7 AESEvent Structure definition

Each field in the AES Event structure is defined below:

AESReserved0 — Reserved by NetWare.

AESWakeUpInterval — Indicates the time interval for waking up your Timeout routine in system clock ticks (approximately 1/18.2 second per tick). Generally, this interval should be small enough to provide reasonable recovery time, but not so small as to affect overall server performance.

AESReserved1 — Reserved by NetWare.

AESProcessToCall — A pointer to the routine that will be called once for each ScheduleNoSleepAESProcessEvent call. A pointer to the AESEvent structure (**EventPtr**) is passed as a parameter to the routine. Additional information required by the driver may be passed to the AESEvent by appending it to the end of the structure.

AESResourceTag — Resource tag returned by AllocateResourceTag when called with the AESProcessSignature during initialization. This field must be initialized prior to making the call to ScheduleNoSleepAESProcessEvent.

AESReserved2 — Reserved by NetWare.

IOCTLRequestStructure

The IOCTLRequestStructure is defined by the Operating System, and a structure built by the OS for each IOCTL request. This structure holds the information necessary for processing special calls made to the Driver for IOCTL functions. The definition of the IOCTLRequestStructure is as follows:

```
IOCTLRequestStruct  struc
    DriverLink          dd  ?
    CardHandle          dd  ?
    CompletionCode      dw  ?
    Function            db  ?
    SubFunction         db  ?
    Parameter           dd  ?
    IOCTLBuffer         dd  ?
IOCTLRequestStruct  ends
```

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

Figure 2-8 An IOCTLRequest Structure

Each field in the IOCTL Request structure is defined below.

DriverLink — This field is used only by the driver. It can be used to link the outstanding IOCTL requests at the driver level. This field is not used by the OS.

CardHandle — This field contains a card handle. This is the same value that AddDiskSystem returned during initialization.

CompletionCode — The driver fills in this field before returning the IOCTL Request structure to the application or NetWare (see Chapter 5 for a complete list of possible completion status codes).

Specific error conditions should be returned using codes appropriate to the calling application. Before the request is returned, the driver may use this field locally.

Function	This field specifies the requested IOCTL function (The IOCTL functions are discussed in detail in Chapter 5).
SubFunction	This field has the required IOCTL subfunction which may indicate the actual function to be performed by the IOCTL (IOCTL request Subfunctions are explained in Chapter 5).
Parameter	This field is often used to specify a DiskStructure address. When used as such, this value is the same as that returned by AddDiskDevice when registering a new device. Other values could also be passed in this field as needed.
IOCTLBuffer	This field, if used, provides a pointer to additional request information (zero if not used). For more information, see Chapter 5.

As explained in the previous section, individual IOCTL requests are specified by the function field in the IOCTL Request structure. If the request can perform more than one task, the function is given in the subfunction field. Figure 2-8 shows the defined IOCTL functions.

0-3	*Novell Assigned (General IOCTLs)*
4-63	*Novell Reserved*
64-255	*IOCTLs assigned to Developers*

Figure 2-9 IOCTL Number Assignments

Novell has reserved IOCTL functions 0 through 63. IOCTLs 64 and up will be assigned by Novell to developers upon request. Novell assigns certified drivers a Driver ID. If another loadable module needs the driver to perform a special IOCTL service, this value could be used as a function number in the IOCTL poll procedure.

IORequestStructure

The IORequestStructure is defined by the Operating System, and a structure is built by the OS for each request. The structure holds information necessary for processing standard I/O calls, and is defined as follows:

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

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

Figure 2-10 An IORequest Structure

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

DriverLink — Previously, this parameter was not used by the OS, and could be used by the driver as a link field, information holder, etc. However, in NetWare v3.11 and subsequent versions, this field contains the handle of a request available on the queue (if any) that is contiguous to the current one. This allows a driver to combine requests. (This is desirable if the device can access multiple blocks of storage at a time.) The field will be zero if no contiguous request is available. Be aware that application NLMs that bypass the OS cache by making direct I/O requests may invalidate this field. Always verify the continuity of the requests. The driver may choose to ignore this information and use the field as before.

DiskHandle — This parameter specifies the target device or disk. This is the same value as that returned by the NetWare routine *AddDiskDevice*. This field is not valid until after the request has been obtained using the *GetRequest* routine.

CompletionCode — The driver fills in this parameter before posting completion of the I/O Request structure to NetWare (see Chapter 6 for the complete list of possible completion codes).

Function

This parameter byte specifies the function number of the I/O request being issued.

The function codes are defined as follows:

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	
Single Set Mark(s)	08h
Write single set mark(s)	
Space forward single set mark(s)	
Space backwards single set mark(s)	
Consecutive Set 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	
Partition Operations	0Ch
Format to partition media	
Select partition	
Return number of partitions	
Return partition size	
Return max. number of partitions that can be defined	
Physical Media Operations	0Dh
Quick erase partition	
Rewind partition	
Go to end of partition	
Security erase partition	
Retention media	
Goto end of partition	
Random Erase	0Eh
Reserved	0Fh-3Fh

Parameter1	Function dependent parameter
Parameter2	Function dependent parameter
Parameter3	Function dependent parameter

Driver Routines

This section summarizes the major routines a driver must provide to interface with NetWare. These routines can be grouped under four headings:

- The Loadable Module Interface
- The Mass Storage Administrative Interface
- IOCTL (I/O Control) Operations Interface
- I/O Operations Interface

Each interface is summarized below. Chapters 3, 4, 5, and 6 discuss these interfaces in greater detail, as well as describing some suggested approaches.

The Loadable Module Interface

A driver must comply with the NetWare Loadable Module (NLM) interface by defining three routine entry points: initialize driver, check driver, and remove driver.

InitializeDriver

The application-defined InitializeDriver routine is called each time a "LOAD" command is issued for the driver. Drivers typically are written so that a load command must be issued for each host adapter. Drivers may recognize and register all associated cards with a single load command (during the InitializeDriver routine). Drivers should also allow the operator to load the driver with a single specified adapter. This will allow the operator to selectively enable only desired host adapters for error determination, etc. (The driver could be written to recognize all adapters only if command line does not specify an adapter.)

InitializeDriver validates and sets up hardware considerations like I/O ports, interrupts, and DMA channels, AES timers, creates structures for adapters and drives, exchanges information with NetWare, and initializes the adapter board. The initialize driver routine may indicate failure to set up an adapter card upon return to its caller. This will cause the system to unload the driver code unless the driver is re-entrant and has been successfully loaded previously.

CheckDriver

The NetWare console command UNLOAD calls the application-defined CheckDriver routine as a precaution before unloading the driver. This routine checks each supported storage device to see if the device is locked by another process. If the device is locked, NetWare displays a warning message on the file server console and allows the console operator to abort the UNLOAD process if desired.

If the device is locked and the operator continues to unload after a warning message, then NetWare will call the RemoveDriver routine (see below).

RemoveDriver

NetWare calls the application-defined RemoveDriver routine only once. RemoveDriver removes the entire driver (code and data images) from file server memory. RemoveDriver does the following:

- Removes (or causes to be removed) all instances of DiskStructure (after associated flush of requests) and CardStructure
- Returns all file server resources like memory, interrupts, I/O ports, and AES timers
- Removes the driver from memory by returning to the caller.

Note: CheckDriver and RemoveDriver are **not** called after a console operator issues a "down" command.

The Mass Storage Control Interface

The Mass Storage Control Interface provides two procedures which are registered with the OS through the AddDiskSystem call. The routines recognize new devices, register them with the Operating System, and remove the devices from the OS when appropriate. The routines are:

- ScanForDevices
- DeleteDevice

ScanForDevices

The ScanForDevices routine is a driver routine called by NetWare to request the driver to look at its configuration and adapters, determine if un-registered devices are attached, and register them. The routine is called at blocking process level, and thus may make any calls to blocking routines such as AddDiskDevice (which is required to register each device).

DO NOT remain in this routine for a significant length of time while waiting for responses from devices. The OS limits the time drivers spend in process level (without causing a task switch or process switch) to approximately 250 milliseconds.

Upon completion of the scan (and registration of any new devices found), the driver must return to the caller. Error conditions encountered are ignored, and it is not possible to return an error code to the caller from this routine. See chapter 4 for more detail regarding this procedure.

DeleteDevice

The NetWare OS deactivates devices that return an I/O request with a Non-Media Error completion code but does not delete them from the system. All corresponding requests on the internal queue are posted with an error status. At a later time, however, the OS may attempt to reactivate the device. If reactivation is not desirable, the driver may choose to remove the device structure from the system. *DeleteDevice* is called for this purpose. This routine processes all pending drive requests, calls *RemoveDiskDevice*, unhooks the DiskStructure from the CardStructure, etc, (driver clean-up), and then calls *DeleteDiskDevice*.

In NetWare v3.11 *DeleteDevice* is registered with the OS though the *AddDiskSystem* call. When the device drives determines that the device should be removed it informs the OS using an *AlertDevice* call and the proper parameter. The OS will then call the *DeleteDevice* routine.

In NetWare v4.xx and v3.1x (excluding v3.11) registration of the routine has been dropped, and the device driver should call the routine directly from a blocking environment.

IOCTL (I/O Control) Operations Interface

The IOCTL Notification (IOCTLPoll) routine is the entry point for the driver's I/O control subroutines. The NetWare OS and loadable modules can access the driver's IOCTL routines by passing an IOCTL Request structure to this routine that contains a function and a subfunction number.

NetWare calls the IOCTLPoll routine to notify the driver that an IOCTL request has been issued. The driver IOCTL processing code should do two things:

- Issue a GetIOCTL to "acquire" the request
- Make a PutIOCTL call to post completion of the IOCTL function request

GetIOCTL may be called with the request handle indicated in the IOCTLPoll entry (specific GetIOCTL), or may be called with zero for a request handle, in which case the next sequential IOCTL request will be obtained. The IOCTL poll routine may elect to issue a GetIOCTL call to acquire the IOCTL request and initiate an operation, or it may elect to simply return to the caller, making a call to GetIOCTL at a later time.

The IOCTLPoll routine may also examine the IOCTL request, determine that the function is not supported by the driver, issue a GetIOCTL call, then issue a PutIOCTL call to post a "not supported" return status (Return status for IOCTLs are defined in Chapter 5).

IOCTL requests are not placed on the elevator queues, but are placed on sequential non-sorted IOCTL queues (one per adapter card), and thus are not queued in the same way that I/O requests are queued. The IOCTL Poll routine is called at non-blocking process time, and thus may not make calls to any blocking routines (any routines indicated in Chapter 7 as non-blocking may be called).

If the IOCTL can be serviced at the time of the IOCTLPoll call, the IOCTLPoll code examines the request structure and routes the request to the proper IOCTL subroutine depending on the function and subfunction numbers. When the IOCTL routine has completed the function, the driver fills in a completion code field , returns the IOCTL Request structure to the calling module by making a call to PutIOCTL, then returns to the caller or point of interrupt.

I/O Operations Interface

I/O operations include the IOPoll, timeout, and interrupt service routines. The entry point to each routine is passed as a parameter to, respectively, the AddDiskDevice, ScheduleNoSleepAESProcessEvent, and SetHardwareInterrupt calls support routines.

IOPoll

NetWare calls the I/O Notification (IOPoll) routine to notify the driver that NetWare issued an I/O request. NetWare passes a pointer to a DiskStructure and an I/O Request structure to the driver IOPoll routine. The IOPoll routine determines if it can "acquire" another request, then makes a call to GetRequest to obtain the next I/O request structure.

Once IOPoll obtains the request, the routine validates the request, executes a write or a read, fills in a completion status code field, returns the I/O request structure to NetWare by making a call to PutRequest, and returns to the caller or point of interrupt (most drivers may need to see if other operations should be started prior to actual return).

The IOPoll entry is called at non-blocking process level, and thus may not make any calls to blocking routines. It is possible to simply take the requests in sorted (by block address) order by supplying a zero instead of the request handle which is provided by the caller. This allows the driver to get the requests in the order the OS has placed them in on the Disk Elevator queues (dual queues, both in ascending order).

It is also possible for a driver to do the requests in the specific order issued (typically for a tape or sequential device), simply by allocating a queue for each device, then placing the request handles sequentially in the device queue immediately upon entry to the IOPoll or IOCTLPoll routines, along with an flag indicating the kind of request (I/O or IOCTL).

When a new request can be started, the driver would then obtain the next request handle (structure pointer) from its own queue, issue a GetRequest or GetIOCTL to obtain the request, then start the indicated request. This routine must operate in concert with the driver ISR to schedule I/O requests and keep the operations flowing (see Chapter 6 for further details).

Timeout

A driver timeout routine is an essential driver routine that may be scheduled for periodic entry as an asynchronous event. It checks adapter boards and determines the amount of time that has elapsed since the card received a request. If the time exceeds an established limit, the Timeout routine may take over and determine if an error condition has occurred.

This routine may also scan to see if any requests may need to be started on devices attached to the card. Setting up a timer to scan for failed operations is essential, since the server must not cease to operate because of an operation that is hung or "never completes" (see Chapter 6 for further details). This routine or routines is called at non-blocking process time as long as the driver made a call to ScheduleNoSleepAESProcessEvent. Timeout exits may be specified to be blocking by using ScheduleSleepAESProcessEvent instead.

Other AESEvents may also be scheduled to handled tasks that cannot be completed in the current environment. For instance, if in an ISR a need arises to make a call to a blocking routine, which is illegal, a blocking AESEvent may be scheduled with a zero time delay to handle the call.

Interrupt Service Routine

For all hardware interrupts, the system Interrupt Service Routine (ISR) receives the interrupt and calls the driver's ISR. A driver requires a separate ISR entry point for each card the driver will support. Portions of the I/O Poll routine may be duplicated in the ISR if desired.

The driver ISR is responsible for the following:

- resetting the interrupt on the adapter card
- clearing the interrupt flag in the system interrupt controller(s). (You **must** call *CDoEndOfInterrupt* to accomplish this step)
- finishing any required part of the function that the driver must do
  (i.e. moving data to a system buffer
  canceling a timer for the operation just completed,
  retrying operations which did not complete successfully,
  posting completion status for operations which completed successfully
  or have exceeded their retry limit,
  looking for further operations to initiate for other requests,
  returning to the caller (the system ISR)

Much of the driver ISR may run with interrupts inhibited due to its function within the driver architecture. Under no circumstances is a driver ISR allowed to make any calls to blocking routines. See Chapter 6 for more specific detail on the driver ISR and its requirements.

Required Device Driver Routines

Routine *	**Definition Point**
InitializeDriver	DRIVER.DEF
CheckDriver	DRIVER.DEF
RemoveDriver	DRIVER.DEF
ScanForDevices	AddDiskSystem
DeleteDevice **	AddDiskSystem
IOCTLPoll	AddDiskSystem
IOPoll	AddDiskDevice
InterruptServiceRoutine (ISR)	SetHardwareInterrupt
TimeOut	ScheduleNoSleepAESProcessEvent *or* ScheduleSleepAESProcessEvent

* = These routine names are not mandatory; developers may choose their own routine names.

** = This routine is required in NetWare v3.1x and is optional and known only to the driver in NetWare v4.xx.