Chapter 3: Loadable Module Interface

As an NLM, drivers must provide three routines that interface with NetWare's loadable module design. These routines must do the following:

A) Initialize Driver - Perform driver allocation and initialization for the structures and hardware related to an adapter card.

B) Check Driver - Check whether any of the devices supported by the driver are actively being used by the operating system, and report their status.

C) Unload Driver - Remove all requests, device and adapter card structures, cancel all timers and operations, and remove the driver from the file server upon request.

Note that A is entered once for **each** "LOAD" command, but B and C are entered **only** once for everything configured for the driver.

Initializing the Driver

The driver must provide an *InitializeDriver* routine which the OS calls when the driver is loaded. *InitializeDriver* must perform the following tasks:

- Validate the command line and Reserve hardware options
- Initialize and register cards and structures
- Load Co-processor firmware from the custom data file, if required.
- Allocate interrupts
- Schedule AES timeout events for error recovery
- If the driver initialization fails:
  - Release allocated hardware options
  - Release interrupts
  - Cancel any AES timers
  - Return any structures or memory allocated
- Return driver initialization status to the OS

Validating and Reserving Hardware Options

The *InitializeDriver* routine must reserve and register the appropriate hardware options for each card. The OS is informed of the hardware options in an *IOConfigurationStructure* passed to *RegisterHardwareOptions.*

The driver may statically allocate a specific number of I/O configuration structures in its data area, or they may be dynamically allocated using the NetWare routine Alloc. For more information about the *IOConfigurationStructure*, see "IOConfigurationStructure" in Chapter 2. The routine *ParseDriverParameters* may be used to obtain I/O configuration parameter information from the command line at the time the driver is loaded, or to query the operator for parameters which are not specified on the command line. The driver can obtain most of this information from the system in either MCA or EISA machines, provided that the slot number is provided in the command line.

The *AdapterOptionsStructure* indicates the acceptable values for your driver's hardware configuration. This structure is usually hard-coded and provides pointers to tables containing options, ordered with defaults first (see the "AdapterOptionStructure" in Chapter 2). The hardware requirements bit map (*needBitMap* parameter specified in *ParseDriverParameters* description in Chapter 7) indicates the particular hardware requirements which *ParseDriverParameters* routine must obtain for the card.

ParseDriverParameters uses the *AdapterOptionsStructure* to provide the user with a list of values to chose from. *ParseDriverParameters* also checks this list against current configurations to resolve any conflicts. It then returns the selected values in the card's *IOConfigurationStructure.* For more information about *ParseDriverParameters*, see Chapter 7. *ParseDriverParameters* requires a pointer to the driver's command line, a pointer to the card's *IOConfigurationStructure*, a hardware requirements bit map, a pointer to the driver's *AdapterOptionsStructure*, and a pointer to the driver's screen handle.

Both the command line pointer and the screen handle are passed to *InitializeDriver* on the stack at the time the routine is called by NetWare. When the driver calls *ParseDriverParameters*, these values must be passed to *ParseDriverParameters* so that it can prompt the user for required configuration information at the console. *InitializeDriver* can also use the screen handle to display error messages (the screen handle becomes invalid upon return to the initialize driver caller).

In order to facilitate future developments in the OS, the driver can optionally provide code to parse the command line in order to obtain a system administrator assigned HBA card number and any other parameters. The card number is passed as a parameter in the *AddDiskDevice* support routine in order to give an explicit address to the devices installed on the card.

Initializing and Registering Cards and Structures

The driver can only initialize the host adapter card and register it with the OS after the necessary hardware options have been validated and reserved (the driver must never issue any instructions to any I/O ports, etc., until this has been completed). The procedure for initializing the card depends entirely on the requirements for that particular card. NetWare places no specific requirements on card initialization.

The NetWare routine AddDiskSystem registers a card with NetWare. When making this call, the driver must pass NetWare a pointer to the card's IOConfiguration structure. The required size of the Card Structure must be passed also, even if it is zero. NetWare will then allocate memory for the Card Structure. After the card is registered, the driver can fill out the Card structure as required. The driver must save a pointer to the Card structure for later reference. For more information about Card structures, see Chapter 2.

If an error occurs during registration, the driver must remove the allocated hardware options by calling DeRegisterHardwareOptions. The driver must also free any interrupts allocated and return any memory that has been allocated. To do so, the driver must call Free. The driver should disable the adapter so that it cannot interfere with server operation.

Loading Co-processor firmware

NetWare allows drivers to read custom data into system memory during initialization. This data can be anything that might be required by a driver. For example, the driver may need to read in firmware to be loaded into a co-processor board. To define the custom file, use the CUSTOM keyword in the driver's definition file followed by the file's name (custom files are simply appended to the driver .dsk file). NetWare passes the custom data file's handle, starting offset, size, and the Read routine address to the Initialize Driver routine, where it should be saved by the driver upon entry. Initialize Driver can read the file into memory by calling the Read routine using the syntax shown below in Figure 3-1.

```
ReadCustomDataRoutine(
    LONG CustomDataFileHandle,
    LONG CustomDataOffset,
    BYTE *CustomDataDestination,
    LONG CustomDataSize);
```

Figure 3-1 ReadCustomDataRoutine Syntax

The driver must supply the destination in memory according to the needs of the adapter card. The read routine executes byte moves to transfer the data from the file to the supplied logical destination address. Since many adapter cards only support word moves to or from shared RAM, it may be necessary to transfer the data to an allocated block of memory before moving it manually to the shared RAM in the adapter card. The read routine reads a maximum of 4K in one read operation. If the custom data file is larger than 4K bytes, it should be read in multiple read operations, adjusting the offset and requested amount as required for each call. The read routine will return an error code if the driver requests a read beyond the end of the custom data. The actual length of the custom data file is passed on the call to the driver initialization, and should be used to read the exact amount. The data read from the custom data file is not interpreted in any way by NetWare, and may be in any form desired by the driver. The custom data file will normally be actual raw machine code for a co-processor card, to be directly loaded onto the card, and may be prepared in any way desired, using any language processors or linkers desired.

Allocating Interrupts

InitializeDriver routines must allocate requested interrupts by calling *SetHardwareInterrupt*. If the driver supports a host adapter also used for DOS and will be used to load NLMs, etc. for NetWare, the driver must also make a call to *CAdjustRealModeInterruptMask.* Interrupts can be sharable or non-sharable. If an interrupt is shareable, it must be indicated as such both in the IOConfigurationStructure registered with the OS and by a parameter passed to the *SetHardwareInterrupt* routine. The driver interrupt service routine (ISR) must also provide logic for handling shared interrupts, which must determine if an ISR entry is for an adapter associated with the driver, and must return this indication back to the OS (the driver ISR caller).

For further information on allocating interrupts and interrupt options, see *SetHardwareInterrupt* in Chapter 7 or the section on "Interrupt Service Routines" in chapter 6. If an error in registering a Card occurs after an interrupt has been allocated, the interrupt must be deallocated by calling *ClearHardwareInterrupt.*

Scheduling an AES Timeout Event

Drivers should use the support routine ScheduleNoSleepAESProcessEvent to schedule an asynchronous timeout event. The timeout event must guard the system from the card's failure. After the card is up and running, the timeout event monitors the card's performance. If a significant delay occurs in the card's operation, the timeout event may intervene and cause a retry, or post completion to the OS indicating an error occurred. For more information about the timeout event, see Chapter 6. If an error is detected during registration of the card, any timer previously made active must be de-activated by calling CancelNoSleepAESProcessEvent.

The AES structure should be allocated at initialization, and must have a valid AES resource tag (also acquired at initialization) placed in it prior to a call to schedule the AES event.

Reporting Errors

Errors that occur during Initialize Driver can be reported at the console using the support routine *OutputToScreen* or optionally *NetwareAlert* (v4.xx) or *QueueSystemAlert* (v3.1x) (see Chapter 7). The driver is passed the screen handle when NetWare calls the driver's Initialize Driver routine. The handle should be saved by the driver for use only during the driver initialization routine.

Note: *NetWareAlert* and *QueueSystemAlert* can be called at any time from any level of execution, including from an ISR, and does not require a Screen Handle.

Initialize Driver

The Initialize Driver routine initializes the driver and performs the necessary operations for activating an adapter card.

Syntax

```
LONG     InitializeDriver(
         LONG NLMHandle,
         LONG ScreenHandle,
         BYTE *LoadCommandLine,
         LONG Reserved0,
         LONG Reserved1,
         LONG CustomDataFileHandle,
         LONG (*ReadCustomDataRoutine)(
            LONG CustomDataFileHandle, LONG CustomDataOffset,
            BYTE *CustomDataDestination, LONG CustomDataSize),
         LONG CustomDataOffset,
         LONG CustomDataSize);
```

Return Values

0	Success
non-zero	Failure

Parameters

NLMHandle	Receives a 4-byte loadable module handle. The Initialize Driver routine needs this value to call many of the NetWare driver support functions.
ScreenHandle	Receives a 4-byte screen handle. If an error occurs during initialization, the Initialize Driver routine can use this value to call the NetWare routine OutputToScreen and display an error message. Note: Once the Initialize Driver routine returns, the disk driver can no longer use this screen handle.
LoadCommandLine	Receives a 4-byte pointer to the command line. During initialization, the driver can pass this value to ParseDriverParameters to fetch hardware configuration information from the command line.
Reserved0	Reserved by NetWare
Reserved1	Reserved by NetWare
CustomDataFileHandle	Passes a 4-byte pointer to the custom data file.
ReadCustomDataRoutine	Receives a pointer to NetWare read-file routine that can read a custom data file.
CustomDataOffset	Receives the starting offset of the custom data file.
CustomDataSize	Receives the length of the custom data file.

Remarks

The exact name given the Initialize Driver routine is arbitrary. NetWare finds the name of a disk driver's Initialization routine in the .DSK file's header. This name must be included in the driver's .DEF file by using the START keyword. Using the .DEF file, the

Initialization Sequence Summary

The following is a suggested sequence to follow in performing driver initialization for a host adapter card (Some steps are optional):

1) Save parameters passed from the OS, including Driver Handle, Initialization Screen Handle and command line buffer pointer (only valid during initialization), Custom Data File Handle, custom data file offset, and custom data file size.

2) Allocate all resource tags required by the driver for the resource types that will be allocated and place in associated structures (see AllocateResourceTag in chapter 7).

3) Verify that the operator is not attempting to load more adapters than the driver will support. If there is an error, go to error step #6.

4) (optional) Call *GetHardwareBusType* to determine the bus type of the processor. This is only required for drivers which can support cards on multiple bus types.

5) (optional) Allocate memory required for the IOConfigurationStructure and any other structure by calling *Alloc* with the desired structure size, as defined in chapter 2. (IOConfigurationStructures can be statically defined in the Driver data segment if desired.) If there is an error, go to error step #6.

6) Initialize the IOConfiguration Structure. (see chapter 2).

7) Place the obtained I/O resource tag in the IOConfigurationStructure.

8) Call *ParseDriverParameters* with the appropriate NeedBitMap value and AdapterOptionStructure (defined in driver - see chapter 2). If there is an error, go to error step #5.

9) (Optional) Call the driver defined command line parser. Save the card number received from the command line option "Card=xx" for later use (in *AddDiskDevice*). Parse all other custom options.

10) Test the adapter card for presence, correct option settings, and functionality. If there is an error, go to error step #5.

11) Call *RegisterHardwareOptions* to validate the options in the IOConfigurationStructure. If there is an error, go to error step #5.

12) Allocate ISR for Card by calling *SetHardwareInterrupt*, specifying the driver ISR address, the IRQ number allocated in step #8, and the previously allocated interrupt resource tag. Call *CAdjustRealModeInterruptMask* only if required (see chapter 7). If an error occurs, go to error step #4.

13) Start the driver AESNoSleep or AESSleep timer (which must provide driver timeout recovery) by calling *ScheduleNoSleepAESProcessEvent* or *ScheduleSleepAESProcessEvent*. If there is an error, go to error step #3.

14) (optional) Read Firmware or other custom data from the Custom Data File and load into the Co-processor. Start the Co-processor executing the loaded firmware. If there is an error, go to error step #2.

15) (optional) Allocate buffers below 16 megabytes (**only** for 16-bit adapters on systems supporting more than 16 megabytes - see Appendix G) by calling *AllocateBufferBelow16Meg*. If there is an error, go to error step # 2.

16) Register the adapter card with the OS by calling *AddDiskSystem*. If there is an error, go to error step # 1.

17) Return to the driver initialization caller with good (0) status.

Error Steps:

1) Deallocate buffers below 16 megabytes (if allocated in step 15) by calling *FreeBufferBelow16Meg*.

2) Delete AES timers by calling *CancelNoSleepAESProcessEvent* or *CancelSleepAESProcessEvent*.

3) Release interrupt by calling *ClearHardwareInterrupt* with the IRQ number and the ISRAddress (First call *CUnAdjustRealModeInterruptMask* only if *CAdjustRealModeInterruptMask* was called in step 12 above).

4) Return allocated hardware resources to the OS by calling *DeRegisterHardwareOptions*.

5) Return IOConfigurationStructure and any other dynamically allocated memory back to OS by calling *Free* (if allocated in step 5.)

6) Return to the driver initialization caller with bad (non-zero) status.

Checking the Driver

As a precautionary measure, NetWare requires mass storage drivers to provide a *CheckDriver* routine to determine if any of the driver's devices are locked. For example, the console command UNLOAD calls the driver's *CheckDriver* routine before unloading the driver.

When NetWare calls the driver's *CheckDriver* routine, the *CheckDriver* routine in turn calls the NetWare routine *CheckDiskCard* for each adapter card registered by the driver. The driver must pass a *CardStructure* pointer (indicating the associated adapter card) and the screen handle passed to the driver check routine by NetWare when calling the CheckDiskCard routine (must be called for each adapter card). The screen handle is used by NetWare to display any console warning messages.

CheckDiskCard will return one of four completion codes (a composite of the status of all devices attached to the card) to the *CheckDriver* routine for each registered card for which the driver makes a call:

0 No devices are locked. The driver can be unloaded safely.

1 At least one device is locked, but has a mirror. The driver can be unloaded without jeopardizing the data if the mirror is controlled from a separate driver.

2 At least one device is locked and does not have a mirror. Unloading the driver will probably jeopardize some data.

3 Identical with #2

If the device is locked (in use), CheckDiskCard also displays a warning message at the console.

The *CheckDiskCard* routine returns a status code that is the composite of the status of all devices registered for the card by the driver.

The *CheckDriver* routine should retain and combine the return codes from all calls to CheckDiskCard for all cards or devices registered by the driver. It should then return the combined status to the caller (the UNLOAD routine). If none of the devices are locked, the UNLOAD routine will proceed by calling the driver's *RemoveDriver* routine. If any device is locked, UNLOAD will prompt the console operator to determine whether to proceed. The console operator can abort the operation or allow it to continue. When the driver's *RemoveDriver* routine returns to the caller (UNLOAD), the driver is deleted from memory.

The following sequence illustrates the interaction between NetWare and the driver. Suppose that a driver supports one card which controls three hard disks. A console operator has just typed "unload sample.dsk <Enter>" at the file server console. The following steps would occur:

1) As a precaution, NetWare calls the driver's *CheckDriver* routine.

2) The *CheckDriver* routine calls NetWare's *CheckDiskCard* routine once for each adapter card registered by the driver.

3) *CheckDiskCard* returns a value of 0 if none of the disks are locked; it returns a value greater than 0 if any disk is locked.

4) The *CheckDriver* routine returns the composite (logical OR) status of all CheckDiskCard calls to the caller (UNLOAD).

5) If the *CheckDriver* routine returns a value of 0, UNLOAD calls the driver's *RemoveDriver* routine.

6) If the *CheckDriver* routine has returned a non-zero value, UNLOAD displays an interactive prompt on the console screen allowing the operator to abort the unload event.

7a) If the operator enters "No," UNLOAD terminates without making a call to the drivers *RemoveDriver* routine to unload the driver.

7b) If the operator entered "Yes," UNLOAD calls the driver's *RemoveDriver* routine.

CheckDriver

CheckDriver is called by NetWare to prompt the investigation of the driver's associated devices before removing the driver from memory.

Syntax
```
LONG CheckDriver(
        LONG ScreenHandle);
```

Return Values

0	Continue unloading the driver.
non-zero	Display the interactive console message.

Parameters

ScreenHandle	Receives a 4-byte handle to the driver's screen display. *CheckDriver* passes this handle when calling the NetWare routine *CheckDiskCard*, which uses this handle to display warning messages when disks are locked. This handle becomes invalid upon return from the check driver call, and must not be referenced by the driver except during the check driver call.

Remarks NetWare's UNLOAD command calls *CheckDriver*. The *CheckDriver* routine name is arbitrary. NetWare finds the name of a disk driver's *CheckDriver* routine in the .DSK file's header. Originally, the name of this routine appears in the driver's definition file following the CHECK keyword.

CheckDriver Routine

```
int             CardCount;                  /* total cards */
int             DeviceCount;            /* devices for current card */
CardStruct      *CardHandle;            /* pointer to current card's Card structure
                                        */
int             result;                     /* status of current card */
int             ccode;                      /* final status of driver */


CheckDriver(LONG ScreenHandle)
{
    Get CardHandle for first card;

    while (CardHandle != 0)
    {
         /* find out status of attached disks */
        result = CheckDiskCard(CardHandle, ScreenHandle);
        ccode = ccode | result;

        CardHandle = CardHandle->nextCard;
    }
    return (ccode);
}
```

Unloading the Driver

NetWare requires a *UnloadDriver* routine that removes all related structures and finally the driver's code image from file server memory when the routine is called by an UNLOAD console command. Please note that all devices and adapter cards are unloaded by a single call to the *UnloadDriver* routine.

The *UnloadDriver* routine must perform the following steps:

1) For each device registered with the OS, complete all pending I/O operations by calling *RemoveDiskDevice*. The OS will then flush all remaining dirty cache buffers; requests are ordered on the elevator queue and a call for each request is made to the driver's *IOPoll* routine. The OS will not return from *RemoveDiskDevice* (a blocking call) until a *GetRequest* and *PutRequest* has been performed for these and all other request remaining on the queue.

2) Deactivate any AES timers for devices, cards, or the driver by calling *CancelNoSleepAESProcessEvent* for each timer.

3) Deallocate any memory allocated for devices, cards, or drivers by calling *Free* for each block of memory obtained by calling *Alloc*. If special buffers have been allocated below 16 megabytes, they should be returned at this time by calling *FreeBufferBelow16Meg* (see Appendix G).

4) Return all *DiskStructures* to the Operating System by calling *DeleteDiskDevice* for each device registered by the driver.

5) Restore all interrupt(s) allocated by the driver for each card by calling *ClearHardwareInterrupt.* (First the driver must call *CUnAdjustRealModeInterruptMask* if *CAdjustRealModeInterruptMask* was called in step 12 of the driver initialization summary described in chapter 3.)

6) Return resources allocated by the driver for the card to the OS by calling *DeRegisterHardwareOptions.*

7) Return all *CardStructures* to the OS by calling *DeleteDiskSystem* for each card registered by the driver.

8) Return all dynamically allocated memory by calling *Free*, *FreeSemiPermMemory*, *FreeBufferBelow16Meg*, etc..

9) Return static driver memory to the OS by returning to the caller (UNLOAD).

Removing and Deleting Devices

The driver calls two NetWare routines to remove and delete a device: *RemoveDiskDevice* and *DeleteDiskDevice*. *RemoveDiskDevice* is passed a handle to the device to be removed. (The device handle was returned to the driver when the driver called *AddDiskDevice.*) *RemoveDiskDevice* removes the specified device from the NetWare list of active devices and issues a Deactive IOCTL. However, the device remains in memory until the driver calls the OS support routine *DeleteDiskDevice. DeleteDiskDevice* is also passed the device handle and returns to NetWare the memory allocated for the device's *DiskStructure*.

Preparing a Card for Deletion

After all the devices for a card have been deleted, the card itself can be removed. Before removing the card, the driver should clear the card's interrupt vector, cancel its timeout routine, and remove its hardware options.

The driver calls the NetWare routine *ClearHardwareInterrupt*. The IRQ level of the hardware interrupt must be passed to the routine.

The card's timeout routine is canceled (if one exists) by calling the NetWare routine *CancelNoSleepAESProcessEvent.* This routine must be passed a pointer to the same AESEvent structure that was used to initiate the timeout event.

To remove the card's registered hardware options, the driver calls the NetWare routine *DeRegisterHardwareOptions*. This routine must be passed a pointer to the card's *IOConfigurationStructure.*

Deleting a Card

Once the card is prepared for deletion, *UnloadDriver* must call the NetWare routine *DeleteDiskSystem*. This routine requires a handle for the *CardStructure* associated with this card. The handle for the structure was returned to the driver when the driver called *AddDiskSystem* during its *InitializeDriver* routine.

UnloadDriver

UnloadDriver is called by NetWare to initiate the removal of a driver's devices and cards.

Syntax void UnloadDriver();

Return Values None

Remarks NetWare's UNLOAD command calls the *UnloadDriver* routine after calling the driver's *CheckDriver* routine.

The name of the *UnloadDriver* routine is arbitrary. NetWare finds the name of a disk driver's remove routine in the .DSK file's header. Originally, the disk driver writer includes this name in the driver's .DEF file after the EXIT keyword. Using the .DEF file, the NetWare 386 loadable module linker puts this routine name in the .DSK file's header.

Unload Driver Outline

```
int CardCount;                      /* the number of cards the driver supports */
int DeviceCount;                    /* the number of devices supported by a card */
IORequestStruct *IORequestPtr;      /* pointer to an I/O Request structure */
AESEventStruct *EventPtr;           /* pointer to a card's timeout event */
IOConfigStruct *IOConfigPtr;           /* pointer to a card's I/O Config struct */
int NextRequest = 0;             /* Get the next request */
DiskStruct *DiskHandle;             /* pointer to current device's Disk structure */
CardStruct *CardHandle;             /* pointer to current card's Card structure */
int Status = 2;                     /* non-zero value for dummy RemoveDiskDevice status */


UnloadDriver()
{
    while (CardCount >= 1)
    {
        Clear interrupts;
        Get a CardHandle for this card;

        while (DeviceCount for this card >=1)
        {
            Get DiskHandle for this device;

            RemoveDiskDevice(DiskHandle,Status);

            DeleteDiskDevice(DiskHandle);
        }

        /* Clear the interrupt vector for this card */
        ClearHardwareInterrupt(HardwareLevel,ISRAddress);

        /* Cancel the timeout event for this card */
        CancelNoSleepAESProcessEvent(EventPtr);

        /* Release I/O ports, etc. for this card */
        DeRegisterHardwareOptions(IOConfigPtr);

        /* Delete this card */
        DeleteDiskSystem(CardHandle,Status);
    }
    Free(MemPtr);
    return();
}
```