Chapter 1: The NetWare Driver Environment

Drivers as NLMs

NetWare file server device drivers are NetWare Loadable Modules (NLMs), allowing a network supervisor to add or delete drivers while the system is running. This chapter gives a brief overview of the NetWare Operating System driver environment with emphasis on the concept of loadable modules. The routines a device driver NLM must provide to interface with the NetWare Operating System are discussed later in Chapter 3.

Device drivers must be converted from object form to a special Loadable Module form. The driver must provide basic NLM interface routines which adhere to NLM interface specifications and facilitate dynamic loading and unloading. The driver must also follow prescribed procedures and must utilize the defined standard routines outlined in Chapter 7 to interface to the Operating System. The driver must make public its required NLM interface routines, Mass Storage Control interface routines, and import any required NetWare Operating System interface routines. The details of this NLM driver creation procedure and definition file keywords are discussed in Appendix A.

Loading and Unloading Drivers

To load a driver, it must be placed 1) on a floppy diskette, 2) in a directory on the DOS partition of the file server's hard disk, or 3) in the SYS:SYSTEM directory of the file server.

After the file server is loaded and running, the console command "LOAD" is used to load the desired driver into file server memory. The following examples show how to load a sample driver from all three sources.

Example load from a floppy disk drive:

load a:sample slot=3 port=200 int=5 card=5 <Enter>

Example load from a local hard disk:

load c:sample card=4 <Enter>

Example load from the file server (only if SYS: volume is mounted):

load sample card=7 <Enter>

The NetWare Loader resolves the drivers import list and dynamically links the driver to the NetWare Operating System.

To unload a driver, use the file server console command "UNLOAD" as follows:

unload sample <Enter>

Unloading a driver causes all outstanding requests for all devices serviced by the driver to be flushed, after which any resources allocated by the driver must be returned to the OS. The driver is removed from memory after it returns to the routine that requested the unload. A reentrant driver (discussed below) must flush all requests and return all resources for all cards for which it has been specified in previously successful "LOAD" commands. A single call to the driver's unload entry point indicates an unload for all associated adapter cards.

Reentrance

A driver may be declared reentrant. Declaring a driver reentrant in the definitions file will cause subsequent "LOAD" commands (multiple adapter cards supported by the driver) for the same driver to use the driver image first loaded into memory. For each instance of the driver initialization routine, the OS passes to the driver a pointer to the command line so that the command line parameters may be process.

The OS support function *ParseDriverParameters* is provided to process driver parameters, check for possible conflicts, and prompt the console operator for required parameters that are missing. Reentrant drivers preserve the integrity of the information specific to each instance loaded. This requires proper use of control blocks, tables, structure pointers, and interrupt disabling in particular areas. Calls to and from the OS (*IOPoll*, *PutRequest*, *GetRequest*, etc) facilitate reentrancy by passing as a parameter the structure pointer that that identify the instance being call.

OS Environment

All drivers are required to run in 32-bit mode regardless of the language used to write the driver. Drivers may always assume SS=ES=DS, but should not assume that the Code Segment Register is identical with DS. Drivers must comply with several execution levels provided by the OS. Drivers must not violate the defined environment of the current execution level. The execution levels and their associated environments are as follows:

A) Blocking Process Level

The Blocking Process Level is an execution level where a process can make calls to system routines that temporarily suspend its (the process') execution while waiting for the completion of another task. At this level the code executes as the operating system's currently scheduled process. Routines called from this level may make calls to blocking routines that can put the current process and the associated thread of execution to sleep until completion of some task. Driver routines called at this level execute as an extension of the current executing process. Any routines described in Chapter 7 may be called at this level, whether indicated as blocking or non-blocking. However, routines indicated in Chapter 7 as requiring the processor interrupts to be disabled must be called with interrupts disabled.

Driver routines called at this level are responsible to do necessary housekeeping required for the "C" compatible interface defined for drivers (see Appendix C for details).

Interrupts are normally enabled upon entry to routines at this level. It is often necessary for a driver to disable interrupts for a period of time to accomplish reentrance, to call system routines, or to maintain driver integrity. Care should be taken to disable interrupts for the absolute minimum period required to accomplish necessary functions. Disabling interrupts for any significant period will cause server performance degradation and poor response time.

Routines at this level may not execute with interrupts disabled for more than 25 milliseconds. They also must return to the Operating System or cause a task switch within 250 milliseconds. If the function to be accomplished by the called routine requires more than the above period, the driver must call *DelayMyself*, *CYieldIfNeeded* (v4.xx), *CRescheduleLast* (v3.1x), or another system routine which causes a task switch so that other NetWare processes may be serviced in a timely fashion. Failure to do so may cause the Operating System to indicate the drivers violation on the server console.

Driver-defined entry points called at blocking process level are:

Driver Initialization
Driver Check
Driver Unload
Driver Scan for Devices
Driver Delete Device
Driver TimeOut - AESProcess Event Entry (Sleep option only)

B) Non-Blocking Process Level

Non-Blocking Process Level is defined as an execution level where a process is not permitted to temporarily block or suspend its thread of execution (by making calls to system routines which suspend the process execution until the specified function is completed). At this level the code executes as the operating system's currently scheduled process. Routines called from this level may not make calls to blocking routines that may put the process and the associated thread of execution to sleep until completion. Driver routines called at this level also execute as an extension of the current executing process. Only system routines indicated in Chapter 7 as non-blocking may be called at this level.

Routines at this level may not execute with interrupts disabled for more than 25 milliseconds. They also must return to the Operating System within 250 milliseconds.

Driver routines called at this level are responsible to do necessary housekeeping required for the "C" compatible interface defined for drivers (see Appendix C for details).

Interrupts are inhibited upon entry to routines at this level, with the exception of the No Sleep AESProcess event entry, which is enabled.

Driver-defined entry points called at non-blocking process level are:

Driver IOCTLPoll
Driver IOPoll
Driver TimeOut - AESProcess Event Entry (No Sleep option)

C) Interrupt Level

The current process is unknown upon entry at this level. Blocking routines that might put the process and the associated thread of execution to sleep until completion may not be called under any circumstance from this level. Only system routines indicated in Chapter 7 as non-blocking may be called at this level.

Interrupts are always inhibited upon entry to routines at this level.

The only entry point at this level is the Driver ISR entry.

The driver's ISR is not required to save or restore registers, because all registers have been saved and segment registers initialized prior to the ISR routine being called by the system ISR entry point. The driver must execute a RET to return from the call, specifically **must not** execute an IRETD before returning, because an IRETD will be issued by the system ISR after the driver returns. The driver must issue required EOI(s) (End Of Interrupt) by calling the *CDoEndOfInterrupt* support routine.

Drivers must be coded with care if interrupts are enabled after issuing necessary EOI(s). Drivers must also protect themselves if they enable interrupts during the ISR routine and the interrupt is shared. (See Chapter 6 for shared interrupt details.) Drivers must clear the interrupt request in the associated adapter card, and must also issue the EOI(s) by calling the NetWare drive support routine.

Driver ISR routines may not execute with interrupts disabled for more than 25 milliseconds. They also must return to the Operating System within 250 milliseconds.