Chapter 2 NetWare Peripheral Architecture Overview

The NetWare Peripheral Architecture (NWPA) is an extension of the NetWare Media Manager, and its goal is to provide NetWare v4.x customers with broader and more reliable driver support for third-party host adapters and storage devices. Eventually, the NWPA will replace the existing NetWare driver specification.

The purpose of this chapter is to provide developers with a conceptual understanding of the NWPA as a complete system. This overview provides a good foundation that information presented in subsequent chapters will build upon. Also incorporated in this overview, is a discussion on the advantages associated with the NWPA and some comments on general NetWare OS issues that affect its operational environment.

2.1 Advantages

The NWPA provides a modular design that separates NetWare driver support into two components, a Host Adapter Module (HAM) and a Custom Device Module (CDM). The HAM is the component associated with the host adapter hardware, and the CDM is the component associated with storage devices or autochangers attached to a host adapter bus. Because they are separated, each component can be developed independently, allowing developers to concentrate their development efforts on their respective components. Allowing developers to specialize in this way, coupled with the fact that the NWPA has a simpler API set, will make NetWare driver development easier. Thus, NetWare drivers built on the NWPA should be more reliable and easier to maintain. Adding to the idea of making driver development easier, the NWPA provides internal support filters that eliminate the need for special driver handling of read-after-write verify, 16 MB addressing limitations on 16-bit host adapters, sector conversion, scatter/gather, and other similar considerations required in the previous NetWare driver specification.

A completely new feature provided by the NWPA is hot replacement. Hot replacement is a term describing the ability to dynamically swap a driver already loaded in file server memory with a newer version without needing to down the server. This ability can save a system operator many hours because disk volumes do not need to be remounted following a version upgrade of a driver.

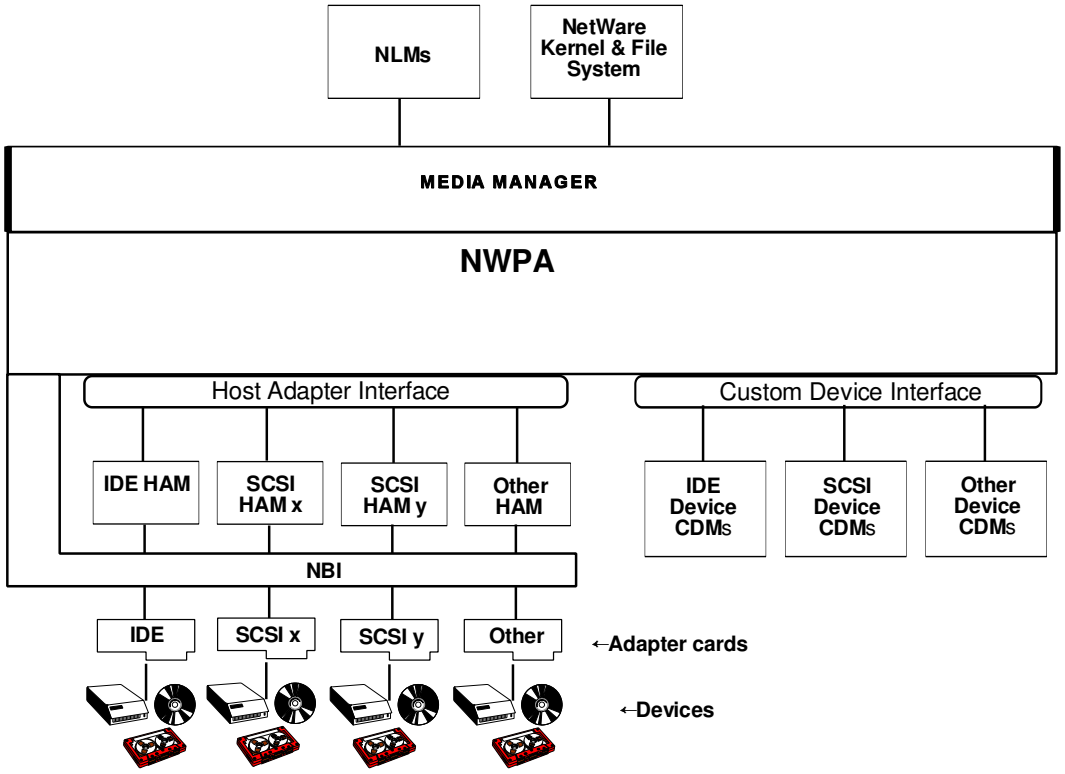
2.2 Conceptual Overview

This section explains the different NWPA components and outlines the general I/O path.

2.2.1 NWPA Components

Figure 2-1 shows the components that comprise the NWPA, and a brief description of each component follows:

Figure 2-1 NWPA Components



Media Manager The Media Manager is the storage management layer of the NetWare v3.12 and 4.x Operating Systems (OS), and is the "brain" that runs the NWPA . The Media Manager provides a robust storage management interface between applications and storage device drivers. The

Media Manger fields application I/O requests and converts them to messages compatible with the NWPA architecture. The Media Manager is described in detail in the **Media Manager Functional Specification and Developer's Guide**, which may be obtained from Novell Labs.

Host Adapter Module (HAM) As previously mentioned, HAMs are the driver components associated with host adapter hardware. These program modules are supplied by third-party developers, and they provide the host adapter interface. HAMs are loaded as NetWare Loadable Modules (NLMs), and they must provide the functionality to route requests to the bus where a specified device is attached.

Host Adapter Interface (HAI) The HAI is a set of APIs within the NWPA that provides an interface for HAMs to communicate with the Media Manager.

Custom Device Module (CDM) As previously mentioned, CDMs are the driver components associated with storage devices. These program modules are supplied by third-party developers, and they implement the functionality to build device-specific commands from I/O messages received from the Media Manager (CDM Messages). CDMs are loaded as NLMs.

Custom Device Interface (CDI) The CDI is a set of APIs within the NWPA that provides an interface for CDMs to communicate with the Media Manager.

CDM Message The CDM message is a data structure paralleling the content and structure of internal Media Manager messages. The Media Manager receives an I/O request form an application, converts it to a CDM message, and passes the message to a CDM. It is from the contents of this structure that a CDM builds a request structure (SuperHACB) specific to a particular hardware-bus protocol. The SuperHACB contains device specific commands.

Super Host Adapter Control Block (SuperHACB) The SuperHACB is a data structure built by a CDM. Each SuperHACB contains a HACB as one of its data members along with some additional data space. CDMs can use the additional data space for whatever purpose they need. The HACB is the protocol-specific request structure containing the data essential to a HAM.

Host Adapter Control Block (HACB) The HACB is an I/O data structure packaged into a protocol-specific command block (such as SCSI or IDE\ATA) . All I/O requests to the HAM are in the form of HACBs, and

the HAM passes these requests on to the devices attached to the hardware bus.

NetWare Bus Interface (NBI) This interface is a hardware abstraction layer that allows modules to be written that are platform independent. Some platforms, for example, may support more than one bus at a time, and each bus will be quite different from each other. The NBI makes platform related issues (such as addressing the Programmable Interface Controller, etc.) transparent to the software modules. See Paragraph 4.2.1 for more details. Chapter 7 contains the NBI related APIs. These APIs are identified by **NPAB_** prefixes.

2.2.2 General Flow of Events

CDMs and HAMs are loaded into file server memory as NetWare Loadable Modules (NLMs). Therefore, they need to provide the OS with standard NLM entry points for load-time module initialization and unload-time returning of system resources. Once initialized, CDMs and HAMs register themselves with the Media Manager specifying additional entry points for receiving I/O requests, and they must specify the type of devices (CDM) or host adapter interfaces (HAM) they will support. In the case of a CDM, one of these entry points is an inquiry routine.The Media Manager calls this routine following CDM registration passing it two arguments. One is a Media Manager generated ID to a registered device matching the type for which the CDM registered, and the other is a handle to the HAM receiving requests for that device. Within the context of the inquiry routine, the CDM determines if it will support the device. If the CDM decides to support the device, it makes a CDI call "binding" itself to the device. Binding means that the CDM tells the Media Manager that it will be the module accepting I/O requests for the specified device, and what functions it will support for that device. In order to bind to a device, the CDM must generate a local bindhandle for the device and pass it as an argument in the CDI call. The Media Manager associates the bindhandle to the device in its object database and, from that time forward, uses the bindhandle to identify that device when talking with that particular CDM. The Media Manager will make subsequent calls to a CDM's inquiry routine whenever a new device object matching the CDM's device type is created.

In the case of a HAM, during its initialization routine its scans hardware slots for adapters matching the bus-protocol it is designed to support. When a matched adapter is found, the HAM generates a unique HAM handle for the adapter. For adapters that provide more than one bus, the HAM must generate a HAM handle for each bus instance. A HAM can be designed to support either a single adapter made by a single manufacturer or multiple adapters made by different manufacturers as long as each adapter matches the HAM's bus protocol type.

To conclude its initialization, a HAM registers a HAM handle and a pair of I/O entry points with the Media Manager via a call to a HAI routine for each adapter/bus instance it will support. One of the entry points is for receiving and routing I/O requests to the appropriate adapter/bus and its attached devices. The other is for handling aborts on pending requests in a specified device's queue. As an example, if a HAM will support an adapter having two buses, the HAM must register a HAM handle and entry point pair twice, once for the first bus and again for the second bus. The HAM handle in each registration must be unique for each bus; however, the HAM has the option of specifying either the same, or a different, entry point pair in each registration. Typically, a HAM will have only one pair of I/O entry points to handle I/O routing to adapters and their devices. After module initialization and registration of its entry points, the HAM is ready to accept HACB I/O requests. At this point, the HAM waits until it receives its first I/O request from the Media Manager. This first request is initiated by the system operator at the command line, and it is the "Scan For New Devices" command. During the context of this request, the HAM scans each adapter/bus it will support and builds its device lists. The HAM must build a device list for each bus it will support and assign a handle to each device attached to a bus. Each device list should then be associated with its corresponding bus through the HAM handle. The HAM determines proper routing of a request by using the HAM handle to identify the proper adapter and bus. It then uses the device handle to index into the bus's device list to identify the proper device. The following is an outline of the general flow of events in the architecture:

1. An application or the OS issues an I/O request to the Media Manager, which then converts the raw request into a CDM Message. The CDM Message supplies all the information necessary to complete the request, such as the operation to perform and data buffers where data is to be moved to or from.

2. The Media Manager then passes a pointer to the CDM Message to the CDM's I/O entry point specified during CDM registration.

3. The CDM builds a SuperHACB from the data in the CDM Message. The CDM then passes a pointer to this SuperHACB to the Media Manager through the CDI interface.

4. The Media Manager then routes the HACB portion of the SuperHACB to the HAM supporting the target device associated with the I/O request.

5. The HAM then ports the device command in the HACB to the appropriate registers of the adapter to which the device is attached.

6. After the device finishes processing the command, the HAM is notified (usually by an interrupt).

7. The HAM does whatever is necessary to complete the HACB I/O request (such as moving data to the buffer(s) specified in the HACB during a READ), places completion information in the HACB, and then passes a pointer to the HACB to the Media Manager through the HAI interface.

8. The Media Manager then does a callback to the CDM passing it a pointer to the original SuperHACB it built. At this point, the CDM checks the completion information in the HACB portion of the SuperHACB to determine the device's error status. The CDM then returns to the Media Manager the completion status of the original CDM message that initiated the CDM-HAM I/O sequence just processed.

2.3 NetWare OS Environment

This section discusses NetWare Operating System (OS) issues that affect the NWPA. The NetWare OS is quite sophisticated and complex; therefore, the information presented in this section deals only with the NWPA.

2.3.1 Operating Mode

All NetWare device 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 is identical with DS.

2.3.2 Multitasking and Process Levels

The NetWare OS uses non-preemptive thread scheduling for its multitasking environment. Non-preemptive thread scheduling provides greater system performance because it minimizes context switches, and it eliminates the need for semaphore-based locking and unlocking code. Ideally, to maintain system performance, all application threads should be designed to quickly perform their respective operations and then return the thread of execution back to the ring process. However, some applications may have code sections where a process has to wait a significant number of CPU cycles to complete. An example of this kind of situation is when a device driver communicates with host adapter hardware to determine what devices are attached to its bus. This task could take several hundred milliseconds. Situations like this impact server performance because the current thread hordes CPU time, and other scheduled threads and critical background OS processes do not get their turn to run. NetWare provides a mechanism to minimize such impacts on server performance by allowing

applications to designate a process level for each thread it wants scheduled. Likewise, process levels are assigned to OS routines and other routines in the NWPA architecture. Drivers must comply with the execution levels provided by the OS and not violate their defined environments. These process levels are defined in the subsections that follow.

2.3.2.1 Blocking Process Level

The blocking process level is defined as an execution level that is permitted to temporarily block or suspend its thread of execution by calling an NWPA routine that suspends 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 make calls to other blocking routines that may put the process and the associated thread of execution to sleep until completion. This level represents the currently schedule and executing task or process. Driver routines called at this level execute as an extension of the current executing process. 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 during these process-level routine to accomplish reentrance, 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. Routines at this level may execute for up to 250 milliseconds before returning to the OS or causing a task switch. If the function to be accomplished by the called routine requires more than the above period, the driver should initiate a task switch by calling the appropriate NWPA routine so that other NetWare processes may be serviced in a timely fashion. Failure to do so may cause the OS to indicate the driver’s violation on the server console.

2.3.2.2 Non-Blocking Process Level

The non-blocking process level is defined as an execution level that is not permitted to temporarily block or suspend its thread of execution. At this level the code executes as the OS's currently scheduled process, and it is guaranteed to run to completion. Routines called from this level may not make calls to routines at the blocking process level that may put the process and the associated thread of execution to sleep. This level represent the currently scheduled and executing task or process. Driver routines called at this level also execute as an extension of the current executing process. Therefore, routines at this level can only call other routines that are at the same level.

2.3.2.3 Interrupt Process 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 non-blocking routines may be called from routines executing at this level. Interrupts are always inhibited upon entry to routines at this level. The only entry point at this level is the *HAM_ISR()* routine.

HAM_ISR() is not required to save or restore registers, as all register have been saved and segment registers initialized prior to *HAM_ISR()* being called by the system. The driver must execute a RET to return from the call, and specifically must not execute an IRETD before return, as IRETD will be issued by the system ISR after *HAM_ISR()* returns. Drivers must protect themselves if they enable interrupts during the ISR routine and the ISR is shared. Driver ISR routines must not be lengthy or cause the driver to run with interrupts disabled for any significant period.