Chapter 4 Host Adapter Module (HAM)

The Host Adapter Module (HAM) is the driver component that provides the software interface to the host adapter hardware, and it is implemented as a NetWare Loadable Module (NLM). Additionally, the HAM's access to the adapter is exclusive. This chapter describes a HAM's function, and it is organized into the following sections:

- **Architecture**
  This section prototypes and describes the entry points, functions, and routines that make up the HAM's architecture and its interface with the NWPA.

- **Operational Overview**
  This section overviews the HAM's functionality by outlining the main flow of events of its procedures.

- **Special Topics**
  This section discusses special topics relevant to a HAM.

4.1 HAM Architecture: Entry Points, Functions, and Routines

This section provides prototypes for the entry points, functions, and routines required in a HAM by the NWPA. A developer may use these prototypes to plumb the shell of a HAM. Detailed descriptions of the data structures and entry points can be found in the technical reference chapters of this developer's guide. To fit properly in the architecture, a HAM is required to provide the following:

- NLM Load/Unload-time Entry Points
- NWPA I/O Entry-Points
- Timeout Routine
- HACB Type Zero Functions (also referred to as HAM Functions)
- Host Adapter Interface Routines

Host adapter interface routines are mentioned here because they are crucial to the HAM architecture. However, this developer's guide does not attempt any specifications on these routines, since they are manufacturer specific. Prototypes and definitions of these routines are the responsibility of HAM developers. Complete functional specifications of the entry points can be found in Chapter 7, and descriptions of the HAM Type Zero functions can be found in Chapter 8. The main flow of each entry point is discussed in the operational overview of this chapter. The names of these entry points are left to the discretion of the HAM developer; however, each entry point must provide the respective functionality described in this guide. For consistency in referring to these entry points and HAM functions within the text and in

code examples, this guide gives each a generic name having a *HAM_* prefix. Whenever an entry point or function with this prefix is encountered, it indicates that the routine is HAM specific. The *italic* typeface indicates that the name is arbitrary.

4.1.1 NLM Load/Unload-Time Entry Points

A HAM must provide three standard NLM entry points for the OS. These entry points are made visible to the OS through a definition (.DEF) file that is processed by the NLMLINK utility. A sample definition file is provided in Appendix A. The prototypes of these entry points, along with their generic names, are as follows:

```
LONG HAM_Load (
    LONG loadHandle,
    LONG screenID,
    BYTE *commandLine
    );
```

HAM_Load() is the HAM's load-time entry point called when the systems operator issues a LOAD command on the HAM from the console. *HAM_Load()* is called on a blocking thread. Through this entry point, a HAM receives its OS-generated resource handle (*LoadHandle*), an ID to the LOAD console screen, and a pointer to the LOAD command line string which contains the hardware resource options specified by the systems operator. These hardware options include resources such as interrupts, DMA channels, memory decoding for memory-mapped I/O, ports, and even custom command-line options. *HAM_Load()* is responsible for allocating any resources needed to make the HAM operational, for configuring the HAM based on the hardware options specified on the LOAD command line, and for registering the HAM and its I/O entry points with the NWPA.

```
LONG HAM_Unload_Check (LONG screenID);
```

HAM_Unload_Check() is the HAM's initial unload-time entry point. The entry point gets called when the systems operator issues an UNLOAD command on the HAM from the console. *HAM_Unload_Check()* is called on a blocking thread. *HAM_Unload_Check()* is responsible for checking to see if any of the HAM's devices are currently being used by an application and return use-status. To do this, *HAM_Unload_Check()* returns the use-status returned by **NPA_Unload_Module_Check()**. For example:

```
LONG HAM_Unload_Check (LONG screenID)
    {
       return (NPA_Unload_Module_Check(...));
    }
```

From this return value, the OS can determine if any of the devices managed by the HAM are locked. If any devices are locked, the OS displays a message at the console listing the devices that will be deactivated and the corresponding NetWare volumes that will be dismounted if the action is

continued. The user then has the option to either continue or abort the unload.

```
void HAM_Unload (void);
```

HAM_Unload() is the HAM's final unload-time entry point, meaning that the unload thread already called *HAM_Unload_Check()* and the systems operator chose to continue. Thus, the unload thread was allowed to continue and make a call to *HAM_Unload()*. *HAM_Unload()* unregisters the HAM from the NWPA and returns allocated resources back to the system. Once the HAM is unloaded, all devices attached to the bus(es) it was managing are inaccessible.

4.1.2 NWPA I/O Entry Points

A HAM must provide additional entry points that allow the NWPA to route I/O requests to the module and to retrieve the resultant data. These entry points are made visible to the system when the HAM registers itself with the NWPA using **NPA_Register_HAM_Module()** . Additionally, all of these entry points have non-blocking context, meaning that they must perform their respective functions quickly and return control back to their respective calling processes. The prototypes of these entry points, along with their generic names, are as follows:

```
LONG HAM_Execute_HACB(
    LONG hamBusHandle,
    struct HACBStruct *HACB);
```

HAM_Execute_HACB() is the HAM's entry point for receiving HACB I/O requests and routing them to their respective devices through the host adapter. As long as the host can accept a request, *HAM_Execute_HACB()* should issue it to the adapter and then return to the calling process. If the host is temporarily unable to accept a request, *HAM_Execute_HACB()* should place the request on an internal queue for the target device and return to the calling process. The fundamental rule for this entry point regarding a HACB I/O request is to "do it or queue it."

```
LONG HAM_Abort_HACB(
    LONG hamBusHandle,
    struct HACBStruct *HACB,
    LONG flag);
```

HAM_Abort_HACB() is the HAM's entry point for receiving aborts on HACB requests. *HAM_Abort_HACB()* locates the target HACB, posts an abort code, and returns the HACB to the CDM layer by calling **HAI_Complete_HACB()**.

```
LONG HAM_Check_Option
```

```
    (
    struct NPAOptionStruct *Option,
     LONG instance
     LONG flag
    );
```

HAM_Check_Option() is the HAM's entry point for receiving and verifying command line options. The entry point is called during two separate NWPA processes: once during the command line parsing phase of HAM initialization and again during the actual registration of hardware options. The HAM invokes these two NWPA processes at different points in its load-time entry point, *HAM_Load()*.

```
LONG HAM_Software_Hot_Replace(
    LONG messageLength,
     void *message
    );
```

> **Note:** **This entry point is optional.** The HAM only needs to implement *HAM_Software_Hot_Replace()* if it plans to support hot replacement.

HAM_Software_Hot_Replace() is the HAM's entry point for dynamically performing version updates. This entry point allows a later-version HAM driver to exchange configuration information with an earlier-version that is currently loaded and operating on the server. The information exchange is in preparation for a dynamic swap of the modules without dismounting any volumes or disrupting the I/O channel for a lengthy period of time. At Novell, this process is called software hot replacement. For more details on how to implement this feature, see section 4.3.4.

```
LONG HAM_ISR (LONG irqLevel);
```

HAM_ISR() is the HAM's interrupt-time entry point, or interrupt service routine (ISR). The NetWare OS fields the actual hardware interrupt generated by the adapter board and routes its handling to the routine that registered for the interrupt. The HAM registers its ISR during its initialization entry point, *HAM_Load()*, using **NPA_Register_HAM_Module()**. The HAM registers the IRQ level it will service using **NPA_Register_Options()** during the hardware options registration phase of *HAM_Load()*. *HAM_ISR()* must provide logic to service completion of all I/O requests, provide the strategy for determining what device completed the request, post appropriate HACB completion codes, and initiate the next request on the device's process queue.

> **Note:** If the HAM intends to support software-hot-replacement, it may only have a single ISR.

4.1.3 Timeout Routine

The HAM must provide a routine that times out HACB requests grossly exceeding expected device-process time. The purpose is to provide a mechanism to return process control back to the OS from a hung-device condition. The HAM's timeout routine runs as a periodic, background process, and it gets initially scheduled for triggering at load-time during *HAM_Load()*. The prototype of this routine, along with its generic name, is as follows:

```
void HAM_Timeout (LONG parameter);
```

HAM_Timeout() is an asynchronous countdown-timer routine set up by calling **NPA_Spawn_Thread()**. *HAM_Timeout()* is triggered after the time interval specified as an input parameter to **NPA_Spawn_Thread()** elapses.

NPA_Spawn_Thread() is a one-shot API, meaning that it will only schedule the triggering of *HAM_Timeout()* once per call made to it. Therefore, after *HAM_Timeout()* triggers and performs its task, it should reschedule itself by calling **NPA_Spawn_Thread()** again in order to continue its periodic triggering.

HAM_Timeout() allows the HAM to time out an I/O request when the time interval specified in the **timeoutAmount** field of the HACB has expired. The timeout countdown begins when the HAM issues the request to the host. For more details about the countdown, refer to the description of the HACB's **timeoutAmount** field in Chapter 3.

4.1.4 HACB Type Zero Functions

HAMs must allow different types of HACB requests to be processed. The HACB's type is the value in its **HACBType** field which is set either by the CDM I/O routine building the HACB or by the NWPA. The CDM, or NWPA, then fills the HACB's command area overlay with a command structure appropriate to the HACB's type. **HACBType=0** requests contain adapter-specific Host command structures. The NWPA requires a HAM to implement functions that handle as many **HACBType=0** requests as are applicable to the adapter the HAM will manage. Some of the **HACBType=0** requests ask for information about the HAM, the host adapter, or attached devices. The HAM receives requests of this type through the union to Host

command block of the HACB. The following is a list of the HACB type zero functions and their generic names. Prototypes are not given because requests to perform these functions are received by the HAM in the form of HACB messages received through the HAM's *HAM_Execute_HACB()* entry point. The HAM determines which function to perform based on the parameters contained in the union to Host command block of the HACB. Refer to Chapter 8, HACB Type Zero Functions for more detailed descriptions.

> **Note:** HACB Type Zero functions are also known as HAM functions, and they are static functions that can generally be completed immediately within the context of *HAM_Execute_HACB()*.

HAM_Return_HAM_Info (mandatory)
HAM_Return_HAM_Info is responsible for supplying the NWPA with information about the HAM. The NWPA initiates this request soon after the HAM is loaded, and the information is in a form defined by the **HAMInfoStruct**.

HAM_Scan_For_Devices (mandatory)
HAM_Scan_For_Devices is responsible for scheduling a blocking process to scan for all devices attached to the selected adapter. This function may schedule the scan and return to the calling process since the calling process has non-blocking context. The HAM schedules the scan process using **NPA_Spawn_Thread()**. The HAM should schedule the process to trigger immediately. It is during the context of the scan process that the HAM builds a device list and generates a unique handle for each device. The NWPA makes these HAM-generated device handles available to the CDM layer. CDMs will use these handles to route HACBs to a particular device by placing the handle value in the HACB's **DeviceHandle** field. A request for this function is initiated by the systems operator at the console and can be called at any time. Additionally, for each time the HAM receives this request, the HAM must respond with a physical scan of the host bus, not just with a scan of an existing device list. If during the physical scan, the HAM discovers new devices or it discovers that some devices have gone away, it should refresh its device list.

HAM_Return_Device_Info (mandatory)
HAM_Return_Device_Info is responsible for supplying the NWPA with information about a device attached to the HAM's adapter. Following the completion of a *HAM_Scan_For_Devices* request, the NWPA initiates a find-first-find-next sequence of these requests until information about each device is returned. The return information for each request is in a form defined by the **DeviceInfoStruct**.

HAM_Unfreeze_Queue (mandatory)
HAM_Unfreeze_Queue is responsible for unfreezing the HAM's HACB request queue for the selected device. The HAM needs to maintain a process queue for each device it services.

HAM_Queue_AEN_HACB (mandatory)
HAM_Queue_AEN_HACB is issued by a CDM, and it directs the HAM to monitor asynchronous hardware events such as a bus reset, a device reset, or a device attention. If this event occurs, the HAM sets a bit mask in the HACB indicating the event and completes the HACB with the AEN status code. Completing the HACB informs the CDM of the event.

HAM_Set_IDE_Device_Config (implement if applicable)
HAM_Set_IDE_Drive_Config is only applicable to IDE\ATA drives, and even then, implementing it is optional. This routine is responsible for changing the transfer block size per IDE\ATA interrupt, thus allowing a CDM to use special commands as they appear in the IDE\ATA specification.

HAM_Tag_Queue_Synch/Asynch (implement if applicable)
HAM_Tag_Queue_Synch/Asynch is issued by a CDM to tell the HAM if a device supports either tag queuing or SCSI synchronous/asynchronous device negotiation.

HAM_Recovery_Reset (mandatory)
HAM_Recovery_Reset is issued by a CDM to tell the HAM to perform a reset of the adapter, bus, or device. The CDM will issue this HAM function when trying to recover from a dead device.

HAM_Deactivation_Notification (optional)
HAM_Deactivation_Notification is issued by a CDM to notify the HAM that it has deactivated a device.

4.1.5 Host Adapter Interface Routines

The HAM is expected to implement routines that use HACB information to construct appropriate command blocks in the adapter-specific protocol and issue them to the host. A HAM is only required to support HACB requests with a type suitable to the adapter it supports. The following is the current NWPA definitions for HACB request types: **HACBType=1** requests have an I/O command structure conforming to SCSI protocol. The HAM receives requests of this type through the union to SCSI command block of the HACB. **HACBType=2** requests have an I/O command structure conforming to IDE/ATA protocol. The HAM receives requests of this type through the union to IDE\ATA command block of the HACB. **HACBType=3** requests have an I/O command structure that conforms to raw Media Manager

messages. The HAM receives requests of this type through the union to CDMPassThrough command block of the HACB. For whatever adapter type the HAM supports, it must provide the adapter-interface-specific routines that implement the respective commands.

4.2 Operational Overview

The information in this section builds on the declarations and prototypes given in the previous section by describing a HAM's major functional procedures and their main flow of events. The information provided here should help to add functionality to a HAM program shell. Detailed definitions of data structures and functional descriptions mentioned in this overview are not included to avoid frequent detours that may detract from main-flow concepts. However, these details are provided in the technical reference chapters of this developer's guide. The following list gives a breakdown of the information in these chapters: Definitions of data structures can be found in Chapter 6, "Technical Reference for NWPA Data Structures." Functional descriptions of HAM entry points and HAI / NWPA support routines can be found in Chapter 7, "Technical Reference for NWPA Routines." Functional descriptions of HACB type zero functions can be found in Chapter 8, "HACB Type Zero Functions." Functional descriptions of NetWare OS support routines can be found in Chapter 10, "OS Support Routines."

Figure 4-1: HAM Initialization

Figure Is Being Updated

4.2.1 Load-time Initialization and Registration

Loading of the HAM can be initiated in multiple ways: by the systems operator at the server console, by a startup file, or by INSTALL. Figure 4-1 and the following steps show the sequence of events for initializing and registering a HAM at load-time. Note: Figure 4-1 is being updated to describe the NBI initialization process and to correctly describe HAM initialization. The next release of this specification will contain this updated flowchar. The following paragraphs do include the NBI paradigm, however.

1. When a HAM is loaded, the OS calls the HAM's *HAM_Load()* entry point passing it *loadHandle*, *screenID*, and *commandLine* as input parameters. *HAM_Load()* is responsible to perform the following:

A. Register the HAM module. The HAM registers its module by calling **NPA_Register_HAM_Module()**. This API sets up the general environment necessary for the HAM to become operational and makes it possible for the HAM to allocate and register any resources it may need.
It is within the context of **NPA_Register_HAM_Module()** that the HAM's *NPAHandle* is assigned a value, and that the following HAM entry points get registered with the NWPA:

HAM_Check_Option()
HAM_Software_Hot_Replace() [1]
HAM_ISR()
HAM_Execute_HACB()
HAM_Abort_HACB()

> **Note:** If the HAM will support multiple adapters, it should call **NPA_Register_HAM_Module()** for each instance it will support. This API accepts a HAM-generated instance number as an input parameter. This instance number should correspond to the adapter card instance being supported by the HAM. A separate instance number is necessary to register different hardware options for each adapter.

B. Verify bus compatibility.
The HAM can check the host bus type by calling **NPA_Return_Bus_Type()**. The HAM can then verify that the bus type is compatible with the type it supports.. If the HAM is NBI aware and

[1] ***HAM_Software_Hot_Replace()*** is an optional entry to be used only if the HAM is going to support hot replacement. For more information about hot replacement, refer to Section 4.3, Special Topics.

is supporting an adapter designed for a bus architecture that provides configuration information on a per-slot basis, (e.g. EISA, MCA, PCI), then do the following:

1. Build the Product_ID option structure. (For details refer to the **NPAOptionStruct** definition in Chapter 6.)
   Option Name = a length preceded & null terminated name of the option (PRODUCT ID)
   Option Type =Product_ID_Option
   Parameter0= BusType
   Parameter1=Pointer to an array of bytes that contains the architecture specific Product ID information
   Parameter2=The size of the array pointed to by Parameter1.
   OptionFlag= USE_THIS_OPTION
   String = Null

2. Add the option using **NPA_Add_Option()** with Instance = 0

3. Parse the option using **NPA_Parse_Option()**.
   During the context of **NPA_Parse_Option()**, the HAM's *HAM_Check_Option()* routine is repeatedly called passing it the option struction with output parameters as follows:
   Parameter0=BusTag
   Parameter1=Slot
   Parameter2=Unique ID
   For each adapter instance found, the *HAM_Check_Option()* routine should store the return information in a configuration table for use later.

4. If the HAM supports the "SLOT=" option, do the following:
   a. Build a Slot Option structure.
   b. Add the Slot Option using **NPA_Add_Option()**
   c. Parse the option using **NPA_Parse_Option()**.
   If a "SLOT=" option matching one of the elements in the HAM's configuration table is present on the command line, the HAM calls **NPA_Get_Card_Config_Info()** passing it the bus tag and unique ID given during the parse of the Product ID option. This routine returns the bus specific configuration information associated with the target adapter.

C. Create a select-list of desired options.
   Options are command line keywords that set operational states such as NWDIAG [2], or specify hardware resources such as interrupts, DMA

[2] For more information about NWDIAG, see Section 4.3.5.

channels, ports, memory decoding, and custom parameters.

For each of these applicable resources, the HAM creates an options list by filling out an instance of an **NPAOptionStruct** with **Flags** set to USE_THIS_OPTION and **Parameter2** = BusTag (if NBI aware) and calling **NPA_Add_Option()**. During the context of **NPA_Add_Option()**, the NPA copies the option information and constructs a "select-list" of valid options for the HAM and adapter. To completely build the option list, the HAM should iteratively fill out the **NPAOptionStruct** instance and call **NPA_Add_Option()** for each option type it desires. Since the NWPA maintains its own copy of option information in constructing the select-list, the HAM can reuse the same **NPAOptionStruct** instance for each call to **NPA_Add_Option()**.

> **Note:** For hardware resource options, if **NPA_Add_Option()** returns a non-zero value, it indicates that the option is already reserved. Also, the NWPA will not add the option to the HAM's select-list.

D. Parse the load command line for specified options.
The HAM calls **NPA_Parse_Options()** to cause the NWPA to match options specified on the command line with those in the HAM's select-list. In turn, **NPA_Parse_Options()** iteratively calls the HAM's *HAM_Check_Option()* entry point for each match it finds. *HAM_Check_Option()* either accepts or rejects the selected option. Each time *HAM_Check_Option()* accepts an option, the NWPA places it on a "use-list."

If there is an option on the command line that does not match anything in the HAM's select-list, it is ignored. However, if after parsing the command line the NWPA finds residual options in the HAM's select-list, it either prompts the user for the options or discards them depending on the bits set in the **Flags** field of each option's **NPAOptionStruct**.

> **Warning:** Hardware options are not physically registered during the context of **NPA_Parse_Options()**. Therefore, the HAM should not try to physically access a resource when its *HAM_Check_Option()* entry point is called during this context.

E. Register the options in the HAM's use-list.
The HAM registers the parsed options (options specified in its use-list) by calling **NPA_Register_Options()**. This API accepts the instance

number introduced in the note of step 1.A. **NPA_Register_Options()** uses this number to associate the group of options being registered with a particular instance of an adapter managed by the HAM.

NPA_Register_Options() physically registers the hardware resources in the use-list making them available to the HAM. Also, similar to the parse phase in step 1.D, **NPA_Register_Options()** iteratively calls the HAM's *HAM_Check_Option()* entry point for each registered option, this time allowing the HAM to physically verify the resource or set internal flags to set an operational mode. An example of setting an internal flag to determine an operational mode is the NWDIAG option introduced in section 4.3.5.

Another reason why *HAM_Check_Option()* gets called during option registration is to provide the HAM with return information pertinent to the option. For example, the memory decode option that pages in memory-mapped I/O space to the system returns a logical address to the HAM. This type of return information is given to the HAM through *HAM_Check_Option()* when it is called under the context of **NPA_Register_Options()**. The actual information is found in the Parameter2 field of the **NPAOptionStruct** pointed at by the entry point's *Option* input parameter.

> **Note:** Steps 1.C - 1.E describe the general paradigm for registering hardware and configuration options. For more detailed information and actual registration examples, refer to the **NPAOptionStruct** in Chapter 6.

F. Allocate memory resources.
The HAM allocates any memory buffers it may need by calling **NPA_Allocate_Memory()**.

G. Schedule the HAM's timeout routine.
The HAM schedules its timeout routine, *HAM_Timeout()*, by calling **NPA_Spawn_Thread()**. The HAM will use this routine to recover from a hung-device condition. *HAM_Timeout()* monitors the elapsed time of a HACB request as specified in the HACB's *TimeoutAmount* field.

> **Note:** **NPA_Spawn_Thread()** is a one-shot API.

H. Reset and make the adapter ready.
The HAM must ensure that is operational and ready to accept HACB requests before going to step I.

I. Activate the host bus.

The HAM calls **HAI_Activate_Bus()** to activate an instance of a host bus. This API requires an exchange of handles that identify the bus instance. The HAM passes a unique handle (*HAMBusHandle*) it generates to identify the bus instance as an input parameter. Then, the NWPA returns its own unique handle (*NPABusHandle*) it will use to identify the bus instance as an output parameter. The HAM must call **HAI_Activate_Bus()** for each bus instance it will manage.

J. Return load status.
If the HAM loaded successfully, *HAM_Load()* should return zero. If the load was unsuccessful, it should do the following:

1. Cancel *HAM_Timeout()* by calling **NPA_Cancel_Thread()** passing it the exact same arguments used in setting up the timeout routine.

2. Return all allocated memory by calling **NPA_Return_Memory()**.

3. Unregister all hardware options by calling **NPA_Unregister_Options()**.

4. Unregister the module if the HAM is to be unloaded by calling **NPA_Unregister_Module()**.

Warning:	**NPA_Unregister_Module()** should not be called if the HAM is only erroring out of the registration of a single instance of itself, but it intends to continue supporting other instances. If it is called, all pending I/O for this HAM will be aborted.

5. Return -1.

If at any time during initialization and registration an uncorrectable error occurs, the HAM must return its resources and back out from the point it reached. For example, if the bus type returned in 1.B is not compatible with what the HAM supports, the HAM only needs to call **NPA_Unregister_Module()** to error out. If the HAM progressed as far as 1.F in the sequence, then the HAM would need to return memory, unregister options, and then unregister the module.
After the HAM is loaded and registered with the OS, it must be ready to receive the following sequence of **HACBType=0** requests:

1. *HAM_Return_HAM_Info()*
2. *HAM_Scan_For_Devices()*
3. *HAM_Return_Device_Info()*

The first request, *HAM_Return_HAM_Info()*, is initiated by the NWPA so that it can get HAM-specific information and add the HAM to its object database. The second request, *HAM_Scan_For_Devices()*, is either initiated from a command-line directive in a startup file or by the systems operator at the console. This HAM function spawns a blocking thread, using **NPA_Spawn_Thread()**, that performs a host bus scan for attached devices. The spawned thread builds the HAM's device list, creates a unique **DeviceHandle** for each device, and fills out an instance of a **DeviceInfoStruct** for each device. The third request, *HAM_Return_Device_Info()*, is initiated by the NWPA so that it can get device-specific information and add an object for each device to its database. The NWPA initiates a find-first-find-next sequence of these requests until information about each device is returned. The return information for each request is in a form defined by the **DeviceInfoStruct**.

> **Note:** The HAM will receive the above sequence of requests for each bus instance it registered at load-time using **HAI_Activate_Bus()**.

4.2.2 Processing HACB I/O Requests

HAM_Execute_HACB() is the HAM's entry point for receiving and executing HACB I/O requests, and it has non-blocking context. This entry point is registered with the NWPA during **NPA_Register_HAM_Module()**. The following steps show the sequence of events for processing a HACB I/O request:

1. The NWPA calls *HAM_Execute_HACB()* passing it *HAMBusHandle* and a pointer to a HACB as input parameters. *HAM_Execute_HACB()* does the following:

   A. Identify the host bus instance.

      The HAM identifies the target bus instance based on the value contained in the *HAMBusHandle* input parameter. The HAM originally generated this *HAMBusHandle* value and registered it for the bus instance using **HAI_Activate_Bus()** at load-time. If the HAM is managing only one host bus, the value in the *HAMBusHandle* input parameter will be the same for all requests. If the HAM is managing multiple buses (when the adapter supports more than one bus or the HAM is managing multiple buses spanned over multiple adapters), the *HAMBusHandle* value is unique to each bus instance.

   B. Identify the target device.

The HAM identifies the target device based on the value contained in the **DeviceHandle** field of the **HACBStruct** instance pointed at by the HACB input parameter. The HAM originally generated this **DeviceHandle** value during the scan thread scheduled by *HAM_Scan_For_Devices()* and reported it to the NWPA during *HAM_Return_Device_Info()*.

C. Execute or queue the request.
If the adapter can immediately accept the request, the HAM should translate the HACB request information into a protocol-specific command block, issue the request to the adapter, and then return to the calling process. If the adapter cannot immediately accept the request, the HAM should place it in an internally managed queue for the target device and return to the calling process.

Note: The NWPA expects the HAM to provide a queue for each device it manages. If a request cannot be immediately issued to the adapter during the context of *HAM_Execute_HACB()*, the HAM must place the request in the target device's queue and return to the calling process. The HAM must pull requests from the queue and execute them at another time during another thread. For a detailed specification on device queue behavior and how it affects *HAM_Execute_HACB()*, see section 4.3.1.3.

2. The target device services the request and at completion, the adapter generates a hardware interrupt. The NetWare OS fields the interrupt and routes servicing to the HAM's ISR entry point, *HAM_ISR()*, passing it *IRQLevel* as an input parameter. *HAM_ISR()* has non-blocking context and does the following:

   A. Determines which device to service.
   The HAM must provide the logic to determine which adapter its managing caused the interrupt.

   B. Ensures that data is transferred correctly.
   If the HAM's adapter does DMA or bus-mastering, the ISR is not concerned with physical data transfer because the transfer buffer was specified when the request was issued to the adapter. However, for host buses that rely on programmed I/O, the ISR needs to perform the transfer. The HACB provides both the virtual (logical) and physical (absolute) addresses of the request's I/O buffer. These addresses are found in the

HACB's **virtualAddress** and **physicalAddress** fields, respectively.

C. Posts completion status to the HACB.
Once the request is complete, *HAM_ISR()* must post the HACB's completion status to its **hacbCompletion** field. Valid completion status values are listed in Chapter 3 under the description of the **hacbCompletion** field. These status codes can reflect successful completion of the request, or they can reflect HACB and/or device errors. For processor-independence reasons, this field must be processed as a LONG and manipulation of its contents should be done arithmetically using macros. The HAM should post HACB completion using the following macro:

```
#define SET_STATUS (UpperWord, LowerWord)((UpperWord)<< 16)
| ((LowerWord)& 0xFFFF))
```

D. Completes the HACB.
A HACB request has two possible completion paths depending on whether or not the NWDIAG option was specified on the command line at load-time.

1. If NWDIAG was not specified, *HAM_ISR()* performs one of the tasks outlined in steps a through c below.

2. If NWDIAG was specified, then *HAM_ISR()* calls the diagnostic API, **HAI_PreProcess_HACB_Completion()**. This API has non-blocking context, and it gives a diagnostic NLM a hook for snooping on post-device-processed HACB information prior to it being completed and passed to the CDM layer. The diagnostic NLM can alter a HACB's completion status (**hacbCompletion**); thereby, introducing false errors to test system behavior. After **HAI_PreProcess_HACB_Completion()** returns, *HAM_ISR()* performs one of the tasks outlined in steps a through c below.

   a. If the request completes successfully and the **Freeze_Queue_Flag** in the HACB's **Control_Info** field is cleared (zero), *HAM_ISR()* completes the HACB by calling **HAI_Complete_HACB()**, initiates the execution of the next HACB in the device's queue, and returns to the calling process.

b. If the request completes successfully and the **Freeze_Queue_Flag** in the HACB's **Control_Info** field is set (one), *HAM_ISR()* completes the HACB by calling **HAI_Complete_HACB()** , freezes the device queue, sets the most-significant-bit in the HACB's **hacbCompletion** field so that the callback CDM can know the device's post-completion queue state, and returns to the calling process. The device's queue must remain frozen until either the HAM receives a *HAM_Unfreeze_Queue* for that device, or it receives a priority HACB request for that device.

c. If there is an error, *HAM_ISR()* completes the HACB by calling **HAI_Complete_HACB()**, freezes the device queue, sets the most-significant-bit in the HACB's **hacbCompletion** field so that the callback CDM can know the device's post-completion queue state, and returns to the calling process.

The device's queue must remain frozen until either the HAM receives a *HAM_Unfreeze_Queue* for that device, or it receives a priority HACB request for that device. Additionally, the low-order 31 bits of the **hacbCompletion** field must remain intact to the value set in step 2.C above. This value indicates the type of error that occurred.

Note: *HAM_ISR()* has other responsibilities regarding the device queue. For a detailed specification on device queue behavior and how it affects *HAM_ISR()*, see section 4.3.1.3.

4.2.3 Aborting a HACB Request

HAM_Abort_HACB() is the HAM's entry point for aborting I/O requests, and it has non-blocking context. This entry point is registered with the NWPA during **NPA_Register_HAM_Module()**. The following shows the sequence of events for aborting a HACB request:

The NWPA calls *HAM_Abort_HACB()* passing it *HAMBusHandle*, a pointer to a HACB, and Flag as input parameters. *HAM_Abort_HACB()* does the following:

A. Identify the host bus instance.
The HAM identifies the target bus instance based on the value contained in the *HAMBusHandle* input parameter. The HAM originally generated this *HAMBusHandle* value and registered it

for the bus instance using **HAI_Activate_Bus()** at load-time. From this *HAMBusHandle*, the HAM should be able to access its device list for the target bus instance.

B. Locate the HACB to be aborted. The NWPA passes a pointer to the HACB that is to be aborted, which the HAM uses to locate the associated request.

C. Determine the appropriate abort action.
HAM_Abort_HACB() has three possible actions depending on the value of the *Flag* input parameter passed by the NWPA.

1. If *Flag* = 0 (unconditional abort case), then *HAM_Abort_HACB()* does ONE of the following:
   a. If the HACB is still in the device queue (clean abort case):
      1. Unlinks the HACB from the device queue.
      2. Places the ABORT code (0x0004) in the upper WORD of the HACB's **hacbCompletion** field and 0x0000 in the lower WORD using the following macro:

```
#define SET_STATUS (UpperWord,LowerWord) ( (UpperWord )<< 16)
| ((LowerWord) & 0xFFFF))
```

      3. Completes the HACB by calling **HAI_Complete_HACB()** passing it the HACB's *HACBPutHandle* as an input parameter.
      4. Returns 0 to notify the NWPA that this was a clean abort, meaning that the HACB was aborted prior to being physically processed by the device.

   b. If the HACB is currently being processed by the device (dirty abort case):

      1. Tags the HACB for abortion at a later time by placing the ABORT code (0x0004) in the upper WORD of the HACB's **hacbCompletion** field and 0x0000 in the lower WORD using the following macro:

```
#define SET_STATUS (UpperWord, LowerWord) ( (UpperWord )<<
16) | ((LowerWord) & 0xFFFF))
```

      Under the dirty abort case, *HAM_Abort_HACB()* must not complete the HACB. The HAM will complete the HACB at a later time during its ISR.

      2. Returns -1 to notify the NWPA that the HACB could

not be cleanly aborted.

3. *HAM_ISR()* must intercept the aborting HACB and do the following:

   a. Place the ABORT code (0x0004) in the upper WORD of the HACB's **hacbCompletion** field and 0x0000 in the lower WORD using the following macro:

```
#define SET_STATUS (UpperWord, LowerWord) ( (UpperWord )<<
16) | ((LowerWord) & 0xFFFF))
```

   b. Complete the HACB by calling **HAI_Complete_HACB()** passing it the HACB's *HACBPutHandle* as an input parameter.

2. If *Flag* = 1 (conditional abort case), then *HAM_Abort_HACB()* does one of the following:

   a. If the HACB is still in the device queue (clean abort):
      1. Unlinks the HACB from the device queue.
      2. Places the ABORT code (0x0004) in the upper WORD of the HACB's **hacbCompletion** field and 0x0000 in the lower WORD using the following macro:

```
#define SET_STATUS (UpperWord, LowerWord) ( (UpperWord )<<
16) | ((LowerWord) & 0xFFFF))
```

      3. Completes the HACB by calling **HAI_Complete_HACB()** passing it the HACB's *HACBPutHandle* as an input parameter.

      4. Returns 0 to notify the NWPA that this was a clean abort, meaning that the HACB was aborted prior to being physically processed by the device.

   b. If the HACB has already been sent to the device, returns -1 to notify the NWPA that the HACB could not be cleanly aborted. The device queue continues to operate normally.

3. If *Flag* = 2 (check for clean abort case), then *HAM_Abort_HACB()* does one of the following:

   a. If the HACB is still in the device queue (clean abort), returns 0 to notify the NWPA that the indicated HACB can

be cleanly aborted. The device queue continues to operate normally.

b. If the HACB has already been sent to the device, returns -1 to notify the NWPA that the indicated HACB cannot be cleanly aborted. The device queue continues to operate normally.

D. If the HAM cannot find the HACB that is to be aborted in any of its lists, it has lost the HACB, which is a BAD condition. The HAM should then return -2 to notify the NWPA of this condition.

Note:	The results of step D will Abend the server. The HAM must keep track of HACB requests it receives.

4.2.4 Unload-time Deregistration

Unloading of the HAM is initiated by the systems operator at the server console. The following steps show the sequence of events at unload-time.

1. When a HAM is unloaded, the OS first calls the HAM's *HAM_Unload_Check()* entry point passing it *ScreenID* as an input parameter. *HAM_Unload_Check()* has blocking context, and it does the following:

   A. Determines if any applications are using any of the devices managed by the HAM.
   HAM_Unload_Check() calls **NPA_Unload_Module_Check()**, which checks the NWPA's database and returns the status of each device attached to the adapter. *HAM_Unload_Check()* returns the use-status from **NPA_Unload_Module_Check()**.
   NPA_Unload_Module_Check() issues a warning message to the console for each device that is locked. Current I/O to these devices will halt if the HAM is unloaded, and the devices will be deactivated.

   B. Returns the composite device status to the calling process.
   A return value of zero indicates that none of the HAM's devices are in use. A return value greater than zero indicates that one or more of the HAM's devices are in use.

2. If *HAM_Unload_Check()* returns zero, the OS calls the HAMs *HAM_Unload()* entry point. If *HAM_Unload_Check()* returns non-zero, the OS issues a message to the console giving the operator a chance to either cancel or continue the unload. Only if the operator

chooses to continue the unload will the OS call the HAM's *HAM_Unload()* entry point.

The OS calls the HAM's *HAM_Unload()* entry point with blocking context, and it does the following:

A. Causes the NWPA to terminate I/O to the HAM.

   HAM_Unload() terminates I/O to the HAM by calling **HAI_Deactivate_Bus()** immediately upon entry. It is during the context of this API that the application is notified that its link to the device is about to be severed. Therefore, the HAM must remain operational and process requests until **HAI_Deactivate_Bus()** returns control. Once this happens, the HAM is guaranteed not to receive any more HACB requests for that bus instance. *HAM_Unload()* must call **HAI_Deactivate_Bus()** for each bus instance being managed by the HAM.

B. Returns resources back to the system.

   1. Ensure that all outstanding HACB's, if any exist, are cancelled with the appropriate HAM UNLOAD completion code described in Appendix B. This action is really a preventative measure. Theoretically, all of these outstanding HACBs should have been aborted during the context of **HAI_Deactivate_Bus()**.
   2. Cancel all asynchronous events, such as timeout handlers, timers, etc., by calling **NPA_Cancel_Thread()** on each event.
   3. Return memory to the system pool by calling **NPA_Return_Memory()**.
   4. Unregister all hardware options using **NPA_Unregister_Options()**.
   5. Unregister the module using **NPA_Unregister_Module()**.

C. Return 0 if successful, or return -1 if unsuccessful.

4.3 Special Topics

This section discusses special topics relevant to the HAM.

4.3.1 HAM Device Queues

A HAM is required to implement and manage a HACB request queue for each device attached to the adapter it supports. Queue management routines are to be provided by the HAM, and they must implement the behavior outlined in this subsection.

> **Note:** If a HAM supports an adapter that does hardware queuing, it needs to implement whatever measures are necessary to ensure that the queue state protocol discussed in this section is followed.

4.3.1.1 Request Hierarchy

HACBType=0 requests have highest priority and should be executed immediately in the order they are received. **HACBType=0** requests really do not have to be queued since they map to HAM-specific functions that do not require processing by a device. An example would be the request corresponding to *HAM_Return_HAM_Info()*. This function merely returns information about the HAM and can be completed immediately. **HACBType=0** requests can happen at any time, and if they need to queued, they should probably be placed in a separate queue explicitly for HAM functions rather than in a device queue. Other **HACBType** requests do map to specific devices, so they do need to be queued if they cannot be executed immediately . These types of requests can be processed as priority HACBs or has normal HACBs depending on whether the **Priority_Flag** is set in the HACB's **Control_Info** field. The following list shows the execution order of HACB requests in a device queue:

1. Priority HACB requests (**Priority_Flag** is set) have the highest priority in the device queue. Immediately upon receipt of a priority HACB, the HAM should place the HACB at the head of the device queue and issue it to the device.

   Priority HACB requests are generally used by the CDM and Media Manager for diagnostics and error recovery. By setting both the **Priority_Flag** and the **Freeze_Queue_Flag**, the CDM or Media Manager can execute HACB requests in lock-step fashion.

   If multiple priority requests are placed in the queue due to a device busy condition, they are executed on a LIFO basis.

2. Normal HACB requests (**Priority_Flag** is cleared) have the lowest priority. These requests are the most frequent and should be placed at the end of the queue.

> **Note:** Unless the queue contains a HACB request with the **Preserve_Order_Flag** set (this condition is described in the next subsection), the HAM can reorder normal requests in a device queue to optimize performance. However, the routines that handle the reordering must implement fairness so that any one, normal HACB request gets executed in a timely fashion instead of always being pushed to the back of the queue.

4.3.1.2 Preserve-Execution-Order Requests

To support sequential devices where preserving the execution-order of requests is critical, the NWPA defines a **Preserve_Order_Flag** for the HACB. This flag also resides in the HACB's **Control_Info** field. HACB's with this flag set are a special-case of normal HACB requests. When the HAM receives a HACB with the **Preserve_Order_Flag** set, it must always place the request at the back of the device queue.

The **Preserve_Order_Flag** indicates that all requests currently positioned in front must be executed before the preserve-order request, and any new requests that get placed in the queue must be executed after the preserve-order request. If multiple preserve-order requests are in the queue, this paradigm must be extended to maintain the prescribed behavior.

Priority HACB requests, as explained in the previous subsection, are the one exception to the preserve-order rule. Priority HACB requests always get placed at the front of the queue even if preserve-order requests are present.

4.3.1.3 Queue State

This subsection defines the state conditions of a device queue. A device queue has two states: frozen or unfrozen. A frozen state means that the issuing of requests in the queue to the adapter/device is halted; however, queue management does not stop. If the HAM receives any new requests for a frozen queue, it still must accept them and place them in their proper sequence in the queue. An unfrozen state means that requests continue to be issued to the adapter/device.

In the event of an error, the HAM is expected to freeze the queue of the device that caused the error, post the appropriate completion status as prescribed in the description of the HACB's **hacbCompletion** field in

Chapter 3, and complete the HACB using **HAI_Complete_HACB()**.

If the HAM is managing multiple devices, freezing one device queue does no affect the state of any of the others. Also, the HAM must keep the queue frozen until that device either receives a *HAM_Unfreeze_Queue()* request or a priority request. After receiving one of these requests, the queue starts up again.

During normal I/O operations, the HAM controls the queue state from two different time points: the HACB receive-time point and the HACB completion-time point. The receive-time point is the HAM's entry point that receives HACB requests prior to sending them to the adapter/device (*HAM_Execute_HACB()*). The completion-time point is the HAM's entry point that completes the HACB after it has been processed by the device (*HAM_ISR()*). Table 4-1 summarizes queue state management.

Table 4-1: Device Queue States

At HACB Receive Time *HAM_Execute_HACB()*	**At HACB Completion Time** *HAM_ISR()*
if (*Priority_Flag* set) { Place at head of queue; Issue to adapter/device; if (*Freeze_Queue_Flag* set) Freeze device queue; return; } **/* For any other HACB at head of queue */** else { if ((queue is empty) && (adapter/device notBUSY)) { Issue to adapter/device; if (Freeze_Queue_Flag set) Freeze device queue; return; } else Place at back of queue; return; }	if (error occurred) { Determine device queue state (frozen/unfrozen) and error code according to Appendix B; Set *hacbCompletion* code from current device queue status and current error code; Complete HACB back to the NWPA; if (device queue is unfrozen) Issue next request in queue to adapter/device; return; } Set the current error code to Successful_Completion; **/* No error occurred, check for the implicit UNFREEZE_QUEUE within HACB */** if ((Priority_Flag == set) && (Freeze_Queue_Flag == clear)) UNFREEZE the queue for the current device; else Do NOT modify current queue state status for this device; Set *hacbCompletion* code from current device queue status and current error code; Complete HACB back to the NWPA; if (device queue is unfrozen) Issue next request in queue to adapter/device; return;
if (HAM's *HAM_Unfreeze_Queue()* function is called) { Unfreeze target device queue; Issue next request in queue to adapter/device; return; }	

> **Note:** The HAM should never freeze a device queue on a request that was dirty aborted. If during its ISR, the HAM detects a HACB request that was dirty aborted, the HAM should complete the HACB with the abort completion code, even if the request generated an error.

4.3.2 Asynchronous Event Notification

The NWPA provides a mechanism for CDMs to request that HAMs notify them of asynchronous events. These include hardware events such as a bus reset, device reset, or a device attention. The CDM requests for asynchronous event notification (AEN) by issuing a **HACBType=0** request to the HAM. **HACBType=0** means that the HACB's union command area is defined by the host adapter command structure.

> **Note:** A HACB request for asynchronous event notification is also referred to as an AEN HACB.

To register for asynchronous event notification, the CDM must issue an AEN HACB with the following information in the HACB's host adapter command block:

Function = 5
Parameter0 = Bitmap indicating the type of events for which the CDM wants to be notified. Currently, the NWPA recognizes the following:
- 0x00000001 Bus reset
- 0x00000002 Device reset
- 0x00000004 Device attention [3]
- 0x00000008 Adapter reset
- 0x00000010 Reserved
  to
  0x80000000

Parameter1 = 0
Parameter2 = 0

These requests must be issued on a per device basis, meaning that the CDM will provide the correct device handle for the device it wants monitored. The device handle is placed in the AEN HACB's *DeviceHandle* field.
The CDM builds the bitmap indicating the events it wants to be informed of, places the bitmap value in the *Parameter0* field of the AEN HACB, and executes the request by calling the NWPA routine

[3] In order for a HAM to detect a device attention, the CDM must first issue commands that will program the device to issue the alert.

CDI_Non_Blocking_Execute_HACB(). This API requires the CDM to provide a pointer to a callback routine as an input parameter.

The HAM receives the AEN HACB through its *HAM_Queue_AEN_HACB()* HAM function and maintains it in a local holding area associated with the target device until an event occurs. These AEN HACBs should not be placed in the device queue since they do not represent I/O requests that need device processing.

After an AEN event occurs, the HAM will check to see if the value in *Parameter0* represents an event that a CDM wants to be notified of. If so, the HAM will freeze the device queue, set a bitmap value in the HACB's **Control_Info** field to indicate which event(s) occurred, place the AEN code (0x80080000) in the HACB's **hacbCompletion** field, and complete the AEN HACB by calling **HAI_Complete_HACB()**. The bit definitions for the return bitmap value are the same as those defined for the *Parameter0* field.

> **Note:** If no CDM has registered for a specific AEN event that occurs, the queue state will not change.

The HAM must be ready to accept multiples of these requests per device. When an event occurs, the HAM should complete all AEN HACBs registered for that event for the target device.

4.3.3 Reentrance

To support multiple adapter cards compatible with a HAM's type, a HAM may be declared reentrant in the definition (.DEF) file. Doing so allows the HAM to maintain multiple instances of itself; however, only one code image is maintained in the file server's memory. Each subsequent LOAD command for this same HAM calls *HAM_Load()* creating a new instance of itself. For each instance, the HAM receives a pointer to the command line so that it can establish a unique I/O configuration for that instance. The HAM must maintain an internal counter to track its instance number; and, when the time comes for the HAM to be unloaded, it must deactivate each bus instance managed by the HAM by calling **HAI_Deactivate_Bus()** and free all allocated resources.

4.3.4 Hot Software Replacement

Hot software replacement is an NWPA feature that provides for dynamic replacement of one version of a HAM driver with an updated version. Replacement is dynamic because the swap can be done without having to

dismount any volumes or disrupt the I/O channel for a lengthy period of time.

Hot software replacement only applies to HAMs from the same manufacturer, which means that the HAM being replaced (old HAM) and the new HAM must have the same vendor ID (*NovellAssignedModuleID*). The vendor ID is assigned to a manufacturer by Novell Labs.

Presently, NWPA does not require HAMs to implement hot software replacement. However, due to market reaction when the feature was demonstrated, this requirement may change. Therefore, it is highly recommended that HAMs implement this feature, as it will be a tremendous value-add to customers.

4.3.4.1 Overview

The objective in hot software replacement is to establish an I/O channel in the new HAM that looks logically identical to the one in the old HAM, from the perspective of NWPA. Logically identical means that the new HAM must show all the same devices, adapters, hardware resources, and handles that were used in the old HAM. Details on the I/O channel are discussed later in this section.

The method used to get configuration information is for the new HAM to pass a series of vendor-specific-messages to the old HAM and the old HAM responding to each message by returning the appropriate configuration information in a buffer provided in each message. The structure of these messages is not defined in NWPA. Their structure and meaning are defined by the manufacturer of the HAMs.

The new HAM sends a vendor-specific message by calling **NPA_Exchange_Message()**. This routine routes the message to the old HAM by calling its *HAM_Software_Hot_Replace()* routine. The old HAM must provide this entry point for hot replace to work. The old HAM keys off of some field (or fields) in the message to determine what information is being asked for, and then copies it into the message buffer. All of this message exchanging is done during the context of the new HAM's *HAM_Load()* (initialization) routine. After the new HAM successfully gets all the channel information it needs to be operational, it succeeds its *HAM_Load()* routine by giving a return value of zero. At that point, NWPA routes I/O through the new HAM.

To understand the general details of hot software replacement, it is necessary to review the elements that constitute an I/O channel in the perspective of NWPA.

NWPA routes I/O by identifying the target device and the bus to which the target device is attached. The identification is done through a series of handles, some generated by the NWPA module and some generated by the HAM. The following is a list of these handles as discussed in this specification:

NWPA Generated	HAM Generated
NPABusHandle	**HamBusHandle**
	DeviceHandle

The **NPABusHandle** and the **HamBusHandle** are the handles used to identify a target bus. There are two handles given to guarantee uniqueness, and they are exchanged during **HAI_Activate_Bus()**. As indicated above, the HAM generates one of the handles and the NWPA generates the other. When the NWPA passes HACBs to any of the HAM's entry points, it will pass the HAM-generated handle to identify the target bus. In the reverse direction, when the HAM indicates a target bus to any of the NWPA routines, it passes the NWPA-generated handle.

The third handle in the I/O channel is the HAM-generated **DeviceHandle**. This is the HAM's unique identifier for a particular device on a given bus. The **DeviceHandle** only needs to be unique within the devices attached to its bus. All HACBs targeted for specific devices will supply the corresponding HAM-generated **DeviceHandle**.

For hot software replacement to work, the new HAM must use all the same handle values that the old HAM used so that NWPA's perspective of the channel does not change. Otherwise, the channel will be disrupted to the point where NetWare volumes associated with the channel will dismount. This is an extremely important point. Since all handle values must be maintained between the swapping modules, and since the two modules are exchanging configuration information across separate, protected memory domains, these handles *cannot* be implemented as memory pointers.

In addition to these handles, hardware resources such as ports, IRQs, and DMA channels are components to the NWPA-HAM I/O channel. When a HAM registers for a hardware resource using **NPA_Register_Options()**, NWPA registers the resources in behalf of the HAM. As far as NetWare is concerned, NWPA owns the resources and merely lends them to the HAM. This facilitates hot software replacement, because neither HAM has to worry about unregistering or registering hardware resources during the swap. Doing so would dissolve the I/O channel from the OS's perspective, and dismount any volumes associated with the channel. The new HAM only needs to find out what resources the old HAM was operating with and configure itself the same way. Once the hot swap is finished, NWPA

redirects the use of those hardware resources to the new HAM. As far as the OS is concerned, nothing has changed.

4.3.4.2 Flow of Events

This section steps through the flow of events in the hot software replace paradigm, describing fundamental concepts on how to implement it. For an example of one specific implementation method, refer to the sample source code for SCSIPS2.HAM in the driver development kit. This kit is on compact disc distributed by Novell Labs.

To see how SCSIPS2.HAM initializes itself for hot replace, look at the DriverStart procedure in the file HAMSCSI.386. To see how SCSIPS2.HAM, acting as a new HAM, requests configuration information, look at the DoHotReplace procedure in the file HAMHOT.386. To see how SCSIPS2.HAM, acting as an old HAM, gives configuration information, look at the ExchgPipe procedure in the file HAMHOT.386.

1. The HAM is loaded invoking its *HAM_Load()* routine. At the beginning of this routine, the HAM calls **NPA_Register_HAM_Module**(). All of the remaining steps occur during the context of the new HAM's *HAM_Load()* routine.

2. NWPA checks its list of existing modules to determine if this is a new load of a module or a reentrant load. A new load means that the module's *LoadHandle* does not match the *LoadHandle* of any other module in the list.

3. If this is a new load, then NWPA checks for a possible hot replace candidate. A hot replace candidate is an existing module that has the same *NovellAssignedModuleID* as the module being loaded:

```
(newModule->LoadHandle!=candidateModule->LoadHandle)&&
(newModule->NovellAssignedModuleID==candidateModule->NovellAssignedModuleID)
```

Important: The NetWare NLM loader assigns the HAM's *LoadHandle*, and it uses the name of the module to calculate it. In order for hot replace to work, the new HAM cannot have the same name as the old HAM. Otherwise, the *LoadHandles* will not be unique, and the load of the new HAM will be mistakenly taken as a reentrant load of the existing HAM. As a suggestion for *LoadHandle* uniqueness, include a revision number in the module name.

4. If a candidate is found, **NPA_Register_HAM_Module**() returns a value of 1.

5. The new HAM then determines if it will do hot replace or fail its load leaving the existing HAM in the I/O channel. If the HAM decides to do hot replace, it should check the command line for any possible options, such as NWDIAG, and then start requesting configuration information using its vendor-specific messaging scheme. The new HAM sends these messages to the existing HAM by calling **NPA_Exchange_Message()**. Each message should include a buffer space sufficient to receive the requested information.

6. The existing (or old) HAM receives these messages through its *HAM_Sofware_Hot_Replace()* entry point. The old HAM determines what information is being asked for and copies it into the message buffer.

   In the SCSIPS2.HAM example, there are fields at the beginning of the message indicating operations to perform. SCSIPS2.HAM, when it acts as a new HAM initiating hot replace, first asks for all the adapters being supported on a find-first-find-next basis. Refer to HAMHOT.386, procedures DoHotReplace and ExchgPipe, for more details.

7. The new HAM continues passing messages until it gets all the information it needs to be operational. The new HAM needs to make sure that it is using all the handle values that the old HAM used and the same hardware resources. The new HAM does not need to register for these resources, NWPA automatically redirects them to the new HAM. When the new HAM is ready to take over function of the old HAM, the new HAM must make a call to **NPA_Register_Options()** to actually perform the module switch.

8. After the new HAM is in place, it succeeds its *HAM_Load()* routine by returning zero. At that point, the new HAM is in the I/O channel handling HACB requests. The old HAM is dormant and can be unloaded by a user. It is important that the old HAM does not try to unregister its hardware resources (**NPA_Unregister_Options()**) during its *HAM_Unload()* routine. NWPA will manage and direct them appropriately.

4.3.5 Diagnostics

For certification purposes through Novell Labs, a HAM is expected to support a diagnostics command line option. The command line keyword that turns this option on is NWDIAG. The HAM must do the following to setup this option:

1. Build an **NPAOptionStruct** with the following information in its fields (refer to Chapter 7):

   Name ="0x6NWDIAG0x0" /* Length preceded string that is also zero terminated */
   Parameter0 = 0
   Parameter1 = 0
   Parameter2 = 0
   Type = 0
   Flags = 0
   String = "NWDIAG" /* ASCII string */

2. Call **NPA_Add_Option()** with a pointer to the **NPAOptionStruct** above as an input parameter.

3. Call **NPA_Parse_Options()** which in turn calls *HAM_Check_Option()* if NWDIAG was found on the command line.

4. Within the context of *HAM_Check_Option()*, check the *Name* and *String* fields of the **NPAOptionStruct** input parameter to verify the found option.

5. Sets an internal flag to turn on the HAM's diagnostic mode.

When the diagnostic flag is set, it directs the HAM to detour its normal HACB completion path. Typically, the HAM, within the context of *HAM_ISR()*, performs the following:

1. Transfers device information from a custom control block (CCB) to the appropriate fields in a HACB including HACB completion (status) information.

2. Determines if the device queue needs to be frozen (freeze queue if an error occurred during request processing).

3. Completes the HACB by calling **HAI_Complete_HACB()**. However, if diagnostics is turned on, the HAM is required to shim a call to **HAI_PreProcess_HACB_Completion()** between steps 1

and 2 above. This API allows a diagnostics NLM the opportunity to snoop or alter HACB information after being processed by a device.

4.3.6 Error Handling and Auto Error Sense

Auto Error Sense is a generic phrase describing the way in which error sense information is automatically returned with an I/O request for a given bus protocol. As an example, for SCSI this phrase refers to auto REQUEST SENSE. Some adapter boards support this feature and others do not. The HAM, during its load-time initialization, is responsible for determining whether or not the feature is to be used. There are three fields in the HACB structure (**HACBStruct**) and one in the **DeviceInfoStruct** that provide NWPA support for auto error sense. The following is a list of these fields:

Fields in the HACBStruct:

```
LONG ErrorSenseBufferLength;
void *VErrorSenseBufferPtr;
void *PErrorSenseBufferPtr;
```

Field in the DeviceInfoStruct:

```
LONG AttributeFlags;
```

If auto error sense is going to be used, then the HAM needs to indicate this by setting the **Auto_Error_Sense_Flag** (0x00000040) in the **AttributeFlags** field of the **DeviceInfoStruct** associated with each device attached to the adapter. The HAM reports the **DeviceInfoStruct** information to the NWPA during *HAM_Return_Device_Info()*. For a device error under the auto error sense case, the HAM must ensure that the sense information gets placed properly into the HACB's error sense buffer. The error sense buffer is defined by the NWPA's **ErrorSenseInfoStruct**, and its length is run-time variable according to the CDM that allocated it. The following is the structure's ANSI C definition:

```
struct ErrorSenseInfoStruct
{
   LONG NumberBytesRequested;
   LONG NumberBytesReturned;
   LONG Reserved[2];
   BYTE ErrorSenseData[1];
 } ;
```

For a description of its fields and its run-time length variability, refer to the structure's reference information in Chapter 7. Also, the NWPA provides the HAM with both virtual and physical (absolute) addresses of the auto error sense buffer, and the buffer is guaranteed to be physically contiguous in memory. These factors should accommodate any transport protocol. The HAM needs to make some special considerations in addition to the method

of error handling this specification already prescribes [4]. These considerations must be dealt with prior to completing the HACB that caused a device error. They are as follows:

- The HAM determines how many bytes of sense data the issuing CDM desires by reading the value in the **NumberBytesRequested** field of the auto error sense buffer. The HAM then builds its adapter-specific command block accordingly, and issues it to the device.

- If less than what the CDM requested, the HAM places the number of error sense bytes that the device actually returned in the **NumberBytesReturned** field. The following formula shows this concept:

```
NumberBytesReturned = min(NumberBytesRequested,
BytesReturnedByDevice);
```

The following assumptions apply to the above formula:

- The CDM must be informed when the length of the sense information returned by the device is less than what the CDM requests.

- The CDM is not concerned with any additional sense information beyond the amount it requested.

4.3.7 Scanning Specific Target IDs and LUNs (Public and Private Devices)

The HAM is responsible for detecting devices attached to the adapters it manages and reporting these devices to the NWPA. The NWPA invokes these tasks through the following HAM functions:

- *HAM_Scan_For_Devices()*
- *HAM_Return_Device_Info()*

In order for devices to be initially detected and recognized by the NetWare OS, an initial "scan for new devices" command must be issued either at the command line or in a .NCF file. When the OS receives this command, it causes the NWPA to issue a scan message to all HAMs loaded on the server. For SCSI, the initial scan message tells each HAM to scan LUN 0, and only LUN 0, of all its target IDs (SCSI IDs) [5].

---

[4] These methods refer to queue state behavior, posting of the appropriate HACB completion code, etc.

[5] There are three reasons why the OS limits its initial scan to LUN 0: 1. Most SCSI devices come hard-addressed for LUN 0. 2. Some LUN 0 devices reflect themseves on the other LUNs of the target ID. To the HAM, these reflections appear as valid devices even though they are just phantoms. 3. Some devices, such as hard disks, will typically hang if any LUN beyond LUN 0 is probed on the target ID.

To make it possible for devices at LUNs other than zero to be detected and recognized, the NWPA provides its own set of scan messages that the CDM can issue to the HAM. CDMs are given the responsibility of initiating these additional scan messages since they have specific knowledge about the devices. Therefore, they know the conditions when to suspect a companion device on another LUN.

The CDM issues these scan messages as **HACBType=0** requests. **HACBType=0** indicates to the HAM that the HACB's union command area is defined by the host adapter command structure. The CDM then sets values in the HACB according to the scan case (or action) it wants the HAM to perform. The NWPA defines four scan cases. These cases are referred to numerically as either Case 0, Case 1, Case 2, or Case 3 corresponding to the value the CDM sets in the **Parameter2** field of the HACB's host command block. Case 0 scans are issued by the the OS, and Cases 1 - 3 scans are issued by a CDM.

Through these scan cases, a CDM can also tell the HAM whether to declare a detected device public or private. A public device is one that has its **Private_Public_Flag** (0x80) cleared in the **AttributeFlags** field of its corresponding **DeviceInfoStruct** object. A public device is visible to any CDM that is interested in that device's device type. The NWPA gives all of these CDMs a chance to look at the device and an opportunity to bind to it. When one of these CDMs decides to bind to the device, it receives the HAM-generated DeviceHandle, which is the token necessary for the CDM to issue I/O to the HAM's device.

A private device is one that has its **Private_Public_Flag** set in the **AttributeFlags** field of its corresponding **DeviceInfoStruct** object. A private device is visible only to the CDM that detected it through a specific scan. During the specific scan, the HAM passes the device's DeviceHandle directly to the CDM. By privately owning this token, the CDM has exclusive access to the HAM's device.

> **Note:** For specifics on how to implement these scan cases and for respective paradigm descriptions, refer to *HAM_Scan_For_Devices()* in Chapter 8. This chapter also gives more details regarding private and public devices.

Note: Under the NWPA, the HAM is not expected to differentiate between "real" and "phantom" devices. This responsibility belongs to the CDM.

4.3.8 Automatic Hardware Detection and Driver Configuration

> **Note:** This section introduces a feature that HAMs may be required to implement in the future. The ramifications of this feature are still under investigation; therefore, the real purpose of this section is to introduce the concept so that HAM developers will be aware of this feature and perhaps be able to lay important groundwork in their current code that will make future incorporation of the feature smoother.

Automatic hardware detection and driver configuration, also known as auto-detect/auto-config, is a feature where the HAM automatically "detects" its host adapter hardware, if it exists on the server. If the HAM does not detect its adapter, it fails to load. If the HAM detects its adapter (or adapters), it tries to "configure" the adapter and register the correct options such as ports/slots, interrupts, DMA channels, etc. Once the adapter is configured properly, the HAM will be told to scan its buses for attached devices and return information on those devices. The OS will then build a list of these devices and try to match them with appropriate CDMs via information given in their respective information (.DDI) files. A future version of this specification will layout the guidelines associated with auto-detect/auto-config and define a paradigm for implementing the feature.

4.3.9 Elevator Queuing

The purpose of the elevator queues in the NWPA is to order the requests for each device in such a way that the total request throughput is increased while assuring that all requests are executed within a reasonable time frame. This is accomplished by a filter internal to NWPA which can bind to all hard disks, CDROM devices, magneto-optical disks, and WORM devices, but not to tape devices. This filter also combines requests into scatter-gather requests.

The elevator filter can be enabled or disabled on a per-device basis using the attribute flags within the **DeviceInfoStruct**. When the **Elevator_Off_Flag** is set, the elevator is disabled. When cleared (default setting except for tape devices) the elevator filter is enabled.

The filter can be adjusted for each HAM by using the **ElevatorThreshold** byte in the **DeviceInfoStruct**. At this point, the requests may be single requests, or scatter-gather requests. The elevator threshold is the number of requests the HAM would like to process at any given time. The default value of this threshold is 2 requests, which allows a request at the device and a request waiting in the queue. The filter will not "elevator" any requests until the HAM has at least its threshold number of requests. Once

the HAM has the threshold number of requests, the elevator filter will then accept, organize, and check requests for scatter-gather possibilities. Whenever the number of requests at the HAM falls below the elevator threshold, the filter will send a set of requests to the HAM. This set may contain as many as 15 requests. The filter will then continue to "elevator" the requests until the number of requests at the HAM again falls below the elevator threshold.

The threshold can be increased to allow more requests at the HAM. This may be necessary in some HAMs to prevent "starving" the device or adapter. It is important, however, to not set the threshold too high or the filter will not have the opportunity to optimize the request order or look for scatter-gather opportunities. If the HAM or adapter already performs its own elevator queuing, the elevator filter should be turned off. This will allow the HAM to receive all requests in the order they are sent.

4.3.10 Vendor-Pass Through API for HAMs

This API (**NPA_HACB_Passthru()**) provides applications the ability to communicate directly with an adapter. This provides a vendor with a communications channel to allow for vendor-specific commands/data to be sent to/from the adapter. HAMs must be able to handle all messages sent to it using this API, although the HAM may return an "unsupported function" error if necessary. It is important to understand that if a request is sent down which causes an error and/or queue freeze, the application must clear up the problem and unfreeze the queue if necessary so that the HAM can process commands normally. See the **NPA_HACB_Passthru()** API in Chapter 7 for more details.