Chapter 5 Custom Device Module (CDM)

The Custom Device Module (CDM) is the driver component that understands a specific device (or family of devices) and the commands that control it. The CDM is implemented as a NetWare Loadable Module (NLM). This chapter describes a CDM's function, and it is organized into the following sections:

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

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

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

5.1 CDM Architecture: Entry Points and Routines

This section provides prototypes for the entry points and routines required in a CDM by the NetWare Peripheral Architecture (NWPA). A developer may use these prototypes to plumb the shell of a CDM. 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 CDM is required to provide the following:

- NLM Load/Unload-time Entry Points
- CDM Entry-Points
- Device Control and I/O Routines

Device control and I/O routines are mentioned here because they are crucial to the CDM 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 CDM developers.

Complete functional specifications of the entry points can be found in Chapter 7, and descriptions of the CDM Messages can be found in Chapter 9. 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 CDM developer; however, each entry point must provide the respective functionality described in this guide.

For consistency in referring to these entry points and routines within the text and in code examples, this guide gives each a generic name having a *CDM_* prefix. Whenever an entry point or function with this prefix is encountered, it indicates that the routine is CDM specific. The italic typeface indicates that the name is arbitrary.

5.1.1 NLM Load/Unload-Time Entry Points

A CDM 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 a NetWare compatible linker utility. The prototypes of these entry points, along with their generic names, are as follows:

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

CDM_Load() is the CDM's load-time entry point. *CDM_Load()* is called on a blocking thread. Through this entry point, a CDM 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 load options associated with the module. For a CDM, these options set the operational states of the program module. They do not apply to hardware resources.

CDM_Load() is responsible for allocating any resources needed to make the CDM operational, for configuring the CDM based on the load options specified on the LOAD command line, and for registering the CDM and its I/O entry points with the NWPA.

```
LONG CDM_Unload_Check (LONG ScreenID);
```

CDM_Unload_Check() is the CDM's initial unload-time entry point. The entry point gets called when an UNLOAD command is issued on the CDM. *CDM_Unload_Check()* is called on a blocking thread. *CDM_Unload_Check()* is responsible for checking to see if any of the CDM's devices are currently being used by an application and return use-status. To do this, *CDM_Unload_Check()* returns the use-status returned by **NPA_Unload_Module_Check()**. From this return value, the OS can determine if any of the devices managed by the CDM are in use. If any devices are in use, 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 CDM_Unload (void);
```

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

5.1.2 CDM Entry Points

A CDM must provide additional entry points for the NWPA. The prototypes of these entry points, along with their generic names, are as follows:

```
LONG CDM_Abort_CDMMessage (LONG parameter);
```

CDM_Abort_CDMMessage() is an entry point that the CDM must provide if it internally queues CDM Messages. Each time a CDM queues a message as opposed to immediately building a HACB and initiating its execution, the CDM must call **CDI_Queue_Message()**. For each message it queues, the CDM must pass the address of an abort routine as an input argument to **CDI_Queue_Message()**. This is the routine that the NWPA calls if an abort is issued on its corresponding message.

As in the case of *CDM_Callback()*, the CDM can define multiple abort routines. The term *CDM_Abort_CDMMessage()* is used to generically refer to one or multiple CDM abort routines.

```
LONG CDM_Callback (
    struct SHACBStruct *SHACB,
    LONG npaCompletionCode
     );
```

CDM_Callback() is a callback entry point so the NWPA can inform the CDM of the completion of a non-blocking execution cycle. A non-blocking execution cycle of a HACB is initiated when the CDM calls **CDI_Execute_HACB()**. The address of the callback routine is an input parameter to **CDI_Execute_HACB()**; thus, a callback is registered with the NWPA for each call of **CDI_Execute_HACB()**. Since the callback link is on a per execution basis, the CDM can either have one global callback, or it can define multiple callback routines, and link the one appropriate to the HACB request being executed. In this manual, the term *CDM_Callback()* generically refers to either of these cases.

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

> **Note:** This entry point is optional. The only reason a CDM would need to provide this routine is if it supports load options and intends to parse the LOAD command line for these options.

CDM_Check_Option() is the CDM'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 CDM initialization and again during the actual registration of options. The CDM invokes these two NWPA processes at different points in its load-time entry point, *CDM_Load()*. This entry point is made visible to the system when the CDM registers itself with the NWPA using **NPA_Register_CDM_Module()**.

```
LONG CDM_Execute_CDMMessage (
    LONG CDMBindHandle,
    struct CDMMessageStruct *Msg
    );
```

CDM_Execute_CDMMessage() is the CDM's entry point for fielding CDM message requests. This routine uses a jump table (or some other form of routing) so that the CDM can route the message to the local control or I/O routine designed to field the current message type. The local routine is required to build the appropriate SHACB to accomplish the request. This entry point is made visible to the system when the CDM registers with the NWPA using **NPA_Register_CDM_Module()**.

```
LONG CDM_Get_Attribute (
    LONG cdmBindHandle,
    void *infoBuffer,
    LONG infoBufferLength,
    LONG attributeID);
```

CDM_Get_Attribute() is the entry point from which the NWPA can retrieve registered device attribute information for an application. This entry point gets registered with the NWPA when the CDM registers the attribute by calling **CDI_Register_Object_Attribute()**.

> **Note:** The CDM registers a get-attribute routine with each call to **CDI_Register_Object_Attribute**(). Therefore, the CDM can implement either one routine to handle all get-attribute calls, or distribute the calls through multiple routines. This developer's guide uses *CDM_Get_Attribute()* to generically refer to either case.

```
LONG CDM_Inquiry (
    LONG npaDeviceID,
    LONG npaBusID,
    struct DeviceInfoStruct *DeviceInfo,
    LONG flags,
    LONG cdmHandle
     );
```

CDM_Inquiry() is the entry point where the NWPA passes the CDM information about either an existing device or a device that just came online with a type that matches the device type for which the CDM registered. This entry point is registered with the system when the CDM registers itself with the NWPA using **NPA_Register_CDM_Module**()**.** It is during the context of this entry point that the CDM can issue passive requests (see note below) to the device to determine if it wants to field I/O requests for the device. If the CDM decides to field requests for the device, it informs the NWPA by binding to the device. Note: During the context of this entry point the CDM must not issue any commands that will change the state or mode of the device. Passive requests are those such as the SCSI MODE SENSE command.

```
LONG CDM_Set_Attribute (
    LONG cdmBindHandle,
    void *infoBuffer,
    LONG infoBufferLength,
    LONG attributeID
    );
```

CDM_Set_Attribute() is the entry point by which the NWPA can set a registered device attribute for an application. This entry point gets registered with the NWPA when the CDM registers the attribute by calling **CDI_Register_Object_Attribute**().

> **Note:** The CDM registers a set-attribute routine with each call to **CDI_Register_Object_Attribute**(). Therefore, the CDM can implement either one routine to handle all set-attribute calls, or distribute the calls through multiple routines. This developer's guide uses *CDM_Set_Attribute()* to generically refer to either case.

5.1.3 Device Control and I/O Routines

A CDM must provide routines that translate CDM Messages containing NWPA functions into HACB requests. Some of these NWPA functions deal with device control, and some deal with I/O. Control functions typically modify the state of devices or media objects such as activating a device or formatting media. I/O functions typically handle the movement of data to and from media such as reads and writes. The appropriate CDM routine to field an incoming CDM Message gets called through the routing mechanism the CDM implements in *CDM_Execute_CDMMessage()*.

5.2 Operational Overview

The information in this section builds on the declarations and prototypes given in the previous section by describing a CDM's major functional procedures and their main flow of events. The information provided here should help to add functionality to a CDM 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 CDM entry points and CDI / NWPA support routines can be found in Chapter 7, "Technical Reference for NWPA Routines."
- Functional descriptions of NetWare OS support routines can be found in Chapter 10, "OS Support Routines."

5.2.1 Load-time Initialization and Registration

Loading of the CDM can be initiated in multiple ways: by the systems operator at the server console, by a startup file, or by INSTALL. The following steps show the sequence of events for initializing and registering a CDM at load-time.

1. When a CDM is loaded, the OS calls the CDM's *CDM_Load()* entry point passing it *LoadHandle*, *ScreenID*, and *CommandLine* as input parameters. *CDM_Load()* is responsible to perform the following:
   A. Register the CDM module.
      The CDM registers its module by calling **NPA_Register_CDM_Module**(). This API sets up the general environment necessary for the CDM to become operational and makes it possible for the CDM to allocate and register any resources it may need. It is within the context of **NPA_Register_CDM_Module**() that the CDM's *NPAHandle* is assigned a value, and that the following CDM entry points get registered with the NWPA:
      CDM_Check_Option() (**Optional**)
      CDM_Execute_CDMMessage()
      CDM_Inquiry()

Note: If a reentrant CDM will support a device under different operational states according to options specified on the command line, it should call **NPA_Register_CDM_Module()** for each load-instance of itself.

NPA_Register_CDM_Module() accepts a CDM-generated instance number as an input parameter. The CDM should use this instance number to associate a group of command-line options with its corresponding load-instance. This way the CDM can distinguish which operational states go with which load-instance.

B. (**Optional: Implement if applicable**) Create a select-list of desired command line options to be used with the NWPA's command line parser.

Note: If the CDM does not support command line options or it plans to do its own command line parsing, it should ignore this step.

Options are command line keywords that set operational states for the CDM.

The CDM creates an options list by filling out an instance of an **NPAOptionStruct** and calling **NPA_Add_Option()**. During the context of **NPA_Add_Option()**, the NWPA copies the option information and constructs a "select-list" of valid options for the CDM. To completely build the option list, the CDM 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 CDM can reuse the same **NPAOptionStruct** instance for each call to **NPA_Add_Option()**.

Note: Since the CDM does not interface directly with the hardware, it should not try to reserve hardware options such as interrupts, ports, DMA channels, etc. CDM command line options should only set operational states for the program module.

C. (**Optional: Implement if applicable**) Parse the load command line for specified options.

The CDM can either do its own parsing or use the NWPA's parser. The NWPA's parser is invoked by calling **NPA_Parse_Options()**. This support routine causes the NWPA to match options specified on the command line with those in the CDM's select-list. In turn,

NPA_Parse_Options() iteratively calls the CDM's *CDM_Check_Option()* entry point for each match it finds. *CDM_Check_Option()* either accepts or rejects the selected option. Each time *CDM_Check_Option()* accepts an option, the NWPA places it in a "use-list." If there is an option on the command line that does not match anything in the CDM's select-list, it is ignored. However, if after parsing the command line the NWPA finds residual options in the CDM'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**.

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

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

E. Prepare the CDM to accept I/O.
The CDM must ensure that is operational and ready to accept CDM Messages before going to step F.

F. Activate the CDM.
The CDM calls **CDI_Register_CDM()** to activate itself. This API requires an exchange of handles that identify the CDM. The CDM passes a unique handle (*CDMHandle*) it generates to identify itself as an input parameter. Then, the NWPA returns its own unique handle (*CDMOSHandle*) it will use to identify the CDM as an output parameter.

G. Return load status.
If the CDM loaded successfully, *CDM_Load()* should return zero. If the load was unsuccessful, it should do the following:

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

2. Unregister all command line options by calling **NPA_Unregister_Options()**.

3. Unregister the module by calling **NPA_Unregister_Module()** if the CDM failed its initial load instance.

> **Warning:** **NPA_Unregister_Module()** should not be called if the CDM 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 CDM will be aborted.

4. Return -1.

If at any time during initialization and registration an uncorrectable error occurs, the CDM must return its resources and back out from the point it reached. For example, if the CDM progressed as far as 1.D in the sequence, then the CDM would need to return memory, unregister options if implemented, and then unregister the module.

5.2.2 Inquiring and Binding to a Device

Once the CDM is initialized and registered, the NWPA calls *CDM_Inquiry()*. This routine is blocking, and it is registered with the NWPA during **NPA_Register_CDM_Module()**. The NWPA passes four arguments to *CDM_Inquiry()*: *NPADeviceID*, *NPABusID*, a pointer to a **DeviceInfoStruct** instance, and *Flag*.

NPADeviceID and *NPABusID* are the object IDs of the device and bus as entered in the NWPA's object database. **DeviceInfoStruct** is a pointer to an interface-specific structure that describes the device in enough detail so that the CDM can determine the device type. The HAM is responsible for supplying this information to the NWPA.

Flag is an indicator telling the CDM the type of operation to perform during *CDM_Inquiry()*. It is within the context of this routine that a CDM "binds" to a device. Binding is where the CDM tells the NWPA to route I/O for a particular device through it, and the CDM also registers the control and I/O functions that it will support for the device.

CDM_Inquiry() is responsible to perform the following:

1. Build and maintain a linked list of device objects to which the CDM is bound.

2. Generate a unique handle (*CDMBindHandle*) of type LONG for each object to which the CDM is bound.

3. Check the **DeviceInfoStruct** for the device information and/or perform any necessary device tests for more information necessary to determine if the CDM should bind to the device.

a. If the CDM decides to bind to the device, create an instance of an **UpdateInfoStruct** structure, fill in its fields with the appropriate information found from **DeviceInfoStruct**, and call **CDI_Bind_CDM_To_Object()** passing the appropriate arguments. Add the device along with its **UpdateInfoStruct** structure to the CDM's object list, and return zero. *CDMBindHandle* is used by the CDM to identify device objects in the list and to map to the object's device information. *CDMBindHandle* is a necessary argument in **CDI_Bind_CDM_To_Object()**.

   At the minimum, the following fields of a device object's **UpdateInfoStruct** structure must be filled in so that the NWPA has enough information to be able to use the device:

   - functionmask
   - controlmask
   - unitsize
   - blocksize
   - capacity
   - preferredunitsize
   - activateflag

A description of each of these fields can be found in Chapter 6 under **UpdateInfoStruct**.

b. If the CDM decides not to bind to the device, it does not call **CDI_Bind_CDM_To_Object()**, does not add the device to the linked list, and returns -1.

CDI_Bind_CDM_To_Object() updates device object information and registers the functions the CDM will support for the device with the NWPA. Besides being called at CDM load time, *CDM_Inquiry()* can be called again if any of the following events occur: (For details, see the description of *CDM_Inquiry()* in Chapter 7)

- A HAM registers a new device with the NWPA that matches the Host Adapter Type and Device Type reported by the CDM in the *Types* parameter of **CDI_Register_CDM()**.

- (For Filter CDMs) A base CDM updates information about a device to which a filter CDM is also bound.

- A device is no longer valid, and the CDM must remove the device and all of the local structures associated with this device from its list.

- An End of Bus condition occurred during a scan for new devices. This means there are no more public devices on this bus. The CDM may then scan for specific devices not found during the normal scan. The specific devices can become public or private devices depending on the Scan function case used. For details, refer to Chapter 8 HACB Type Zero Functions under Function 1-HAM_Scan_For_Devices

- An End of Bus condition occurred when the bus is being deactivated (i.e. when the HAM associated with the bus is being unloaded). The CDM must remove any private devices on this bus and all of the local structures associated with these devices from its list.

5.2.2.1 Updating Device Object Information

A CDM can update device information, such as the functions it will support for a device, by doing the following:

1. Instantiate a new **UpdateInfoStruct** structure initialized with -1 in each field.
2. Change the appropriate **UpdateInfoStruct** fields to the new values.
3. Call **CDI_Object_Update()** including the structure address and *CDIBindHandle* as arguments.

Note:	A -1 indicates that the field does not get updated, thereby maintaining its previous state from either the last update event or when the CDM originally bound to the device.

5.2.2.2 Function Masking

As previously mentioned, a CDM must register the functions it will support for a given device with the NWPA. This section describes how this is done.

Applications assume a certain functionality set, and the NWPA encompasses these by providing a general set of I/O and control messages that it can issue for devices. The CDM is expected to implement routines that translate these messages into actual commands recognized by a device. The way in which the NWPA recognizes the I/O and control routines that a CDM will support for a device are through the respective values set in the functionmask and controlmask fields of the device object's **UpdateInfoStruct**. Each field is 32 bits wide, and each bit position within a field corresponds to an NWPA function. When bits within a field are set, it indicates to the NWPA that the CDM supports that function. Inversely, a CDM can remove supported I/O or control functions by clearing the appropriate bits within these fields and updating the device object information. Refer to Chapter 6 for more information about the **UpdateInfoStruct** and its fields.

5.2.3 Processing CDM Messages

When the NWPA receives an application I/O request, it converts the request into a **CDMMessageStruct** (CDM message) data structure. The fields in this structure contain all the necessary information that a CDM needs to build a device command. The NWPA routes a CDM message to the CDM that has

1. Bound to the device the NWPA wants to access.
2. Registered the desired support function that the NWPA wants to call.

Once a CDM receives a CDM message, it generally performs one of the following actions:

- Creates a SuperHACB request and executes it.
- Places the CDM message in a process queue.
- Chains the CDM message down to another CDM. This action only applies to filter CDMs.

The following subsections address each of these actions.

5.2.3.1 Creating and Executing a SuperHACB Request

Since most application requests to the NWPA are either control or I/O requests, creating and executing SuperHACB requests is the most frequent action. The following is a list of the phases involved:

- Receiving the request and mapping to a CDM function
- Building and Executing a SuperHACB request

5.2.3.1.1 Receiving a Request and Function Mapping As described in Chapter 7, "CDM Message", a NWPA control or I/O request is first packaged into a **CDMMessageStruct** and then passed to a CDM. *CDM_Execute_CDMMessage()* is the CDM's entry point for receiving a **CDMMessageStruct**. The main purpose of *CDM_Execute_CDMMessage()* is to map a general NWPA control or I/O request (in the form of a **CDMMessage**) into a specific CDM function call. The NWPA calls this function passing it *CDMBindHandle* and a pointer to a **CDMMessage** structure. *CDMBindHandle* identifies the desired device and its specific information. The **CDMMessage** structure contains the information describing the request.

NWPA control and I/O requests are equated to unique hexadecimal function codes (0x0000 - 0x0047). When the NWPA makes a request, it places the appropriate function code in the Function field of the **CDMMessageStruct**. The CDM uses this code to determine what function it is to perform. A list of NWPA request codes can be found in Chapter 9, "CDM Message Types".

5.2.3.1.2 Building and Executing a SuperHACB Request Once a NWPA request is mapped to a CDM function, the CDM function has the responsibility to build a SuperHACB and execute it. The control and I/O routines discussed in section 5.1.2.3 are the functions that do the building and executing. Each function accepts a *CDMBindHandle* and a pointer to a **CDMMessageStruct** as arguments.

The Control and I/O routines are responsible to perform the following:

1. Allocate a SHACBStruct using **CDI_Allocate_HACB**(). For optimal performance, a CDM should maintain a re-usable pool of these structures. The control and I/O routines can then recycle the SuperHACBs from this pool in executing and completing requests. Doing this saves the overhead of memory allocation/deallocation. Conditionalize the control and I/O routines to call **CDI_Allocate_HACB**() only if the SuperHACB pool is depleted during the context of that particular control or I/O routine.

2. Fill in the SuperHACB fields with the appropriate information relative

to the request. Refer to Chapter 3, "Host Adapter Control Block" for details on what is expected to be placed in the SuperHACB structure.

3. Call **CDI_Execute_HACB()** passing the appropriate arguments. One argument that is necessary is a pointer to *CDM_Callback()*. By passing this address, the NWPA ensures that the CDM is notified of the request completion by calling *CDM_Callback()*.

5.2.4 Error Handling

This section describes the CDM's general error handling paradigm and how this paradigm is affected by *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 adapter supports the feature and whether or not it is to be used. The CDM, on the other hand, must be ready to support either case. 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;
```

The CDM finds out if a device is set up for auto error sense by checking the **Auto_Error_Sense_Flag** (`0x00000040`) in the **AttributeFlags** field of the device's **DeviceInfoStruct**. If the flag is set, the device does auto error sense. If the flag is not set, the device does not do auto error sense. The CDM receives a pointer to this structure as an input parameter to its *CDM_Inquiry()* routine. Thus, at bind-time the CDM knows the error sense mode of the device.

The following subsections describe the CDM's error handling paradigm, first without auto error sense, and then with auto error sense, respectively.

5.2.4.1 Without Auto Error Sense

Error handling without auto error sense is probably the simplest case for CDMs. The paradigm is as follows:

- When a device error occurs, the HAM freezes that device's queue, posts the appropriate completion code [1] to the HACB and then completes the HACB.
- The CDM receives the HACB through its callback routine, checks the completion code and realizes that the I/O request generated a device error.
- The CDM then spawns a blocking, error-recovery thread that issues another HACB request for error sense information. The CDM can request as much error sense information as the device and transport protocol can provide. The error sense data is retrieved under the normal HACB I/O channel, meaning that the CDM and HAM use the HACB's data buffer.
- The CDM keys off the error sense information to determine what it will attempt to do to recover from the error condition.
- When the error condition is remedied, the CDM completes the I/O request's corresponding CDM message with a successful completion code and allows normal I/O to the device to continue. If the CDM determines that the error condition cannot be remedied, it completes the corresponding CDM message with an appropriate error code.

> **Important:** For the no-auto-error-sense case, the CDM should zero out the **errorSenseBufferLength** and **vErrorSenseBufferPtr** fields of the HACB.

5.2.4.2 With Auto Error Sense

> **Note:** The information presented in this subsection is tightly coupled to the reference information for the **ErrorSenseInfoStruct** found in Chapter 6.

For the auto error sense case, the CDM is expected to do these additional steps in building the HACB I/O request:

---

[1] For SCSI, a device error generates a check condition, and the appropriate completion code is 0x80010002. For IDE\ATA, a device error sets the error bit of the IDE/ATA Status register, and the appropriate completion code is 0x80010001. Refer to Appendix B for more information.

- Allocate an I/O contiguous, auto error sense buffer and assign its address to the HACB's **vErrorSenseBufferPtr** field.
  This buffer is where the return sense information is to be placed, and its structure is defined by the NWPA's **ErrorSenseInfoStruct**. The size of this buffer is the size of the NWPA's **ErrorSenseInfoStruct** plus however many BYTES of auto error sense data the CDM wants returned. For more details refer to the **ErrorSenseInfoStruct** reference information in Chapter 6.

> **Note:** Since the CDM may need a number of these error sense buffers, a suggestion would be to allocate a pool and reuse them as needed. Also, the CDM should not be concerned with assigning anything to the HACB's **pErrorSenseBufferPtr** field. The NWPA will calculate the sense buffer's physical address and place it in this field at execute-time.

- Place a value in the auto error sense buffer's **numberBytesRequested** field to indicate the desired number of sense BYTES it would like the device to return.
- Place a value in the HACB's **errorSenseBufferLength** field indicating the byte-size of the HACB's auto error sense buffer.

At callback time (*CDM_Callback()*), if the CDM detects a device error on the HACB request, the CDM attempts error recovery as prescribed in the previous subsection. However, instead of issuing another HACB to request error sense, it looks in the HACB's sense buffer to get the error information. However, prior to reading the error information, the CDM should check the value in the **numberBytesReturned** field of the buffer. This field indicates the actual number of sense BYTES the device provided.

5.2.5 Registering Device Attributes

Attributes are parameters associated with a device such as unitsize, blocksize, capacity, preferred unitsize, tape read/write formats, etc. Device attributes should be registered during the context of *CDM_Inquiry()* after the CDM has had a chance to query the device and bind to it. The CDM informs an application of a device's attributes by registering them with the NWPA. Also, if the device supports it, a CDM can allow an application to set attributes. The following steps outline the procedure for registering device attributes.

1. The CDM fills out an instance of an **AttributeInfoStruct** for each attribute it intends to register.

2. The CDM then calls **CDI_Register_Object_Attribute**() for each

attribute in step 1. **CDI_Register_Object_Attribute**() accepts a pointer to the **AttributeInfoStruct** instance associated with the attribute and pointers to the attribute's *CDM_Get_Attribute()* and *CDM_Set_Attribute()* routines as input parameters.

> **Note:** If the attribute is not settable, then the input parameter corresponding to *CDM_Set_Attribute()* should be set to zero.

The NWPA informs an application of the data type associated with the *infoBuffer* parameter of these routines by the value the CDM specified in the **attributeType** field of the attribute's **AttributeInfoStruct**.

3. Through a set of NWPA APIs, the application can then get and set attributes. The following code example shows how to register a minimum blocksize attribute:

```
/*- Prototypes of get and set routines for this attribute -*/
LONG CDM_Get_Attribute(
   LONG cdmBindHandle,
   LONG *infoBuffer,
   LONG infoBufferLength,
   LONG attributeID);

LONG CDM_Set_Attribute(
   LONG cdmBindHandle,
   LONG *infoBuffer,
   LONG infoBufferLength
   LONG attributeID);

/*- Data type, AttributeID, Buffer Length definitions -*/
#define MM_BYTE 0x02
#define MM_LONG 0x04
#define MIN_BLOCKSIZE 0x4E494D12
#define MM_BUFFERLEN 0x04

/*- Attribute Information -*/
struct AttributeInfoStruct MinimumBlocksize = {
   MIN_BLOCKSIZE,
   MM_LONG,
   MM_BUFFERLEN,
   "\x17MINIMUM_BLOCKSIZE"};

/*- Register Attribute -*/
cCode = CDI_Register_Object_Attribute(
   NWPAHandle,
   CDMBindHandle,
   &MinimumBlocksize,
   CDM_Get_Attribute,
   CDM_Set_Attribute);
```

5.2.6 Unload-time Deregistration

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

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

   A. Determines if any applications are using any of the devices managed by the CDM.

      CDM_Unload_Check() calls **NPA_Unload_Module_Check()**, which checks the NWPA's database and returns the status of each device attached to the adapter. *CDM_Unload_Check()* returns the use-status returned by **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 CDM 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 CDM's devices are in use. A return value greater than zero indicates that one or more of the CDM's devices are in use.

2. If *CDM_Unload_Check()* returns zero, the OS calls the CDMs *CDM_Unload()* entry point. If *CDM_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 CDM's *CDM_Unload()* entry point. The OS calls the CDM's *CDM_Unload()* entry point with blocking context, and it does the following:
   A. Causes the NWPA to terminate I/O to the CDM. *CDM_Unload()* terminates I/O to the CDM by calling **CDI_Unregister_CDM()** 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 CDM must remain operational and process requests until **CDI_Unregister_CDM()** returns control. Once **CDI_Unregister_CDM()** returns control to *CDM_Unload(),* the CDM is guaranteed not to receive any more I/O requests for that device.

B. Returns resources back to the system.

1. Abort all outstanding AEN HACBs using **CDI_Abort_HACB()**.

2. If the CDM is controlling any private devices, they must either be made public using Case 2 of the Scan function, or be removed using Case 3 of the Scan function. For more details, refer to **Chapter 8 HACB Type 0 Functions** - *HAM_Scan_For_Devices*. (Function 1)

3. Cancel all asynchronous events, such as timeout handlers, timers, etc., by calling **NPA_Cancel_Thread()**.

4. Return memory to the system pool by calling **NPA_Return_Memory()**.

5. Unregister the module using **NPA_Unregister_Module()**.

C. Return 0

5.3 Special Topics

This section discusses special topics related to the CDM.

5.3.1 Device Queue Behavior

Refer to Section 4.3.1.3 for details on the queue behavior a CDM can expect for a device.

5.3.2 Scanning Specific Target IDs and LUNs

Note: The procedure discussed in this subsection only applies to SCSI.

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 [2].

During the "scan for new devices" thread, the NWPA iteratively calls the CDM's *CDM_Inquiry()* routine for each device found on the SCSI bus that matches the device type the CDM registered for. As mentioned in section 5.2.2, *CDM_Inquiry()* is the entry point where the CDM gets a chance to look at a device and determine if it will field I/O requests for the device by binding to it. However, given the OS's initial scan paradigm, the CDM will only see devices attached to LUN 0 of any given 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.

These scan messages are in the form of **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 (particularly the fields of the host adapter command block) according to the scan case (or action) it wants the HAM to perform.

The NWPA defines three scan cases. These cases are referred to numerically as either Case 1, Case 2, or Case 3 corresponding to the value the CDM sets in the *parameter2* field of the host command block.

---

[2] See footnote 5 in Chapter 4 for the explanation of this limitation.

The remaining subsections provide more details, and they are presented as follows:

5.3.2.1 Public vs. Private Devices

5.3.2.2 Scan Case Parameters and Descriptions

5.3.2.3 Scan Completion Codes

5.3.2.1 Public vs. Private Devices

A public device is one that is visible to any CDM that registers with a matching device type. The NWPA makes a public device visible to these CDMs by calling their respective *CDM_Inquiry()* entry points, thus giving any one of them the opportunity to bind to the device. A public device has a corresponding object in the NWPA's device database, and the **Private_Public_Flag** in the **attributeFlags** field of the device's **DeviceInfoStruct** is cleared. Because a corresponding object exists in the NWPA's device database, a public device is also visible to applications. Any application can reserve a public device and issue control and I/O messages to it.

> **Note:** All LUN 0 devices detected in the initial "scan for new devices" command are public.

A private device is one that is visible only to the CDM that detected it through a specific target ID and LUN scan. This CDM has exclusive ownership of the device.

A private device does not have a corresponding object in the NWPA's device database, and the **Private_Public_Flag** in the **attributeFlags** field of the device's **DeviceInfoStruct** is set. Because a corresponding object does not exist in the NWPA's device database, a private device is invisible to applications.

The purpose of the private-device-feature is to allow a CDM to present a group of devices that enhance each other's functionality as a single device to the NWPA. This prevents a competing CDM from stealing one of the devices from the group. A scenario where this feature might be useful is a magazine device, addressed at LUN 1, attached to a public tape drive at LUN 0. The CDM can present these devices as one device with both tape and magazine functionality.

5.3.2.2 Scan Case Parameters and Descriptions

This subsection describes the three different scan cases (Case 1, Case 2, and Case 3) that a CDM can issue to the HAM. Included are specifications of the HACB input parameters associated with each scan case and also their respective outputs. The CDM provides the input values prior to issuing the scan request. After the request completes, the CDM reads the applicable outputs to interpret results. The value in the HACB's **hacbCompletion** field is the key for determining if any additional outputs are valid.

A table listing possible scan completion codes is provided in the next subsection.

Case 1: Probe Specified Target ID and LUN, Return Info and Make Detected Device Private

INPUTS to HACB:

Field Values in Host Command Block:

Function = 1
Parameter0 = LUN
Parameter1 = Target ID [3]
Parameter2 = 1

Field Values in Main HACB:

DeviceHandle = -1 indicating a request to probe a specific target ID and LUN to detect a device that is new to the CDM.
OR
= DeviceHandle of a device already owned by the CDM. The CDM received this handle either from a previous Case 1 scan request (if the device is private) or at bind-time (if the device is public).

VirtualAddress = Address of I/O contiguous memory location where the device's **DeviceInfoStruct** information is to be returned. This buffer should be allocated using **NPA_Allocate_Memory()**

OUTPUTS from HACB:

hacbCompletion = Appropriate scan completion code.

---

[3] The Target ID is equivalent to the value in the *BusID* field of the **DeviceInfoStruc**t associated with the current device. The current device is the one on which the current iteration of *CDM_Inquiry()* is being called, and the *DeviceInfo* parameter points to the device's corresponding **DeviceInfoStruct**.

If hacbCompletion == Successful_Completion:

Control_Info	=	Sizeof(struct DeviceInfoStruct)
VirtualAddress	=	Pointer to the buffer where the HAM will copy the device's DeviceInfoStruct information. This structure contains the DeviceHandle that gives the CDM access to the device.

A Case 1 scan allows the CDM to do the following:

If (DeviceHandle = -1):

- Detect a new device at the specified target ID and LUN, and if one exists, make it private.

If (DeviceHandle = Existing handle to a device owned by the CDM):

- Verify that the device still exists at the specified target ID and LUN and, if the device's current status is public, change it to private.

DeviceHandle == -1

The CDM issues this instance of a Case 1 scan during its *CDM_Inquiry()* routine when it knows that the LUN 0 device on which the inquiry is being called supports additional devices at other LUNs. In addition, a Case 1 scan indicates that the CDM wants to control these additional devices privately and present the group as a single device with extended functionality to the NWPA. Since *CDM_Inquiry()* is called on a blocking thread, the CDM can issue the request using **CDI_Blocking_Execute_HACB**().

If a device responds at the specified target ID and LUN, it is declared private and information about the device is returned in the HACB's data buffer. The structure of the return information is defined by the NWPA's **DeviceInfoStruct**. This structure includes a **DeviceHandle** that allows access to the device, and the bus-specific inquiry information about the device [4].

The CDM uses this return information to determine if the device is real or a phantom, and, if the device is real, to decide whether or not it wants to field I/O requests for the device. As a part to making this decision, the CDM can use the **DeviceHandle** to issue non-destructive HACB requests to get additional information about the device. A non-destructive request is one that does not alter the current state of the device, such as a SCSI MODE SENSE command.

If the device is real and the CDM wants to field requests for the device, it remembers the device's **DeviceHandle**. This is a private **DeviceHandle** giving the CDM exclusive access to the device. No other CDM can have

[4] The inquiry information in specific to the bus interface. For SCSI, this information is identical to that returned by the standard INQUIRY command. For IDE\ATA, this information is identical to that returned by the IDENTIFY command.

access to this device until the owner CDM relinquishes control by either issuing a Case 3 scan or by declaring the device public.

> **Note:** In order for a CDM to really have access to a private device, it must first be bound to a public, companion device on that same target ID. Otherwise, the NWPA will not route I/O to the private device.
>
> The CDM should bind to the public device on which its *CDM_Inquiry()* routine was called, which is also the thread in which the CDM scanned and detected the private device.

If the CDM determines it does not want to field requests for the device or that the device is a phantom, it should issue a Case 3 scan to remove the device object from the HAM's device list.

DeviceHandle == Existing Handle

The CDM can issue this instance of a Case 1 scan whenever it deems appropriate, as long as the CDM owns the target device through either a previous Case 1 scan (private case) or through a previous bind to the device (public case). A valid **DeviceHandle** to the target device is what constitutes ownership. If the target device responds at the specified target ID and LUN, the same information generated from the scan request that first detected the device is returned in the HACB's data buffer. Additionally, if the device's status was originally public, it is changed to private.

Since the scan invokes an actual probe of the bus, the CDM should spawn a blocking thread (**NPA_Spawn_Thread()**) to execute this instance of a Case 1 scan.

> **Note:** Whenever, the CDM issues a scan request that changes a device's status (public to private or private to public), it must follow up the scan with a call to **CDI_Rescan_Bus()**. This call updates the NWPA's device database.

Case 2: Probe Specified Target ID and LUN, Return Info, and Make Detected Device Public

INPUTS to HACB:

Field Values in Host Command Block:

Function = 1
Parameter0 = LUN
Parameter1 = Target ID
Parameter2 = 2

Field Values in Main HACB:

DeviceHandle = -1 indicating a request to probe a specific target ID and LUN to detect a device that is new to the CDM.
OR
= **DeviceHandle** of a device already owned by the CDM. The CDM received this handle either from a previous Case 1 scan request (if the device is private) or at bind-time (if the device is public).

VirtualAddress = Address of I/O contiguous memory location where the device's **DeviceInfoStruct** information is to be returned. This buffer should be allocated using **NPA_Allocate_Memory()**.

OUTPUTS from HACB:

hacbCompletion = Appropriate scan completion code.

If Successful_Completion:

Control_Info = Sizeof(struct **DeviceInfoStruct**)
VirtualAddress = Pointer to the buffer where the HAM will copy the device's **DeviceInfoStruct** information. This structure contains the **DeviceHandle** that gives the CDM access to the device.

A Case 2 scan allows the CDM to do the following:

If (**DeviceHandle** == -1):

- Detect a new device at the specified target ID and LUN, and if one exists, make it public.

If (**DeviceHandle** == Existing handle to a device owned by the CDM):

- Verify that the device still exists at the specified target ID and LUN and, if the device's current status is private, change it to public.

DeviceHandle == -1

The CDM issues this instance of a Case 2 scan during its *CDM_Inquiry()* routine when it knows that the LUN 0 device on which the inquiry is being called supports additional devices at other LUNs. In addition, a Case 2 scan indicates that the CDM wants to present these additional devices, singly, as public objects so that they can be controlled by an application. Since *CDM_Inquiry()* is called on a blocking thread, the CDM can issue the request using **CDI_Blocking_Execute_HACB()**.

If a device responds at the specified target ID and LUN, it is declared private and information about the device is returned in the HACB's data buffer. The structure of the return information is defined by the NWPA's **DeviceInfoStruct**. This structure includes a **DeviceHandle** that allows access to the device, and the bus-specific inquiry information about the device [5].

The CDM uses this return information to determine if the device is real or a phantom, and, if the device is real, to decide whether or not it wants to field I/O requests for the device. As a part to making this decision, the CDM can use the **DeviceHandle** to issue non-destructive HACB requests to get additional information about the device. A non-destructive request is one that does not alter the current state of the device, such as a SCSI MODE SENSE command.

If the device is real and the CDM wants to field requests for the device, it will conclude the current iteration of its *CDM_Inquiry()* routine with a call to **CDI_Rescan_Bus()**. This call causes the NWPA to create an object for the device and place the object in its device database, which is critical to making the device public.

Unlike the private device paradigm (Case 1 scan), the CDM must not take control of a public device during the same *CDM_Inquiry()* thread in which it detected (via a Case 2 scan) the device. Instead, the CDM must wait until its *CDM_Inquiry()* routine gets called again, for that device, and officially bind to it using **CDI_Bind_CDM_To_Object()**. At that time, the CDM sets up the I/O channel by remembering the device's **DeviceHandle**. Adherence to this public-device-paradigm is essential, and it is a requirement for CDM certification.

Note:	For the CDM's *CDM_Inquiry()* routine to be called on the new public device, the CDM must have registered for that device's device type.

[5] See previous footnote.

If the CDM determines it does not want to field requests for the device or that the device is a phantom, it should issue a Case 3 scan to remove the device object from the HAM's device list.

DeviceHandle == Existing Handle

The CDM can issue this instance of a Case 2 scan whenever it deems appropriate, as long as the CDM owns the target device through either a previous Case 1 scan (private case) or through a previous bind to the device (public case). A valid **DeviceHandle** to the target device is what constitutes ownership. If the target device responds at the specified target ID and LUN, the same information generated from the scan request that first detected the device is returned in the HACB's data buffer. Additionally, if the device's status was originally private, it is changed to public. Since the scan invokes an actual probe of the bus, the CDM should spawn a blocking thread (**NPA_Spawn_Thread()**) to execute this instance of a Case 2 scan.

> **Note:** Whenever, the CDM issues a scan request that changes a device's status (public to private or private to public), it must follow up the scan with a call to **CDI_Rescan_Bus()**. This call updates the NWPA's device database.

Case 3: Remove Device Object from Device List

INPUTS from HACB:

Field Values in Host Command Block:

Function	=	1
Parameter0	=	LUN
Parameter1	=	Target ID
Parameter2	=	3

Field Values in Main HACB:

DeviceHandle	=	Valid **DeviceHandle** of the target device. This DeviceHandle proves that the invoking CDM has ownership of the device; therefore, the CDM has the right to discard the device's object.

OUTPUTS to HACB:

hacbCompletion	=	Appropriate scan completion code.

A Case 3 scan request allows the CDM to remove the target device from the HAM's device list, and it causes the HAM to free any objects associated with the device. The purpose of a Case 3 scan is twofold: It allows the CDM to delete phantom devices from the HAM's device list; and, at unload-time, it allows a CDM to relinquish private devices under the CDM's control.

5.3.2.3 Scan Completion Codes

Table 5-1 summarizes the scan completion codes described in the specific cases above. The table also includes additional error-completion codes common to all scan cases. These completion codes get posted to the HACB's **hacbCompletion** field:

Table 5-1 Scan Completion Codes

Upper WORD (16 bits)	Lower WORD (16 bits)	Description
0x0000	0x0000	Successful Completion: The current scan operation completed successfully.This completion code applies to all scan cases.For Case 1 and Case 2 scans, this completion code indicates that a device responded at the specified Target ID and LUN, and the information returned in the HACB's data buffer is valid.
0x000A	0x0000	General Failure: Default scan-error category. The cause of the error is unknown, and any information contained in the HACB's data buffer is invalid.This completion code applies to all scan cases.
	0x0001 .	Device Not Found: No device responded at the specified Target ID and LUN. Anyinformation contained in the HACB's data buffer is invalid.This completion code applies to Case 1 and Case 2 scans
	0x0002	Bad Target ID/LUN: The Target ID and/or LUN specified in the HACB's host adaptercommand block was/were invalid. Any information contained in the HACB's data bufferis invalid.This completion code applies to all scan cases.
	0x0003	Target In Use: The target object is owned by another CDM. Therefore, the current scanrequest could not be executed.This completion code applies to Case 1, Case 2, and Case 3 scans.
	0x0004	Object Not Found: A CDM issued a Case 3 scan to remove a device object from theHAM's device list that does not exist. The object does not exist because no previous Case1 or Case 2 scan was issued on the specified Target ID and LUN to create it. Anyinformation contained in the HACB's data buffer is invalid.This completion code applies to Case 3 scans.
Novell reserves the right to add additional completion codes.		

5.3.3 Removable Media Support

The NWPA provides a specific set of control functions to support removable-media devices. The NWPA packages these control functions in CDM Messages. Table 5-2 contains a list of these functions including their respective NWPA function numbers, **ControlMask** (**UpdateInfoStruct**) enabling bits, and support requirements.

Table 5-2: NWPA's Removable Media Control Functions

Description	Function Number	ControlMask Bit	Support
Activate / Deactivate	0x00000003	0x00000008	Mandatory
Mount / Dismount	0x00000004	0x00000010	Mandatory
Lock / Unlock	0x00000007	0x00000080	Optional
Insert / Remove	0x0000001B	0x08000000	Optional

Detailed descriptions of these message functions can be found in Chapter 9, and the NWPA expects all CDMs managing removable-media devices to support the functions marked Mandatory. The remaining subsections describe the use of these control functions.

5.3.3.1 Mount, Lock, and Activate Messages

The key difference between CDMs supporting fixed-media devices and CDMs supporting removable-media devices is that the *CDM_Inquiry()* routine of a removable-media CDM may get called at a time when there is no media in the device. As mentioned in section 5.2.2 "Inquiring and Binding to a Device," *CDM_Inquiry()* is the entry point where the CDM queries the device and decides whether or not it will "bind" (**CDI_Bind_CDM_To_Object**()) to the device. Part of the bind process requires the CDM to fill out an instance of an **UpdateInfoStruct** for the device and its media. Since a piece of media may not be loaded at bind-time, it is impossible for the CDM to know all of the information necessary to fill out the **UpdateInfoStruct** and make the device active.

The purpose of the Mount function is to give a CDM a second opportunity (other than at bind-time) to get the additional information it needs to complete the **UpdateInfoStruct** after a piece of media gets loaded into the device. The Activate function follows the Mount, and its purpose is to allow the CDM to handshake with the NWPA to indicate that the device is active and ready to receive I/O. The following is an outline of the binding and activating paradigm for removable-media devices

1. The CDM's *CDM_Inquiry()* routine (the bind-time entry point) is called and no media is loaded in the device.

2. The CDM decides it wants to bind to the removable-media device.
   A. The CDM fills out an instance of an **UpdateInfoStruct** for the device providing as much information as it can determine at the time. The **removableFlag** field should be set to 0x0001, the **activateFlag** field should be set to 0x0000, and the **controlMask** field should have the appropriate bits set indicating that the CDM supports the removable-media functions introduced in Table 5-1. At the very least, the bits corresponding to the mandatory functions should be set, and the CDM must provide a routine (or routines) that implement these control functions.

   B. The CDM completes the bind by calling **CDI_Bind_CDM_To_Object**() passing it a pointer to the device's **UpdateInfoStruct** as an input parameter. Without media in the device, the device remains inactive until a user physically loads media into it and indicates the load to the system by issuing a console command.

3. At some later time when a piece of media is inserted into the device, the NWPA informs the CDM by issuing a **Mount** message. During the context of the **Mount**, the CDM issues HACB requests to confirm the existence of the media, get the additional device/media information it needs to complete the device's **UpdateInfoStruct**, and prepare the media for I/O. The CDM concludes the Mount by reporting the updated information to the NWPA by calling **CDI_Object_Update**().

4. If the appropriate bit is set in the **controlMask** field of the device's **UpdateInfoStruct**, the CDM may (see the following note) receive a **Lock** message following the **Mount**. If the CDM receives this message, it should lock the media in the device.

> **Note:** The **Lock** function only applies to stand-alone, removable-media devices, not to changer or magazine type devices. The **Lock** function is issued at the discretion of the controlling application.

5. To confirm that the device is ready to receive I/O, the NWPA issues an **Activate** message. During the context of the **Activate** function, the CDM should make sure that the media is capable of being activated, set the **activateFlag** field of the device's **UpdateInfoStruct** to 0x0001, and report the object update to the NWPA by calling **CDI_Object_Update**().

Upon completion of the **Activate**, the CDM must be ready to accept I/O requests for the device.

5.3.3.2 Dismount and Deactivate Messages

The CDM may receive a media **Dismount** or device **Deactivate** message at any time. When the CDM receives either one of these messages, it should set the **activateFlag** of the device's **UpdateInfoStruct** to 0x0000 and update the object by calling **CDI_Object_Update**(). In the case of a **Dismount**, the CDM should reset (set to -1) the other fields of the device's **UpdateInfoStruct** prior to updating the object

5.3.3.3 Insert and Remove Messages

If the CDM indicates that it supports **Insert** and **Remove** functions for a device, the CDM should issue an insert media command, if an **Insert** message is received, or an eject media command, if a **Remove** message is received, to the device.

5.3.4 Magazine Support

The NWPA also supports magazine devices, and it allows upper layers to control the magazine and the media associated with the device. Before listing the NWPA specific control functions that provide magazine support, it is essential to discuss the NWPA's concept of a magazine device:

- A magazine is considered to be a static set of media and slots that can be associated with a single device. That is, only one magazine can be associated with one device at one time . A device that goes beyond this concept fits the autochanger paradigm discussed in the next section.

- Magazine devices also fall into the removable-media device category; therefore, CDMs managing magazines must support the removable-media control functions listed in Table 5-1 as well as the control functions discussed in this section (listed in Table 5-3).

Table 5-3 lists the additional NWPA functions required for magazine support. Since these control functions only apply to magazines, the NWPA groups them into one CDM Message category called **Magazine Functions**. This is the reason why a CDM can indicate support of these functions by setting a single bit in the device's **ControlMask**. The NWPA assigns 0x0000001D as the NWPA group-function number for the message category, and an individual member in the group is referenced by an NWPA sub-function number. Refer to Chapter 9 for more details on NWPA

numbering of this group and for detailed descriptions of each function in the group.

Table 5-3: Additional NWPA Control Functions for Magazine Support
ControlMask: 0x20000000

NWPA Group Function Number: 0x0000001D

Description	Sub-Function Number	Support
Return Magazine Info	0x00000000	Mandatory
Return Magazine Media Mapping	0x00000002	Mandatory
Magazine Select Media	0x00000003	Mandatory
Magazine Deselect Media	0x00000004	Mandatory
Magazine Load	0x00000005	Mandatory
Magazine Unload	0x00000006	Mandatory
Magazine Eject	0x00000007	Mandatory

A CDM supporting a magazine binds to a device using the removable device paradigm described in the previous section. However, before the CDM calls **CDI_Bind_CDM_To_Object()**, it should set the **acceptsMagazinesFlag** field of the device's **UpdateInfoStruct** to 0x0001, and also the **magazineLoadedFlag** field if a magazine is currently loaded. Again, all this is done at bind-time during the context of the CDM's *CDM_Inquiry()* routine.

If at bind-time the CDM indicated that a magazine was not loaded in the device, the magazine device will remain inactive until one gets loaded. Then, at some later time when a magazine is placed in the device, the NWPA informs the CDM by issuing a **Magazine Load** message. This message function directs the CDM to verify the existence of a magazine and prepare it for use.

After the CDM confirms that the magazine is loaded, or if at bind-time the CDM indicated that a magazine was already loaded in the device, the NWPA issues a **Return Magazine Info** message to the CDM. This message function directs the CDM to return the number of storage slots in the currently loaded magazine.

After the CDM completes the **Return Magazine Info** message, the NWPA issues a **Return Magazine Media Mapping** message. This message function directs the CDM to inventory the magazine and the device to

which the magazine is attached. The CDM then returns a byte-map indicating whether or not media is loaded in the device and which slots in the magazine have media. As an example, suppose the following:

- The magazine is attached to a tape drive.
- The number of storage slots returned by the **Return Magazine Info** function equaled 4.
- Media is not currently loaded in the tape drive.
- Magazine storage slots "1" and "2" contain media.
- Magazine storage slots "3" and "4" are empty.

The return byte-map would contain the following information:

Slot / Byte Index	Byte-Map Value	Comment
[0]	0x00	Indicates that the tape drive, slot[0], is empty.
[1]	0x01	Indicates that magazine storage slot[1] has media.
[2]	0x01	Indicates that magazine storage slot[2] has media.
[3]	0x00	Indicates that magazine storage slot[3] is empty.
[4]	0x00	Indicates that magazine storage slot[4] is empty.

The slot indexes shown above have particular significance. The NWPA will use these indexes as slot indicators in subsequent magazine control functions. Index 0 will always indicate the device, and indexes 1 to n indicate the respective storage slots of the magazine, where n is the total number of storage slots.

Successful completion of the **Return Magazine Media Mapping** message concludes magazine initialization. At this point, an application can select a piece of media and prepare it to receive I/O. The application selects a piece of media by issuing a **Magazine Select Media** message. When the CDM receives this message, it should move the piece of media indicated by the storage slot index number (specified in one of the message's input

parameters) into the device. After the media is in the device, the **Mount** and **Activate** sequence described in the removable-media section begins. After the device is activated, the CDM will receive I/O requests for the device.

After an application is done with the media, it can remove the media from the device by issuing a **Magazine Deselect Media** message. When the CDM receives this message, it should remove the media currently in the device and return it to the storage slot indicated by the slot index number given in the message.

When the CDM receives a **Magazine Unload** message, it should unload the magazine, clear the **magazineLoadedFlag** field of the device's **UpdateInfoStruct**, and update the device object by calling **CDI_Object_Update()**.

A CDM supporting devices with magazines must support the **Magazine Eject** function. However, for some magazine devices, ejects happen as part of the deselect process. In this case, when the CDM receives this message, it should just successfully complete the message without performing any action. For those magazine devices that require an explicit eject, the CDM should issue a magazine eject to the device.

5.3.5 Changer Support

AutoChanger support has also been added to NetWare 4.X platforms. The NWPA creates an object for the autochanger and has device, media, and slot objects associated with the autochanger. The concept of a autochanger to the NWPA is a non-static set of media or magazines (media can be changed via the mailbox) associated with one or more devices and also a mailslot where new media can be added or taken out of the changer.

The NWPA support of an autochanger is very similar to the earlier NetWare 4.0 support provided through the DDFS. The CDM messages that are specific to an autochanger are:

1. ReturnChangerMediaMapping
2. ChangerCommand

Along with the above CDM messages, magazine CDM messages may also be supported. Novell has found it useful to treat changers as changers full of magazines since most changers contain double sided media. This paradigm requires the changer to emulate magazine behavior. This can be done by registering an Enhancer CDM as part of the changer CDM. This Enhancer CDM binds to the devices associated with the changer. It is the responsibility of the CDM to figure out which devices belong to the

changer. SCSI provides for the changer to give Target ID's of the devices associated with it. The Enhancer part of the changer CDM can then intercept magazine control functions bound for the device and have them executed by the changer.

Step by Step

In the CDM inquiry, the CDM will need to bind to the changer and most of the information that needs to be filled out in the **UpdateInfoStruct** will be available. The fields that should be filled in at binding time are:

1. controlmask - The changer functions above.

2. numberofslots - slots in changer.

3. numberofmailboxes - number of exchange slots.(mailboxes)

4. numberofdevices - number of devices in the changer

5. deviceobjects[1] - This field is variable length.

The NWPA object ids for each device in the order they will be addressed in the future i.e. deviceobjects[3] is device #4 in a changer command.

The NWPA object ID's for the devices may not be available at the time of the CDM inquiry, in which case that field should not be changed and the device should not be activated. The device ID's for the changer's devices can be found in a number of ways. The above mentioned registering an enhancer for devices of the right type and checking the SCSI target ID's of the devices against those in the changer is one way. Another might be to use the NWPA APIs to walk the object tree and compare those SCSI target IDs. However it is done, once the object id's for the device are found they should be filled in and updated using **CDI_Object_Update**().

The ReturnChangerMediaMapping CDM message passes a buffer that is a byte map to be filled in by the CDM for the devices, slots and mailboxes of the changer, in that order. This buffer should be filled in with the locations of the media in the devices, slots, and mailboxes. After this, the changer objects will be built in the NWPA and an Activate CDM message will be sent. The changer should be activated by the CDM using **CDI_Object_Update**(). Once the changer has been activated, applications can send changer commands to the CDM.

The CDM will receive Changer Commands from an application for moves of media from a destination to a source. The numbers of the destination and source are the order given in the byte map of the

ReturnChangerMediaMapping buffer. The preload command will be sent to the CDM just before a move from a mailbox is done to allow the CDM to do any needed preparation for the insert of the media.

After the preload the user will be prompted at the console to insert media into the mailbox after the user has acknowledged the insert of media the CDM will receive the move command with the mailbox as the source. The changer eject command will be received by the CDM after a move where the mailbox is the destination this allows for the CDM to do a rotate out on the changer if it is needed

5.3.6 Asynchronous Hardware Event Notification

CDMs can request that HAMs notify them of any hardware events, such as a bus reset, device reset, or a device attention, that may occur. To do this, the CDM must issue a **HACBType=0** HACB request placing the following information in the HACB's union to Host 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 [6]
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 must provide the correct device handle for the device it wants monitored. The device handle is placed in the 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 HACB, and executes the request by calling **CDI_Non_Blocking_Execute_HACB()**.

The HAM receives the HACB and maintains it in a local holding area associated with the target device until an event occurs. These HACBs

[6] 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.

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 CDM receives notification of the event when it gets the HACB through *CDM_Callback()*. *CDM_Callback()* is required to check the **hacbCompletion** field on every HACB it receives. At callback time when the CDM receives a HACB having an AEN completion code, it decodes the bitmap value in the **control_Info** field to determine which event occurred, and takes appropriate action. If desired, the CDM can re-issue the AEN HACB. To minimize a possible "notification-not-available" window, the CDM should re-issue the AEN HACB.

5.3.7 Avoiding Buffer Mismatches

The CDM must have a check similar to the pseudo-code below to accommodate applications that allocate oversized buffers when using tape devices running in fixed-block mode.

```
If (!Variable_Block_Mode)
{
      if ((unitsize * unitcount) <= CDMMsg->BufferLength)
          hacb->DataBufferLength = unitsize*unitcount;
     else
          hacb->DataBufferLength = CDMMsg->BufferLength;
 }
```

5.3.8 Vendor-Pass Through API for CDMs

This API (**NPA_CDM_Passthru**()) provides applications the ability to communicate directly with a device. This provides a vendor with a communications channel to allow for vendor-specific commands/data to be sent to/from the device. For example, vendor-specific device diagnostic information could be sent to an application using this API. CDMs must register for the specific functions that it will process. See the **NPA_CDM_Passthru**() API in Chapter 7 for more details.