Chapter 7 Technical Reference for NWPA Routines

This chapter is a technical reference for the routines that are part of the NWPA. Technical information is supplied for the routines that are provided by the NWPA, and functional descriptions are supplied for the routines that a HAM or CDM is expected to implement.

CDM-Specific

- Custom-Device-Interface routines that are identified in the text by a CDI_ prefix. These routines are part of the NWPA, and they provide CDMs with an interface to the NWPA allowing them to register as CDM modules and build and initiate HACB requests.

- Functional descriptions of the interface routines that a CDM is required to implement. These routines are identified in the text by a *CDM_* prefix. In general, these routines are expected to succeed with a return value of zero. However, three of the routines (*CDM_Abort_CDMMessage()*, *CDM_Unload_Check()*, and *CDM_Execute_CDMMessage()*) give return values based on certain conditions. These conditions and their respective return values are specified.

HAM-Specific

- Host-Adapter-Interface routines that are identified in the text by the **HAI_** prefix. These routines provide HAMs with an interface to the NWPA allowing them to register as HAM modules and report HACB request completions.

- Functional descriptions of the interface routines that a HAM is required to implement. These routines are identified in the text by a *HAM_* prefix. In general, these routines are expected to succeed with a return value of zero. However, three of the routines, *HAM_Abort_HACB()*, *HAM_Unload_Check()*, and *HAM_ISR()*, give return values based on certain conditions. These conditions and their respective return values are specified.

General NWPA

- General NWPA support routines that are identified in the text by the **NPA_** prefix. These routines provide CDMs and HAMs with a stable interface to the NetWare OS.

The technical reference information is listed in alphabetical order according to routine names. The following is a list of the routines referenced in this chapter:

CDI_Abort_HACB

Purpose: Issues an abort request to a device.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG CDI_Abort_HACB (
    LONG reserved,
    LONG hacbPutHandle,
    LONG flag);
```

Parameters:

Inputs:

reserved The CDM should set this parameter to zero.

hacbPutHandle Handle to the HACB request being aborted. The value of this parameter is obtained from the **hacbPutHandle** field of the original SHACB's member HACB.

flag Flag indicating the type of abort the HAM is to perform. Its possible values are as follows:

0x00000000	This value tells the HAM to unconditionally abort the HACB even if it has already been sent to the device.
0x00000001	This value tells the HAM to conditionally abort the HACB if aborting only entails the unlinking of the HACB from the device queue. This is referred to as a clean abort.
0x00000002	This value tells the HAM to check and see if the HACB can be cleanly aborted, but not to perform an abort.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Abort_HACB()** is used by a CDM to abort a HACB sent to a HAM.

CDI_Allocate_HACB

Purpose: Allocates SHACBs that are used to communicate with the HAM.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG CDI_Allocate_HACB(
    LONG cdmosHandle,
    struct SHACBStruct **SHACB);
```

Parameters:

Inputs:

cdmosHandle The CDM's handle for using the **CDI_** APIs. The value of *cdmosHandle* was assigned during **CDI_Register_CDM()**, and it is used in conjunction with the CDM-generated *cdmHandle* to uniquely identify a CDM when it interfaces with the NWPA through the **CDI_** API set.

SHACB Address of a pointer to a memory storage location of type *SHACBStruct*. For a detailed description of the data structure refer to Chapter 6. The following is the structure's ANSI C definition:

```
typedef struct SHACBStruct
{
    LONG cdmSpace[8];
    struct HACBStruct HACB;
} SHACB;
```

Outputs:

SHACB Receives a pointer to the newly allocated **SHACBStruct**.

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Allocate_HACB()** is used by a CDM to allocate a SHACB. It is during the context of this routine that the SHACB's **HACBPutHandle** field is assigned a value by the NWPA. The CDM must not alter the value in this field. A SHACB allocated with **CDI_Allocate_HACB()** is not guaranteed to be below the 16 megabyte boundary. Also, certain fields in the member HACB are pre-initialized by the NWPA at allocation, and their values must be maintained. Therefore, do not clear or zero the HACB. Additionally, to adhere to SFT III (System Fault Tolerance) requirements, only the information in two of the HACB's fields get returned to upper system layers. These are the **Control_Info** and **hacbCompletion** fields described in section 3.3.2. The NWPA guarantees the member HACB's data buffer to be physically contiguous.

CDI_Bind_CDM_To_Object

Purpose: Binds a CDM to a device and registers with the NWPA the I/O and control functions that the CDM will support for the device.

Architecture Type: All

Thread Context: Blocking

Syntax:

```
LONG CDI_Bind_CDM_To_Object (
    LONG cdmosHandle,
    LONG npaDeviceID,
    LONG cdmBindHandle,
    LONG *cdiBindHandle,
    struct UpdateInfoStruct *info,
    LONG infoSize);
```

Parameters:

Inputs:

cdmosHandle The CDM's handle for using the **CDI_** APIs. The value of *cdmosHandle* was assigned during **CDI_Register_CDM()**, and it is used in conjunction with the CDM-generated *cdmHandle* to uniquely identify a CDM when it interfaces with the NWPA through the **CDI_** API set.

npaDeviceID The object ID that the NWPA assigned to the target device in its device database. This value is passed to the CDM through its *CDM_Inquiry()* entry point.

cdmBindHandle A unique handle generated by the CDM to identify the device to which it intends to bind. Following the bind, this handle will be the token the NWPA passes to the CDM when routing I/O messages to a device. From this handle, the CDM must be able to locate the target device's information including the HAM-generated **DeviceHandle** and the NWPA-generated **NPABusID**.

cdiBindHandle Address of a local variable of type LONG.

info A pointer to an **UpdateInfoStruct**. This structure contains the information telling the NWPA what functions the CDM will support for the device. For a detailed description of this structure, refer to Chapter 6. The following is the structure's ANSI C definition:

```
struct UpdateInfoStruct
    {
        BYTE Name[64];
        LONG mediaType;
        LONG cartridgeType;
        LONG unitSize;
        LONG blockSize;
        LONG capacity;
        LONG preferredUnitsize;
        LONG functionMask;
```

```
    LONG controlMask;
    LONG unfunctionMask;
    LONG uncontrolMask;
    LONG mediaSlot;
    BYTE activateFlag;
    BYTE removableFlag;
    BYTE readOnlyFlag;
    BYTE magazineLoadedFlag;
    BYTE acceptsMagazinesFlag;
    BYTE objectInChangerFlag;
    BYTE objectIsLoadableFlag;
    BYTE lockFlag;
    LONG diskGeometry;
    LONG reserved[7];
    union
    {
        struct ChangerInfo
            {
             LONG numberOfSlots;
             LONG numberOfExchangeSlots;
             LONG numberOfDevices;
             LONG deviceObjects[n];
            } ci;
    } u1;
} ;
```

infoSize

The size of the **UpdateInfoStruct** pointed at by *info*.

Outputs:
cdiBindHandle

Receives an NWPA generated handle for the target device to which the CDM is binding. This handle is the NWPA's counterpart to the CDM's *cdmBindHandle*.

Return Value:

0 if successful.
Non-zero if unsuccessful.

Description:

CDI_Bind_CDM_To_Object() is used to bind a device object to a CDM. This routine is used within the context of *CDM_Inquiry()*.

CDI_Blocking_Execute_HACB

Purpose: Initiates the execution of a HACB request by issuing it to a HAM.

Architecture Type: All

Thread Context: Blocking

Syntax:

```
LONG CDI_Blocking_Execute_HACB (
       LONG npaBusID,
       LONG hacbPutHandle);
```

Parameters:

Inputs:

npaBusID The object ID that the NWPA assigned to the target bus in its object database. The CDM received this ID through its *CDM_Inquiry()* entry point during which it bound to the device.

hacbPutHandle Handle to the HACB request being executed. The value of this parameter is obtained from the **HACBPutHandle** field of the original SHACB's member HACB.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Blocking_Execute_HACB()** is used if the CDM must issue multiple HACBs to the HAM to complete a single CDM message request. This routine must be called from a blocking thread. Typically, a CDM will use **CDI_Blocking_Execute_HACB()** within the context of *CDM_Inquiry()*, also a blocking thread, to test a device to see if it should bind to the device. **CDI_Blocking_Execute_HACB()** causes the OS to treat the current thread as if it were the current process. This ensures that a request is carried to completion, and instructions immediately following this call can expect the request data to be present. Consequently, since **CDI_Blocking_Execute_HACB()** runs a HACB request to completion, a callback is not necessary unlike the requirement for its non-blocking counterpart, **CDI_Execute_HACB()**.

CDI_Chain_Message

Purpose: Chains CDM message requests through layers of CDM filters prior to being received by a translator CDM (also referred to as a base CDM) where the message is converted to a SHACB. This routine is only used by filter CDMs.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG CDI_Chain_Message(
    LONG cdiBindHandle,
    LONG msgPutHandle,
    LONG *cdmMessage,
    void (*callback)(),
    LONG parameter);
```

Parameters:

Inputs:

cdiBindHandle The NWPA-generated bind handle that was assigned to the calling CDM when it bound to the target device using **CDI_Bind_CDM_To_Object()**.

msgPutHandle Handle to the CDM Message (**CDMMessageStruct**) being passed downward. The value of this parameter is obtained from the **MsgPutHandle** field of the **CDMMessageStruct**.

cdmMessage Pointer to the chained CDM Message casted to a pointer to LONG.

callback Address of the filter CDM's callback routine. The NWPA calls this routine when the translator (base) CDM completes the CDM message associated with the request. If the filter CDM does not require a callback, then this field should be set to zero.

parameter The input parameter of the filter CDM's callback routine. This routine can be whatever is needed to identify the chained message. If the filter CDM does not require a callback, then this field should be set to zero.

Outputs: None.

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Chain_Message()** is used by filter CDMs to chain CDM messages through each layer in a CDM filter chain until the message is received by a translator (base) CDM. Each filter CDM in the chain has the ability to alter ("massage") CDM message information before passing the message to the next filter. The translator CDM is the last link in the chain, meaning that no more data massaging of the CDM message is performed. Instead, as the last

link in the chain, the translator CDM converts the CDM message into a SHACB request and initiates its execution. **CDI_Chain_Message()** allows the filter CDM to specify a callback routine, so that it can be notified when the request cycle associated with the message has been completed. If there are multiple filter CDMs then their respective callbacks are called in reverse order, thereby, rippling completion-notification upward through the chain.

CDI_Complete_Message

Purpose: Informs the NWPA that a message request has been completed.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG CDI_Complete_Message(
    LONG msgPutHandle,
    LONG npaCompletionCode,
    LONG appReturnCode);
```

Parameters:

Inputs:

msgPutHandle Handle to the CDM message (**CDMMessageStruct**) from which the SHACB being completed was built. The value of this parameter is obtained from the **MsgPutHandle** field of the **CDMMessageStruct**.

npaCompletionCode This is zero for no error or non-zero if it should contain an error code. The NWPA completion codes are listed below:

```
#define ERROR_NO_ERROR_FOUND                        0X00000000
#define ERROR_ABORT_UNCLEAN                         0X00000003
#define ERROR_ABORT_CLEAN                           0x0000000A
#define ERROR_CORRECTED_MEDIA_ERROR                 0x00000010
#define ERROR_MEDIA_ERROR                           0x00000011
#define ERROR_DEVICE_ERROR                          0x00000012
#define ERROR_ADAPTER_ERROR                         0x00000013
#define ERROR_NOT_SUPPORTED_BY_DEVICE               0x00000014
#define ERROR_NOT_SUPPORTED_BY_DRIVER               0x00000015
#define ERROR_PARAMETER_ERROR                       0x00000016
#define ERROR_MEDIA_NOT_PRESENT                     0x00000017
#define ERROR_MEDIA_CHANGED                         0x00000018
#define ERROR_PREVIOUSLY_WRITTEN                    0x00000019
#define ERROR_MEDIA_NOT_FORMATTED                   0x0000001A
#define ERROR_BLANK_MEDIA                           0x0000001B
#define ERROR_END_OF_MEDIA                          0x0000001C
#define ERROR_FILE_MARK_DETECTED                    0x0000001D
#define ERROR_SET_MARK_DETECTED                     0x0000001E
#define ERROR_WRITE_PROTECTED                       0x0000001F
#define ERROR_OK_EARLY_WARNING                      0x00000020
#define ERROR_BEGINNING_OF_MEDIA                    0x00000021
#define ERROR_MEDIA_NOT_FOUND                       0x00000022
#define ERROR_MEDIA_NOT_REMOVED                     0x00000023
#define ERROR_UNKNOWN_COMPLETION                    0x00000024
#define ERROR_IO_ERROR                              0x00000028
#define ERROR_CHANGER_SOURCE_EMPTY                  0x00000029
#define ERROR_CHANGER_DEST_FULL                     0x0000002A
#define ERROR_CHANGER_JAMMED                        0x0000002B
#define ERROR_MAGAZINE_NOT_PRESENT                  0x0000002D
#define ERROR_MAGAZINE_SOURCE_EMPTY                 0x0000002E
#define ERROR_MAGAZINE_DEST_FULL                    0x0000002F
#define ERROR_MAGAZINE_JAMMED                       0x00000030
#define ERROR_ABORT_CAUSED_PRIOR_ERROR              0x00000031
#define ERROR_CHANGER_ERROR                         0x00000032
#define ERROR_MAGAZINE_ERROR                        0x00000033
#define ERROR_BLOCKSIZE_MISMATCH                    0x00000034
#define ERROR_DECOMPRESSION_ALGORITHM_MISMATCH      0x00000035
```

Application return code. This parameter passes specific information directly

appReturnCode from the CDM to a NWPA application.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Complete_Message()** is used by a CDM to notify the NWPA that a specific HACB request has been completed. **CDI_Complete_Message()** is generally called within the context of the CDM's *CDM_Callback()* routine, which is the point where the CDM is notified that a HACB request has been completed. *CDM_Callback()* is responsible for checking the value in the HACB's **hacbCompletion** field to determine the request's completion status. If the field value is zero, it indicates that the request completed without error, and **CDI_Complete_Message()** should be called with *npaCompletionCode* = 0x00000000 (NO ERROR). If the field value is non-zero, it indicates that an error occurred while processing the request. In the error case, *CDM_Callback()* can do one of the following:

Option 1: Map the error into one of the NWPA completion codes applicable to the condition and call **CDI_Complete_Message()** with *NPACompletionCode* equal to this code.

Option 2: Spawn a blocking, error handling thread using **NPA_Spawn_Thread()** and return. The spawned error handling thread can request sense information and try to remedy the error. If the error is remedied and the request can be completed successfully, then **CDI_Complete_Message()** should be called within the context of the error handling routine with *npaCompletionCode* = 0x00. However, if the error cannot be remedied, then the error handling routine should perform the tasks prescribed in option 1. If the error is severe enough, the device may need to be deactivated.

Additionally, **CDI_Complete_Message()** provides the channel for a CDM to ripple specific information up to an application. For example, a tape application may require an I/O request to return the actual number of blocks read/written from/to a device. The CDM provides this information via the *appReturnCode* parameter

CDI_Execute_HACB

Purpose: Initiates the execution of a SHACB request.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG CDI_Execute_HACB (
       LONG msgPutHandle,
       LONG hacbPutHandle,
       LONG (*CDM_Callback)());
```

Parameters:

Inputs:

msgPutHandle Handle to the CDM message (**CDMMessageStruct**) from which the SHACB was built. The value of this parameter is obtained from the **MsgPutHandle** field of the **CDMMessageStruct**.

hacbPutHandle Handle to the HACB request being executed. The value of this parameter is obtained from the **HACBPutHandle** field of the original SHACB's member HACB.

CDM_Callback Address of the CDM routine to be called when the HACB request completes. A callback routine must be specified for each issued request.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Execute_HACB()** is used by a CDM to initiate the execution of a HACB request by routing a HACB to the HAM supporting the target device. Most HACB requests should be executed using this routine.

CDI_Non_Blocking_Execute_HACB

Purpose: Allows the CDM to issue AEN HACBs to the HAM.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG CDI_Non_Blocking_Execute_HACB(
        LONG npaBusID,
        LONG hacbPutHandle,
        LONG (*CDM_Callback)());
```

Parameters:

Inputs:

npaBusID The object ID that the NWPA assigned to the target bus in its object database. The CDM received this ID through its *CDM_Inquiry()* entry point during which it bound to the device.

hacbPutHandle Handle to the HACB request being issued. The value of this parameter is obtained from the **HACBPutHandle** field of the original SHACB's member HACB.

CDM_Callback Address of the CDM routine to be called when the HACB request completes. A callback routine must be specified for each issued request.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Non_Blocking_Execute_HACB()** is used by a CDM to issue Asynchronous Event Notification (AEN) HACBs to the HAM. The CDM indicates which device it wants the AEN to monitor by placing the appropriate handle in the HACB's **DeviceHandle** field. For more information about AEN HACBs, refer to section 4.3.2.

CDI_Object_Update

Purpose: Allows the CDM to update device object information

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG CDI_Object_Update (
       LONG cdmosHandle,
       LONG cdiBindHandle,
       struct UpdateInfoStruct *info,
       LONG infoSize,
       LONG reasonFlag);
```

Parameters:

Inputs:

cdmosHandle The CDM's handle for using the **CDI_** APIs. The value of *cdmosHandle* was assigned during **CDI_Register_CDM()**, and it is used in conjunction with the CDM-generated *CDMHandle* to uniquely identify a CDM when it interfaces with the NWPA through the **CDI_** API set.

cdiIBindHandle The NWPA-generated bind handle that was assigned to the calling CDM when it bound to the target device using **CDI_Bind_CDM_To_Object()**.

info A pointer to an **UpdateInfoStruct**. This structure contains the information telling the NWPA what items will be updated for the target device. For a detailed description of this structure, refer to Chapter 6. The following is the structure's ANSI C definition:

```
struct UpdateInfoStruct
{
       BYTE name[64];
       LONG mediaType;
       LONG cartridgeType;
       LONG unitSize;
       LONG blockSize;
       LONG capacity;
       LONG preferredUnitsize;
       LONG functionMask;
       LONG controlMask;
       LONG unfunctionMask;
       LONG uncontrolMask;
       LONG mediaSlot;
       BYTE activateFlag;
       BYTE removableFlag;
       BYTE readOnlyFlag;
       BYTE magazineLoadedFlag;
       BYTE acceptsMagazinesFlag;
       BYTE objectInChangerFlag;
       BYTE objectIsLoadableFlag;
       BYTE lockFlag;
       LONG diskGeometry;
       LONG reserved[7];
       union
       {
              struct ChangerInfo
              {
```

```
                        LONG numberOfSlots;
                        LONG numberOfExchangeSlots;
                        LONG numberOfDevices;
                        LONG deviceObjects[n];
                } ci;
        } u1;
} ;
```

infoSize — The size of the **UpdateInfoStruct** pointed at by *info*.

reasonFlag — A NWPA recognized code corresponding to the reason why the update is being done. The following is a list of valid codes that may be placed in this field:

ALERT_UNKNOWN	0X00000000
ALERT_DRIVER_UNLOAD	0X00000001
ALERT_DEVICE_FAILURE	0X00000002
ALERT_PROGRAM_CONTROL	0X00000003
ALERT_MEDIA_DISMOUNT	0X00000004
ALERT_MEDIA_EJECT	0X00000005
RESERVED2	0X00000006
RESERVED3	0X00000007
ALERT_MEDIA_LOAD	0X00000008
ALERT_MEDIA_MOUNT	0X00000009
ALERT_DRIVER_LOAD	0X0000000A
RESERVED4	0X0000000B
RESERVED5	0X0000000C
ALERT_MAGAZINE_LOAD	0X0000000D
ALERT_MAGAZINE_UNLOAD	0X0000000E
RESERVED6	0X0000000F
ALERT_CHECK_DEVICE	0X00000010
ALERT_CONFIGURATION_CHANGE	0X00000011
RESERVED7	0X00000012
RESERVED8	0X00000013
ALERT_LOST_HARDWARE_FAULT_TOLERANCE	0X00000014
RESERVED9	0X00000015
RESERVED10	0X00000016
RESERVED11	0X00000017
ALERT_DEVICE_END_OF_MEDIA	0X00000018
ALERT_MEDIA_INSERTED	0X00000019
RESERVED12	0X0000001A
RESERVED13	0X0000001B
RESERVED14	0X0000001C

None

Outputs:

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Object_Update()** is used by a CDM to update device object information with the NWPA. Typically, object updating is done when the CDM needs to deactivate a device or put in capacity, unitsize, or blocksize information for a removable device on a mount. Although it is not a specific NWPA requirement, it is good practice for a CDM to store the device object information for each device it supports into a local structure. Whenever device information is updated, the update information should

also be mirrored into the local storage structure. Doing this allows the CDM to know the current operational information for each device it supports. However, to save the NWPA time and overhead in performing the update, the CDM should allocate a reusable **UpdateInfoStruct** to use exclusively as an input parameter to **CDI_Object_Update()**. Then, when an update is necessary, the CDM should do the following:

1. Set all of the fields of the reusable **UpdateInfoStruct** to -1. This is easily accomplished using the OS routine **CSetB()**.

2. Place the new values in the fields that are to be updated, thereby, leaving a -1 in all of the fields that are not to be updated. The -1 indicates a no-change condition to the NWPA.

> **Note:** Updated field values should be mirrored into the corresponding fields of device's local storage structure.

3. Call **CDI_Object_Update()** to update the device object information with the NWPA.

CDI_Queue_Message

Purpose: Registers an abort routine with the NWPA for a CDM that internally queues CDM messages.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG CDI_Queue_Message(
        LONG msgPutHandle,
        LONG (*AbortRoutine)(),
        LONG abortParameter,
        void (*ExecuteRoutine)(),
        LONG executeParameter);
```

Parameters:

Inputs:

msgPutHandle Handle to the CDM message (**CDMMessageStruct**) from which the SHACB was built. The value of this parameter is obtained from the **MsgPutHandle** field of the **CDMMessageStruct**.

AbortRoutine Address of the CDM's internal queue abort routine. Since an abort routine is registered on a per enqueue basis, a CDM can have more than one. However, within this manual, this routine is generically referred to as *CDM_Abort_CDMMessage()*.

abortParameter Input parameter to *CDM_Abort_CDMMessage()*. This parameter can contain anything that the CDM needs to complete the abort. Typically, this parameter is a handle to the original CDM message that initiated the request. To avoid memory problems, however, this parameter should not be a memory pointer.

ExecuteRoutine **(Optional)** A pointer to a CDM entry point where the NWPA can send postponed requests from the NetWare elevators. This functionality is mainly applicable to CDM filters, and even then it is limited to a small audience of developers. If a developer does not understand the explanation given here, then this is not a feature the developer needs. If not used, which is the typical case, this parameter should be set to zero.

executeParameter **(Optional)** Input parameter to the routine specified in *ExecuteRoutine*. Like *ExecuteRoutine*, this functionality is applicable to a limited audience. Typically, this parameter should be set to zero.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Queue_Message()** is used by a CDM that does internal queuing of CDM messages. Generally, a CDM will not need to do internal queuing, unless the CDM must build multiple HACB requests to accomplish a single CDM message request issued by the NWPA. A CDM must call **CDI_Queue_Message()** each time it queues a message, that is, every time it does not call either **CDI_Execute_HACB()** or **CDI_Chain_Message()** (filter CDMs only) within the context of *CDM_Execute_CDMMessage()* for that message. For each message the CDM queues, **CDI_Queue_Message()** registers an abort routine that can be called by the NWPA in case an abort is issued on that request. **CDI_Queue_Message()** only implies that a message is enqueued. The CDM must provide the actual enqueue/dequeue functionality. Dequeuing is implied when either **CDI_Execute_HACB()**, **CDI_Blocking_Execute_HACB()**, or **CDI_Complete_Message()** is called on the message.

CDI_Register_CDM

Purpose: Registers a CDM with the NWPA.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG CDI_Register_CDM(
        LONG *cdmHandle,
        LONG cdmHandle,
        LONG types,
        BYTE *name,
        LONG npaHandle);
```

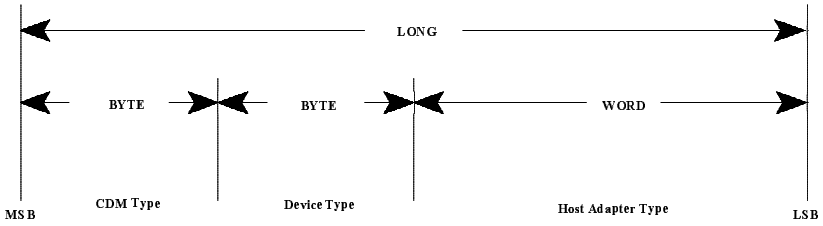
Parameters:

Inputs:

cdmosHandle Address of a local variable of type LONG.

cdmHandle Handle that the CDM generated for itself. This handle is the CDM's own unique identifier. It is used in conjunction with the OS-generated *cdmosHandle* to uniquely identify a CDM when it interfaces with the NWPA through the **CDI_** API set.

types A packed LONG containing information that identifies for the NWPA the CDM's CDM type (filter, enhancer, or base-translator), and the device types and host adapter type it supports. The parameter is divided as follows:



Possible values for CDM types

0x01 Base-Translator
0x02 Enhancer
0x03 Filter

Possible values for device types:

0x00 Direct-access device (magnetic disk)
0x01 Sequential-access device (magnetic tape)
0x02 Printer device
0x03 Processor device
0x04 Write once device (some optical disks)
0x05 CD-ROM device
0x06 Scanner device

0x07 Optical memory device (some optical disks)
0x08 Media changer device (jukebox) or magazine
0x09 Communications device
0x0A-0B Defined by ASC IT8 (Graphic Arts Pre-Press)
0x0C-1E Reserved
0x1F Unknown or no device type
0xFF Call *CDM_Inquiry()* for every type of device

Possible values for host adapter types:
0x0001 SCSI
0x0002 IDE\ATA
0x0003 Custom
0x0004-00FE Reserved
0xFFFF Any bus type

name Length-preceded string containing the CDM's name. Maximum string length is 64 bytes where byte 0 contains the string length and bytes 1 through 63 can contain characters.

npaHandle The CDM's handle for using the **NPA_** APIs. Its value was assigned during **NPA_Register_CDM_Module()**.

Outputs:

cdmosHandle Receives a CDM-OS handle used as a communication token between the CDM and the NWPA. This handle is used in conjunction with the CDM-generated **CDMHandle** to uniquely identify a CDM when it interfaces with the NWPA through the **CDI_** API set.

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Register_CDM()** is used to register the module as a CDM and make its entry points, registered during **NPA_Register_CDM_Module()**, visible to the system. This is the last routine called within *CDM_Load()* prior to *CDM_Load()* returning its thread to the OS calling process.

CDI_Register_Object_Attribute

Purpose: Registers device attributes with the NWPA, which then makes these attributes visible to the application layer.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG CDI_Register_Object_Attribute(
       LONG npaHandle,
       LONG cdmBindHandle,
       struct AttributeInfo *info,
       LONG (*GetRoutine),
       LONG (*SetRoutine));
```

Parameters:

Inputs:

npaHandle The CDM's handle for using the **NPA_** APIs. Its value was assigned during **NPA_Register_CDM_Module**().

cdmBindHandle Handle generated by the CDM to uniquely identify the device. This is the handle the CDM passed to **CDI_Bind_CDM_To_Object**() when it bound to the device.

info A pointer to an **AttributeInfoStruct** structure. This structure contains specific information about an attribute. For a detailed description of this structure, refer to Chapter 6. The following is the ANSI C definition of the structure:

```
struct AttributeInfoStruct
{
       LONG attributeID;
       LONG attributeType;
       LONG attributeLength;
       BYTE attributeName[64];
};
```

GetRoutine Pointer to a local CDM entry point (*CDM_Get_Attribute()*) responsible for returning attribute information. The following is the ANSI C prototype of this entry point:

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

For a given attribute, the CDM indicates the expected data type of the **InfoBuffer** input parameter by the value it places in the **AttributeType** field of the attribute's **AttributeInfoStruct** at registration. A pointer to this structure is passed to the attribute registration routine,

CDM_Get_Attribute() places the return attribute information in the location pointed at by the *InfoBuffer* input parameter and the byte-length of the return information in the location pointed at by the *infoBufferLength* input parameter.

SetRoutine If the attribute is not settable, this field is set to zero. If the attribute is settable, this field contains a pointer to a local CDM entry point (*CDM_Set_Attribute()*) responsible for setting attribute information. The following is the ANSI C prototype of this entry point:

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

CDM_Set_Attribute() sets the attribute to the information contained in the infoBuffer input parameter. The length of this buffer is specified in the infoBufferLength input parameter. If the attribute change affects any of the information that the CDM originally reported to the NWPA during its bind to the device, it must update these changes to the NWPA by filling out the appropriate fields of an **UpdateInfoStruct** and calling **CDI_Object_Update()**. The context of the set routine is blocking; therefore, the CDM can issue any necessary commands to set the mode of the device.

None

Outputs:

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Register_Object_Attribute()** allows a CDM to present attribute information about a device it manages to the application layer. To present the information, a CDM must register a get-routine (*CDM_Get_Attribute()*) that returns attribute information into a buffer provided by the calling process. If a device attribute can be changed by an application, then the CDM must register a set-routine (*CDM_Set_Attribute()*).

CDI_Return_HACB

Purpose: Returns memory allocated for a SHACB back to the system memory pool.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG CDI_Return_HACB (
       LONG cdmosHandle,
       LONG hacbPutHandle);
```

Parameters:

Inputs:

cdmosHandle The CDM's handle for using the **CDI_** APIs. The value of *cdmosHandle* was assigned during **CDI_Register_CDM**(), and it is used in conjunction with the CDM-generated *CDMHandle* to uniquely identify a CDM when it interfaces with the NWPA through the **CDI_** API set.

hacbPutHandle Handle to the HACB being deallocated. The value of this parameter is obtained from the **hacbPutHandle** field of the original SHACB's member HACB.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Return_HACB**() is used by a CDM to return the memory allocated for a SHACB to the system memory pool. Typically, **CDI_Return_HACB**() is called when a SHACB structure becomes corrupted and cannot be reused for building subsequent requests or when the CDM is ready to unload.

CDI_Rescan_Bus

Purpose: This API is used by the CDM to update the NWPA's device object database anytime the CDM changes the private/public status of a device it controls.

Architecture Type: All

Thread Context: Blocking

Syntax: `LONG CDI_Rescan_Bus ( LONG npaBusID ):`

Parameters:

Inputs:

npaBusID The object ID that the NWPA assigned to the target bus in its object database. The CDM received this target ID as an input parameter to its *CDM_Inquiry()* entry point.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: The primary use of this API is to place devices that were originally detected by the CDM via the Case 2 scan (see *HAM_Scan_For_Devices*) back into the object database maintained by the Media Manager so that they can be available to other applications.

CDI_Unbind_CDM_From_Object

Purpose: Unbinds a CDM from a device object.

Architecture Type: All

Thread Context: Blocking

Syntax:

```
LONG CDI_Unbind_CDM_From_Object (
       LONG cdmosHandle,
       LONG cdiBindHandle);
```

Parameters:

Inputs:

cdmosHandle The CDM's handle for using the **CDI_** APIs. The value of *cdmosHandle* was assigned during **CDI_Register_CDM**(), and it is used in conjunction with the CDM-generated *CDMHandle* to uniquely identify a CDM when it interfaces with the NWPA through the **CDI_** API set.

cdiBindHandle The NWPA-generated bind handle that was assigned to the calling CDM when it bound to the target device using **CDI_Bind_CDM_To_Object**().

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Unbind_CDM_From_Object**() is used by the CDM to unbind itself from a device. When a CDM is unbound, it no longer has to handle requests for that device. Typically, the CDM calls this routine at unload time within the context of *CDM_Unload()*. However, if somehow the CDM determines that it should no longer support a device, it can call **CDI_Unbind_CDM_From_Object**(), and it will no longer have to handle requests for that device.

CDI_Unregister_CDM

Purpose: Unregisters a CDM and its entry points from the NWPA.

Architecture Type: All

Thread Context: Blocking

Syntax:

```
LONG CDI_Unregister_CDM (
       LONG cdmosHandle,
       LONG cdmHandle);
```

Parameters:

Inputs:

cdmosHandle The CDM's handle for using the **CDI_** APIs. The value of *cdmosHandle* was assigned during **CDI_Register_CDM()**, and it is used in conjunction with the CDM-generated *CDMHandle* to uniquely identify a CDM when it interfaces with the NWPA through the **CDI_** API set.

cdmHandle Handle that the CDM generated for itself. This handle is the CDM's own unique identifier. It is used in conjunction with the OS-generated *cdmosHandle* to uniquely identify a CDM when it interfaces with the NWPA through the **CDI_** API set. Also, the CDM must be able to access its device list through this handle.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **CDI_Unregister_CDM()** is used to unregister the CDM from the NWPA prior to being unloaded. It is called within the context of *CDM_Unload()* to flush pending I/O before being the CDM is unloaded.

CDM_Abort_CDMMessage

Purpose: The CDM's entry for receiving aborts on messages it has queued.

Thread Context: Non-Blocking

Syntax:
```
LONG CDM_Abort_CDMMessage (LONG parameter);
```

Parameters:

Inputs:

parameter The NWPA passes the value of this parameter, which is the *parameter* specified as an input argument to **CDI_Queue_Message()**. The CDM decides the value of this parameter, which can be anything it needs to complete the abort. Typically, this parameter is a handle to the original CDM message that initiated the request. To avoid memory problems, this parameter should not be a memory pointer.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: *CDM_Abort_CDMMessage()* is the CDM's entry point for receiving requests to abort messages in its process queue. This routine, and its input parameter, become visible to the NWPA during **CDI_Queue_Message()**. The CDM is required to provide *CDM_Abort_CDMMessage()* only if it will provide its own internal request queue. CDMs that support devices, such as tape devices, that require multiple HACB requests to execute a command fall into this category. For such devices, *CDM_Abort_CDMMessage()* must provide the means to not only remove pending HACB requests from a queue, it must be able to abort HACB requests already sent to the HAM by calling **CDI_Abort_HACB()**

CDM_Callback

Purpose: The CDM's entry point for being notified of the completion of a non-blocking HACB request.

Thread Context: Non-Blocking

Syntax:

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

Parameters:

Inputs:

SHACB The NWPA passes the value of this parameter, which is a pointer to the SHACBStruct encapsulating the **HACBStruct** that contains the data of the request just completed. For a detailed description of this structure and its member **HACBStruct**, refer to Chapter 3. The following is the structure's ANSI C definition:

```
typedef struct SHACBStruct
{
       LONG cdmSpace[8];
       struct hacbStruct HACB;
} SHACB;
```

npaCompletionCode The NWPA generates and passes the value of this parameter, which is a completion code for an internal NWPA process. If the value of this parameter is zero, it means that the value in the HACB's **hacbCompletion** field is valid; therefore, normal callback processing should be performed. If the value of this parameter is non-zero, it means that an internal messaging error has occurred. In this case, *CDM_Callback()* should simply complete the request by calling **CDI_Complete_Message()** passing it the value of *NPACompletionCode* as the API's *NPACompletionCode* input parameter.

Outputs: None

Return Value: 0 to succeed

Description: *CDM_Callback()* is the CDM's entry point for being notified of HACB completion. Within the context of *CDM_Callback()*, the CDM can check a HACB's completion status (provided *NPACompletionCode* == 0) and determine a course of action. Depending on a HACB's completion status, contained in the HACB's **hacbCompletion** field, the CDM can do one of the following:

Option 1: If the HACB completion status is successful (**hacbCompletion**=0x0000), complete the HACB by calling **CDI_Complete_Message()** with a value of zero in the *NPACode* input parameter.

Option 2: If the HACB completion status indicates an error (**hacbCompletion**=0x0001 to 0x0008), translate the error into an appropriate NWPA error code, and complete the HACB by calling **CDI_Complete_Message()** with the NWPA error code as the value in the *NPACode* input parameter.

Option 3: If the HACB completion status indicates an error, spawn a blocking, error handling thread to try and remedy the error. In this situation, the CDM must provide some error handling routines. If the error handling routine can remedy the error, then within its context it should complete the HACB as described in option 1. If the error could not be remedied, then the error handling routine should complete the HACB as described in option 2.

CDM_Callback() becomes visible to the NWPA when the CDM executes a HACB request by calling **CDI_Execute_HACB()**. Along with a pointer to the HACB to be executed, the CDM supplies the address of the *CDM_Callback()* as an input parameter to **CDI_Execute_HACB()**. The CDM must supply these parameters for each HACB request it executes. The NWPA associates the specified HACB request with the specified callback routine, and makes the callback after the HACB request completes. Since a callback routine is specified for each call to **CDI_Execute_HACB()**, the CDM can provide either one all-inclusive callback routine or a set of callback routines where each provides specific functionality specially designed for a certain type of HACB request. In this manual, however, the term *CDM_Callback()* is used to generically refer to either case.

Important: *CDM_Callback()* should not hold the current thread for any lengthy amount of time, and it must not make any calls to blocking processes. If blocking threads such as error handling threads are necessary, then *CDM_Callback()* should spawn them using **NPA_Spawn_Thread()**, and then relinquish control by returning to the calling process.

CDM_Check_Option

Purpose: The CDM's entry point for accepting and verifying the command line options parsed by **NPA_Parse_Options()** are valid for the CDM.

Thread Context: Non-Blocking

Syntax:

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

Parameters:

Inputs:

option The NWPA passes the value of this parameter, which is a pointer to the **NPAOptionStruct** associated with this instance of the CDM module. The following is the structure's ANSI C definition:

```
struct NPAOptionStruct
        {
                BYTE name[32];
                LONG parameter0;
                LONG parameter1;
                LONG parameter2;
                WORD type;
                WORD flags;
                BYTE string[n];
        };
```

instance The NWPA passes the value of this parameter, which is a CDM-generated number identifying a device instance. The NWPA will use this number to associate different groups of options with a particular device being managed by the CDM.

flag The NWPA passes the value of this parameter, which indicates the process that called *CDM_Check_Option()*. This parameter is defined as follows;

0x00000000 Called by **NPA_Parse_Options()**.
0x00000001 Called by **NPA_Register_Options()**.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: *CDM_Check_Option()* is registered with the NWPA during **NPA_Register_CDM_Module()**, and it is called by the NWPA during two different phases of CDM initialization. *CDM_Check_Option()* is called by **NPA_Parse_Options()** during the command-line parsing phase and again by **NPA_Register_Options()** during the options registration phase.

When called under the context of **NPA_Parse_Options**(), the CDM should only determine if the current option is acceptable. Under this context, the NWPA has not physically associated the options with a device instance in its database.

When called under the context of **NPA_Register_Options**(), the NWPA has already placed the options in its database, and the CDM can set its operational states accordingly.

Since CDMs do not directly interface with the hardware, they should not attempt to register for hardware options such as interrupts, DMA channels, ports, etc. CDM command-line options should only set software, operational modes for the CDM.

If the CDM determines that an error occurred in registering its options, it will need the to unregister these options using **NPA_Unregister_Options**() passing *Instance* as an input parameter.

CDM_Execute_CDMMessage

Purpose: The CDM's entry point for receiving a CDM message which routes them to the proper CDM control or I/O routine to build a SHACB request.

Thread Context: Non-Blocking

Syntax:

```
LONG CDM_Execute_CDMMessage(
       LONG cdmBindHandle,
       struct CDMMessageStruct *msg);
```

Parameters:

Inputs:

cdmBindHandle The NWPA passes the value of this parameter, which is a handle to the device being targeted by the CDM Message request (**CDMMessageStruct**). The CDM generated the value of *cdmBindHandle* during the context of *CDM_Inquiry()* when it bound to the device. The CDM bound to the device by calling **CDI_Bind_CDM_To_Object**(). From this handle, the CDM locates the target device's information including the HAM-generated **DeviceHandle** and the NWPA-generated **NPABusID**.

msg The NWPA passes the value of this parameter, which is a pointer to the **CDMMessageStruct** containing the data from which a CDM control or I/O routine will build a SHACB. For a detailed description of this structure refer to Chapter 6. The following is the ANSI C definition:

```
struct CDMMessageStruct
{
       LONG msgPutHandle;
       LONG function;
       LONG parameter0;
       LONG parameter1;
       LONG parameter2;
       LONG bufferLength;
       void* buffer;
       LONG cdmReserved[2]; } ;
```

Outputs: None

Return Value: Returns the return value of the internal CDM routine called to service the request:
0 if the CDM routine executed successfully.
Non-zero if the specified function is not supported by the CDM.

Description: *CDM_Execute_CDMMessage()* is the CDM's entry point for receiving and routing a CDM message to the proper CDM routine that will convert the message into a SHACB.

CDM_Get_Attribute

Purpose: The CDM entry point from which applications may retrieve attribute information for a specific attribute.

Thread Context: Non-Blocking

Syntax:

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

Parameters:

Inputs:

cdmBindHandle The NWPA passes the value of this parameter, which is a handle to the device being targeted by the CDM Message request (**CDMMessageStruct**). The CDM generated the value of *cdmBindHandle* during the context of *CDM_Inquiry()* when it bound to the device. The CDM bound to the device by calling **CDI_Bind_CDM_To_Object**(). From this handle, the CDM locates the target device's information including the HAM-generated **DeviceHandle** and the NWPA-generated **NPABusID**.

infoBuffer This points to where the information associated with the attribute being retrieved will be stored by *CDM_Get_Attribute()*.

infoBufferLength Size of the *infoBuffer* in bytes.

attributeID The ID of the attribute selected. This is the ID that was registed by the CDM for this attribute during **CDI_Register_Object_Attribute**().

Outputs: None

Return Value: 0 to succeed.

Description: *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.

CDM_Inquiry

Purpose: The CDM's entry point for inquiring online devices and determining whether or not it will bind to the device.

Thread Context: Blocking

Syntax:

```
LONG CDM_Inquiry(
       LONG npaDeviceID,
       LONG npaBusID,
       struct DeviceInfoStruct *deviceInfo,
       LONG flag,
       LONG cdmHandle);
```

Parameters:

Inputs:

npaDeviceID The NWPA passes the value of this parameter, which is the object ID that the NWPA assigned to the target device in its device database.

npaBusID The NWPA passes the value of this parameter, which is the object ID that the NWPA assigned to the target bus in its object database. If *Flag* is set to 0x00000003 or 0x00000004, this is the only valid parameter for this API. All other parameters will be set to 0.

deviceInfo The NWPA passes the value of this parameter, which is a pointer to a **DeviceInfoStruct**. The HAM supporting the target device fills in this structure with all the pertinent device information that the CDM may need to send I/O to the device and determine if it should bind to the device. Additionally, this structure has an **InquiryInfoStruct** as a data member that contains bus-specific inquiry information. For a detailed description of this structure, refer to Chapter 6. The following is the structure's ANSI C definition:

```
typedef struct DeviceInfoStruct
       {
       LONG deviceHandle;
       BYTE deviceType;
       BYTE unitNumber;
       BYTE busID;
       BYTE cardNo;
       LONG attributeFlags;
       LONG haxDataPerTransfer;
       LONG haxLengthSGElement;
       BYTE haxSGElements;
       BYTE reserved1[2];
       BYTE elevatorThreshold;
       LONG maxUnitsPerTransfer;
       WORD haType;
       union /* Device Specific Information */
       {
              struct /* SCSI Synchronous Information */
              {
                     BYTE transferPeriodFactor;
                     BYTE offset;
              } SCSI;
              struct /* Other Device Information */
```

```
                {
                        BYTE reserved2[2];
                } OTHER;
        } INFO;
        struct InquiryInfoStruct InquiryInfo;
}deviceInfoDef;
```

flag The NWPA passes the value of this parameter, which indicates the type of inquiry to perform. This parameter can have one of the following values:

0x00000000 Indicates a new device and the CDM should check it and bind to it if the device meets the CDM's bind conditions.

0x00000001 (Applies only to filter CDMs) Indicates that the CDM is already bound to the specified device, but device information has changed. Therefore, the CDM may need to bind again or issue an object update. To base-translator and enhancer CDMs, this constitutes a no-op.

0x00000002 Indicates to the CDM that the specified device is no longer valid; therefore, the CDM should remove the device from its list and free any local structures associated with the device.

0x00000003 Indicates to the CDM that an End of Bus condition has occurred during a Scan For New Devices. This means that 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 more details, refer to **Chapter 8 HACB Type Zero Functions** under Function 1- *HAM_Scan_For_Devices* If this flag is set, *NPABusID* is the only valid parameter for this API. All other parameters will be set to 0.

0x00000004 Indicates to the CDM that 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. This is done by using Scan case 3 of *HAM_Scan_For_Devices* If this flag is set, *NPABusID* is the only valid parameter for this API. All other parameters will be set to 0.

The NWPA passes the value of this parameter, which is the identifier the CDM generated for itself and registered with the NWPA during **CDI_Register_CDM()**.

cdmHandle None

Outputs:

Return Value: 0 to succeed.

Description: *CDM_Inquiry()* is the CDM's entry point for logically binding to a device. A logical bind means that the CDM will field message requests for the device, and indicates this to the NWPA by calling **CDI_Bind_CDM_To_Object()** and returning zero from this routine. This entry point gets registered with the NWPA during **NPA_Register_CDM_Module()**. Immediately after CDM registration, the NWPA calls *CDM_Inquiry()* for each device matching the device type that the CDM registered for with **CDI_Register_CDM()**. It receives subsequent calls each time a new device with that device type comes online. The CDM registers the device types it will support--along with the host adapter interface it will support--by placing the appropriate values in the *Types* input parameter of **CDI_Register_CDM()**.

CDM_Inquiry() is responsible for building and maintaining a CDM's device list. It does this by binding to devices matching the device type the CDM is designed to support. To bind to a device, a CDM must generate a *CDMBindHandle* from which the CDM can identify the device and access essential device information, such as the device's handle and the handle of the HAM supporting the device. Next, it must create an instance of an **UpdateInfoStruct** for the device, fill in its fields with the appropriate information, and pass both the CDMBindHandle and a pointer to the **UpdateInfoStruct** to **CDI_Bind_CDM_To_Object()**. This is all done within the context of *CDM_Inquiry()*. *CDM_Inquiry()* is a blocking process, and part of its purpose is to allow a CDM the opportunity to issue non-intrusive commands (such as a mode sense) to determine if it should bind to the device. These commands should be issued using **CDI_Blocking_Execute_HACB()**. The CDM should not issue any command that may change the state of the device during the context of *CDM_Inquiry()*.

Note: If the CDM decides not to logically bind to a device, *CDM_Inquiry()* must return a non-zero return code.

CDM_Set_Attribute

Purpose: This is the local CDM entry point responsible for setting attribute information for a specific attribute.

Thread Context: Non-Blocking

Syntax:

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

Parameters:

Inputs:

cdmBindHandle The NWPA passes the value of this parameter, which is a handle to the device being targeted by the CDM Message request (**CDMMessageStruct**). The CDM generated the value of *cdmBindHandle* during the context of *CDM_Inquiry()* when it bound to the device. The CDM bound to the device by calling **CDI_Bind_CDM_To_Object**(). From this handle, the CDM locates the target device's information including the HAM-generated **DeviceHandle** and the NWPA-generated **NPABusID**.

infoBuffer This points to where *CDM_Set_Attribute* will find the information regarding the desired setting of the selected attribute.

infoBufferLength Size of the *infoBuffer* in bytes.

attributeID The ID of the attribute to be set. This is the ID that was registed by the CDM for this attribute during **CDI_Register_Object_Attribute**().

Outputs: None

Return Value:

Description: *CDM_Set_Attribute()* is the entry point from 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.

CDM_Load

Purpose: The CDM's load-time entry point for initializing and registering the CDM.

Thread Context: Blocking

Syntax:

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

Parameters:

Inputs:

loadHandle The OS assigns the value of this parameter when it receives a command line request to load the CDM. Its value is the CDM's load handle, and the OS uses this handle to keep track of the CDM.

screenID The OS passes the value of this parameter, which is a handle to the console. The **NPA_** routines that output messages to the console require this handle as an argument, and it is provided in case the CDM needs to output any screen messages during its initialization.

commandLine This parameter is a pointer to the command line. The OS passes this pointer so that the CDM can receive command line configuration options if any are required.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful (fails the load).

Description: *CDM_Load()* is the initial entry point for a CDM, and it performs CDM initialization and registration. This routine becomes visible to the OS when the definition (.DEF) file is processed by the NLMLINK utility. When the CDM is loaded, the OS calls *CDM_Load()* passing it three parameters, *loadHandle*, *screenID*, and *commandLine*. *loadHandle* and *screenID* are generated by the OS to be used in allocating resources and for outputting console error messages that may occur during the load process. *commandLine* is a pointer to the command line arguments specified by the system operator at load time. These arguments may specify any configuration required by the CDM, provided that the CDM supports command line options.

CDM_Unload

Purpose: The CDM's last unload-time entry point that prepares the CDM for unloading and returns resources back to the system.

Thread Context: Blocking

Syntax:
```
LONG CDM_Unload (void);
```

Parameters: None

Return Value: 0 to succeed

Description: *CDM_Unload()* is the CDM's entry point from the OS when it receives an UNLOAD command for this CDM. *CDM_Unload()* is then responsible for releasing all appropriate resources, cleaning up any Hacbs it generated and issued, and unregistering the CDM. Upon entry into this routine the CDM needs to stay operational until all its pending I/O is flushed and the NWPA quiesces any incoming I/O. To do this, the first call the CDM should make within *CDM_Unload()* is to **CDI_Unregister_CDM()**. It is during the context of **CDI_Unregister_CDM()** that the NWPA flushes pending I/O and quiesces new I/O for this CDM. Upon return from **CDI_Unregister_CDM()**, the CDM is guaranteed not to have any pending I/O. It is at this point that the CDM starts cleaning up its resources.

Note: The CDM absolutely must abort any outstanding Hacbs it generates and issues, such as Asynchronous Event Notification Hacbs. Otherwise, the CDM will cause the server to Abend.

CDM_Unload_Check

Purpose: The CDM's first unload-time entry point that checks to see which devices are currently bound to the CDM. This entry point is called by the OS prior to calling *CDM_Unload()*.

Thread Context: Non-Blocking

Syntax:

```
LONG CDM_Unload_Check (LONG screenID);
```

Parameters:

Inputs:

screenID The OS passes the value of this parameter, which is a handle to the console. The **NPA_** routines that output messages to the console require this handle as an argument.

Outputs: None

Return Value: 0 if no devices are locked, meaning that the CDM can be cleanly unloaded. Non-zero if the CDM is bound to one or more devices.

Description: *CDM_Unload_Check()* is called when the OS receives a request from the console to unload the CDM . This routine is responsible for checking to see if any of the CDM's devices are currently being used (locked) by an application. *CDM_Unload_Check()* checks the lock status for a particular adapter by calling **NPA_Unload_Module_Check()**. The OS looks at the return value to determine if the CDM can be cleanly unloaded, meaning that there are no current I/O processes on the devices controlled by this CDM. 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.

HAI_Activate_Bus

Purpose: Activates a bus instance managed by the HAM.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG HAI_Activate_Bus(
       LONG *npaBusHandle,
       LONG hamBusHandle,
       LONG npaHandle);
```

Parameters:

Inputs:

npaBusHandle Address of a local variable of type LONG.

hamBusHandle HAM-generated handle to a bus instance the HAM is managing. From this handle, the HAM must be able to locate its list of devices attached to the bus.

npaHandle The HAM's handle for using the **NPA_** APIs. Its value was assigned during **NPA_Register_HAM_Module()**.

Outputs:

npaBusHandle Receives an NWPA generated handle for the target bus the HAM is managing. This handle is the NWPA's counterpart to the HAM's *hamBusHandle*. This handle is used in conjunction with the HAM-generated *hamBusHandle* to uniquely identify a HAM when it interfaces with the NWPA through the **HAI_** API set.

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **HAI_Activate_Bus()** is used to activate a bus instance managed by the HAM. This is the last API called within *HAM_Load()* prior to *HAM_Load()* returning its thread to the OS.

HAI_Complete_HACB

Purpose: Used by the HAM to complete a HACB I/O request.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:
```
LONG HAI_Complete_HACB (LONG hacbPutHandle);
```

Parameters:

Inputs:

hacbPutHandle Value of the **hacbPutHandle** field of the HACB being completed. This handle is assigned by the NWPA when a CDM issues the HACB to the HAM.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **HAI_Complete_HACB()** is used to post completion of a HACB I/O request to the NWPA, whether the request completed successfully, with an error, or aborted.

HAI_Deactivate_Bus

Purpose: Deactivates a bus instance of the HAM.

Architecture Type: All

Thread Context: Blocking or Non-Blocking

Syntax:

```
LONG HAI_Deactivate_Bus(
       LONG npaBusHandle,
       LONG hamBusHandle,
       LONG npaHandle);
```

Parameters:

Inputs:

npaBusHandle NWPA-generated handle to the target bus. This parameter was output to the HAM from **HAI_Activate_Bus()** when the bus was activated.

hamBusHandle HAM-generated handle to the target bus instance the HAM is managing. From this handle, the HAM must be able to locate its list of devices attached to the bus. The HAM passed this parameter to **HAI_Activate_Bus()** when the bus was activated.

npaHandle The HAM's handle for using the **NPA_** APIs. Its value was assigned during **NPA_Register_HAM_Module()**.

Outputs: None.

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **HAI_Deactivate_Bus()** is used to deactivate a bus in preparation for the HAM to be unloaded. It is called within the context of *HAM_Unload()* to flush pending I/O before being the HAM is unloaded. This API must be called for each bus instance that the HAM supports.

HAI_PreProcess_HACB_Completion

Purpose: Used to allow a diagnostic NLM to interject HACB errors.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
void HAI_PreProcess_HACB_Completion (LONG hacbPutHandle);
```

Parameters:

Inputs:

hacbPutHandle Handle to the HACB request being preprocessed. The value of this parameter is obtained from the **hacbPutHandle** field of the current HACB.

Outputs: None

Return Value: None

Description: A HAM only uses **HAI_PreProcess_HACB_Completion()** if the NWDIAG option was specified on the command line. The HAM calls **HAI_PreProcess_HACB_Completion()** after the HACB request has been processed by a device, but before the HAM determines proper queue state and completes the HACB using **HAI_Complete_HACB()**. For more information, refer to section 4.3.5.

HAM_Abort_HACB

Purpose: Aborts HACB requests received by a HAM.

Thread Context: Non-Blocking

Syntax:

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

Parameters:

Inputs:

hamBusHandle The NWPA passes the value of this parameter, which is the HAM-generated handle to the target bus instance the HAM is managing. From this handle, the HAM must be able to locate its list of devices attached to the bus. The HAM passed this parameter to **HAI_Activate_Bus()** when the bus was activated.

HACB The NWPA passes a pointer to the HACB request that is to be aborted. Refer to Chapter 3 for a definition and description of this structure.

flag The NWPA passes the value of this parameter. The value of this parameter indicates the type of abort to perform. Its possible values are:

0x00000000	This value tells the HAM to unconditionally abort the HACB even if it has already been sent to the device.
0x00000001	This value tells the HAM to conditionally abort the HACB if aborting only entails the unlinking of the HACB from the device queue. This is a clean abort.
0x00000002	This value tells the HAM to check and see if the HACB can be cleanly aborted, but not to perform an abort.

Outputs: None

Return Value: The following table indicates the proper return value associated with each input flag value:

Input Flag↔ --- **Return Value** ↕	**Unconditional Abort** 0x00000000	**Conditional Abort** 0x00000001	**Check Abort Status** 0x00000002
0	Indicates the HACB was cleanly aborted. The HACB was completed with the Abort completion code within the context of this routine.	Same as Unconditional Abort.	Indicates clean abort if an abort was to be issued on the HACB

Input Flag↔ --- **Return Value** ↕	**Unconditional Abort** 0x00000000	**Conditional Abort** 0x00000001	**Check Abort Status** 0x00000002
-1	Indicates that the HACB could not be aborted cleanly within the current thread context. The HAM will flag the HACB and abort it later during its ISR. This means the HAM will complete the HACB with the Abort completion code in the ISR.	Indicates that the HACB could not be aborted cleanly during the context of this routine. Therefore, the HAM took no action on the HACB.	Indicates dirty abort if an abort was to be issued on theHACB.
-2	The HAM could not find the target HACB. Essentially, the HAM lost the HACB request. The result of losing the HACB will be an Abend.		

Description: *HAM_Abort_HACB()* is the HAM's entry point for aborting I/O requests, and it is a non-blocking routine. This routine is registered with the NWPA during **NPA_Register_HAM_Module()**. The NWPA passes three arguments to *HAM_Abort_HACB()*. The first two arguments are exactly the same as those passed to *HAM_Execute_HACB()*. The third argument is the *Flags* parameter, and its value indicates the conditions that determine the abort type. When an unconditional abort is indicated, *HAM_Abort_HACB()* is required to cancel the indicated HACB request no matter what. If the HACB is currently in the device queue, the abort merely entails unlinking the HACB from the queue, placing the abort code (0x0004) in its **hacbCompletion** field, calling **HAI_Complete_HACB()**, and returning a zero. This abort case is referred to as a clean abort. If the HACB has already been sent to the device, then the value in *Flags* must be visible to *HAM_ISR()* so that it can abort the HACB request even after it was processed by the device. The NWPA guarantees that aborts are done during a single thread with interrupts disabled; therefore, no new requests are pulled from the device queue and issued to the device during an abort sequence. This ensures that a calling process can issue a clean abort check, and if the abort can be done cleanly, issue the abort without entering a critical-race window where the request gets sent to the device somewhere between the check request and the abort request.

HAM_Check_Option

Purpose: The HAM's entry point for accepting and verifying the command line options parsed by **NPA_Parse_Options()** are valid for the HAM. These command line options indicate hardware resources such as interrupts, ports, DMA channels, shared memory decoding, etc.

Thread Context: Non-Blocking

Syntax:

```
LONG HAM_Check_Option(
        struct NPAOptionStruct *option,
        LONG instance,
        LONG flag);
```

Parameters:

Inputs:

option The NWPA passes the value of this parameter, which is a pointer to the **NPAOptionStruct** associated with this instance of the HAM module. The following is the structure's ANSI C definition:

```
struct NPAOptionStruct
{
        BYTE name[32];
        LONG parameter0;
        LONG parameter1;
        LONG parameter2;
        WORD type;
        WORD flags;
        BYTE string[n];
} ;
```

instance The NWPA passes the value of this parameter, which is a HAM-generated number corresponding to an adapter card instance being managed by the HAM. The NWPA will use this number to group a set of hardware options with a particular adapter instance.

flag The NWPA passes the value of this parameter, which indicates the process that called *HAM_Check_Option()*. This parameter is defined as follows;

0x00000000	Called by **NPA_Parse_Options()**
0x00000001	Called by **NPA_Register_Options()**.

None

Outputs:

Return Value: 0 to accept option.
Non-zero to reject option.

Description: *HAM_Check_Option()* is registered with the NWPA during **NPA_Register_HAM_Module()**, and it is called by the NWPA during two different phases of HAM initialization. *HAM_Check_Option()* is called by

NPA_Parse_Options() during the command-line parsing phase and again by **NPA_Register_Options()** during the options registration phase.
NPA_Parse_Options() iteratively calls *HAM_Check_Option()* for each option found in the HAM's select list. *HAM_Check_Option()* is responsible for accepting or rejecting the selected option. This routine can logically check the compatibility of the option combination for each iteration. If the option is accepted, then the NWPA places the option into a use list. The HAM should not try to ping any resources under this context because it does not physically own them at this time.
NPA_Register_Options() iteratively calls *HAM_Check_Option()* for each option found in the HAM's use list. *HAM_Check_Option()* again is responsible for accepting or rejecting the selected option. This time, however, the HAM can ping resources to validate them because the NWPA physically registers them for the HAM. If the HAM determines that an error occurred in registering its options, it will need the to unregister these options using **NPA_Unregister_Options()** passing Instance as an input parameter. Also, if a HAM is to support hot replacement, this routine should be designed to accept configuration data from the module being replaced. The NWPA quiesces requests on the elevator of the active HAM while the two modules swap data. To properly support data swapping, the HAMs should pass data indexes rather than data pointers.

HAM_Execute_HACB

Purpose: The HAM's entry point for receiving a HACB request and routing it to the appropriate device queue.

Thread Context: Non-Blocking

Syntax:

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

Parameters:

Inputs:

hamBusHandle The NWPA passes the value of this parameter, which is the HAM-generated handle to a bus instance the HAM is managing. From this handle, the HAM must be able to locate its list of devices attached to the bus.

HACB The NWPA passes a pointer to the HACB request that is to be processed.

Outputs: None

Return Value: 0 to succeed.

Description: *HAM_Execute_HACB()* is the HAM's entry point for receiving and executing I/O requests, and it must be a non-blocking routine. This routine is registered with the NWPA by the use of **NPA_Register_HAM_Module()**.

Note: *HAM_Execute_HACB()* is responsible for controlling the device queue state. For more information about how this entry point controls queue state, refer to section 4.3.1.3. For more information about indicating queue state to the CDM, refer to the description of the HACB's **hacbCompletion** field in Chapter 3 and Appendix B.

HAM_ISR

Purpose: The HAM's interrupt-time entry point. This entry point determines the request causing an interrupt, completes I/O transfers, posts HACB completion status, and completes HACB requests.

Thread Context: Interrupt Level, Non-Blocking

Reqirements: This routine cannot make calls to blocking routines.

Syntax:
```
LONG HAM_ISR (LONG irqLevel);
```

Parameters:

Inputs:

irqLevel The OS passes the value of this parameter, which is a value indicating the interrupt level on which to take action. The HAM specified the interrupt level in the **Option parameter0** field of the **NPAOptionStruct** registered for the HAM during **NPA_Register_Options**().

Outputs: None

Return Value: 0 if interrupt was serviced successfully.
Non-zero if interrupt was not serviced.

Description: *HAM_ISR()* is registered with the NWPA during **NPA_Register_HAM_Module**(), and it is the HAM's entry point for being notified of hardware interrupts. The term "notified" is used here because actual hardware interrupts are vectored to a system ISR within the OS. The NWPA automatically channels the interrupt from the OS to the HAM through this entry point, and the state upon entering *HAM_ISR()* is with interrupts disabled. *HAM_ISR()* must determine the adapter that caused the interrupt, determine if an error occurred for the request, complete the HACB, and send a new HACB to the device from the device's process queue. If no error occurred, then *HAM_ISR()* transfers I/O data to/from the buffer indicated in the HACB (in the case of programmed I/O), places the appropriate completion code in the **hacbCompletion** field, calls **HAI_Complete_HACB**(), and sends a new HACB request in the process queue to the device. If an error occurred, then *HAM_ISR()* freezes that device's process queue, places the appropriate error completion code in the **hacbCompletion** field, and calls **HAI_Complete_HACB**() on the HACB.

Note: *HAM_ISR()* is responsible for controlling the device queue state, and for calling a diagnostic hook if a HAM-local diagnostic flag is set. For more information about how this entry point controls queue state, refer to section 4.3.1.3.

For more information about indicating queue state to the CDM, refer to the description of the HACB's **hacbCompletion** field in Chapter 3 and Appendix B.

HAM_Load

Purpose: The load-time entry point for initializing and registering a HAM.

Thread Context: Blocking

Syntax:

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

Parameters:

Inputs:

loadHandle The OS assigns the value of this parameter when it receives a command line request to load the HAM. This handle is used to identify the HAM.

screenID The OS passes this parameter's value, which is a handle to the console. The **NPA_** routines that output messages to the console require this handle as an argument, and it is provided in case the HAM needs to output any screen messages during its initialization.

commandLine This parameter is a pointer to the command line. The OS passes this pointer so the HAM can receive command line configuration options.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful (fails the load).

Description: *HAM_Load()* is the initial entry point for a HAM, and it performs HAM initialization and registration. This routine becomes visible to the OS when the definition (.DEF) file is processed by the NLMLINK utility. When the HAM is loaded, the OS calls *HAM_Load()* passing it the parameters listed above. *loadHandle* and *screenID* are generated by the OS to be used in allocating resources and for outputting console error messages that may occur during the load process. *commandLine* is a pointer to the command line arguments specified by the system operator at load time. These arguments specify I/O port addresses and ranges, memory decode addresses and lengths, interrupts, and DMA addresses.

Note: Since the HAM may need to do some I/O with an adapter during its initialization, *HAM_Load()* is a blocking process. It is called within the context of the NetWare LOAD utility. Additionally, the HAM may disable interrupts (see **NPA_Interrupt_Control()**) within the context of this routine if the adapters being checked are under heavy I/O traffic. Disabling interrupts may not be necessary, however, if the HAM does disable interrupts within *HAM_Load()*, the HAM must enable them before returning from *HAM_Load()*.

HAM_Software_Hot_Replace

Purpose: The HAM's entry point for exchanging configuration information with a newer version HAM.

Thread Context: Non-Blocking

Syntax:

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

Parameters:

Inputs:

messageLength The NWPA passes the value of this parameter, which is the length in bytes of the data being passed between the modules.

message The NWPA passes the value of this parameter, which is a pointer to the buffer containing the data to be passed.

Outputs: None

Return Value: 0 to succeed.

Description: *HAM_Software_Hot_Replace()* is the entry point that **NPA_Exchange_Message()** uses to pass data between a HAM already in server memory and the HAM being newly loaded to replace it. This routine is registered with the NWPA during **NPA_Register_HAM_Module()**. **NPA_Exchange_Message()** is the channel that links the respective hot replace entry points for each HAM. Through this entry point, the replacement HAM can request configuration information from the HAM being replaced so that the new HAM can simply take the old HAM's place and be immediately operational.

Warning:	Since the older version of the HAM will be removed from memory, data-passing between the two modules must be done by handles or indexes to avoid the passing of bad memory pointers.

HAM_Timeout

Purpose: Provides a recovery mechanism from forever-in-error conditions.

Thread Context: Blocking or Non-Blocking (based on **NPA_Spawn_Thread()** setting).

Syntax:

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

Parameters:

Inputs:

parameter Parameter specified in **NPA_Spawn_Thread()** when this routine was scheduled. This parameter is used to assist in the timeout process.

Outputs: None.

Return Value: None

Description: This routine is used as a background error-recovery routine. It gets scheduled for periodic entry as an asynchronous event by calling **NPA_Spawn_Thread()**, and it executes after the elapse of the time interval specified in the *clockTicks* argument passed to **NPA_Spawn_Thread()**. The time interval between iterations is left up to the HAM developer. A routine scheduled with **NPA_Spawn_Thread()** executes its thread only once. Therefore, for periodic execution, this routine must, within its own context, reschedule itself by calling **NPA_Spawn_Thread()**. This routine should be initially scheduled within the context of *HAM_Load()*. If an I/O request or other host adapter action hangs while being processed, the HAM should not indefinitely wait to service it. Doing so could cause a forever-in-error condition from which the HAM cannot recover. The routine provides a rescue mechanism for such a condition by allowing the HAM to regain process control if the I/O is not completed in the allotted time specified in the **TimeoutAmount** field of the HACB. The routine must be able to access a list of all HACB requests that are currently being processed by devices supported by the HAM. This list must be updated each time a device completes a HACB request and accepts a new one. Each time it is executed, This routine should scan the HACB list and decrement the time in each HACB's **TimeoutAmount** field by the value specified in *clockTicks*. If a HACB's **TimeoutAmount** value reaches zero, this routine should:

1. Unlink the HACB from the list.
2. Place the timeout error code (0x0002) in its **hacbCompletion** field.
3. Call **HAI_Complete_HACB()** on the HACB.

Warning:	*HAM_Timeout()* needs to check the timeout granularity set in the HACB's **ControlFlags** field (bit 3). From this check *HAM_Timeout()* can determine the HACB's timeout unit of measure before blindly decrementing the value in **TimeoutAmount**. A unit conversion may be necessary to make the units of *ClockTicks* compatible with the units of **TimeoutAmount**.

HAM_Unload

Purpose: The HAM's last unload-time entry point that prepares the HAM for unloading and returns resources back to the system.

Thread Context: Blocking

Syntax:
```
LONG HAM_Unload (void);
```

Parameters: None.

Return Value: 0 to succeed.

Description: *HAM_Unload()* is the HAM's entry point from the OS when it receives an UNLOAD command for this HAM. *HAM_Unload()* is then responsible for releasing all appropriate resources and unregistering the HAM. Upon entry into this routine the HAM needs to stay operational until all its pending I/O is flushed and the NWPA quiesces any incoming I/O. To do this, the first call the HAM should make within *HAM_Unload()* is to **HAI_Deactivate_Bus()**. It is during the context of **HAI_Deactivate_Bus()** that the NPA actually flushes pending I/O and quiesces new I/O for this HAM. Upon return from **HAI_Deactivate_Bus()**, the HAM is guaranteed not to have any pending I/O.

HAM_Unload_Check

Purpose: The HAM's first unload-time entry point that checks to see which devices are currently bound to the HAM. This entry point is called by the OS prior to calling *HAM_Unload()*.

Thread Context: Non-Blocking

Syntax:

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

Parameters:

Inputs:

screenID The OS passes the value of this parameter, which is a handle to the console. The **NPA_** routines that output messages to the console require this handle as an argument.

Outputs: None

Return Value: 0 if no devices are locked, meaning that the HAM can be cleanly unloaded.

Non-zero if one or more devices are currently being used by an application.

Description: *HAM_Unload_Check()* is called when the OS receives a request from the console to unload the HAM . This routine is responsible for checking to see if any of the HAM's devices are currently being used (locked) by an application. *HAM_Unload_Check()* checks the lock status for a particular adapter by calling **NPA_Unload_Module_Check()**.

The OS looks at the return value to determine if the HAM can be cleanly unloaded, meaning that there are no current I/O processes on the devices controlled by this HAM. 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.

In*x*

Purpose: Takes a bus identifier and an I/O address in that bus's I/O address space and performs whatever operations are necessary to acquire and return the requested data.

Thread Context: Non-Blocking

Syntax:

```
BYTE In8 (
       LONG busTag,
       void *ioAddr );

WORD In16 (
       LONG busTag,
       void *ioAddr );

LONG In32 (
       LONG busTag,
       void *ioAddr );
```

Parameters:

Inputs:

busTag An architecture dependent value returned by **NPAB_Get_Bus_Tag()**. This value specifies the bus on which the operation is to be performed.

ioAddr The I/O address in the bus architecture of the adapter from where the input is to occur.

Outputs: None

Return Value: An unsigned value of the size and data type defined by each respective routine.

Description: These routines are only used by HAMs written for adapters intended for bus architectures that have an I/O address space. The HAM is expected to use the routine appropriate to the data width of the port from where the input is to occur.

The value of *ioAddr* should be the port address the HAM would normally expect for the given bus architecture. For example, if an ISA card with a base port address of 300h is placed on an EISA bus, the HAM will set *ioAddr* to 300h when it wants to input from that base port.

InBuff*x*

Purpose: Takes a bus identifier, an I/O address in that bus's I/O address space, a destination buffer in the CPU's logical address space, and a count of transfer data units to perform whatever operations are necessary to acquire and return the requested number of data units into the destination buffer.

Thread Context: Non-Blocking

Syntax:

```
LONG InBuff8 (
       BYTE *buffer,
       LONG busTag,
       void *ioAddr,
       LONG count );

LONG InBuff16 (
       BYTE *buffer,
       LONG busTag,
       void *ioAddr,
       LONG count );

LONG InBuff32 (
       BYTE *buffer,
       LONG busTag,
       void *ioAddr,
       LONG count );
```

Parameters:

Inputs:

buffer The logical memory address of the destination buffer. This address is in the CPU's logical address space.

busTag An architecture dependent value returned by **NPAB_Get_Bus_Tag()**. This value specifies the bus on which the operation is to be performed.

ioAddr The I/O address in the bus architecture of the adapter from where the input is to occur.

count The number of transfer units in the specified data size.

Outputs: None

Return Value:

0 - The requested operation was completed successfully.
1 - Memory protection prevented by the completion of the requested operation.
3 - Memory error occurred while attempting to perform the requested operation.
4 - One of the parameters was invalid.
5 - The requested operation could not be completed.

Description: These routines are only used by HAMs written for adapters intended for bus architectures that have an I/O address space. The HAM is expected to use the routine appropriate to the data width of the port from where the input is to occur. A buffer is filled with data from the specified I/O address with the number of data units specified (*count*). The buffer address will fill forward.

The value of *ioAddr* should be the port address the HAM would normally expect for the given bus architecture. For example, if an ISA card with a base port address of 300h is placed on an EISA bus, the HAM will set *ioAddr* to 300h when it wants to input from that base port.

NPA_Add_Option

Purpose: Specifies command line options and configuration information that can be parsed out and registered for this instance of the application.

Architecture Type: All

Requirements: The **NPAOptionStruct** must be initialized before calling this routine.

Thread Context: Non-Blocking

Syntax:

```
LONG NPA_Add_Option(
       LONG npaHandle,
       struct NPAOptionStruct *option);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module()** or **NPA_Register_HAM_Module()**, respectively.

option Pointer to the **NPAOptionStruct** associated with this CDM / HAM. The **NPAOptionStruct** contains information about hardware options associated with this CDM / HAM. The following is the ANSI C definition of the structure:

```
struct NPAOptionStruct{
       BYTE name[32];
       LONG parameter0;
       LONG parameter1;
       LONG parameter2;
       WORD type;
       WORD flags;
       BYTE string[n];
} ;
```

Outputs: None

Return Value: 0 if successful, Non-zero if unsuccessful.

Description: **NPA_Add_Option()** is used to query the systems operator for command line parameters that will be used by the CDM/HAM. The command line parameters identify configuration information and reserve hardware resources needed by a CDM / HAM. **Option* is an optional parameter for CDMs since they do not generally require the reservation of hardware resources or command line configuration information. Therefore, in the case of a CDM, the value of **Option* can be set to zero. However, for HAMs this is the mechanism for setting port, interrupt, DMA channel, and memory decode options. A description of the **NPAOptionStruct** can be found in Chapter 6.

NPA_Allocate_Memory

Purpose: Allocates a block of system memory for local use of the module. The memory block is returnable to the system.

Architecture Type: All

Thread Context: Non-Blocking/Blocking (See *flag* below for details)

Requirements: This routine cannot be called at interrupt level. If *flag* is set to Blocking, this routine must be called in a blocking context.

Syntax:

```
LONG NPA_Allocate_Memory(
        LONG npaHandle,
        void **virtualPointer,
        void **physicalPointer,
        LONG bufferSize,
        LONG flag,
        LONG *sleptFlag);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module**() or **NPA_Register_HAM_Module**(), respectively.

virtualPointer Address of pointer to memory storage location of the desired data type.

physicalPointer Address of pointer to memory storage location of the desired data type.

bufferSize Size, in bytes, of the memory block being requested.

flag Indicator telling the NWPA the type of allocation being requested. Knowing the allocation type allows the NWPA to track the memory resource. This parameter can have one of the following values:

0x00000000	Indicates a normal memory request.
0x00000001	Indicates a request for I/O contiguous memory.
0x00000002	Indicates a request for memory below 16 MB (supporting adapters using DMA).
0x00000004	Selects the Blocking version of this routine. If this flag is set, this routine may sleep (block) to allow a single additional attempt to allocate the requested memory. If it was required to sleep to allocate the memory, the *sleptFlag* parameter will be non-zero. Use of this flag requires the call to be made in a blocking context. If this flag is set on a 3.12 NetWare Server, it will default to a normal

memory request (0x00000000) and *sleptFlag* will be ignored.

sleptFlag A pointer to where the Sleep indicator is to be placed. If *flag* is not set to Blocking, this parameter is not used and should be set to zero.

Outputs:

virtualPointer Receives the starting virtual address of the allocated memory block from the OS.

physicalPointer Receives the starting physical (absolute) address of the allocated memory block from the OS.

sleptFlag This parameter is only used if *Flag* is set to Blocking. A non-zero value indicates that the routine went to sleep to complete the allocation request.

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_Allocate_Memory()** is used to allocate system memory required by a CDM / HAM, such as special data structures or buffers. This allocation will be on paragraph (16-byte) boundaries. The CDM / HAM must provide the storage locations (*virtualPointer* and *physicalPointer*) for the outputs it receives during this call. **NPA_Allocate_Memory()** is passed the two pointer-to-pointer variables and a buffer size. **NPA_Allocate_Memory()** allocates a memory block of the requested size and assigns its starting virtual address to one of the pointer-to-pointer variables and assigns its starting physical address to the other variable. The virtual address is the logical NetWare address of the allocated memory block. The physical address is the absolute hardware address of the allocated memory block, and it is provided to support adapters using DMA. The memory allocated by this routine is not initialized to any value, it is raw memory. The CDM / HAM is responsible for initializing allocated memory. Additionally, this routine may be called during the context of any process, except a process within an interrupt level. Memory should not be allocated at the interrupt level. Memory is returned to the system pool using **NPA_Return_Memory()**.

Note: If the CDM allocates a memory buffer that will be accessed by a HAM, it must allocate the memory as an I/O buffer.

NPA_Cancel_Thread

Purpose: Cancels asynchronous blocking or non-blocking threads of execution scheduled for an NWPA application.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: Interrupts must be disabled.

Syntax:

```
LONG NPA_Cancel_Thread(
      LONG npaHandle,
      LONG (*ExecuteRoutine)(),
      LONG parameter);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs. Its value was assigned during **NPA_Register_CDM_Module()** or **NPA_Register_HAM_Module()**, respectively.

ExecuteRoutine Pointer to the CDM / HAM routine that was originally passed into **NPA_Spawn_Thread()** when the thread was originally spawned.

parameter Parameter value that was originally passed into **NPA_Spawn_Thread()** when the thread was originally spawned.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful, meaning that the spawned thread has already begun execution.

Description: **NPA_Cancel_Spawn_Thread()** is used to cancel an instance of an asynchronous thread that was spawned using **NPA_Spawn_Thread()**. The NWPA uses the input parameters, *ExecuteRoutine* and *Parameter*, to identify the thread to cancel; therefore, these two parameters must match exactly with the parameters passed to **NPA_Spawn_Thread()**. A return value of zero indicates that the spawned thread was successfully cancelled. A non-zero return value indicates that the spawned thread could not be cancelled because it is currently running. A CDM/HAM must make a separate call for each spawned thread it wishes to cancel. Additionally, a CDM/HAM must call this routine for all pending threads that it spawned before it can unload.

NPA_CDM_Passthru

Purpose: Sends a CDM Message to a device in order to receive status or diagnostic information about the device. It is used for vendor specific commands.

Architecture Type: All

Thread Context: Blocking

Requirements: None.

Syntax:

```
LONG NPA_CDM_Passthru(
      LONG  *appReturnCode,
      LONG  mmDeviceID,
      LONG  function,
      LONG  vendorID,
      LONG  parameter1,
      LONG  parameter2,
      LONG  parameter3,
      LONG  BufferLength,
      void  *Buffer);
```

Parameters:

Inputs:

mmDeviceID Media Manager object ID for the device. See the **Media Manager Functional Specification and Developer's Guide** for details on how to obtain this ID.

function Must be either 0x1E or 0x3E for the PassThru function.

vendorID Novell assigned vendor ID. This is used to confirm compatibility between vendor-specific applications and vendor-specific CDMs. Must be 0x100 or greater.

parameter1 Vendor specific.

parameter2 Vendor specific.

parameter3 Vendor specific.

bufferLength Length of the buffer in bytes.

buffer Address of buffer passed to the CDM to send or receive data.

Outputs:

appReturnCode Value returned by the managing CDM during **CDI_Complete_Message()**. It can be any LONG value understood by both the application and the custom CDM

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_CDM_Passthru()** is used to send vendor specific requests to the managing CDM of a device. This command sends a request, then returns when the request is complete. The CDM must check the *VendorID* to verify that the request came from an appropriate application. The CDM must also register the acceptance of these passthru requests by setting bit 0x40000000 in the function mask for 0x3E support and/or control mask for 0x1E support (reference **CDI_Object_Update()**). The CDM must understand the parameters being sent, and take the necessary action including HACBs to the device if needed.

NPA_Delay_Thread

Purpose: Delays the current process for a specified number of clock ticks.

Architecture Type: All

Thread Context: Blocking

Requirements: This routine must be called only from a blocking process level.

Syntax:

```
LONG NPA_Delay_Thread(
       LONG npaHandle,
       LONG clockTicks);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs. Its value was assigned during **NPA_Register_CDM_Module()** or **NPA_Register_HAM_Module()**, respectively.

clockTicks Value specifying the time in clock ticks to let this process sleep. A clock tick translates to 1/18th of a second (55ms).

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_Delay_Thread()** is used to cause the current process to sleep for the number of clock ticks specified in the *clockTicks* parameter. During its sleep period, the process temporarily yields its thread. The purpose of this routine is to prevent a blocking process--that will not complete for at least a specified time period--from dominating vital resources and blocking other vital NWPA processes. After the specified time elapses, the thread is returned, and the process continues from the point after it called **NPA_Delay_Thread()**.

NPA_Exchange_Message

Purpose: Provides a communication link between two different versions of a HAM in order to facilitate software hot replacement.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPA_Exchange_Message(
       LONG npaHandle,
       LONG messageLength,
       void *message);
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

messageLength Size, in bytes, of the message (data buffer) being passed or received.

message Pointer to the message (data buffer) being passed or received.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_Exchange_Message()** is used to exchange I/O configuration information between an older version of a module and the updated version that will replace it by calling the new *HAM_Software_Hot_Replace()* routine. Calling **NPA_Exchange_Message()** is dependent on the return value of **NPA_Register_HAM_Module()**. If the return value is zero, it indicates that the load event is either an initial load of the module or a new instance of the module. If the return value is one, it indicates that the module currently being loaded should hot replace the already loaded module having the same *NovellAssignedModuleID* value. In the case where the return value equals zero, normal initialization and registration should take place excluding a call to **NPA_Exchange_Message()**. In the case where the return value equals one, **NPA_Exchange_Message()** needs to be called in order for the two modules to communicate with each other.
NPA_Exchange_Message() should be called within the context of *HAM_Load()*.

NPA_Get_Version_Number

Purpose: Provides the revision level of the current NWPA version.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:
```
LONG NPA_Get_Version_Number(LONG *revisionNumber);
```

Parameters:

Inputs: None

Outputs:

revisionNumber The NWPA version number currently running. The return value is in the format 00XXYYZZ, where XX is the major revision level, YY is the minor revision level, and ZZ is the sub-minor revision level (interpreted as a letter with 01=A and 26=Z). Example: a value of 00022002 would mean NWPA Version 2.20B.

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_Get_Version_Number()** allows developers to have access to the current version number of the NWPA that is running. This number may be used to maintain version and feature compatibility on a server between HAMs, CDMs and the NWPA.

NPA_HACB_Passthru

Purpose: Sends a HACB message to a device in order to receive status or diagnostic information about the device. It is used for vendor-specific commands.

Architecture Type: All

Thread Context: Blocking

Requirements: None.

Syntax:

```
LONG NPA_HACB_Passthru(
      LONG mmAdapterID,
      struct HACBDef *HACB);
```

Parameters:

Inputs:

mmAdapterID The Media Manager object ID for the adapter.This can be obtained by using **MM_Find_Object_Type(0, &id)** to get the ID, then **MM_Return_Generic_Info()** to get the name of the HAM, and verify it is the correct one. See the **Media Manager Functional Specification and Developer's Guide** for details.

HACB Address of a HACB to be sent.

Outputs: None

Return Value: 0 if successful, Non-zero if unsuccessful.

Description: **NPA_HACB_Passthru()** is used to send vendor specific requests directly to a HAM. The request is sent, and the call returns after the request is complete. The application must have an understanding of the device and handle any errors that occur as a result of the requests. The application must make sure that the HAM is returned to its original condition (i.e. queue frozen or unfrozen) when finished with the requests. The requests can be **HACBType**=0 Functions 0-3, or **HACBType**=0x100 or greater. When non-HACBType 0 requests are sent, the **HACBType** must be the Novell assigned vendor ID. The HAM must check this field and report an Unsupported Interface Type (0x00030044) error if the vendor id is not supported. The HAM must otherwise service the request and send the appropriate command to the device as needed. The Command Block Overlay Area can be used as needed for the request. It is important to remember that the data in this overlay area goes to the HAM only. This data, if changed by the HAM may not be seen by the application upon return. All data passed from the HAM to the application must go through the buffer addressed by *vDataBufferPtr*.

NPA_Interrupt_Control

Purpose: Performs interrupt masking capabilities on the default (primary) system I/O bus.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG NPA_Interrupt_Control(
       LONG npaHandle,
       LONG irqLevel,
       LONG flag);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module**() or **NPA_Register_CDM_Module**(), respectively.

irqLevel Value indicating the interrupt level on which to take action.

flag Value indicating the type of action to perform. This parameter can have one of the following values:

0x00000000	Enable interrupts (This will unmask the IRQ level)
0x00000001	Disable interrupts (This will mask the IRQ level)
0x00000002	Check the hardware interrupt.

Outputs: None

Return Value:

Input Flag Value	Return Value
0x00000000	0 if enabling interrupts was successful. Non-zero if enabling interrupts was unsuccessful.
0x00000001	0 if disabling interrupts was successful. Non-zero if disabling interrupts was unsuccessful.
0x00000002	0 if the interrupts at the specified level are disabled. Non-zero if the interrupts at the specified level are enabled.

Description: **NPA_Interrupt_Control**() is used to either unmask an interrupt, mask an interrupt, or check the current masking of an interrupt at the specified level on the default system I/O bus. The action to be performed is determined by the value of the *flag* parameter passed into **NPA_Interrupt_Control**() as discussed above. Implementation of this routine involves the setting or testing of bits in the hardware's interrupt mask register.

NPA_Micro_Delay

Purpose: Delays a set number of microseconds for use in allowing for interface delays etc.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: Maximum *count* of 10,000 microseconds (10 milliseconds)

Syntax: `LONG NPA_Micro_Delay(LONG count);`

Parameters:

Inputs:

count The number (between 0 and 10,000) of microseconds to delay.

Outputs: None

Return Value: 0 if successful.
Non-zero if *count* was not a valid number

Description: **NPA_Micro_Delay**() is used to delay for a short amount of time while allowing an interface state to change, etc. The thread will not be switched, and the interrupt state will not change. This call can be made during interrupt service routines (ISRs); however, it is recommended it be used sparingly within ISRs so that interrupts are not disabled for extensive periods of time.

Note: The resolution of this timer is approximately 10 microseconds.

NPA_Parse_Options

Purpose: Parses the command line at LOAD time for configuration options.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: This routine must be called from a blocking process level. When used correctly, this routine is called within the context of the module's initialization routine (*CDM/HAM_Load()*), which is a blocking process.

Syntax:

```
LONG NPA_Parse_Options(
        LONG npaHandle,
        LONG screenID,
        BYTE *commandLine);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module()** or **NPA_Register_CDM_Module()**, respectively.

screenID Handle to the server console that was passed into the *CDM/ HAM_Load()* routine.

commandLine Pointer to the characters entered on the command line at load time. Its value was passed into the *CDM/HAM_Load()* routine.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_Parse_Options()** is used to parse the command line parameters specified by the systems operator. Once the command line is parsed, **NPA_Parse_Options()** calls *CDM/HAM_Check_Option()* so that the CDM / HAM can validate the command line options and set its I/O configuration.

NPA_Register_CDM_Module

Purpose: Registers a CDM with the NWPA.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: This routine is the first API called during *CDM_Load()*. Additionally, the module must provide the storage locations for the outputs it receives during this call.

Syntax:

```
LONG NPA_Register_CDM_Module(
       LONG *npaHandle,
       LONG NovellAssignedModuleID,
       LONG loadHandle,
       LONG (*CDM_Check_Option)(),
       LONG (*CDM_Execute_CDMMessage)(),
       LONG (*CDM_Inquiry)(),
       LONG instance);
```

Parameters:

Inputs:

npaHandle Address of a local variable of type LONG.

NovellAssignedModuleID The CDM vendor ID assigned by Novell Labs. This parameter is a unique ID associating a CDM with its manufacturer. Every CDM must have its own unique ID.

loadHandle Handle that the OS assigned to the CDM at load time. The value for this parameter was passed into the CDM's load-time entry point, *CDM_Load()*.

CDM_Check_Option Pointer to the *CDM_Check_Option()* entry point called during the parsing of load-time command line options and again at option registration.

> **Note:** For a CDM, command line options should only indicate operational modes for the software module. They must not indicate hardware options such as interrupts, ports, DMA channels, etc. If the CDM does not support command line options, this parameter should be set to zero.

CDM_Execute_CDMMessage Pointer to the *CDM_Execute_CDMMessage()* routine, which is the CDM's main entry point for receiving and routing CDM Messages.

CDM_Inquiry Pointer to the *CDM_Inquiry()* entry point, which is the CDM's routine for checking device information and determining whether or not to bind to a device.

instance A CDM-generated number identifying a device instance. The NWPA will use this number to associate different groups of options with a particular device being managed by the CDM.

> **Note:** If the CDM does not support command line options, this parameter should be set to zero.

Outputs:

npaHandle Receives a unique NWPA handle for the CDM module. This handle is a tag the NWPA uses to track the CDM module, and it is a required argument for using the **NPA_** APIs.

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_Register_CDM_Module()** is used to register the CDM module with the NPA, along with the application's entry points. This routine should be the first API called during the module's load-time entry point, *CDM_Load()*. It is during the context of this API that the CDM receives its unique NWPA handle. This handle is a necessary argument for using the other **NPA_** APIs that provide system resources to the module.

NPA_Register_For_Event_Notification

Purpose: Registers a procedure to be called prior to specific system events.

Architecture Type: All

Thread Context: Blocking

Requirements: Must be called only from a blocking process level.

Syntax:

```
LONG NPA_Register_For_Event_Notification(
  LONG npaHandle,
  LONG *eventHandle,
  LONG eventType,
  LONG priority,
  LONG (*WarnRoutine)(
        void (*OutputRoutine)(void *ControlString, ...),
        LONG parameter),
  void (*ReportRoutine)(
        LONG parameter));
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module**() or **NPA_Register_CDM_Module**(), respectively.

eventType Indicates the type of event for which the caller wishes notification. The following describes events for which notification may be received, the type of notification that can be made (Warn, Report, or both), the thread context of the notification call (blocking or non-blocking), and the defined use of the input parameter (*parameter*) passed to the notification call (*WarnRoutine()* or *ReportRoutine()*).

Type Definition	Type Number (In Decimal)
EVENT_VOL_SYS_MOUNT The input parameter is undefined. The Report Routine is called immediately after vol SYS is mounted. The Report Routine may block the thread.	0
EVENT_VOL_SYS_DISMOUNT The input parameter is undefined. Both the Warn and Report routines are called before vol SYS is dismounted. The Report Routine may block the thread.	1
EVENT_ANY_VOL_MOUNT The input parameter is the volume number. The Report routine is called immediately after any volume is mounted. The Report Routine may block the thread.	2

EVENT_ANY_VOL_DISMOUNT 3
The input parameter is the volume number.
Both the Warn and Report routines are called before any volume is dismounted.
The Report Routine may block the thread.

EVENT_DOWN_SERVER 4
The input parameter is undefined.
Both the Warn and Report routines are called before the server is shut down.
The Report Routine may block the thread.

EVENT_CHANGE_TO_REAL_MODE 5
The input parameter is undefined.
The Report routine is called before the server changes to real mode.
The Report Routine may not block the thread.

EVENT_RETURN_FROM_REAL_MODE 6
The input parameter is undefined.
The Report routine is called after the server has returned from real mode.
The Report Routine may not block the thread.

EVENT_EXIT_TO_DOS 7
The input parameter is undefined.
The Report routine is called before the server exits to DOS.
The Report Routine may block the thread.

EVENT_MODULE_UNLOAD 8
The input parameter is the module handle.
Both the Warn and Report routines are called before a module is unloaded from the console command line. Only the Report Routine is called when a module unloads itself. The Report Routine may block the thread.

EVENT_ACTIVATE_SCREEN 14
The input parameter is the Screen ID.
The Report routine is called after the screen becomes the active screen.
The Report Routine may block the thread.

EVENT_UPDATE_SCREEN 15
The input parameter is the Screen ID.
The Report routine is called after a change is made to the screen image.
The Report Routine may block the thread.

EVENT_UPDATE_CURSOR 16
The input parameter is the Screen ID.
The Report routine is called after a change to the cursor position or state occurs.
The Report Routine may not block the thread.

EVENT_KEY_WAS_PRESSED 17
The input parameter is undefined.
The Report routine is called after any key on the keyboard is pressed (including shift/alt/control).
The Report Routine is called at interrupt time, it may not block the thread.

EVENT_DEACTIVATE_SCREEN 18
The input parameter is the Screen ID.
The Report routine is called after the screen becomes inactive.
The Report Routine may not block the thread.

EVENT_OPEN_SCREEN 20
The input parameter is the Screen ID for the newly created screen.
The Report routine is called after the screen is created.
The Report Routine may block the thread.

EVENT_CLOSE_SCREEN 21
The input parameter is the Screen ID for the screen being closed.
The Report routine is called before the screen is closed.
The Report Routine may block the thread.

EVENT_MODULE_LOAD 27
The input parameter is the module handle.
The Report routine is called after the module has been loaded.
The Report Routine may block the thread.

EVENT_GENERIC 32

priority

The priority used to call this notification routine. Priorities are defined as follows:

Priority Definition	Priority Number (in Decimal)
EVENT_PRIORITY_OS	0
EVENT_PRIORITY_APPLICATION	20
EVENT_PRIORITY_DEVICE	40

WarnRoutine

A pointer to a routine that is called when the OS makes an EventCheck call. If the warn routine does not want the event to occur, it must output a message and then return a non-zero value. Most event notification routines are called at process level, but some are made at interrupt level (meaning the thread may not be blocked). The above table of event types specifies which events must be checked to determine if the event allows its thread to be blocked.

ReportRoutine

A pointer to a routine that is called when the OS makes an EventReport call. Most event notification routines are called at process level, but some are made at interrupt level (meaning the thread may not be blocked). The above table of event types specifies which events must be checked to determine if the event allows its thread to be blocked.

Receives a 32-bit handle to the registered event. This event handle is passed as an input parameter to **NPA_Unregister_Event_Notification()**.

Outputs:
eventHandle

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: On some occasions a driver is required to perform some action prior to the OS terminating, switching to real mode, exiting to DOS, etc. The driver should call **NPA_Register_For_Event_Notification()** providing notification procedure pointers as indicated above. Even though the calls to register and unregister the event notification are blocking, the actual call to the event notification procedure provided by the driver is not always made from blocking process level (the environment varies with the particular event being reported). The Warn Routine will be provided with two parameters when called. The first is the Output Routine which must be used to output messages (the Output Routine must be called with a control string and as many parameters as the control string indicates), and the second is the parameter described in each of the event types above. When the Report routine is called it is passed a single parameter. This is the same parameter described in each of the event types described above.

NPA_Register_HAM_Module

Purpose: Registers a HAM with the NWPA.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: This routine is the first API called during *HAM_Load()*. Additionally, the module must provide the storage locations for the outputs it receives during this call.

Syntax:

```
LONG NPA_Register_HAM_Module(
        LONG *npaHandle,
        LONG NovellAssignedModuleID,
        LONG loadHandle,
        LONG (*HAM_Check_Option)(),
        LONG (*HAM_Software_Hot_Replace)(),
        LONG (*HAM_ISR)(),
        LONG (*HAM_Execute_HACB)(),
        LONG (*HAM_Abort_HACB)()
        LONG instance);
```

Parameters:

Inputs:

npaHandle Address of a local variable of type LONG.

NovellAssignedModuleID The HAM vendor ID assigned by Novell Labs. This parameter is a unique ID associating a HAM with its manufacturer. Every HAM must have its own unique ID.

loadHandle Handle that the OS assigned to the HAM at load time. The value for this parameter was passed into the HAM's load-time entry point, *HAM_Load()*.

HAM_Check_Option Pointer to the *HAM_Check_Option()* entry point called during the parsing of load-time command line options and again at option registration.

HAM_Software_Hot_Replace Pointer to the *HAM_Software_Hot_Replace()* entry point used in dynamically updating versions of a HAM.

> **Note:** Hot replacement is an optional feature for a HAM. If the HAM does not support hot replacement, it should set this parameter to zero. Doing so will force the NWPA to never allow hot replacement of this HAM.

HAM_ISR Pointer to the *HAM_ISR()* routine, which is the HAM's Interrupt Service Routine (ISR).

HAM_Execute_HACB Pointer to the *HAM_Execute_HACB()* routine, which is the HAM's main entry point for receiving HACB I/O requests.

HAM_Abort_HACB Pointer to the *HAM_Abort_HACB()* routine, which is the HAM's main entry point for receiving aborts on HACB I/O requests.

instance A HAM-generated number identifying an adapter card instance. The NWPA will use this number to associate different groups of registered hardware options with a particular adapter card being managed by the HAM.

Outputs:

npaHandle Receives a unique NWPA handle for the HAM module. This handle is a tag the NWPA uses to track the HAM module, and it is a required argument for using the **NPA_** APIs.

> **Note:** The NWPA recognizes reentrant modules, meaning that a single code image of the HAM will manage multiple adapters. Therefore, if a reentrant HAM calls **NPA_Register_HAM_Module()** again to assign a new instance number to the new adapter card instance, the NWPA will ensure that the value output to this variable is the same for each call.

Return Value: 0 if successful and not a hot replace case. 1 if successful and hot replace case. Other non-zero value if unsuccessful.

Description: **NPA_Register_HAM_Module()** is used to register the HAM module with the NWPA, along with the application's entry points. This routine should be the first API called during the module's load-time entry point, *HAM_Load()*. It is during the context of this API that the HAM receives its unique NWPA handle. This handle is a necessary argument for using the other **NPA_** APIs that provide system resources to the module. **NPA_Register_HAM_Module()** also determines if a version of a HAM currently loaded in server memory is to be hot replaced with a newer HAM version. **NPA_Register_HAM_Module()** makes this determination by comparing the *NovellAssignedModuleID* and the *loadHandle* of a newly loaded HAM with other HAMs that are already loaded. If there is a match in *NovellAssignedModuleID* values between the newly loaded HAM and an already loaded HAM, but their respective *loadHandle* values differ, then the NWPA determines that the newly loaded HAM is hot replacing the already loaded HAM.

NPA_Register_Options

Purpose: Registers options that have been parsed out from the command line, and, for those modules that support it, initiates hot replacement.

Architecture Type: All

Thread Context: Blocking

Requirements: This routine must be called only from a blocking process level. When used correctly, this routine is called within the context of the module's initialization routine (*CDM/HAM_Load()*), which is a blocking process.

Syntax:

```
LONG NPA_Register_Options(
        LONG npaHandle,
        LONG instance);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module()** or **NPA_Register_CDM_Module()**, respectively.

instance The instance number the CDM or HAM intends to associate with the current group of options being registered. This instance corresponds to either a CDM's device instance or a HAM's adapter instance. This instance number was what the CDM or HAM passed to **NPA_Register_CDM_Module()** or **NPA_Register_HAM_Module()**, respectively.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_Register_Options()** is used to register the command line options allowed by a CDM/HAM. These options can be custom parameters, or as in the case of the HAM, they may specify the interrupt, port, and DMA range values allowed by the HAM. Command line options may be anything needed by the CDM/HAM in from custom initialization parameters. **NPA_Register_Options()** must be called during CDM/HAM initialization within the context of *CDM/HAM_Load()*. Any data structures required by the module should be allocated prior to making this call, because once this routine returns, the module must be ready to accept I/O requests.

NPA_Return_Bus_Type

Purpose: Returns a bitmap indicating the I/O bus type.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None

Syntax:
```
LONG NPA_Return_Bus_Type (LONG npaHandle);
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs. Its value was assigned during **NPA_Register_HAM_Module()**.

Outputs: None

Return Value: Bitmap defined as follows:

0x00000001	MCA
0x00000002	EISA
0x00000004	PCI
0x00000008	PCMCIA
0x00000010	ISA

Description: **NPA_Return_Bus_Type()** is used to determine the processor bus type, for use by the HAM during its initialization/registration routine (*HAM_Load()*). This routine is only valid when used with machines having an Intel based architecture.

NPA_Return_Memory

Purpose: Returns previously allocated memory to the system.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPA_Return_Memory(
       LONG npaHandle,
       void *virtualPointer);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module()** or **NPA_Register_CDM_Module()**, respectively.

virtualPointer Pointer to the logical NetWare address of the memory block being returned to the system. The memory block must have been originally allocated using **NPA_Allocate_Memory()**.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_Return_Memory()** is used to return allocated system memory, such as special data structures or buffers required by a CDM / HAM, back to the system's memory pool. To minimize impacts on overall server performance, CDMs and HAMs are expected to periodically clean up local memory. It is essential that local memory be returned before unloading.
NPA_Return_Memory() may be called during the context of any process, except a process within an interrupt level. Memory should not have been allocated at the interrupt level. This routine is intended to return memory blocks that were allocated using **NPA_Allocate_Memory()**.

NPA_Spawn_Thread

Purpose: Schedules execution of a blocking or non-blocking asynchronous event, or a timer-interrupt-level callback.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPA_Spawn_Thread(
       LONG npaHandle,
       void (*ExecuteRoutine)(),
       LONG parameter,
       LONG clockTicks,
       LONG flag);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module()** or **NPA_Register_CDM_Module()**, respectively.

ExecuteRoutine Pointer to the CDM / HAM routine that is called to execute the spawned thread.

parameter Input parameter required by *ExecuteRoutine*. If *ExecuteRoutine* does not require an input parameter value, set *Parameter* equal to zero.

clockTicks Value specifying the time in clock ticks to elapse before this thread is initiated. A clock tick translates to 1/18th of a second (55ms).

flag Value specifying whether the spawned thread is blocking or non-blocking:

0x00000000	Indicates a non-blocking thread. (Default)
0x00000001	Indicates a blocking thread.
0x00000002	Indicates the thread is scheduled to execute during the timer chip interrupt following the specified tick count.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_Spawn_Thread()** is used to schedule an asynchronous background thread for a CDM / HAM that becomes active after the time specified in the

clockTicks parameter. If the value in *clockTicks* is zero, the thread is immediately scheduled. Whether scheduling is immediate or delayed the thread is initiated by NetWare calling the entry point whose address was passed into **NPA_Spawn_Thread()** as an argument. **NPA_Spawn_Thread()** can be used to set up an entry point for a background timer or to create a designated gremlin process that can run throughout the time that the CDM / HAM is loaded in file server memory. An example of a gremlin process is the HAM's timeout handler that monitors the allowable execution time of an I/O request specified in the **TimeoutAmount** field of a HACB. If the value of the *flag* parameter is zero, **NPA_Spawn_Thread()** schedules a non-blocking thread. If the spawned thread is non-blocking, no blocking calls can be issued during its context. On the other hand, if the value of the *flag* parameter is one, **NPA_Spawn_Thread()** schedules a blocking thread from which other blocking calls can be made. However, as much as possible, blocking calls should be kept to a minimum to avoid impact on server performance.

In the case where *flag* equals 2 (timer interrupt time callback), the execute routine must adhere to interrupt level constraints. In addition, if NetWare is running in a non-dedicated environment (such as NetWare for OS/2 or NetWare for Windows) the execute routine must be concerned about the watchdog timer, which could result in a system NMI causing ill effects. It is suggested that an interrupt time callback keep its execution time under 20 milliseconds.

Note: **NPA_Spawn_Thread()** is a one-shot thread. In order to reschedule an asynchronous thread for execution, **NPA_Spawn_Thread()** must be called again.

NPA_System_Alert

Purpose: Allows a CDM or HAM to queue alert messages to the console screen and notify the system of hardware or software problems during threads where the driver does not have access to the console's screen handle.

Architecture Type: All

Thread Context: Non-Blocking

Syntax:

```
LONG NPA_System_Alert(
        LONG npaHandle,
        BYTE *controlString,
        LONG alertMask,
        LONG targetNotifyMask,
        LONG alertID,
        LONG alertClass,
        LONG alertSeverity,
        LONG paramCount,
                args...);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module()** or **NPA_Register_CDM_Module()**, respectively.

controlString Pointer to a null-terminated control string similar to that used in the C sprintf() function, including embedded returns, line-feeds, tabs, bells, and % specifiers (except floating-point specifiers).

alertMask A bit-mask indicating how the alert gets posted. Valid values are:

QUEUE_THIS_ALERT_MASK	0x00000001
ALERTID_VALID_MASK	0x00000002
ALERT_LOCUS_VALID_MASK	0x00000004
ALERT_EVENT_NOTIFY_ONLY_MASK	0x00000008
ALERT_NO_EVENT_NOTIFY_MASK	0x00000010

This field is usually set to QUEUE_THIS_ALERT_MASK .

targetNotifyMask A bit-mask identifying the destination of the notification:

NOTIFY_CONNECTION_BIT	0x00000001
NOTIFY_EVERYONE_BIT	0x00000002
NOTIFY_ERROR_LOG_BIT	0x00000004
NOTIFY_CONSOLE_BIT	0x00000008

This field is usually set to NOTIFY_CONSOLE_BIT.

alertID Provides error code for system log:

OK	0x00000000
ERR_HARD_FAILURE	0x000000FF

alertClass Indicates the class of the error:

CLASS_UNKNOWN	0x00000000
CLASS_TEMP_SITUATION	0x00000002
CLASS_HARDWARE_ERROR	0x00000005
CLASS_BAD_FORMAT	0x00000009
CLASS_MEDIA_FAILURE	0x00000011
CLASS_CONFIGURATION_ERROR	0x00000015
CLASS_DISK_INFORMATION	0x00000018

alertSeverity Indicates the severity of the error:

SEVERITY_INFORMATIONAL	0x00000000
SEVERITY_WARNING	0x00000001
SEVERITY_RECOVERABLE	0x00000002
SEVERITY_CRITICAL	0x00000003
SEVERITY_FATAL	0x00000004
SEVERITY_OPERATION_ABORTED	0x00000005

paramCount The number of additional arguments being passed in the args input parameter. If no arguments are to be passed, set this parameter to zero.

Note: This routine accepts up to four additional arguments.

args Additional arguments corresponding to the % specifiers contained in the *ControlString* input parameter. If no % specifiers are contained in *ControlString*, then this parameter does not need to be used.

Outputs: None

Return Value: 0 if successful.
-1 if the NWPA object for the calling CDM/HAM cannot be found.
-2 if *paramCount* is out of range (exceeds 4).

Description: The main purpose of **NPA_System_Alert()** is to give CDMs and HAMs a method of issuing alert messages to the console screen without having to provide the console's screen handle. The only time that a CDM or HAM has access to a valid console screen handle is during its load-time initialization and unload routines. The handles passed to these two routines should not be saved. They are only valid during the context of the respective routines. By using **NPA_System_Alert()**, CDMs and HAMs alleviate cursor and negotiation conflicts with other NLMs that may try to access the console screen.

NPA_Unload_Module_Check

Purpose: Determines if a module can be cleanly unloaded meaning that no applications are currently using devices it controls.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPA_Unload_Module_Check(
       LONG npaHandle,
       LONG NovellAssignedModuleID,
       LONG screenID);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's module handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module()** or **NPA_Register_HAM_Module()**, respectively.

NovellAssignedModuleID The vendor ID assigned by Novell Labs. This parameter is the unique ID associating a module with its manufacturer.

screenID ID to the server console. Its value was passed to the HAM through *HAM_Unload_Check()*.

Outputs: None

Return Value: 0 if no devices are locked.
Non-zero if one or more devices are locked by an application.

Description: **NPA_Unload_Module_Check()** is used to determine if an application is currently using any devices controlled by the module. A CDM or HAM should call this API within the context of their respective unload-time entry points, *CDM_Unload_Check()* and *HAM_Unload_Check()* respectively. The OS will call these entry points when the UNLOAD command is issued on the CDM / HAM from the command line. The purpose of **NPA_Unload_Module_Check()** is to determine if the module can be cleanly unloaded without losing any current I/O processes.

NPA_Unregister_Event_Notification

Purpose: Unregisters a notification procedure previously registered with **NPA_Register_For_Event_Notification()**.

Architecture Type: All

Thread Context: Blocking

Requirements: Must be called only from a blocking process level.

Syntax:
```
LONG NPA_Unregister_Event_Notification(LONG eventHandle);
```

Parameters:

Inputs:

eventHandle 32-bit value identifying the notification procedure to be unregistered. This value was output by **NPA_Register_For_Event_Notification()** when the notification procedure was registered.

Outputs: None

Return Value: 0 if unregistering the notification procedure was successful.
-1 *eventHandle* was an invalid parameter.

Description: **NPA_Unregister_Event_Notification()** removes the notification procedure specified in *eventHandle* from a list of procedures scheduled to be called by the Media Manager prior to (or following) specific system events. The notification-procedure identifier, *eventHandle*, is an output parameter of **NPA_Register_For_Event_Notification()**, which is the routine used to register the procedure. If a notification procedure was registered, then it must be unregistered prior to the driver being unloaded.

NPA_Unregister_Module

Purpose: Unregisters an NWPA application with the NWPA.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: This routine should only be called with intent to remove the module's code image from server memory.

Syntax:

```
LONG NPA_Unregister_Module(
       LONG npaHandle,
       LONG NovellAssignedModuleID);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module()** or **NPA_Register_CDM_Module(),** respectively.

NovellAssignedModuleID The CDM/HAM vendor ID assigned by Novell, Inc.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_Unregister_Module()** is used to unregister the CDM / HAM, along with its respective entry points, from the NWPA. The intent of this routine is prepare the module for having its code image removed from server memory. **NPA_Unregister_Module()** should be called within the context of the module's exit routine (*CDM/HAM_Unload()*).

Warning:	**NPA_Unregister_Module()** should not be used to exit a single instance of a reentrant module. Doing so will crash the other instances that are still running.

NPA_Unregister_Options

Purpose: Unregisters the configuration options associated with an NWPA application.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: To unregister options on an instance basis, the module must provide the appropriate handle to the load-instance.

Syntax:

```
LONG NPA_Unregister_Options(
       LONG npaHandle,
       LONG instance);
```

Parameters:

Inputs:

npaHandle The CDM's or HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_CDM_Module**() or **NPA_Register_CDM_Module**(), respectively.

instance The instance number the CDM or HAM associated with the current group of options being unregistered. This instance corresponds to either a CDM's device instance or a HAM's adapter instance. This is the instance number that the CDM or HAM passed to **NPA_Register_CDM_Module**() or **NPA_Register_HAM_Module**(), respectively.

> **Note:** By setting this parameter to -1, the NWPA unregisters all option instances associated with the CDM or HAM.

Outputs: None

Return Value: 0 if successful.
Non-zero if unsuccessful.

Description: **NPA_Unregister_Options**() is used to unregister the command line options associated with a module (or an instance of itself) prior to being unloaded. This API is called within the context of *CDM/HAM_Unload()*.

NPAB_Get_Alignment

Purpose: Called to obtain alignment requirements of the underlying platform.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPAB_Get_Alignment(
       LONG npaHandle,
       LONG type );
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

type 0 - Alignment requirement
1 - Best case alignment
Other - Undefined

Outputs: None

Return Value: Power of 2, byte-boundary data alignment requirement.

Description: If *type* is equal to 0, the function returns the data alignment requriement of a data object of an arbitrary type for the platform to function without execeptions or corrupted data. All operations and "real world" use of these operations should be considered in determining this value. That is, if DMAing into an arbitrary memory location can cause data corruption due to noncoherent caching, then the function should return a value equal to at least the cache line size. Without this function, you cannot write platform independent DMA code, since the code cannot determine what characteristics it must meet. If *type* is equal to 1, the function returns the data alignment requirement for the platform to function at its best performance. The value returned for *type* equal to 0 should always be less than or equal to the value returned for *type* equal to 1. For most Intel processor based platforms, *type* equal to 0 should return a 0 and *type* equal to 1 should return the bus width of the processor (4 for a 386 or 486). An HP-PA-RISC machine should return 32 for both *type* equal to 0 and *type* equal to 1, due to the requirements of the memory cache.

NPAB_Get_Bus_Info

Purpose: Returns the size of the bus addresses associated with *busTag*.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPAB_Get_Bus_Info(
       LONG npaHandle,
       LONG busTag,
       LONG *physicalMemAddrSize,
       LONG *ioAddrSize );
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

busTag An architecture dependent value returned by **NPAB_Get_Bus_Tag()**. It specifies the bus on which the operation is to be performed.

Outputs:

physicalMemAddrSize The size in bits of a physical address on the bus specified by *busTag*.

ioAddrSize The size in bits of an I/O address on the bus specified by *busTag*.

Return Value: 0 - The requested operation was completed successfully.
6 - The specified bus does not exist.

Description: See **Purpose:** above.

NPAB_Get_Bus_Name

Purpose: Gets the *busTag* name.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPAB_Get_Bus_Name(
       LONG npaHandle,
       LONG busTag,
       BYTE **busName );
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

busTag An architecture dependent value returned by **NPAB_Get_Bus_Tag()**. It specifies the bus on which the operation is to be performed.

Outputs:

busName This parameter gets a pointer to a NULL-terminated string, which is the architecture and platform dependent name of the specified bus.

Return Value: 0 - The requested operation was completed successfully.
4 - One of the parameters was invalid.

Description: The returned string belongs to the NetWare Bus Interface (NBI) and must not be modified by the HAM. If the HAM needs to reference this string at some later time, it should make a local copy of it.

NPAB_Get_Bus_Tag

Purpose: Takes the optional, user supplied *busName* parameter and returns a *busTag*.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPAB_Get_Bus_Tag(
       LONG npaHandle,
       BYTE *busName,
       LONG *busTag );
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

busName Pointer to an architecture dependent string that is determined by the platform developer. It specifies the bus on which the HAM's hardware is to be found.

Outputs:

busTag Receives a system architecture dependent value identifying a specific bus in the system. The HAM should save this value as it is needed as an input parameter to subsequent HAI/NBI routines and for registering hardware resources.

Return Value: 0 - The requested operation was completed successfully.
6 - No bus format that corresponds with *busName* was found.

Description: The HAM should not interpret *busName* or *busTag*, but simply use them as described in this specification.

A *busTag* value of 0 always refers to the default expansion system bus. A *busTag* value of -1 always refers to the processor (CPU) bus.

NPAB_Get_Bus_Type

Purpose: Returns a value indicating the bus type of the bus specified by *busTag*.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPAB_Get_Bus_Type(
        LONG npaHandle,
        LONG busTag,
        LONG *busType );
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

busTag This parameter is an architecture dependent value returned by **NPAB_Get_Bus_Tag()**. It specifies on which bus the operation is to be performed.

Outputs:

busTag A value indicating one of the following bus types as follows

0=ISA
1=MCA
2=EISA
3=PCMCIA
4=PCI
5=VESA
6=NuBus
7=Open Firmware Motherboard

Return Value: 0 - The requested operation was completed successfully.
4 - Parameter error, *busTag* was invalid.

Description: This routine returns a value indicating the bus type of the specified bus. All instances of a particular bus type return the same value. For example, all EISA buses return 2.

NPAB_Get_Card_Config_Info

Purpose: Retrieves and returns configuration information for bus architectures that keep this information on a per slot basis.

Architecture Type: All

Thread Context: Blocking

Requirements: None.

Syntax:

```
LONG NPAB_Get_Card_Config_Info(
       LONG npaHandle,
       LONG busTag,
       LONG uniqueID,
       LONG size,
       LONG param1,
       LONG param2,
       void *configInfo );
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

busTag An architecture dependent value returned by **NPAB_Get_Bus_Tag()**. It specifies the bus on which the operation is to be performed.

uniqueID Architecture dependent value returned by **NPAB_Get_Unique_Identifier()** or **NPAB_Search_Adapter()** that specifies the location on the bus where the adapter card is found.

size Specifies the number of bytes to be returned into the configuration buffer.

param1
param2 Bus architecture dependent values that further specify what information is to be returned.

Outputs:

configInfo A pointer to a bus architecture dependent structure used to receive the returned information. The caller needs to be sure that the buffer is at least *size* bytes long.

Return Value:

0 - The requested operation was completed successfully.
4 - One of the parameters was invalid.
5 - *busTag* denotes a bus type for which the slot has not configuration information.
6 - The *uniqueID* input parameter has no card present.

Description: Call **NPAB_Get_Card_Config_Info()** only if the *busTag* that identifies a bus has configuration information for the bus on a per slot basis. It is the

caller's responsibility to know how much and what sort of information is returned, so that *configInfo* is set pointing to a sufficiently large space and the resulting information can be interpreted. *Param1* and *param2* are defined on a per bus architecture basis. In other words, their meanings must be the same on all implementations of a particular bus but will vary from one bus to another. One or both of these parameters can be unused, and if unused, should be set to 0.

The following are the parameter values for the specified bus type.

EISA Bus

size	320
param1	EISA configuration block number
param2	n/a
configInfo	filled in with EISA configuration information for the specified *uniqueID*. For a definition of the information returned, see *EISA Specification*.

MCA Bus

size	8
param1	n/a
param2	n/a
configInfo	filled in with I/O port values from POS0 - POS7 (100h - 107h) for the specified *uniqueID*. For a definition of the information returned, see *Personal System/2 Hardware Interface Technical Reference*.

PCI Bus

size	256
param1	PCI function number
param2	n/a
configInfo	filled in with PCI configuration information for the specified *uniqueID*. For a definition of the information returned, see *PCI Local Bus Specification*.

PC Card Bus (PCMCIA)

size	large enough to contain the 37 bytes of information returned by **GetConfigurationInfo** (PCMCIA call) plus room for the tuples.
param1	n/a
param2	n/a
configInfo	filled in with PCMCIA configuration information for the specified *uniqueID*. The information is the data returned by **GetConfigurationInfo** (a PCMCIA call) and as many of the tuples as there is buffer space. For a definition of the information returned, see *PCMCIA Standards*.

NPAB_Get_Unique_Identifier

Purpose: Returns a bus-specific value that uniquely identifies a specific device (such as an OEM chip set) on an adapter.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPAB_Get_Unique_Identifier(
       LONG npaHandle,
       LONG busTag,
       LONG *parameters,
       LONG parameterCount,
       LONG *uniqueID );
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

busTag A system architecture dependent value returned by **NPAB_Get_Bus_Tag()**. It specifies on which bus the operation is to be performed.

parameters A bus-architecture-dependent array of parameters needed by the system to generate the unique identifier. These parameters specify values like slot and function. The following are the parameter values for each bus type:

EISA Bus

parameterCount	1
parameters[0]	physical slot number

MCA Bus

parameterCount	1
parameters[0]	physical slot number

PCI Bus

parameterCount	2
parameters[0]	0 (PCI version 2.0) physical slot number (PCI version 2.1)
parameters[1]	bus/device/function number combination equivalent to the value returned from the **PCIBIOSFindDevice** function.

PC Card (PCMCIA) Bus

parameterCount	TBD
parameters[0]	TBD

PnP ISA Bus

parameterCount	2
parameters[0]	CSN
parameters[1]	logical device number

Note: Novell provides a registry of the meanings of these parameters for each bus.

parameterCount The number of elements in the input parameter array, *parameters*.

Outputs:

uniqueID Receives the architecture-dependent value that uniquely identifies a specific device on an adapter.

Return Value:
0 - The requested operation was completed successfully.
4 - The *busTag* parameter was invalid.
6 - The function is not available.

Description: This routine allows for ergonomic parameters used in identifying adapters placed in physical slots and the functions on the adapter to be converted to system architecture-dependent values required in the operation of the adapter. Unique identifiers are interpreted only by other HAI/NBI routines. To the caller they are a "magic cookie" with no predefined format.

NPAB_Read_Config_Space

Purpose: Retrieves and returns configuration information for the bus architecture that keeps this information on a per slot basis.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPAB_Read_Config_Space(
       LONG npaHandle,
       LONG dataType,
       LONG busTag,
       LONG uniqueID,
       LONG offset,
       void *readData );
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

dataType Indicates the data type (and size) of the output data:

0 - BYTE	8 bits
1 - WORD	16 bits
2 - LONG	32 bits

busTag A system architecture dependent value returned by **NPAB_Get_Bus_Tag()**. It specifies on which bus the operation is to be performed.

uniqueID The unique identifier for the specified adapter or function as returned by **NPAB_Get_Unique_Identifier()**, **NPAB_Search_Adapter()**, or **NPAB_Scan_Card_Info()**.

offset The byte offset into the specified adapter or function's configuration space of the item to be read.

Outputs:

readData Receives an unsigned value of type *dataType*.

Return Value:

0 - The requested operation was completed successfully.
4 - The *busTag* parameter was invalid.
6 - The function is not available.

Description: This routine takes a bus identifier and an offset in that bus's configuration space and performs whatever operations are necessary to acquire and return the requested data.

This routine is provided only for drivers that need to interact with configuration space. On most buses, **NPAB_Get_Card_Config_Info()** will satisfy a driver's needs.

> **Note:** For most buses, this routine will do nothing. It has meaning only on buses that have a configuration address space that is separate from memory or I/O space (for example, a PCI bus).

NPAB_Scan_Bus_Info

Purpose: Specifies the buses that are available on the system.

Architecture Type: All

Thread Context: Blocking

Requirements: None.

Syntax:

```
LONG NPAB_Scan_Bus_Info(
        LONG npaHandle,
        LONG *scanSequence,
        LONG *busTag,
        LONG *busType,
        BYTE **busName );
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

scanSequence Initialized to -1 to start the first search iteration.

Outputs:

scanSequence Receives a system-generated sequence value to be passed into subsequent calls to this routine.

busTag Receives an architecture-dependent value used by the system to identify the bus found in the current search iteration.

busType Receives a value indicating the bus type of the target bus found in the current search iteration:

0 = PC ISA bus
1 = PC MCA bus
2 = PC EISA bus
3 = PC Card (PCMCIA) bus
4 = PCI bus
5 = VESA local bus
6 = NuBus
7=Open Firmware Motherboard

busName Receives a pointer to a static, NULL-terminated, architecture-dependent string for the target bus found in the current search iteration. This string is determined by the system platform developer. The caller should not modify this string. To reference this string, make a copy of it.

Return Value:

0 - The requested operation was completed successfully.
4 - One or more of the parameters was invalid.
6 - There are no more buses

Description: This routine scans the system for available buses on a find-first-find-next basis. The routine returns *busTag*, *busType*, and *busName* information about the target bus for each iteration.

NPAB_Search_Adapter

Purpose: Takes a bus type and a pointer to a product ID and returns a bus tag and unique identifier indicating where the specified product (adapter board) was found.

Architecture Type: All

Thread Context: Blocking

Requirements: None.

Syntax:

```
LONG NPAB_Search_Adapter(
       LONG npaHandle,
       LONG *scanSequence,
       LONG busType,
       LONG productIDLength,
       BYTE *productID,
       LONG *busTag,
       LONG *uniqueID );
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

scanSequence Initialized to -1 to start the first search iteration.

busType Indicates the bus type on which to perform the search:

0 = PC ISA bus
1 = PC MCA bus
2 = PC EISA bus
3 = PC Card (PCMCIA) bus
4 = PCI bus
5 = VESA local bus
6 = NuBus
7= Open Firmware Motherboard

productIDLength Byte-length of the product ID string.

productID Pointer to a bus-architecture-dependent parameter that uniquely identifies an adapter board/peripheral/system option. For example, for an EISA bus, the EISA product ID is defined in the *EISA Specification* document.

Outputs:

scanSequence Receives a system-generated sequence value to be passed into subsequent calls to this routine.

busTag Receives an architecture-dependent value used by the system to identify the bus on which the adapter was found in the current search iteration.

uniqueID Receives an architecture-dependent value identifying the specific device or function. Iterative calls to this routine will return information for each instance of the *productID* and compatible products, including multiple instances on a single card (each have a different function number). The slot number associated with the adapter can be gleaned from *uniqueID* using **NPAB_Get_Unique_Identifier()**.

Return Value:
0 - The requested operation was completed successfully.
4 - One or more of the parameters was invalid.
6 - No more items present

Description: The HAM calls this routine reiteratively to find all adapter instances with the specified product ID. The routine returns the bus tag and the system unique ID for each adapter instance.

This routine can only be used if the HAM's adapter has a unique product ID associated with it that can be read by NetWare's bus interface (NBI). Also, the product ID must be retrievable according to some accepted standard, such as EISA, MCA, or PCI.

NPAB_Write_Config_Space

Purpose: Writes information to the configuration space for the bus architecture that keeps this information on a per slot basis.

Architecture Type: All

Thread Context: Non-Blocking

Requirements: None.

Syntax:

```
LONG NPAB_Write_Config_Space(
       LONG npaHandle,
       LONG dataType,
       LONG busTag,
       LONG uniqueID,
       LONG offset,
       void *writeData );
```

Parameters:

Inputs:

npaHandle The HAM's handle for using the **NPA_** APIs, assigned during **NPA_Register_HAM_Module()**.

dataType Indicates the data type (and size) of the output data:

0 - BYTE	8 bits
1 - WORD	16 bits
2 - LONG	32 bits

busTag A system architecture dependent value returned by **NPAB_Get_Bus_Tag()**. It specifies on which bus the operation is to be performed.

uniqueID The unique identifier for the specified adapter or function as returned by **NPAB_Get_Unique_Identifier()**, **NPAB_Search_Adapter()**, or **NPAB_Scan_Card_Info()**.

offset The byte offset into the specified adapter or function's configuration space of the item to be read.

writeData Pointer to the data item of type *dataType* that is to be written in the specified configuration address on the specified bus.

Outputs: None

Return Value:

0 - The requested operation was completed successfully.
4 - The *busTag* parameter was invalid.
6 - The function is not available.

Description: This routine takes a value, a bus identifier and an offset in that bus's configuration space and performs whatever operations are necessary to deliver the value to the specified location.

This routine is provided only for drivers that need to interact with configuration space. Usually, any "writes" to configuration space are done by the system or a configuration management utility before any drivers are loaded.

> **Note:** For most buses, this routine will do nothing. It has meaning only on buses that have a configuration address space that is separate from memory or I/O space (for example, a PCI bus).

Out*x*

Purpose: Takes a bus identifier, a value, and an I/O address in that bus's I/O address space and performs whatever operations are necessary to deliver the value to the specified place.

Thread Context: Non-Blocking

Syntax:

```
void Out8 (
       LONG busTag,
       void *ioAddr,
       BYTE outputVal );

void Out16 (
       LONG busTag,
       void *ioAddr,
       WORD outputVal );

void Out32 (
       LONG busTag,
       void *ioAddr,
       LONG outputVal );
```

Parameters:

Inputs:

busTag An architecture dependent value returned by **NPAB_Get_Bus_Tag()**. This value specifies the bus on which the operation is to be performed.

ioAddr The I/O address in the bus architecture of the adapter to which the output is to occur.

outputVal The value to be sent to the specified I/O address on the specified bus. The type of this value must correspond with the routine being called.

Outputs: None

Return Value: None

Description: These routines are only used by HAMs written for adapters intended for bus architectures that have an I/O address space. The HAM is expected to use the routine appropriate to the data width of the port to which the output is to occur.

The value of *ioAddr* should be the port address the HAM would normally expect for the given bus architecture. For example, if an ISA card with a base port address of 300h is placed on an EISA bus, the HAM will set *ioAddr* to 300h when it wants to output to that base port.

OutBuff*x*

Purpose: Takes a bus identifier, an I/O address in that bus's I/O address space, a source buffer in the CPU's logical address space, and a count of transfer data units to perform whatever operations are necessary to output the specified number of data units from the source buffer to the I/O address.

Thread Context: Non-Blocking

Syntax:

```
LONG OutBuff8 (
       LONG busTag,
       void *ioAddr,
       void *buffer,
       LONG count );

LONG OutBuff16 (
       LONG busTag,
       void *ioAddr,
       void *buffer,
       LONG count );

LONG OutBuff32 (
       LONG busTag,
       void *ioAddr,
       void *buffer,
       LONG count );
```

Parameters:

Inputs:

busTag An architecture dependent value returned by **NPAB_Get_Bus_Tag()**. This value specifies the bus on which the operation is to be performed.

ioAddr The I/O address in the bus architecture of the adapter to which the output is to occur.

buffer The logical memory address of the source buffer. This address is in the CPU's logical address space.

count The number of transfer units in the specified data size.

Outputs: None

Return Value:

0 - The requested operation was completed successfully.
1 - Memory protection prevented by the completion of the requested operation.
3 - Memory error occurred while attempting to perform the requested operation.
4 - One of the parameters was invalid.
5 - The requested operation could not be completed.

Description: These routines are only used by HAMs written for adapters intended for bus architectures that have an I/O address space. The HAM is expected to use the routine appropriate to the data width of the port to which the output is to occur. The specified number of data units from the source buffer is output to the specified I/O address.

The value of *ioAddr* should be the port address the HAM would normally expect for the given bus architecture. For example, if an ISA card with a base port address of 300h is placed on an EISA bus, the HAM will set *ioAddr* to 300h when it wants to output to that base port.