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

C 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.

C 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

C 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.

C 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

C 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
```

appReturnCode Application return code. This parameter passes specific information directly 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 **HACBPutHandl**e 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

Outputs: None

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);
```

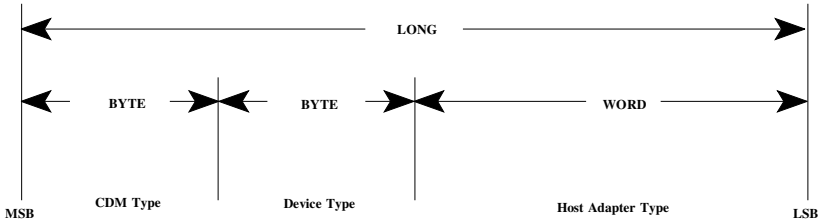
Parameters:

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.

Outputs: None

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.

cdmHandle 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()**.

Outputs: None

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.