# Table of Contents

1. Introduction ...................................................................................................... 1
   1.1. About this Guide ............................................................................................ 1
   1.2. Documentation Conventions .......................................................................... 1
   1.3. Key DM3 Architecture Concepts ................................................................... 2
   1.4. DM3 Direct Interface Overview .................................................................... 3
       1.4.1. DM3 Direct Interface Host Library ....................................................... 4
       1.4.2. DM3 Device Drivers.............................................................................. 5
   1.5. DM3 Hardware ............................................................................................... 5
   1.6. DM3 Firmware ............................................................................................... 6
2. Function Summary .......................................................................................... 7
   2.1. Naming Conventions ...................................................................................... 7
   2.2. Calling Functions Asynchronously ................................................................. 8
       2.2.1. OVERLAPPED Structure ...................................................................... 8
       2.2.2. Handling Asynchronous Function Returns ............................................ 9
   2.3. Calling Functions Synchronously ................................................................. 11
       2.3.1. Handling Synchronous Function Returns ............................................. 12
   2.4. Function Categories ...................................................................................... 15
       2.4.1. Cluster Management Functions............................................................ 16
       2.4.2. Component Management Functions..................................................... 17
       2.4.3. Debug Support Functions .................................................................... 17
       2.4.4. Stream I/O Functions ........................................................................... 18
       2.4.5. Message I/O Functions ........................................................................ 18
       2.4.6. Exit Notification Functions .................................................................. 20
3. Function Reference ........................................................................................ 21
   mntAllocateMMB( ) - allocates and clears a Message Block ............................... 22
   mntAttachMercStream( ) - opens a stream and attaches it to a stream handle ..... 25
   mntClearMMB( ) - clears the command and reply message areas ...................... 29
   mntClusterAllocate( ) - finds and allocates a cluster ............................................ 34
   mntClusterByComp( ) - finds the cluster that owns an instance ......................... 37
   mntClusterCompByAttr( ) - finds a component with specific attributes ............ 39
   mntClusterConfigLock( ) - locks a specific cluster ..................................... 42
   mntClusterConfigUnlock( ) - unlocks a previously-locked cluster ................. 44
   mntClusterConnect( ) - interconnects the ports of two instances ...................... 46
   mntClusterCreate( ) - creates a new cluster ................................................. 50
mntClusterDeactivate( ) - deactivates connections ............................................... 53
mntClusterDestroy( ) - destroys an empty cluster .............................................. 56
mntClusterDisconnect( ) - breaks an existing connection between ports .......... 58
mntClusterFind( ) - finds a cluster that has specific attributes ........................... 61
mntClusterFree( ) - releases an allocated cluster ............................................. 63
mntClusterSlotInfo( ) - finds the time slots assigned to a port ....................... 65
mntClusterTSAssign( ) - assigns time slots to a cluster’s SCbus resource .......... 68
mntClusterTSUnassign( ) - unassigns a timeslot from an SCbus resource ......... 71
mntCompAllocate( ) - reserves and locks a specific component instance .......... 74
mntCompFind( ) - finds a component ............................................................ 77
mntCompFindAll( ) - returns component addresses with specified attributes .. 81
mntCompFree( ) - releases an allocated component instance ............................ 86
mntCompUnuse( ) - marks component instances as not being in use ............... 88
mntCompUse( ) - marks component instances as being in use .......................... 90
mntCompleteStreamIo( ) - completes pending stream I/O requests .................. 92
mntCopyMMB( ) - copies the specified Message Block .................................... 93
mntDetachMercStream( ) - detaches a stream ............................................... 94
mntEnumMpathDevice( ) - enumerates existing Mpath devices ....................... 96
mntEnumStrmDevice( ) - enumerates existing Stream devices ....................... 98
mntFreeMMB( ) - frees the specified Message Block .................................... 100
mntGetBoardsByAttr( ) - lists boards with matching attributes ...................... 101
mntGetDrvVersion( ) - retrieves the driver version string .............................. 105
mntGetLibVersion( ) - retrieves the Direct Interface library version string ..... 106
mntGetMpathAddr( ) - returns the message path source address .................... 109
mntGetStreamHeader( ) - gets the out-of-band stream attributes .................. 111
mntGetStreamInfo( ) - gets global board-specific stream information .......... 114
mntGetTLSmmb( ) - retrieves the thread-local storage MMB ......................... 116
mntNotifyRegister( ) - enables notification of sub-component failure .......... 118
mntNotifyUnregister( ) - disables notification of sub-component failure ...... 121
mntRegisterAsyncMessages( ) - enables receipt of asynchronous messages .... 123
mntRegisterAsyncStreams( ) - enables receipt of asynchronous stream data .... 126
mntSendMessage( ) - sends the message specified in the MMB ..................... 129
mntSendMessageWait( ) - builds an MMB, sends it, then synchronously waits for I/O completion .......................................................... 131
mntSetExitNotify( ) - enables notification of Mpath device failure ............... 133
mntSetStreamHeader( ) - sets the out-of-band stream attributes .................. 135
mntSetStreamIOTimeout( ) - sets the stream I/O request timeout value ......... 138
mntSetTraceLevel( ) - enables or disables trace statements .......................... 139
Table of Contents

mntTerminateStream( ) - cancels a persistent stream ......................................... 141
mntTrace( ) - sends trace statements to a file ..................................................... 143
mntTransGen( ) - generates a message transaction ID ....................................... 145
qMsgVarFieldGet( ) – gets typed fields from a message payload ...................... 146
qMsgVarFieldPut( ) – puts typed fields into a message payload ...................... 150

4. Macro Reference ......................................................................................... 155
   4.1. Multiple Message Block ........................................................................ 155
   4.2. MMB Control Header Macros ............................................................... 156
   4.3. DM3 Message Macros ........................................................................... 163
      4.3.1. DM3 Message Pointer Macros ....................................................... 163
      4.3.2. DM3 Message Header Macros ....................................................... 164
   4.4. DM3 Messages with Payloads................................................................. 166
      4.4.1. Messages With Fixed Payloads ....................................................... 166
      4.4.2. Messages with Variable Payloads ................................................... 167

5. Data Types, Structures, and Error Codes ................................................. 169
   5.1. Data Types ............................................................................................ 169
   5.2. Data Structures ..................................................................................... 170
      5.2.1. MSB Stream Buffer Structure ....................................................... 170
      5.2.2. STRM_HDR Stream Header Structure ......................................... 171
      5.2.3. STRM_INFO Stream Information Structure .................................. 172
      5.2.4. QBoardAttr Board Attribute Structure ........................................ 173
      5.2.5. QCompAttr Component Attribute Structure .................................. 173
      5.2.6. QCompDesc Component Descriptor Structure .............................. 175
      5.2.7. QValueAttr Board Attribute Specification Structure .................... 177
   5.3. Error Code Definitions ........................................................................... 177
      5.3.1. Windows NT Error Codes .............................................................. 177
      5.3.2. Dialogic Library and Driver Error Codes ....................................... 180

Index ................................................................................................................ 183
List of Tables

Table 1. Direct Interface Host Library Function Categories ......................................... 15
Table 2. Cluster Management Functions ........................................................................ 16
Table 3. Component Management Functions ................................................................ 17
Table 4. Debug Support Functions ................................................................................. 17
Table 5. Stream I/O Functions ....................................................................................... 18
Table 6. Message I/O Functions ..................................................................................... 19
Table 7. Exit Notification Functions .............................................................................. 20
Table 8. Messages with Fixed Payloads ....................................................................... 167
Table 9. Messages with Variable Payloads .................................................................... 168
Table 10. Data Type Definitions .................................................................................... 169
Table 11. Component Attribute Values ......................................................................... 174
Table 12. Windows NT General Error Codes ............................................................... 178
Table 13. Windows NT Stream Error Codes ................................................................. 179
Table 14. Dialogic Error Codes ..................................................................................... 180
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>DM3 Direct Interface Components</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Handling Asynchronous Function Returns</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Handling Synchronous Function Returns</td>
<td>13</td>
</tr>
<tr>
<td>Figure 4</td>
<td>General MMB Structure</td>
<td>156</td>
</tr>
</tbody>
</table>
1. Introduction

The DM3 Direct Interface host library functions, macros, and data structures that provide access to the DM3 devices under Windows NT are described in this document. Use the Direct Interface in conjunction with the Win32 API to produce highly native DM3-based applications.

1.1. About this Guide

This guide, the *DM3 Direct Interface Function Reference for Windows NT*, contains the following:

**Chapter 1** provides a brief overview of the DM3 Direct Interface.

**Chapter 2** summarizes the Direct Interface host library functions and describes their syntax convention.

**Chapter 3** provides complete details about all Direct Interface host library functions, which are listed alphabetically.

**Chapter 4** contains descriptions of macros provided with the Direct Interface.

**Chapter 5** describes data structures, data types, parameters, and constants used by the Direct Interface host library functions.

For information on creating applications with the Direct Interface, refer to the guide *Using the DM3 Direct Interface for Windows NT*. For details on library functions, macros, and data structures, refer to this guide, the *DM3 Direct Interface Function Reference for Windows NT*.

1.2. Documentation Conventions

The following conventions are used throughout this guide:

- New terms are shown in *italic text*.
- Important words or phrases are shown in **bold text**.
DM3 Direct Interface Function Reference for Windows NT

- File names are shown in lowercase italic text, such as stddefs.h.
- Function names are shown in boldface with parentheses, such as mntSendMessage().
- Data structure field names and function parameter names are shown in boldface, as in timeout.

1.3. Key DM3 Architecture Concepts

This section offers a brief explanation of the concepts that you must be familiar with before you begin working with DM3 products. For more information about these concepts, see the DM3 Mediastream Architecture Overview.

- DM3 is an architecture on which a set of Dialogic products are built. The DM3 architecture is open, layered, and flexible, encompassing hardware as well as software components.

- A DM3 resource is a conceptual entity implemented in firmware that runs on DM3 hardware. A resource contains a well defined interface or message set, which the host application uses when accessing the resource. The message set for each resource is described in a DM3 Resource User's Guide.

  Resource firmware consists of multiple components that run on the DM3 core platform software. The DM3 GlobalCall resource is an example of such a resource, providing all of the features and functionality necessary for handling calls.

- A component is an entity that comprises a DM3 resource. A component runs on a DM3 control processor or signal processor depending on its function. Certain components handle configuration and management issues, while others process stream data.

  To access the features of a resource, the host exchanges messages and stream data with certain components of that resource. During runtime, components inside a resource communicate (via messages) with other components of that resource, as well as with components of other resources.

- A component instance is a logical entity that represents a single thread of control for the operations associated with a DM3 component. DM3 components generally support multiple instances so that a single component on a single processor can be used to process multiple streams or channels.
Instances are addressable units and DM3 messages may be sent to individual instances of a component.

- A DM3 **message** is a formatted block of data exchanged between the host and component instances, between component instances and the core platform software, as well as between the DM3 component instances themselves.

  The DM3 architecture implements different kinds of messages, based on the functionality of the message sender and recipient. Messages can initiate actions, handle configuration, affect operating states, and indicate that events have occurred.

- A **cluster** is a collection of DM3 component instances that share specific timeslots on the network interface or the Time Division Multiplexed (TDM) bus, and which therefore operate on the same data stream. The cluster concept in the DM3 architecture corresponds generally to the concept of a “group” in S.100, or to a “channel” in conventional Dialogic architectural terminology. Component instances are bound to a particular cluster and its assigned timeslots in an allocation operation.

- A **port** is a logical entity that represents the point at which Pulse Code Modulated (PCM) data can flow between component instances in a cluster. Ports are classified and designated in terms of data flow direction and the type of component instance that provides the port.

### 1.4. DM3 Direct Interface Overview

The DM3 Direct Interface is a low-level interface. By sending and receiving messages, the Direct Interface provides access to the DM3-based embedded system, and shields you from device driver specifics. You can use the Direct Interface as the foundation from which you can build a higher-level API. Win32 file- and resource-management services are available to you when using the Direct Interface.

The Direct Interface consists of the DM3 Host Library and DM3 Device Drivers (a Class Driver and a Protocol Driver). Applications communicate with the host library; the device drivers are not accessed directly.

*Figure 1* illustrates the host and embedded portions of a generic DM3-based system.
1.4.1. DM3 Direct Interface Host Library

The DM3 Direct Interface host library (mnti.lib) is the lowest-level interface for accessing DM3 devices. Use the library in conjunction with the Win32 API to produce native Windows NT applications. The DM3 Direct Interface provides configuration management, message allocation, messaging, cluster and time slot management, and data stream services.

All device handles used with the Direct Interface are native Win32 handles and are passed directly to Win32 event functions. The host library protects shared data structures from being overwritten when they are used by multiple threads.

An application built with the Direct Interface uses the **Multiple Message Block** (MMB) as the primary data structure. The MMB is used to send messages to and receive messages from the DM3 embedded system.
1. Introduction

1.4.2. DM3 Device Drivers

DM3 device drivers include the Dialogic Class Driver and Dialogic Protocol Driver. Application developers do not need to access these drivers directly; the Host Library is used to communicate with these drivers.

The Dialogic Class Driver (dlgcmcd.sys) is the highest-level driver that interacts with the Dialogic Protocol Driver. The Class Driver recognizes DM3 device names (Mpath for messages and Strm for streams) and supports all Win32 API I/O function calls that perform bulk data transfers, including CreateFile(),.ReadFile(), and WriteFile().

The Dialogic Protocol Driver (dlgcmpd.sys) is the lowest-level driver that handles all I/O operations between a DM3 embedded system and the host machine. The Protocol Driver communicates through shared memory (Shared RAM) that is mapped to the system address space. For PCI devices, this mapping takes place when the Protocol Driver loads and initializes. (More precisely, the PCI configuration process is handled by Windows NT at boot time and later, the Protocol Driver discovers and claims the DM3 boards.) The Protocol Driver supports both PIO (Programmed Input/Output) and DMA (Direct Memory Access).

1.5. DM3 Hardware

The hardware used in a DM3 embedded system is a modular and scaleable implementation of the DM3 architecture. A DM3 product consists of one baseboard, up to three signal-processing daughterboards, and other hardware components.

The baseboard hardware is available in the following form factors:
- PCI (Peripheral Component Interconnect)
- CompactPCI (Compact Peripheral Component Interconnect)
- VME (Versa Module Europa)

A configured hardware assembly is installed in a chassis. For details about installing a particular board assembly, refer to the Quick Install Card packaged with the product.
1.6. DM3 Firmware

At system startup, binary code is downloaded to the DM3 board assembly. The firmware on the assembly is the ultimate target of all I/O operations. It includes components, kernels, and service managers.

For more information about these concepts, see the *DM3 Mediastream Architecture Overview*. 
2. Function Summary

Direct Interface host library functions are summarized and listed by category in this chapter. Calling functions asynchronously and synchronously is also described.

2.1. Naming Conventions

The following naming conventions are used throughout this manual:

- **Function Names** are shown in bold mixed case type, such as `mntSetTraceLevel()`. Each function name begins with "mnt" followed by one or more words describing that function. Each word within a function name begins with a capital letter, there are no separator characters, and the name ends with a set of parentheses.

- **Macro Names** are shown in one of two ways, depending on the macro type. Macros used to access DM3 messages and Multiple Message Blocks (MMBs) are shown in non-bold uppercase type, such as MNT_GET_CMD_QMSG. Macros used to access DM3 structures are shown in non-bold mixed case type, such as QResultError_get.

- **Data Type Names (typedef)** are shown in non-bold type, such as char, and PQBoardAttr. Data type names can be uppercase, lowercase, or mixed case.

- **Constant Definitions (#define)** are shown in non-bold uppercase type, such as MNTI_STATE_PRE_INIT and MNTI_STATE_INITIALIZED. Each constant definition begins with “MNTI” followed by one or more words describing that constant. Underscore separators between words aid readability. Related constant definitions share the same first word.

- **Parameter Names** are shown in bold mixed case type, such as nTimeout. Words within a parameter name begin with a capital letter, such as nReplyCount. Pointer parameter names begin with either “p” or “lp,” such as pAttr or lpMMB. Within each function syntax table in Chapter 3. Function Reference, input parameter names are listed above output parameter names.
2.2. Calling Functions Asynchronously

All Direct Interface host library functions that accept the lpOverlapped parameter can operate in either asynchronous or synchronous mode. If the lpOverlapped parameter is non-NULL, the call is in an asynchronous (overlapped) I/O mode and the function returns immediately before the actual I/O completes.

2.2.1. OVERLAPPED Structure

When calling a function asynchronously, you must set the lpOverlapped parameter to a non-NULL value. The OVERLAPPED structure is a Win32 API asynchronous I/O data structure. An application normally allocates and initializes this structure, then passes it to the Win32 API functions, such as ReadFile( ) and WriteFile( ). An application can specify the hEvent field in the OVERLAPPED structure to the Win32 API wait-for-object functions, such as WaitForSingleObject( ), to provide notification of asynchronous function completion.

The application is responsible for managing the OVERLAPPED structure. If multiple requests are outstanding on the same device, each request must be associated with a unique OVERLAPPED structure.

If the message path handle, which is specified through the hDevice parameter, has been opened with the FILE_FLAG_OVERLAPPED flag set in the dwFlagsAndAttributes parameter in the CreateFile( ) function call, the application can pass a valid lpOverlapped parameter with the request. The calling thread can use any wait function to wait for the event object, a member of the OVERLAPPED structure, to be signaled, then call the GetOverlappedResult( ) function to determine the operation’s results.

If the specified message path handle has been opened without setting the FILE_FLAG_OVERLAPPED flag, the lpOverlapped parameter should be set to NULL. The function either completes the operation synchronously or times out. If the function returns TRUE, it has completed successfully. Otherwise, it has failed or timed out, and the calling thread must call the GetLastError( ) function to retrieve the error.
2. Function Summary

2.2.2. Handling Asynchronous Function Returns

The operations detailed below and the flow chart in Figure 2 describe the steps to follow when a function returns that was called asynchronously.

1. A Direct Interface function will always return FALSE when called asynchronously. Call the Win32 GetLastErr() function to retrieve an error code. The error code may be one of three types: Windows NT (defined in winerror.h), DM3 Direct Interface (defined in dllmnti.h), or DM3 Kernel (defined in qkernerr.h).

2. If GetLastError() returns ERROR_IO_PENDING, it indicates the operation has not completed. Wait for function completion using the Win32 wait-for-object functions WaitForSingleObject() or WaitForMultipleObjects() depending on the number of expected objects.

If GetLastError() returns a different error code, process it as either a Windows NT error or DM3 Direct Interface error.

3. Upon function completion, call the GetOverlappedResult() function.

4. Call the MNT_GET_REPLY_QMSG() macro to find the reply message.

5. Use the QMSG_GET_MSGTYPE() macro on the reply message to determine the reply message type.

6. If the message type is QResultError, call the QResultError_get() macro and process the kernel error (defined in qkernerr.h).

7. If the message type is not QResultError, the function has completed successfully and the result message contents may be processed.
Figure 2. Handling Asynchronous Function Returns
This code fragment provides a general example of handling a function return asynchronously.

```c
if (mntSendMessage(DevHandle, lpMMB, &Overlapped) == FALSE) {
    // Call GetLastError to get the error code
    ErrorCode = GetLastError();
    if (ErrorCode == ERROR_IO_PENDING) {
        // Now wait for operation to complete
        if ((WaitForSingleObject(DevHandle, INFINITE)) == WAIT_FAILED) {
            // perform error handling
            return(FALSE);
        }
        if (GetOverlappedResult(DevHandle, &Overlapped,
                        &RecvByteCount, FALSE) == FALSE) {
            // Call GetLastError to get the error code
            ErrorCode = GetLastError();
            // perform error handling
            return(FALSE);
        }
    }
}

/* If send message is successful, retrieve results */
MNT_GET_REPLY_QMSG(lpMMB, 1, &pMsg);

/* Check for firmware error */
QMSG_GET_MSGTYPE(pMsg, &ReplyType);

if (ReplyType == QResultError) {
    /* Error, print error code */
    QResultError_t qr;
    QResultError_get(pMsg, &qr, Offset);
    printf("Error %x\n", qr.errorCode );
    goto cleanup;
}
```

### 2.3. Calling Functions Synchronously

Some Direct Interface host library functions, such as `mntAllocateMMB()`, work only in synchronous mode. As stated earlier, most functions can operate either asynchronously or synchronously depending on the `lpOverlapped` parameter.
2.3.1. Handling Synchronous Function Returns

The operations detailed below and the flow chart in Figure 3 describe the steps to follow when a function returns that was called synchronously.

If the function return value is TRUE, it indicates that the driver successfully processed the arguments. Any expected function outputs will have valid contents. For example, if the `mountCompFind()` function is called in synchronous mode and valid arguments are sent and returned, when the TRUE return message is received, the variable pointed to by the `lpInstance` argument will contain the returned component descriptor.

If the function return value is FALSE, the function call has failed. Perform the following steps to process the failure:

1. Call the Win32 `GetLastError()` function to retrieve an error code. The error code may be one of three types: Windows NT (defined in `winerror.h`), DM3 Direct Interface (defined in `dllmnti.h`), or DM3 Kernel (defined in `qkernerr.h`).
2. Logically AND the mask constant `ERROR_MNT_BASE` with the value returned from `GetLastError()` to determine if the error is Windows NT or Direct Interface.
3. If `GetLastError()` returns `ERROR_MNT_MERCURY_KERNEL`, it indicates a DM3 Kernel error has occurred.
   If `GetLastError()` returns a different error code, process it as either a Windows NT error or Direct Interface error.
4. Call the `mntGetTLSmmb()` function, which returns a pointer to the reply message contained in the thread-local-storage MMB.
5. Use the `QMSG_GET_MSGTYPE()` macro on the reply message to determine the reply message type.
6. If the message type is `QResultError`, call the `QResultError_get()` macro and process the kernel error (defined in `qkernerr.h`).
7. If the message type is not `QResultError`, the error is undefined.
2. Function Summary

Figure 3. Handling Synchronous Function Returns
This code fragment provides a general example of handling a function return synchronously.

```c
/* Issue the command */
if ( mntClusterCompInfo( hMCD,
    mntTransGen(),
    &clusterAddr,
    &count,
    compDescs,
    DEF_TIMEOUT,
    NULL,
    NULL ) == FALSE ) {
    printf( "mntClusterCompInfo failed %d", GetLastError() );
    /* If send message is successful, retrieve results */
    mntGetTLSmmb( &lpMMB, NULL, &pMsg );

    /* Check for firmware error */
    QMSG_GET_MSGTYPE(pMsg, &ReplyType);
    if (ReplyType == QResultError) {
        /* Error, print error code */
        QResultError_t qr;
        QResultError_get(pMsg, &qr, Offset);
        printf("Error %x\n", qr.errorCode );
        goto cleanup;
    }
    return(1);
}

/* Success! comp desc array is filled in by mntClusterCompInfo() */
printf("mntClusterCompInfo successful count = %d\n", count);
```
2. Function Summary

2.4. Function Categories

The following sections divide the function calls in the DM3 Direct Interface for Windows NT into categories. Categories are listed in Table 1. Each function call in a category is related by the task that the function performs.

Table 1. Direct Interface Host Library Function Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster management functions</td>
<td>Provide a set of tools to manage clusters and time slots</td>
</tr>
<tr>
<td>Component management functions</td>
<td>Provide a set of configuration and registration services for control of firmware components and component instances</td>
</tr>
<tr>
<td>Debug support functions</td>
<td>Provide a set of services that allow the run-time collection of data for debug tracing and the background verification of application code and data.</td>
</tr>
<tr>
<td>Stream I/O functions</td>
<td>Provide access to bulk data transfers to and from stream devices</td>
</tr>
<tr>
<td>Message I/O functions</td>
<td>Provide a set of services for generating, transferring, and accessing messages passed between the host and component instances</td>
</tr>
<tr>
<td>Exit Notification functions</td>
<td>Provide on/off switching of exit notification services</td>
</tr>
</tbody>
</table>
2.4.1. Cluster Management Functions

The Direct Interface host library cluster management functions provide a set of tools to manage clusters and time slots.

Table 2. Cluster Management Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mntClusterActivate( )</td>
<td>Activates an OUT-port connection</td>
</tr>
<tr>
<td>mntClusterAllocate( )</td>
<td>Finds and allocates a cluster</td>
</tr>
<tr>
<td>mntClusterByComp( )</td>
<td>Finds the cluster that owns an instance</td>
</tr>
<tr>
<td>mntClusterByAttr( )</td>
<td>Finds a component with specific attributes</td>
</tr>
<tr>
<td>mntClusterConfigLock( )</td>
<td>Locks a specific cluster</td>
</tr>
<tr>
<td>mntClusterConfigUnlock( )</td>
<td>Unlocks a previously locked cluster</td>
</tr>
<tr>
<td>mntClusterConnect( )</td>
<td>Interconnects the ports of two instances</td>
</tr>
<tr>
<td>mntClusterCreate( )</td>
<td>Creates a new cluster</td>
</tr>
<tr>
<td>mntClusterDeactivate( )</td>
<td>Deactivates connections</td>
</tr>
<tr>
<td>mntClusterDestroy( )</td>
<td>Destroys an empty cluster</td>
</tr>
<tr>
<td>mntClusterDisconnect( )</td>
<td>Breaks an existing connection between ports</td>
</tr>
<tr>
<td>mntClusterFind( )</td>
<td>Finds a cluster that has specific attributes</td>
</tr>
<tr>
<td>mntClusterFree( )</td>
<td>Releases an allocated cluster</td>
</tr>
<tr>
<td>mntClusterSlotInfo( )</td>
<td>Finds time slots assigned to a port</td>
</tr>
<tr>
<td>mntClusterTSAssign( )</td>
<td>Assigns time slots to a cluster’s SCbus resource</td>
</tr>
<tr>
<td>mntClusterTSUnassign( )</td>
<td>Unassigns a time slot from a cluster’s SCbus resource</td>
</tr>
</tbody>
</table>
2. Function Summary

2.4.2. Component Management Functions

The Direct Interface host library component management functions provide a set of configuration and registration services for control of application firmware components and component instances.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mntCompAllocate( )</td>
<td>Reserves and locks a specific component instance</td>
</tr>
<tr>
<td>mntCompFind( )</td>
<td>Finds a component</td>
</tr>
<tr>
<td>mntCompFindAll( )</td>
<td>Returns a list of component addresses matching specified attributes</td>
</tr>
<tr>
<td>mntCompFree( )</td>
<td>Releases an allocated component instance</td>
</tr>
<tr>
<td>mntCompUnuse( )</td>
<td>Marks component instances as not being in use</td>
</tr>
<tr>
<td>mntCompUse( )</td>
<td>Marks component instances as being in use</td>
</tr>
</tbody>
</table>

2.4.3. Debug Support Functions

The Direct Interface host library debug support functions provide a set of services that allow the run-time collection of data for debug tracing and the background verification of application code and data.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mntGetDrvVersion( )</td>
<td>Retrieves the driver version string from the Class Driver (DLGCMCD)</td>
</tr>
<tr>
<td>mntGetLibVersion( )</td>
<td>Retrieves the host library version string from the Class Driver (DLGCMCD)</td>
</tr>
<tr>
<td>mntSetTraceLevel( )</td>
<td>Enables or disables trace statements</td>
</tr>
<tr>
<td>mntTrace( )</td>
<td>Sends trace statements to a file</td>
</tr>
<tr>
<td>mntTransGen( )</td>
<td>Generates a message transaction ID</td>
</tr>
</tbody>
</table>
2.4.4. Stream I/O Functions

The Direct Interface host library stream I/O functions provide access to bulk data transfers to and from stream devices.

Table 5. Stream I/O Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mntAttachMercStream()</td>
<td>Opens a stream and attaches the stream ID to a stream handle.</td>
</tr>
<tr>
<td>mntCompleteStreamIo()</td>
<td>Completes pending stream I/O requests</td>
</tr>
<tr>
<td>mntCheckStreamOrphans()</td>
<td>Checks for orphan bytes</td>
</tr>
<tr>
<td>mntDetachMercStream()</td>
<td>Deallocates a reference to a stream ID</td>
</tr>
<tr>
<td>mntGetMercStreamID()</td>
<td>Retrieves a stream ID</td>
</tr>
<tr>
<td>mntGetStreamHeader()</td>
<td>Gets the out-of-band stream attributes</td>
</tr>
<tr>
<td>mntGetStreamInfo()</td>
<td>Gets global board-specific stream information</td>
</tr>
<tr>
<td>mntRegisterAsyncStreams()</td>
<td>Registers a number of stream buffers for receipt of asynchronous stream data</td>
</tr>
<tr>
<td>mntSetStreamHeader()</td>
<td>Sets the out-of-band stream attributes</td>
</tr>
<tr>
<td>mntSetStreamIOTimeout()</td>
<td>Sets the stream read or write request timeout value</td>
</tr>
<tr>
<td>mntTerminateStream()</td>
<td>Cancels a persistent stream</td>
</tr>
</tbody>
</table>

2.4.5. Message I/O Functions

The Direct Interface host library message I/O functions provide a set of services for generating, transferring, and accessing messages passed between the host and component instances.
## 2. Function Summary

### Table 6. Message I/O Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mntAllocateMMB( )</td>
<td>Allocates and clears an MMB (multiple message block)</td>
</tr>
<tr>
<td>mntClearMMB( )</td>
<td>Clears the command and reply message areas</td>
</tr>
<tr>
<td>mntCopyMMB( )</td>
<td>Copies the specified MMB (multiple message block)</td>
</tr>
<tr>
<td>mntEnumMpathDevice( )</td>
<td>Enumerates existing Mpath devices</td>
</tr>
<tr>
<td>mntEnumStrmDevice( )</td>
<td>Enumerates existing Stream devices</td>
</tr>
<tr>
<td>mntFreeMMB( )</td>
<td>Frees the specified MMB (multiple message block)</td>
</tr>
<tr>
<td>mntGetBoardsByAttr( )</td>
<td>Lists boards that match a list of caller-supplied attributes</td>
</tr>
<tr>
<td>mntGetMpathAddr( )</td>
<td>Gets the message path source address</td>
</tr>
<tr>
<td>mntGetTLSmmb( )</td>
<td>Gets the thread-local storage MMB</td>
</tr>
<tr>
<td>mntRegisterAsyncMessages( )</td>
<td>Registers a number of buffers for receipt of asynchronous messages</td>
</tr>
<tr>
<td>mntSendMessage( )</td>
<td>Asynchronously sends the message specified in the MMB (multiple message block)</td>
</tr>
<tr>
<td>mntSendMessageWait( )</td>
<td>Builds and sends a message then synchronously waits for the I/O completion.</td>
</tr>
<tr>
<td>qMsgVarFieldGet( )</td>
<td>Gets a number of typed fields from a message payload</td>
</tr>
<tr>
<td>qMsgVarFieldPut( )</td>
<td>Puts a number of typed fields into a message payload</td>
</tr>
</tbody>
</table>
2.4.6. Exit Notification Functions

The Direct Interface host library exit notification functions allow messages to be sent to registered addresses whenever an unexpected termination occurs. Two types of exit notification are possible: messages sent to an application upon sub-component failure and messages sent to the platform upon Mpath failure. The functions listed in Table 7 provide on/off switching of exit notification.

### Table 7. Exit Notification Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mntNotifyRegister( )</td>
<td>Enables sub-component exit notification to application</td>
</tr>
<tr>
<td>mntNotifyUnregister( )</td>
<td>Disables sub-component exit notification to application</td>
</tr>
<tr>
<td>mntSetExitNotify( )</td>
<td>Enables/disables Mpath exit notification to board</td>
</tr>
</tbody>
</table>
3. Function Reference

This chapter describes the Direct Interface functions and lists them alphabetically.

The following conventions are used throughout this chapter:

- New terms are shown in italic text.
- Important words or phrases are shown in bold text.
- Function names are shown in boldface with parentheses, such as `mntSendMessage()`.
- Data structure field names and function parameter names are shown in boldface, as in `timeout`.
- Messages are shown in italic text, such as `QResultComplete`.

**NOTE:** In this manual, the terms `MercMpath` and `Mpath` are used interchangeably. Similarly, the `MercStrm` and `Strm` device names are also used interchangeably.
mntAllocateMMB() allocates and clears a Message Block

| Name:       | LPMMB mntAllocateMMB(nCommandSize, nReplyCount, nReplyMaxSize) |
| Inputs:     | ULONG nCommandSize  ● bytes required                     |
|            | ULONG nReplyCount   ● expected replies                    |
|            | ULONG nReplyMaxSize ● size of replies                      |
| Outputs:    | None                                                           |
| Returns:    | LPMMB a pointer to an MMB                                    |
|             | NULL when specified MMB could not be allocated                |
| Includes:   | qhostlib.h                                                   |
| Category:   | message I/O function                                         |
| Mode:       | synchronous                                                  |

### Description

The `mntAllocateMMB()` function allocates and clears a Message Block, then returns its pointer. All specified byte sizes are rounded up to the next word boundary.

This function automatically sets the endian and version flags in the command QMsg header and sets the default MATCH_ON_SRC_ADDR flag in the MMB control block. This function also sets the command message payload size.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nCommandSize</td>
<td>number of bytes required for the command message. This number is normally equal to the command message header size plus the command message payload size. Maximum value for this parameter is MNT_MAX_COMMAND_SIZE. If set to zero, an empty message with no command to send is indicated. In this case, nReplyCount must be non-zero to receive asynchronous event messages. Note that the actual size recorded in the MMB is the command message payload size minus the QMsg size.</td>
</tr>
<tr>
<td>nReplyCount</td>
<td>expected number of replies from the DM3 platform. Maximum value for this parameter is MNT_MAX_REPLY_COUNT.</td>
</tr>
</tbody>
</table>
**allocates and clears a Message Block**  

`mntAllocateMMB()`  

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nReplyMaxSize</td>
<td>maximum size in bytes of all replies that might be received by the host. Each reply size is equal to the QMsg size plus the reply message (payload) size. For example, if you expect two replies with sizes MSG_REPLYSIZE_1 and MSG_REPLYSIZE_2, set this parameter to the sum of the two reply sizes. Maximum value for this parameter plus the nCommandSize value is MNT_MAX_CMD_PAYLOADSIZE.</td>
</tr>
</tbody>
</table>

Use the following macro to get the command QMsg message pointer:

```
MNT_GET_CMD_QMSG (LPMMB lpMMB, QMsgRef *pMsg)
```

Use the following macro to get the reply QMsg message pointer:

```
MNT_GET_REPLY_QMSG (LPMMB lpMMB, ULONG ReplyNumber, QMsgRef *pMsg)
```

To access the first reply message, the macro requires the command QMsg payload size to be defined.

To access reply messages other than the first reply message (`ReplyCount > 1`), the macro requires that the previous reply message “message size” or “payload size” be defined.

To create an MMB for an asynchronous event message, set the `CommandSize` parameter equal to zero. The `mntAllocateMMB()` function allocates an MMB consisting of a command QMsg with no payload and a reply section determined by `nReplyMaxSize`. An empty message is indicated by using the macro `MNT_SET_MMB_EMPTY_MSG(lpMMB)`.

### Cautions

None.

### Errors

- **ERROR_INVALID_PARAMETER**  
  - An invalid parameter was specified in the argument list.
mntAllocateMMB() allocates and clears a Message Block

ERROR_MNT_MMB_ALLOC_FAILED  • The MMB could not be allocated.

Result Messages

None.

See Also

• mntClearMMB()
• mntFreeMMB()
**Name:** BOOL mntAttachMercStream(hDevice, nBoardNumber, nModeFlags, lpMercStreamID, lpStreamSize, nTimeout, lpOverlapped)

**Inputs:**
- HANDLE hDevice: device handle
- ULONG nBoardNumber: board number
- USHORT nModeFlags: mode flags
- PULONG lpMercStreamID: stream ID
- PULONG lpStreamSize: stream size
- USHORT nTimeout: time to wait
- LPOVERLAPPED lpOverlapped: overlapped pointer

**Outputs:**
- PULONG lpMercStreamID: stream ID
- PULONG lpStreamSize: stream size

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** stream I/O function

**Mode:** asynchronous or synchronous

---

### Description

The `mntAttachMercStream()` function opens a stream and attaches it to a stream handle. If this function passes a valid, non-zero stream ID, it will be specified in the open-stream message to the DM3 board and that stream will be attached to the specified stream handle. The `nModeFlags` and `lpStreamSize` parameters specify the stream characteristics and `nBoardNumber` specifies the DM3 board on which the stream will be opened. When called synchronously, the locations pointed to by `lpMercStreamID` and `lpStreamSize` are filled in with the stream ID of the opened stream and the actual size of the stream, respectively.

Set the `nModeFlags` parameter by logically ORing the flags. Set only one flag in each of the following pairs:

- MNT_STREAM_FLAG_READ or MNT_STREAM_FLAG_WRITE
- MNT_STREAM_FLAG_NO_FLUSH or MNT_STREAM_FLAG_FLUSH

The `lpStreamSize` parameter requests the size of the stream buffer used by the stream device to transfer data. Available stream sizes are configured when the board is initialized. Therefore, the buffer size that the board allocates and actually uses for this stream might not be the same as what was requested; however, the actual size will always be greater than or equal to the requested size. Call the
`mntAttachMercStream()` opens a stream and attaches it to a stream handle

`mntGetStreamHeader()` function to obtain the actual stream buffer size (stored in the `actualSize` field of the PSTRM_HDR structure).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>stream device handle</td>
</tr>
<tr>
<td>nBoardNumber</td>
<td>board number</td>
</tr>
<tr>
<td>nModeFlags</td>
<td>stream attributes for this stream:</td>
</tr>
<tr>
<td></td>
<td>MNT_STREAM_FLAG_READ: read stream</td>
</tr>
<tr>
<td></td>
<td>MNT_STREAM_FLAG_WRITE: write stream</td>
</tr>
<tr>
<td></td>
<td>MNT_STREAM_FLAG_NO_FLUSH: use existing data in the read stream</td>
</tr>
<tr>
<td></td>
<td>MNT_STREAM_FLAG_FLUSH: flush the stream</td>
</tr>
<tr>
<td></td>
<td>MNT_STREAM_FLAG_IGNORE_HEADER: requests that a DM3 GStream is opened. (GStreams contain no header information.)</td>
</tr>
<tr>
<td></td>
<td>MNT_STREAM_FLAG_PERSISTENT: marks the stream for persistent mode operation</td>
</tr>
<tr>
<td>lpMercStreamID</td>
<td>points to an existing stream ID, or to zero if a new stream is to be opened. Returns the open stream ID if the call is synchronous.</td>
</tr>
<tr>
<td>lpStreamSize</td>
<td>pointer to the stream size requested. (Default = MNT_STREAMSIZE_NORMAL)</td>
</tr>
<tr>
<td></td>
<td>For the synchronous call, the actual size allocated is returned.</td>
</tr>
<tr>
<td>nTimeout</td>
<td>timeout (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

**Cautions**

1. The flush options, MNT_STREAM_FLAG_NO_FLUSH and MNT_STREAM_FLAG_FLUSH, apply to the action taken on the board, not by the host-side driver.
2. The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. *Calling Functions Asynchronously* for more details.
opens a stream and attaches it to a stream handle  mntAttachMercStream()

■ Errors

ERROR_ADAP_HDW_ERROR  • Board is not available to be initialized.
ERROR_INVALID_FUNCTION  • The stream handle specified is of the wrong type.
ERROR_INVALID_HANDLE  • An invalid handle was specified in the argument list.
ERROR_INVALID_PARAMETER  • An invalid parameter was specified in the argument list.
ERROR_MNT_ALREADY_OPEN  • The specified stream has been opened already.
ERROR_MNT_BAD_STREAM_ID  • An invalid stream ID was specified in the argument list.
ERROR_PIPE_BUSY  • A stream is already attached to the specified handle OR the specified stream is already open.

■ Result Messages

None.

■ See Also

- mntGetStreamHeader( )
- mntGetStreamInfo( )
- mntCheckStreamOrphans( )
mntCheckStreamOrphans()  

**Name:** BOOL mntCheckStreamOrphans(hDevice, lpOrphanBytes)  

**Inputs:**  
HANDLE hDevice  
- device handle  

**Outputs:**  
PULONG lpOrphanBytes  
- pointer  

**Returns:**  
TRUE if successful, FALSE if error  

**Includes:** qhostlib.h  

**Category:** stream I/O function  

**Mode:** synchronous

### Description

The `mntCheckStreamOrphans()` function checks for orphan bytes associated with the specified Stream device. Upon successful return, the location pointed to by `lpOrphanBytes` is filled in with the actual byte count of any orphan bytes. If it finds no orphan bytes, `lpOrphanBytes` will be zero.

During application development, you can use this function to clear the read stream before read calls are made. Call `mntCheckStreamOrphans()` in a loop until it returns a zero in `lpOrphanBytes` to empty the read stream buffers.

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a Stream device</td>
</tr>
<tr>
<td>lpOrphanBytes</td>
<td>orphan byte count</td>
</tr>
</tbody>
</table>

### Cautions

None.

### Errors

- **ERROR_INVALID_HANDLE**  
  - An invalid handle was specified in the argument list.

### Result Messages - None.

### See Also

None.
clears the command and reply message areas  

*Name:* BOOL mntClearMMB(lpMMB)

*Inputs:* LPMMB lpMMB  • pointer to MMB to cleared

*Outputs:* None

*Returns:* TRUE if successful, FALSE if error

*Includes:* qhostlib.h

*Category:* message I/O function

*Mode:* synchronous

---

**Description**

The `mntClearMMB()` function clears the command and reply message areas in the specified Message Block, but leaves some of the MMB header fields intact. This function should be called before the MMB is filled with command messages.

This function sets the endian and version flags in the command QMsg. This function sets the MATCH_ON_SRC_ADDR flag in the MMB control block. This function also sets the command payload size in the command QMsg.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpMMB</td>
<td>pointer to the MMB to be cleared</td>
</tr>
</tbody>
</table>

**Cautions**

None.

**Errors**

ERROR_INVALID_PARAMETER  • An invalid parameter was specified in the argument list.

**Result Messages**

None.

**See Also**

• `mntAllocateMMB()`  
• `mntFreeMMB()`
mntClusterActivate( ) activates an OUT-port connection

**Name:**
BOOL mntClusterActivate(hDevice, nTransID,
ClusterDesc, SCDesc, SCPortID, ClientDesc, nOptions,
nTimeout, lpMMB, lpOverlapped)

**Inputs:**
- HANDLE hDevice
- QTrans nTransID
- QCompDesc ClusterDesc
- QCompDesc SCDesc
- QPortDef SCPortID
- QCompDesc ClientDesc
- UCHAR nOptions
- USHORT nTimeout
- LPMMB lpMMB
- LPOVERLAPPED lpOverlapped

**Outputs:** None.

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h
mercdefs.h

**Category:** cluster management function

**Mode:** asynchronous or synchronous

---

### Description

The `mntClusterActivate()` function activates an OUT-port connection in a cluster. The main use of this function, from the host, is to activate an SCbus OUT-port in the SCbus resource. This allows data to flow from the TDM bus into any IN-port in the cluster that is connected to the SCbus OUT-port. If you call this function synchronously and it finds a standard error message with a QResultError type, it returns FALSE with an ERROR_MNT_MERCURY_KRNL error code.

This function’s parameters define a cluster’s address, an SCbus resource address, and a default behavior for Simple Talker protocol. The combination of cluster address, SCbus resource address, and port ID, uniquely identify the SCbus port.

This function allows the host to provide for the full Talker Protocol. Support for this protocol allows IN-ports inside the cluster to switch between the SCbus OUT-ports and the OUT-ports within the cluster. The `ClientDesc` parameter specifies the address to which to send any connection management messages. The `ClientDesc` parameter also models a component. The primary purpose of this parameter is to provide an address for the Talker Protocol messages needed to
activates an OUT-port connection \textit{mntClusterActivate( )}

manage the connection. If the client cannot support the Talker Protocol, you must set the \texttt{ClientDesc} parameter to NULL.

When the \texttt{ClientDesc} parameter is NULL, the \texttt{nOptions} parameter supports a simple Talker Protocol for the connection. If the \texttt{QCLUST\_AutoReject} option is set, and another OUT-port within the cluster requests to interrupt the SCbus’s OUT-port connection to an IN-port within the cluster, the request is rejected. If the \texttt{QCLUST\_AutoAccept} option is set and another cluster OUT-port requests to interrupt the connection, the connection is broken and the interrupting port is activated. When the interrupting cluster port ends the interruption, the client’s connection is reactivated.

If the activation fails (such as when another connection to the port is already active and cannot be interrupted) and the \texttt{QCLUST\_AutoReject} option is set, the connection is not activated, and the operation fails with an \texttt{ERROR\_MNT\_CLUSTER\_BUSY} error code.

The \textit{mntClusterActivate( )} function integrates the DM3 cluster switching model with non-DM3 switching systems.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{hDevice}</td>
<td>handle to a message path device returned from the \texttt{CreateFile( )} function</td>
</tr>
<tr>
<td>\texttt{nTransID}</td>
<td>transaction ID</td>
</tr>
<tr>
<td>\texttt{ClusterDesc}</td>
<td>cluster instance that owns the SCbus resource to activate</td>
</tr>
<tr>
<td>\texttt{SCDesc}</td>
<td>address of the SCbus resource that has the OUT-port</td>
</tr>
</tbody>
</table>
| \texttt{SCPortID} | SCbus resource port specifications. Use this to specify port direction:  
QPORT\_DIR\_IN: data transmitted to the TDM bus  
QPORT\_DIR\_OUT: data received from the TDM bus |
| \texttt{ClientDesc} | address of the client that owns and manages the time slots to be connected. This is the address to which all Talker protocol messages are sent. Specify as NULL if you use the Option parameter to determine connection behavior relative to the Talker Protocol. |
mntClusterActivate( ) activates an OUT-port connection

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nOptions</td>
<td>connection behavior for the Talker Protocol. Specify either of the following: QCLUST_AutoReject: automatically reject suspend requests. The connection cannot be interrupted by another resource. QCLUST_AutoAccept: never reject suspend requests. The connection can be interrupted by another resource.</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The mntClusterActivate( ) function causes the QClusterActivate kernel message (defined in mercdefs.h) to be sent. The QClusterActivate message size is defined as QClusterActivate_Size.

■ Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. Calling Functions Asynchronously for more details.

■ Errors

ERROR_ADAP_HDW_ERROR • Board is not available to be initialized.
ERROR_INVALID_HANDLE • An invalid handle was specified in the argument list.
ERROR_INVALID_PARAMETER • An invalid parameter was specified in the argument list.
ERROR_MNT_MERCURY_KRNL • See result message QResultError for details.
ERROR_MNT_MMB_ALLOC_FAILED • The MMB could not be allocated.
activates an OUT-port connection

mntClusterActivate()
**mntClusterAllocate( )**

**finds and allocates a cluster**

<table>
<thead>
<tr>
<th>Name:</th>
<th>BOOL mntClusterAllocate(hDevice, nTransID, lpClusterDesc, lpAttr, nTimeout, lpMMB, lpOverlapped)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
<td>HANDLE hDevice, QTrans nTransID, PQCompDesc lpClusterDesc, PQCompAttr lpAttr, USHORT nTimeout, LPMMB, LPOVERLAPPED lpOverlapped</td>
</tr>
<tr>
<td>Outputs:</td>
<td>PQCompDesc lpClusterDesc, LPMMB, LPOVERLAPPED lpOverlapped, LPOVERLAPPED lpOverlapped</td>
</tr>
<tr>
<td>Returns:</td>
<td>TRUE if successful, FALSE if error</td>
</tr>
<tr>
<td>Includes:</td>
<td>qhostlib.h</td>
</tr>
<tr>
<td>Category:</td>
<td>cluster management function</td>
</tr>
<tr>
<td>Mode:</td>
<td>asynchronous or synchronous</td>
</tr>
</tbody>
</table>

**Description**

The `mntClusterAllocate( )` function finds and allocates a cluster that has specific attributes. This function searches for a cluster that is partially specified by `lpClusterDesc` parameter and matches the attributes specified in the `lpAttr` list parameter. When the function returns (synchronously), the location pointed to by `lpClusterDesc` is filled in with the identifier of the cluster that was just allocated. If you call this function synchronously and it finds a standard error message with a QResultError type, it returns FALSE with an ERROR_MNT_MERCURY_KRNL error code.

The `lpClusterDesc` and `lpAttr` parameters together provide the information needed by the Resource Manager for allocating the desired cluster.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the <code>CreateFile( )</code> function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>lpClusterDesc</td>
<td>on input, partial cluster descriptor (must contain the destination board address); on output, cluster allocated.</td>
</tr>
<tr>
<td>lpAttr</td>
<td>an array of attributes, a key/value set.</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
</tbody>
</table>
finds and allocates a cluster

mntClusterAllocate( )

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The mntClusterAllocate() function causes the QClusterAllocate kernel message (defined in mercdefs.h) to be sent. The QClusterAllocate message size is defined as QClusterAllocate_Size.

■ Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. Calling Functions Asynchronously for more details.

■ Errors

ERROR_ADAP_HDW_ERROR • Board is not available to be initialized.
ERROR_INVALID_HANDLE • An invalid handle was specified in the argument list.
ERROR_INVALID_PARAMETER • An invalid parameter was specified in the argument list.
ERROR_MNT_MERCURY_KRNL • See result message QResultError for details.
ERROR_MNT_MMB_ALLOC_FAILED • The MMB could not be allocated.

■ Result Messages

QClusterResult

Successful completion. The body of this message contains a single data field which may be retrieved via the QClusterResult_get( ) macro:
theInstance (type QCompDesc): descriptor of the allocated cluster

QResultError

Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get( ) macro:
mntClusterAllocate()  

finds and allocates a cluster

**errorCode** (type Uint32): an unsigned integer that indicates the specific cause of the failure.

- **See Also**
  - mntClusterFree()
finds the cluster that owns an instance  \hspace{1cm} mntClusterByComp() 

| Name: | BOOL mntClusterByComp(hDevice, nTransID, CompDesc, lpClusterDesc, nTimeout, lpMMB, lpOverlapped) |
| Inputs: | HANDLE hDevice ● device handle | QTrans nTransID ● transaction ID | QCompDesc CompDesc ● instance in cluster | PQCompDesc lpClusterDesc ● cluster pointer | USHORT nTimeout ● time to wait | LPMMB lpMMB ● MMB pointer | LPOVERLAPPED lpOverlapped ● overlapped pointer |
| Outputs: | PQCompDesc lpClusterDesc ● cluster pointer |
| Returns: | TRUE if successful, FALSE if error |
| Includes: | qhostlib.h |
| Category: | cluster management function |
| Mode: | asynchronous or synchronous |

**Description**

The `mntClusterByComp()` function finds the cluster that owns an instance. This function finds which cluster is bound with the instance specified in the `CompDesc` parameter. If you call this function synchronously, upon successful return it fills in the location pointed to by `lpClusterDesc` with the address of the bound cluster. However, if this function finds a standard error message with a `QResultError` type, it returns FALSE with an `ERROR_MNT_MERCURY_KRNL` error code.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the <code>CreateFile()</code> function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>CompDesc</td>
<td>instance in a cluster</td>
</tr>
<tr>
<td>lpClusterDesc</td>
<td>pointer to the cluster found</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The `mntClusterByComp()` function causes the `QClusterByComp` kernel message (defined in `mercdefs.h`) to be sent. The `QClusterByComp` message size is defined as `QClusterByComp_Size`. 
mntClusterByComp( )  

finds the cluster that owns an instance

■ Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. Calling Functions Asynchronously for more details.

■ Errors

ERROR_ADAP_HDW_ERROR  
- Board is not available to be initialized.

ERROR_INVALID_HANDLE  
- An invalid handle was specified in the argument list.

ERROR_INVALID_PARAMETER  
- An invalid parameter was specified in the argument list.

ERROR_MNT_MERCURY_KRNL  
- See result message QResultError for details.

ERROR_MNT_MMB_ALLOC_FAILED  
- The MMB could not be allocated.

■ Result Messages

QClusterResult

Successful completion. The body of this message contains a single data field which may be retrieved via the QClusterResult_get( ) macro:

theInstance (type QCompDesc): descriptor of the allocated cluster

QResultError

Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get( ) macro:

errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

■ See Also

- mntClusterFind( )
finds a component with specific attributes

Name: BOOL mntClusterCompByAttr(hDevice, nTransID, ClusterDesc, lpAttr, lpCompDesc, nTimeout, lpMMB, lpOverlapped)

Inputs:
- HANDLE hDevice: device handle
- QTrans nTransID: transaction ID
- QCompDesc ClusterDesc: cluster to search
- PQCompAttr lpAttr: attributes list
- PQCompDesc lpCompDesc: component instance ptr
- USHORT nTimeout: time to wait
- LPMMB lpMMB: MMB pointer
- LPOVERLAPPED lpOverlapped: overlapped pointer

Outputs:
- PQCompDesc lpCompDesc: component instance ptr

Returns: TRUE if successful, FALSE if error

Includes: qhostlib.h

Category: cluster management function

Mode: asynchronous or synchronous

Description

The mntClusterCompByAttr( ) function finds a component with specific attributes. This function searches the cluster specified in the ClusterDesc parameter for a component that matches the attributes specified in the lpAttr parameter. If you call this function synchronously, upon successful return it fills in the location pointed to by lpCompDesc with the descriptor of the component instance that matches the specified attributes. However, if this function finds a standard error message with a QResultError type, it returns FALSE with an ERROR_MNT_MERCURY_KRNL error code.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the CreateFile() function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>ClusterDesc</td>
<td>cluster that owns the component instance</td>
</tr>
<tr>
<td>lpAttr</td>
<td>an array of attributes, a key/value set. If you specify only the Std_ComponentType attribute in the lpAttr parameter, this function finds a specific type of component instance in a cluster.</td>
</tr>
<tr>
<td>lpCompDesc</td>
<td>pointer to the component instance that matches lpAttr</td>
</tr>
</tbody>
</table>
\textit{mntClusterCompByAttr( )} \hspace{1cm} \textit{finds a component with specific attributes}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMBB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The \textit{mntClusterCompByAttr( )} function causes the \textit{QClusterCompByAttr} kernel message (defined in \textit{mercdefs.h}) to be sent. The \textit{QClusterCompByAttr} message size is defined as \textit{QClusterCompByAttr\_Size}.

\subsection*{Cautions}

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. \textit{Calling Functions Asynchronously} for more details.

\subsection*{Errors}

- \textbf{ERROR\_ADAP\_HDW\_ERROR} \hspace{1cm} \textit{Board is not available to be initialized.}
- \textbf{ERROR\_INVALID\_HANDLE} \hspace{1cm} \textit{An invalid handle was specified in the argument list.}
- \textbf{ERROR\_INVALID\_PARAMETER} \hspace{1cm} \textit{An invalid parameter was specified in the argument list.}
- \textbf{ERROR\_MNT\_MERCURY\_KRNL} \hspace{1cm} \textit{See result message \textit{QResultError} for details.}
- \textbf{ERROR\_MNT\_MBB\_ALLOC\_FAILED} \hspace{1cm} \textit{The MMB could not be allocated.}

\subsection*{Result Messages}

\textit{QComponentResult}

Successful completion. The body of this message contains a single data field which may be retrieved via the \textit{QComponentResult\_get( )} macro: \textbf{theInstance} (type QCompDesc): the fully qualified address of the component instance that has the specified attributes
finds a component with specific attributes \textit{mntClusterCompByAttr( )}

\texttt{QResultError}

Unsuccessful. The body of this message contains a single data field which may be retrieved via the \texttt{QResultError_get( )} macro:

\texttt{errorCode} (type \texttt{Uint32}): an unsigned integer that indicates the specific cause of the failure.

\bf{See Also}

- \texttt{mntClusterByComp( )}
**mntClusterConfigLock( )**

locks a specific cluster

**Name:** BOOL mntClusterConfigLock(hDevice, nTransID, ClusterDesc, nTimeout, lpMMB, lpOverlapped)

**Inputs:**
- HANDLE hDevice: device handle
- UTTrans nTransID: transaction ID
- QCompDesc ClusterDesc: target cluster
- USHORT nTimeout: time to wait
- LPMMB lpMMB: MMB pointer
- LPOVERLAPPED lpOverlapped: overlapped pointer

**Outputs:** None.

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** cluster management function

**Mode:** asynchronous or synchronous

#### Description

The mntClusterConfigLock( ) function locks a specific cluster to disable the automatic deallocation of its components in case the host address, such as the source address of the Mpath device, goes away. If you call this function synchronously and it finds a standard error message with a QResultError type, it returns FALSE with an ERROR_MNT_MERCURY_KRNL error code.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the CreateFile( ) function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>ClusterDesc</td>
<td>cluster to lock</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure that is large enough for the required command message</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The mntClusterConfigLock( ) function causes the QClusterLock kernel message (defined in mercdefs.h) to be sent. The QClusterLock message size is defined as QClusterLock_Size.

42
locks a specific cluster

mntClusterConfigLock( )

■ Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. Calling Functions Asynchronously for more details.

■ Errors

None.

■ Result Messages

QResultComplete
Successful completion. The message body contains no data fields.

QResultError
Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get() macro:
errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

■ See Also

• mntClusterConfigUnlock()
The `mntClusterConfigUnlock()` function unlocks a previously-locked cluster to re-enable the automatic deallocation of its components in case the host address, such as the source address of the Mpath device, goes away. If you call this function synchronously and it finds a standard error message with a QResultError type, it returns FALSE with an ERROR_MNT_MERCURY_KRNL error code.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the <code>CreateFile()</code> function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>ClusterDesc</td>
<td>cluster to unlock</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure that is large enough for the required command message</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The `mntClusterConfigUnlock()` function causes the `QClusterUnlock` kernel message (defined in `mercedes.h`) to be sent. The `QClusterUnlock` message size is defined as `QClusterUnlock_Size`. 
unlocks a previously-locked cluster \hspace{1cm} \texttt{mntClusterConfigUnlock()} \\

\section*{Cautions}

1. If you call this function synchronously, you must retrieve the passed parameters via a call to \texttt{mntGetTLSmmb()}.

2. The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. \textit{Calling Functions Asynchronously} for more details.

\section*{Errors}

None.

\section*{Result Messages}

\texttt{QClusterUnlockCmplt}

Successful completion. The reply message payload contains two data fields which may be retrieved via the \texttt{QClusterUnlockCmplt_get()} macro:

- \texttt{clusterUnlocked} (type UInt8): flag indicating cluster was unlocked
- \texttt{count} (type UInt8): the number of instances unlocked

\texttt{QResultError}

Unsuccessful. The body of this message contains a single data field which may be retrieved via the \texttt{QResultError_get()} macro:

- \texttt{errorCode} (type Uint32): an unsigned integer that indicates the specific cause of the failure.

\section*{See Also}

- \texttt{mntClusterConfigLock()}
mntClusterConnect( )  

interconnects the ports of two instances

**Name:**  
```c
BOOL mntClusterConnect(hDevice, nTransID, ClusterDesc, InstDesc1, PortID1, InstDesc2, PortID2, nTimeout, lpMMB, lpOverlapped)
```

**Inputs:**  
- HANDLE hDevice: device handle  
- QTrans nTransID: transaction ID  
- QCompDesc ClusterDesc: cluster instance  
- QCompDesc InstDesc1: component instance  
- QPortDef PortID1: type and port  
- QCompDesc InstDesc2: component instance  
- QPortDef PortID2: type and port  
- USHORT nTimeout: time to wait  
- LPMMB lpMMB: MMB pointer  
- LPOVERLAPPED lpOverlapped: overlapped pointer

**Outputs:**  
None.

**Returns:**  
TRUE if successful, FALSE if error

**Includes:**  
qhostlib.h

**Category:**  
cluster management function

**Mode:**  
asynchronous or synchronous

---

**Description**

The `mntClusterConnect()` function interconnects the ports of two instances. The primary purpose of this function is to allow the reconfiguration of a cluster.

This function connects the ports bound with the instance specified in the `InstDesc1` parameter to the ports bound with the instance specified in the `InstDesc2` parameter. If no types are specified, the port of each instance is connected as follows:

- If each instance has a primary IN- and OUT- port, the OUT-port of each instance is connected with the IN-port of the other, forming a full-duplex connection.
- If an instance has only one primary port, it is connected to the primary port of the other instance to create a half-duplex connection. Half-duplex connections are always OUT-port to IN-port.

You can use the `PortID1` and `PortID2` parameters to specify the type of connection to make. This is necessary if you need to make the connection between non-primary ports. The type parameter can specify any or all of the following port attributes:

46
interconnects the ports of two instances  \( \text{mntClusterConnect( )} \)

- Port type:
  - QPORT_TYPE_ECHO
  - QPORT_TYPERESOURCE
  - QPORT_TYPE_NETWORK
  - QPORT_TYPE_SCBUS
  - QPORT_TYPE_PRIMARY (default)

- Port direction:
  - QPORT_DIR_IN
  - QPORT_DIR_OUT (Specifying both IN and OUT results in full-duplex connection.)

- Port instance. Use if there are multiple instances of ports that have the same type and the same direction. The instance is a number in the range, 1 through 255.

**NOTE:** If the type parameter resolves to more than one port, as many connections as possible are made. For example this function makes a full-duplex connection if \( \text{PortID1} \) is a resource, \( \text{InstDesc1} \) has both IN- and OUT-port resources, and \( \text{InstDesc2} \) has a pair of IN- and OUT-ports.

For the host, this function can be used to interconnect:

- Ports in the same cluster
- Ports in separate clusters on the same board
- Ports in separate clusters on separate boards (future use)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the CreateFile( ) function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>ClusterDesc</td>
<td>cluster instance that ( \text{InstDesc1} ) occupies</td>
</tr>
<tr>
<td>InstDesc1</td>
<td>component instance connected to ( \text{InstDesc2} )</td>
</tr>
<tr>
<td>PortID1</td>
<td>type of port(s) to connect in ( \text{InstDesc1} ). This can be NULL for simple default connections.</td>
</tr>
<tr>
<td>InstDesc2</td>
<td>component instance connected to ( \text{InstDesc1} )</td>
</tr>
<tr>
<td>PortID2</td>
<td>type of port(s) to connect in ( \text{InstDesc2} ). This can be NULL for simple default connections.</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
</tbody>
</table>
mntClusterConnect( ) interconnects the ports of two instances

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The mntClusterConnect( ) function causes the QClusterConnect kernel message (defined in mercdefs.h) to be sent. The QClusterConnect message size is defined as QClusterConnect_Size.

■ Cautions

1. The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. Calling Functions Asynchronously for more details.

2. There are restrictions on how this function interconnects ports if used from the host.
   - Ports in the same cluster: No restrictions.
   - Ports in separate clusters, same board: Might fail if internal routing is not available and board is not configured to use a Timeslot Broker to request external connections.
   - Ports in separate clusters, separate boards: Fails if system is not configured to use a Timeslot Broker.

The connections parameters PortID1 and PortID2 can be specified as NULL. This results in the default connection between the primary ports of each instance.

■ Errors

- **ERROR_ADAP_HDW_ERROR**
  - Board is not available to be initialized.

- **ERROR_INVALID_HANDLE**
  - An invalid handle was specified in the argument list.

- **ERROR_INVALID_PARAMETER**
  - An invalid parameter was specified in the argument list.

- **ERROR_MNT_MERCURY_KRNL**
  - See result message QResultError for details.

- **ERROR_MNT_MMB_ALLOC_FAILED**
  - The MMB could not be allocated.
interconnects the ports of two instances \texttt{mntClusterConnect( )}

- **Result Messages**

  \texttt{QResultComplete}  
  Successful completion. The message body contains no data fields.

  \texttt{QResultError}  
  Unsuccessful. The body of this message contains a single data field which may be retrieved via the \texttt{QResultError} \_get( ) macro:  
  \texttt{errorCode} (type \texttt{Uint32}): an unsigned integer that indicates the specific cause of the failure.

- **See Also**
  - \texttt{mntClusterDisconnect( )}
### Description

The `mntClusterCreate()` function creates a new cluster and returns the cluster identifier. The null-terminated list of attributes specified in the `lpAttr` parameter associates the attributes with the cluster. If you call this function synchronously, upon successful return it fills in the location pointed to by `lpClusterDesc` with the descriptor of the newly created cluster. However, if this function finds a standard error message with a QResultError type, it returns FALSE with an `ERROR_MNT_MERCURY_KRNL` error code.

The cluster is created on the board specified in the `BrdAddr` field of the component descriptor defined in the `BrdAddr` parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hDevice</code></td>
<td>handle to a message path device returned from the <code>CreateFile()</code> function</td>
</tr>
<tr>
<td><code>nTransID</code></td>
<td>transaction ID</td>
</tr>
<tr>
<td><code>BrdAddr</code></td>
<td>component descriptor address of board on which to create this cluster</td>
</tr>
<tr>
<td><code>lpAttr</code></td>
<td>null-terminated list of attributes to assign to the new cluster</td>
</tr>
</tbody>
</table>
creates a new cluster \textit{mntClusterCreate( )}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{lpClusterDesc}</td>
<td>upon return, pointer to the cluster instance that has been created</td>
</tr>
<tr>
<td>\texttt{nTimeout}</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>\texttt{lpMMB}</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>\texttt{lpOverlapped}</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The \textit{mntClusterCreate( )} function causes the \textit{QClusterCreate} kernel message (defined in \textit{mercdefs.h}) to be sent. The \textit{QClusterCreate} message size is defined as \textit{QClusterCreate\_Size}.

\section*{Cautions}

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2, \textit{Calling Functions Asynchronously} for more details.

\section*{Errors}

\begin{itemize}
  \item \texttt{ERROR_ADAP_HDW_ERROR} \quad Board is not available to be initialized.
  \item \texttt{ERROR_INVALID_HANDLE} \quad An invalid handle was specified in the argument list.
  \item \texttt{ERROR_INVALID_PARAMETER} \quad An invalid parameter was specified in the argument list.
  \item \texttt{ERROR_MNT_MERCURY_KRNL} \quad See result message \textit{QResultError} for details.
  \item \texttt{ERROR_MNT_MMB_ALLOC_FAILED} \quad The MMB could not be allocated.
\end{itemize}

\section*{Result Messages}

\textit{QClusterResult}

Successful completion. The body of this message contains a single data field which may be retrieved via the \texttt{QClusterResult\_get( )} macro:
\begin{itemize}
  \item \texttt{theInstance} (type \texttt{QCompDesc}): descriptor of the created cluster
\end{itemize}
mntClusterCreate() creates a new cluster

QResultError

Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get() macro:

errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

See Also

- mntClusterDestroy()
**deactivates connections**

**mntClusterDeactivate()**

**Name:**

BOOL mntClusterDeactivate(hDevice, nTransID,
ClusterDesc, SCDesc, SCPortID, nTimeout, lpMMB,
lpOverlapped)

**Inputs:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| HANDLE  | handle to a message path device returned from the
| hDevice | CreateFile() function         |
| QTrans  | transaction ID                |
| QCCompDesc | cluster instance          |
| QCCompDesc | SCbus component              |
| QPortDef | ID of resource port          |
| USHORT  | time to wait                  |
| LPMMB   | MMB pointer                   |
| LPOVERLAPPED | overlapped pointer         |

**Outputs:**

None.

**Returns:**

TRUE if successful, FALSE if error

**Includes:**

qhostlib.h

**Category:**

cluster management function

**Mode:**

asynchronous or synchronous

---

### Description

The mntClusterDeactivate() function deactivates connections that have specified OUT-ports. The main use of this function from the host is to disable data flowing from an SCbus OUT-port to IN-ports inside a cluster. This function informs the kernel that the TDM data flowing out of the SCbus OUT-port has stopped, and that the Talker protocol should be disabled for this port.

This function’s parameters define a cluster address and an SCbus resource address. This function disables any Simple Talker protocol default behavior that had been previously enabled through the mntClusterActivate() function. The SCbus port is uniquely identified by the combination of cluster address, SCbus resource address, and SCbus resource port.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the CreateFile() function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>ClusterDesc</td>
<td>cluster instance that owns the SCbus resource to deactivate</td>
</tr>
<tr>
<td>SCDesc</td>
<td>address of the SCbus resource in the cluster</td>
</tr>
</tbody>
</table>

53
mntClusterDeactivate( ) deactivates connections

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCPortID</td>
<td>specific SCbus resource port (type and direction)</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The mntClusterDeactivate( ) function causes the QClusterDeactivate kernel message (defined in meredefs.h) to be sent. The QClusterDeactivate message size is defined as QClusterDeactivate_Size.

**Cautions**

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. Calling Functions Asynchronously for more details.

**Errors**

- ERROR_ADAP_HDW_ERROR: Board is not available to be initialized.
- ERROR_INVALID_HANDLE: An invalid handle was specified in the argument list.
- ERROR_INVALID_PARAMETER: An invalid parameter was specified in the argument list.
- ERROR_MNT_MERCURY_KRNL: See result message QResultError for details.
- ERROR_MNT_MMB_ALLOC_FAILED: The MMB could not be allocated.

**Result Messages**

QResultComplete
Successful completion. The message body contains no data fields.
deactivates connections

mntClusterDeactivate()

QResultError
Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get() macro: 
errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

See Also

- mntClusterActivate()
**mntClusterDestroy( )** destroys an empty cluster

**Name:**
BOOL mntClusterDestroy(hDevice, nTransID, ClusterDesc, nTimeout, lpMMB, lpOverlapped )

**Inputs:**
- HANDLE hDevice
- QTrans nTransID
- QCompDesc ClusterDesc
- USHORT nTimeout
- LPMMB lpMMB
- LPOVERLAPPED lpOverlapped

**Outputs:** None.

**Returns:**
TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** cluster management function

**Mode:** asynchronous or synchronous

---

### Description

The **mntClusterDestroy( )** function destroys an empty cluster.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the <strong>CreateFile( )</strong> function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>ClusterDesc</td>
<td>address of the cluster to delete</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The **mntClusterDestroy( )** function causes the **QClusterDestroy** kernel message (defined in **mercdefs.h**) to be sent. The **QClusterDestroy** message size is defined as **QClusterDestroy_Size**.

### Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. *Calling Functions Asynchronously* for more details.
**destroys an empty cluster**

`mntClusterDestroy()`

---

### Errors

- **ERROR_ADAP_HDW_ERROR**
  - Board is not available to be initialized.

- **ERROR_INVALID_HANDLE**
  - An invalid handle was specified in the argument list.

- **ERROR_INVALID_PARAMETER**
  - An invalid parameter was specified in the argument list.

- **ERROR_MNT_MERCURY_KRNL**
  - See result message `QResultError` for details.

- **ERROR_MNT_MMB_ALLOC_FAILED**
  - The MMB could not be allocated.

---

### Result Messages

**QResultComplete**

Successful completion. The message body contains no data fields.

**QResultError**

Unsuccessful. The body of this message contains a single data field which may be retrieved via the `QResultError_get()` macro:

- **errorCode** (type Uint32): an unsigned integer that indicates the specific cause of the failure.

---

### See Also

- `mntClusterCreate()`
**mntClusterDisconnect()** breaks an existing connection between ports

**Name:**

```c
BOOL mntClusterDisconnect(hDevice, nTransID, ClusterDesc, InstDesc1, PortID1, InstDesc2, PortID2, nTimeout, lpMMB, lpOverlapped )
```

**Inputs:**

- **hDevice**
  - handle to a message path device returned from the `CreateFile()` function
- **nTransID**
  - transaction ID
- **ClusterDesc**
  - cluster instance to which to send the disconnect message
- **InstDesc1**
  - component instance
- **PortID1**
  - type and port
- **InstDesc2**
  - component instance
- **PortID2**
  - type and port
- **nTimeout**
  - time to wait
- **lpMMB**
  - MMB pointer
- **lpOverlapped**
  - overlapped pointer

**Outputs:** None.

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** cluster management function

**Mode:** asynchronous or synchronous

---

**Description**

The `mntClusterDisconnect()` function breaks an existing connection between ports that are bound with the instances specified in the `InstDesc1` and `InstDesc2` parameters. If no types are specified, each primary port that is connected to the instance specified in the `InstDesc2` parameter is disconnected.

You can use the `PortID1` and `PortID2` parameters to specify the types of connections to break. The `PortID1` parameter specifies the type of ports defined in the `InstDesc1` parameter. The `PortID2` parameter specifies the type of ports defined in `InstDesc2` parameter. You should set both of these to NULL.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hDevice</strong></td>
<td>handle to a message path device returned from the <code>CreateFile()</code> function</td>
</tr>
<tr>
<td><strong>nTransID</strong></td>
<td>transaction ID</td>
</tr>
<tr>
<td><strong>ClusterDesc</strong></td>
<td>cluster instance to which to send the disconnect message</td>
</tr>
<tr>
<td><strong>InstDesc1</strong></td>
<td>component instance to disconnect from <code>InstDesc2</code></td>
</tr>
</tbody>
</table>
breaks an existing connection between ports  

**mntClusterDisconnect()**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortID1</td>
<td>type of port(s) to disconnect in <strong>InstDesc1</strong>. This should be NULL.</td>
</tr>
<tr>
<td>InstDesc2</td>
<td>component instance to disconnect from <strong>InstDesc1</strong></td>
</tr>
<tr>
<td>PortID2</td>
<td>type of port(s) to disconnect in <strong>InstDesc2</strong>. This should be NULL.</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The **mntClusterDisconnect()** function causes the **QClusterDisconnect** kernel message (defined in **mercdefs.h**) to be sent. The **QClusterDisconnect** message size is defined as QClusterDisconnect_Size.

**Cautions**

1. You should specify the **PortID1** and **PortID2** parameters as NULL.
2. The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. **Calling Functions Asynchronously** for more details.

**Errors**

- **ERROR_ADAP_HDW_ERROR**  
  - Board is not available to be initialized.
- **ERROR_INVALID_HANDLE**  
  - An invalid handle was specified in the argument list.
- **ERROR_INVALID_PARAMETER**  
  - An invalid parameter was specified in the argument list.
- **ERROR_MNT_MERCURY_KRNL**  
  - See result message **QResultError** for details.
- **ERROR_MNT_MMB_ALLOC_FAILED**  
  - The MMB could not be allocated.
**mntClusterDisconnect()** breaks an existing connection between ports

- **Result Messages**

  **QResultComplete**
  Successful completion. The message body contains no data fields.

  **QResultError**
  Unsuccessful. The body of this message contains a single data field which may be retrieved via the `QResultError_get()` macro:
  - **errorCode** (type Uint32): an unsigned integer that indicates the specific cause of the failure.

- **See Also**
  - `mntClusterConnect()`
The `mntClusterFind()` function finds a cluster that has specific attributes. If you call this function synchronously, upon successful return it fills in the location pointed to by `lpClusterDesc` with the descriptor of the cluster that matches the specified attributes. If you call this function synchronously and it finds a standard error message with a QResultError type, it returns FALSE with an ERROR_MNT_MERCURY_KRNL error code.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the <code>CreateFile()</code> function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>lpClusterDesc</td>
<td>on input, cluster descriptor through which to search (must contain the destination board address); on output, descriptor of the found cluster.</td>
</tr>
<tr>
<td>lpAttrs</td>
<td>an array of attributes, a key or value set</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure that is large enough for the required command message</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>
mntClusterFind( ) finds a cluster that has specific attributes

The mntClusterFind( ) function causes the QClusterFind kernel message (defined in mercdefs.h) to be sent. The QClusterFind message size is defined as QClusterFind_Size.

■ Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. Calling Functions Asynchronously for more details.

■ Errors

None.

■ Result Messages

QClusterResult

Successful completion. The body of this message contains a single data field which may be retrieved via the QClusterResult_get( ) macro:

theInstance (type QCompDesc): descriptor of the allocated cluster

QResultError

Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get( ) macro:

errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

■ See Also

• mntClusterByComp( )
**releases an allocated cluster**

*mntClusterFree()*

<table>
<thead>
<tr>
<th>Name:</th>
<th>BOOL mntClusterFree(hDevice, nTransID, ClusterDesc, nTimeout, lpMMB, lpOverlapped)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
<td>HANDLE hDevice, nTransID, ClusterDesc, nTimeout, lpMMB, lpOverlapped</td>
</tr>
<tr>
<td></td>
<td>• device handle, transaction ID, cluster to free, time to wait, MMB pointer, overlapped pointer</td>
</tr>
<tr>
<td>Outputs:</td>
<td>None.</td>
</tr>
<tr>
<td>Returns:</td>
<td>TRUE if successful, FALSE if error</td>
</tr>
<tr>
<td>Includes:</td>
<td>qhostlib.h</td>
</tr>
<tr>
<td>Category:</td>
<td>cluster management function</td>
</tr>
<tr>
<td>Mode:</td>
<td>asynchronous or synchronous</td>
</tr>
</tbody>
</table>

---

**Description**

The `mntClusterFree()` function releases an allocated cluster. If you call this function synchronously and it finds a standard error message with a QResultError type, it returns FALSE with an ERROR_MNT_MERCURY_KRNL error code.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the <code>CreateFile()</code> function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>ClusterDesc</td>
<td>cluster to be freed</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The `mntClusterFree()` function causes the `QClusterFree` kernel message (defined in `mercdefs.h`) to be sent. The `QClusterFree` message size is defined as QClusterFree_Size.

---

**Cautions**

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. *Calling Functions Asynchronously* for more details.
**mntClusterFree( )** releases an allocated cluster

- **Errors**
  None.

- **Result Messages**
  
  **QResultComplete**
  Successful completion. The message body contains no data fields.

  **QResultError**
  Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get() macro:
  
  **errorCode** (type Uint32): an unsigned integer that indicates the specific cause of the failure.

- **See Also**
  - mntClusterAllocate( )
finds the time slots assigned to a port  \hspace{3cm} \textit{mntClusterSlotInfo( )}

**Name:** \hspace{2cm} BOOL \text{mntClusterSlotInfo}(hDevice, nTransID, ClusterDesc, SCDesc, SCPortID, lpClusterInfo, nSlots, lpSlots, nTimeout, lpMMB, lpOverlapped)

**Inputs:**
- HANDLE \text{hDevice} \hspace{2cm} \text{device handle}
- QTrans \text{nTransID} \hspace{2cm} \text{transaction ID}
- QCompDesc \text{ClusterDesc} \hspace{2cm} \text{cluster instance}
- QCompDesc \text{SCDesc} \hspace{2cm} \text{SCbus resource}
- QPortDef \text{SCPortID} \hspace{2cm} \text{port type}
- QClusterSlotInfoResult_t *lpClusterInfo \hspace{2cm} \text{cluster data}
- BYTE \text{nSlots} \hspace{2cm} \text{time slots number}
- PUSHORT lpSlots \hspace{2cm} \text{time slots array}
- USHORN \text{nTimeout} \hspace{2cm} \text{time to wait}
- LPMMB \text{lpMMB} \hspace{2cm} \text{MMB pointer}
- LPOVERLAPPED \text{lpOverlapped} \hspace{2cm} \text{overlapped pointer}

**Outputs:** None.

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** cluster management function

**Mode:** asynchronous or synchronous

---

**Description**

The \textit{mntClusterSlotInfo( )} function finds the time slots assigned to a port. The \text{SCDesc} and \text{SCPortID} parameters define the port. If you call this function synchronously and it finds a standard error message with a QResultError type, it returns FALSE with an ERROR_MNT_MERCURY_KRNL error code.

**Parameter**  \hspace{1cm} **Description**

\text{hDevice} \hspace{2cm} handle to a message path device returned from the \textit{CreateFile( )} function

\text{nTransID} \hspace{2cm} transaction ID

\text{ClusterDesc} \hspace{2cm} cluster instance that owns the port to which the time slots have been assigned

\text{SCDesc} \hspace{2cm} SCbus resource that owns the SCbus ports

\text{SCPortID} \hspace{2cm} SCbus resource port type and direction:

\begin{itemize}
  \item QSCBUS_PORT_IN
  \item QSCBUS_PORT_OUT
\end{itemize}
**mntClusterSlotInfo( ) finds the time slots assigned to a port**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpClusterInfo</td>
<td>returned cluster information returned (by a synchronous call only). The structure includes the width that indicates the actual number of time slots allocated to this resource.</td>
</tr>
<tr>
<td>nSlots</td>
<td>number of time slots allocated in the array specified by the lpSlots parameter</td>
</tr>
<tr>
<td>lpSlots</td>
<td>array of time slots allocated to the SCbus resource (by a synchronous call only)</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The **mntClusterSlotInfo( )** function causes the **QClusterSlotInfo** kernel message (defined in *mercdefs.h*) to be sent. The **QClusterSlotInfo** message size is defined as QClusterSlotInfo_Size.

**Cautions**

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. *Calling Functions Asynchronously* for more details.

**Errors**

- **ERROR_ADAP_HDW_ERROR**
  - Board is not available to be initialized.
- **ERROR_INVALID_HANDLE**
  - An invalid handle was specified in the argument list.
- **ERROR_INVALID_PARAMETER**
  - An invalid parameter was specified in the argument list.
- **ERROR_MNT_MERCURY_KRNL**
  - See result message **QResultError** for details.
- **ERROR_MNT_MMB_ALLOC_FAILED**
  - The MMB could not be allocated.
Result Messages

QClusterSlotInfoResult

Successful completion. The body of this message contains a variable-size payload which includes five fixed data fields followed by a variable-length list of data items. The QClusterSlotInfoResult_get( ) macro is used to extract the fixed fields into a data structure of type QClusterSlotInfoResult_t, which contains the following elements:

- instDesc (type QCompDesc): descriptor of the cluster
- portId (type Uint24): port ID the information pertains to
- width (type UInt8): number of timeslots used; this value also indicates the number of SlotId fields in the variable-length list.
- encoding (type UInt8): type of encoding used on this port
- idlePattern (type UInt8): type of idle pattern used on this port

The remainder of the message body contains a variable-length list of data fields with width members. Use qMsgVarFieldGet( ) with an initial offset of QClusterSlotInfoResult_Size to retrieve these values.

- SlotId (type Uint16): an SCbus timeslot number

QResultError

Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get( ) macro:

- errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

See Also

- mntClusterTSAssign( )
- mntClusterTSUnassign( )
**mntClusterTSAssign( )** assigns time slots to a cluster’s SCbus resource

**Name:** BOOL mntClusterTSAssign(hDevice, nTransID, ClusterDesc, SCDesc, SCPortID, nWidth, nEncoding, nIdle, lpSlotId, nTimeout, lpMMB, lpOverlapped)

**Inputs:**
- HANDLE hDevice / device handle
- QTrans nTransID / transaction ID
- QCompDesc ClusterDesc / cluster instance
- QCompDesc SCDesc / SCbus resource
- QPortDef SCPortID / port type
- UCHAR nWidth / time slots number
- UCHAR nEncoding / PCM encoding
- UCHAR nIdle / PCM idle pattern
- PUSHORT lpSlotId / timeslot list pointer
- USHORT nTimeout / time to wait
- LPMMB lpMMB / MMB pointer
- LPOVERLAPPED lpOverlapped / overlapped pointer

**Outputs:** None.

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** cluster management function

**Mode:** asynchronous or synchronous

**Description**

The `mntClusterTSAssign( )` function assigns time slots to a cluster’s SCbus resource. This function allows Resource and Network OUT-ports to transmit TDM data to the SCbus, and it allows Resource and Network IN-ports to receive data from the TDM bus. This function finds which cluster is bound with the instance specified in the `ClusterDesc` parameter.

This function’s parameters define a cluster, an SCbus resource, and a set time slots. This function establishes a logical link between the logical SCbus IN or OUT-ports and a set of TDM time slots. Ports in the cluster that are transmitting data to the SCbus IN-port have data transmitted to the SCbus after this function has been called. (The connections are activated within the cluster through Talker Protocol.). Ports in the cluster that are connected to the SCbus OUT-port receive data from that port after the `mntClusterActivate( )` function has been called to activate the connection. The SCbus OUT-ports need the host to control Talker Protocol for the port.
assigns time slots to a cluster’s SCbus resource  \textit{mntClusterTSAssign( )}

You can use the \textit{mntClusterTSUnassign( )} function to unassign a time slot and stop transmission to and reception from the TDM bus.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{hDevice}</td>
<td>handle to a message path device returned from the \texttt{CreateFile( )} function</td>
</tr>
<tr>
<td>\texttt{nTransID}</td>
<td>transaction ID</td>
</tr>
<tr>
<td>\texttt{ClusterDesc}</td>
<td>cluster instance that owns the port to which to connect the time slots</td>
</tr>
<tr>
<td>\texttt{SCDesc}</td>
<td>SCbus resource that owns the SCbus ports</td>
</tr>
<tr>
<td>\texttt{SCPortID}</td>
<td>SCbus resource port type and direction</td>
</tr>
<tr>
<td>QSCBUS_PORT_IN:</td>
<td>data transmitted to the TDM bus</td>
</tr>
<tr>
<td>QSCBUS_PORT_OUT:</td>
<td>data received from the TDM bus</td>
</tr>
<tr>
<td>\texttt{nWidth}</td>
<td>number of time slots with which to link. This must match the width of the SCbus resource width attribute.</td>
</tr>
<tr>
<td>\texttt{nEncoding}</td>
<td>PCM encoding used for data on the time slots:</td>
</tr>
<tr>
<td>QSCBUS_ENCODING_ALAW:</td>
<td>sets A-Law encoding</td>
</tr>
<tr>
<td>QSCBUS_ENCODING_MULAW:</td>
<td>sets (\mu)-Law encoding</td>
</tr>
<tr>
<td>\texttt{nIdle}</td>
<td>idle pattern used on the time slots:</td>
</tr>
<tr>
<td>QSCBUS_IDLE_ALAW:</td>
<td>sets A-Law idle pattern</td>
</tr>
<tr>
<td>QSCBUS_IDLE_MULAW:</td>
<td>sets (\mu)-Law idle pattern</td>
</tr>
<tr>
<td>\texttt{lpSlotId}</td>
<td>list of time slots numbers that identify the time slots to be connected. Use the \texttt{nWidth} parameter to specify the number of time slots.</td>
</tr>
<tr>
<td>\texttt{nTimeout}</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>\texttt{lpMMB}</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>\texttt{lpOverlapped}</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The \textit{mntClusterTSAssign( )} function causes the \textit{QClusterSlotAssign} kernel message (defined in \texttt{mercreds.h}) to be sent. The \textit{QClusterSlotAssign} message size is defined as QClusterSlotAssign\_Size.
mntClusterTSAssign() assigns time slots to a cluster’s SCbus resource

■ Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. Calling Functions Asynchronously for more details.

■ Errors

ERROR_ADAP_HDW_ERROR • Board is not available to be initialized.

ERROR_INVALID_HANDLE • An invalid handle was specified in the argument list.

ERROR_INVALID_PARAMETER • An invalid parameter was specified in the argument list.

ERROR_MNT_MERCURY_KRNL • See result message QResultError for details.

ERROR_MNT_MMB_ALLOC_FAILED • The MMB could not be allocated.

■ Result Messages

QResultComplete
Successful completion. The message body contains no data fields.

QResultError
Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get() macro:
errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

■ See Also

• mntClusterTSUnassign()
unassigns a timeslot from an SCbus resource  

mntClusterTSUnassign()

Name:  

BOOL mntClusterTSUnassign(hDevice, nTransID, ClusterDesc, SCDesc, SCPortID, nTimeout, lpMMB, lpOverlapped)

Inputs:  

HANDLE hDevice  
QTrans nTransID  
QCompDesc ClusterDesc  
QCompDesc SCDesc  
QPortDef SCPortID  
USHORT nTimeout  
LPMMB lpMMB  
LPOVERLAPPED lpOverlapped

Outputs:  
None.

Returns:  
TRUE if successful, FALSE if error

Includes:  
qhostlib.h

Category:  
cluster management function

Mode:  
asynchronous or synchronous

Description

The mntClusterTSUnassign() function unassigns a timeslot from an SCbus resource. This removes the ability of a resource to transmit to or receive from the TDM bus.

This function’s parameters define a cluster and an SCbus resource. The function removes the link between the logical SCbus In or OUT ports and a set of physical TDM bus time slots. Ports in the cluster that are transmitting data to the SCbus IN-port no longer have data transmitted to the TDM bus after this function is called. Ports in the cluster that are connected to the SCbus OUT-port no longer receive data from that port. Unassigning the SCbus OUT-port has the effect of calling the mntClusterDeactivate() function before the unassignment takes place.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the CreateFile() function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>ClusterDesc</td>
<td>cluster instance that owns the port from which to disconnect the time slots</td>
</tr>
<tr>
<td>SCDesc</td>
<td>SCbus resource that owns the SCbus ports</td>
</tr>
</tbody>
</table>
mntClusterTSUnassign() unassigns a timeslot from an SCbus resource

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCPortID</td>
<td>SCbus resource port specifications. Use this to specify port direction:</td>
</tr>
<tr>
<td></td>
<td>QPORT_DIR_IN: data transmitted to the TDM bus</td>
</tr>
<tr>
<td></td>
<td>QPORT_DIR_OUT: data received from the TDM bus</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The `mntClusterTSUnassign()` function causes the `QClusterSlotUnassign` kernel message (defined in `mercdefs.h`) to be sent. The `QClusterSlotUnassign` message size is defined as `QClusterSlotUnassign_Size`.

- **Cautions**

  The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. *Calling Functions Asynchronously* for more details.

- **Errors**

  - `ERROR_ADAP_HDW_ERROR`  
    - Board is not available to be initialized.
  - `ERROR_INVALID_HANDLE`  
    - An invalid handle was specified in the argument list.
  - `ERROR_INVALID_PARAMETER`  
    - An invalid parameter was specified in the argument list.
  - `ERROR_MNT_MERCURY_KRNL`  
    - See result message `QResultError` for details.
  - `ERROR_MNT_MMB_ALLOC_FAILED`  
    - The MMB could not be allocated.

- **Result Messages**

  - `QResultComplete`  
    - Successful completion. The message body contains no data fields.
unassigns a timeslot from an SCbus resource

QResultError
Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get( ) macro:

errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

See Also

• mntClusterTSAssign( )
**mntCompAllocate( )** reserves and locks a specific component instance

- **Inputs:**
  - `HANDLE hDevice` - device handle
  - `QTrans nTransID` - transaction ID
  - `PQCompDesc lpInstance` - component instance
  - `PQCompAttr pAttrs` - attributes array
  - `USHORT ClusterDesc` - cluster to allocate into
  - `USHORT nTimeout` - time to wait
  - `LPMMB lpMMB` - MMB pointer
  - `LPOVERLAPPED lpOverlapped` - overlapped pointer

- **Outputs:**
  - `QCompDesc lpInstance` - component instance

- **Returns:**
  - TRUE if successful, FALSE if error

- **Includes:**
  - `qhostlib.h`

- **Category:** component management function

- **Mode:** asynchronous or synchronous

### Description

The `mntCompAllocate( )` function reserves and locks a specific component instance. This function allocates a component instance that matches the requirements specified in the `lpInstance` and `pAttrs` parameters.

If you call this function asynchronously, the fully qualified and allocated component instance is returned in the MMB reply message.

If you call this function synchronously, upon successful return it fills in the location pointed to by `lpInstance` with the descriptor of the allocated component instance. However, if this function receives a standard error message with a QResultError type, it returns FALSE with the ERROR_MNT_MERCURY_KRNL error code.

The `lpInstance` and `pAttrs` parameters together provide the information needed for the Resource Manager to select an instance of the desired component.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hDevice</code></td>
<td>handle to a message path device returned from the <code>CreateFile( )</code> function</td>
</tr>
<tr>
<td><code>nTransID</code></td>
<td>transaction ID</td>
</tr>
</tbody>
</table>
reserves and locks a specific component instance  

mntCompAllocate()

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpInstance</td>
<td>on input, desired component instance to reserve and lock; on output, descriptor of the allocated component instance.</td>
</tr>
<tr>
<td>pAttrs</td>
<td>an array of component attributes, a key/value set.</td>
</tr>
<tr>
<td>ClusterDesc</td>
<td>cluster in which to allocate the component</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The mntCompAllocate() function causes the QCompInstAllocate kernel message (defined in mercdefs.h) to be sent. The QCompInstAllocate message size is defined as QCompInstAllocate_Size.

■ Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2, Calling Functions Asynchronously for more details.

■ Errors

ERROR_ADAP_HDW_ERROR  • Board is not available to be initialized.
ERROR_INVALID_HANDLE  • An invalid handle was specified in the argument list.
ERROR_INVALID_PARAMETER  • An invalid parameter was specified in the argument list.
ERROR_MNT_MERCURY_KRNL  • See result message QResultError for details.
ERROR_MNT_MMB_ALLOC_FAILED  • The MMB could not be allocated.
mntCompAllocate( ) reserves and locks a specific component instance

- Result Messages

If you call this function synchronously, it first examines the reply message to check for successful component allocation. If it returns TRUE, it then returns the component address in the lpInstance parameter.

If you call this function asynchronously, and it returns FALSE, the GetLastError( ) function should retrieve the ERROR_IO_PENDING code. In this case, you need to call one of the Win32 API wait functions, such as WaitForMultipleObjects( ). After the wait function returns, call the GetOverlappedResult( ) function to get the results of the operation.

QComponentResult
Successful completion. The body of this message contains a single data field which may be retrieved via the QComponentResult_get( ) macro:
theInstance (type QCompDesc): the fully qualified address of the component instance that has the specified attributes

QResultError
Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get( ) macro:
errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

- See Also
- mntCompFree( )
finds a component

mntCompFind( )

Name: BOOL mntCompFind(hDevice, nTransID, lpInstance, pAttrs, nTimeout, lpMMB, lpOverlapped)

Inputs:
- HANDLE hDevice
- QTrans nTransID
- PQCompDesc lpInstance
- PQCompAttr pAttrs
- USHORT nTimeout
- LPMMB lpMMB
- LPOVERLAPPED lpOverlapped

Outputs:
- PQCompDesc lpInstance

Returns: TRUE if successful, FALSE if error

Includes: qhostlib.h

Category: component management function
Mode: asynchronous or synchronous

Description

The mntCompFind( ) function finds a component. The function returns a component address that matches the requirements specified in the lpInstance and pAttrs parameters.

If you call this function asynchronously, you need to examine the reply message contained in the MMB. If no qualified component is found, an error is indicated in the message.

If you call this function synchronously, upon successful return it fills in the location pointed to by lpInstance with the descriptor of the component instance that matches the specified attributes. However, if this function finds a standard error message with a QResultError type, it returns FALSE with an ERROR_MNT_MERCURY_KRNL error code.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the CreateFile( ) function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>pAttrs</td>
<td>an array of component attributes, a key/value set.</td>
</tr>
<tr>
<td>lpInstance</td>
<td>on input, desired component instance on output, component instance that was found.</td>
</tr>
</tbody>
</table>
mntCompFind() finds a component

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The mntCompFind() function causes the QCompFind kernel message (defined in mercdefs.h) to be sent. The QCompFind message size is defined as QCompFind_Size.

A component instance descriptor has the following format:

```c
typedef struct {
    UInt16 node;        /* reserved for node address Not currently used*/
    UInt8 board;        /* board ID within host */
    UInt8 processor;    /* processor identifier */
    UInt8 component;    /* component identifier */
    UInt8 instance;     /* instance number (or task id) */
} QCompDesc;
```

A fully specified component instance contains non-nil values in the processor, component, and instance fields. The node and board fields are always ignored. lpInstance should contain only a partially specified address with at least the processor and possibly the component specified.

lpInstance should be partially specified so the instance field is set to QCOMP_I_NIL; if it is not set to nil, it is ignored. The component field is normally set to QCOMP_C_NIL if the request is intended to find a component matching the specified attributes, but it can contain a component identifier. If the component field is non-nil, the function completes successfully if the specified component has the attributes specified; otherwise, an error is returned. The processor field also can be set to its nil value (QCOMP_P_NIL). If lpInstance is not specified, the selection is based completely on the attribute defined in pAttrs.

The pAttrs argument references an array of QCompAttr structures. Attributes are used to identify the capabilities available in components. They can be used to differentiate components that perform the same type of function, such as audio coders which support different coding algorithms.
finds a component

mntCompFind()  

A value of type QCompAttr is a structure of the format:

```c
typedef struct
{
    UInt32 key;    /* A key defining the type of attribute */
    Int32 value;   /* the value of this attribute */
} QCompAttr;
```

The list of attributes returned is terminated by an entry with a null key, QATTR_NULL. The use of attributes to select among components is accomplished by providing a list of attributes. A component instance qualifies if its component is registered with attributes that match the attributes supplied in the `pAttrs` array. A match is indicated if the specified attribute and the registered attribute have the same `key` and `value`. If the attribute is specified in the `pAttrs` array with the `value` QATTR_ANY, it matches any occurrence of any registered attribute with the same `key`.

If the `pAttrs` array is a simple list of attributes, a component instance qualifies for selection if it matches all of the attributes listed, as well as the non-wild card fields of the `lpInstance` argument.

This selection mechanism can be modified by the use of two special keys: QATTR_OR and QATTR_NOT. These are not actual attributes, but act as operators in the `pAttrs` attribute list. The presence of a QATTR_OR attribute (the value is ignored) has the effect of logically ORing the match results of the two attributes following QATTR_OR attribute. For example, the list (A, B, QATTR_OR, C, D) qualifies a component that has the attributes which match A and B and (C or D).

The QATTR_NOT operator attribute key inverts the match of the attribute following it in the list. For example, the list (A, B, QATTR_NOT, C) qualifies a component that has the attributes which match A and B and does not have an attribute which matches C.

Note that attribute matching follows the order of the elements in the `pAttrs` array and makes a single pass without any backtracking. A component fails to qualify for allocation as soon as the first non-matching attribute is found.
mntCompFind() finds a component

■ Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. Calling Functions Asynchronously for more details.

■ Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_ADAP_HDW_ERROR</td>
<td>Board is not available to be initialized.</td>
</tr>
<tr>
<td>ERROR_INVALID_HANDLE</td>
<td>An invalid handle was specified in the argument list.</td>
</tr>
<tr>
<td>ERROR_INVALID_PARAMETER</td>
<td>An invalid parameter was specified in the argument list.</td>
</tr>
<tr>
<td>ERROR_MNT_MERCURY_KRNL</td>
<td>See result message QResultError for details.</td>
</tr>
<tr>
<td>ERROR_MNT_MMB_ALLOC_FAILED</td>
<td>The MMB could not be allocated.</td>
</tr>
</tbody>
</table>

■ Result Messages

QComponentResult

Successful completion. The body of this message contains a single data field which may be retrieved via the QComponentResult_get() macro:

theInstance (type QCompDesc): the fully qualified address of the component instance that has the specified attributes

QResultError

Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get() macro:

errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

■ See Also

- mntCompFindAll()
returns component addresses with specified attributes mntCompFindAll()

Name: BOOL mntCompFindAll (hDevice, nTransID, startMask, endMask, lpAttr, nTimeout, lpMMB, lpOverlapped)

Inputs: HANDLE hDevice device handle
       QTrans nTransID transaction ID
       QCompDesc startMask starting address
       QCompDesc endMask ending address
       PQCompAttr lpAttr attribute list
       USHORT nTimeout time to wait
       LPMMB lpMMB MMB pointer
       LPOVERLAPPED lpOverlapped overlapped pointer

Outputs: None

Returns: TRUE if successful, FALSE if error

Includes: qhostlib.h

Category: Component Management

Mode: Asynchronous

Description

The mntCompFindAll() function returns component addresses with specified attributes.

This function returns a list of addresses and associated attributes for components on the board specified in startMask that match the requirements specified in the lpAttr array. The component addresses and attributes are returned in the body of a QCompMultipleResult message. If no qualified components are found, a QResultError message is returned. The search begins with the processor and component specified in startMask and continues sequentially through all components up to the processor and component specified in endMask.

This function may find more matching components than can be returned in a single result message, in which case the address of the next matching component is returned in the nextComponent field of the QCompMultipleResult message. To retrieve the addresses and attributes of the additional matching components, call mntCompFindAll() again with startMask set to nextComponent. This process can be repeated until nextComponent is NIL, which indicates that the result message contains all valid results for the specified search.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the CreateFile() function</td>
</tr>
</tbody>
</table>
mntCompFindAll() returns component addresses with specified attributes

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nTransID</td>
<td>transaction identifier to be used for all messages generated by this function</td>
</tr>
<tr>
<td>startMask</td>
<td>component descriptor that specifies the starting point for the search; this descriptor must specify the board, but may use NIL values for the processor and component to start at the first component on the board.</td>
</tr>
<tr>
<td>endMask</td>
<td>component descriptor that specifies the processor and component at which to stop the search; setting these descriptor fields to NIL values searches to the last component on the board.</td>
</tr>
<tr>
<td>lpAttr</td>
<td>array containing a null-terminated list of attributes that the components must match</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

A component instance descriptor has the following format:

```c
typedef struct
{
    Uint16 node;  /* reserved for node address Not currently used*/
    Uint8 board;  /* board ID within host */
    Uint8 processor; /* processor identifier */
    Uint8 component; /* component identifier */
    Uint8 instance; /* instance number (or task id) */
} QCompDesc;
```

This function ignores the node and instance fields in the startMask and endMask arguments (the node field is currently always ignored). The board field in the startMask descriptor must specify the board to be searched. The processor and component fields in the startMask and endMask descriptors may be specified in order to limit the search to a specific range. Setting startMask.processor to the nil value, QCOMP_P_NIL, starts the search with the first processor on the board; setting endMask.processor to the nil value ends the search on the last processor on the board. Setting startMask.component to the nil value, QCOMP_C_NIL, starts the search with the first component on the specified starting processor; setting a nil value for endMask.component ends the search on the last component on the specified ending processor.
returns component addresses with specified attributes  \texttt{mntCompFindAll()}  

The \texttt{lpAttr} argument references an array of QCompAttr structures which have the format:

```c
typedef struct
{
  UInt32 key; /* A key defining the type of attribute */
  Int32 value; /* the value of this attribute */
} QCompAttr;
```

The list of attributes in the \texttt{lpAttr} array is terminated by an entry with a null \texttt{key}, QATTR_NULL (the \texttt{value} is ignored). A component qualifies if it is registered with attributes which match the attribute(s) supplied in the \texttt{lpAttr} array. A match is indicated if the specified attribute and the registered attribute have the same \texttt{key} and \texttt{value}. An attribute that is specified in the \texttt{lpAttr} array with the \texttt{value} QATTR_ANY matches any occurrence of any registered attribute with the specified \texttt{key}.

If the \texttt{lpAttr} array is a simple list of attributes, a component qualifies for selection if it matches \texttt{all} of the attributes listed. This selection mechanism may be modified by the use of two special attribute keys: QATTR_OR and QATTR_NOT (the attribute value is ignored for these special keys). These are not actual attributes but act as operators in the attribute list.

The presence of a QATTR_OR attribute has the effect of OR’ing the match results of the two attributes following QATTR_OR attribute. For example, the list (A, B, QATTR_OR, C, D) qualifies a component that has attributes that match A and B and (C or D).

The QATTR_NOT operator attribute key inverts the match of the attribute following it in the list. For example, the list (A, B, QATTR_NOT, C) qualifies a component which has attributes that match A and B and which does not have an attribute that matches C.

Note that the matching of attributes follows the order of the elements in the \texttt{lpAttr} array and makes a single pass without any backtracking. A component fails to qualify for allocation as soon as the first non-matching attribute is found.

The \texttt{mntCompFindAll()} function causes the \texttt{QCompFindAll} kernel message (defined in \texttt{mercdefs.h}) to be sent. The \texttt{QCompFindAll} message size is defined as QCompFindAll_Size.
**mntCompFindAll()**  *returns component addresses with specified attributes*

### Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. *Calling Functions Asynchronously* for more details.

### Errors

- **ERROR_ADAP_HDW_ERROR**
  - Board is not available to be initialized.
- **ERROR_INVALID_HANDLE**
  - An invalid handle was specified in the argument list.
- **ERROR_INVALID_PARAMETER**
  - An invalid parameter was specified in the argument list.
- **ERROR_MNT_MERCURY_KRNL**
  - See result message `QResultError` for details.
- **ERROR_MNT_MMB_ALLOC_FAILED**
  - The MMB could not be allocated.

### Result Messages

**QCompMultipleResult**

Successful completion. The body of this message contains a variable-size payload which includes two fixed data fields followed by a variable-length list of data items. The `QCompMultipleResult_get( )` macro is used to extract the fixed fields into a data structure of type `QCompMultipleResult_t`, which contains the following elements:

- **count** (type `UInt8`): value representing the number of component descriptors in the variable part of the message body.
- **NextComponent** (type `QCompDesc`): if the search specification yielded more results than can fit in this result message, this field contains the component descriptor of the next matching component; if the body of this result message contains all of the results for the specified search, this field is set to NIL.

The remainder of the message body contains a variable-length list of data fields with **count** members. Each component descriptor is followed by a variable number of attributes associated with the component in a null-
returns component addresses with specified attributes  
mntCompFindAll() 

terminated list. Use qMsgVarFieldGet() with an initial offset of QCompMultipleResult_Size to retrieve these values.

theComponent (type QCompDesc): the component instance descriptor of a component that satisfies the search criteria. The message may contain one or more component addresses, as indicated by count, each of which is followed by a variable-length attribute list.

lpAttr (type QCompAttr): associated with the preceding component descriptor; the number of attributes is variable, and the end of the attribute list is indicated by a null attribute.

QResultError

Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get() macro:

errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

See Also

- mntCompFind()
### Description

The `mntCompFree()` function releases an allocated component instance back into a pool of available component instances. If you call this function synchronously and it finds a standard error message with a QResultError type, it returns FALSE with an ERROR_MNT_MERCURY_KRNL error code.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the <code>CreateFile()</code> function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>theInstance</td>
<td>specifies the desired component instance to be freed</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The `mntCompFree()` function causes the `QCompInstFree` kernel message (defined in `mercdef.h`) to be sent. The `QCompInstFree` message size is defined as `QCompInstFree_Size`.

---

### Name:

`BOOL mntCompFree(hDevice, nTransID, theInstance, nTimeout, lpMMB, lpOverlapped)`

**Inputs:**

- `hDevice`: device handle
- `nTransID`: transaction ID
- `theInstance`: instance to be freed
- `nTimeout`: time to wait
- `lpMMB`: MMB pointer
- `lpOverlapped`: overlapped pointer

**Outputs:** None.

**Returns:** TRUE if successful, FALSE if error

**Includes:** `qhostlib.h`

**Category:** component management function

**Mode:** asynchronous or synchronous
releases an allocated component instance \textit{mntCompFree}() 

\section*{Cautions}

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2, \textit{Calling Functions Asynchronously} for more details.

\section*{Errors}

\begin{tabular}{ll}
\textbf{ERROR_ADAP_HDW_ERROR} & \begin{itemize}
  \item Board is not available to be initialized.
\end{itemize} \\
\textbf{ERROR_INVALID_HANDLE} & \begin{itemize}
  \item An invalid handle was specified in the argument list.
\end{itemize} \\
\textbf{ERROR_INVALID_PARAMETER} & \begin{itemize}
  \item An invalid parameter was specified in the argument list.
\end{itemize} \\
\textbf{ERROR_MNT_MERCURY_KRNL} & \begin{itemize}
  \item See result message \textit{QResultError} for details.
\end{itemize} \\
\textbf{ERROR_MNT_MMB_ALLOC_FAILED} & \begin{itemize}
  \item The MMB could not be allocated.
\end{itemize}
\end{tabular}

\section*{Result Messages}

\textit{QResultComplete}

Successful completion. The message body contains no data fields.

\textit{QResultError}

Unsuccessful. The body of this message contains a single data field which may be retrieved via the \textit{QResultError_get}() macro:

\textbf{errorCode} (type Uint32): an unsigned integer that indicates the specific cause of the failure.

\section*{See Also}

\begin{itemize}
  \item \textit{mntCompAllocate}()
\end{itemize}
**mntCompUnuse()**

marks component instances as not being in use

**Name:**

```c
BOOL mntCompUnuse(hDevice, nTransID, nCount,
lpCompList, nTimeout, lpMMB, lpOverlapped)
```

**Inputs:**

- **HANDLE hDevice**
  - device handle
- **QTrans nTransID**
  - transaction ID
- **ULONG nCount**
  - instances count
- **PQCompDesc lpCompList**
  - instances array
- **USHORT nTimeout**
  - time to wait
- **LPMMB lpMMB**
  - MMB pointer
- **LPOVERLAPPED lpOverlapped**
  - overlapped pointer

**Outputs:** None.

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** component management function

**Mode:** asynchronous or synchronous

---

**Description**

The `mntCompUnuse()` function marks component instances as not being in use by the source address assigned to the device handle. This applies only to component instances that have previously been marked, through the `mntCompUse()` function, as being in use by the source address. If you call the `mntCompUnuse()` function synchronously and it finds a standard error message with a QResultError type, it returns FALSE with an ERROR_MNT_MERCURY_KRNL error code.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the <code>CreateFile()</code> function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID</td>
</tr>
<tr>
<td>nCount</td>
<td>number of instances in an array</td>
</tr>
<tr>
<td>lpCompList</td>
<td>array of component instances to mark as not in use</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The `mntCompUnuse()` function causes the QCompUnuse kernel message (defined in `mercdefs.h`) to be sent. The QCompUnuse message size is defined as QCompUnuse_Size.

88
marks component instances as not being in use mntCompUnuse()

■ Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. Calling Functions Asynchronously for more details.

■ Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_ADAP_HDW_ERROR</td>
<td>Board is not available to be initialized.</td>
</tr>
<tr>
<td>ERROR_INVALID_HANDLE</td>
<td>An invalid handle was specified in the argument list.</td>
</tr>
<tr>
<td>ERROR_INVALID_PARAMETER</td>
<td>An invalid parameter was specified in the argument list.</td>
</tr>
<tr>
<td>ERROR_MNT_MERCURY_KRNL</td>
<td>See result message QResultError for details.</td>
</tr>
<tr>
<td>ERROR_MNT_MMB_ALLOC_FAILED</td>
<td>The MMB could not be allocated.</td>
</tr>
</tbody>
</table>

■ Result Messages

QResultComplete
Successful completion. The message body contains no data fields.

QResultError
Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get() macro:
errorCode (type Uint32): an unsigned integer that indicates the specific cause of the failure.

■ See Also

- mntCompUse()
The `mntCompUse()` function marks component instances as being in use by the source address assigned to the device handle. If you call this function synchronously and it finds a standard error message with a `QResultError` type, it returns `FALSE` with an `ERROR_MNT_MERCURY_KRNL` error code.

Each source address is assigned to an Mpath device name. When a device handle is closed after using component instances, the driver notifies the DM3 board that the application with this source address has terminated. The DM3 board forwards this notification to the MercPath’s in-use component instances so they can perform appropriate cleanup tasks.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hDevice</code></td>
<td>handle to a message path device returned from the <code>CreateFile()</code> function</td>
</tr>
<tr>
<td><code>nTransID</code></td>
<td>transaction ID</td>
</tr>
<tr>
<td><code>nCount</code></td>
<td>number of instances in an array</td>
</tr>
<tr>
<td><code>lpCompList</code></td>
<td>an array of component instances to mark as in use</td>
</tr>
<tr>
<td><code>lpPayload</code></td>
<td>An array of <code>nCount</code> size, representing a payload for the corresponding component instance in <code>lpCompList</code>.</td>
</tr>
<tr>
<td><code>nTimeout</code></td>
<td>time (in seconds) to wait for a response</td>
</tr>
</tbody>
</table>
marks component instances as being in use

mntCompUse()

Parameter | Description
---|---
lpMMB | pointer to an MMB structure
lpOverlapped | pointer to an OVERLAPPED structure

The mntCompUnuse( ) function causes the QCompUse kernel message (defined in mercdefs.h) to be sent. The QCompUse message size is defined as QCompUse_Size.

- **Cautions**
  The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. **Calling Functions Asynchronously** for more details.

- **Errors**
  - ERROR_ADAP_HDW_ERROR: Board is not available to be initialized.
  - ERROR_INVALID_HANDLE: An invalid handle was specified in the argument list.
  - ERROR_INVALID_PARAMETER: An invalid parameter was specified in the argument list.
  - ERROR_MNT_MERCURY_KRNL: See result message QResultError for details.
  - ERROR_MNT_MMB_ALLOC_FAILED: The MMB could not be allocated.

- **Result Messages**
  **QResultComplete**
  Successful completion. The message body contains no data fields.

  **QResultError**
  Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get( ) macro: 
  **errorCode** (type Uint32): an unsigned integer that indicates the specific cause of the failure.

- **See Also - mntCompUnuse( )**
**mntCompleteStreamIo( )** *completes pending stream I/O requests*

**Name:** BOOL mntCompleteStreamIo(hDevice)

**Inputs:**
- HANDLE hDevice  
  * device handle

**Outputs:** None

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** stream I/O function

**Mode:** synchronous

---

**Description**

The `mntCompleteStreamIo( )` function completes pending stream I/O requests on the stream currently attached to the specified device.

**NOTE:** While the `mntCompleteStreamIo( )` function itself works in the synchronous mode, the actual reads or writes complete asynchronously. Therefore, you need to be prepared for these premature I/O completions. Each premature I/O completion returns as successful with the actual number of bytes transferred.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>stream device handle</td>
</tr>
</tbody>
</table>

**Cautions - None.**

**Errors**

- **ERROR_BAD_COMMAND**  
  * The specified handle does not have an attached stream.

- **ERROR_INVALID_FUNCTION**  
  * The stream handle specified is of the wrong type.

- **ERROR_INVALID_HANDLE**  
  * An invalid handle was specified in the argument list.

**Result Messages - None.**

**See Also - None.**
**Name:** LPMMB mntCopyMMB(lpMMB)

**Inputs:** LPMMB lpMMB  
pointer to MMB to be copied

**Outputs:** None

**Returns:** LPMMB a pointer to a new MMB
NULL when the new MMB could not be allocated

**Includes:** qhostlib.h

**Category:** message I/O function

**Mode:** synchronous

■ **Description**

The `mntCopyMMB()` function copies the specified Message Block to a newly created MMB.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpMMB</td>
<td>pointer to the MMB from which to copy</td>
</tr>
</tbody>
</table>

■ **Cautions**

None.

■ **Errors**

`ERROR_INVALID_PARAMETER`  
• An invalid parameter was specified in the argument list.

■ **Result Messages**

None.

■ **See Also**

`mntAllocateMMB()`
mntDetachMercStream()  

detaches a stream

**Name:**  
BOOL mntDetachMercStream(hDevice, nTimeout, lpOverlapped)

**Inputs:**  
- HANDLE hDevice  
  - device handle  
- USHORT nTimeout  
  - time to wait for response  
- LPOVERLAPPED lpOverlapped  
  - overlapped pointer

**Outputs:**  
None.

**Returns:**  
TRUE if successful, FALSE if error

**Includes:**  
qhostlib.h

**Category:**  
stream I/O function

**Mode:**  
asynchronous or synchronous

---

### Description

The `mntDetachMercStream()` function detaches a stream from the specified stream device. If all references to a particular stream ID have been detached, the stream is closed. You can no longer read from or write to that stream.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>stream device handle</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

### Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. *Calling Functions Asynchronously* for more details.

### Errors

- **ERROR_ADAP_HDW_ERROR**  
  - Board is not available to be initialized.

- **ERROR_BAD_COMMAND**  
  - The specified handle does not have an attached stream.

- **ERROR_INVALID_FUNCTION**  
  - The stream handle specified is of the wrong type.
*detaches a stream*  

*mntDetachMercStream()*

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_INVALID_HANDLE</td>
<td>• An invalid handle was specified in the argument list.</td>
</tr>
<tr>
<td>ERROR_INVALID_PARAMETER</td>
<td>• An invalid parameter was specified in the argument list.</td>
</tr>
<tr>
<td>ERROR_MNT_STRM_ALREADY_CLOSED</td>
<td>• The specified stream ID has been closed.</td>
</tr>
<tr>
<td>ERROR_MNT_STRM_NOT_OPEN</td>
<td>• The specified stream ID is not open.</td>
</tr>
</tbody>
</table>

### Result Messages

None.

### See Also

None.
**mntEnumMpathDevice( ) enumerates existing Mpath devices**

**Name:** \( \text{BOOL mntEnumMpathDevice(Mode, lpDeviceName, lpDeviceNameSize, lpDevStatus)} \)

**Inputs:**
- ULONG \( \text{Mode} \) / request mode

**Outputs:**
- LPCSTR \( \text{lpDeviceName} \) / device name pointer
- PULONG \( \text{lpDeviceNameSize} \) / length of device name
- PULONG \( \text{lpDevStatus} \) / current device status

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** message I/O function

**Mode:** synchronous

### Description

The `mntEnumMpathDevice( )` function enumerates existing Mpath devices. Upon successful return, the function fills in the locations pointed to by `lpDeviceName`, `lpDeviceNameSize`, and `lpDevStatus` with the device name, device name length, and current device status.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>enumeration method:</td>
</tr>
<tr>
<td></td>
<td>MNT_FIRST_AVAILABLE: the <code>lpDeviceName</code> parameter contains the first unused Mpath device.</td>
</tr>
<tr>
<td></td>
<td>MNT_GET_FIRST: function returns the first device currently defined in the system.</td>
</tr>
<tr>
<td></td>
<td>MNT_GET_NEXT: function returns the next device in the list.</td>
</tr>
<tr>
<td><code>lpDeviceName</code></td>
<td>pointer to the device name</td>
</tr>
<tr>
<td><code>lpDeviceNameSize</code></td>
<td>device name length</td>
</tr>
</tbody>
</table>
enumerates existing Mpath devices  

\textbf{mntEnumMpathDevice( )}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpDevStatus</td>
<td>current status of the device returned in the \texttt{lpDeviceName} parameter. This status can be any of the following:</td>
</tr>
<tr>
<td></td>
<td>MERC_DEVICE_STATUS_FREE: Device has not been opened.</td>
</tr>
<tr>
<td></td>
<td>MERC_DEVICE_STATUS_INUSE_EXCLUSIVE: Device has been opened by an application that specified exclusive access.</td>
</tr>
<tr>
<td></td>
<td>MERC_DEVICE_STATUS_INUSE_SHARED: Device has been opened by an application that specified shared access (for read, write, or both).</td>
</tr>
</tbody>
</table>

\section*{Cautions}

There is no guarantee that any subsequent call to the \texttt{CreateFile( )} function will succeed. As always, the caller must be prepared to handle an error return.

\section*{Errors}

\begin{itemize}
\item \texttt{ERROR\_MNT\_SYSTEM\_ERR}  
  Direct Interface system error.  
  (An internal error occurred within the MNTI DLL.)
\item \texttt{ERROR\_INVALID\_PARAMETER}  
  An invalid parameter was specified in the argument list.
\item \texttt{ERROR\_FILE\_NOT\_FOUND}  
  No device was found that matches the specified criteria.
\end{itemize}

\section*{Result Messages}

None.

\section*{See Also}

\texttt{mntEnumStrmDevice( )}
**mntEnumStrmDevice()**  
enumerates existing Stream devices

<table>
<thead>
<tr>
<th>Name:</th>
<th>BOOL mntEnumStrmDevice(Mode, lpDeviceName, lpDeviceNameSize, lpDevStatus)</th>
</tr>
</thead>
</table>
| Inputs: | ULONG Mode  
lPCSTR lpDeviceName  
PULONG lpDeviceNameSize  
PULONG lpDevStatus |
| Outputs: | • request mode  
• device name pointer  
• length of device name  
• current device status |
| Returns: | TRUE if successful, FALSE if error |
| Includes: | qhostlib.h |
| Category: | message I/O function |
| Mode: | synchronous |

**Description**

The `mntEnumStrmDevice()` function enumerates existing Stream devices. Upon successful return, the function fills in the locations pointed to by `lpDeviceName`, `lpDeviceNameSize`, and `lpDevStatus` with the device name, device name length, and current device status.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| Mode      | enumeration method:  
  MNT_FIRST_AVAILABLE: the `lpDeviceName` parameter contains the first unused Stream device.  
  MNT_GET_FIRST: function returns the first device currently defined in the system.  
  MNT_GET_NEXT: function returns the next device in the list. |
| lpDeviceName | pointer to the device name |
| lpDeviceNameSize | device name length |
enumerates existing Stream devices

\[ \text{mntEnumStrmDevice}( ) \]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpDevStatus</td>
<td>current status of the device returned in the lpDeviceName parameter. This status can be any of the following:</td>
</tr>
<tr>
<td></td>
<td>MERC_DEVICE_STATUS_FREE: Device has not been opened.</td>
</tr>
<tr>
<td></td>
<td>MERC_DEVICE_STATUS_INUSE_EXCLUSIVE: Device has been opened by an application that specified exclusive access.</td>
</tr>
<tr>
<td></td>
<td>MERC_DEVICE_STATUS_INUSE_SHARED: Device has been opened by an application that specified shared access (for read, write, or both).</td>
</tr>
</tbody>
</table>

**Cautions**

There is no guarantee that any subsequent call to the CreateFile( ) function will succeed. As always, the caller must be prepared to handle an error return.

**Errors**

- `ERROR_MNT_SYSTEM_ERR` • Direct Interface system error. (An internal error occurred within the MNTI DLL.)
- `ERROR_INVALID_PARAMETER` • An invalid parameter was specified in the argument list.
- `ERROR_FILE_NOT_FOUND` • No device was found that matches the specified criteria.

**Result Messages**

None.

**See Also**

\[ \text{mntEnumMpathDevice}( ) \]
**mntFreeMMB( )**  frees the specified Message Block

**Name:**  BOOL mntFreeMMB(lpMMB)

**Inputs:**  LPMMB lpMMB  • pointer to MMB to be freed

**Outputs:**  None

**Returns:**  TRUE if successful, FALSE if error

**Includes:**  qhostlib.h

**Category:**  message I/O function

**Mode:**  synchronous

---

**Description**

The *mntFreeMMB( )* function frees the specified Message Block.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpMMB</td>
<td>pointer that was returned from a successful call to the <em>mntAllocateMMB( )</em> function</td>
</tr>
</tbody>
</table>

**Cautions**

None.

**Errors**

ERROR_INVALID_PARAMETER  • An invalid parameter was specified in the argument list.

**Result Messages**

None.

**See Also**

*mntAllocateMMB( )*
lists boards with matching attributes

mntGetBoardsByAttr( )

Name: BOOL mntGetBoardsByAttr(pAttr, MaxAttrs, pBoardAttr, pTotalEntries, pBoardsFound)

Inputs:
PQValueAttr pAttr  ● board attributes list
ULONG MaxAttrs  ● maximum attributes
PULONG pTotalEntries  ● number of entries specified in the attributes array

Outputs:
PQBoardAttr pBoardAttr  ● matching boards
PULONG pBoardsFound  ● boards found

Returns: TRUE if successful, FALSE if error

Includes: qhostlib.h

Category: message I/O function

Mode: synchronous

Description

The mntGetBoardsByAttr( ) function lists boards with matching attributes. This function accesses the NT registry and reads the attributes of each board configured on the system. It then compares the listed attributes against the attributes provided by the caller in pAttr. Upon successful return, the function fills in the locations pointed to by pBoardAttr and pBoardsFound with an array of board attributes matching the specified pAttr values and the number of boards with matching attributes.

Parameter Description

pAttr array of Registry value attributes to be matched
MaxAttrs maximum number of attributes that can be stored in the array pointed to by the pBoardAttr parameter
pBoardAttr array of board attributes that matched the specifications in pAttr
pTotalEntries number of entries in pBoardAttr used for input and output
pBoardsFound number of boards found

The pAttr argument references an array of QValueAttr structures. These attributes identify available board capabilities. A value of type QValueAttr is a structure of the format:

typedef struct
{
    char   ValueName[MNT_MAX_VALUE_NAME_SIZE];
};
mntGetBoardsByAttr() lists boards with matching attributes

```c
typedef struct {
    char ValueName[MNT_MAX_VALUE_NAME_SIZE];
    ULONG ValueType;
    char Value[MNT_MAX_VALUE_SIZE];
    ULONG BoardNo;
} QBoardAttr;
```

Where:

ValueName: contains a NULL terminated string specifying the name of the value which matched.

ValueType: is one of the Win32 registry types; REG_DWORD, REG_SZ, or REG_MULTISZ.

The list of attributes returned is terminated by the entry with a null key, QATTR_NULL. A match is indicated if the specified attribute and the registered attribute have the same name and value.

The pBoardAttr parameter references an array of QBoardAttr structures. These attributes identify available board capabilities. A value of type QBoardAttr is a structure of the format:

```c
typedef struct {
    char ValueName[MNT_MAX_VALUE_NAME_SIZE];
    ULONG ValueType;
    char Value[MNT_MAX_VALUE_SIZE];
    ULONG BoardNo;
} QBoardAttr;
```

Where:

ValueName: contains a NULL terminated string specifying the name of the value which matched.

ValueType: is one of the Win32 registry types; REG_DWORD, REG_SZ, or REG_MULTISZ.
lists boards with matching attributes  
mntGetBoardsByAttr( )

Value: is the current value of the value named in ValueName.

BoardNo: contains the logical board ID of the board which contained the matching attribute.

The mntGetBoardsByAttr( ) function lists each board and the attribute that the board matched in the attribute list. Multiple listings of one board are possible if the board matches various attributes provided in the pAttr parameter.

The attributes list is terminated by the entry with a null key, QATTR_NULL, if there is enough space in the attribute list to list all the boards and the null key. If the null key is absent, the mntGetBoardsByAttr( ) function did not completely list all the boards matching the attributes. A match is indicated if the specified attribute and the registered attribute have the same name and value.

■ Cautions

None.

■ Errors

ERROR_CANTOPEN  • Cannot open registry key.
ERROR_CANTREAD  • Cannot read registry key.
ERROR_INVALID_HANDLE  • An invalid handle was specified in the argument list.
ERROR_INVALID_PARAMETER  • An invalid parameter was specified in the argument list.
ERROR_MNT_CANTCLOSE  • Cannot close registry key.
ERROR_MNT_INVALID_VALUE_TYPE  • An invalid value type was specified in the attribute list.
ERROR_MNT_NO_BOARDS_BY_ATTR  • No boards match the specified criteria.
ERROR_MNT_NO_MEM  • The attribute list does not have enough space to list any matches.
mntGetBoardsByAttr() lists boards with matching attributes

■ Result Messages

None.

■ See Also

None.
The `mntGetDrvVersion()` function retrieves the driver version string from the Class Driver (DLGCMCD). Upon successful return, the function fills in the location pointed to by `lpVersion` with the driver version string.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpVersion</td>
<td>driver version</td>
</tr>
</tbody>
</table>

### Cautions

The `lpVersion` version string must be the same size as `MNT_VERSION_STRING_SIZE`.

### Errors

- **ERROR_INVALID_PARAMETER**
  - An invalid parameter was specified in the argument list.

- **ERROR_INSUFFICIENT_BUFFER**
  - The version string buffer is too small.

### Result Messages

None.

### See Also

- `mntGetLibVersion()`
**mntGetLibVersion**

**Name:** BOOL mntGetLibVersion(lpVersion)

**Inputs:** None

**Outputs:** LPCSTR lpVersion

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** debug support function

**Mode:** synchronous

### Description

The **mntGetLibVersion** function retrieves the Direct Interface library version string from the Class Driver (DLGCMCD). Upon successful return, the function fills in the location pointed to by **lpVersion** with the Direct Interface library version string.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpVersion</td>
<td>Direct Interface library version string</td>
</tr>
</tbody>
</table>

### Cautions

The **lpVersion** version string must be the same size as MNT_VERSION_STRING_SIZE.

### Errors

- **ERROR_INVALID_PARAMETER**

  An invalid parameter was specified in the argument list.

### Result Messages

None.

### See Also

- **mntGetDrvVersion**

---

106
returns the stream ID \textit{mntGetMercStreamID( )}

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{Name:} & BOOL mntGetMercStreamID(hDevice, lpMercStreamID, lpBoardNumber) \\
\hline
\textbf{Inputs:} & \\
HANDLE hDevice & \textbullet \ device handle \\
\textbf{Outputs:} & \\
PULONG lpMercStreamID & \textbullet \ pointer to stream ID \\
PULONG lpBoardNumber & \textbullet \ pointer to board number \\
\hline
\textbf{Returns:} & TRUE if successful, FALSE if error \\
\textbf{Includes:} & qhostlib.h \\
\textbf{Category:} & stream I/O function \\
\textbf{Mode:} & synchronous \\
\hline
\end{tabular}
\end{center}

\section{Description}

The \textit{mntGetMercStreamID( )} function returns the stream ID currently associated with the specified Stream device handle. Upon successful return, the function fills in the locations pointed to by \textit{lpMercStreamID} and \textit{lpBoardNumber} with the stream identifier and the board number.

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{Parameter} & \textbf{Description} \\
\hline
hDevice & Stream device handle \\
lpMercStreamID & on return, contains the stream ID \\
lpBoardNumber & on return, contains the board number \\
\hline
\end{tabular}
\end{center}

\section{Cautions -}

None.

\section{Errors}

\begin{itemize}
\item \textbf{ERROR_BAD_COMMAND} \textbullet \ The specified handle does not have an attached stream.
\item \textbf{ERROR_INVALID_FUNCTION} \textbullet \ The stream handle specified is of the wrong type.
\item \textbf{ERROR_INVALID_HANDLE} \textbullet \ An invalid handle was specified in the argument list.
\item \textbf{ERROR_INVALID_PARAMETER} \textbullet \ An invalid parameter was specified in the argument list.
\end{itemize}
\textit{mntGetMercStreamID()} \hspace{1cm} \textit{returns the stream ID}

\section*{Result Messages}

None.

\section*{See Also}

None.
returns the message path source address  

**mntGetMpathAddr()**

<table>
<thead>
<tr>
<th>Name:</th>
<th>BOOL mntGetMpathAddr(hDevice, lpSrcAddr, lpDestAddr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
<td>HANDLE hDevice</td>
</tr>
<tr>
<td></td>
<td>PQCompDesc lpSrcAddr</td>
</tr>
<tr>
<td></td>
<td>PQCompDesc lpDestAddr</td>
</tr>
<tr>
<td>Returns:</td>
<td>TRUE if successful, FALSE if error</td>
</tr>
<tr>
<td>Includes:</td>
<td>qhostlib.h</td>
</tr>
<tr>
<td>Category:</td>
<td>Message I/O function</td>
</tr>
<tr>
<td>Mode:</td>
<td>synchronous</td>
</tr>
</tbody>
</table>

### Description

The `mntGetMpathAddr()` function returns the message path source address bound to the specified device. Upon successful return, the function fills in the locations pointed to by `lpSrcAddr` and `lpDestAddr` with the source address assigned to the specified Mpath device and the destination address used in the most recent I/O request, if any.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>Mpath device handle</td>
</tr>
<tr>
<td>lpSrcAddr</td>
<td>pointer to the source address assigned to the specified Mpath device</td>
</tr>
<tr>
<td>lpDestAddr</td>
<td>pointer to the destination address used in the most recent I/O request</td>
</tr>
</tbody>
</table>

### Cautions

None.

### Errors

- **ERROR_INVALID_HANDLE**
  - An invalid handle was specified in the argument list.

- **ERROR_INVALID_PARAMETER**
  - An invalid parameter was specified in the argument list.
mntGetMpathAddr() returns the message path source address

- **Result Messages**
  None.

- **See Also**
  None.
gets the out-of-band stream attributes  

mntGetStreamHeader()  

**Name:**  
BOOL mntGetStreamHeader(hDevice, lpHeader)  

**Inputs:**  
HANDLE hDevice  
• device handle  
PSTRM_HDR lpHeader  
• pointer to local memory area  

**Outputs:**  
PSTRM_HDR lpHeader  
• pointer to stream header info  

**Returns:**  
TRUE if successful, FALSE if error  

**Includes:**  
qhostlib.h  

**Category:**  
stream I/O function  

**Mode:**  
synchronous  

---  

### Description  

The **mntGetStreamHeader( )** function gets the out-of-band stream attributes that are defined by the structure pointed to by the **lpHeader** parameter. Upon successful return, the function fills in the location pointed to by **lpHeader** with stream header information.  

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>Stream device handle</td>
</tr>
<tr>
<td>lpHeader</td>
<td>pointer to a local memory area containing out-of-band stream attributes</td>
</tr>
</tbody>
</table>

The underlying bulk data stream is passed in blocks between the host and the DM3 platform. These blocks carry attribute data that can control data transfer and provide out-of-band data associated with the stream and the blocks.

The **lpHeader** structure is as follows:

```c
typedef struct {
    ULONG sequence;
    UCHAR bufFlags;  // MNT_EOD - End of Data = 0x01
    // MNT_EOT - End of Transmission = 0x02
    // MNT_EOF - End of File = 0x04 (equivalent to EOS)
    // MNT_USER1 - User specified flag = 0x08
    // MNT_USER2 - User specified flag = 0x10
    // MNT_USER3 - User specified flag = 0x20
    // MNT_USER4 - User specified flag = 0x40
    // MNT_USER5 - User specified flag = 0x80
    UCHAR encoding;
    UCHAR pad1;       // reserved for future use
    UCHAR sysFlags;   // read-only
    // STREAM_CLOSED = 0x01
    // STREAM_BROKEN = 0x02
    ULONG canTakeLimit; // read-only
    ULONG initialCanTake; // read-only
} PSTRM_HDR;
```
mntGetStreamHeader() gets the out-of-band stream attributes

```c
ULONG currentCanTake;   // read-only
ULONG requestedSize;    // read-only
ULONG actualSize;       // read-only
} STRM_HDR, *PSTRM_HDR;
```

The **sequence** field is used as an incrementing counter as blocks are written. This field is automatically filled by the lower level stream data block transport code.

The **bufFlags** field indicates the out-of-band stream attributes as defined below:

- The **MNT_EOD** flag indicates the end of a valid grouping of data blocks. It terminates an operation, such as a data transfer, without closing the stream.
- The **MNT_EOT** flag indicates the end of a collection of groupings that have been delineated by **MNT_EOD** flags. Without closing the stream, it marks such operations as a forced termination of a grouping of operations in which the data transfer groupings were buffered onto a stream, but were not yet processed at the time of termination.
- The **MNT_EOF** flag indicates the end of a stream. It is normally set in the last block of a stream when the writer closes its end of the stream.
- The **MNT_USERn** flags can be used for any application-level purpose.

The **encoding** field indicates the calling processor byte ordering convention (big-endian or little-endian).

The **sysFlags** are read-only flags as defined below:

- The **STREAM_CLOSED** flag is set when **EOS** is detected on an incoming data node.
- The **STREAM_BROKEN** flag is set when the stream device has been closed. All write requests fail with a broken stream error.

### Cautions

None.

### Errors

- **ERROR_BAD_COMMAND**
  - The specified handle does not have an attached stream.
gets the out-of-band stream attributes  

$\textit{mntGetStreamHeader( )}$

<table>
<thead>
<tr>
<th>Result Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_INVALID_FUNCTION</td>
<td>The stream handle specified is of the wrong type.</td>
</tr>
<tr>
<td>ERROR_INVALID_HANDLE</td>
<td>An invalid handle was specified in the argument list.</td>
</tr>
<tr>
<td>ERROR_INVALID_PARAMETER</td>
<td>An invalid parameter was specified in the argument list.</td>
</tr>
</tbody>
</table>

- **Result Messages**

None.

- **See Also**

  - $\textit{mntGetStreamInfo( )}$
  - $\textit{mntSetStreamHeader( )}$
mntGetStreamInfo() gets global board-specific stream information

Name: BOOL mntGetStreamInfo(BoardNumber, lpStrmInfos)
Inputs: ULONG BoardNumber • board number
        PSTRM_INFO lpStrmInfos • STRM_INFO pointer
Outputs: PSTRM_INFO lpStrmInfos • pointer to stream info
Returns: TRUE if successful, FALSE if error
Includes: qstream.h
Category: stream I/O function
Mode: synchronous

### Description

The `mntGetStreamInfo()` function gets global board-specific stream information, such as the available stream sizes. The `qstream.h` include file contains the STRM_INFO structure. Upon successful return, the function fills in the location pointed to by `lpStreamInfos` with global board-specific stream information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoardNumber</td>
<td>DM3 board number</td>
</tr>
<tr>
<td>lpStrmInfos</td>
<td>pointer to an array that contains stream information</td>
</tr>
</tbody>
</table>

The STRM_INFO structure is defined in `qstream.h` as follows:

```c
typedef struct {
    int    NumStrmGroups;
    int    DataBlockSize;
    STRM_GROUP_CFG StrmGroups[MNT_STREAM_MAX_NUM_GROUPS];
} STRM_INFO, *PSTRM_INFO;
```

The `NumStrmGroups` field defines the number of stream groups available. A stream group is used for defining a number of streams with different stream size. (Maximum value is 20.)

The `DataBlockSize` field defines the default data block size, currently set at 4032 bytes.

### Cautions

None.

114
gets global board-specific stream information  

\textit{mntGetStreamInfo( )}

\begin{itemize}
  \item \textbf{Errors}
  \begin{itemize}
    \item ERROR_GEN_FAILURE
    \begin{itemize}
      \item Direct Interface internal error has occurred.
    \end{itemize}
    \item ERROR_INVALID_PARAMETER
    \begin{itemize}
      \item An invalid parameter was specified in the argument list.
    \end{itemize}
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \textbf{Result Messages}
  \begin{itemize}
    \item None.
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \textbf{See Also}
  \begin{itemize}
    \item \textit{mntGetStreamHeader( )}
  \end{itemize}
\end{itemize}
**mntGetTLSmmb()** retrieves the thread-local storage MMB

**Name:** BOOL mntGetTLSmmb(lppMMB, cmdMsg, replyMsg)

**Inputs:** None.

**Outputs:**
- LPMMB *lppMMB* • TLS MMB pointer
- QMsgRef *cmdMsg* • command pointer
- QMsgRef *replyMsg* • reply pointer

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** Message I/O function

**Mode:** synchronous

---

**Description**

The mntGetTLSmmb( ) function retrieves the thread-local storage MMB maintained by the Direct Interface host library. Thread-local storage enables data to be associated with a specific program thread. You will typically use this function if a synchronous function call has failed, and you need to examine the firmware reply message. Upon successful return, this function fills in the locations pointed to by lppMMB, cmdMsg, and replyMsg.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lppMMB</td>
<td>pointer to the thread-local storage MMB</td>
</tr>
<tr>
<td>cmdMsg</td>
<td>pointer to the command message within the MMB</td>
</tr>
<tr>
<td>replyMsg</td>
<td>pointer to the reply message, if any, within the MMB</td>
</tr>
</tbody>
</table>

**Cautions**

Check for NULLs before using these pointers.

**Errors**

- **ERROR_INVALID_PARAMETER** • An invalid parameter was specified in the argument list.
- **ERROR_MNT_NO_MEM** • No thread-local storage MMB was found.
retrieves the thread-local storage MMB

mntGetTLSmmb()

■ Result Messages

None.

■ See Also

None.
**mntNotifyRegister()**  
*enables notification of sub-component failure*

<table>
<thead>
<tr>
<th>Name:</th>
<th>BOOL mntNotifyRegister(hDevice, nTransID, compDesc, nTimeout, lpMMB, lpOverlapped)</th>
</tr>
</thead>
</table>
| Inputs: | HANDLE hDevice  
QTrans nTransID  
QCompDesc compDesc  
USHORT nTimeout  
LPMMB lpMMB  
LPOVERLAPPED lpOverlapped |
| Returns: | TRUE if successful, FALSE if error |
| Includes: | qhostlib.h |
| Category: | exit notification services |
| Mode: | asynchronous |

---

**Description**

The **mntNotifyRegister()** function enables notification of sub-component failure. Once completed, the caller will receive notification once any sub-components have terminated unexpectedly.

This function registers the address of the message path device to be notified when a sub-component on an SP on the DM3 board specified by **compDesc** terminates due to a catastrophic failure. After this function has been called, a **QFailureNotify** message is sent to the registered address whenever an unexpected termination of any SP sub-component occurs. The registration performed by this function remains in effect until the target board is restarted or until the registration is cancelled with a call to **mntNotifyUnregister()**. While the registration is in effect, any number of **QFailureNotify** messages (including none) may be sent to the registered address.

**Parameter** | **Description**
--- | ---
**hDevice** | handle to a message path device returned from the CreateFile() function
**nTransID** | transaction ID to be used in all messages generated by this function
**compDesc** | partially specified component address. The board address in this descriptor indicates the location of the SP sub-component that has terminated.
enables notification of sub-component failure \textit{mntNotifyRegister( )}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{nTimeout}</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>\texttt{lpMMB}</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>\texttt{lpOverlapped}</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

The \texttt{mntNotifyRegister( )} function causes the \textit{QRegisterNotify} kernel message (defined in \textit{mercdefs.h}) to be sent. The \textit{QRegisterNotify} message size is defined as \texttt{QRegisterNotify\_Size}.

\section*{Cautions}

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. \textit{Calling Functions Asynchronously} for more details.

\section*{Errors}

- \texttt{ERROR\_ADAP\_HDW\_ERROR} • Board is not available to be initialized.
- \texttt{ERROR\_INVALID\_HANDLE} • An invalid handle was specified in the argument list.
- \texttt{ERROR\_INVALID\_PARAMETER} • An invalid parameter was specified in the argument list.
- \texttt{ERROR\_MNT\_MERCURY\_KRNL} • See result message \textit{QResultError} for details.
- \texttt{ERROR\_MNT\_MMB\_ALLOC\_FAILED} • The MMB could not be allocated.

\section*{Result Messages}

- \textit{QResultComplete} Successful registration. The message body contains no data fields.
- \textit{QFailureNotify} Sub-component failure notification message. This message is only sent in the event that an SP sub-component on the specified board terminates
mntNotifyRegister( ) enables notification of sub-component failure

unexpectedly. Any number of these messages may be sent following a single call to mntNotifyRegister().

The body of the QFailureNotify message contains a variable-size payload which includes a single fixed data field followed by a variable-length list of data items. Use the QFailureNotify_get( ) macro to extract the fixed field into a data structure of type QFailureNotify_t, which contains the following element:

**count** (type Uint8): the number of component descriptors contained in the variable part of the message body.

The remainder of the message body contains a variable-length list of data fields with **count** members. Each component descriptor is followed by a variable number of attributes associated with the component in a null-terminated list. Use qMsgVarFieldGet( ) with an initial offset of QFailureNotify_Size to retrieve these values.

**component** (type QCompDesc): the component address of an SP component that terminated. The message may contain one or more of these failed component addresses, as indicated by **count**, each of which is followed by a variable-length attribute list.
**attr** (type QCompAttr): an attribute associated with the preceding **component**. The number of such attributes is variable and the end of the list is indicated by a null attribute.

**QResultError**

Unsuccessful. The body of this message contains a single data field which may be retrieved via the QResultError_get( ) macro:

**errorCode** (type Uint32): an unsigned integer that indicates the specific cause of the failure.

■ See Also

- mntNotifyUnregister( )
- mntSetExitNotify( )
**disables notification of sub-component failure**  

**mntNotifyUnregister( )**

<table>
<thead>
<tr>
<th>Name</th>
<th>BOOL mntNotifyUnregister(hDevice, nTransID, compDesc, nTimeout, lpMMB, lpOverlapped)</th>
</tr>
</thead>
</table>
| Inputs: | HANDLE hDevice ● device handle  
QTrans nTransID ● transaction ID  
QCompDesc compDesc ● partially specified component address  
USHORT nTimeout ● time to wait  
LPMMB lpMMB ● MMB pointer  
LPOVERLAPPED lpOverlapped ● overlapped pointer |
| Returns: | TRUE if successful, FALSE if error |
| Includes: | qhostlib.h |
| Category: | exit notification services |
| Mode: | asynchronous |

### Description

The **mntNotifyUnregister( )** function disables notification of sub-component failure.

This function cancels the exit notification registration of the address in **hDevice**. After this function has been called, the specified device no longer receives a notification message when an unexpected termination of any SP sub-component occurs.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the <em>CreateFile( )</em> function</td>
</tr>
<tr>
<td>nTransID</td>
<td>transaction ID to be used in all messages generated by this function</td>
</tr>
<tr>
<td>compDesc</td>
<td>partially specified component address. The board address in this descriptor indicates the location of the SP sub-component that has terminated.</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpMMB</td>
<td>pointer to an MMB structure</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>
mntNotifyUnregister( )  

The `mntNotifyUnregister( )` function causes the `QUnregisterNotify` kernel message (defined in `mercdefs.h`) to be sent. The `QUnregisterNotify` message size is defined as `QUnregisterNotify_Size`.

### Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. *Calling Functions Asynchronously* for more details.

### Errors

- **ERROR_ADAP_HDW_ERROR**  
  - Board is not available to be initialized.

- **ERROR_INVALID_HANDLE**  
  - An invalid handle was specified in the argument list.

- **ERROR_INVALID_PARAMETER**  
  - An invalid parameter was specified in the argument list.

- **ERROR_MNT_MERCURY_KRNL**  
  - See result message `QResultError` for details.

- **ERROR_MNT_MMB_ALLOC_FAILED**  
  - The MMB could not be allocated.

### Result Messages

- **QResultComplete**  
  - Successful completion. The message body contains no data fields.

- **QResultError**  
  - Unsuccessful. The body of this message contains a single data field which may be retrieved via the `QResultError_get( )` macro:

  - **errorCode** (type `Uint32`): an unsigned integer that indicates the specific cause of the failure.

### See Also

- `mntNotifyRegister( )`
- `mntSetExitNotify( )`

122
### Description

The `mntRegisterAsyncMessages()` function enables receipt of asynchronous messages through a set of MMB structures and corresponding event object handles. As with the `mntSendMessage()` function, make sure that you prepare the MMBs properly so that they are ready to be sent to the DM3 board. As each MMB completes, its associated event is set by the driver and the MMB is already filled with the reply message. The calling application must reset the event as soon as the MMB is free for reuse. Until the event is reset, the driver cannot use the associated MMB to repost the I/O request. This also means that the event must be a manual-reset type.

For each low-latency asynchronous message, you should specify two or more MMBs and associated events to ensure that no events will be missed. Otherwise, the driver resorts to a coarse one-second-resolution timer in checking whether the MMB is ready for reuse as indicated by its event object being in the non-signaled state.

Unlike the MMBs that you use with the `mntSendMessage()` function, you can set an infinite time out through the `mntRegisterAsyncMessages()` function. Use the defined constant MNT_NO_TIMEOUT to wait indefinitely. You can use an infinite timeout, for instance, if a network alarm is expected. Otherwise, if an MMB times out, the event is signaled, and you must examine the `actualReplyCount` field in the MMB structure before you process any reply messages.

<table>
<thead>
<tr>
<th>Name</th>
<th>BOOL mntRegisterAsyncMessages(hDevice, nCount, lpEvents, lpMMBs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
<td>hDevice (device handle)</td>
</tr>
<tr>
<td></td>
<td>nCount (number of array elements)</td>
</tr>
<tr>
<td></td>
<td>lpEvents (event array pointer)</td>
</tr>
<tr>
<td></td>
<td>lpMMBs (MMB array pointer)</td>
</tr>
<tr>
<td>Outputs:</td>
<td>None.</td>
</tr>
<tr>
<td>Returns:</td>
<td>TRUE if successful, FALSE if error</td>
</tr>
<tr>
<td>Includes:</td>
<td>qhostlib.h</td>
</tr>
<tr>
<td>Category:</td>
<td>message I/O function</td>
</tr>
<tr>
<td>Mode:</td>
<td>synchronous</td>
</tr>
</tbody>
</table>

---

**enables receipt of asynchronous messages **

**mntRegisterAsyncMessages()**

---

123
mntRegisterAsyncMessages( ) enables receipt of asynchronous messages

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the CreateFile( ) function.</td>
</tr>
<tr>
<td>nCount</td>
<td>number of entries in either the lpEvents or lpMMBs parameter. If set to zero, any previous registration is nullified. Arrays specified in both the lpEvents and lpMMBs parameters must have at least the number of entries specified in the nCount parameter. Maximum value for this parameter is MNT_MAX_ASYNC_MSGS.</td>
</tr>
<tr>
<td>lpEvents</td>
<td>pointer to the event handle array. Each event in this array is associated sequentially with the corresponding MMB in the array specified in the lpMMBs parameter. All events must be the manual-reset type.</td>
</tr>
<tr>
<td>lpMMBs</td>
<td>pointer to the LPMMB array. Each element in this array must point an MMB that has been properly initialized and set up just as if it were to be passed to the mntSendMessage( ) function.</td>
</tr>
</tbody>
</table>

Cautions

Each mntRegisterAsyncMessages( ) function call cancels and overrides any previous registration. Specifying a zero in the nCount parameter effectively cancels all notifications. Furthermore, you must not free any buffers described in the MMBs until after you specifically un-register by calling the routine with an nCount of 0. Use this function judiciously and only as necessary because it results in additional resources and workload in the driver space. Before exiting the process or thread, remember to deregister by calling this function with its nCount parameter set to zero.

Please note that the MMB’s that you submit to this call must be all empty messages; that is, you cannot send any command messages. They can only be used to receive messages.
enables receipt of asynchronous messages  
mntRegisterAsyncMessages( )

**Errors**

- **ERROR_BAD_COMMAND**
  - The specified handle does not have an attached stream.

- **ERROR_INVALID_HANDLE**
  - An invalid handle was specified in the argument list.

- **ERROR_INVALID_PARAMETER**
  - An invalid parameter was specified in the argument list.

- **ERROR_NOACCESS**
  - A bad (non-NULL) pointer was passed OR unable to lock down memory.

- **ERROR_NOT_ENOUGH_MEMORY**
  - The driver cannot allocate the required memory for this function.

**Result Messages**

None.

**See Also**

mntSendMessage( )
mntRegisterAsyncStreams() enables receipt of asynchronous stream data

### Name
BOOL mntRegisterAsyncStreams (hDevice, nCount, lpEvents, lpBuffers, lpMSBs)

### Inputs
- **HANDLE** hDevice: device handle
- **ULONG** nCount: number of array elements
- **HANDLE** *lpEvents: event array pointer
- **PVOID** *lpBuffers: buffer array pointer
- **LPMSB** *lpMSBs: MSB array pointer

### Outputs
None.

### Returns
TRUE if successful, FALSE if error

### Includes
qhostlib.h

### Category
stream I/O function

### Mode
synchronous

### Description
The **mntRegisterAsyncStreams()** function enables receipt of asynchronous stream data through a set of Stream Buffer (MSB) structures and corresponding event object handles. As a stream read operation completes, its associated event is set by the driver and the buffer is already filled with the stream data. The calling application must reset the event as soon as the buffer is free for reuse. Until the event is reset, the driver cannot use the associated buffer and MMB to repost the I/O request. This also means that the event must be a manual-reset type.

For each low-latency asynchronous read operation, you should specify two or more MSBs and associated events to ensure that no data will be missed. Otherwise, the driver resorts to a coarse one-second-resolution timer in checking whether the MSB and buffer are ready for reuse as indicated by its event object being in the non-signaled state.

To cancel notification, specify zero (0) in the **nCount** parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the <strong>CreateFile()</strong> function</td>
</tr>
</tbody>
</table>
enables receipt of asynchronous stream data  \texttt{mntRegisterAsyncStreams()}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{nCount}</td>
<td>number of entries in either the \texttt{lpEvents}, \texttt{lpBuffers}, or \texttt{lpMSBs} parameter. If set to zero, any previous registration is nullified. Arrays specified in the \texttt{lpEvents}, \texttt{lpBuffers}, or \texttt{lpMSBs} parameters must have at least the number of entries specified in the \texttt{nCount} parameter. Maximum value for this parameter is MNT_MAX_ASYNC_STRMS.</td>
</tr>
<tr>
<td>\texttt{lpEvents}</td>
<td>pointer to the event handle array. Each event in this array is associated sequentially with the corresponding MSB in the array specified in the \texttt{lpMSBs} parameter. All events must be the manual-reset type.</td>
</tr>
<tr>
<td>\texttt{lpBuffers}</td>
<td>pointer to the buffer array. Each element in this array will hold the data associated with read from the stream.</td>
</tr>
<tr>
<td>\texttt{lpMSBs}</td>
<td>pointer to the LPMSB array. Each element in this array must point to an MSB that has been properly initialized with a timeout and transfer length.</td>
</tr>
</tbody>
</table>

The Stream Buffer (MSB) structure is defined as follows:

```c
typedef struct {
    STRM_HDR    strmHdr;
    ULONG       readCompletionMask;
    USHORT      timeout;
    ULONG       xferLen;
    ULONG       xferDone;
} MSB, *PMSB, *LPMSB;
```

Where:

- \texttt{strmHdr}  stream header returned from \texttt{mntGetStreamHeader()} ( )
- \texttt{readCompletionMask}  mask set in \texttt{mntSetStreamHeader()} ( )
- \texttt{timeout}  same value as set in \texttt{mntSetIOTimeout()} ( )
- \texttt{xferLen}  size of the buffer corresponding to this MSB
- \texttt{xferDone}  returned size from the read
**mntRegisterAsyncStreams()** enables receipt of asynchronous stream data

### Cautions

Each `mntRegisterAsyncStreams()` function call cancels and overrides any previous registration. Specifying a zero in the `nCount` parameter effectively cancels all notifications. Furthermore, you must not free any buffers or MSBs until after you specifically un-register by calling the routine with an `nCount` of 0. Use this function judiciously and only as necessary because it results in additional resources and workload in the driver space. Before exiting the process or thread, remember to deregister by calling this function with its `nCount` parameter set to zero.

Please note that the MMB’s that you submit to this call must be all empty messages; that is, you cannot send any command messages. They can only be used to receive messages.

### Errors

- **ERROR_BAD_COMMAND**
  - The specified handle does not have an attached stream.

- **ERROR_INVALID_FUNCTION**
  - The stream handle specified is of the wrong type.

- **ERROR_INVALID_HANDLE**
  - An invalid handle was specified in the argument list.

- **ERROR_INVALID_PARAMETER**
  - An invalid parameter was specified in the argument list.

- **ERROR_NOACCESS**
  - A bad (non-NULL) pointer was passed OR unable to lock down memory.

- **ERROR_NOT_ENOUGH_MEMORY**
  - The driver cannot allocate the required memory for this function.

### Result Messages

None.

### See Also

`mntRegisterAsyncMessages()`
The `mntSendMessage()` function sends the message specified in the MMB.

Whether or not the call blocks depends on how the `hDevice` parameter was created. If the FILE_FLAG_OVERLAPPED flag was specified in the `CreateFile()` function call, this call blocks immediately, but returns with FALSE. If the user then calls the `GetLastError()` function and it returns `ERROR_IO_PENDING`, the message has been sent successfully, but it will complete at a later time when a reply is received.

If this function is called synchronously, it will not return until all operations are completed. For example, if two reply messages are expected and one is received immediately, the function blocks until the second reply message is received.

### Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hDevice</code></td>
<td>handle to a message path device returned from the <code>CreateFile()</code> function</td>
</tr>
<tr>
<td><code>lpMMB</code></td>
<td>pointer that was returned from a successful call to the <code>mntAllocateMMB()</code> function</td>
</tr>
<tr>
<td><code>lpOverlapped</code></td>
<td>pointer to an OVERLAPPED structure</td>
</tr>
</tbody>
</table>

### Cautions

The application is responsible for managing the OVERLAPPED structure. Refer to 2.2. *Calling Functions Asynchronously* for more details.
`mntSendMessage()` sends the message specified in the MMB

### Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_ADAP_HDW_ERROR</td>
<td>Board is not available to be initialized.</td>
</tr>
<tr>
<td>ERROR_INVALID_HANDLE</td>
<td>An invalid handle was specified in the argument list.</td>
</tr>
<tr>
<td>ERROR_INVALID_PARAMETER</td>
<td>An invalid parameter was specified in the argument list.</td>
</tr>
<tr>
<td>ERROR_MNT_MMB_INVALID_CMDSIZE</td>
<td>Command size is too large.</td>
</tr>
</tbody>
</table>

### Result Messages

None.

### See Also

- `mntAllocateMMB()`  
- `mntFreeMMB()`
builds an MMB, sends it, then synchronously waits for I/O completion. mntSendMessageWait() 

**Name:** BOOL mntSendMessageWait(hDevice, nMsgType, bEmptyMsg, nPayloadSize, lpPayload, nReplyCount, lpDestAddr, lpReplyType, lppReply)

**Inputs:**
- **HANDLE** hDevice • device handle
- **ULONG** nMsgType • type of message to send
- **BOOL** bEmptyMsg • empty message flag
- **ULONG** nPayloadSize • message payload size
- **PVOID** lpPayload • payload pointer
- **ULONG** nReplyCount • replies expected
- **PQCompDesc** lpDestAddr • destination address

**Outputs:**
- **PULONG** lpReplyType • reply message type
- **QMsgRef** *lppReply • reply message pointer

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** message I/O function

**Mode:** synchronous

### Description

The mntSendMessageWait() function builds an MMB, sends it, then synchronously waits for I/O completion. Upon successful return, the function fills in the locations pointed to by lpReplyType and lppReply.

The mntSendMessageWait() function is provided as a convenience; it allocates the required MMB, fills in the MMB and command message header information, sends the message to its destination, and waits for reply message(s). You can achieve the same results by calling mntAllocateMMB(), using the message macros described in *Chapter 4. Macro Reference* to fill in the message header fields, and then calling mntSendMessage().

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the CreateFile() function</td>
</tr>
<tr>
<td>nMsgType</td>
<td>type of message. Typically defined in a header file, such as stddefs.h.</td>
</tr>
<tr>
<td>bEmptyMsg</td>
<td>if TRUE, indicates an empty message, which is expected rather than sent.</td>
</tr>
<tr>
<td>nPayloadSize</td>
<td>message payload size</td>
</tr>
</tbody>
</table>
mntSendMessageWait() builds an MMB, sends it, then synchronously waits for I/O completion.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpPayload</td>
<td>pointer to the message payload structure</td>
</tr>
<tr>
<td>nReplyCount</td>
<td>number of replies expected. The call completes only if the destination address of the reply messages matches the host source address assigned to the device specified in the hDevice parameter.</td>
</tr>
<tr>
<td>lpDestAddr</td>
<td>pointer to the destination component instance address</td>
</tr>
<tr>
<td>lpReplyType</td>
<td>pointer to the reply message type</td>
</tr>
<tr>
<td>*lppReply</td>
<td>pointer to a reply message. Upon return, the caller can examine and access the reply message as needed.</td>
</tr>
</tbody>
</table>

- **Cautions - None.**

- **Errors**
  - ERROR_ADAP_HDW_ERROR: Board is not available to be initialized.
  - ERROR_INVALID_HANDLE: An invalid handle was specified in the argument list.
  - ERROR_INVALID_PARAMETER: An invalid parameter was specified in the argument list.
  - ERROR_MNT_NO_MEM: Not enough memory is available for the MMB.
  - ERROR_MNT_MERCURY_STD_MSG: The message reply type is StdMsgError. Check the reply message payload for details.

- **Result Messages**

None.

- **See Also**

None.
### Description

The `mntSetExitNotify()` function enables notification of Mpath device failure. This function enables the driver to send exit notification to the DM3 board upon failure of the specified Mpath device. To avoid an extraneous notification, you must disable this capability by calling `mntSetExitNotify()` and setting `enable` to FALSE. Otherwise, these notifications can affect system performance and behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>handle to a message path device returned from the <code>CreateFile()</code> function</td>
</tr>
<tr>
<td>board</td>
<td>identifies the DM3 board to which the exit notification should be sent</td>
</tr>
<tr>
<td>enable</td>
<td>on/off toggle for exit notification. Set to TRUE to enable exit notification; set to FALSE to disable exit notification.</td>
</tr>
</tbody>
</table>

### Cautions

None.

### Errors

- **ERROR_BAD_COMMAND**
  - The specified handle does not have an attached stream.

- **ERROR_INVALID_HANDLE**
  - An invalid handle was specified in the argument list.
**mntSetExitNotify( )** enables notification of Mpath device failure

ERROR_INVALID_PARAMETER

- An invalid parameter was specified in the argument, such as an invalid board number.

■ Result Messages

None.

■ See Also

- mntNotifyRegister( )
- mntNotifyUnregister( )
sets the out-of-band stream attributes

**Name:** BOOL mntSetStreamHeader(hDevice, pHeader, ReadCompletionMask)

**Inputs:**
- HANDLE hDevice: device handle
- PSTRM_HDR pHeader: header pointer
- ULONG ReadCompletionMask: mask

**Outputs:** None

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** stream I/O function

**Mode:** synchronous

### Description

The `mntSetStreamHeader()` function sets the out-of-band stream attributes that are defined by the structure pointed to by the `pHeader` parameter. The underlying bulk data stream is passed in blocks between the host and the DM3 platform. These blocks carry attribute data that can control data transfer and provide out-of-band data associated with the stream blocks.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>Stream device handle</td>
</tr>
<tr>
<td>pHeader</td>
<td>pointer to the stream header</td>
</tr>
<tr>
<td>ReadCompletionMask</td>
<td>an optional mask that determines when the read is completed. The user selects when the read is completed by setting the flags defined below:</td>
</tr>
<tr>
<td></td>
<td>COMPLETE_ON_EOD: 0x01</td>
</tr>
<tr>
<td></td>
<td>COMPLETE_ON_EOT: 0x02</td>
</tr>
<tr>
<td></td>
<td>COMPLETE_ON_EOF: 0x04</td>
</tr>
<tr>
<td></td>
<td>COMPLETE_ON_USR1: 0x08</td>
</tr>
<tr>
<td></td>
<td>COMPLETE_ON_USR2: 0x10</td>
</tr>
<tr>
<td></td>
<td>COMPLETE_ON_USR3: 0x20</td>
</tr>
<tr>
<td></td>
<td>COMPLETE_ON_USR4: 0x40</td>
</tr>
<tr>
<td></td>
<td>COMPLETE_ON_USR5: 0x80</td>
</tr>
</tbody>
</table>

The `ReadCompletionMask` as defined below, specifies the out-of-band stream attributes expected after a call to the `ReadFile()` function:
mntSetStreamHeader() sets the out-of-band stream attributes

- The COMPLETE_ON_EOD flag indicates the end of a valid grouping of data blocks. It terminates an operation, such as a data transfer, without closing the stream.

- The COMPLETE_ON_EOT flag indicates the end of a collection of groupings that have been delineated by COMPLETE_ON_EOT flags. Without closing the stream, it marks such operations as a forced termination of a grouping of operations in which the data transfer groupings were buffered onto a stream, but were not yet processed at the time of termination.

- The COMPLETE_ON_EOF flag indicates the end of a file or stream. It is normally set in the last block of a stream when the writer closes the end of that stream.

- The COMPLETE_ON_USERn flags can be used for any application-level purpose.

The pHeader structure is defined as follows:

typedef struct {
    ULONG sequence;
    UCHAR bufFlags;   // MNT_EOD - End of Data = 0x01
    // MNT_EOT - End of Transmission = 0x02
    // MNT_EOF - End of File = 0x04 (equivalent to EOS)
    // MNT_USER1 - User specified flag = 0x08
    // MNT_USER2 - User specified flag = 0x10
    // MNT_USER3 - User specified flag = 0x20
    // MNT_USER4 - User specified flag = 0x40
    // MNT_USER5 - User specified flag = 0x80
    UCHAR encoding;
    UCHAR pad1;             // reserved for future use
    UCHAR sysFlags;         // read-only
    // STREAM_CLOSED = 0x01
    // STREAM_BROKEN = 0x02
    ULONG canTakeLimit;     // read-only
    ULONG initialCanTake;   // read-only
    ULONG currentCanTake;   // read-only
    ULONG requestedSize;    // read-only
    ULONG actualSize;       // read-only
} STRM_HDR, *PSTRM_HDR;

The sequence field is used as an incrementing counter as blocks are written. This field is automatically filled by the lower level stream data block transport code.

The bufFlags field indicates the out-of-band stream attributes as defined below:

- The MNT_EOD flag indicates the end of a valid grouping of data blocks. It terminates an operation, such as a data transfer, without closing the stream.
sets the out-of-band stream attributes \textit{mntSetStreamHeader()} \par

- The \texttt{MNT\_EOT} flag indicates the end of a collection of groupings that have been delineated by \texttt{MNT\_EOD} flags. Without closing the stream, it marks such operations as a forced termination of a grouping of operations in which the data transfer groupings were buffered onto a stream, but were not yet processed at the time of termination.

- The \texttt{MNT\_EOF} flag indicates the end of a file or stream. It is normally set in the last block of a stream when the writer closes its end of the stream.

- The \texttt{MNT\_USERn} flags can be used for any application-level purpose.

The \texttt{encoding} field is set to the calling processor byte ordering convention (big-endian or little-endian).

\section*{Cautions}

None.

\section*{Errors}

\begin{tabular}{ll}
ERROR\_BAD\_COMMAND & \begin{itemize}
  \item The specified handle does not have an attached stream.
\end{itemize} \\
ERROR\_INVALID\_FUNCTION & \begin{itemize}
  \item The stream handle specified is of the wrong type.
\end{itemize} \\
ERROR\_INVALID\_HANDLE & \begin{itemize}
  \item An invalid handle was specified in the argument list.
\end{itemize} \\
ERROR\_INVALID\_PARAMETER & \begin{itemize}
  \item An invalid parameter was specified in the argument list.
\end{itemize}
\end{tabular}

\section*{Result Messages}

None.

\section*{See Also}

- \texttt{mntGetStreamHeader()}
**mntSetStreamIOTimeout( )**  
*sets the stream I/O request timeout value*

<table>
<thead>
<tr>
<th>Name:</th>
<th>BOOL mntSetStreamIOTimeout(hDevice, nTimeout)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
<td></td>
</tr>
</tbody>
</table>
| HANDLE | hDevice | device handle  
| USHORT | nTimeout | timeout |
| Outputs: | None |
| Returns: | TRUE if successful, FALSE if error |
| Includes: | qhostlib.h |
| Category: | stream I/O function |
| Mode: | synchronous |

**Description**

The `mntSetStreamIOTimeout( )` function sets the stream I/O request timeout value (in seconds). If you set the `Timeout` parameter to 0, the driver uses a default timeout of 30 seconds.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>Stream device handle</td>
</tr>
<tr>
<td>nTimeout</td>
<td>timeout value (in seconds) of each stream read or write request</td>
</tr>
</tbody>
</table>

**Cautions - None.**

**Errors**

- **ERROR_BAD_COMMAND**  
  - The specified handle does not have an attached stream.
- **ERROR_INVALID_HANDLE**  
  - An invalid handle was specified in the argument list.
- **ERROR_INVALID_PARAMETER**  
  - An invalid parameter was specified in the argument list.

**Result Messages - None.**

**See Also - None.**
**enables or disables trace statements**

**mntSetTraceLevel()**

- **Name:** BOOL mntSetTraceLevel(TraceLevel, lpTraceDeviceName)
- **Inputs:**
  - ULONG TraceLevel: trace status
  - LPSTR lpTraceDeviceName: trace device name
- **Outputs:** None
- **Returns:** None
- **Includes:** qhostlib.h
- **Category:** debug support function
- **Mode:** synchronous

### Description

The **mntSetTraceLevel()** function enables or disables trace statements. Once this function returns, call **mntTrace()** to send trace statements to a file. Trace information gathered via this function is for program debugging only; use the board-level trace utility for board debugging.

#### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwTraceLevel</td>
<td>trace level. This can be either of the following:</td>
</tr>
<tr>
<td></td>
<td>MNTI_TRACE_LEVEL0: Level 0 disables tracing</td>
</tr>
<tr>
<td></td>
<td>MNTI_TRACE_LEVEL1: Level 1 enables tracing</td>
</tr>
<tr>
<td>lpTraceDeviceName</td>
<td>device name to which tracing information is sent.</td>
</tr>
<tr>
<td></td>
<td>This can be a file, printer, or serial port. Can be set to NULL if tracing is being disabled.</td>
</tr>
</tbody>
</table>

Because the **mntSetTraceLevel()** function internally calls the Windows **CreateFile()**, **WriteFile()**, and **CloseHandle()** functions, the trace output can go to any native Win32 API I/O device. If you are disabling tracing by setting the **dwTraceLevel** parameter to **MNTI_TRACE_LEVEL0**, you can set the **lpTraceDeviceName** parameter to NULL.

The Direct Interface uses the critical section and lock file commands to serialize writing trace statements to the file. Therefore, the trace statements do not interfere with each other in the trace file for multi-threaded and multi-process applications. The DLL creates and initializes a critical section for the trace control block shown below. By default, the DLL initializes the trace level to **MNTI_TRACE_LEVEL0**.
mntSetTraceLevel() enables or disables trace statements

If the dwTraceLevel parameter is set to MNTI_TRACE_LEVEL0 and the current level is MNTI_TRACE_LEVEL1, tracing is disabled.

If the dwTraceLevel parameter is set to MNTI_TRACE_LEVEL1, tracing varies according to the current trace level:

- If the current trace level is MNTI_TRACE_LEVEL0 (trace disabled), the mntSetTraceLevel() function opens a new trace device by calling the CreateFile() function with the name specified in the dwTraceDeviceName parameter.
- If the current trace level is MNTI_TRACE_LEVEL1 (trace enabled), the mntSetTraceLevel() function first closes the current trace device, then opens a new trace device by calling the CreateFile() function with the name specified in the dwTraceDeviceName parameter.

When viewing the debug file, use Write or Wordpad for best results.

■ Cautions

None.

■ Errors

ERROR_INVALID_PARAMETER  • An invalid parameter was specified in the argument list.

ERROR_MNT_NO_TRACE_HANDLE  • The specified trace device could not be opened.

■ Result Messages

None.

■ See Also

- mntSetTrace()
### Description

The `mntTerminateStream()` function cancels a persistent stream identified by `nMercStreamID`. The specified stream must have been opened using `mntAttachMercStream()` with `nModeFlags` set to `MNT_STREAM_FLAG_PERSISTENT`. Before you call the `mntTerminateStream()` function, you should close the stream by calling `mntDetachStream()`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>stream device handle</td>
</tr>
<tr>
<td>nBoardNumber</td>
<td>board number</td>
</tr>
<tr>
<td>nModeFlags</td>
<td>stream attributes for this stream:</td>
</tr>
<tr>
<td></td>
<td><code>MNT_STREAM_FLAG_READ</code> read stream</td>
</tr>
<tr>
<td></td>
<td><code>MNT_STREAM_FLAG_WRITE</code> write stream</td>
</tr>
<tr>
<td>nMercStreamID</td>
<td>identifies an existing stream</td>
</tr>
<tr>
<td>nTimeout</td>
<td>time (in seconds) to wait for a response</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>pointer to an <code>OVERLAPPED</code> structure</td>
</tr>
</tbody>
</table>

### Cautions - None.
### Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_ADAP_HDW_ERROR</td>
<td>Board is not available to be initialized.</td>
</tr>
<tr>
<td>ERROR_INVALID_FUNCTION</td>
<td>The stream handle specified is of the wrong type.</td>
</tr>
<tr>
<td>ERROR_INVALID_HANDLE</td>
<td>An invalid handle was specified in the argument list.</td>
</tr>
<tr>
<td>ERROR_INVALID_PARAMETER</td>
<td>An invalid parameter was specified in the argument list.</td>
</tr>
</tbody>
</table>

### See Also

- mntAttachMercStream( )
- mntDetachMercStream( )

*mntTerminateStream( )* cancels a persistent stream
The `mntTrace()` function sends trace statements to a file, following `printf()` conventions. You must first call the `mntSetTraceLevel()` function to enable tracing and specify the trace output type. Trace information gathered via this function is for program debugging only; use the board-level trace utility for board debugging.

Because each trace statement is prefixed with process and thread IDs, the user can identify the invoking thread in a multi-threaded program. The Direct Interface uses the critical section and lock file commands to serialize writing trace statements to the file. Therefore, the trace statements do not interfere with each other in the trace file for multi-threaded and multi-process applications.

Each trace statement should include the function from which it is invoked and other information that can help the user debug the problem. Once the format string size is expanded (filled in), it should be less than 200 characters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pszFmt</code></td>
<td>format string. Expanded string size should be less than 200 characters.</td>
</tr>
<tr>
<td>/* args */</td>
<td>arguments to be embedded into the format string</td>
</tr>
</tbody>
</table>

**Cautions**

None.
mntTrace() sends trace statements to a file

■ Errors

None.

■ Result Messages

None.

■ See Also

• mntSetTraceLevel()
generates a message transaction ID

mntTransGen()

Name: QTrans mntTransGen(void)
Inputs: None
Outputs: None
Returns: QTrans TransactionID • transaction identifier
Includes: qhostlib.h
Category: debug support function
Mode: synchronous

Description

The mntTransGen() function generates a message transaction ID. This function returns a pseudo-unique transaction identifier for use in messages. This ID is unique within the QTrans type range until the mntTransGen() function has generated all IDs, at which time they begin to be repeated.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransactionID</td>
<td>message transaction identifier</td>
</tr>
</tbody>
</table>

Cautions

None.

Errors

None.

Result Messages

None.

See Also

None.
qMsgVarFieldGet( )  gets typed fields from a message payload

| Name: | BOOL qMsgVarFieldGet (msg, count, pOffset, fieldDef, pTarget, ...) |
| Inputs: | QMsgRef msg ● referenced message  
UInt32 count ● number of fields to get  
UInt32 *pOffset ● offset  
QMsgField fieldDef ● data element |
| Outputs: | void *pTarget ● referenced variable |
| Returns: | TRUE if successful, FALSE if error |
| Includes: | qhostlib.h |
| Category: | message I/O function |
| Mode: | synchronous |

### Description

The qMsgVarFieldGet( ) function gets typed fields from a message payload.

This function performs a structured copy of the contents of the number of fields specified by `count` from the message referenced by `msg` into locally defined variables.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>reference to a message that contains fields to be copied</td>
</tr>
<tr>
<td>count</td>
<td>number of fields to copy from the message; must match the number of (fieldDef, pTarget) pairs specified in the function call</td>
</tr>
<tr>
<td>pOffset</td>
<td>pointer to a variable that contains the offset of a field within the message body. When the function is called, the variable specifies the offset of the first data field to copy; if zero, fields are copied according to the offsets contained in the field definitions. When the function completes, the variable is updated to reference the next field that has not been copied.</td>
</tr>
<tr>
<td>fieldDef</td>
<td>field definition of a data field to copy; always paired with a pTarget. Field definitions contain the data type, size, and offset within the buffer of the field. If the variable referenced by pOffset is non-zero, the offset is ignored and the function copies successive fields starting at the specified offset.</td>
</tr>
</tbody>
</table>
**gets typed fields from a message payload**  

$q\text{MsgVarFieldGet()}$  

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pTarget</td>
<td>pointer to the variable where the copied contents of a field is placed; always paired with a fieldDef which defines the data type of the result.</td>
</tr>
</tbody>
</table>

The count argument specifies the number of $(\text{fieldDef}, \text{pTarget})$ argument pairs that follow the pOffset argument. For each pair, the data element in the message data defined by the fieldDef is copied into the variable referenced by the associated pTarget. The data being copied is interpreted as a particular data type defined by fieldDef. The message data is converted from a standard message format into the native format of the specified data type of the executing processor.

After all fields have been copied, the variable referenced by the pOffset argument is updated to reference the next uncopied field in the message.

If the variable referenced by the pOffset argument is non-zero when $q\text{MsgVarFieldGet()}$ is called, the list of $(\text{fieldDef}, \text{pTarget})$ pairs is interpreted as containing only generic field definitions. A field definition normally contains the data type, number of elements, and offset within the buffer of the field. A generic field definition contains only the data type and number of elements. If a non-zero offset is specified, the copy from the buffer begins at the offset and proceeds using the list of field definitions to perform the copies and translations.

If the variable referenced by the pOffset argument is zero when $q\text{MsgVarFieldGet()}$ is called, all $(\text{fieldDef}, \text{pTarget})$ pairs containing absolute field definitions must precede any generic definitions because the first generic definition is interpreted as a field immediately following the last absolute definition.

Field definitions are message-specific values which encode the data type, number of elements, and offset within a message. They are normally created by an off-line tool (the MMDL translator) which generates a header file containing the field definitions for a message or group of messages.

This function provides a mechanism that can read an entire message data structure from the message into a local structure and also provides support for variable-length message data. The MMDL tool which generates the message field definitions also generates local structure definitions and data access macros that call this library function to copy the entire body of the message.
qMsgVarFieldGet() gets typed fields from a message payload

The following types can be encoded within field definitions:

<table>
<thead>
<tr>
<th>QDataType</th>
<th>Description (typedef)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT_INT8</td>
<td>8-bit signed integer (Int8)</td>
</tr>
<tr>
<td>QT_INT16</td>
<td>16-bit signed integer (Int16)</td>
</tr>
<tr>
<td>QT_INT24</td>
<td>24-bit signed integer (Int24)</td>
</tr>
<tr>
<td>QT_INT32</td>
<td>32-bit signed integer (Int32)</td>
</tr>
<tr>
<td>QT_UINT8</td>
<td>8-bit unsigned integer (UInt8)</td>
</tr>
<tr>
<td>QT_UINT16</td>
<td>16-bit unsigned integer (UInt16)</td>
</tr>
<tr>
<td>QT_UINT24</td>
<td>24-bit unsigned integer (UInt24)</td>
</tr>
<tr>
<td>QT_UINT32</td>
<td>32-bit unsigned integer (UInt32)</td>
</tr>
<tr>
<td>QT_CHAR</td>
<td>1 Character in native format (Char)</td>
</tr>
<tr>
<td>QT_MEMREF</td>
<td>Reference to allocated global memory (QMemRef)</td>
</tr>
<tr>
<td>QT_STREAMREF</td>
<td>Processor independent open stream reference (QStreamRef)</td>
</tr>
<tr>
<td>QT_ATTR</td>
<td>Reference to component attribute (QAttr)</td>
</tr>
<tr>
<td>QT_PARM</td>
<td>Reference to parameter (QParm)</td>
</tr>
<tr>
<td>QT_COMPDESC</td>
<td>Reference to component descriptor (QCompDesc)</td>
</tr>
<tr>
<td>QT_BUFREF</td>
<td>Reference to a buffer (QBufRef)</td>
</tr>
</tbody>
</table>

1 Native format character strings are converted one character per addressable location; packed strings are converted with the characters packed within a word in the order supported by the processor.

NOTE: If an integer or unsigned integer type is converted from a wider format (for example, QT_INT16 on a 24-bit word processor), the high-order bits beyond the width of the target type are ignored, which can cause unexpected results if the value is out of the range of the target type. If the conversion is to a wider format, the value is sign-extended if it is an integer type or zero-extended if it is an unsigned integer type.

■ Cautions

qMsgVarFieldGet() performs conversions from a DM3 standard representation of a data type into a processor-specific version of the type. If the type cannot be converted to a valid representation—for example, a 32-bit integer type on a 24-bit processor—the results are undefined.
gets typed fields from a message payload  qMsgVarFieldGet()

- Errors

  ERROR_INVALID_PARAMETER  • An invalid parameter was specified in the argument list.

- See Also

  • qMsgVarFieldPut()
### qMsgVarFieldPut( )

**Name:** BOOL qMsgVarFieldPut (msg, count, pOffset, fieldDef, pSource,...)

**Inputs:**
- QMsgRef msg: referenced message
- UInt32 count: number of fields
- UInt32 *pOffset: offset
- QMsgField fieldDef: data element
- void *pSource: referenced variable

**Outputs:** None

**Returns:** TRUE if successful, FALSE if error

**Includes:** qhostlib.h

**Category:** Messaging Services

**Mode:** Synchronous

---

### Description

The **qMsgVarFieldPut( )** function puts typed fields into a message payload.

This function performs a structured copy of the contents of the number of fields specified by **count** into the message referenced by **msg** from locally defined variables.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>reference to a message that contains fields to be filled</td>
</tr>
<tr>
<td>count</td>
<td>number of fields to fill in the message; must match the number of (fieldDef, pSource) pairs specified in the function call</td>
</tr>
<tr>
<td>pOffset</td>
<td>pointer to a variable that contains the offset of a field within the message body. When the function is called, the variable specifies the offset of the first data field to fill; if zero, fields are filled according to the offsets contained in the field definitions. When the function completes, the variable is updated to reference the next field that has not been filled.</td>
</tr>
<tr>
<td>fieldDef</td>
<td>field definition of a message data field to fill; always paired with a pSource. Field definitions contain the data type, size, and offset within the buffer of the field. If the variable referenced by pOffset is non-zero, the offset is ignored and the function fills successive fields starting at the specified offset.</td>
</tr>
</tbody>
</table>
puts typed fields into a message payload \( q\text{MsgVarFieldPut}( ) \)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p\text{Source} )</td>
<td>pointer to the variable that contains the data to be copied into a field; always paired with a \text{fieldDef} which defines the size and type of the source data.</td>
</tr>
</tbody>
</table>

The count argument specifies the number of \((\text{fieldDef}, \ p\text{Source})\) argument pairs that follow the \( p\text{Offset} \) argument. For each pair, the data element in the message defined by the \text{fieldDef} is copied from the variable referenced by the associated \( p\text{Source} \) into the message. The data being copied is interpreted as a particular data type defined by \text{fieldDef}. The message data is converted into a standard message format from the native format of the specified data type of the executing processor. After all fields have been copied, the variable referenced by the \( p\text{Offset} \) argument is updated to reference the next uncopied field in the message.

If the variable referenced by the \( p\text{Offset} \) argument is non-zero when \( q\text{MsgVarFieldPut}( ) \) is called, the list of \((\text{fieldDef}, \ p\text{Source})\) pairs are interpreted as containing only generic field definitions. A field definition normally contains the data type, number of elements, and offset with buffer of the field. A generic field definition contains only the data type and number of elements. If a non-zero offset is specified, the copy into the buffer begins at the offset and proceeds using the list of field definitions to perform the copies and translations.

If the variable referenced by the \( p\text{Offset} \) argument is zero when \( q\text{MsgVarFieldPut}( ) \) is called, all \((\text{fieldDef}, \ p\text{Source})\) pairs containing absolute field definitions must precede any generic definitions because the first generic definition is interpreted as a field immediately following the last absolute definition.

Field definitions are message-specific values which encode the data type, number of elements, and offset within a message for a field of the message data area. They are normally created by an off-line tool (the MMDL translator) which generates a header file containing the field definitions for a message or group of messages.

This function provides a mechanism that can write an entire message data structure to the message from a local structure and also provides support for variable-length message data. The MMDL tool which generates the message data field definitions also generates local structure definitions and data access macros that call this library function to copy the entire message data area.

The following types can be encoded within field definitions:
qMsgVarFieldPut( )  puts typed fields into a message payload

<table>
<thead>
<tr>
<th>QDataType</th>
<th>Description (typedef)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT_INT8</td>
<td>8-bit signed integer (Int8)</td>
</tr>
<tr>
<td>QT_INT16</td>
<td>16-bit signed integer (Int16)</td>
</tr>
<tr>
<td>QT_INT24</td>
<td>24-bit signed integer (Int24)</td>
</tr>
<tr>
<td>QT_INT32</td>
<td>32-bit signed integer (Int32)</td>
</tr>
<tr>
<td>QT_UINT8</td>
<td>8-bit unsigned integer (UInt8)</td>
</tr>
<tr>
<td>QT_UINT16</td>
<td>16-bit unsigned integer (UInt16)</td>
</tr>
<tr>
<td>QT_UINT24</td>
<td>24-bit unsigned integer (UInt24)</td>
</tr>
<tr>
<td>QT_UINT32</td>
<td>32-bit unsigned integer (UInt32)</td>
</tr>
<tr>
<td>QT_CHAR</td>
<td>Character in native format (Char)</td>
</tr>
<tr>
<td>QT_MEMREF</td>
<td>Reference to allocated global memory (QMemRef)</td>
</tr>
<tr>
<td>QT_STREAMREF</td>
<td>Processor independent open stream reference (QStreamRef)</td>
</tr>
<tr>
<td>QT_ATTR</td>
<td>Reference to component attribute (QAttr)</td>
</tr>
<tr>
<td>QT_PARM</td>
<td>Reference to parameter (QParm)</td>
</tr>
<tr>
<td>QT_COMPDESC</td>
<td>Reference to component descriptor (QCompDesc)</td>
</tr>
<tr>
<td>QT_BUFREF</td>
<td>Reference to a buffer (QBufRef)</td>
</tr>
</tbody>
</table>

1 Native format character strings are converted one character per addressable location; packed strings are converted with the characters packed within a word in the order supported by the processor.

**NOTE:** If an integer or unsigned integer type is converted from a wider format (for example, QT_INT16 on a 24-bit word processor), the high-order bits beyond the width of the target type are ignored, which can cause unexpected results if the value is out of the range of the target type. If the conversion is to a wider format, the value is sign-extended if it is an integer type or zero-extended if it is an unsigned integer type.

**Cautions**

qMsgVarFieldPut( ) performs conversions to a DM3 standard representation of a data type from a processor-specific version of the type. Conversions to data types which are not supported by the processor may have unexpected results.

**Errors**

ERROR_INVALID_PARAMETER  • An invalid parameter was specified in the argument list.
puts typed fields into a message payload  

See Also

- qMsgVarFieldGet()
4. Macro Reference

The DM3 Direct Interface includes macros which allow you to easily set and retrieve message fields. This chapter contains a brief description of DM3 messages, Multiple Message Block (MMB) contents, and information on the message-related macros in the DM3 Direct Interface.

A DM3 message has a fixed-format header and may optionally have a body that contains additional data in typed fields. A DM3 message body is also called a message payload. All DM3 messages that are sent and received are carried or contained in an MMB “wrapper” structure.

The following types of macros are part of the DM3 Direct Interface:
- MMB control header macros
- DM3 message pointer macros
- DM3 message header macros
- DM3 message payload macros

4.1. Multiple Message Block

All DM3 messages that are sent and received are carried or contained in a multiple message block (MMB) “wrapper” structure, which is acquired by calling the mntAllocateMMB() function. As shown in Figure 4, an MMB structure consists of two or three sections in sequence: MMB control header, command message header and command message payload (optional). An MMB can also contain one or more reply messages, each of which is a complete structure of type QMsg with a possible payload attached.

The header and payload information in an MMB is in processor-specific format, based on the processor’s endian-type. Although the MMB structure is defined in an include file, it should be treated opaquely. Use the Direct Interface macros to resolve the endian-type issues; do not access the MMB structure directly.
Figure 4. General MMB Structure

4.2. MMB Control Header Macros

This section contains an alphabetical listing of the multiple message block (MMB) control header macros defined in *dllnti.h*. These macros allow you to get and set the control header fields in an MMB.
4. Macro Reference

MNT_GET_MMB_ACTUAL_REPLY_COUNT(lpMMB, UCHAR *ActualReplyCount)

This macro retrieves the actual reply messages contained in the specified MMB. (The number of actual reply messages may be different from the number of expected reply messages.)

lpMMB is a pointer to the desired multiple message block (MMB).

*ActualReplyCount is the number of reply messages.

MNT_GET_MMB_CMD_SIZE(lpMMB, USHORT *CmdSize)

This macro retrieves the command message size contained in the specified MMB.

lpMMB is a pointer to the desired multiple message block (MMB).

*CmdSize is the size of the command message.

MNT_GET_MMB_CMD_TIMEOUT(lpMMB, USHORT *Timeout)

This macro retrieves the timeout that was set for the command message in the specified MMB.

lpMMB is a pointer to the desired multiple message block (MMB).

*Timeout is the timeout value (in seconds).

MNT_GET_MMB_CURRENT_REPLY_OFFSET(lpMMB, USHORT *ReplyOffset)

This macro retrieves the offset for the first reply message in the specified MMB.

lpMMB is a pointer to the desired multiple message block (MMB).
*ReplyOffset* is the offset location of the first reply message.

MNT_GET_MMB_EMPTY_MSG (lpMMB, *value)

This macro retrieves a particular I/O completion flag setting in the specified message block. EMPTY_MSG is an optional flag setting used to identify an *empty message* MMB.

- **lpMMB** is a pointer to the desired multiple message block (MMB).
- **value** is the completion flag setting, where 1 indicates the flag is set and 0 indicates the flag is not set.

MNT_GET_MMB_EXPECTED_REPLY_COUNT(lpMMB, UCHAR *ExpectedReplyCount)

This macro retrieves the number of expected reply messages in the specified MMB. (The number of actual reply messages may be different from the number of expected reply messages.)

- **lpMMB** is a pointer to the desired multiple message block (MMB).
- **ExpectedReplyCount** is the number of reply messages that were expected.

MNT_GET_MMB_MATCH_ON_DEST_ADDR(lpMMB, *value)

This macro retrieves a particular I/O completion flag setting in the specified message block. MATCH_ON_DEST_ADDR is an optional flag setting that enables you to receive reply messages only from the same component instance specified in the command message of the MMB.

- **lpMMB** is a pointer to the desired multiple message block (MMB).
- **value** is the completion flag setting, where 1 indicates the flag is set and 0 indicates the flag is not set.
MNT_GET_MMB_MATCH_ON_MSGTYPE (lpMMB, *value)

This macro retrieves a particular I/O completion flag setting in the specified message block. MATCH_ON_MSGTYPE is an optional flag setting that enables you to receive reply messages returned with the same message type as in the message sent.

- **lpMMB** is a pointer to the desired multiple message block (MMB).
- **value** is the completion flag setting, where 1 indicates the flag is set and 0 indicates the flag is not set.

MNT_GET_MMB_MATCH_ON_SRC_ADDR (lpMMB, *value)

This macro retrieves a particular I/O completion flag setting in the specified message block. MATCH_ON_SRC_ADDR is a required flag that is set by default. When this flag is set, messages will not complete unless the destination address of the incoming message matches the source address of the command message in the message block.

- **lpMMB** is a pointer to the desired multiple message block (MMB).
- **value** is the completion flag setting, where 1 indicates the flag is set and 0 indicates the flag is not set.

MNT_GET_MMB_MATCH_ON_TRANS_ID (lpMMB, *value)

This macro retrieves a particular I/O completion flag setting in the specified message block. MATCH_ON_TRANSACTION_ID is an optional flag setting that enables you to receive reply messages returned with the same transaction ID as in the message sent.

- **lpMMB** is a pointer to the desired multiple message block (MMB).
- **value** is the completion flag setting, where 1 indicates the flag is set and 0 indicates the flag is not set.
DM3 Direct Interface Function Reference for Windows NT

MNT_GET_MMB_REPLY_MAX_SIZE(lpMMB, USHORT *ReplySize)
This macro retrieves the reply message size allocation for the specified message block.

- lpMMB is a pointer to the desired multiple message block (MMB).
- *ReplySize is the size allocated for reply messages.

MNT_SET_MMB_CMD_SIZE(lpMMB, USHORT CmdSize)
This macro sets the command message size contained in the specified MMB.

- lpMMB is a pointer to the desired multiple message block (MMB).
- CmdSize is the size of the command message.

MNT_SET_MMB_CMD_TIMEOUT(lpMMB, USHORT Timeout)
This macro sets the length of time to wait before indicating failure for the command message in the specified MMB.

- lpMMB is a pointer to the desired multiple message block (MMB).
- Timeout is the timeout value (in seconds).

MNT_SET_MMB_EMPTY_MSG (lpMMB)
This macro sets a particular I/O completion flag in the specified message block. EMPTY_MSG is an optional flag setting that identifies an empty message MMB that has no command message but has room for a specified number of reply messages. Empty message MMBs are used in conjunction with the optional
MATCH_ON_MSG_TYPE flag to receive asynchronous messages such as alarms or events.

lpMMB is a pointer to the desired multiple message block (MMB).

MNT_SET_MMBEXPECTED_REPLY_COUNT(lpMMB, UCHAR ExpectedReplyCount)

This macro sets the number of expected reply messages in the specified MMB and is typically used for empty message MMBs. (The number of actual reply messages may be different from the number of expected reply messages.)

lpMMB is a pointer to the desired multiple message block (MMB).

ExpectedReplyCount is the number of reply messages that were expected.

MNT_SET_MMB_MATCH_ON_DEST_ADDR (lpMMB)

This macro sets a particular I/O completion flag in the specified message block. MATCH_ON_DEST_ADDR is an optional flag setting that enables you to receive reply messages only from the same component instance specified in the command message of the MMB. When this flag is set, messages will not complete unless the source address of the incoming message matches the destination address of the command message in the message block.

lpMMB is a pointer to the desired multiple message block (MMB).

MNT_SET_MMB_MATCH_ON_MSGTYPE (lpMMB)

This macro sets a particular I/O completion flag in the specified message block. MATCH_ON_MSGTYPE is an optional flag setting that enables you to receive reply messages returned with the same message type as in the message sent. When this flag is set, messages will not complete unless the message type of the incoming message matches the message type of the command message in the
message block. Use this flag in conjunction with an empty message to receive asynchronous messages such as alarms or events.

  lpMMB is a pointer to the desired multiple message block (MMB).

MNT_SET_MMB_MATCH_ON_SRC_ADDR (lpMMB)

This macro sets a particular I/O completion flag in the specified message block. MATCH_ON_SRC_ADDR is a required flag that is set by default. When this flag is set, messages will not complete unless the destination address of the incoming message matches the source address of the command message in the message block.

  lpMMB is a pointer to the desired multiple message block (MMB).

MNT_SET_MMB_MATCH_ON_TRANS_ID (lpMMB)

This macro sets a particular I/O completion flag in the specified message block. MATCH_ON_TRANSACTION_ID is an optional flag setting that enables you to receive reply messages returned with the same transaction ID as in the message sent. When this flag is set, messages will not complete unless the transaction ID of the incoming message matches the transaction ID of the command message in the message block.

  lpMMB is a pointer to the desired multiple message block (MMB).

MNT_SET_MMB_REPLY_MAX_SIZE(lpMMB, USHORT ReplySize)

This macro sets the reply message size allocation contained in the specified message block.

  lpMMB is a pointer to the desired multiple message block (MMB).
  ReplySize is the size allocated for reply messages in the MMB.
4.3. DM3 Message Macros

An MMB consists of two or three sections in sequence: MMB control header, command message header, and command message payload (optional). An MMB can also contain one or more reply messages, each of which is a complete structure of type QMsg with a possible payload attached. The macros in the following sections are used to find, set, and retrieve the message headers of both command and reply messages from within the MMB structure.

4.3.1. DM3 Message Pointer Macros

This section contains an alphabetical listing of the DM3 message pointer macros defined in dllmnti.h. Use the following macros on a specified multiple message block (MMB) to get the pointer to the command or reply QMsg structures that it contains. After you have the pointer to the QMsg structure, use the information described in 4.3.2. DM3 Message Header Macros and 4.4. DM3 Messages with Payloads to access the message header and payload data.

MNT_GET_CMD_QMSG(LPMMB lpMMB, QMsgRef *pMsg)

This macro retrieves a pointer to the command message contained in the specified MMB.

lpMMB is a pointer to the desired multiple message block (MMB).

*pMsg identifies the location of the command message in the specified MMB.

MNT_GET_REPLY_QMSG(LPMMB lpMMB, ULONG ReplyNumber, QMsgRef *pMsg)

This macro retrieves a pointer to a designated reply message contained in the specified MMB. (Use the MNT_GET_MMB_REPLY_MAX_SIZE macro first to determine the number of reply messages in the MMB.)

lpMMB is a pointer to the desired multiple message block (MMB).
DM3 Direct Interface Function Reference for Windows NT

ReplyNumber identifies the reply message for which a pointer is desired.
*pMsg identifies the location of the reply message in the specified MMB.

4.3.2. DM3 Message Header Macros

This section contains an alphabetical listing of the DM3 message header macros defined in qmsg.h. Use the macros to set and retrieve header information from command and reply messages contained in an MMB wrapper.

QMSG_GET_DESTADDR (QMsgRef pMsg, QCompDesc *pDestAddress)

This macro retrieves the destination address of the specified message.

pMsg is a pointer to the desired message.

*pDestAddress is the message destination’s address.

QMSG_GET_MSGSIZE (QMsgRef pMsg, ULONG *MsgSize)

This macro retrieves the size of the specified message.

pMsg is a pointer to the desired message.

*MsgSize is the size of the specified message (in bytes).

QMSG_GET_MSGTYPE (QMsgRef pMsg, ULONG *MessageType)

This macro retrieves the type of the specified message.

pMsg is a pointer to the desired message.

*MessageType is the message type.
QMSG_GET_SRCADDR (QMsgRef pMsg, QCompDesc *pSourceAddress)

This macro retrieves the source address of the specified message.

- **pMsg** is a pointer to the desired message.
- **pSourceAddress** is the message originator’s address.

QMSG_GET_TRANS (QMsgRef pMsg, QTrans *TransactionID)

This macro retrieves the transaction identifier of the specified message.

- **pMsg** is a pointer to the desired message.
- **TransactionID** is the message’s transaction identifier.

QMSG_SET_DESTADDR (QMsgRef pMsg, QCompDesc pDestAddress)

This macro sets the destination address of the specified message.

- **pMsg** is a pointer to the desired message.
- **pDestAddress** is the message destination’s address.

QMSG_SET_MSGSIZE (QMsgRef pMsg, ULONG MsgSize)

This macro sets the size of the specified message.

- **pMsg** is a pointer to the desired message.
- **MsgSize** is the size of the specified message (in bytes).
QMSG_SET_MSGTYPE (QMsgRef pMsg, ULONG MessageType)

This macro sets the type of the specified message.

- `pMsg` is a pointer to the desired message.
- `MessageType` is the message type.

QMSG_SET_SRCADDR (QMsgRef pMsg, QCompDesc pSourceAddress)

This macro sets the source address of the specified message.

- `pMsg` is a pointer to the desired message.
- `pSourceAddress` is the message originator’s address.

QMSG_SET_TRANS (QMsgRef pMsg, QTrans TransactionID)

This macro sets the transaction identifier of the specified message.

- `pMsg` is a pointer to the desired message.
- `TransactionID` is the message’s transaction identifier.

4.4. DM3 Messages with Payloads

This section contains tables listing DM3 messages that require the use of payload macros defined in `qmsg.h`. Use the macros to set and retrieve payload information from command and reply messages contained within an MMB structure.

4.4.1. Messages With Fixed Payloads

DM3 messages may have a body with a known, predefined size, called a fixed payload. Table 8 lists messages containing fixed payload information and maps
them to the Direct Interface functions that can receive these messages. Refer to the specific function description for details on extracting the payload contents.

### Table 8. Messages with Fixed Payloads

<table>
<thead>
<tr>
<th>Message Name</th>
<th>Received by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>QClusterResult</td>
<td>mntClusterAllocate( ), mntClusterByComp( ), mntClusterCreate( ), mntClusterFind( )</td>
</tr>
<tr>
<td>QClusterUnlockCmplt</td>
<td>mntClusterConfigUnlock( )</td>
</tr>
<tr>
<td>QComponentResult</td>
<td>mntClusterCompByAttr( ), mntCompAllocate( ), mntCompFind( )</td>
</tr>
<tr>
<td>QResultError</td>
<td>mntClusterActivate( ), mntClusterAllocate( ), mntClusterByComp( ), mntClusterByAttr( ), mntClusterConfigLock( ), mntClusterConnect( ), mntClusterCreate( ), mntClusterDeactivate( ), mntClusterDestroy( ), mntClusterDisconnect( ), mntClusterFind( ), mntClusterFree( ), mntClusterSlotInfo( ), mntClusterTSAssign( ), mntClusterTSUnassign( ), mntCompAllocate( ), mntCompFind( ), mntCompFindAll( ), mntCompFree( ), mntCompUnuse( ), mntCompUse( ), mntNotifyRegister( ), mntNotifyUnregister( )</td>
</tr>
</tbody>
</table>

#### 4.4.2. Messages with Variable Payloads

A variable payload is the body of a DM3 message that includes one or more variable fields. The `qMsgVarFieldGet()` and `qMsgVarFieldPut()` functions must be used to access the variable portion of a message payload. Table 9 lists messages containing variable payloads and maps them to the Direct Interface functions that can receive these messages. Refer to the specific function description for details on extracting the variable payload contents.
### Table 9. Messages with Variable Payloads

<table>
<thead>
<tr>
<th>Message Name</th>
<th>Received by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>QClnusterSlotInfoResult</td>
<td>mntClusterSlotInfo( )</td>
</tr>
<tr>
<td>QComponentMultipleResult</td>
<td>mntCompFindAll( )</td>
</tr>
<tr>
<td>QFailureNotify</td>
<td>mntNotifyRegister( )</td>
</tr>
</tbody>
</table>
5. Data Types, Structures, and Error Codes

This chapter contains information on:

- Data types
- Data structures
- Error code definitions

5.1. Data Types

The Direct Interface host library is distributed with a number of include files. For DM3 Kernel-related data types, consult mercdefs.h. For standard DM3 messages and parameters, consult stddefs.h. Component-related structures are defined in qcomplib.h. Some of the most common data types are listed in the following table.

Table 10. Data Type Definitions

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct QCompAttr</td>
<td>Structure that contains a component attribute identifier (the key) and a specific attribute value associated with that key.</td>
</tr>
<tr>
<td>struct QCompDesc</td>
<td>Structure that defines a component instance. It is a record that contains board, processor, and component type identifiers; and the instance number. It is the component instance address for all messages.</td>
</tr>
<tr>
<td>struct QMsg</td>
<td>Local representation of the standard DM3 message structure. You should access this structure only through the access macros described in Chapter 4. Macro Reference.</td>
</tr>
</tbody>
</table>
Data Type  | Definition
---|---
UInt24 QTrans  | Transaction identifier that is a standard element of a DM3 message. Use it as a parameter in a function call that returns an asynchronous message as a result. The transaction ID is returned in the reply message. Transaction identifiers should be unique within each process.

5.2. Data Structures

This section alphabetically lists the data structures used by the Direct Interface functions and discusses the fields they contain.

5.2.1. MSB Stream Buffer Structure

This data structure is used by the `mntRegisterAsyncStreams()` function and is defined in `mmb.h`.

```c
typedef struct {
    STRM_HDR strmHdr;
    ULONG readCompletionMask;
    USHORT timeout;
    ULONG xferLen;
    ULONG xferDone;
} MSB, *PMSB, *LPMSB;
```

- `strmHdr` stream header returned from `mntGetStreamHeader()`
- `readCompletionMask` mask set in `mntSetStreamHeader()`
- `timeout` same timeout value as set in `mntSetIOTimeout()`
- `xferLen` size of the buffer corresponding to the MSB
- `xferDone` returned size from the read
5. Data Types, Structures, and Error Codes

5.2.2. STRM_HDR Stream Header Structure

This data structure is defined in `mmb.h`.

typedef struct {
    ULONG sequence;
    UCHAR buffFlags;   // MNT_EOD - End of Data = 0x01
    // MNT_EOT - End of Transmission = 0x02
    // MNT_EOF - EndOfFile=0x04(equivalent to EOS)
    // MNT_USER1 - User specified flag = 0x08
    // MNT_USER2 - User specified flag = 0x10
    // MNT_USER3 - User specified flag = 0x20
    // MNT_USER4 - User specified flag = 0x40
    // MNT_USER5 - User specified flag = 0x80
    UCHAR encoding;
    UCHAR pad1;            // reserved for future use
    UCHAR sysFlags;        // read-only
    // STREAM_CLOSED = 0x01
    // STREAM_BROKEN = 0x02
    ULONG canTakeLimit;    // read-only
    ULONG initialCanTake;  // read-only
    ULONG currentCanTake;  // read-only
    ULONG requestedSize;   // read-only
    ULONG actualSize;      // read-only
} STRM_HDR, *PSTRM_HDR;

sequence used as an incrementing counter as blocks are written. This field is automatically filled by the lower level stream data block transport code.
bufFlags  indicates the out-of-band stream attributes as defined below:

- The MNT_EOD flag indicates the end of a valid grouping of data blocks. It terminates an operation, such as a data transfer, without closing the stream.

- The MNT_EOT flag indicates the end of a collection of groupings that have been delineated by MNT_EOD flags. Without closing the stream, it marks such operations as a forced termination of a grouping of operations in which the data transfer groupings were buffered onto a stream, but were not yet processed at the time of termination.

- The MNT_EOF flag indicates the end of a file or stream. It is normally set in the last block of a stream when the writer closes its end of the stream.

- The MNT_USERn flags can be used for any application-level purpose.

encoding  set to the calling processor byte ordering convention (big-endian or little-endian)

5.2.3. STRM_INFO Stream Information Structure

This data structure is defined in qstream.h.

typedef struct {
    int       NumStrmGroups;
    int       DataBlockSize;
    STRM_GROUP_CFG StrmGroups[MNT_STREAM_MAX_NUM_GROUPS];
} STRM_INFO, *PSTRM_INFO;

typedef struct {
    UInt32    GroupID;
    UInt32    NumStreams;
    UInt32    StreamSize;
} STRM_GROUP_CFG;

NumStrmGroups  defines the number of stream groups available. A stream group is used for defining a number of streams with different stream size. (Maximum value is 20.)
5. Data Types, Structures, and Error Codes

DataBlockSize defines the default data block size, currently set at 4032 bytes.

5.2.4. QBoardAttr Board Attribute Structure

This data structure is used by the mntGetBoardsByAttr( ) function and is defined in qmsg.h.

typedef struct {
    char     ValueName[MNT_MAX_VALUE_NAME_SIZE];
    ULONG    ValueType;
    char     Value[MNT_MAX_VALUE_SIZE];
    ULONG    BoardNo;
} QBoardAttr, *PQBoardAttr;

ValueName contains a NULL terminated string specifying the name of the value which matched.

ValueType one of the Win32 registry types; REG_DWORD, REG_SZ, or REG_MULTISZ.

Value current value of the value named in ValueName.

BoardNo contains the logical board ID of the board which contained the matching attribute.

5.2.5. QCompAttr Component Attribute Structure

A value of type QCompAttr (defined in qcomplib.h) is a structure of the format:

typedef struct {
    ULONG  key;
    LONG   value;
} QCompAttr, *PQCompAttr;

The key / value pairs described below always occur in arrays. The end of the array is marked with special null values.
key

Uniquely identifies component attribute type. Each identifier key is defined as either unique (only one QCompAttr structure and hence only one value associated with key) or shared (multiple QCompAttr structures with different values may be associated with key). Two standard keys are defined which identify attributes that should be defined for every component. These required attribute keys are:

- SysAttrCompType: Generic component type attribute
- SysAttrCompId: Unique component ID attribute

Additionally, there are four special values defined for the key field which function as operators in a list of QCompAttr structures:

- QATTR_NOT: Used to effect a non-match in selection by attribute
- QATTR_OR: Used to logically OR two attributes in selection by attribute
- QATTR_AND: Used to logically AND two attributes in selection by attribute
- QATTR_NULL: Null key

value

Encoded value of attribute. If no specific value is to be specified for the attribute, the canonical value QATTR_ANY should be set in the value field; this value is equal to the most negative 32-bit integer, which is unavailable as an attribute value. Refer to Table 11 for a list of possible key / value pairs.

**Table 11. Component Attribute Values**

<table>
<thead>
<tr>
<th>Attribute Key</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SysAttrCompType</td>
<td>StdPlayer</td>
<td>A standard player component</td>
</tr>
<tr>
<td>SysAttrCompType</td>
<td>StdRecorder</td>
<td>A standard recorder component</td>
</tr>
<tr>
<td>SysAttrCompType</td>
<td>StdCoder</td>
<td>A standard coder component</td>
</tr>
</tbody>
</table>
5. Data Types, Structures, and Error Codes

<table>
<thead>
<tr>
<th>Attribute Key</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SysAttrCompType</td>
<td>SysComponent</td>
<td>A standard DM3 system service</td>
</tr>
<tr>
<td>SysAttrCompId</td>
<td>MercConfigMgr</td>
<td>The configuration manager</td>
</tr>
<tr>
<td>SysAttrCompId</td>
<td>MercHostDriver</td>
<td>The host interface driver</td>
</tr>
<tr>
<td>SysAttrCompId</td>
<td>MercIPCDriver</td>
<td>The CP-SP interface driver</td>
</tr>
<tr>
<td>SysAttrCompId</td>
<td>MercResourceMgr</td>
<td>The resource manager</td>
</tr>
<tr>
<td>SysAttrCompId</td>
<td>MercSlotMgr</td>
<td>The timeslot manager</td>
</tr>
<tr>
<td>SysAttrCompId</td>
<td>MercStreamMgr</td>
<td>The global memory stream manager</td>
</tr>
<tr>
<td>Any Key</td>
<td>QATTR_ANY</td>
<td>Matches any value</td>
</tr>
</tbody>
</table>

5.2.6. QCompDesc Component Descriptor Structure

The data type QCompDesc is a structure which is the local representation of a DM3 component descriptor. This data structure is defined in `qcomplib.h`. A component descriptor has the following format:

```c
typedef struct{
    USHORT node;
    UCHAR board;
    UCHAR pad1;
    UCHAR processor;
    UCHAR component;
    UCHAR instance;
    UCHAR pad2;
}QCompDesc, *PQCompDesc;
```

- **node**: Currently unused
- **board**: Identifies a specific board within the system. The following standard identifiers are currently defined:
  - QCOMP_B_SELF
  - QCOMP_B_HOST
  - QCOMP_BNIL

---

175
processor  Identifies the processor where an instance resides. The following standard identifiers are currently defined:
    QCOMP_P_HOST
    QCOMP_P_CP
    QCOMP_P_SP
    QCOMP_P_SELF
    QCOMP_P_NIL

component  Identifies the type of component being addressed. The following standard identifiers are currently defined:
    QCOMP_C_SYS_SERVICE
    QCOMP_C_TASK
    QCOMP_C_STREAM
    QCOMP_C_INVALID
    QCOMP_C_NIL

instance  Identifies the type of instance being addressed. The following standard identifiers are currently defined:
    QCOMP_I_COMPONENT
    QCOMP_I_HMSGDRIVER
    QCOMP_I_HSTREAMDRV
    QCOMP_I_IPCDRIVER
    QCOMP_I_CONFIGMGR
    QCOMP_I_RESOURCENGR
    QCOMP_I_SMP
    QCOMP_I_BSTREAM_TSK
    QCOMP_I_CLUSTERMGR
    QCOMP_I_SRAM
    QCOMP_I_IDLE_TSK
    QCOMP_I_FTIMER
    QCOMP_I_QAGENT
    QCOMP_I_NIL

To partially specify a component instance, the instance field must be set to QCOMP_I_NIL. The processor and component fields may also optionally be set to their null values (QCOMP_P_NIL and QCOMP_C_NIL) as wild card values.
5. Data Types, Structures, and Error Codes

5.2.7. QValueAttr Board Attribute Specification Structure

The QValueAttr data structure is used by the `mntGetBoardsByAttr()` function and is defined in `qmsg.h`.

```c
typedef struct {
    char     ValueName[MNT_MAX_VALUE_NAME_SIZE];
    ULONG    ValueType;
    BYTE     ValueFlag;
    char     Value[MNT_MAX_VALUE_SIZE];
} QValueAttr, *PQValueAttr;
```

**ValueName** a NULL terminated string specifying the name of the value to find or the wild card "*" which can be used to indicate a match on any value name.

**ValueType** one of the Win32 registry types: REG_DWORD, REG_SZ, or REG_MULTISZ.

**ValueFlag** may be NULL to indicate a match on the value specified in Value or MNT_MATCH_ANY.VALUE to match on any value.

**Value** the value to match.

5.3. Error Code Definitions

If any Direct Interface host library function returns FALSE, you should call the `GetLastError()` function to retrieve the error. This is a Win32 API convention that the Direct Interface host library observes. There are two error-code classes: Dialogic and Windows NT. To determine if it’s a Direct Interface host library error, use the ERROR_MNT_BASE as a mask.

5.3.1. Windows NT Error Codes

Window NT provides error codes that can occur during general Win32 API function calls and during stream I/O operations. *Table 12* lists some of the possible Windows NT general error codes. Refer to `winerror.h` for details.
### Table 12. Windows NT General Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ERROR_FILE_NOT_FOUND</td>
<td>System cannot find specified file.</td>
</tr>
<tr>
<td>6</td>
<td>ERROR_INVALID_HANDLE</td>
<td>Handle is incorrect.</td>
</tr>
<tr>
<td>8</td>
<td>ERROR_NOT_ENOUGH_MEMORY</td>
<td>Not enough storage available to process this command.</td>
</tr>
<tr>
<td>31</td>
<td>ERROR_GEN_FAILURE</td>
<td>Device attached to the system is not functioning.</td>
</tr>
<tr>
<td>87</td>
<td>ERROR_INVALID_PARAMETER</td>
<td>Parameter is incorrect.</td>
</tr>
<tr>
<td>122</td>
<td>ERROR_INSUFFICIENT_BUFFER</td>
<td>Data area passed to a system call is too small.</td>
</tr>
<tr>
<td>997</td>
<td>ERROR_IO_PENDING</td>
<td>Overlapped I/O operation is in progress</td>
</tr>
<tr>
<td>998</td>
<td>ERROR_NOACCESS</td>
<td>Invalid access to memory location.</td>
</tr>
<tr>
<td>1011</td>
<td>ERROR_CANTOPEN</td>
<td>Configuration registry could not be opened.</td>
</tr>
<tr>
<td>1012</td>
<td>ERROR_CANTREAD</td>
<td>Configuration registry key could not be read.</td>
</tr>
</tbody>
</table>

If a stream I/O operation fails, the Class Driver (DLGCMD) and Protocol Driver (DLGCMPD) can return a Windows NT stream error code. *Table 13* lists some of the possible Windows NT stream error codes.
### Table 13. Windows NT Stream Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Name</th>
<th>Stream Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR or ERROR_SUCCESS</td>
<td>Read</td>
<td>Three possible cases:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Stream header matches user-specified completion mask, and request completes with current transfer count.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Sender has closed stream. All pending reads completed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Requested bytes have been read.</td>
</tr>
<tr>
<td>1</td>
<td>ERROR_INVALID_FUNCTION</td>
<td>Read or Write</td>
<td>Handle passed does not belong to the Stream device, or the requested action is inconsistent, such as a write request for a read stream.</td>
</tr>
<tr>
<td>21</td>
<td>ERROR_NOT_READY</td>
<td>Read or Write</td>
<td>The board is not in a ready state.</td>
</tr>
<tr>
<td>22</td>
<td>ERROR_BAD_COMMAND</td>
<td>Read or Write</td>
<td>There is no open or attached stream on the device handle passed.</td>
</tr>
<tr>
<td>57</td>
<td>ERROR_ADAP_HDW_ERR</td>
<td>Read or Write</td>
<td>There is a hardware error on the board.</td>
</tr>
<tr>
<td>71</td>
<td>ERROR_REQ_NOT_ACCEPT</td>
<td>Read or Write</td>
<td>The board has rejected the close- or open-stream request made by the host driver.</td>
</tr>
<tr>
<td>109</td>
<td>ERROR_BROKEN_PIPE</td>
<td>Write</td>
<td>The reader has closed the stream. Your application should properly close the stream.</td>
</tr>
</tbody>
</table>
DM3 Direct Interface Function Reference for Windows NT

### Error Code Reference

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Name</th>
<th>Stream Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>121</td>
<td>ERROR_SEM_TIMEOUT</td>
<td>Read or Write</td>
<td>The I/O request has timed out. If the timeout value is set through through the [\text{mntSetStreamIOTimeout}() function, a 30-second default is used.</td>
</tr>
<tr>
<td>170</td>
<td>ERROR_BUSY</td>
<td>Read or Write</td>
<td>The stream cannot be closed due to its non-zero reference count.</td>
</tr>
<tr>
<td>231</td>
<td>ERROR_PIPE_BUSY</td>
<td>Read or Write</td>
<td>The stream cannot be closed due to outstanding I/O requests.</td>
</tr>
<tr>
<td>997</td>
<td>ERROR_IO_PENDING</td>
<td>Read or Write</td>
<td>The I/O request has been accepted and is pending. Normal in asynchronous I/O.</td>
</tr>
<tr>
<td>1117</td>
<td>ERROR_IO_DEVICE</td>
<td>Read</td>
<td>Stream’s orphan buffer has overrun. The application is not reading quickly enough. Can be due to heavy system load.</td>
</tr>
</tbody>
</table>

#### 5.3.2. Dialogic Library and Driver Error Codes

If error checking in either the host library or driver layer detects a problem, error checking returns a Dialogic error code. Table 14 lists some of the possible Dialogic error codes. Please refer to dllmnti.h for details.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE0000000</td>
<td>ERROR_MNT_MMB_ALLOC_FAILED</td>
<td>Unable to allocate MMB</td>
</tr>
<tr>
<td>0xE0000001</td>
<td>ERROR_MNT_INVALID_VALUE_TYPE</td>
<td>Invalid Registry value type encountered</td>
</tr>
</tbody>
</table>
### Data Types, Structures, and Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE0000002</td>
<td>ERROR_MNT_NO_MCD_VERSION_ID</td>
<td>Unable to retrieve DLGCMCD version ID</td>
</tr>
<tr>
<td>0xE0000003</td>
<td>ERROR_MNT_NO_TRACE_HANDLE</td>
<td>Unable to open trace handle</td>
</tr>
<tr>
<td>0xE0000004</td>
<td>ERROR_MNT_CANTCLOSE</td>
<td>Unable to close registry key</td>
</tr>
<tr>
<td>0xE0000005</td>
<td>ERROR_MNT_INVALID_ATTR_KEY</td>
<td>Invalid attribute key</td>
</tr>
<tr>
<td>0xE0000006</td>
<td>ERROR_MNT_NO_BOARDS_BY_ATTR</td>
<td>Unable to get boards by attributes</td>
</tr>
<tr>
<td>0xE0000007</td>
<td>ERROR_MNT_NO_MEM</td>
<td>Unable to allocate memory for thread-local-storage MMB.</td>
</tr>
<tr>
<td>0xE0000008</td>
<td>ERROR_MNT_SYSTEM_ERR</td>
<td>Direct Interface system error</td>
</tr>
<tr>
<td>0xE0000009</td>
<td>ERROR_MNT_MERCURY_STD_MSG</td>
<td>Standard error message received</td>
</tr>
<tr>
<td>0xE000000A</td>
<td>ERROR_MNT_MERCURY_KRNL</td>
<td>DM3 Kernel error message received</td>
</tr>
<tr>
<td>0xE000000B</td>
<td>ERROR_MNT_HEAP_FREE_FAILED</td>
<td>Not used</td>
</tr>
<tr>
<td>0xE000000C</td>
<td>ERROR_MNT_HEAP_ALLOC_FAILED</td>
<td>Not used</td>
</tr>
<tr>
<td>0xE000000D</td>
<td>ERROR_MNT_INVALID_CMDSIZE</td>
<td>Invalid command size specified</td>
</tr>
</tbody>
</table>
Index

A
Architecture, DM3
definition, 2
asynchronous function calls, 8
asynchronous function returns, 9
asynchronous functions
OVERLAPPED structure, 8

C
calling functions
asynchronously, 8
synchronously, 11
Class Driver, 5
cluster management functions, 16
mntClusterActivate( ), 30
mntClusterAllocate( ), 34
mntClusterByComp( ), 37
mntClusterComByAttr( ), 39
mntClusterConfigLock( ), 42
mntClusterConfigUnlock( ), 44
mntClusterConnect( ), 46
mntClusterCreate( ), 50
mntClusterDeactivate( ), 53
mntClusterDestroy( ), 56
mntClusterDisconnect( ), 58
mntClusterFind( ), 61
mntClusterFree( ), 63
mntClusterSlotInfo( ), 65
mntClusterTSAssign( ), 68
mntClusterTSUnassign( ), 71

CompactPCI, 5
Component
definition, 2
compartment management functions, 17

mntCompAllocate( ), 74
mntCompFind( ), 77
mntCompFree( ), 86
mntCompUnuse( ), 88
mntCompUse( ), 90

CreateFile( ), 5

D
data structures
Direct Interface, 170
MSB, 170
QBoardAttr, 173
QCompAttr, 173
QCompDesc, 175
QValueAttr, 177
STRM_HDR, 171
STRM_INFO, 172
data types
Direct Interface, 169
debug support functions, 17
mntGetDrvVersion( ), 105
mntGetLibVersion( ), 106
mntSetTraceLevel( ), 139
mntTrace( ), 143
mntTransGen( ), 145

Dialogic Class Driver, 5
Dialogic error codes, 180
Dialogic Protocol Driver, 5

Direct Interface
cluster management functions, 16
component management functions, 17
data structures, 170
data types, 169
debug support functions, 17
trace codes, 177
DM3 Direct Interface Function Reference for Windows NT

exit notification functions, 20
message I/O functions, 18
stream I/O functions, 18
DLGCMD, 5
DLGCMPD, 5
DM3 architecture
key concepts, 2
DM3 Direct Interface host library, 4
DM3 Direct Interface Overview, 3
DM3 embedded system, 5
DM3 firmware, 4
DM3 Hardware, 5
DM3 host library, 4
DM3 message header macros, 164
QMSG_GET_DESTADDR, 164
QMSG_GET_MSGSIZE, 164
QMSG_GET_MSGTYPE, 165
QMSG_GET_SRCADDR, 165
QMSG_GET_TRANS, 165
QMSG_SET_DESTADDR, 165
QMSG_SET_MSGSIZE, 165
QMSG_SET_MSGTYPE, 166
QMSG_SET_SRCADDR, 166
QMSG_SET_TRANS, 166
DM3 message pointer macros, 163
MNT_MNT_GET_CMD_QMSG, 163
MNT_MNT_GET_REPLY_QMSG, 163
DM3 messages with payloads, 166
DMA, 5
E
error codes
definitions, 177
Dialogic, 180
Windows NT, 177
error handling
asynchronous, 9
asynchronous code example, 11
synchronous, 12
synchronous code example, 14
exit notification functions, 20
F
functions
asynchronous, 8
cluster management
mntClusterActivate(), 30
mntClusterAllocate(), 34
mntClusterByComp(), 37
mntClusterByAttr(), 39
mntClusterConfigLock(), 42
mntClusterConfigUnlock(), 44
mntClusterConnect(), 46
mntClusterCreate(), 50
mntClusterDeactivate(), 53
mntClusterDestroy(), 56
mntClusterDisconnect(), 58
mntClusterFind(), 61
mntClusterFree(), 63
mntClusterSlotInfo(), 65
mntClusterTSAssign(), 68
mntClusterTSUnassign(), 71
component management
mntCompAllocate(), 74
mntCompFind(), 77
mntCompFree(), 86
mntCompUnuse(), 88
mntCompUse(), 90
debug support
mntGetDrvVersion(), 105
mntGetLibVersion(), 106
mntSetTraceLevel(), 139
mntTrace(), 143
mntTransGen(), 145
message I/O
mntAllocateMMB(), 22
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mntCheckStreamOrphans()</td>
<td>28</td>
</tr>
<tr>
<td>mntClearMMB()</td>
<td>29</td>
</tr>
<tr>
<td>mntCopyMMB()</td>
<td>93</td>
</tr>
<tr>
<td>mntEnumMpathDevice()</td>
<td>96</td>
</tr>
<tr>
<td>mntEnumStrmDevice()</td>
<td>98</td>
</tr>
<tr>
<td>mntFreeMMB()</td>
<td>100</td>
</tr>
<tr>
<td>mntGetBoardsByAttr()</td>
<td>101</td>
</tr>
<tr>
<td>mntGetMpathAddr()</td>
<td>109</td>
</tr>
<tr>
<td>mntGetTLSmmb()</td>
<td>116</td>
</tr>
<tr>
<td>mntRegisterAsyncMessages()</td>
<td>123</td>
</tr>
<tr>
<td>mntRegisterAsyncStreams()</td>
<td>126</td>
</tr>
<tr>
<td>mntSendMessage()</td>
<td>129</td>
</tr>
<tr>
<td>mntSendMessageWait()</td>
<td>131</td>
</tr>
<tr>
<td>qMsgVarFieldGet()</td>
<td>146</td>
</tr>
<tr>
<td>qMsgVarFieldPut()</td>
<td>150</td>
</tr>
<tr>
<td>mntAttachMercStream()</td>
<td>25</td>
</tr>
<tr>
<td>mntCompleteStreamIo()</td>
<td>92</td>
</tr>
<tr>
<td>mntDetachMercStream()</td>
<td>94</td>
</tr>
<tr>
<td>mntGetMercStreamID()</td>
<td>107</td>
</tr>
<tr>
<td>mntGetStreamHeader()</td>
<td>111</td>
</tr>
<tr>
<td>mntGetStreamInfo()</td>
<td>114</td>
</tr>
<tr>
<td>mntGetStreamInfo()</td>
<td>135</td>
</tr>
<tr>
<td>mntSetStreamIOTimeout()</td>
<td>138</td>
</tr>
</tbody>
</table>

**H**

Hardware, 5

**I**

Introduction to DM3 architecture definition, 2

**M**

macros
- DM3 message header, 164
- DM3 message pointer, 163
- DM3 messages with payloads, 166
- MMB control header, 156

Message definition, 3

message I/O functions, 18
- mntAllocateMMB(), 22
- mntCheckStreamOrphans(), 28
- mntClearMMB(), 29
- mntCopyMMB(), 93
- mntEnumMpathDevice(), 96
- mntEnumStrmDevice(), 98
- mntFreeMMB(), 100
- mntGetBoardsByAttr(), 101
- mntGetMpathAddr(), 109
- mntGetTLSmmb(), 116
- mntRegisterAsyncMessages(), 123
- mntRegisterAsyncStreams(), 126
- mntSendMessage(), 129
- mntSendMessageWait(), 131
- qMsgVarFieldGet(), 146
- qMsgVarFieldPut(), 150

**MBB, 155**

MMB control header macros, 156
- MNT_GET_MMB_ACTUAL_REPLY_COUNT, 157
- MNT_GET_MMB_CMD_SIZE, 157
- MNT_GET_MMB_CMD_TIMEOUT, 157
- MNT_GET_MMB_CURRENT_REPLY_OFFSET, 157
- MNT_GET_MMB_EMPTY_MSG, 158
- MNT_GET_MMB_EXPECTED_REPLY_COUNT, 158
- MNT_GET_MMB_MATCH_ON_DEST_ADDR, 158
- MNT_GET_MMB_MATCH_ON_MSGTYPE, 159
- MNT_GET_MMB_MATCH_ON_RC_ADDR, 159
- MNT_GET_MMB_MATCH_ON_TRANS_ID, 159
- MNT_GET_MMB_REPLY_MAX_SIZE, 160
DM3 Direct Interface Function Reference for Windows NT

MNT_SET_MMB_CMD_SIZE, 160
MNT_SET_MMB_CMD_TIMEOUT, 160
MNT_SET_MMB_EMPTY_MSG, 160
MNT_SET_MMB_EXPECTED_REPLY_COUNT, 161
MNT_SET_MMB_MATCH_ON_DEST_ADDR, 161
MNT_SET_MMB_MATCH_ON_MSGTYPE, 161
MNT_SET_MMB_MATCH_ON_SRC_ADDR, 162
MNT_SET_MMB_MATCH_ON_TRANS_ID, 162
MNT_SET_MMB_REPLY_MAX_SIZE, 162
MNT_GET_MMB_MATCH_ON_TRANS_ID, 159
MNT_GET_MMB_REPLY_MAX_SIZE, 160
MNT_GET_MMB_REPLY_QMSG, 163
MNT_SET_MMB_CMD_SIZE, 160
MNT_SET_MMB_CMD_TIMEOUT, 160
MNT_SET_MMB_EMPTY_MSG, 160
MNT_SET_MMB_EXPECTED_REPLY_COUNT, 161
MNT_SET_MMB_MATCH_ON_DEST_ADDR, 161
MNT_SET_MMB_MATCH_ON_MSGTYPE, 161
MNT_SET_MMB_MATCH_ON_SRC_ADDR, 162
MNT_SET_MMB_MATCH_ON_TRANS_ID, 162
MNT_SET_MMB_REPLY_MAX_SIZE, 162
mntAllocateMMB(), 22
mntAttachMercStream(), 25
mntCheckStreamOrphans(), 28
mntClearMMB(), 29
mntClusterActivate(), 30
mntClusterAllocate(), 34
mntClusterByComp(), 37
mntClusterCompByAttr(), 39
mntClusterConfigLock(), 42
mntClusterConfigUnlock(), 44

MMB structure, 156
MNT_GET_CMD_QMSG, 163
MNT_GET_MMB_ACTUAL_REPLY_COUNT, 157
MNT_GET_MMB_CMD_SIZE, 157
MNT_GET_MMB_CMD_TIMEOUT, 157
MNT_GET_MMB_CURRENT_REPLY_OFFSET, 157
MNT_GET_MMB_EMPTY_MSG, 158
MNT_GET_MMB_EXPECTED_REPLY_COUNT, 158
MNT_GET_MMB_MATCH_ON_DEST_ADDR, 158
MNT_GET_MMB_MATCH_ON_MSG_TYPE, 159
MNT_GET_MMB_MATCH_ON_SRC_ADDR, 159

186
Index

mntClusterConnect(), 46
mntClusterCreate(), 50
mntClusterDeactivate(), 53
mntClusterDestroy(), 56
mntClusterDisconnect(), 58
mntClusterFind(), 61
mntClusterFree(), 63
mntClusterSlotInfo(), 65
mntClusterTSAssign(), 68
mntClusterTSUnassign(), 71
mntCompAllocate(), 74
mntCompFind(), 77
mntCompFree(), 86
mntCompleteStreamIo(), 92
mntCompUnuse(), 88
mntCompUse(), 90
mntCopyMMB(), 93
mntDetachMercStream(), 94
mntEnumMpathDevice(), 96
mntEnumStrmDevice(), 98
mntFreeMMB(), 100
mntGetBoardsByAttr(), 101
mntGetDrvVersion(), 105
mntGetLibVersion(), 106
mntGetMercStreamID(), 107
mntGetMpathAddr(), 109
mntGetStreamHeader(), 111
mntGetStreamInfo(), 114
mntGetTLSmb(), 116
mntRegisterAsyncMessages(), 123
mntRegisterAsyncStreams(), 126
mntSendMessage(), 129
mntSendMessageWait(), 131
mntSetStreamHeader(), 135
mntSetStreamIOTimeout(), 138
mntSetTraceLevel(), 139
mntTerminateStream(), 141
mntTrace(), 143
mntTransGen(), 145
MSB, 170
multiple message block, 155

P
PCI, 5
PIO, 5
Protocol Driver, 5

Q
QBoardAttr, 173
QClusterResult message, 35, 38, 51, 62
QClusterSlotInfoResult message, 67
QClusterUnlockCmplt message, 45
QCompAttr, 173
QCompDesc, 175
QCompMultipleResult message, 84
QComponentResult message, 40, 76, 80
QFailureNotify message, 119
QMSG_GET_DESTADDR, 164
DM3 Direct Interface Function Reference for Windows NT

QMSG_GET_MSGSIZE, 164
QMSG_GET_MSGTYPE, 164
QMSG_GET_SRCADDR, 165
QMSG_GET_TRANS, 165
QMSG_SETDESTADDR, 165
QMSG_SET_MSGSIZE, 165
QMSG_SET_MSGTYPE, 166
QMSG_SET_SRCADDR, 166
QMSG_SET_TRANS, 166
qMsgVarFieldGet(), 146
qMsgVarFieldPut(), 150
QResultComplete message, 33, 43, 49,
54, 57, 60, 64, 70, 72, 87, 89,
91, 119, 122
QValueAttr, 177

R
ReadFile(), 5

Resource
definition, 2

Result messages
QClusterResult, 35, 38, 51, 62
QClusterSlotInfoResult, 67
QClusterUnlockCmplt, 45
QCompMultipleResult, 84
QComponentResult, 40, 76, 80
QFailureNotify, 119
QResultComplete, 33, 43, 49, 54,
57, 60, 64, 70, 72, 87, 89,
91, 119, 122

S
stream I/O functions, 18
mntAttachMercStream(), 25
mntCompleteStreamIo(), 92

mntDetachMercStream(), 94
mntGetMercStreamID(), 107
mntGetStreamHeader(), 111
mntGetStreamInfo(), 114
mntSetStreamHeader(), 135
mntSetStreamIOTimeout(), 138
mntTerminateStream(), 141

STRM_HDR, 171
STRM_INFO, 172
synchronous function calls, 11
synchronous function returns, 12

V
VME, 5

W
Windows NT
error codes, 177
WriteFile(), 5