Using the DM3 Direct Interface

for Windows NT

05-0987-001

COPYRIGHT NOTICE

All contents of this document are subject to change without notice and do not represent a commitment on the part of Dialogic Corporation. Every effort is made to ensure the accuracy of this information. However, due to ongoing product improvements and revisions, Dialogic Corporation cannot guarantee the accuracy of this material, nor can it accept responsibility for errors or omissions. No warranties of any nature are extended by the information contained in these copyrighted materials. Use or implementation of any one of the concepts, applications, or ideas described this document or on Web pages maintained by Dialogic-may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic does not condone or encourage such infringement. Dialogic makes no warranty with respect to such infringement, nor does Dialogic waive any of its own intellectual property rights which may cover systems implementing one or more of the ideas contained herein. Procurement of appropriate intellectual property rights and licenses is solely the responsibility of the system implementer. The software referred to in this document is provided under a Software License Agreement. Refer to the Software License Agreement for complete details governing the use of the software.

Publication Date: April, 1998

Part Number: 05-0987-001

Dialogic Corporation
1515 Route 10
Parsippany NJ 07054

Technical Support
Phone: 973-993-1443
Fax: 973-993-8387
BBS: 973-993-0864
Email: CustEng@dialogic.com

For **Sales Offices** and other contact information, visit our website at **http://www.dialogic.com**

Table of Contents

List of Tables

List of Figures

1. Introduction

This guide presents the methods you can use for developing applications based on DM3 products. Use this guide in conjunction with the *DM3 Direct Interface Function Reference for Windows NT* and other guides.

The Direct Interface provides host developers with the lowest level of control over the DM3 embedded system and offers a great degree of flexibility. Use the Direct Interface either to build your applications or to build a functional software layer to emulate other application programming interfaces (APIs).

To help you implement common features and routine functions, Dialogic offers Application Foundation Code with many DM3 products. See the document entitled *DM3 Application Foundation Code for Windows NT*.

As shown in *Figure 1. The DM3 Direct Interface in a System*, the Direct Interface (DI) is the interface between an application and the embedded system. The DI is a host library which offers access to the device drivers and is the only way to get driver-level access. For details, see *Chapter 2. Understanding the Direct Interface*.

Note that while the Direct Interface can be used across all DM3 products under Windows NT, the functionality of a DM3 product is provided by firmware-based resources (which are downloaded to the DM3 hardware). Different DM3 products may have unique capabilities including some capabilities added by third-party developers.

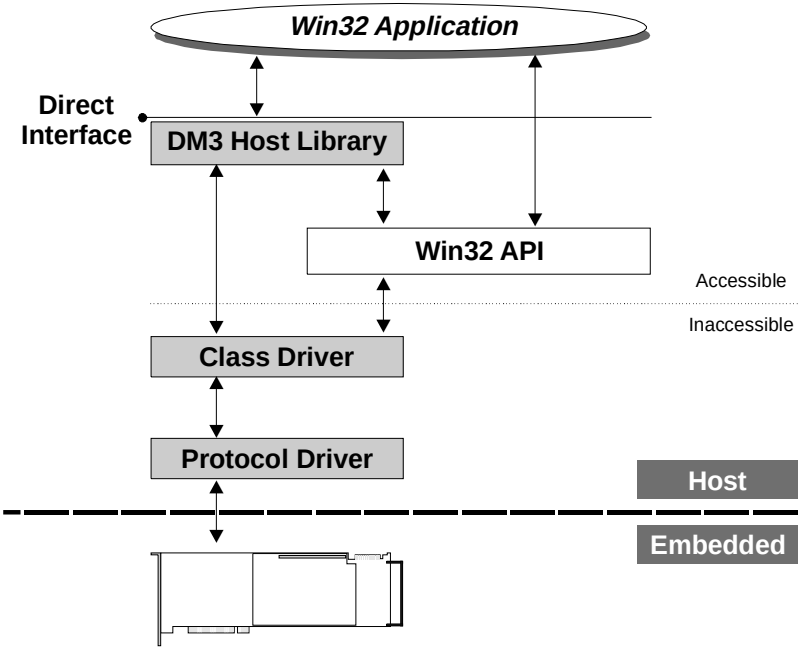

Figure 1. The DM3 Direct Interface in a System

1.1. Information in This Guide

This guide is arranged into the following chapters:

- **Chapter 1: Introduction** provides a brief introduction to the Direct Interface and offers a broad context for understanding its function with the DM3 Architecture.
- **Chapter 2: Understanding the Direct Interface** explains the concepts you need to understand before you begin programming.
- **Chapter 3: Windows NT Programming Models** describes the pros and cons of different programming models.
- **Chapter 4: Calling Direct Interface Functions** briefly discusses some issues you need to consider when calling functions using the Direct Interface Host Library.

- **Chapter 5: DM3 Devices** provides details about the device types used to support messaging and bulk data transfers.
- **Chapter 6: Using Messages** shows how to build, access, format, send, receive, and cancel messaging operations.
- **Chapter 7: Using Data Streams** shows how to write, read, and cancel bulk data transfers.
- **Chapter 8: Using Clusters** shows how a host application can control clusters on the DM3 embedded system.
- **Chapter 9: Exit Notification** shows how to enable your application to respond to system failures
- **Chapter 10: Error Handling** discusses how to retrieve errors from the host and from the embedded system.
- **Chapter 11: Direct Interface Programming Guidelines** provides several helpful tips to use when using Direct Interface function calls.
- **Chapter 12: Compiling and Linking an Application** provides some necessary tips to use when compiling and linking.
- **Chapter 13: Debugging** lists some facts about debugging.
- **Chapter 14: Tools and Utilities** contains instructions on how to use the utilities supplied with a DM3 product.
- **Index**

1.2. How to use This Guide

This guide shows how to use the DM3 Direct Interface for Windows NT to build programs. The low-level and flexible nature of the Direct Interface allows you build either of the following programs on your host processor:

- applications based on DM3 embedded firmware resources
- application programming interfaces

1.2.1. Typeface Conventions

The following conventions are used throughout DM3 software:

- **Function Names** begin with a lowercase “mnt” followed by one or more words describing the function. Each word within the function name begins with a capital letter, there are no separator characters, and the name ends with a set of parentheses; for example, **mntCompFind( )**. Function names are always presented in boldface type.
- **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)** begin with an uppercase “Q” followed by one or more words describing the data type. Each word within the data type name begins with a capital letter, and there are no separator characters; for example, QStatus. This convention may sometimes be overruled by the conventions of the operating system.
- **Constant Definitions (#define)** are shown in non-bold uppercase type, such as MNTI_STATE_PRE_INIT and MNTI_STATE_INITIALIZED. Underscore separators between words aid readability. Related constant definitions share the same first word.

- **Parameter Names** begin with a lowercase letter; words within the name begin with a capital letter. Pointer parameters begin with the letters “lp”. Examples include **mode, theInstance,** and **lpCount.** The function parameters are ordered with inputs appearing before outputs. Parameters are always shown in boldface type.
- **Message Names** begin with the name of the component (or sometimes an abbreviation of the name) to which it pertains, an underscore, and then the letters *Msg.* Some examples are *Player_MsgStop* and *Recorder_MsgStop.* There are also standard messages (which many components support) that begin with *Std_Msg* (for example, *Std_MsgError)* . Message names are presented in italic type.
- **Field Names** used in data structures are presented in boldface type.
- **File Names** are lowercase and shown in italic type. Example: *coders.h*
- **Code Examples** and **Command Line Input** are shown in a small constant-width font. For example:

```
typedef struct {
    UInt8         userType;
    QTStreamType  stream;
    UInt8         instance;
} QPortId;
```

- **Variables** within a command line are shown in italics.
  For example: `edit` *`myfile`*

NOTE: Typographic conventions are not used within a code example.

1.2.2. Other Relevant Guides and References

Use the information in this guide in conjunction with these other sources of information:

- *DM3 Direct Interface Function Reference for Windows NT*
- *DM3 Mediastream Architecture Overview Guide*
- *DM3 Standard Component Interface Message Reference*

1.3. Overview: DM3 Family of Products

Dialogic's DM3 product family now includes a full range of voice, fax, speech, network interface and internet telephony technologies. DM3 is the industry's broadest and most scaleable product line, enabling developers to create more powerful computer telephony applications.

The entire Dialogic DM3-based family of products is available in the following hardware form factors:

- PCI (Peripheral Component Interconnect)
- CompactPCI (compact Peripheral Component Interconnect)
- VME (Versa Module Europa)

These new technologies build upon the industry's most scalable CT component product line, enabling developers to create powerful DM3-based solutions:

- Voice Processing - Dialogic PCI products scale up to 120 channels of both voice processing and network interface per card with the DM3-based QuadSpan Series.
- Fax Processing - Dialogic's PCI fax line is scalable up to 24 or 30 channels of fax with the DM3 Fax series, the highest density fax resource on the market today.
- Network Interface - Dialogic network interfaces include the QuadSpan DTI Series for PCI and CompactPCI, designed to provide a powerful set of

advanced call processing features that developers can use to create cost-efficient, high channel density switching systems.

- Internet Telephony - The Dialogic DM3 IPLink family of PCI and CompactPCI Internet telephony platforms is fully compatible with leading H.323 client applications such as Microsoft NetMeeting, Intel Internet Video Phone and VocalTec Internet Phone. DM3 IPLink-based servers enable individuals to communicate directly over the data network—from phone to phone, fax to fax, PC to phone, phone to PC and Web browser to phone.

1.4. 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 Guide*.

- **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.

2. Understanding the Direct Interface

Concepts you need to understand before you begin programming are discussed in this section. Reading through this section will help explain some of the aspects of the Direct Interface for Windows NT. See the *DM3 Mediastream Architecture Overview Guide* for a more complete discussion.

The architecture of any DM3-based system consists of the following items:

- host software
- firmware modules
- hardware

2.1. Host Software

The DM3 Direct Interface is a low-level message-based 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 term “Direct Interface” is applied to the library that offers the lowest-level access to the DM3 embedded system, regardless of the way it is implemented under a certain operating system. For Windows NT, the DM3 Host Library accesses DM3 Device Drivers (a Class Driver and a Protocol Driver). Applications communicate only with the host library; the device drivers are not accessed directly.

Figure 2 illustrates the host and embedded portions of a generic DM3-based system.

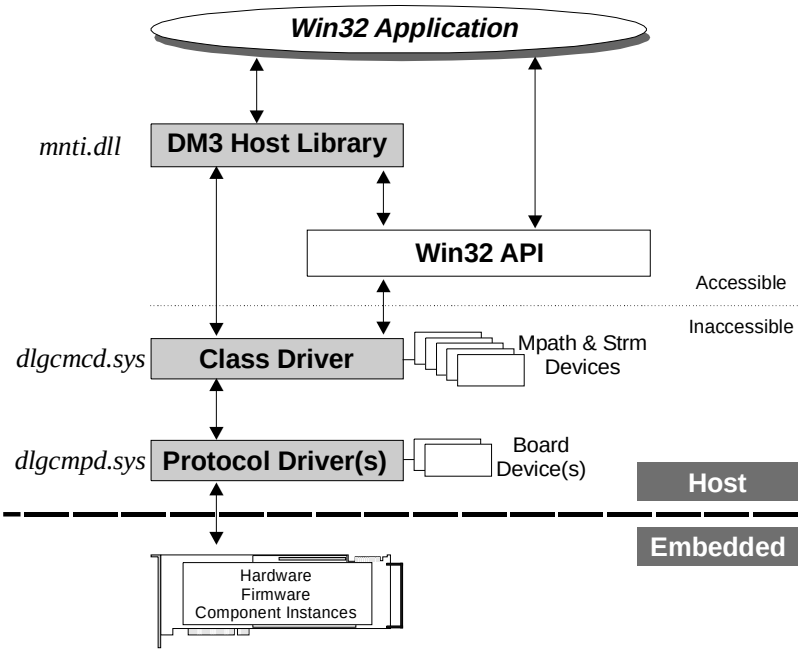

Figure 2. DM3 Direct Interface Components

2.1.1. 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 internal shared data structures from being overwritten when they are used by multiple threads.

An application built with the Direct Interface for Windows NT 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. See the section entitled *Multiple Message Blocks (MMBs)* for more information.

2.1.2. 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 **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).

2.2. 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 (PCI, CompactPCI, or VME), up to three signal-processing daughterboards, and other hardware components. For details on the DM3 hardware architecture, see the *DM3 Mediastream Architecture Overview Guide*.

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.

2.3. 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 the DM3 software architecture, see the *DM3 Mediastream Architecture Overview Guide*.

2.4. Understanding Data Communication

The DM3 architecture uses messages and bulk data streams as its two major communication mechanisms. Messages primarily pass commands, results, and other events between the host application and the embedded system. Data streams primarily pass large amounts of data, such as audio or fax data, between the host and the embedded system.

For more information on the DM3 devices used for input and output, see *Chapter 5. DM3 Devices*.

2.4.1. Understanding Messaging

Messages are passed between the host and the embedded system via a structure called the Multiple Message Block (MMB). Host applications should not access the MMB structure directly. Instead, use the macros provided with the Direct Interface that resolve the endian-type issues.

Macros exist to handle these five types of messages:

- **Simple Messages**
  Messages that contain no payload data, only status information, such as an operation completion message.
- **Fixed-size Messages**
  Messages that contain a pre-defined payload of a known size.
- **Variable Payload Messages**
  Messages containing an array payload information of a size that varies based on the commands that were sent.
- **Variable List Messages**
  These contain a list of different types of arrays of various payloads.
- **KVSet Messages**
  Messages with Key/Value sets containing attribute information for board(s), component(s), etc.

Multiple Message Blocks (MMBs)

An application built using the Direct Interface host library 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.

A Multiple Message Block (MMB) must be allocated for passing messages. The memory block is made up of the following sections:

- MMB Header
- Command Message Fixed Header
- Command Message Payload

- Reply Message Fixed Header (optional)
- Reply Message Payload (optional)

The MMB contains space for one MMB Header and Command Message section and for any number of reply messages, each with its own header and payload, as shown in *Figure 3. MMB Structure*.

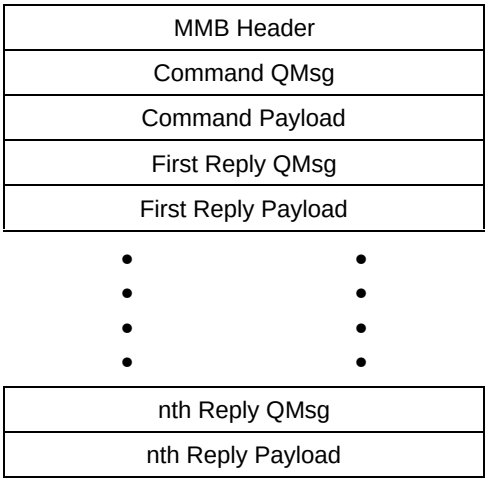

Figure 3. MMB Structure

The header and payload information in an MMB is in a 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 and not accessed directly.

2.4.2. Understanding Data Streams

Streams are the method by which large amounts of data are sent between the host and a component on any processor and between a component on a processor and the TDM Bus. Streams are based on a point-to-point unidirectional pipe-like model.

Conceptually, a particular stream is opened at each end and has a single reader and a single writer. The basic I/O method provides an unstructured word stream regardless of the underlying physical stream implementation. Data buffering is done invisibly.

2.4.3. Understanding Eventing and Run-Time Control

The Direct Interface uses Win32 API function calls to process messages coming from the DM3 embedded system. Sometimes, a change in state in the embedded system may cause an "unsolicited" message to be sent from the embedded system to the host application.

The following types of messages may be sent to the application:

- reply messages (in response to a command message sent from the application)
- unsolicited messages (the host must be set up to receive these types of messages)

The recommended method for recognizing and processing reply and unsolicited messages is through **I/O Completion Ports** (IOCP). See the Win32 SDK documentation for more information.

For each message expected from the DM3 embedded system, you must set up an MMB data structure. When a message is received, the MMB will be populated with relevant message parameters. If you're using an asynchronous multithreaded model, you should assign a specific I/O completion key and an **Overlapped** pointer to the specific message type.

If an application might receive asynchronous unsolicited messages from the embedded system, it must set up an empty MMB data structure. This is the only

way to guarantee that unsolicited messages will be received. For more information, see *Section 6. Using Messages*.

3. Windows NT Programming Models

Choosing a programming model may be the most important decision you make about the design of your application. Deciding your approach now can increase your program's efficiency or decrease the amount of time you might spend developing an application.

It is important to understand the following terms as they apply to application design:

- **Single-threaded**
  Your application contains only one thread which controls one or more devices. The following pseudo-code shows a typical form for a single-threaded program:

```
void main()       //  The main routine is the single thread
{
     for (i=0;i<NumThreads;i++)
     {
              myThread( );
     }
}
```

- **Multithreaded**
  Your application contains more than one thread, each of which can control one or more devices. The following pseudo-code shows a typical form for a multithreaded program:

```
   void main( )
   {
        for (i=0;i<NumThreads;i++)
        {
              CreateThread(..., myThread,--- )
        }

       //Wait for all threads to Stop
  }
```

- **Synchronous**
  In synchronous programming, each function blocks thread execution until the function completes. This includes callback models supported by the Win32 API.

- **Asynchronous**
  In asynchronous programming, the calling thread or process performs further operations while a called function completes. When the function completes, the application receives an event notification.

Based on the goals and complexity of your program, you may decide to follow one of these models:

- Asynchronous Multithreaded or Asynchronous Single-threaded
- Synchronous Multithreaded or Synchronous Single-threaded

3.1. Choosing a Programming Model

The following chart shows the various decision points which will lead you to favor one model over the other:

Table 1. Choosing a Programming Model

IF…	THEN choose…
❑ Your program flow is complicated ❑ Actions between devices are closely coupled ❑ Your application must be efficient ❑ Your application supports a large number of devices ❑ Your program requires a state machine ❑ Your program must wait for multiple devices on a single thread	An asynchronous model
❑ You plan to integrate DM3 devices with other devices (such as a database)	Asynchronous Multithreaded
❑ Your application will not integrate DM3 devices with other devices	Asynchronous Single-threaded
❑ Your program flow is simple ❑ Actions between devices are loosely coupled ❑ Each thread controls only one device	Synchronous Multithreaded
❑ Your program services only one device at a time	Synchronous Single-threaded

The remainder of this section discusses the advantages, disadvantages, and some programming notes for each model.

3.2. Asynchronous Multithreaded Model

Due to the high number of devices that DM3 allows you to control, Dialogic recommends using the asynchronous multithreaded model. In asynchronous multithreaded application programming, you create multiple threads, each of which controls one or more devices. In such an application, each thread has its own specific state machine for the devices that it controls. For example, you can have one grouping of devices that provides fax services and another grouping that provides Interactive Voice Response (IVR) services, while both share the same processing space and database resources.

An asynchronous multithreaded program does not block execution while waiting for a function to complete; this would interfere with the processing requirements of other devices also being managed by the thread. An asynchronous model allows you to create an event-driven state machine for each device. Each function returns immediately and allows thread processing to continue. Subsequently, when an event is returned (signifying the completion of an operation), state machine processing can continue.

Using the asynchronous multithreaded model requires a familiarity with I/O Completion Ports (see *Section 4.1.1. OVERLAPPED Structure*). After issuing an asynchronous function, your application should use the **GetQueuedCompletionStatus( )** function to wait for events on Dialogic devices. You may use either of the available Win32 Synchronization objects to achieve the asynchronous behavior you require (such as **WaitForSingleObject( )** and **WaitForMultipleObjects( )**).

3.2.1. Asynchronous Multithreaded Model Trade-offs

Asynchronous programming offers the following advantages:

- Requires fewer system resources than the synchronous model because the you use only a few threads for a large number of devices.
- Provides better control of DM3 applications that have high channel density.
- Reduces system overhead by minimizing thread context switching.
- Simplifies the coordination of events from many devices.

- Allows you to run entire portions of the application with a single thread in an application controlling many devices (including non-Dialogic devices).

Asynchronous programming offers the following disadvantages:

- This model is typically the most complex to develop due to the thread synchronization and coordination required.
- The asynchronous multithreaded model requires the development of a state machine.

3.2.2. Asynchronous Multithreaded Programming Notes

- If you use I/O Completion Ports, use **GetQueuedCompletionStatus( )** to find the status of the operation.
- After the event is processed, your application must determine what asynchronous function should be issued next depending on what event has occurred and the last state of the device when the event occurred.
- Do not use any DM3 device in more than one grouping. Otherwise, it is impossible to determine which thread receives the event.

3.3. Asynchronous Single-threaded Model

If you choose to avoid managing the complexities of multiple threads, then asynchronous single-threaded programming is recommended for applications that have large numbers of devices. However, as the total number of devices in the thread increases, your application may reach a point where the latency in servicing events and devices becomes intolerable. This may cause your application to perform poorly and responsiveness may suffer.

3.3.1. Asynchronous Single-threaded Model Trade-offs

Asynchronous single-threaded programming offers the following advantages:

- Requires a considerably less complex model than an asynchronous multithreaded model.
- Achieves a high level of resource management by combining multiple devices in a single thread.
- Simplifies the coordination of events from many devices.
- Requires fewer system resources than any synchronous model because any asynchronous model can use one thread for many devices.

Asynchronous single-threaded programming offers the following disadvantages:

- May require the development of a state machine.
- Asynchronous applications are typically more complex to develop than a synchronous application.

3.3.2. Asynchronous Single-threaded Programming Notes

- After an event is processed, your application must determine what asynchronous function should be issued next depending on what event has occurred and the last state of the device when the event occurred.

3.4. Synchronous Multithreaded Model

In a synchronous multithreaded model, the operating system can put individual device threads to sleep while allowing threads that control other devices to continue their actions without interruption. When a function completes, the operating system wakes up the function's thread so that processing continues. For example, if the application is playing a file as a result of a certain function call, the calling thread does not continue execution until the function call has completed and the function has terminated.

Typically, you can use this model to write code and create a thread for each device that needs to run this code. You do not need event-driven state machine processing because each function runs uninterrupted to completion.

Choose the synchronous multithreaded model when you are programming an application that has:

- Only a few devices.
- Simple and straight flow control with only one action per device occurring at any time.

3.4.1. Synchronous Multithreaded Model Trade-offs

Synchronous multithreaded programming offers the following advantages:

- The synchronous multithreaded model is the easiest to program and maintain, therefore it allows quicker application development than asynchronous models. The synchronous model is the least complex programming model that allows realistic usage of DM3 products.

Synchronous multithreaded programming offers the following disadvantages:

- Because the main thread creates a separate thread for each device, this model requires a high level of system resources. This can limit the maximum device density.
- Because a synchronous operation blocks thread execution, the thread cannot perform any other processing.
- Unsolicited events are not processed until the thread calls a specific function.

3.4.2. Synchronous Multithreaded Model Programming Notes

- You should use the synchronous multithreaded model only for simple and straight flow control with only one action per device occurring at any time.
- Because each function in the synchronous multithreaded model blocks execution in its thread, your application's main thread must create a separate thread for each device.

3.5. Synchronous Single-threaded Model

Using a synchronous single-threaded model is not recommended for production-level DM3 applications. Use this model only for proof-of-concept testing or quick programming exercises. With a synchronous single-threaded model, you can only service one device at a time. Due to the high-density nature of DM3, using a synchronous single-threaded programming model is not practical.

As an example, if an application is waiting for an inbound call on one channel and playing a file on another channel, the "WaitForCall" function could block indefinitely. However, the "PlayFile" call needs real-time servicing (for actions such as transferring data down to the hardware). Using this model would lead to intolerable latencies incurred during the play of the file and translate into a poor quality play.

3.6. Multi-process Applications

Developing a multi-process application, where the application essentially spawns a copy of itself, is not a recommended approach for Dialogic's DM3-based products under Windows NT. Forking a process is not recommended as there would be a substantial performance degradation.

4. Calling Direct Interface Functions

Some issues you may need to consider when using the Direct Interface in your application are discussed in this chapter. It is important to understand the difference between calling functions synchronously and asynchronously, what happens when you call certain types of functions.

NOTE: For a complete discussion of each function, data structure, and error code, see the *DM3 Direct Interface Function Reference for Windows NT*. Also, to help you implement common features and routine functions, Dialogic offers Application Foundation Code with many DM3 products. See the document entitled *DM3 Application Foundation Code for Windows NT*.

Some Direct Interface functions execute only on the host computer and others execute on the DM3 embedded system. By setting the **lpOverlapped** parameter, a function can be called either synchronously (when **lpOverlapped** is set to NULL) or asynchronously (when **lpOverlapped** is set to non-NULL) depending on whether you want your thread to block.

As shown in *Figure 4. Calling Functions From Your Application*, functions that execute on the host are synchronous, while those that execute on the board are asynchronous, even if you call them synchronously.

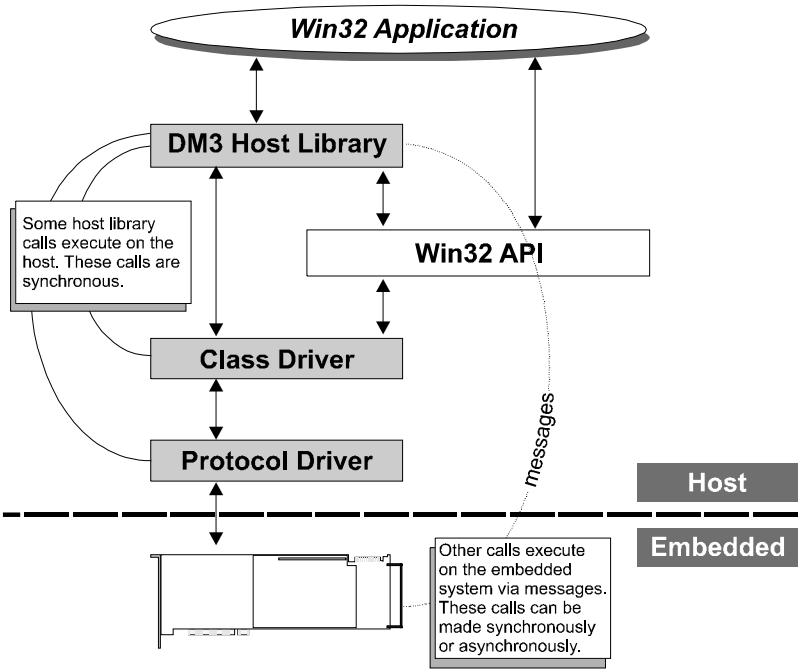

Figure 4. Calling Functions From Your Application

4.1. 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.

If you choose to set the **lpOverlapped** parameter to NULL, the call is considered synchronous and your thread will block until the function completes.

4.1.1. OVERLAPPED Structure

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( )**.

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 has been opened without the FILE_FLAG_OVERLAPPED flag, the **lpOverlapped** parameter should be set to NULL. The function either completes the operation or times out. If the function returns TRUE, it has completed successfully. Otherwise, it has failed or timed out, and the calling thread calls the **GetLastError( )** function to retrieve the error.

4.1.2. I/O Completion Ports

An I/O completion port is a Windows NT scheduling construct. It is tied directly to a device handle and any I/O requests made to it. Using I/O completion ports is recommended if you want the notifications to match the I/O completions and you want to minimize context switches among your worker threads.

Use the Win32 API function call **CreateIoCompletionPort( )**, to create and set the parameters for an I/O Completion Port or to add handles to existing I/O Completion Ports.[1]

---

[1] Using I/O completion ports is fully documented in Win32 documentation. For a thorough discussion, see Jeffrey Richter's *Advanced Windows*, 3rd ed. Redmond, Wash: Microsoft Press.

CreateIoCompletionPort( ) returns the handle of the I/O completion port and takes the following arguments:

Parameter	Description
HFileHandle	File handle of device (in this case, the DM3 Mpath or Strm device, see *Chapter 5. DM3 Devices*) to associate with the I/O Completion Port
HExistingCompletionPort	Handle of I/O Completion Port if already created
DwCompletionKey	The key value associated with the device
DwNumberOfConcurrentThreads	Maximum number of concurrent threads you will allow to be running to process I/O completions

The following list shows additional items to remember:

- Note that the sequential order of notifications is not necessarily the same sequential order of I/O completions.
- While I/O completions can feed into scheduling algorithms, they are entirely asynchronous in nature. For example, by the time a thread is scheduled to run again, there may be any number of MMBs that may have completed.
- You must create the file handle of the DM3 device (an Mpath or Strm Device) with the FILE_FLAG set to FILE_FLAG_OVERLAPPED. This allows data movement of the specified device to "overlap" in time with other processing.
- Once an I/O completion port has been created, and DM3 devices are associated it, use the Win32 function call **GetQueuedCompletionStatus( )** to report the completion of the asynchronous I/O.
- Associate multiple handles with the I/O Completion Port by calling **CreateIOCompletionPort( )** additional times. You can either assign a completion key during each association (that is, one key per handle), or associate the same key for many handles.

The following sample code shows how a programmer might set up a C function using the items discussed previously:

```
////////////////////////////////////////////////////////////////////
//        NAME :  Dm3CompSetAsyncParams()
// DESCRIPTION :  Provides Async parameters to be used while sending
//                and receiving messages in async mode
//       INPUT :  lpComp      - the component instance
//                hIOCP       - the io completion port to use
//                dwIocpKey   - The key to be associated with the MPath
//                              used by the given component instance
//      OUTPUT :  None
//     RETURNS :  DM3SUCCESS or DM3FAIL
//    CAUTIONS :  Use GetLastError() to get error info
//                The application should remember the key passed into this
//                function. When this key is returned in the main loop by
//                GetQueuedCompletionStatus(), the application should call
//                Dm3CompProcIoCompletion() , to enable this component to
//                dispatch messages to the user of this object.
////////////////////////////////////////////////////////////////////
DM3STATUS Dm3CompSetAsyncParams(LPDM3COMP    lpComp,
                                HANDLE      hIOCP,
                                DWORD       dwIocpKey)
{

   if (lpComp == (LPDM3COMP)NULL)
   {
      SetLastError(ERROR_INVALID_PARAMETER);
      return DM3FAIL;
   }

   if (hIOCP != INVALID_HANDLE_VALUE)
   {
      lpComp->fSyncMode = FALSE;
      lpComp->hIOCP     = hIOCP;
      lpComp->dwIocpKey = dwIocpKey;

      /*
       * Associate the MPath with the given IO Completion port
       */
      if (CreateIoCompletionPort(lpComp->hMPath,
                                 hIOCP ,
                                 dwIocpKey,
                                 0)
                                 != hIOCP)
      {
         return DM3FAIL;
      }
      /*
       * We have successfully associated the MPath with the given
       * IO Completion port
       */
      return DM3SUCCESS;
   }
   else
   {
      SetLastError(ERROR_INVALID_PARAMETER);
      return DM3FAIL;
   }
}
```

Use **GetQueuedCompletionStatus( )** to return the number of bytes transferred, the completion key, and the address of the OVERLAPPED structure. Within Win32, the OVERLAPPED structure is used during asynchronous data movement and its pointer is also used as an anchor for DM3-specific message data structures. For a complete description of the **GetQueuedCompletionStatus( )** Win32 API call and the Win32 OVERLAPPED structure please refer to Win32 documentation.

4.1.3. Handling Asynchronous Function Returns

The operations detailed below and the flow chart in *Figure 5* describe the steps to follow when a function is called asynchronously.

1. A Direct Interface function will always return FALSE when called asynchronously. 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. 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( )**, **WaitForMultipleObjects( ),** or **GetQueuedCompletionStatus( )**.
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.

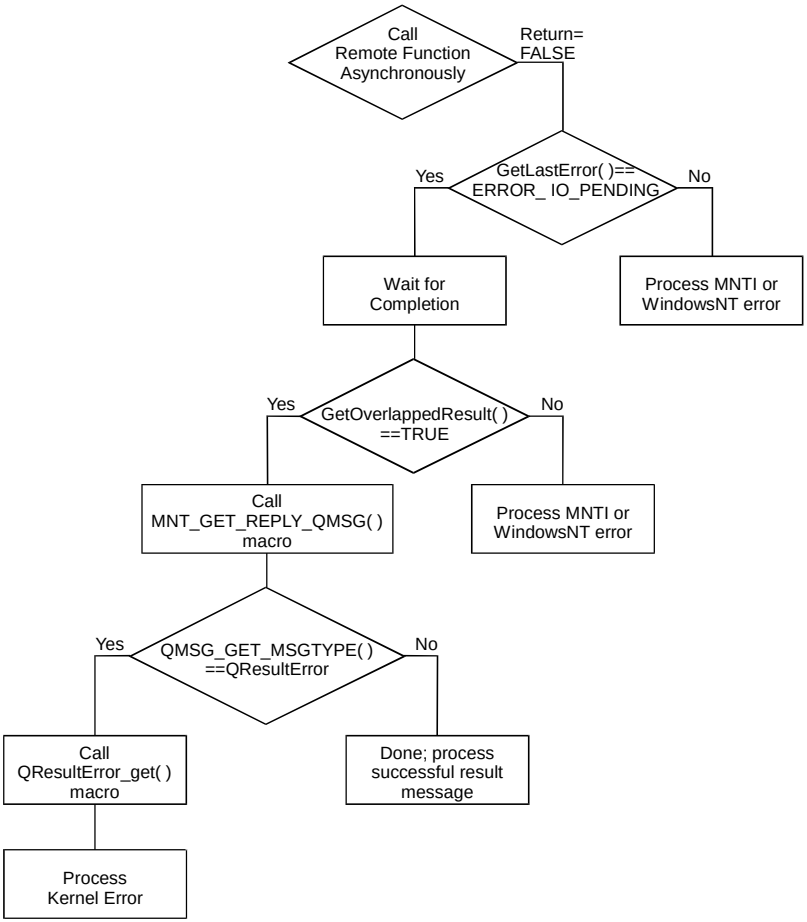

Figure 5. Handling Asynchronous Function Returns

This code fragment provides a general example of handling a function return asynchronously.

```
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;
    }
}
```

4.2. 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.

4.2.1. Handling Synchronous Function Returns

The operations detailed below and the flow chart in *Figure 6* describe the steps to follow when a function is 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 **mntCompFind( )** 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.

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.
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.

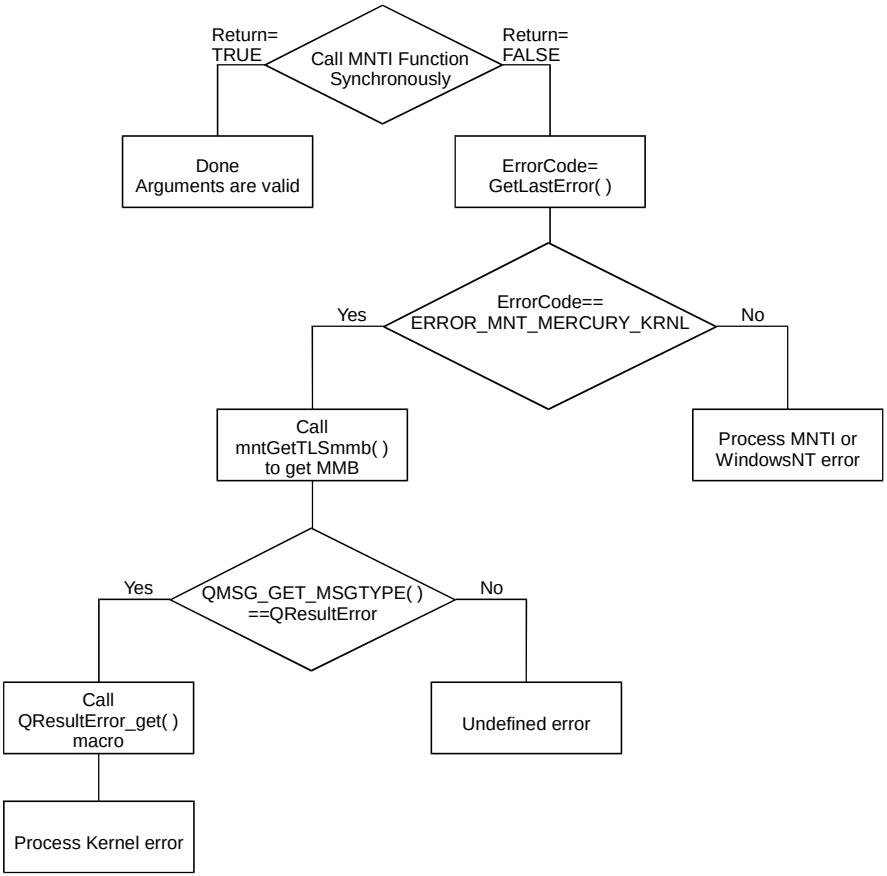

Figure 6. Handling Synchronous Function Returns

This code fragment provides a general example of handling a function return synchronously.

```
/* 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);
```

5. DM3 Devices

The Direct Interface is like other custom APIs that adhere to the Win32 model; it requires a kernel-mode device driver which it can talk to (in this case, the Class Driver). For custom operations, such as the messaging I/O, the Class Driver supports a full array of specialized functions (by using the Win32's IOCTL function) which make the Direct Interface completely compliant with the Win32 API.

When the Class Driver initializes, it creates a number of device names that end up in NT's object namespace, specifically under the `\Device` path. It also creates corresponding symbolic links under the `\DosDevices` root which the application uses in the **CreateFile( )** call. Without the handle returned from this call, no I/O is possible. The semantics of these device names are important.

The two types of DM3 I/O are differentiated primarily by size and the application protocol with the resource:

- **messages** are used for small transfers (as in command/reply messages)
- **streams** are used for bulk data transfers.

5.1. Device Names

You can use two different device types in your Direct Interface-based application:

- **Mpath**
  Mpath is a message type device type to support messaging I/O operations between the host and the embedded system. It allows the application to establish a logical connection to the driver.
- **Strm**
  Strm is a stream device type to support I/O operations for bulk data. It allows the application to establish a logical connection to a DM3 component on the embedded system.

NOTE: These are logical devices whose sole purpose is to serve as the bridge between the user space applications and the kernel space (that is, the Class Driver).

The Class Driver (DLGCMCD) implements both the Mpath and Strm devices and assigns names for each device sequentially (for example, MercMpath1, MercMpath2, MercMpath3, through MercMpath*n*, and MercStrm1, MercStrm2, MercStrm3, through MercStrm*n*).

5.1.1. Message Paths (Mpath)

A message path (Mpath) device is a generic conduit for communicating with DM3 component instances.[2] Because it is not bound permanently to its destination endpoint, you can use an Mpath device to communicate with any valid destination instance address by loading the handle of the device in the **mntSendMessage( )** function call. Since Mpath devices are not board-specific, you can use them to communicate with any DM3 board assembly.

The source address of an Mpath device is assigned at its creation time by the Class Driver, and it cannot be changed. In fact, much as with a client TCP port number (an "ephemeral" port), you should not generally be concerned about the source address. In most cases, the application only needs to know the destination address.

NOTE: There are instances when your host application would need to know about the source address (QCompDesc) of an Mpath device. For example, if your application must be notified of an asynchronous message from the firmware, the application must provide its own source address in the MMB.

Therefore, you can use a single Mpath device to communicate with any number of component instances. This approach can be feasible for one-time initializations. However, you would employ multiple Mpath devices if you need to communicate concurrently (asynchronously) with multiple destinations. In such a case, you can use multiple Win32 API handles with associated OVERLAPPED structures using the default source address matching.

---

[2] More specifically, a handle to an Mpath device is a ticket to the Class Driver space. Since the Class Driver created the device in the first place, once a handle is assigned to it, it is entirely up to NT's I/O Manager to maintain the link. For each handle returned to the user space, there is a corresponding file object in the kernel space which happens to be a waitable object. This is the base support for the various wait synchronization calls in Win32. Thus, it's possible to wait-synch upon handles directly or event objects of your own.

The question of how many Mpath devices to use boils down to this:

- If you wish to depend on the default source address matching and would like to do overlapped I/O in different threads, then you need multiple Mpath devices to correspond to each thread.
- If you are willing to use MMBs with proper matching criteria (the minimum being the source address), then you could open the same Mpath multiple times and bind the resulting handles to the same I/O completion port. This is convenient since you can use the key as a pointer to your I/O context information. But going further, it should be possible to use just a single handle to a single Mpath and rely upon a super, customized OVERLAPPED structure that would contain the context as well. (Note that if you open the Win32 handle with FILE_FLAG_OVERLAPPED, you can use this handle for asynchronous operations only.)

  There is, however, one caveat to this strategy and that pertains to exit notification (see *Section 9. Exit Notification*).

Using Mpath Devices in a Multithreaded Application

Always use multitasking and multithreading judiciously. Since they all share the same address space (as well as resources such as device handles), they incur overhead. Threads within a process incur less context-switch overhead than that among processes.

Although using threads is relatively straightforward, designing and implementing proper synchronization between them can become complex. Attempt to abide by the following guidelines:

- minimize the number of threads employed
- absolutely minimize the need for interactions such as sharing common device handles.

If you must share an Mpath handle across processes, then use **DuplicateHandle( )** to pass it via an Inter-Process Communication (IPC) mechanism such as shared memory. Further, make sure that MMBs are qualified with proper matching criteria to ensure proper I/O notifications when using the same Mpath (see *Section 6.4.1. Matching Criteria*).

If you wish to employ multiple threads in multiple processes and freely use both synchronous and asynchronous function calls, it's safest to use unique Mpath devices (thus, unique DM3 source addresses). Otherwise, you must be careful to use MMBs that are qualified with proper matching criteria. This avoids erroneous completion notifications.

The cost of an Mpath device is a small amount of non-paged pool space which poses only a minimal impact to the application design (if any). However, since there is a finite number of Mpath devices (specified either via a Registry parameter or the Class Driver default), you should not automatically allocate them on a permanent basis unless they are required to receive truly asynchronous messages.

Also, when you close the device handle and there are no more references to it (that is, you're performing the last close), the associated Mpath device name is freed and available for subsequent requests.

5.1.2. Stream Paths (Strm)

Unlike an Mpath device, a stream device is not a generic conduit. A stream device is a Win32 API vehicle for getting to a particular DM3 stream on a specific platform. Make this association by calling the **mntAttachMercStream( )** function and specifying the target board number and the stream number (if known).

A stream is unidirectional; it performs either read or write operations. You cannot write to a read stream or read from a write stream. Any attempt to do so results in an error. The direction of the stream is specified in the **mntAttachMercStream( )** function call, by setting the **mode** parameter to as either MNT_STREAM_FLAG_READ or MNT_STREAM_FLAG_WRITE.

A stream number of zero (0) has a special meaning; it indicates that an unallocated stream of specified size and direction should be created and opened.

5.1.3. Board Number

As a Direct Interface programmer, it is up to you to locate or discover the DM3 components with which you must communicate. Essential to both messaging and stream I/O is the address of the component instance, which contains the logical

board number. (NOTE: It is called the “logical” board number since it does not have to be equal to the physical number assigned by the Protocol Driver on its initialization.) In the Windows NT Registry, after a successful installation and configuration of the DM3 boards (assigned by the Dialogic Configuration Manager software), there are entries describing physical board instance numbers and associated logical attributes, one of which is the logical board number.

You can determine the board number by calling **mntGetBoardsByAttr( )** with the desired qualifying attributes.

This board number must be properly filled into the destination DM3 address field of an Multiple Message Block (MMB).

Example: Finding Boards in a DM3 System

The following sample code shows how to find all the boards in a DM3 system by using the **mntGetBoardByAttr( )** function.

```
#include  <stdio.h>
#include  <errno.h>
#include  <signal.h>
#include  <Windows.h>

#include  <Qhostlib.h>
#include <Qcluster.h>
#include  <mercdefs.h>
#include  <stddefs.h>

#define DEF_TIMEOUT   60/* Default timeout for MNTI functions */

#define MAX_NO_OF_BOARDS 4

void main( )
{
   int              n;
   QValueAttr      valAttr[2];
   ULONG           MaxBoardAttrs = MAX_NO_OF_BOARDS+1;
   QBoardAttr      boardAttr[MAX_NO_OF_BOARDS+1];
   ULONG           boardsFound=0x0;
   ULONG           maxBoards=1;

   DWORD           errorCode;
   UCHAR           boardNum;

   // Initilize memory

   ZeroMemory(valAttr, 2*sizeof(QValueAttr));
   ZeroMemory(boardAttr,MaxBoardAttrs*sizeof(QBoardAttr));

   // Fill out valAttr structure to locate Dm3 boards

   strcpy(valAttr[0].ValueName, "CurrentState");
```

```
   strcpy(valAttr[0].Value, "Running");
   valAttr[0].ValueType = REG_SZ;
   valAttr[0].ValueFlag = 0; /* match on 'Value' field */

   // Call mntGetBoardsByAttr function to search registry
   // to find boards with the matching attributes

   if (mntGetBoardsByAttr(valAttr,MaxBoardAttrs,
                          boardAttr,
                          &maxBoards,
                          &boardsFound) == FALSE)
   {
      errorCode = GetLastError();
      printf("Can't get boards attributes (0x%x)\n",
                                          errorCode);
      exit(0);
   }

   printf("Number of Dm3 boards found: %d \n",(int)boardsFound);
   for (n=0;n<(int) boardsFound;n++)
   {
      printf("Board Num: %d \n",boardAttr[n].BoardNo );
   }
}
```

5.2. Obtaining Device Name Strings

When coding your application, you create a handle for a device by passing the device name strings to the **CreateFile( )** function (see *5.3. Obtaining File Handles to Communicate with DM3 Devices*). You can obtain the device name strings by calling the **mntEnumMpathDevice( )** function for message devices or the **mntEnumStrmDevice( )** function for stream devices.

The sample code in section *5.3. Obtaining File Handles to Communicate with DM3 Devices* shows the syntax of the **mntEnum…Device( )** function.

5.2.1. Avoiding Sharing Violations

There is one potential pitfall when obtaining device name strings. If you try to get device names simultaneously in multiple threads, and make a subsequent **CreateFile( )** function call, it can fail with an ERROR_SHARING_VIOLATION due to contention. In other words, each **mntEnumMpathDevice( )** or **mntEnumStrmDevice( )** function call simply returns the next available, unopened device. Therefore, the thread that reaches the Class Driver first, wins.

In threads under your control, you can protect all calls to the **mntEnumMpathDevice( )**, **mntEnumStrmDevice( )**, and **CreateFile( )** functions as atomic operations. Simple retries, with or without delays, can work as well. Alternatively, you can stagger the start of the threads in your application.

5.3. Obtaining File Handles to Communicate with DM3 Devices

Once the device name is used to define a handle, you can open the device by assigning the handle to the **CreateFile( )** function.

The following sample code shows how to use the Direct Interface to create a utility function to create a message path.

```
//////////////////////////////////////////////////////////////////////
//        NAME :  Dm3CreateMPath()
// DESCRIPTION :  Utility function to create a Dm3 Message path
//       INPUT :  None.
//      OUTPUT :  None.
//     RETURNS :  Handle to the message path
//    CAUTIONS :  The message path is always opened with FILE_FLAG_OVERLAPPED
//////////////////////////////////////////////////////////////////////
HANDLE Dm3CreateMPath()
{
   ULONG    ulDevStatus          = 0;
   ULONG    ulMpathDevNameSize   = 0;
   HANDLE   hMPath               = INVALID_HANDLE_VALUE;

   CHAR     szMpathDeviceName[MNTI_MAX_DEVICE_NAME_SIZE];

   do
   {

      /*
       * Get first available Mpath device
       */
      if (!mntEnumMpathDevice(MNT_FIRST_AVAILABLE,
                      szMpathDeviceName,
                      &ulMpathDevNameSize,
                      &ulDevStatus))
      {
         return INVALID_HANDLE_VALUE;
      }

      /*
       * Open Mpath device handle
       */
      hMPath =  CreateFile(szMpathDeviceName,
                  GENERIC_WRITE | GENERIC_READ,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  FILE_FLAG_OVERLAPPED,
                  NULL);
   }
   while (hMPath == INVALID_HANDLE_VALUE);

   return hMPath;
}
```

6. Using Messages

This section describes how to build, send, and receive DM3 messages. To perform messaging I/O operations, you need a filled-in MMB (see the section entitled *Multiple Message Blocks (MMBs)*)and a handle to an Mpath device (see *Section 5.1.1. Message Paths (Mpath)*). The Direct Interface provides macros to fill in the fields of an MMB structure. After you have a Win32 API handle to the Mpath device, you can call **mntSendMessage( )**.

Messaging can be divided into several distinct parts:

1. Creating an I/O Completion Port (see *Section 4.1.2. I/O Completion Ports*)
2. Requesting an Mpath device name
3. Creating a handle to the Mpath Device
4. Allocating an MMB
5. Filling in MMB fields
6. Sending the message
7. Retrieving the Reply
8. Extracting Payload Data (if any)

There are many ways to retrieve message data after the Win32 system alerts the application to an I/O completion. When building your application, Dialogic recommends using I/O Completion Ports because this method is both efficient and easily scaled. Once the application has the MMB pointer, the complete received message may be accessed. See *Section 4.1.2. I/O Completion Ports*.

6.1. Requesting an Mpath Device Name

Enumerate a DM3 Mpath device by using the following Direct Interface function call:

- **mntEnumMpathDevice( )**

This function returns an available *Mpath device* that matches the specified criteria. Mpath devices are used for communicating with any DM3 component on any DM3 board in a system. The source address of the Mpath device is determined at creation time, but the desired destination address must be loaded into the MMB before sending a message. A message path to any component on any board can be established by loading its address as the destination address of an Mpath. The Mpath device type supports messaging I/O operations.

6.2. Creating a Handle to the Mpath Device

Open the device using the following Win32 API function call:

- **CreateFile( )**

When the device is created/opened, you must set a File_Flag argument. By setting the flag to FILE_FLAG_OVERLAPPED, you indicate that this device may operate either synchronously or asynchronously. This function returns a Win32 device handle for the **DeviceName** that was previously specified in the **mntEnumMpathDevice( )** call.

Call **mntCompFind( )** or **mntCompFindAll( )** to get the destination address of the device. These function calls are presented in the *DM3 Direct Interface Function Reference for Windows NT*.

6.3. Allocating a Multiple Message Block (MMB)

Allocate and initialize an MMB with the following Direct Interface function:

- **mntAllocateMMB( )**

You must allocate system memory for both the data that will comprise the message to be sent, and the expected reply message. The **nReplyMaxSize** parameter indicates the maximum size in bytes of all replies that might be received by the host.

NOTE: There is no MMB structure associated with stream devices. Instead, a stream has defined components at both ends to source and synchronize data.

6.4. Filling in MMB Fields

This section discusses how to format a message. DM3 messages are sent through a system in a packed-byte format. Pre-built message access macros are provided which transparently deal with the Endian issues of packing and unpacking message data.

DM3 message macros are defined in a set of resource-specific host-side header files and in the standard message header (*stddefs.h*).

The following types of macros are provided:

- **MMB Control Header Macros**
  These macros are available to access the MMB header. Examples of these macros are MNT_PUT_MMB_CMD_SIZE and MNT_PUT_MMB_EXPECTED_REPLY_COUNT.
- **DM3 Message Macros**
  There are three types of DM3 Message Macros:
  - DM3 Message Pointer macros
  - DM3 Message Header macros
  - DM3 Message Payload macros

For details on specific macros, consult the *DM3 Direct Interface Function Reference for Windows NT*.

Use the DM3 message macros to get the pointer to the command or reply messages, to access command or reply message headers, and to extract command or reply message payloads. MNT_GET_CMD_QMSG( ), QMSG_SET_MSGTYPE( ), and QComponentResult_get( ) are examples of message pointer, message header, and message payload macros. Command and reply message headers are of the type QMsg.

6.4.1. Matching Criteria

Setting flags on the MMB allows the driver to match reply messages from the firmware to the command message sent from the host. Before sending an MMB, you can set the matching criteria in the Flags field. The driver uses these to match incoming replies and declare the I/O requests to be completed. *Table 2* lists the criteria on which you can match.

It is important to do proper matching, otherwise your application may never receive messages intended for it. Matching is an AND operation (not an OR operation), the matched reply must meet all matching criteria.

The broadest coverage is provided by the MATCH_ON_SRC_ADDR flag, which is the default. It matches all replies destined for a host-side DM3 address.

Adding any or all of the optional completion option flags lets you tighten the matching requirements as follows:

- Add the optional MATCH_ON_DEST_ADDR flag if you wish to receive reply messages only from the same component instance specified in the MMB.
- Add the optional MATCH_ON_TRANSACTION_ID flag if you expect reply messages returned with the same transaction ID as in the message sent.
- Add the optional MATCH_ON_MSG_TYPE flag if you expect reply messages returned with the same message type as in the message sent. Use this flag in conjunction with an empty message to receive asynchronous messages such as alarms or events.

Use different matching criteria for expected replies and unsolicited messages:

- For expected replies, use the MATCH_ON_TRANSACTION_ID flag, and avoid using MATCH_ON_MSG_TYPE (since most commands can have either successful or unsuccessful reply messages).
- For unsolicited messages, use the MATCH_ON_MSG_TYPE flag (because you are setting up an MMB to receive a specific type of message, such as *Std_MsgEvtDetected*), and avoid using MATCH_ON_TRANSACTION_ID.

NOTE: Although the Direct Interface automatically sets the MATCH_ON_SRC_ADDR, other flags are optional. Therefore, although you don't need to set this flag yourself, you cause no harm by doing so.

Table 2. Matching Criteria

Matching Criteria	**Required or Optional**	**the match is between:**
MATCH_ON_SRC_ADDR	Required and set by default.	destination address of the incoming message and the source address of the command message in the MMB.
MATCH_ON_DEST_ADDR	Optional	source address of the incoming message and the destination address of the command message in the MMB.
MATCH_ON_TRANSACTION_ID	Optional	transaction ID of the incoming message and the transaction ID of the command message in the MMB.
MATCH_ON_MSG_TYPE	Optional	message type of the incoming message and the message type of the command message in the MMB.

6.5. Sending the Message

Once the MMB has been allocated and filled in, you can send it and begin waiting for the reply. Send the message using the following Direct Interface function call:

- **mntSendMessage( )**

Make sure you include

- the correct Mpath device
- the MMB pointer to reference the correct message data for sending of the data and to place expected replies
- the overlapped pointer (for Win32 to use) during asynchronous data movement.

Wait for the reply using one of the following Win32 functions:

- **WaitForSingleObject( )**
- **WaitForMultipleObjects( )**
- **GetQueuedCompletionStatus( )**

NOTE: Using **GetQueuedCompletionStatus( )** is possible only if an I/O completion port has been created and the DM3 device has been associated with the IOCP (see *Section 4.1.2. I/O Completion Ports*).

6.5.1. Sending Asynchronously or Synchronously

Functions that can operate asynchronously or synchronously accept the **lpOverlapped** parameter. Specify the mode by setting the **lpOverlapped** parameter to either NULL (for synchronous) or non-NULL (for asynchronous or overlapped). When asynchronous, the function returns immediately before the actual I/O completes.

For asynchronous, set the **lpOverlapped** parameter to the OVERLAPPED pointer. The OVERLAPPED structure is an asynchronous I/O data structure from the Win32 API.

If you're using a synchronous model, the call will block until either an error or a reply is received. Then, the application must look at the reply portion of the MMB and extract the reply message.

6.5.2. Example: Sending and Receiving a Simple Message

The following example shows an example function called **Dm3CompProcIoCompletion( )**. This sends a simple message (with no payload) to a component. If synchronous mode is set, this returns the reply message. Otherwise, callback is called whenever a message is received.

```
#define DM3COMP_SEND_SIMPLE_MSG(lpComp, MsgType)                    \
{                                                                   \
   LPMMB       lpMMB       = NULL;                                  \
   QMsgRef     lpMsg       = NULL;                                  \
   ULONG       ulCmdSize   = 0;                                     \
                                                                    \
   ulCmdSize = sizeof(QMsg) + MsgType##_Size;                       \
                                                                    \
   lpMMB = mntAllocateMMB( ulCmdSize,                               \
                           (lpComp)->ucExpectedReplyCount,          \
                           (lpComp)->ulMaxReplySize);               \
                                                                    \
   if (lpMMB != (LPMMB)NULL)                                        \
   {                                                                \
      MNT_GET_CMD_QMSG(lpMMB, &lpMsg);                              \
      QMSG_SET_MSGTYPE(lpMsg, MsgType);                             \
      Dm3CompSendAndRecvMsg(lpComp, lpMMB);                         \
   }                                                                \
}
```

6.5.3. Example: Sending a Fixed-Size Message

This following macro shows the code to send a message with a fixed size payload.

```
#define DM3COMP_SEND_FIXED_SIZE_MSG(lpComp, MsgType, lpData)      \
{                                                                 \
   LPMMB    lpMMB    = NULL;                                      \
   QMsgRef  lpMsg    = NULL;                                      \
   UINT     unOffset = 0;                                         \
   ULONG    ulCmdSize= 0;                                        \
                                                                  \
   ulCmdSize = sizeof(QMsg) + MsgType##_Size;                     \
                                                                  \
   lpMMB = mntAllocateMMB( ulCmdSize,                             \
                           (lpComp)->ucExpectedReplyCount,        \
                           (lpComp)->ulMaxReplySize);             \
   if(lpMMB != (LPMMB)NULL)                                       \
   {                                                              \
      MNT_GET_CMD_QMSG(lpMMB, &lpMsg);                            \
      QMSG_SET_MSGTYPE(lpMsg, MsgType);                           \
      MsgType##_put(lpMsg, lpData, unOffset);                     \
      Dm3CompSendAndRecvMsg(lpComp, lpMMB);                       \
   }                                                              \
}
```

6.5.4. Example: Sending a Variable Payload Message

The following example shows a macro that sends a message with a fixed and a variable sized payload. If syncronous mode is enabled, it returns with the reply message. This macro is used to append an array of data structures to the end of fixed portion of the payload. The payload's fixed portion typically includes a count for the variable payload.

```
#define DM3COMP_SEND_VAR_SIZED_MSG(lpComp,                          \
                                   MsgType,                         \
                                   lpData,                          \
                                   VarCount,                        \
                                   VarFieldDef,                     \
                                   lpVarData)                       \
{                                                                   \
   LPMMB    lpMMB    = NULL;                                        \
   QMsgRef  lpMsg    = NULL;                                        \
   UINT     unOffset = 0;                                           \
   ULONG    ulCmdSize= 0;                                           \
   INT      nCounter = 0;                                           \
                                                                    \
   ulCmdSize = sizeof(QMsg) +                                       \
               MsgType##_Size +                                     \
               (sizeof(VarFieldDef##_t) * VarCount);                \
                                                                    \
   lpMMB = mntAllocateMMB(ulCmdSize,                                \
                          (lpComp)->ucExpectedReplyCount,           \
                          (lpComp)->ulMaxReplySize);                \
                                                                    \
   if (lpMMB != (LPMMB)NULL)                                        \
   {                                                                \
      MNT_GET_CMD_QMSG(lpMMB, &lpMsg);                              \
      QMSG_SET_MSGTYPE(lpMsg, MsgType);                             \
      MsgType##_put(lpMsg, lpData, unOffset);                       \
                                                                    \
      unOffset = MsgType##_varStart;                                \
      for (nCounter = 0; nCounter < (INT)(VarCount); ++nCounter)    \
      {                                                             \
         qMsgVarFieldPut(lpMsg, 1, &unOffset,                       \
                         VarFieldDef, &(lpVarData[nCounter]));      \
      }                                                             \
      Dm3CompSendAndRecvMsg(lpComp, lpMMB);                         \
   }                                                                \
}
```

6.5.5. Example: Sending a Variable List Message

This example shows how to send a message that contains a list of elements at the end of the fixed portion of the payload.

```
#define DM3COMP_SEND_LIST_MSG(lpComp,                          \
                              MsgType,                         \
                              lpData,                          \
                              VarCount,                        \
                              lpVarData)                       \
{                                                              \
   LPMMB    lpMMB    = NULL;                                   \
   QMsgRef  lpMsg    = NULL;                                   \
   UINT     unOffset = 0;                                      \
   ULONG    ulCmdSize= 0;                                      \
   INT      nCounter = 0;                                      \
                                                               \
   ulCmdSize = sizeof(QMsg) +                                  \
               MsgType##_Size +                                \
               (sizeof(MsgType##_List_t) * VarCount);          \
                                                               \
   lpMMB = mntAllocateMMB(ulCmdSize,                           \
                          (lpComp)->ucExpectedReplyCount,      \
                          (lpComp)->ulMaxReplySize);           \
                                                               \
   if (lpMMB != (LPMMB)NULL)                                   \
   {                                                           \
      MNT_GET_CMD_QMSG(lpMMB, &lpMsg);                         \
      QMSG_SET_MSGTYPE(lpMsg, MsgType);                        \
      MsgType##_put(lpMsg, lpData, unOffset);                  \
                                                               \
      unOffset = MsgType##_varStart;                           \
      for (nCounter = 0; nCounter < (INT)(VarCount); ++nCounter)  \
      {                                                        \
         MsgType##_List_put(lpMsg,&(lpVarData[nCounter]),unOffset); \
      }                                                        \
      Dm3CompSendAndRecvMsg(lpComp, lpMMB);                    \
   }                                                           \
}
```

6.5.6. Example: Sending a KVSet Message

This example shows a macro that prepares a message that has supplementary KV set data.

```
#define DM3COMP_PREP_KVS_MSG(lpComp, lpMMB, MsgType, lpData, KvSize)    /
{                                                                       /
   QMsgRef  lpMsg    = NULL;                                            /
   UINT     unOffset = 0;                                               /
   ULONG    ulCmdSize= 0;                                               /
                                                                        /
   ulCmdSize = sizeof(QMsg) + MsgType##_Size + KvSize;                  /
                                                                        /
   (lpMMB) = mntAllocateMMB(ulCmdSize,                                  /
                            (lpComp)->ucExpectedReplyCount,             /
                            (lpComp)->ulMaxReplySize);                  /
   if(lpMMB != (LPMMB)NULL)                                             /
   {                                                                    /
      MNT_GET_CMD_QMSG(lpMMB, &lpMsg);                                  /
      QMSG_SET_MSGTYPE(lpMsg, MsgType);                                 /
      MsgType##_put(lpMsg, lpData, unOffset);                           /
   }                                                                    /
}
```

6.6. Retrieving a Reply Message

Extract the reply from the MMB using the message macros. Once the application receives an event, the type of message received will determine the routine to be executed. Message macros are used to get the MMB pointer and to extract the data from the MMB.

This technique is the sole method for capturing events from the DM3 embedded system. Various methods are available for the Win32 subsystem to inform the application that a message has been received. When using the IOCP, a device key and overlapped pointer are furnished by the Win32 function call **GetQueuedCompletionStatus( )**. Use that data to resolve the MMB pointer.

Since every asynchronous message or stream device must have a unique overlapped pointer, use the value of the overlapped pointer for a user-defined structure. By type casting this overlapped pointer to this user-defined structure, the overlapped pointer then serves two purposes:

- the intended use of the overlapped pointer for the Win32 subsystem
- as the pointer to the user defined structure.

This structure will contain at least the MMB pointer and the device handle.

Once you're done using an MMB, you will want to free up the memory by using **mntFreeMMB( )**.

6.6.1. Retrieving Messages from the Completion Port

If you are building an application that uses DM3 boards and other Dialogic boards, you'll need to parse the returned message to determine which card the message is from. Here's an example shown in pseudo-code:

```
switch (CompletionKey) {
case SRL_KEY:
Parse SRL event
case IPT_KEY:
Parse IPT event
```

1. Use the Win32 function **GetQueuedCompletionStatus( )** to retrieve the OVERLAPPED structure on which I/O was successful.
2. Retrieve the MMB associated with this overlapped structure when you sent the message.
3. Use the Direct Interface macros to retrieve the reply mesage header and payload (if any) of this MMB.
4. Retrieve the reply message type. The message may indicate a successful completion or an error.
5. Process the reply accordingly.

The following example is from DM3 Application Foundation Code:

```
// Check for event on IO Completion port.
   bOk = GetQueuedCompletionStatus(hIocp,
                                   &dwByteCount,
                                   &dwDm3Key,
                                   &lpOverlapped,
                                   1000);
   // Get the Last Error.
   dwLastError = GetLastError();

   if (!bOk)
   {
      if (dwLastError == WAIT_TIMEOUT)
      {
         return TRUE;
      }
   }

// Process IO Completion Event.
   switch(dwDm3Key)
   {
      case DM3_COMP_KEY:
         Dm3CompProcIoCompletion(lpOverlapped, dwLastError);
         break;

      case DM3_STREAM_KEY:
```

```
            Dm3StrmProcIoCompletion(lpOverlapped, dwLastError);
            break;

         case DM3_FILE_KEY:
            Dm3FileProcIoCompletion(lpOverlapped, dwLastError);
            break;

         default:
            printf("Received unknown Dm3Key\n");
            return FALSE;
            break;
      }

      return TRUE;
}
```

6.7. Handling Unsolicited Messages

Some messages are sent by the DM3 firmware or drivers, but they are not sent in response to a command message from the application. These are called **unsolicited messages**.

To specify an unsolicited message MMB, use the MNT_SET_MMB_EMPTY_MSG( ) macro. This macro posts an empty message MMB that has no command message to send, but has room for a specified number of replies. Set the destination instance address in the MMB (even though no message is actually being sent).

When an unsolicited message arrives, the device driver must match it to one of the pending MMBs using the destination address. Unless a single thread is dedicated for fielding all messages from the DM3 board, an empty MMB should specify more qualified matching criteria such as the destination address or the message type, and/or the transaction ID.

6.7.1. Waiting for an Event

The application will wait for any asynchronous I/O to complete using the following Win32 function call:

- **GetQueuedCompletionStatus( )**

NOTE: This function call will block until some activity completes or the specified timeout occurs.

6.8. Canceling Pending Messages

To cancel a specific message that has already been sent, call the following Win32 function:

- **CancelIo( )**

If the message is canceled, your application will be notified via an I/O Completion Port with an error code of ERROR_IO_ABORTED.

6.9. Example: Sending a Message and Receiving a Reply

The following code segment shows the typical steps in sending a Std_MsgSetParm message and receiving a Std_MsgSetParmCmplt reply. These steps include:

1. Enumerating for an Mpath device name.
2. Obtaining a handle to the device.
3. Finding a DM3 board that matches your requirements.
4. Getting a destination address on the target board.
5. Allocating an MMB for the command and reply messages, then initializing the MMB.
6. Sending the message and waiting for a reply.
7. Extracting the reply from the MMB, then processing it after the reply is received.

NOTE: For details on DM3 messages and related macros, please see the specific component interface specifications in the guide entitled *DM3 Standard Component Interface Messages*.

```
// Example: Sending a Message and Receiving a Reply

#include  <stdio.h>
#include  <errno.h>
#include  <signal.h>
#include  <Windows.h>

#include  <Qhostlib.h>
#include <Qcluster.h>
#include  <mercdefs.h>
#include  <stddefs.h>
```

```
// QVS resource header files

#include <tscdefs.h>

#define DEF_TIMEOUT   60 /* Default timeout for MNTI functions */

#define MAX_NO_OF_BOARDS 4

// Prototypes

BOOL sendMsg( );

// Main Routine

void main( )
{
  if ( sendMsg( )!=TRUE)   printf("Error in sendMsg \n");
}

// Code segment showing how to send a Std_MsgSetParm
// message using MNTI

BOOL sendMsg( )
{
   QMsgRef              pMsg;
   LPMMB           lpMMB;
   QCompDesc       InstDesc;
   QCompAttr       Attr[2];
   CHAR            DeviceName[MNTI_MAX_DEVICE_NAME_SIZE];
   ULONG           DeviceNameSize;
   DWORD           DeviceStatus;
   HANDLE          hMpath, hMsgEvent;
   DWORD           ErrorCode;
   UCHAR           boardNum=0;
   DWORD           CommandSize;
   DWORD           ReplyCount;
   DWORD           ReplyMaxSize;
   DWORD           offset=0;
   Std_MsgSetParm_t Parm;
   QTrans          TransID;
   DWORD           RecvByteCount;
   DWORD           ReplyType;
   OVERLAPPED      Overlapped;

   // Find an Mpath device name

   if (mntEnumMpathDevice(MNT_FIRST_AVAILABLE,
                  DeviceName,
                  &DeviceNameSize,
                  &DeviceStatus) == FALSE)
   {
       // Call GetLastError to get the error code

       ErrorCode = GetLastError();

       // perform error handling

       printf("Error %d in mntEnumMpathDevice \n",ErrorCode);
       return(FALSE);
   }
```

```
// Open the device file and get a handle

if ((hMpath = CreateFile(DeviceName,
                GENERIC_READ | GENERIC_WRITE,
                FILE_SHARE_READ | FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                FILE_FLAG_OVERLAPPED,
                NULL))== INVALID_HANDLE_VALUE)
{
  // Call GetLastError to get the error code

  ErrorCode = GetLastError();

  // perform error handling

     printf("Error %d in CreateFile \n",ErrorCode);
  return(FALSE);
}

// Find the TSC instance

Attr[0].key=Std_ComponentType;
Attr[0].value=TSC_Std_ComponentType;
Attr[1].key=QATTR_NULL;
Attr[1].value=0xfb;

InstDesc.node      = 0;
InstDesc.board     = 0;
InstDesc.processor = QCOMP_P_NIL;
InstDesc.component = QCOMP_C_NIL;
InstDesc.instance = 1;

if ( mntCompFind(hMpath,mntTransGen(),
              &InstDesc,Attr,DEF_TIMEOUT,
              NULL,NULL)== FALSE)
{
    // Call GetLastError to get the error code
    ErrorCode = GetLastError();
    printf("Error %d in mntComp \n",ErrorCode);
    return (FALSE);
}
else
    printf("TSC QCompDesc: (%d:%d:%d:%d:%d) \n",
              (InstDesc.node) ,
              (InstDesc.board) ,
              (InstDesc.processor) ,
              (InstDesc.component) ,
              (InstDesc.instance) );


// Allocate an MMB for our purpose

CommandSize = sizeof(QMsg) + Std_MsgSetParm_Size;
ReplyCount = 1;
ReplyMaxSize = sizeof(QMsg) + Std_MsgSetParmCmplt_Size;
if ( (lpMMB = mntAllocateMMB(CommandSize, ReplyCount, ReplyMaxSize))== NULL)
{
    ErrorCode = GetLastError();
    // perform error handling
    printf("Error %d in mntAllocateMMB \n",ErrorCode);
    return(FALSE);
```

```
}

// Get the start of command QMsg

MNT_GET_CMD_QMSG(lpMMB, &pMsg);

// Fill in message header

QMSG_SET_MSGTYPE(pMsg, Std_MsgSetParm);
QMSG_SET_DESTADDR(pMsg, &InstDesc);

// Should keep the Transaction ID unique for each message

TransID = mntTransGen();
QMSG_SET_TRANS(pMsg, TransID);

// set the parameter for the message body

Parm.Num = TSC_ParmEncoding;
Parm.Val = TSC_ParmEncoding_Mulaw;

// Copy message body to MMB

Std_MsgSetParm_put(pMsg, &Parm, offset);

// MMB Ready to ship.  Setup overlapped for async message sending

ZeroMemory((PVOID) &Overlapped, sizeof(OVERLAPPED));

hMsgEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
Overlapped.Internal     = 0;
Overlapped.InternalHigh = 0;
Overlapped.Offset       = 0;
Overlapped.OffsetHigh   = 0;
Overlapped.hEvent       = hMsgEvent;

printf("Ready to send Msg \n");
// Now send the message to the board

if (mntSendMessage(hMpath, 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( (Overlapped.hEvent),
                                       INFINITE)) == WAIT_FAILED)
       {
            // perform error handling
                printf("WaitForSingleObject Failed \n");
            return(FALSE);
       }
       if (GetOverlappedResult(hMpath,&Overlapped,
                                    &RecvByteCount,FALSE) == FALSE)
       {
            // Call GetLastError to get the error code
            ErrorCode = GetLastError();
                printf("Error %d in GetOverlappedResult \n",
                                                          ErrorCode);
            // perform error handling
            return(FALSE);
```

```
         }
      }
   }
   else
      printf("Error %d in mntSendMessage \n",ErrorCode);

   // At this point, the reply message is in the MMB reply section

   MNT_GET_REPLY_QMSG(lpMMB,1, &pMsg);
   QMSG_GET_MSGTYPE(pMsg, &ReplyType);

  // Now process according to reply type

  printf("Received MsgType: %d  Expected Type: %d \n",
                   ReplyType,Std_MsgSetParmCmplt);

  return TRUE;
}
```

6.10. Using Attributes to Find a Component

When selecting a component that you want to use, pass the criteria of your selection to a function call. Do this by passing specific **attributes** of the component. Attributes describe a component's characteristics, for example, an Automatic Speech Recognition (ASR) component may have attributes that allow you to select it based on certain exclusive features (such as discrete recognition, word spotting, or the technology developed by a particular vendor).

To select a component, pass an array of keys and values to either **mntClusterByCompFind( )**, **mntCompFind( )** or **mntCompFindAll( )**. Note that when you're developing this array, you must end the list of attributes by defining the key as QATTR_NULL.

The names of attributes and their values are defined in the resource's documentation and header file. The header files are found in the inc directory which contains a number of *<resource>defs.h* files (for example, *playdefs.h*, *recdefs.h*).

The following code example shows an array (ClusterAttrs) that can be passed into a …CompFind( ) function:

```
QCompAttr     ClusterAttrs[3];
ClusterAttr[0].key = Std_ComponentType;
ClusterAttr[0].value = TSC;
ClusterAttr[1].key = TSC_AttrProtocolBase;
ClusterAttr[1].value = TSC_ParmProtocolBase_H323;
ClusterAttr[2].key = QATTR_NULL;
```

6.10.1. Standard Component Types

Dialogic has defined the following standard component types. These are defined as constants in header files included with the components available on the board. If the DM3 system you're working with uses these technologies, you can use these by making the statement `key = Std_ComponentType`, followed with a `value` set to one of the following:

For this standard component type…	**set the value to…**
Audio Decoder	ADec_Std_ComponentType
Audio Encoder	AEnc_Std_ComponentType
Call Analysis	CA_Std_ComponentType
Channel Associated Signaling (CAS)	CAS_Std_ComponentType
Common Channel Signaling (CCS)	CCS_Std_ComponentType
Channel Protocol (CHP)	CHP_Std_ComponentType
Line Control	LCON_Std_ComponentType
NetTSC	NetTSC_Std_ComponentType
Package Version	PkgVersion_Std_ComponentType
Player	Player_Std_ComponentType
Recorder	Recorder_Std_ComponentType
SCBus	QSCRES_Std_Component_Type
Signal Event Buffer	SB_Std_ComponentType
Signal Event Detector	SD_Std_ComponentType
Springware	Spng_Std_ComponentType
Tone Generator	Tgen_Std_ComponentType
Telephony Services Component (TSC)	TSC_Std_ComponentType
Waveform Generator	WGen_Std_ComponentType

7. Using Data Streams

The DM3 device driver uses data blocks to pass data streams between the host and the DM3 embedded system. These blocks also carry attribute information that you can use to control data transfer. To set attribute information properly, the host application needs to be aware of this block-oriented data transfer.

Writing streams (bulk data) is similar to writing messages because the stream I/O operations complete as soon as the driver writes them to the Shared RAM on the embedded system. However, the completion of this type of I/O operation indicates only that the stream data was delivered to the on-board memory, not that it was properly picked up and delivered to its destination instance. It is the responsibility of the application and resource protocol whether such an acknowledgment is expected or not.

Figure 7 shows an overview of stream flow.

Because the message and stream data travel through independent queues, you cannot assume a first-sent/first-received sequence between the messages and stream data; there is no guarantee that sent messages and stream data are received by the component instance in the same sequence. To address this issue, some resource-specific streams have header flags associated with each transferred data block. The Direct Interface provides the following predefined flags:

- EOD (end of data)
- EOT (end of transmission)
- EOF (end of file)

Depending on the resource, you may or may not need to use the header flags. For more information on these flags, see *Section 7.1.3. Setting Stream Flags*.

There are five additional bits that you can use for defining other stream header flags. Specific meanings and their usage protocols are entirely determined by the application and its counterpart on the embedded system.

For requests made by **ReadFile( )** function calls, the device driver matches these header flags against those in the incoming data block. If the device driver finds matching flags, the I/O request completes successfully. Although you can examine

the flags by calling the **mntGetStreamHeader( )** function, you need not do so unless the requested transfer count differs from the actual count. In any case, if the **ReadFile( )** function returns TRUE, the returned transfer count is correct regardless of whether the I/O operation was synchronous or asynchronous.

NOTE: You can also specify MNT_STREAM_FLAG_IGNORE_HEADER in the **mntAttachMercStream( )** call in order to instruct the device driver to ignore the header portion of the SRAM data block, in effect, defeating the processing of the above flags.

Streams are uni-directional, with one end opened to read and the other opened to write. If the read stream has been closed by the sending end, all pending read requests complete prematurely, and COMPLETE_ON_EOF is set in the **bufFlags** field, and STREAM_CLOSED is set in the **sysFlags** field. If the read stream has not been closed by the sending end, the read operation completes when either the specified number of bytes have been received or the time out value expires (ERR_SEM_TIMEOUT).

Process: Stream Flow

Stream flow generally occurs according to a process similar to the following outline. This discussion corresponds to *Figure 7. Direct Interface Stream Flow*:

1. The application calls **mntEnumStrmDevice( )**, receives a Strm device, and calls **CreateFile( )** for a Win32 handle.
2. Then, the application calls **mntAttachMercStream( )** to open a stream in either the read or write direction (by setting the **nModeFlags** parameter to either MNT_STREAM_FLAG_READ or MNT_STREAM_FLAG_WRITE).
3. The application can issue multiple reads or writes to the Strm device using Win32 overlapped I/O (and await completion via Win32 synchronization calls), or block for synchronous execution.
4. The Class Driver and Protocol Driver handle internal mappings and stream management functions.
5. The firmware on the embedded system transfers stream data between the host and the board.

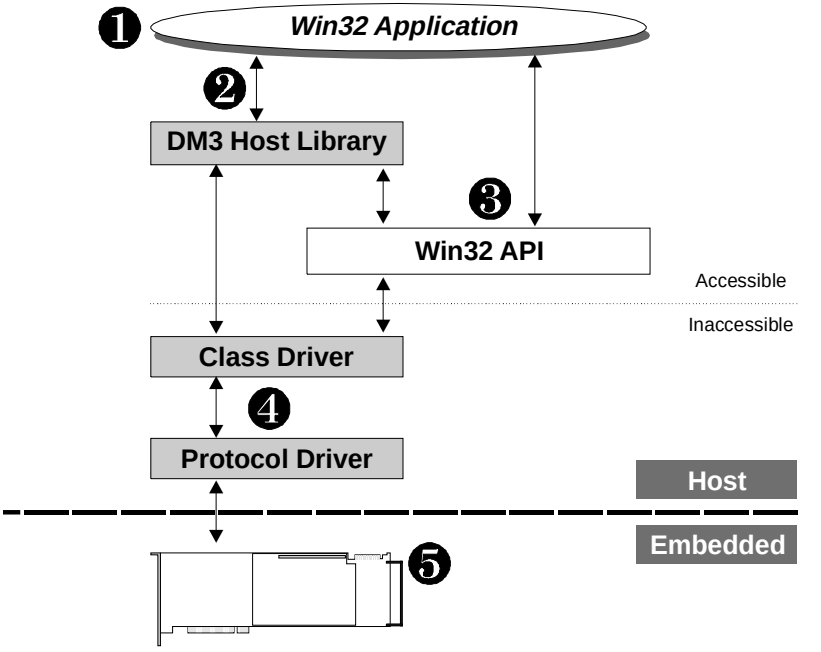

Figure 7. Direct Interface Stream Flow

7.1. Writing Stream Data

Writing stream data means transmitting data from the host application to the DM3 embedded system. For sample code, see *Section 7.1.1. Example: Writing Stream Data*.

Procedure

To program write streams, use the following procedure. Please note that enumerating a stream device name and creating a file handle must be atomic functions (see *Section 5.2.1. Avoiding Sharing Violations* for more information).

1. Enumerate a stream device by using the following Direct Interface Function call:

- **mntEnumStrmDevice( )**

  A *Strm* device is used to move large amounts of data between the host and the DM3 card; a *Strm* is created to be either a read or write stream. *Strm* devices are for streaming I/O operations.

2. Open the device handle by using the following Win32 API function call:

   - **CreateFile( )**

   Pass the GENERIC_WRITE and FILE_FLAG_OVERLAPPED flags in this function. Setting the flag to FILE_FLAG_OVERLAPPED indicates that asynchronous data movement will be done for this device. This function returns a Win32 device handle for the specified DeviceName; the same DeviceName that was previously specified in the **mntEnumStrmDevice( )** Direct Interface function call.

3. Get a unique stream ID using the following Direct Interface function call:

   - **mntAttachMercStream( )**

4. Pass the **hDevice**, **nBoardNumber**, **nModeFlags**, **lpMercStreamID**, **nStreamSize**, and **nTimeout** parameters to the **mntAttachMercStream( )** function. The Win32 device handle returned by the **CreateFile( )** function call is passed to this function. This function attaches a stream ID to the specified Stream device. Pass an OVERLAPPED structure to this function if you want the command message to go to the firmware asynchronously; pass NULL if synchronously. The mode flag should specify MNT_STREAM_FLAG_WRITE.

5. If the resource on the DM3 embedded system requires header flags, specify the fields in the STRM_HDR structure by calling the following function:

   - **mntSetStreamHeader( )**

   Pass the **bufFlags** fields to the function.

6. Set up the data to be transferred by setting up an OVERLAPPED structure and passing it to the **WriteFile( )** function to send the data.

7. Repeat step 6 until all data blocks, except the last, have been sent to the DM3 board.

8. If the resource on the DM3 embedded system requires header flags when the last data block has been sent, set the appropriate flag in the **bufFlags** field of

STRM_HDR structure (such as MNT_EOT, MNT_EOS, or MNT_EOD). Then, pass the **bufFlags** parameter to the **mntSetStreamHeader( )** function.

9. Call the **WriteFile( )** function to send the last data block.
10. If necessary, call the **mntDetachMercStream( )** function to close the stream device.

7.1.1. Example: Writing Stream Data

The following code segment shows the typical steps in setting up and sending data to a stream. Please note that this is a synchronous example. These steps include:

1. Enumerating a stream device name.
2. Opening the device file and obtaining the Win32 handle.
3. Getting the destination address of a Player instance.
4. Attaching a stream to the stream device.
5. Writing data to the stream.

```
// Example: Writing Stream Data
#include <stdio.h>
#include <windows.h>
#include <qhostlib.h>

#include <tscdefs.h>
#include <playdefs.h>
#include <coders.h>

#define DEF_TIMEOUT  60

BOOL writeStrmData(HANDLE hFile);

void main( )
{
   HANDLE hFile;

   /* Open data file */

   if ( (hFile = CreateFile( "mercury.rvx",
                             GENERIC_READ,
                             0,
                             NULL,
                             OPEN_EXISTING,
                             FILE_ATTRIBUTE_NORMAL,
                             NULL
                             )) == INVALID_HANDLE_VALUE )
   {
        printf("Can't get a handle to mercury.rvx \n");
        exit(0);
   }
```

```
   // Open data file

   SetFilePointer( hFile, 0, NULL, FILE_BEGIN);


   // Call "writeStrmData" to send data to player

   printf("Calling writeStrmData \n");
   if ( writeStrmData(hFile) !=TRUE)   printf("Error in writeStrmData \n");
}

//
// Code segment showing how to setup and
// write to a stream assigned to a player instance
//

#define   BUFF_SIZE 4032

BOOL writeStrmData(HANDLE hFile)
{
   ULONG              boardNum=0;
   int                timeSlot=1;
   HANDLE             hMpath, hStrm;
   CHAR               DeviceName[MNTI_MAX_DEVICE_NAME_SIZE];
   ULONG              DeviceNameSize;
   DWORD              DeviceStatus;
   DWORD              ErrorCode;
   UCHAR              OutBuffer[BUFF_SIZE];
   int                instance;
   DWORD              bytesRead,bytesXfer;
   QCompDesc          primaryPort;
   QCompAttr          attrs[4];
   QCompDesc          theCluster;
   QCompDesc          thePlayer;
   ULONG              streamID = 0, strmSize = MNT_STREAMSIZE_NORMAL;
   ULONG              rmType;
   QMsgRef            rmPtr;
   int                done,cnt;
   Player_MsgStart_t  playerStart;


   // Find an Mpath device name
   // we'll use this device to send/receive messages
   //

   if (mntEnumMpathDevice(MNT_FIRST_AVAILABLE, DeviceName,
                          &DeviceNameSize, &DeviceStatus) == FALSE)
   {
       // Call GetLastError to get the error code
       ErrorCode = GetLastError();
       // perform error handling
       return(FALSE);
   }

   // Use Win32 API function CreateFile to associate a native handle
   // to Mpath device

   if ((hMpath = CreateFile(DeviceName,
                   GENERIC_WRITE,
                   FILE_SHARE_WRITE,
                   NULL,
                   OPEN_EXISTING,
```

```
                FILE_FLAG_OVERLAPPED,
                NULL))== INVALID_HANDLE_VALUE)
{
    // Call GetLastError to get the error code
     ErrorCode = GetLastError();
     // perform error handling
     return(FALSE);
}

// Find the TSC (front end) component

primaryPort.board     = (UCHAR) boardNum;
primaryPort.processor = QCOMP_P_CP;
primaryPort.component = QCOMP_C_NIL;
primaryPort.instance = timeSlot;

attrs[0].key   = Std_ComponentType;
attrs[0].value = TSC_Std_ComponentType;
attrs[1].key   = QATTR_NULL;
attrs[1].value = 0xfb;

if (mntCompFind(hMpath, mntTransGen(), &primaryPort, attrs,
             DEF_TIMEOUT,NULL, NULL ) == FALSE)
{
    printf( "mntCompFind failed %d", GetLastError() );
    // perform error handling
    return(FALSE);
}

// Get the cluster associated with our TSC component

if (mntClusterByComp(hMpath, mntTransGen(), primaryPort,
                  &theCluster,
                  DEF_TIMEOUT, NULL, NULL ) == FALSE)
{
    // perform error handling
    printf( "mntClusterByComp failed %d", GetLastError() );
    return(FALSE);
}

// Find the player allocated to the cluster

thePlayer.board     = (UCHAR) boardNum;
thePlayer.processor = QCOMP_P_CP;
thePlayer.component = QCOMP_C_NIL;
thePlayer.instance  = QCOMP_I_NIL;

attrs[0].key   = Std_ComponentType;
attrs[0].value = Player_Std_ComponentType;
attrs[1].key   = QATTR_NULL;
attrs[1].value = 0xfb;

if ( mntClusterCompByAttr(hMpath,mntTransGen( ),theCluster,
                           attrs,&thePlayer,
                          DEF_TIMEOUT,NULL,NULL)== FALSE)
{
   // perform error handling
   printf( "mntClusterCompByAttr failed %x", GetLastError() );
   return(FALSE);
}

//
```

```
// Get first available Stream device
// we'll use this for writing stream data
//

if (mntEnumStrmDevice(MNT_FIRST_AVAILABLE, DeviceName,
                      &DeviceNameSize,
                      &DeviceStatus) == FALSE)
{
    // Call GetLastError to get the error code
    ErrorCode = GetLastError();
    // perform error handling
    return(FALSE);
}

// Open Stream device handle

if ((hStrm = CreateFile(DeviceName, GENERIC_WRITE,FILE_SHARE_WRITE,
                        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                        NULL)) == INVALID_HANDLE_VALUE)
{
    // Call GetLastError to get the error code
    ErrorCode = GetLastError();
    // perform error handling
    printf("Error %d in CreateFile for stream \n",ErrorCode);
    return(FALSE);
}

// Attach (open) a stream to our Stream device

if ((mntAttachMercStream(hStrm,
                    boardNum,              // BoardNumber
               MNT_STREAM_FLAG_WRITE, // ModeFlags
                &streamID,              // MercStreamID,
                &strmSize,             // StreamSize,
                DEF_TIMEOUT,
                NULL)) == FALSE)
{
    // Call GetLastError to get the error code
    ErrorCode = GetLastError();
    // perform error handling
    printf("%d: mntAttachMercStream failed: error = %d, strm# %ld\n",
                instance, ErrorCode, streamID);
    return(FALSE);
}

// At this point, the streamID is set to the returned value
// by the driver and we need to notify the player to read
// from the stream

playerStart.StreamID  = streamID;
playerStart.Decoding  = MULAW64D;
playerStart.StartMode = Player_MsgStart_StartMode_NORMAL;
playerStart.StartPos  = 0;

if (mntSendMessageWait(hMpath, Player_MsgStart, NORMAL_MSG,
                   sizeof(Player_MsgStart_t),
                   &playerStart, 1, &thePlayer, NULL, NULL) == FALSE)
{
   ErrorCode = GetLastError();
   printf("Error %d in mntSendMessageWait \n",ErrorCode);
   return(FALSE);
}
```

```
    //
    // Now we can begin writing to the stream in a loop by:
    // reading from a disk file then
    // writing it out to the stream
    //

    done=0;
    cnt=0;
    while (done==0)
    {
        // Read data from disk

        if (ReadFile(hFile, (LPVOID)OutBuffer, BUFF_SIZE,
                      &bytesRead, NULL) == FALSE)
        {
            // perform error handling
            printf("ReadFile on file failed %d\n", GetLastError());
            return(FALSE);
        }

        // Check if last block

         if ( bytesRead < BUFF_SIZE )
         {
            // If last, set EOS flag to signify end of play
            STRM_HDR StrmHdr;
            ZeroMemory(&StrmHdr, sizeof(StrmHdr));
            StrmHdr.bufFlags = MNT_EOS;
            mntSetStreamHeader(hStrm, &StrmHdr, 0);
            done=1;
         }

         printf("Writing %d block to stream \n",cnt++);

         // Write data to stream

         if (WriteFile(hStrm, (LPVOID)OutBuffer, bytesRead,
                        &bytesXfer, NULL) == FALSE)
         {
              // perform error handling
              printf( "WriteFile failed %d\n", GetLastError() );
              return(FALSE);
         }
    }

    // Wait for play stopped message from the player

    if (mntSendMessageWait(hMpath, Player_MsgStopped,EMPTY_MSG,0,
                    NULL, 1, &thePlayer, &rmType, &rmPtr) == FALSE)
    {
         // perform error handling
         ErrorCode = GetLastError();
         printf("%d: Unable to send stop-play message. Last error = (%x)h\n",
                                     instance, ErrorCode);
         return(FALSE);
    }

    return TRUE;
}
```

7.1.2. Flow Control

Programming write streams is much simpler than programming read streams:

- First, after preparing the buffer, you can call the **WriteFile( )** function; not much can go wrong except for a possible time out.
- Second, as long as there is room in the Shared RAM, the Driver simply pumps data out as soon and as fast as possible. The Class Driver monitors the flow control messages and properly paces the output so as to not overwhelm the DM3 embedded system.

This flow control applies only in the host-to-board direction. The DM3 embedded system regulates the flow through the Can_Take messages that are specific to the streams.

The stream flow control is generally transparent to the Direct Interface programmer. However, you must **never** in any single call, write more data than the stream size without first notifying the receiving instance to begin consuming the data. For example, in writing to a 16-kb stream assigned to a player instance, you must not write more than 16 kb before sending the start-play message.

7.1.3. Setting Stream Flags

You can use stream flags to convey application-specific meanings to counterpart components that must understand and comply with the protocol.

Before issuing write requests, you need to set the stream flags by calling the **mntSetStreamHeader( )** function. The driver transmits the latest stream flag settings along with the data blocks. Therefore, before you alter the stream header flags, you need to make sure that all preceding writes have completed successfully.

Here's an example showing how to play several files that have different decoding formats:

1. Use the MNT_EOT flag to indicate the separate format demarcations in the stream. For a Dialogic standard Player component, the MNT_EOT flag must accompany the last data block; otherwise, it is discarded and an error message is returned.

2. After the last block has been sent, reset the MNT_EOT flag, then send the next set of data blocks.
3. After all files have completed playing, set the MNT_EOS flag in the header and make the last write request. Of course, if appropriate, you can also call the **mntDetachMercStream( )** function to close the stream.

For a list of Read/Write errors, see the *DM3 Direct Interface Function Reference for Windows NT*.

7.1.4. Canceling Stream Writes

To cancel a specific stream write that is in progress, calling the following Win32 function will cancel all I/O for that handle:

- **CancelIo( )**

If the stream write is canceled, your application will be notified via an I/O Completion Port with an error code of ERROR_IO_ABORTED.

7.2. Reading Stream Data

Reading stream data means the host application is receiving data from the DM3 embedded system. For sample code, see *7.2.1. Example: Reading Stream Data*.

To program read streams, use the following procedure:

1. Find an available stream device, then obtain its stream device name by calling the following function:
   - **mntEnumStrmDevice( )**
2. Obtain the stream device handle by passing the stream device name to the following Win32 function:
   - **CreateFile( )**

   Pass the GENERIC_READ, FILE_SHARE_READ, and FILE_FLAG_OVERLAPPED flags in this function.
3. Attach a stream ID to the specified Stream device by calling the following Direct Interface function:

- **mntAttachMercStream( )**

  Pass the **hDevice**, **nBoardNumber**, **nModeFlags**, **lpMercStreamID**, **nStreamSize**, and **nTimeout** parameters to the function. The mode flag should specify MNT_STREAM_FLAG_READ.

4. Set up the memory location to store the read data and set up an OVERLAPPED structure.
5. Specify the ReadCompletionMask. Pass these parameters to the following function:

   - **mntSetStreamHeader( )**

6. Call the Win32 function **ReadFile( )**. Then, set up the following condition:

   - If it returns FALSE, call the **GetLastError( )** function.
   - If its **LastError** parameter contains ERROR_IO_PENDING, call the **WaitForSingleObject( )** and **GetOverlappedResult( )** functions to get the results.

7. If the actual bytes read is equal to the bytes requested, repeat step 6 to post another read.

   NOTE: If the actual bytes read is ***not*** equal to the bytes requested, retrieve the stream header by calling the **mntGetStreamHeader( )** function. Check if either the requested completion flag is set, such as MNT_EOD; or if the stream has been closed by the sending component, indicated by the STREAM_CLOSED **sysFlags** of the header. If so, break out of the read loop. Otherwise, an error has occurred, and you must analyze it.

8. If necessary, call the **mntDetachMercStream( )** function to close the stream device.
9. Use **CloseHandle( )** per Win32 conventions.

7.2.1. Example: Reading Stream Data

The following code segment shows the typical steps receiving data from a stream. Please note that the example shows a synchronous operation. These steps include:

1. Enumerating a stream device name.
2. Opening the device file and obtaining the Win32 handle.
3. Getting the destination address of a Recorder instance.
4. Attaching a stream to the Stream device.
5. Reading data from the stream.

```
// Example: Reading Stream Data

#include <stdio.h>
#include <windows.h>
#include <qhostlib.h>

#include <tscdefs.h>
#include <recdefs.h>
#include <coders.h>

#define DEF_TIMEOUT  60

BOOL readStrmData(HANDLE hFile);

void main( )
{
     HANDLE hFile;

    /* Open data file */

    if ( (hFile = CreateFile( "readTest.dat",
                              GENERIC_WRITE,
                              0,
                              NULL,
                              CREATE_ALWAYS,
                              FILE_ATTRIBUTE_NORMAL,
                              NULL
                              )) == INVALID_HANDLE_VALUE )
    {
        printf("Can't get a handle to readTest.dat \n");
        exit(0);
    }

    // Open data file

    SetFilePointer( hFile, 0, NULL, FILE_BEGIN);

     // Call routine "readStrmData" to get data from recorder

     if ( readStrmData(hFile)!=TRUE)  printf("Error in readStrmData \n");

}

// Code segment showing how to Setup and read from a stream
```

```
#define   BUFF_SIZE 4032

BOOL readStrmData(HANDLE hFile)
{
   CHAR                 DeviceName[MNTI_MAX_DEVICE_NAME_SIZE];
   ULONG                DeviceNameSize;
   DWORD                DeviceStatus;
   HANDLE               hMpath;
   DWORD                ErrorCode;
   ULONG                boardNum=0;
   UCHAR                InBuffer[BUFF_SIZE];
   int                  instance;
   QCompDesc            primaryPort;
   QCompAttr            attrs[4];
   QCompDesc            theCluster;
   QCompDesc            theRecorder;
   int                  timeSlot = 1;
   ULONG                done = 0;
   HANDLE               hStrm;
   ULONG                streamID = 0;
   ULONG                strmSize = MNT_STREAMSIZE_NORMAL;
   STRM_HDR             header;     /* Retrieved stream header */
   ULONG                duration = 6; // record duration in sec
   Recorder_MsgStart_t  recorderStart;
   Std_MsgSetParm_t     setParm;
   ULONG                rmType;
   QMsgRef              rmPtr;
   DWORD                bytesRead;

   // Find an Mpath device name
   // we'll use this device to send/receive messages
   //

   if (mntEnumMpathDevice(MNT_FIRST_AVAILABLE,
                          DeviceName, &DeviceNameSize,
                          &DeviceStatus) == FALSE)
   {
       // Call GetLastError to get the error code
        ErrorCode = GetLastError();
        // perform error handling
        printf("Error %d in mntEnumMpathDevice \n");
        return(FALSE);
   }

   if ((hMpath = CreateFile(DeviceName,
                   GENERIC_READ | GENERIC_WRITE,
                   FILE_SHARE_READ | FILE_SHARE_WRITE,
                   NULL,
                   OPEN_EXISTING,
                   FILE_FLAG_OVERLAPPED,
                   NULL))== INVALID_HANDLE_VALUE)
   {
        // Call GetLastError to get the error code
        ErrorCode = GetLastError();
        // perform error handling
        printf("Error %d in CreateFile for mpath \n",ErrorCode);
        return(FALSE);
   }

   // Find the TSC (front end) component
```

```
primaryPort.board     = (UCHAR) boardNum;
primaryPort.processor = QCOMP_P_CP;
primaryPort.component = QCOMP_C_NIL;
primaryPort.instance  = timeSlot;

attrs[0].key   = Std_ComponentType;
attrs[0].value = TSC_Std_ComponentType;
attrs[1].key   = QATTR_NULL;
attrs[1].value = 0xfb;

if (mntCompFind(hMpath, mntTransGen(), &primaryPort, attrs,
                        DEF_TIMEOUT, NULL, NULL ) == FALSE)
{
   // perform error handling
   printf( "mntCompFind failed %d", GetLastError() );
   return(FALSE);
}

// Get the cluster associated with our TSC component

if (mntClusterByComp(hMpath, mntTransGen(), primaryPort, &theCluster,
                        DEF_TIMEOUT, NULL, NULL ) == FALSE)
{
   // perform error handling
   printf( "mntClusterByComp failed %d", GetLastError() );
   return(FALSE);
}

// Find the allocated recorder to the cluster

theRecorder.board     = (UCHAR) boardNum;
theRecorder.processor = QCOMP_P_CP;
theRecorder.component = QCOMP_C_NIL;
theRecorder.instance  = QCOMP_I_NIL;

attrs[0].key   = Std_ComponentType;
attrs[0].value = Recorder_Std_ComponentType;
attrs[1].key   = QATTR_NULL;
attrs[1].value = 0xfb;

if ( mntClusterCompByAttr(hMpath,mntTransGen( ),theCluster,
                          attrs,&theRecorder,
                          DEF_TIMEOUT,NULL,NULL)== FALSE)
{
   // perform error handling
   printf( "mntClusterCompByAttr failed %x", GetLastError() );
   return(FALSE);
}

// Get first available Stream device
// we'll use this for writing stream data
//

if (mntEnumStrmDevice(MNT_FIRST_AVAILABLE, DeviceName,
                        &DeviceNameSize, &DeviceStatus) == FALSE)
{
     // Call GetLastError to get the error code
     ErrorCode = GetLastError();
     // perform error handling
     printf("Error %d in mntEnumStrmDevice \n",ErrorCode);
     return(FALSE);
}
```

```
// Open Stream device handle

if ((hStrm = CreateFile(DeviceName, GENERIC_READ,
                        FILE_SHARE_READ, NULL,
                      OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                        NULL)) == INVALID_HANDLE_VALUE)
{
     // Call GetLastError to get the error code
     ErrorCode = GetLastError();
     // perform error handling
     printf("Error %d in Create File for stream device \n",ErrorCode);
     return(FALSE);
}

// Attach (open) a stream to our Stream device

if ((mntAttachMercStream(hStrm,
                  boardNum,                  // BoardNumber
                MNT_STREAM_FLAG_READ,    // ModeFlags
                &streamID,               // MercStreamID,
                &strmSize,               // StreamSize,
                DEF_TIMEOUT,
                NULL)) == FALSE)
{
     // Call GetLastError to get the error code
     ErrorCode = GetLastError();
     // perform error handling
     printf("Error %d in mntAttachMercStream \n",ErrorCode);
     return(FALSE);
}

// At this point, the streamID is set to the returned value
// by the driver and we need to notify the Recorder to record
// for so many seconds
//

setParm.Num = Recorder_ParmDuration;
setParm.Val = 1000*duration;

// Send the message to the board

if (mntSendMessageWait(hMpath, Std_MsgSetParm, NORMAL_MSG,
                  sizeof(Std_MsgSetParm_t),
                 &setParm, 1, &theRecorder, NULL, NULL) == FALSE)
{
    ErrorCode = GetLastError();
    // perform error handling
    printf("%d: Unable to send set-parm message. Last error = %d\n",
                           instance, ErrorCode);
    return(FALSE);
}

// Set up the start-record message and
// send it to the recorder
//

recorderStart.StreamID  = streamID;
recorderStart.Encoding  = OKI32E;
recorderStart.StartMode = Recorder_MsgStart_StartMode_TIMED;

if (mntSendMessageWait(hMpath, Recorder_MsgStart, NORMAL_MSG,
```

```
                        sizeof(Recorder_MsgStart_t),&recorderStart,
                        1, &theRecorder, &rmType, &rmPtr) == FALSE)
    {
        ErrorCode = GetLastError();
        // Perform error handling
        printf("\n%d: Unable to send start-record message. Last error = %d",
                    instance, ErrorCode);
        if (ErrorCode == ERROR_MNT_MERCURY_STD_MSG)
        printf("\n%d: Merc errs: %x, %x\n",
                        instance, rmPtr->type, rmPtr->msgsize);
        return(FALSE);
    }

    //
    // now we're ready to read the record stream
    // clearing all read-completion mask allows EOS to complete the read
    //
    mntSetStreamHeader(hStrm, NULL, 0);

    done=0;
    while ( done==0)
    {
       // Read data from driver
        if (ReadFile(hStrm, (LPVOID)InBuffer,
                        BUFF_SIZE, &bytesRead, NULL) == FALSE)
        {
           // perform error handling
           printf("%d: ReadFile failed %d; read=%d\n", instance,
                                        GetLastError(), bytesRead );
           return(FALSE);
        }
        else
            printf("Read %d bytes from DM3 stream \n",bytesRead);
        if (bytesRead != BUFF_SIZE)
        {
           mntGetStreamHeader(hStrm, &header);
           if (header.bufFlags & MNT_EOD
                   || header.sysFlags & STREAM_CLOSED)  done = 1;
        }

       // Write the buffer just read to a disk file

        if (WriteFile(hFile, (LPVOID)InBuffer,
                        BUFF_SIZE, &bytesRead, NULL) == FALSE)
        {
            printf( "WriteFile failed %d\n", GetLastError() );
            exit(0);
        }
     }
    // Wait for record complete
    if (mntSendMessageWait(hMpath, Recorder_MsgStopped,EMPTY_MSG,0,
                    NULL, 1, &theRecorder, &rmType, &rmPtr) == FALSE)
    {
        ErrorCode = GetLastError();
        // perform error handling
        printf("Unable to receive stopped message. Last error = %d\n",
                                                                ErrorCode);
        return(FALSE);
    }

    return TRUE;
}
```

7.2.2. Protocol Driver Buffering

When the Protocol Driver reads the incoming data blocks and attempts to find a pending read request, it might not find any, especially under a heavy system load. In this case, the Protocol Driver buffers the blocks into an orphan buffer until a request is made. To avoid this costly extra copying, make sure that you post read requests promptly. If non-paged system buffers are available, the Protocol Driver can buffer all overflows and service the subsequent read requests. However, under extreme conditions (or if the application simply goes away), the orphan buffer can fill up and be in an overrun condition (STATUS_DATA_OVERRUN or ERROR_IO_DEVICE). The stream header's **sysFlags** field is set to STREAM_OVERRUN.

NOTE: The orphan buffer can contain residue from a previous I/O operation. If you want to ensure there's a "clean pipe" before you start to read from a stream, you need to first read and discard the residue in the orphan buffer. The **mntCheckStreamOrphans( )** host library function returns the number of orphaned bytes, if any. When a particular stream is opened in write mode, the Protocol Driver automatically frees up orphans for a previous read stream.

The Protocol Driver buffers overflow messages also, but while there is one buffer for each stream (that is, multiple buffers per board), there is only one orphan message buffer per board. As long as requests for a message read make it down to the Protocol Driver in time, no messages will be lost. You should always attempt to avoid creating orphans.

7.2.3. Specifying Read Buffer Sizes

When you call the **ReadFile( )** function, you must specify a buffer and its size. Determining the optimum size of the buffer can be challenging.

Most developers who perform intensive I/O operations find an optimum buffer size through experimenting with different sizes. It helps if you understand how the underlying driver moves the data. Typical buffer sizes tend to be 16 Kb, 32 Kb, or 64 Kb. However, buffer size depends on device capability. For example, you would probably not specify a 64-Kb buffer size for a dial-up PPP connection, although 64 Kb might be fine for an ATM or NIC card.

NOTE: When you make read calls, Windows NT locks down your provided buffers in the working set of your process. Therefore, there may be a practical limit to the number of asynchronous I/O operations that you can post.

7.2.4. Canceling Stream Reads

To cancel a specific stream read that is in progress, call the following Win32 function:

- **CancelIo( )**

If the read is canceled, your application will be notified via an I/O Completion Port with an error code of ERROR_IO_ABORTED.

8. Using Clusters

NOTE: WITH CURRENT QUADSPAN AND IPLINK RELEASES, CLUSTERS ARE PRE-BUILT BY DIALOGIC AND SHOULD NOT BE MODIFIED BY APPLICATION DEVELOPERS.

8.1. Host Application Cluster Control

This section deals with the mechanics of how a host application controls clusters, how it uses them to exchange TDM data on the SCbus, and how it directs component instances to exchange TDM data on the network front end. A host application using the DM3 embedded system controls clusters by performing the following:

1. Find a cluster.
2. Add component instances to a cluster.
3. Add SCbus resources with input or output ports to a cluster.
4. Assign SCbus timeslots to SCbus resources.
5. Remove SCbus resources from a cluster.
6. Maintain Talker protocol for SCbus output ports.

Advanced Tasks:

7. Change default cluster connections.
8. Connect clusters on the same board together.

Table 3 gives a summary of the host library functions used to accomplish each task.

Table 3. Host Cluster Control Tasks

To Do This…	**Use Host Library Function(s)…**
Find a cluster	**mntClusterFind( )** **mntClusterByComp( )**
Add components to the cluster*	**mntCompAllocate( )**
Add SCbus resources with input and output ports to clusters*	**mntCompAllocate( )**
Assign SCbus timeslots to SCbus resources	**mntClusterTSAssign( )**
Remove SCbus resources from cluster*	**mntCompFree( )**
Manage Talker protocol for SCbus output ports	**Full Talker Functions:** **mntClusterActivate( )** **mntClusterDeactivate( )**
For Advanced Tasks…	**Use Host Library Function(s)…**
Change default cluster configuration*	**mntClusterDisconnect( )** **mntClusterConnect( )**
Connect cluster ports in different clusters on the same board *	**mntClusterConnect( )**

* This task is not applicable for this release.

8.1.1. Finding a Cluster

Existing clusters may have component instances added to them. If a TSP component exists on the board, most applications will allocate components into a TSP's pre-existing cluster. To add instances to an existing cluster, first find a cluster with the necessary attributes.

To find clusters, two functions are used:

mntCompFind( ) Finds a component with the specified set of attributes

mntClusterByComp( ) Finds the cluster that a specified component belongs to

For example, to find the TSP cluster controlling T-1 timeslot 6, use the **mntCompFind( )** function specifying a TSP component with timeslot 6. After the component is found, retrieve the TSP's cluster by calling the **mntClusterByComp( )** function using the found component address.

8.1.2. Adding Components to Clusters

DM3 component instances are added to clusters during component allocation. When a host application allocates a component with the **mntCompAllocate( )** function, it must specify the cluster to which it belongs.

When component instances are added to a cluster, a set of default connections are automatically established. The kernel maps central ports to valid non-central ports, that is, IN-ports are connected to OUT-ports. A TSP component instance always has a pair of central ports, therefore, whenever a TSP component instance is in a cluster, it will be connected to any instances added to that cluster.

Occasionally, SCbus resources are configured as central ports, typically if the cluster does not contain a TSP instance.

Figure 8 is an example of the default connection map created when a cluster contains a TSP, player, recorder, tone generator, and signal detector. The TSP component instance is shaded to indicate that it is the central instance with two

central ports. For most applications, it is not necessary to configure clusters differently than the default configuration.

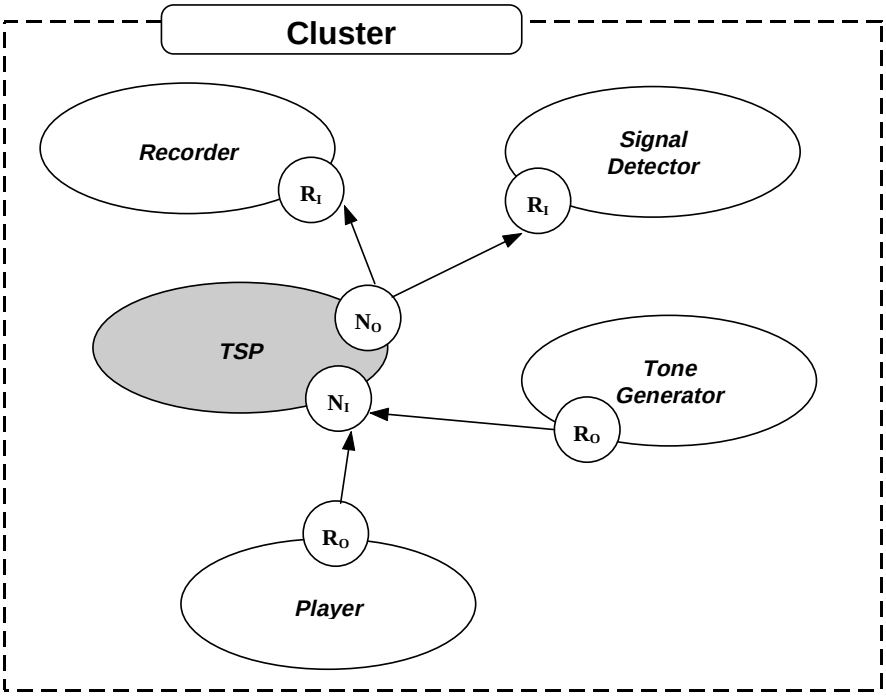

Figure 8. Default Cluster Connections Example

8.1.3. Assigning an SCbus Timeslot to an SCbus Resource

When an SCbus resource is created, it does not have a specific SCbus timeslot assigned to it. The SCbus IN-ports are used to transmit TDM data into a specific SCbus timeslot and the SCbus OUT-ports are used to receive data from a specific SCbus timeslot.

To assign a timeslot, an application uses the **mntClusterTSAssign( )** function call specifying the:

- cluster
- SCbus resource component address
- SCbus resource port identity
- SCbus timeslot number

NOTE: For this release, **mntClusterTSAssign( )** is only valid for SCbus OUT-ports.

To stop data from being transmitted over the SCbus, **mntClusterTSUnassign( )** can be called to clear any timeslot assignments from the SCbus port.

8.1.4. Talker Protocol

When an SCbus resource with an $\mathbf{S_O}$ port is part of a cluster, DM3 talker protocol must be followed. The host application has several choices:

- Follow full DM3 talker protocol.
- Follow a simple DM3 talker protocol.
- Provide the address of a component that follows full DM3 Talker protocol.

Simple Talker Protocol

Simple Talker protocol provides the means for a host application to add SCbus resources that "talk" (transmit TDM data) with minimal application talker protocol overhead. This is accomplished by using the **mntClusterActivate( )** and **mntClusterDeactivate( )** host library function calls.

The **mntClusterActivate( )** call is used to activate the connections from the SCbus OUT-port to the IN-ports inside the cluster. For example, in the figure below, when the SCbus OUT-port is made active by the host application, the network IN-port is accepting TDM data from the SCbus and the Tone generator and the Player connections are not active.

Once the connection is active, what happens when the Player wants to generate TDM data? This is dependent on how **mntClusterActivate( )** was used.

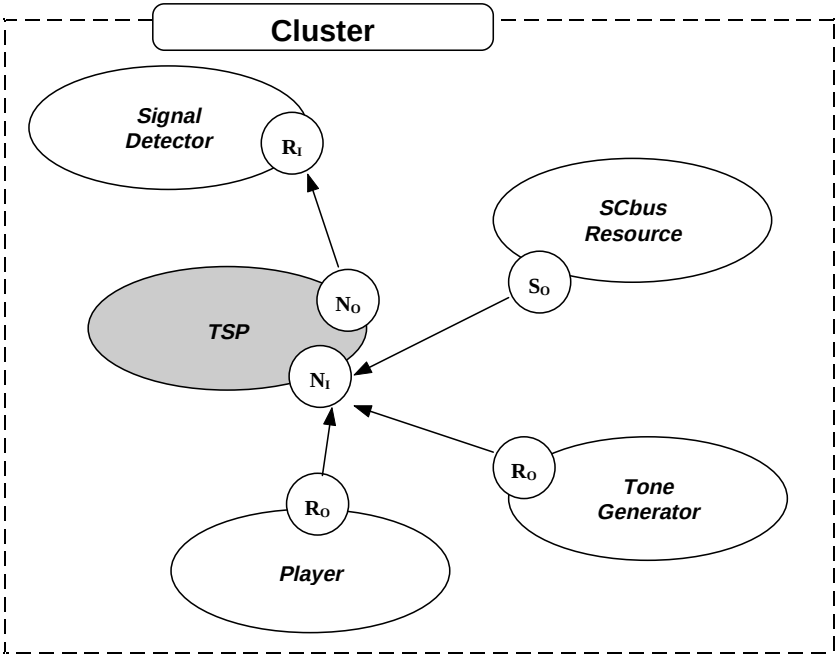

Figure 9. SCbus Resource Talking

When the function is called, one of two *default talker protocol response* options is supplied. The default response informs the kernel how to handle the situation. Valid options are defined in *mercdefs.h*. They are:

QCLUST_AutoReject	Data from the SCbus cannot be interrupted by any other OUT-port resource in the cluster. The connection between the $\mathbf{S_O}$ port and all the input ports it connects to will remain active until the host application explicitly deactivates the connection with a **mntClusterDeactivate( )** function call.
QCLUST_AutoAccept	Data from the SCbus can be interrupted by any other OUT-port resource in the cluster. The connection between the $\mathbf{S_O}$ port and cluster input ports can be temporarily suspended and re-established after the interrupting resource has finished. No notification will take place if this occurs.

Full Talker Protocol

A host application can act as a proxy for a resource that outputs data on the SCbus and transmits data into the $\mathbf{S_O}$ port of a cluster.

A resource that outputs data must be able to send a set of commands to request to talk, and must be able to reply to kernel requests for interruptions with a specific set of messages. These messages are summarized below:

Message Name	Action	Response Message
QClusterSuspend	kernel request to stop output of data	*QClusterSuspendResult*
QClusterResume	kernel request to resume output of data	*QClusterResumeResult*
QClusterActivate	component request to output data	*QClusterActivateComplete*
QClusterDeactivate	component informing kernel that it has stopped outputting data	*QClusterDectivateComplete*

8.1.5. Changing the Default Cluster Configuration

NOTE: This functionality is not implemented for this release.

This is considered an advanced task since the default cluster configuration should handle most situations.

Reconfiguring a cluster means that the host application will connect ports inside a cluster to each other in a configuration that is different than the default behavior. The figure below is a default cluster connection map that results when a TSP is in a cluster with a player instance and an SCbus resource.

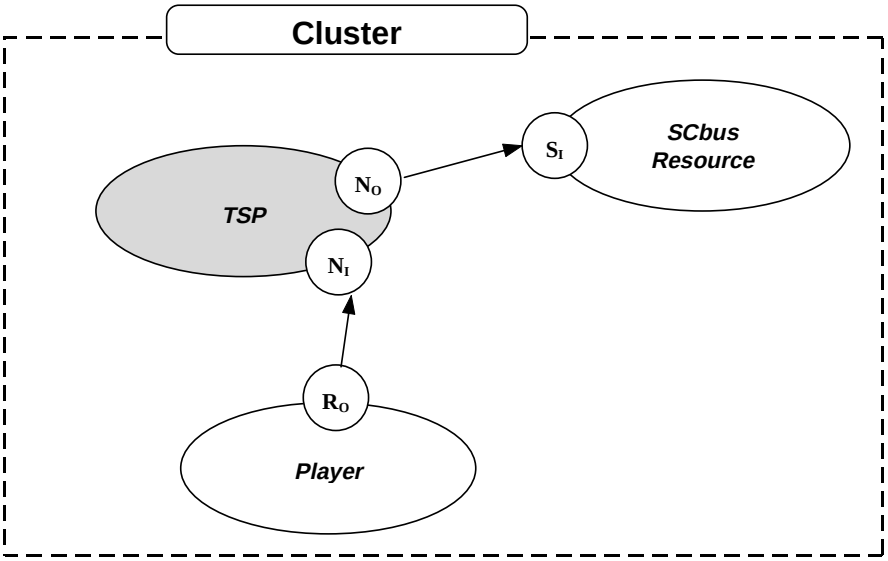

Figure 10. Default Cluster Connections Example

It may be desirable to configure the player resource to output to the $\mathbf{S_I}$ port as well as the $\mathbf{N_I}$ port temporarily in a drop and insert situation. To establish a connection between the $\mathbf{R_O}$ and $\mathbf{S_I}$ ports, call **mntClusterConnect( )** specifying the cluster, $\mathbf{S_I}$ port, and $\mathbf{R_O}$ port. This results in a new connection map as shown in Figure 11.

To return to the original connection map, call **mntClusterDisconnect( )** specifying the cluster, $\mathbf{S_I}$ port and $\mathbf{R_O}$ port.

8.1.6. Finding Cluster Assignment

To find the cluster that a component instance is part of, call:

mntClusterbyComp( ) Given an instance descriptor, finds the cluster to which it is allocated.

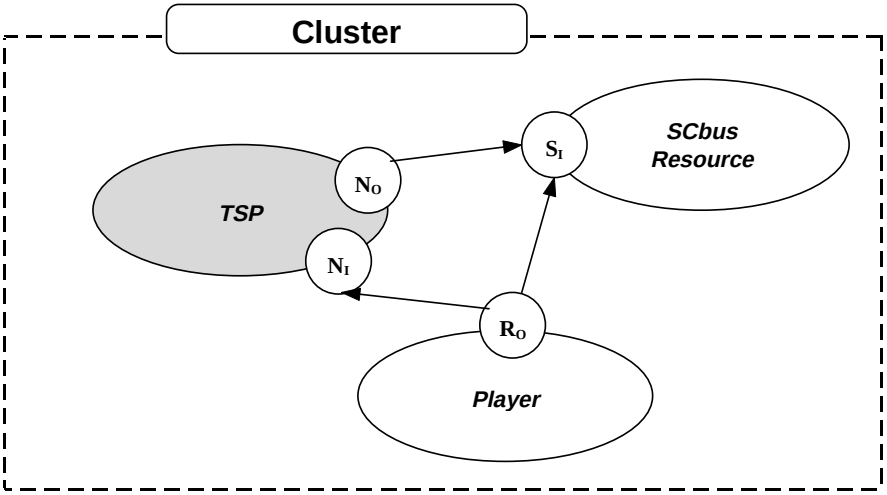

Figure 11. Reconfigured Cluster

8.1.7. Connecting Ports on the Same Board

Clusters on the same board can sometimes be connected without using SCbus timeslots. To connect two clusters together, use the **mntClusterConnect( )** function. If the cluster ports specified can be connected without SCbus timeslots, the function will succeed. If they cannot, the connection will fail.

9. Exit Notification

Direct Interface functionality called *exit notification* allows you to be notified when a host application or specific component instance terminates. There are two types of exit notification and each must be explicitly enabled in your application.

- **Board-level exit notification**
  If a component instance exits and/or fails, the host application is notified by the DM3 Kernel via a *QFailureNotify* message.
- **Application exit notification**
  If a host application exits and/or fails, the driver notifies the DM3 Kernel via a *QExitNotify* message when the Mpath device is closed. The driver does not distinguish an abnormal termination from a normal exit; it blindly exercises the exit notification logic upon the last close of the Mpath device.

Due to its impact on performance, the exit notification feature is typically used only during application development. Production applications should be designed to exit gracefully by closing opened streams and releasing allocated instances.

9.1. Setting up Board-level Exit Notification

To enable board-level exit notification, an application issues an **mntNotifyRegister( )** call. The DM3 Kernel records the sender's address. If a component fails, the application is notified via a *QFailureNotify* message. Note that the application enabling notification must queue a request to receive this message. To disable component exit notification, the application issues an **mntNotifyUnregister( )** call on the same source address (Mpath).

9.2. Setting up Application Exit Notification

There are two steps needed to set up application exit notification to the embedded system. First, the host application must call the **mntSetExitNotify( )** function on the Mpath to enable the sending of the *QExitNotify* exit notification message. When enabled, whether the handle is closed implicitly (via crash or failure) or explicitly (via **CloseHandle( )**), a *QExitNotify* message is sent for that Mpath, that is, for that application/Mpath's source address. Note, if the application exits

gracefully, to avoid undue overhead it should disable the notification using the **mntSetExitNotify( )** function.

Next the application must register with the DM3 Kernel the fact that it is "using" certain component instances by calling the **mntCompUse( )** function. When the handle associated with the source address is closed and a *QExitNotify* message is sent, the DM3 Kernel will send the *QExitNotify* to each component instance identified as "used." However, if a particular component instance is also being "used" by another application, the kernel only removes that address from the used list and does not send *QExitNotify*. At that point, the component instance will know to stop any active operations and clean up any resources. Similarly, the **mntCompAllocate( )** function causes a sort of implicit "use" as a side effect where the calling application address is added to the user list.

Call the **mntCompUnuse( )** function to disable exit notification by the DM3 Kernel to the component instances listed in the payload, that is, to remove the caller from the "used" list.

Application exit notification applies to clusters as well. When an application calls **mntClusterCreate( )** or **mntClusterAllocate( ),** the cluster is considered "used" by the application. If the DM3 Kernel receives a *QExitNotify* message for that source address, then one of two things may happen. If the cluster was locked by a call to **mntClusterConfigLock( )**, then the cluster is freed, assuming no other application are "using" it. If the cluster was not locked and no more applications are "using" it, then the *QExitNotify* message is forwarded to all component instances within the cluster. After all the component instances have responded, then the cluster is freed and destroyed.

10. Error Handling

10.1. Retrieving Errors from the Host

Use the Win32 API function call **GetLastError( )** to retrieve error information from the host machine. See *Section 4.1.3. Handling Asynchronous Function Returns.*

10.2. Retrieving Error Codes from the Embedded System

You will retrieve errors much differently depending on whether you called the function synchronously or asynchronously.

10.2.1. Synchronous Platform Function Calls

If a synchronous function returns TRUE, it has completed successfully; no further action is necessary. If it returns FALSE, it has failed; you must call the **GetLastError( )** function to get the error code. See *Section 4.2.1. Handling Synchronous Function Returns*.

10.2.2. Asynchronous Platform Function Calls

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, 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 must 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.

11. Direct Interface Application Guidelines

Here's a list of things to remember when using the Direct Interface.

11.1. Design & Development

Here's a few general guidelines for when you're designing and developing your application:

- Your program must be robust enough to clean up after itself. Specifically, any allocated component instances should be freed and any open streams should be closed.
- Use the WinNT Structured Exception Handling (SEH) feature to run your cleanup code and cause the system to notify your application when certain situations occur.
- Use the WinNT **SetConsoleCtrlHandler( )** function to catch CTRL-C and BREAK entries from the keyboard.

11.2. Performance Issues

The following list shows how to avoid certain performance issues:

- Avoid orphans messages and streams of any kind (that is, data copies). Use asynchronous I/O to avoid this situation.
- Use the MercMon and PerfMon tools to watch I/O traffic as well as to get information about orphan messages, streams, and timeouts.
- Avoid page faults by using Pview and PerfMon to understand the details of the activity on the board.

11.2.1. Pending I/O Requests

To avoid orphan messages or streams, you will typically need to post more than one request. The number of requests you post depends on your application and the load you expect it to handle under a particular system configuration.

For the unsolicited messages that you need to field throughout the life of the application (for example, alarms or errors), you should allocate permanent MMBs and use the **mntRegisterAsyncMessages( )** function call. During development and testing, use the **MercMon** utility (see *Section 14.4. MercMon*) to see if any orphan messages are created. Please note, however, that MercMon's counter of orphan messages shows the total cumulative number of orphaned messages, not the current orphan count.

If the orphan message volume does not drop back to zero, your application is not reading them. Use the **omdump** tool (see *Section 14.6. Omdump*) to examine the orphan messages and take appropriate action.

In terms of performance impact, remember that when the Protocol Driver receives a message, it has to search through all pending MMBs for a potential match. Therefore, it's best to optimize by posting just enough MMBs to avoid orphans. This applies to reading streams as well.

Simply because of the required buffering needed, orphan streams are likely to have greater impact on performance. Be sure to use MercMon to monitor orphan streams while running your application. If you see any, you should increase the number of posted reads. For convenience, use **mntRegisterAsyncStreams( )** to post multiple read buffers. Keep in mind that for each read posted, its buffer is mapped to system space and its physical pages are locked. Thus, there is a practical limit to how many reads you can post before you begin to impact other areas of the system.

12. Compiling and Linking an Application

How you compile and link is partially a function of your C or C++ environment. However, there a few things to keep in mind as you perform these functions:

- There are two versions of the Direct Interface library: one containing debug functions (that enables logging information), and another without debug functions.

Table 4. Filenames of Libraries

If you're using…	**and you want a library…**	**link this file:**
Microsoft Visual C++	with debug functionality	*mntid.lib*
	without debug functionality	*mnti.lib*
Borland C++	with debug functionality	*mntid_b.lib*
	without debug functionality	*mnti_b.lib*

13. Debugging

Because the Direct Interface program is a communications application that has strict requirements for performance and robustness, debugging might consist of not only logging API calls, and but also capturing and recording the actual I/O traffic. For this reason, the Direct Interface provides two debugging facilities, tracing and the Protocol Driver trace log.

During your development and testing phases, you should use the debug version of the Direct Interface (*mntid.lib*) that supports debug tracing. When you no longer need debug functionality, link in the non-debug version of the library (*mnti.lib*).

13.1. Tracing

When tracing is enabled, all Direct Interface functions output the tracing to a disk file that you can examine after the program has run. This can be especially helpful if you run several processes and/or threads. In the non-debug version of the Direct Interface, all tracing calls are null operations and present no overhead.

13.2. Protocol Driver Trace Log

The Protocol Driver provides logging services that capture both inbound and outbound messages and streams. Only the system administrator can enable Protocol Driver tracing. Enabled traced events are written to the *mpd_dbug.dat* file in the `%SystemRoot%\system32\drivers` directory. Because Protocol Driver tracing impacts system performance, you should use it only if absolutely necessary, and only during testing.

Caution

Enabling Protocol Driver tracing can add further confusion during debugging because it might subtly alter real time application behavior. Use this feature only as a last resort.

13.3. Cleaning Up after Exits and Crashes

One facet of writing a reliable and robust Direct Interface application program is to properly release DM3 resources each time the program terminates, whether normally or abnormally. Specifically, you must release all allocated component instances and close all open streams. Otherwise, you will eventually lose these critical resources and be unable to access the DM3 embedded system.

For the development phase of your application, the Direct Interface provides exit notification functionality (described in *9. Exit Notification*) to help you clean up at program termination. Once your application is headed into its production phase however, You should not use or depend on these functions.

For your production application, you should use Windows NT structured exception handling to clean up properly at program exit. For Windows NT console programs, you need to provide your own Ctrl+C and Ctrl+Break handlers that properly close all open streams through **mntDetachMercStream( )** function calls and free all component instances through **mntCompFree( )** function calls.

14. Tools and Utilities

This chapter contains instructions for a number of tools and utilities that are packaged with the Direct Interface. Use these tools for a variety of tasks during your development cycle:

- **dm3stderr**
  This utility acts like a virtual "tip" or serial port session to the DM3 board.
- **Mercmon**
  This tool logs and reports any issues raised from the DM3 embedded system.
- **Mpdtrace**
  This program can enable or disable driver debugging, and retrieve the driver debug buffer.
- **qerror**
  This utility displays a string associated with the error code returned in a *QResultError* message from the board.
- **Omdump**
  This utility can be used to dump Orphaned Messages to a file.
- **strmstat**
  This utility displays a stream’s current state.

14.1. dm3stderr

This utility acts like a virtual "tip" or serial port session to the DM3 board. All DM3 boards send “printfs” to both the serial port and to the host. This program polls the board and displays those “printfs” from the resources and kernel to the screen.

The `dm3stderr` utility takes the following parameters:

Parameter	**Meaning**
-b *board number*	The number of the board in the system. This is required.
-d *debug level*	Sets the debug level.
-f *filename*	The name of the file in which you want to capture the output.
-h	Displays a help screen.
-v	Displays the version number of the software.

14.1.1. Example

```
dm3stderr -b1 -f output.txt
```

This will grab the printf data from board 1 and display it to the screen and save it to file "output.txt".

14.2. qerror

This utility displays a string associated with the error code returned in a *QResultError* message from the board. This code is generated with a PERL script directly from the header files, so it is as accurate as the comments in the header files are.

The source to this code is arranged so that the "get error string" function can be pulled out and used in any application.

14.2.1. Usage

Enter qerror on the command line in the following manner:

```
qerror [option_list] ERRORCODE
```

The `qerror` utility takes the following options:

Parameter	Meaning
-b *number*	Base of the input number. Use `10` for decimal or `16` for hexadecimal. 16 is the default.
-d *debug level*	Set the debug level in a range from `0` to `5`. 0 is the default.
-h	Displays a help screen.
-v	Displays the version number of the software.

14.2.2. Example

```
qerror 28008
 ERROR==> 0x28008 (163848)
 Kernel Error:
   Cluster does not exist or cannot be found
```

14.3. kernelver

This utility will query the kernel on any processor for its version number. Use the `-l` option to "ping" the board if you just want to generate message traffic or as a good test to see if the kernel is running.

The `kernelver` command takes the following parameters:

Parameter	Meaning
-b *board number*	The number of the board in the system. This is required.
-d *debug level*	Set the debug level in a range from `0` to `5`. 0 is the default.
-p *processor number*	The number of the processor on the board. This is always required.
-l *loop*	Number of times to grab version.
-h	Displays a help screen.
-v	Displays the version number of the software.

14.3.1. Example

- `kernelver -b0 -p2`
  Gets the version of the kernel running on processor 2.

14.4. MercMon

Both the Class Driver and the Protocol Driver maintain various counters that can aid in monitoring system activities and interpreting certain behaviors. Since MercMon is a read-only application, permissions are granted to all users.

14.4.1. Usage

The `MercMon` utility takes the following parameters:

Parameter	**Meaning**
/t *milliseconds*	The timer period in milliseconds. Default is 1000.
/l *seconds*	The log period in seconds. Default is 600.

Table 5 lists the Class Driver counters. *Table 6* lists the Protocol Driver counters.

You can also use NT's PerfMon utility to monitor the DM3 boards. Select either "MCD Device" (for the Class Driver) or "MPD Device" (for the Protocol Driver) object and add specific counters of interest.

Table 5. Class Driver Counters

Class Driver Counter	**Description**
CanTakes	number of Can_Take messages received. The DM3 board can send Can_Take messages to control the flow from the host.
CloseStrmErrs	number of stream close request errors.
FailMpathFind	number of times that the Class Driver has failed to find an Mpath device.
FailStrmFind	number of times that the Class Driver has failed to find a Stream device.
OpenedStrms	number of currently opened streams.

Class Driver Counter	Description
OpenStrmErrs	number of stream open request errors.
Reads	number, in kilobytes, of read requests successfully completed.
ReadTimeouts	number of stream read requests that have timed out.
SendTimeouts	number of message write requests that have timed out.
SplitWrites	number of split writes that the Class Driver has made. For each large write request, the Class Driver splits it into multiple partial transfers.
StrmCloses	number of stream close requests received
StrmOpens	number of stream open requests received
TotalReads	number of stream read requests received
TotalSends	number of message send requests received
TotalWrites	number of stream write requests received
Writes	number, in kilobytes, of write requests successfully completed.
WriteTimeouts	number of stream write requests that have timed out.

Table 6. Protocol Driver Counters

Protocol Driver Counters	Description
AsyncMsgQ	number of asych message read requests received
AsyncMsgQDone	number of message read requests completed.
BadSramOffset	number of invalid values that the Protocol Driver has found in the control structures in the DM3-board-resident SRAM. These control structures consist of circular buffers and a chained list of free data blocks. **This count should always be 0**.

Protocol Driver Counters	Description
BigMsgsRcvd	number of big messages received from the DM3 board through data blocks. The Protocol Driver uses a data block for each message larger than 24 bytes.
BigMsgsSent	number of big messages sent
BogusInterrupts	number of times that the Protocol Driver received interrupts from the DM3 board while its status register contained 0.
DmaInterrupts	because the Protocol Driver does not support DMA, this value should always be 0.
DpcOverruns	number of times that the Protocol Driver could not queue its DPC processing in response to interrupts. If this counter is increasing persistently, it indicates an overloaded system. In other words, the system cannot keep up with its incoming interrupts. You need to make some hardware upgrades.
IrpsCanceled	number of canceled I/O requests.
MsgInQ	number of message read requests received
MsgInQDone	number of message read requests completed
MsgInSram	the cumulative total of messages read from the SRAM.
MsgOutQ	number of message send requests received
MsgOutQDone	number of message send requests completed
MsgOutSram	number of messages written to the SRAM.
MsgOverruns	number of times that the Protocol Driver could not buffer incoming message data. If this counter is increasing persistently, you need to increase the value in the orphanageMsgLen parameter.

Protocol Driver Counters	**Description**
MsgsInPerSramSession	number of messages that the Protocol Driver has read from the SRAM during any inbound session. This counter fluctuates between 0 and integer values that vary according to DM3-board-generated traffic.
MsgsOutPerSramSession	number of messages that the Protocol Driver has written to the SRAM during any outbound session. This counter fluctuates between 0 and integer values that vary according to application-generated traffic.
MsgTimeouts	number of message read and send requests that have timed out
NoInDataDpcIsr	number of times that the Protocol Driver received interrupts from the DM3 board and attempted to read the SRAM buffers, but there were nothing to be read. This can happen if the Protocol Driver's timer routine has already emptied the buffers. This timer routine runs at the interval specified in the sramOutTimer parameter.
OrphanStrms	number of times that the Protocol Driver had to buffer streams. This indicates that read requests are not getting to the Protocol Driver quickly enough. Unless there are also overruns and timeouts, this counter can be increasing without indicating a major problem.
OrphanStrmVol	aggregate number, in bytes, of all orphan buffers. If there are orphan streams, you should see this counter fluctuate up and down, hopefully down to 0.
OrphMsgMatches	number of message read requests satisfied by the message orphan buffer.
OrphMsgs	number of orphan messages received. Please note that this is the total count of orphan messages, not the current count.

Protocol Driver Counters	Description
OrphMsgVolume	current size, in bytes, of the message orphanage.
OrphStrmMatches	number of stream read requests satisfied by an orphan buffer.
SramDataFull	number of times that the Protocol Driver found the SRAM data queue full, and could not write to it.. If this count is not 0, you might need to increase the value in the maxHostDataXfer parameter.
SramGrantInterrupts	number of grant interrupts received by the Protocol Driver. The Protocol Driver must obtain permission before it can access the DM3 board-resident SRAM. Unless such permission has already been granted, the Protocol Driver must wait for a grant interrupt. This number should increase, but much more slowly than the number in the SramInterrupts counter.
SramGrantLost	number of SRAM grants that the Protocol Driver could not find. Each time the Protocol Driver finds an invalid value in the SRAM control structures, it tries to find a grant in the SRAM grant status register. **This count should always be 0**.
SramInterrupts	number of DM3 board interrupts that indicated that messages and/or data were ready to be read.
SramMsgFull	number of times that the Protocol Driver found the SRAM message queue full, and could not write to it.
StrmInQ	number of stream read requests received by the Protocol Driver. For example, these can be **ReadFile( )** function calls.
StrmInSram	number of stream blocks read from the SRAM.
StrmInQDone	number of stream read requests completed
StrmOutQ	number of stream send requests received

Protocol Driver Counters	**Description**
StrmOutQDone	number of stream send requests completed
StrmOutSram	number of stream blocks written to the SRAM.
StrmOverruns	number of times that the Protocol Driver could not buffer incoming stream data. Any non-zero value here dictates that you need to take remedial action. You might need to adjust the application. You might need to provide more physical memory. Clearly, if this counter is increasing persistently, it indicates a serious problem that you must address.
StrmsInPerSramSession	number of stream blocks that the Protocol Driver has read from the SRAM during any inbound session. This counter fluctuates between 0 and integer values that vary according to DM3-board-generated traffic. If this count is consistently low, you can increase the value in the hwIntInterval parameter.
StrmsOutPerSramSession	number of stream blocks that the Protocol Driver has written to the SRAM during any outbound session. This counter fluctuates between 0 and integer values that vary according to application-generated traffic.
StrmTimeouts	number of times that read requests have timed out. Although this is not as serious as stream overruns, this counter should not be increasing constantly.
UnknownInterrupts	number of times that the Protocol Driver received interrupts from the DM3 board, and could not determine the reasons. This value should always be 0.

14.5. Mpdtrace

This program can enable or disable driver debugging, and retrieve the driver debug buffer. The driver debug buffer is enabled and retrieved on a per board basis.

14.5.1. Usage

The `mpdtrace` command takes the following parameters:

Parameter	**Meaning**
/b *board number*	Board number. This is required.
/tr *tracing enable/disable*	0/1 enables or disables tracing.
/mi *message in enable/disable*	0/1 enables or disables message in.
/mo *message out enable/disable*	0/1 enables or disables message out.
/si *stream in enable/disable*	0/1 enables or disables stream in.
/so *stream out enable/disable*	0/1 enables or disables stream out.

14.5.2. Examples

- `mpdtrace /b3 /tr0`
  Enables driver tracing on board 3
- `mpdtrace /b3 /tr1`
  Disables driver tracing on board 3

14.6. Omdump

The omdump utility dumps orphan messages and places them in a file. An orphan message is defined as an MMB that was posted but there were no receive buffers for it.

14.6.1. Usage

The `omdump` command takes the following parameters:

Parameter	**Meaning**
/b *board number*	Board number. This is required.
/f *filename*	The name of the output file.
/?	Displays a help screen.

14.6.2. Examples

The following shows the command line and the output file:

- `omdump /b 0 /f orphan.dmp`
  This command dumps the messages in the orphan buffer to a file called *orphan.dmp*. The output is in the following format:

```
Current Time: Fri Feb 13 11:35:06 1998


#0
   status: alive with longevity=11
     flag: 80
      xid: 0x00000001, 1
     type: 0x00000020, 32
  srcDesc: 0:0:1:15:1
  dstDesc: 0:0:0:2:51
  msgSize: 24
   msg# 0: 0x00000001
   msg# 1: 0x0000123a
   msg# 2: 0x0013283c
   msg# 3: 0x00030001
   msg# 4: 0x0000000a
   msg# 5: 0x0000000c

#1
   status: alive with longevity=11
     flag: 80
      xid: 0x00000001, 1
```

```
   type: 0x00000020, 32
srcDesc: 0:0:1:15:1
dstDesc: 0:0:0:2:62
msgSize: 24
 msg# 0: 0x00000001
 msg# 1: 0x0000123a
 msg# 2: 0x0013283c
 msg# 3: 0x00030001
 msg# 4: 0x0000000a
 msg# 5: 0x0000000c
```

14.7. strmstat

This utility displays the current state of the stream(s) specified.

States shown include:

- Closing
- Closed
- Close failed
- Opening
- Opened for write
- Opened for read
- Open failed

14.7.1. Usage

The `strmstat` command takes the following parameters:

Parameter	**Meaning**
/i *number of streams*	The number of stream on which to report.
/b *board number*	Board number. This is required.
/s *start stream*	The first stream to report.
/?	Displays a help screen.

14.8. Examples

The following shows the command line and the output file:

- `strmstat /b 0 /i 16`
  This command displays the stream state for stream IDs 1 to 16 on board 0. The output is simliar to the following example:

```
Stream Status for Board = 0
Stream ID       State           Additional Info
     1          Closed
     2          Closed
     3          Closed
     4          Closed
     5          Closed
     6          Closed
     7          Closed
     8          Opened          Write
     9          Opened          Write
    10          Opened          Write
    11          Opened          Write
    12          Opened          Write
    13          Opened          Write
    14          Opened          Write
    15          Opened          Write
    16          Opened          Write
TOTALS
Closed = 7      Opened = 9      Closing = 0     Opening = 0
CloseErr = 0    OpenErr = 0     Unknown = 0
```

Index

O

P

Q

R

S

T

U

W

NOTES

NOTES

NOTES

DOCUMENTATION FEEDBACK FORM

Document Title: Using the DM3 Direct Interface for Windows NT

Publication Date: April, 1998 **Part Number:** 05-0987-001

1. Please rate this document in the following areas:

	Excellent	Good	Adequate	Fair	Poor	N/A
Accuracy	❑	❑	❑	❑	❑	❑
Clarity	❑	❑	❑	❑	❑	❑
Ease of Use	❑	❑	❑	❑	❑	❑
Relevance to Job	❑	❑	❑	❑	❑	❑
Code Examples	❑	❑	❑	❑	❑	❑
Organization	❑	❑	❑	❑	❑	❑
Completeness	❑	❑	❑	❑	❑	❑
Figures/Illustrations	❑	❑	❑	❑	❑	❑
Appearance	❑	❑	❑	❑	❑	❑
Overall Satisfaction	❑	❑	❑	❑	❑	❑

2. How can we improve this document?

❑ Improve the index
❑ Improve the organization
❑ Improve overviews and introductions
❑ Include more illustrations and figures
❑ Add more/better quick reference aids
❑ Add more troubleshooting information
❑ Add more step-by-step procedures and tutorials
❑ Make it more concise
❑ Add more detail
❑ Add more/better code examples
❑ Make it less technical
❑ Make it more technical

3. Please include any other comments on an additional sheet.

4. FAX this form to **DIALOGIC DOCUMENTATION MANAGER** at **(973) 993-5916.**

NAME: _____ **COMPANY:** _____

PHONE: _____ **ADDRESS:** _____