INSIDE MACINTOSH

QuickTime Conferencing

Developer Press

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Figures, Tables, and Listings

Chapter 1 Introduction to QuickTime Conferencing 1-1

Chapter 2 Conference Component 2-1

Chapter 3 Browser Components 3-1

Chapter 11 MovieTalk Protocol Messages 11-1

About This Book

This book describes QuickTime Conferencing, an extension to the Macintosh Operating System that allows you to integrate real-time collaboration services into your application. This book contains the information you need to use QuickTime Conferencing components and to design your own components.

If you are new to programming for Macintosh computers, you should read the book *Inside Macintosh: Overview* for an introduction to general concepts of Macintosh programming. You should also read other books in the *Inside Macintosh* series for specific information about other aspects of the Macintosh Toolbox and the Macintosh Operating System. In particular, to benefit most from this book, you should already be familiar with the Component Manager, as described in *Inside Macintosh: More Macintosh Toolbox*.

Format of a Typical Chapter

Most of the chapters in this book follow a standard structure. For example, the chapter "Conference Component" includes the following four sections:

- "About the Conference Component." This section provides an overview of the features provided by the conference component. You should read this section for a general understanding of the component and what tasks you can use it for.
- "Using the Conference Component." This section provides instructions on using the conference component. It describes how to use the most common routines, gives related user interface information, provides code samples, and supplies additional information. You should read this section if you need to use the services of the conference component.
- "Conference Component Reference." This section provides a complete reference to the constants, data structures, and routines provided by the conference component. Each routine description follows a standard format, which presents the routine declaration followed by a description of every parameter of the routine. Some routine descriptions also give additional information, such as circumstances under which you cannot call the routine.
- "Summary of the Conference Component." This section provides the C interface for the constants, data structures, routines, and result codes associated with the conference component.

In addition, some chapters contain sections that provide background information about a topic or advanced information such as how to create a component that conforms to the QuickTime Conferencing API.

Conventions Used in This Book

Inside Macintosh uses various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain information, such as parameter blocks, appears in special formats so that you can scan it quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and routines are shown in Courier (`this is Courier`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary at the end of this book.

Types of Notes

There are several types of notes used in *Inside Macintosh*.

Note
A note like this contains information that is interesting but possibly not essential to an understanding of the main text. ◆

IMPORTANT
A note like this contains information that is essential for an understanding of the main text. ▲

▲ **WARNING**
Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. ▲

Development Environment

The system software routines described in this book are available using C interfaces. How you access these routines depends on the development environment you are using. This book shows the interface to system software routines provided by the Macintosh Programmer's Workshop (MPW).

Code listings in this book show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and, in most cases, tested. However, Apple Computer does not intend that you use these code samples in your application.

For More Information

APDA is Apple's worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most products. APDA offers convenient payment and shipping options including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone:	800-282-2732 (United States) 800-637-0029 (Canada) 716-871-6555 (elsewhere in the world)
Fax:	716-871-6511
AppleLink:	APDA
America Online:	APDAorder
CompuServe:	76666,2405
Internet:	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information on registering signatures, file types, Apple events, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
1 Infinite Loop, M/S 303-2T
Cupertino, CA 95014

Introduction to QuickTime Conferencing

Contents

This chapter introduces the features and application program interface of QuickTime Conferencing, a set of functions and network protocols that you can use in your application to share time-based media across local and wide area networks. QuickTime Conferencing provides multimedia conferencing and collaboration services and is network, media, and platform independent.

Using QuickTime Conferencing, your application can help users find other users on a network, set up connections between users, send media data back and forth, and close down connections easily.

This chapter begins with a discussion of QuickTime Conferencing features, followed by an introduction to the concepts underlying QuickTime Conferencing and an overview of the QuickTime Conferencing architecture. This chapter will help you decide what level of QuickTime Conferencing support you want to incorporate in your application.

About QuickTime Conferencing

QuickTime Conferencing consists of a set of software components that work with the standard QuickTime components to provide real-time collaboration services over a network. A **component** is a piece of code, managed by the Component Manager, that provides a defined set of services to one or more clients. Applications, system extensions, as well as other components can use the services of a component. To develop applications that use QuickTime Conferencing you need to be familiar with the Component Manager and how components are opened and used in the Macintosh Operating System. See *Inside Macintosh: More Macintosh Toolbox* for information about the Component Manager.

Depending on the level of customization you wish to add to the basic QuickTime Conferencing services, you may also need specific knowledge of QuickTime and the standard QuickTime components, networking and network protocols, and PowerTalk services.

This section provides an overview of QuickTime Conferencing features and the services QuickTime Conferencing can provide to your application.

Multimedia Collaboration

QuickTime Conferencing provides system software support for real-time multimedia communications. Real-time collaboration services such as shared workspace, shared windows, and shared applications can be layered on top of this foundation.

Examples of multimedia collaboration include

- videoconferencing between two or more people over a local or wide area network
- sharing text, graphics, and annotations on a virtual “whiteboard”
- delivering lectures or demonstrations to remote locations

Network Independence

QuickTime Conferencing takes advantage of the existing network interfaces and software in the Macintosh. QuickTime Conferencing includes components that support AppleTalk over EtherTalk, LocalTalk, and TokenTalk. QuickTime Conferencing also includes components that support TCP/IP and ISDN connections.

QuickTime Conferencing can also take advantage of OpenTransport protocols and services. In this case, OpenTransport provides all of the network services needed by QuickTime Conferencing.

QuickTime Conferencing is designed so that Apple and third-party developers can create new transport components to support additional network interfaces and protocols.

Interoperability

Because the QuickTime Conferencing architecture is independent of any particular application, media type, or operating system, it provides a foundation for development of cross-platform collaboration solutions. By supporting existing and future standards for videoconferencing and multimedia collaboration, QuickTime Conferencing allows applications on the Macintosh and other platforms to interoperate.

Extensible Architecture

QuickTime Conferencing provides a flexible API and a real-time protocol for multimedia collaboration. Because the QuickTime Conferencing architecture is modular, you can create components to support additional networks and protocols, compression algorithms, or other features.

You can differentiate your products by adding new features or extending the basic QuickTime Conferencing services. Here are a few examples of how you can add value to QuickTime Conferencing:

- Creating new network or transport components to support other communications standards. Translators can be created to support other media-aware networking protocols as well as to support QuickTime Conferencing protocols on existing network systems. There are also opportunities for broadband networks, wireless communications systems, and interactive digital cable systems.
- Development of software or hardware compression components appropriate for live network transmission.
- New collaborative applications that use video and audio to augment other forms of collaboration.
- New network services such as conference bridging, media routing, and media servers.

QuickTime Conferencing Concepts

This section discusses the key concepts underlying QuickTime Conferencing and describes the relationship between QuickTime Conferencing and other parts of the Macintosh Operating System.

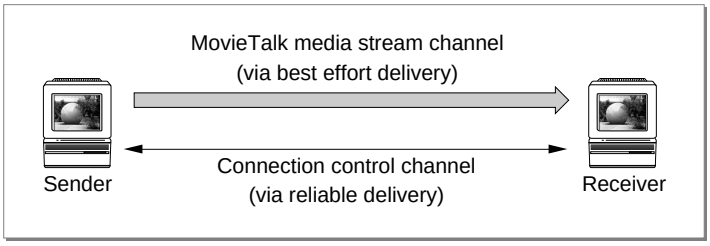
Multimedia Network Protocols

QuickTime Conferencing provides an open architecture that can support any number of multimedia network protocols. In addition to support for emerging international standards such as **H.320** and **RTP**, QuickTime Conferencing defines a set of protocols known collectively as the MovieTalk protocol for exchanging media and control information over networks.

The **MovieTalk protocol** provides a standardized method for setting up, maintaining, and breaking down connections, as well as for delivering media data efficiently. The protocol is carried over two distinct channels across the network. The **control channel** is used to transmit control information about a connection. This might include status information about a connection, information describing the media format, or a request to end a connection. Your application can also use this channel to transmit application-specific information, such as user events, text, or pictures. The control channel uses a reliable network service to ensure that the information sent from one side will be received intact on the other, in order and without error.

A MovieTalk **media channel** is used to send media data across the network. This channel typically uses the network's datagram service for "best effort" delivery. The media channel is more efficient than the control channel for moving data, but does not provide error recovery mechanisms when a packet is lost, corrupted, or received out of sequence. This is generally acceptable for real-time media, where the primary goal is to keep data flowing. Figure 1-1 depicts the MovieTalk control and media channels.

Figure 1-1 MovieTalk control and media channels



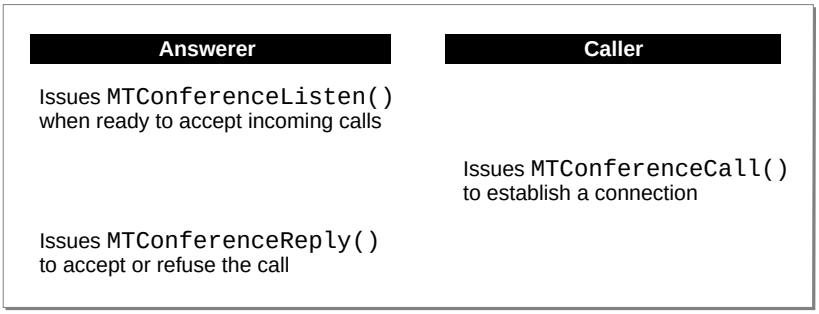
The MovieTalk protocol is implemented inside a QuickTime Conferencing transport component. MovieTalk protocol messages consist of a header and payload and are

designed to permit fast and efficient processing. There are a small number of standard messages that define the complete protocol, making it easy to interoperate between applications and platforms. The chapter "Transport Components" in this book describes the operation of these components and how to create new ones. The chapter "MovieTalk Protocol Messages" describes the messages that comprise the MovieTalk protocol.

Connection Types

The relationship between QuickTime Conferencing applications sending media data to each other across a network is referred to as a **connection**. A connection is established by exchanging specific MovieTalk protocol messages between a caller (the application originating the call) and an answerer (the one accepting or rejecting the call). However, application developers need not be concerned with the underlying protocol because QuickTime Conferencing provides high-level routines for opening, maintaining, and closing connections. For example, Figure 1-2 shows a sequence of QuickTime Conferencing function calls that can be used to set up a connection.

Figure 1-2 Initiating a connection

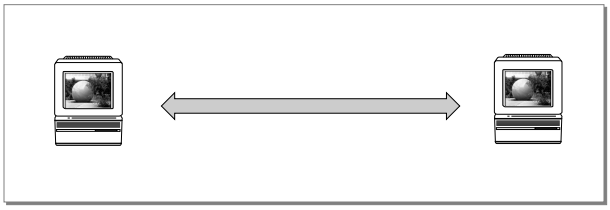


QuickTime Conferencing uses the existing network software and hardware to establish connections to other QuickTime Conferencing clients on the network. Connections can be configured in one of three ways: point-to-point, multipoint, and multicast.

Point-to-Point Connections

The most common type of connection is a conventional two-way or **point-to-point connection**. Control and media information flow across the connection bidirectionally. This is generally useful for private telephone (or videophone) communications, collaborative work, and client-server applications. Figure 1-3 shows a point-to-point connection.

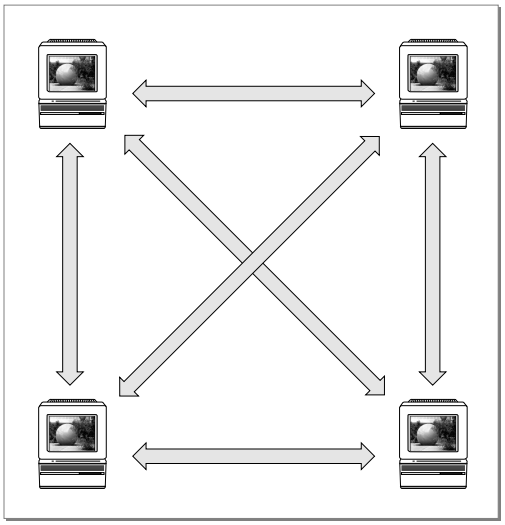
Figure 1-3 A point-to-point connection



Multipoint Connections

In a **multipoint connection**, QuickTime Conferencing clients may be connected to more than one other client on the network. Control and media data flow between each connected client. Typically, these connections are considered "fully connected," meaning that each client has a direct link to every other client in the connection. This is most often used in small conference and workgroup situations. Figure 1-4 shows a multipoint connection.

Figure 1-4 A multipoint connection



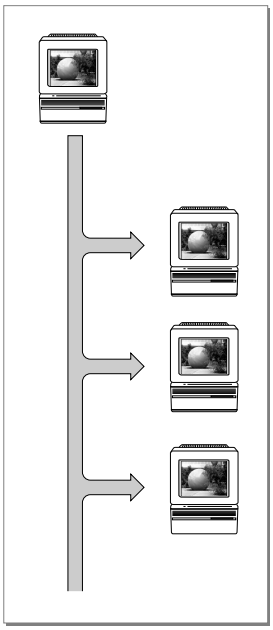
Multicast Connections

A **multicast connection** provides one-to-many media delivery. Control and media data flow one way from a sender to any number of receivers. For example, in a lecture or "broadcast" application, you might want to transmit both video and sound across an enterprise network.

Multicast data typically goes only to those receivers requesting the data and consumes only one stream of network bandwidth per data sender. In order to use multicast

connections, a multicast network protocol and a shared media network must be available. QuickTime Conferencing uses the AppleTalk Multicast Datagram Protocol and the Simple Multicast Routing Protocol to provide this capability over Ethernet networks. Figure 1-5 shows a multicast connection.

Figure 1-5 A multicast connection



QuickTime and QuickTime Conferencing

QuickTime Conferencing uses a number of standard QuickTime components, such as the sequence grabber, image compressor, and video digitizer components. This approach allows QuickTime Conferencing to take advantage of the functionality that QuickTime provides and to expand as QuickTime expands. For example, new image compressor components developed for QuickTime can also be used by QuickTime Conferencing for network media connections.

The component architecture allows developers to add new functionality to QuickTime Conferencing in a modular manner. For example, you could create a transport component that supports a new multimedia protocol. When you register your component with the Component Manager, existing QuickTime Conferencing applications could use the new protocol.

Because of the relationship between QuickTime and QuickTime Conferencing, users must have QuickTime installed in order to use QuickTime Conferencing.

PowerTalk and QuickTime Conferencing

PowerTalk is a set of Macintosh Operating System extensions that enhance the collaboration capabilities of the Macintosh. PowerTalk features include a unified network browsing mechanism, encryption and authentication services, electronic mail services, and a system-wide address catalog system.

QuickTime Conferencing applications can use PowerTalk functions to provide enhanced collaboration on Macintosh computers that have PowerTalk installed. For example, the QuickTime Conferencing browser component can use the PowerTalk directory services. This offers users the convenience of a single interface for locating other QuickTime Conferencing users on all types of networks, and allows them to save this information in their personal catalog.

QuickTime Conferencing does not require PowerTalk, but will take advantage of PowerTalk services if they are available. For more information about PowerTalk services, see *Inside Macintosh: AOCE Application Interfaces*.

QuickTime Conferencing Architecture

QuickTime Conferencing combines two technologies: a set of QuickTime components for creating and controlling media streams, and the MovieTalk protocols for transmitting those streams over a network. QuickTime Conferencing includes these components

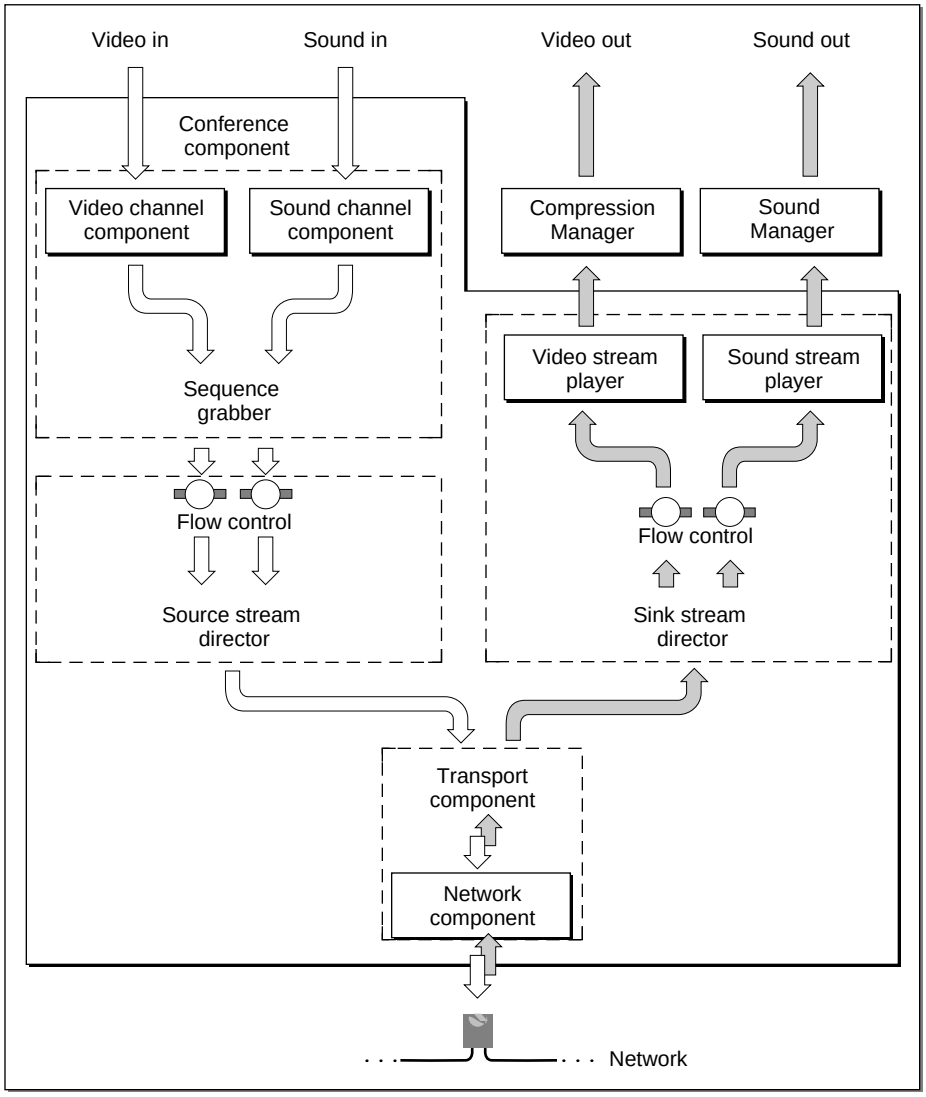
- conference component
- browser components
- stream controller components
- stream director components
- stream player components
- flow control components
- recorder components
- transport components
- network components
- utility components

The Conference Component

The QuickTime Conferencing components are organized hierarchically, with the conference component providing a high-level interface for applications. The conference component provides all the services necessary to implement QuickTime Conferencing. Although applications are not required to use the conference component, your application must use the conference component if you want it to interoperate with the Apple Media Conference application or other applications that use the conference

component. Figure 1-6 shows how the conference component encapsulates the lower-level components.

Figure 1-6 The QuickTime Conferencing component architecture



The conference component provides functions for establishing and maintaining point-to-point and multipoint connections. You can use the conference component to add QuickTime Conferencing capabilities to your application without having to understand or interact directly with the underlying components. If you want to extend the functionality provided by the conference component, you can use low-level components as needed while still relying on the conference component to manage connections. For example, you can use browser components to obtain network

addresses, or stream controller components to customize the way media data is presented to the user.

When you open the conference component, it registers itself with the selected network component. If multicast services are available, the conference component automatically switches to a multicast connection when the number of parties in a conference exceeds two. Because the conference component supports a distributed, rather than centralized, conferencing model, the party that initiates a multiparty conference can leave at any time without affecting the connections between the remaining parties.

The chapter "Conference Component" in this book describes the conference component in detail and provides examples that show you how to incorporate QuickTime Conferencing into your application.

The Browser Components

The QuickTime Conferencing browser components provide a user interface for locating and selecting other QuickTime Conferencing users on a network. Browsers return a MovieTalk address data structure that your application can use to establish a connection.

QuickTime Conferencing includes these browser components

- A PowerTalk browser that provides access to PowerTalk network browsing facilities as well as PowerTalk personal address books and directories.
- An AppleTalk browser component that uses a zone/name interface, much like the Chooser.
- A TCP/IP browser that accepts an Internet node name or domain address.
- An ISDN browser that accepts ISDN telephone numbers.

If your application allows users to make outgoing calls, you will need to read the chapter "Browser Components" in this book to learn how to open and use the browser components.

The Stream Controller Components

Stream controller components provide the user interface for MovieTalk connections. You can use stream controller components to customize the presentation of visible and audible media data and to provide controls for pausing, recording, adjusting the volume, and so on.

The conference component automatically manages the stream controller for your application. If your application uses the conference component you do not need to call a stream controller component directly unless you need more control over how the media stream is presented or you want to implement application-specific user interface features.

The chapter "Stream Controller Components" in this book describes how to customize the standard stream controller. Figure 4-1 on page 4-5 shows the standard stream controller your application receives through the conference component.

The Stream Director Components

Stream director components help configure and connect components, and manage the streams of media data that flow between them. The tasks performed by stream director components include

- negotiating media formats across the network by sending stream description messages through the transport component
- moving media data between the sequence grabber and transport components
- creating, configuring, and removing stream player components based on media format information that it obtains
- demultiplexing streams of media data received from the transport component and forwarding that data to the appropriate stream player component

If your application uses the conference component, you generally do not need to interact directly with stream director components because the conference component manages them for you. Stream director components are described in detail in the chapter "Stream Director Components" in this book.

The Stream Player Components

Stream player components display media data from a QuickTime Conferencing stream source, usually a stream director. Stream player components provide a standard API for displaying media data, independent of media type. QuickTime Conferencing currently provides stream player components for video and audio streams. Additional stream players may be added by Apple or third parties to support other types of media data, such as text or MPEG streams.

Because stream players are managed automatically by stream director components, applications do not normally need to call stream player functions directly. If you want to create your own media stream type, you should read the chapter "Stream Player Components" in this book.

The Flow Control Components

Flow control components are used by the stream director components to manage the flow of media data between remote ends of a connection. Flow control components conserve network bandwidth and help ensure a smooth flow of data across a connection, independent of CPU performance differences or network traffic conditions. In the absence of a flow control component, data flows freely through a connection.

QuickTime Conferencing provides standard flow control components for video and sound. The functions of the video flow control component include

- computing a time-weighted measurement of network performance, which provides a statistical indicator of successful network delivery and local throughput
- communicating with the video flow control component at the remote end to increase or decrease the flow of data to improve performance

The sound flow control component manages the flow of sound media data and implements an algorithm for echo reduction.

Applications do not interact directly with flow control components—they are called only by stream director components. See the chapter "Flow Control Components" in this book for more information.

The Recorder Components

Recorder components are used to record the media data being carried in a connection. This feature is useful for preserving a record of a conference, and can be configured to record one or more members. The recorder component provided by Apple creates QuickTime movies.

To use a recorder component you attach it to one or more stream director components and specify where the movie file should be stored. The media data being handled by those stream directors will be recorded into a multi-track movie.

Applications that use the conference component usually do not need to interact directly with recorder components. The chapter "Recorder Components" in this book describes recorders in more detail and explains how you can create your own.

The Transport Components

QuickTime Conferencing transport components are responsible for setting up, maintaining, and breaking down connections. Transport components implement the media-specific protocol, exchange state information, determine error conditions, and transfer media data efficiently. QuickTime Conferencing includes transport components that implement MovieTalk network protocols to deliver control and media data using reliable and best effort mechanisms.

Additional transport components can be created to use other real-time media protocols. Transport components translate the data that arrives over the network into the format used internally by QuickTime Conferencing, and convert media and control requests from other QuickTime Conferencing components into signals appropriate to the network protocol.

Because the conference component manages transport components automatically, applications that use the conference component do not normally need to interact directly with transport components. If your are developing a new transport component, or require greater control over transport component services, you should read the chapter "Transport Components" in this book.

The Network Components

QuickTime Conferencing network components provide a standard interface to the network software and hardware of the Macintosh. Transport components use the services of network components to deliver control and media data across a network.

In addition to managing the transfer of control and media data, network components provide functions for network address management. QuickTime Conferencing uses the concept of network-independent addresses to specify entities on the network for call control purposes. For network protocols that include directory services, the network component provides functions for network name registration, removal, lookup, and address extraction.

QuickTime Conferencing supplies the following standard network components:

- **AppleTalk network component**
  This component uses AppleTalk network software and hardware to carry data across a connection. The AppleTalk Data Stream Protocol (ADSP) provides the reliable connection for the connection control channel, while AppleTalk Datagram Delivery Protocol (DDP) is used for the media stream channel. This network component works with any supported AppleTalk transport layer and is designed for local area networks.
- **TCP/IP network component**
  The TCP/IP network component delivers media and control data across a TCP/IP network. This component is designed to be used in local area networks as well as the Internet. The TCP/IP network component uses the Transmission Control Protocol (TCP) to carry control information and the User Datagram Protocol (UDP) to carry media data.
- **ISDN network component**
  The ISDN network component carries both media and control data across a reliable ISDN channel. This network component is designed primarily for wide area networks.
- **AppleTalk multicast network component**
  This network component uses multicast extensions to AppleTalk to broadcast a single source of media and control data to multiple receivers. When used in conjunction with multicast routers, the multicast network component can operate within multiple-zone networks.
- **Open Transport network component**
  For systems that use Open Transport, this network component can be configured to use any network service available through Open Transport.

Additional QuickTime Conferencing network components can be created to support other types of networks.

Applications that use the conference component or transport components do not need to interact directly with network components. For information about developing and using network components see the chapter “Network Components” in this book.

The Utility Components

QuickTime Conferencing includes two component types that provide utility functions to other components and to applications. The **idler component** provides periodic processing time when it is safe to allocate or move memory. The **memory component** provides enhanced memory management services.

The idler component is normally used by other QuickTime Conferencing components and not directly by applications. If you are developing a QuickTime Conferencing

component, you should read about the idler component in the chapter "Utility Components" in this book.

The memory component provides important functions for both components and applications. In particular, the memory component can service memory requests at interrupt time and supports multiple grow-zone functions. QuickTime Conferencing components and applications that include a grow-zone function should use the services of the memory component for more efficient use of available memory.

Conference Component

Contents

This chapter describes the conference component. The **conference component** allows your application to establish and maintain peer-to-peer conferences with one or more remote systems. This chapter is divided into the following major sections:

- "About the Conference Component" introduces you to the features and functions of the conference component and discusses its relationships with other QuickTime Conferencing components.
- "Using the Conference Component" provides several examples of how you can use the conference component in your application.
- "Conference Component Reference" contains detailed technical information about the interface supported by the conference component.
- "Summary of the Conference Component" provides a summary of the conference component's interface.

This chapter is intended for developers of applications that use conference components.

The conference component is a Component Manager component. Therefore, you need to be familiar with components and how to interact with them before you can use the conference component. If components are new to you, see the chapter that discusses the Component Manager in *Inside Macintosh: More Macintosh Toolbox*.

About the Conference Component

This section introduces the conference component. The conference component allows your application to establish communications between two or more users on a network. In a conference, your application can exchange both control information and media data. Conference component conferences are peer oriented, requiring no centralized management.

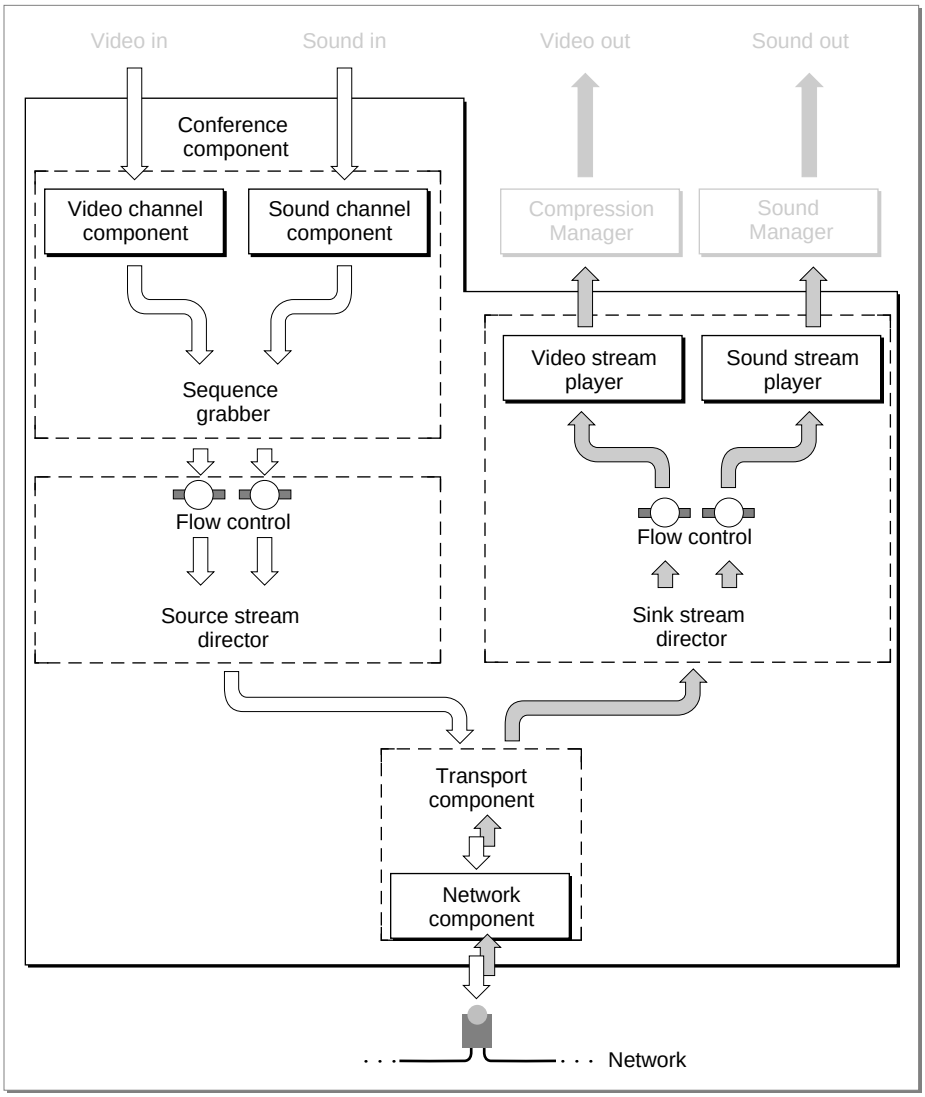
The conference component provides its functionality by working with several different QuickTime Conferencing components on your behalf. The conference component manages streams using stream director and stream player components, it interacts with the underlying network using transport components, and it works with media data using controller components. Users generally use a browser component to find other potential conference participants. Figure 2-1 depicts the relationships between the conference component and other QuickTime Conferencing components.

The conference component's main function is to establish and maintain a bidirectional control channel between every member of a conference. Conferencing applications use the control channel to exchange nonmedia data that is pertinent to the conference. This data might include user identification information or other information that is germane to the application's operation. Conferencing applications define the format and content of these control messages by establishing their own control protocols. If you are developing a conferencing application, you will need to define such a control protocol for your own use.

In addition, the conference component provides facilities that allow you to work easily and effectively with other QuickTime Conferencing components in order to provide and

present media data to conference members. Controller components allow you to provide media data to a conference and can present conference media data to the user. To do so, the conference component works with appropriate controller components, which, in turn, may use other QuickTime components, such as sequence grabbers.

Figure 2-1 The relationship between the conference component and other QuickTime Conferencing components



The conference component is designed around an event architecture. Conferencing applications must poll the conference component regularly to detect events. The application must then take appropriate action in order to continue to participate in the conference.

About Conferences

A conference that is managed by the conference component allows an application to easily establish communications with applications on other computers. Conferences themselves place few restrictions on client applications. For example, there is no limit to the size of a conference, and members can join and leave at will. Even the member that started the conference can leave at any time—the remaining conference members continue to communicate with one another. Further, a single application may participate in more than one conference at a time.

However, conferences do have some characteristics that you need to understand before you can proceed to use them. This section addresses those concepts and is divided into the following topics:

- "Service Types" discusses conference service types and how you use them.
- "Starting and Joining Conferences" describes how you go about starting or joining a conference.
- "Conference Modes" provides you information about the types of conferences supported by the conference component.

Service Types

The conference component uses service type values to identify different types of conferences that are available on a network. Before two applications can communicate with one another, they must agree on the protocol they are going to use. The service type identifies the protocol.

For example, you might be interested in creating a particular type of videophone application. You would require your users to install copies of your application on each system that is to participate in the videophone conferences. Naturally, your application would need to exchange control information before opening any media channels between systems, and you would define the nature of those control message exchanges. In order to make sure that your users were, indeed, talking with your application, you would specify a unique service type that identifies your application on the network. In this case, let's call that service type "My Simple Videophone." You would make sure that your application allowed the user to connect with applications that support a service type of "My Simple Videophone." In this manner, you assure yourself that all conference members are using the same protocol to communicate.

As conference-based applications develop, you can expect that one or more service types will emerge as de facto standards. As this happens, you may want to make sure that your application can interact with those standard applications. In order to do so, you will want to make your application aware of those standard protocols, providing support for at least a subset of their functionality. For example, Apple's videophone application, Apple Media Conference, uses a service type value of "VideoPhone." If you want your application to be able to interact with SimpleTalker, you will need to support the media protocols used by that application.

You list a service type on a network when you instruct the conference component to listen for incoming calls. You use the `MTConferenceListen` function to do so. When you call that function, you specify a list of networks and service types that you support.

The service type value is expressed in a manner that is consistent with the underlying network. For networks, such as AppleTalk, that support named services, the service type values are themselves names. For networks that use other naming schemes, the service type values comply. For instance, on TCP/IP networks, where port numbers are used to identify well-known services, a service type may be nothing more than an agreed-upon port number.

For more information about registering service types, see "A Simple Conferencing Application" beginning on page 2-10.

Starting and Joining Conferences

Starting a conference is as simple as making yourself available to other systems on a network by listening for incoming calls. When you listen for incoming calls, you publish your service type. Other systems on the network can then connect with you by selecting your service. You listen for incoming calls by calling the `MTConferenceListen` function. Other potential participants call you using the `MTConferenceCall` function.

Because the conference component implements a peer-to-peer management model, conferences, once started, take on a life of their own. Any member of a conference can leave a conference at any time. As long as there are other members left, the conference continues to exist, even if the member who originally established the conference leaves.

As a conference grows from two to more members, the conference component automatically takes advantage of the network's ability to efficiently route the conference's messages. If the underlying network supports multicast messaging, the conference component converts the conference's media data flows from point-to-point to multicast. This conversion happens transparently to your application.

New members may join a conference at any time. Joining a conference involves browsing the network for people or servers with which you want to connect and then establishing a conference with that entity. If that entity is already in a conference, you may be allowed to join that conference, depending upon whether the conference accepts new participants. Thus, conferences with more than two participants are created by merging separate conferences.

When a member joins a conference, the conference component tells each of the conference's participants that a new member has joined. The new member's conference component then initiates connections with all of the other members of the conference. Once this process is complete, each member of the conference can interact with every other member.

For more details about receiving incoming calls, see "A Simple Conferencing Application" beginning on page 2-10.

Conference Modes

By default, the conference component creates conferences that allow all conference members to exchange media data with one another and allow new members to join and participate freely at any time. However, the conference component allows you to control these conference characteristics by manipulating the conference's mode. A conference's mode governs several aspects of conference operation, in particular

- whether a member sends or receives media data
- whether a member's data is to be shared with new participants
- whether all conference members are able to join with one another in fully connected conferences

You set a conference's mode before you make yourself available for conferencing by calling the `MTConferenceListen` function. Use the `MTConferenceMode` function to establish the conference's mode. Whenever a new participant joins the conference, the conference component provides the conference's mode to that participant in either an `mtPhoneRingingEvent`, `mtIncomingCallEvent`, or `mtMemberJoiningEvent` event message, depending upon how the new member is added.

For examples of different ways to manipulate conference modes, see "A Conference Without Media Data" beginning on page 2-20 and "A Conference Server" beginning on page 2-23. For more information about working with conference modes, see "Configuring the Conference Component" beginning on page 2-26.

Control Channels and Media Channels

Once you have received or placed a call, you have established a control channel with the other conference participants. Your application can then use the control channel to exchange application-specific information with other participants. For example, your application might exchange detailed user information with other participants, allowing other users to identify each conference member more precisely. Your application defines the format and content of the data that is exchanged over this control channel.

The conference component uses the underlying network's reliable messaging facilities to support the control channel. As a result, as long as the conference is active, you are assured that your control messages will reach other active participants. The conference component provides a facility that allows you to determine the level of control-message support offered by other conference participants. This facility is discussed later in this section, in "Control Message Capabilities." For more information about sending and receiving control messages, see "Exchanging Messages" beginning on page 2-54.

This control channel is not well suited to exchanging media data, however. Typically, reliable network channels incur significantly more communications overhead than can be tolerated by time-critical media data. Therefore, the conference component also supports separate media channels for a conference's media data. These media channels generally use more efficient but less reliable network facilities. In order to exchange media data in a conference, you must also establish media channels with each participant.

Unlike the control channel, the format and content of the media channel are governed by the nature of the media itself. If you are sending QuickTime video data, the conference component manages the exchange of chunks of video data that are appropriate to the movie. The stream's image descriptor carries the configuration information that allows conference participants to understand the media data. In order to present the media data to the user, your application must use an appropriate controller component. The facilities offered by the conference component that help you work with media controllers are discussed later in this section, in "Controllers."

Control Message Capabilities

Your application has complete control over the format and content of its control messages. The conference component allows you to exchange these control messages with other conference participants. However, it's not unusual for communications protocols to change over time (in fact, it's only unusual if they don't change). As a result, it's reasonable to expect that your control protocol may change over time and that you may have users running different versions of your application on the same network. In order to help you manage different control protocol versions, the conference component provides capability negotiation features.

These features are also useful in the event that you are planning to interoperate with a standard conferencing application protocol.

These negotiation features allow you to exchange protocol support information with other potential participants before you join, or allow a new participant to join, a conference. This information can indicate the protocol version supported by the potential participant. This allows you to determine whether you can communicate with the potential participant's application.

This information can also indicate whether any given protocol message is required before starting a conference. In many cases, conference applications do not require specific control-message exchanges. However, if your application requires that all conference participants authenticate themselves before they join a conference, you might use control channel messages to conduct the necessary authentication. In this case, you would probably require that each application exchange certain messages before allowing a new participant to join.

For an example of an application that uses control messages to exchange a required password, see "A Simple Conferencing Application" beginning on page 2-10. For more information about negotiating message capabilities, see "Exchanging Messages" beginning on page 2-54.

Controllers

While it is possible to have conferences without media data, where participants exchange data only by means of control messages, media-rich conferences are far more interesting. A conference participant's media channel carries the user's media data to other participants. In order both to provide and to present media data, your application must work with appropriate controller components. The conference component provides facilities that allow you to obtain and use controller components in a conference.

For an example of working with controllers, see "A Simple Conferencing Application" beginning on page 2-10. For more information about obtaining controllers, see "Working With Controllers" beginning on page 2-46. For more information about using controllers in a conference, see "Activating Conferences" beginning on page 2-43.

Auxiliary Media Sources

The conference component does not limit the number of media sources any participant may supply to a conference. In general, each participant provides one media source to the conference and receives one source from each other participant. In some cases, though, it is useful for a participant to deliver one or more additional media sources to a conference. For example, while using a videophone application to chat with another user, you might decide to share an interesting QuickTime movie. Using the conference component, you could do so by opening an auxiliary media source for that movie.

Auxiliary media sources are treated much like any other conference media source. You need to obtain an appropriate controller component to provide the media data, and you need to assign the controller to the conference. The conference component provides facilities that allow you to do all of these things. For an example, see "A Conference With Auxiliary Sources" beginning on page 2-22.

Using the Conference Component

This section provides examples that show you how you can use the conference component in your application. Each of the examples highlights a different aspect of the services provided by the conference component and includes sample code illustrating how to use those services.

This section contains the following examples:

- "Opening and Closing the Conference Component" shows how to use the Component Manager to open and close an instance of the conference component.
- "A Simple Conferencing Application" presents a basic conferencing application.
- "A Conference Without Media Data" shows how you would change the basic application to support a text-only conference.
- "A Conference With Auxiliary Sources" provides an example of working with auxiliary media sources.
- "A Conference Server" shows how to set up a send-only conference.

Opening and Closing the Conference Component

As is the case with all Component Manager components, you must use Component Manager functions in order to gain access to the conference component. If you are not

familiar with the Component Manager, refer to the appropriate chapter in *Inside Macintosh: More Macintosh Toolbox*.

To open an instance of the conference component for your use, call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. The following code fragment uses the `OpenDefaultComponent` function:

```
cci = OpenDefaultComponent(kMTConferenceType,
                           kMTMovieTalkSubType);
```

This function requires you to identify the type and subtype of the component you desire. The `kMTConferenceType` constant identifies the component type—in this case, the conference component.

Conference components use their subtype value to identify the type of conference they support.

When you are done with the conference component instance, use the `CloseComponent` function to close the instance:

```
result = CloseComponent(cci);
```

When you do so, the conference component removes you from any remaining conferences and releases any associated system resources.

A Simple Conferencing Application

This example shows how you can use the conference component to set up and participate in a conference that carries media data. This program from which this sample code was derived implements a multi-user videophone application.

The conference established by this application has a number of notable characteristics:

- First, it arbitrarily limits the number of participants to four.
- Second, it allows participants to grab still images and capture movies.
- Third, it includes support for a clear-text (nonencrypted) password facility that is used to authenticate conference participants.

The source code in this example provides the basis on which the subsequent examples are built. Listing 2-1 contains the program's important data declarations.

Listing 2-1 A simple conferencing application

```
/*------------------------------------------------------------
   Data types, data structures, etc.
*------------------------------------------------------------*/

typedef  EventRecord    *EventRecordPtr;
typedef  Byte           *BytePtr;
```

```
#define  PW_CLEAR       'pw-C'      /* clear-text password */
#define  MAXMEMBERS     3           /* maximum participants */

/*
   A handy structure for participant information
*/
typedef struct {
   Rect                    box;
   MTConferenceMember      member;
   MTControllerComponent   controller;
} Member, *MemberPtr;

#define  WRECT(top, left)  (top),(left),(top+120),(left+160)

/*
   Global storage
      Allocate and initialize array of Member structures; note
      positioning of each participant's viewing rectangle
*/
   Member         gMember[MAXMEMBERS+1]= {
                        {  WRECT(100, 100)},
                        {  WRECT(100, 300)},
                        {  WRECT(300, 100)},
                        {  WRECT(300, 300)},
                     };
/*
      Other handy globals
*/
   MTCString63       gUserName, gPassword;

   ComponentInstance gCC, gBC;
   MTConferenceToken gConference;
```

The `Start` function, presented in Listing 2-2, initializes the conference component and starts listening for incoming connections. You might define a similar function and call it during your program's initialization processing.

Listing 2-2 Initializing the conference component

```
/*-------------------------------------------------------------
   Start: Start the conference component
*-------------------------------------------------------------*/
```

```
void Start(void)
{
   StringHandle   sh;
   Handle         srvt;

   sh = GetString(kMTAppleTalkUserName);

   strncpy(gUserName, *sh+1, **sh);

   gCC = OpenDefaultComponent(kMTConferenceType,
                              kMTMovieTalkSubType);

/*    obtain and initialize my controller; note option settings */
   gMember[0].controller = MTConferenceNewPreparedController(gCC,
                           &gMember[0].box,
                           mtMediaSourcePrepMask +
                           mtGrabVideoPrepMask +
                           mtGrabAudioPrepMask +
                           mtWindowVisiblePrepMask +
                           mtControllerVisiblePrepMask +
                           mtEnableSnapshotPrepMask +
                           mtEnableRecordPrepMask,
                           0);

   srvt = GetResource('srvt', rServiceTypes);
   HLock(srvt);

   MTConferenceListen(gCC, gUserName, gUserName, *srvt);
   ReleaseResource(srvt);

   gBC = OpenDefaultComponent(kMTBrowserType,
                              kMTAppleTalkSubType);
}
```

This application stores its service-type list in a resource that has the following Rez definition:

```
resource 'srvt' (rServiceTypes, "Service Types") {
   "mtlkatlk\tSeeWorld\nmtlktcpi\t222\nmtlkisdn\t-\n"
};
```

Note that this function does not set the conference's mode. This application uses the conference component's default mode setting, allowing all conference members to interact freely and exchange both control and media data.

In your program's main event loop, you should check for conference events. The code fragment shown in Listing 2-3 demonstrates one way to do this. When you receive a conference component event, your application must process the event appropriately. The `DoConferenceEvent` function is discussed later in this section.

Listing 2-3 Checking for conference component events

```
EventRecord       tEvent;
MTConferenceEvent cEvent;

for(;!gQuit;){
   if(MTConferenceGetNextEvent(gCC, &cEvent))
      DoConferenceEvent(&cEvent);

   if(WaitNextEvent(everyEvent, &tEvent, 1, 0))
      DoToolboxEvent(&tEvent);
}
```

When your application shuts down, you must be sure to close your connection to the conference component, as well as any other QuickTime Conferencing components you may have used. The following code fragment shows how to use the Component Manager's `CloseComponent` function to do so.

```
CloseComponent(gCC);
```

A videophone application needs to allow the user to place and accept calls with other users. The `CallOne` function in Listing 2-4 shows how this simple videophone application places a call to a remote user. Your application might invoke a function like this one when the user selects an appropriate menu item. You would handle incoming calls as part of your normal conference component event handling. This is discussed later in this section.

Listing 2-4 Establishing a call to a remote user

```
/*------------------------------------------------------------
   CallOne: call a new participant
*------------------------------------------------------------*/
void CallOne(MTNamePtr np)
{
   MemberPtr   mp;

   if(!(mp = FindMember(0)))        /* find a member structure */
      return;
```

```
   mp->member = MTConferenceCall(gCC, "See Wiz", np);
}
```

In addition to placing calls, a videophone application must allow the user to end a call. Again, you might present this functionality to the user as a menu item. The `DoMenuHangUp` function, shown in Listing 2-5, uses the `MTConferenceTerminate` function to end a call.

Listing 2-5 Ending a conference call

```
/*------------------------------------------------------------
   DoMenuHangUp: handle hang-up menu item
*------------------------------------------------------------*/
void DoMenuHangUp(void)
{
   MTConferenceTerminate(gCC, gConference);
   gConference = 0;              // just to suppress menu choice
}
```

When your application receives a conference component event, you must process that event appropriately. The `DoConferenceEvent` function, shown in Listing 2-6, receives all of this application's conference events and then calls the appropriate local routine to handle each one. Conference component events are discussed in more detail in "Managing Conference Events" beginning on page 2-31.

Listing 2-6 Handling conference events

```
/*------------------------------------------------------------
   DoConferenceEvent: main conference event handler
*------------------------------------------------------------*/
void DoConferenceEvent(MTConferenceEventPtr ce)
{
   switch(ce->what){
   case mtIncomingCallEvent:
      DoIncomingCall(ce);
      break;

   case mtMemberJoiningEvent:
      DoMemberJoining(ce);
      break;

   case mtConferenceReadyEvent:
```

```
      DoConfReady(ce);
      break;

   case mtConferenceTerminatedEvent:
      DoConfTerminated(ce);
      break;

   case mtMemberReadyEvent:
      DoMemberReady(ce);
      break;

   case mtMemberTerminatedEvent:
      DoMemberTerminated(ce);
      break;

   case mtPhoneRingingEvent:
      DoPhoneRinging(ce);
      break;

   case mtRefusedEvent:
      /* process error */
      break;

   case mtFailedEvent:
      /* process error */
      break;

   case mtSnapshotTakenEvent:
      SaveSnapshot(ce->surprise);
      break;

   case mtMovieRecordedEvent:
      SaveMovie(ce->surprise);
   }

   if(ce->surprise)
      DisposeHandle(ce->surprise);
}
```

The `DoIncomingCall` function, shown in Listing 2-7, accepts an incoming call from a remote user. This routine first verifies that the caller is willing to exchange media data with the local user. If the remote user only wants to receive data and not send any to the local user, the application rejects the call. If there are fewer than the maximum allowable number of participants, the application accepts the call.

Listing 2-7 Receiving an incoming call

```
/*-------------------------------------------------------------
   DoIncomingCall: handle incoming call event
*-------------------------------------------------------------*/
void DoIncomingCall(MTConferenceEventPtr ce)
{
   MemberPtr      mp;
   OSErr          reason;

   reason = 0;

   if(!(ce->bonus & mtReceiveMediaModeMask))
      reason = mtConnectionRefusedErr;    // no snooping allowed!
   else if(mp = FindMember(0))
      mp->member = ce->who;
   else{
      reason = mtOutOfResourcesErr;

      /* process error */
   }

   MTConferenceReply(gCC, ce->who, reason);
}
```

As its first indication that a remote user wants to establish a connection with the local user, your application receives an incoming call event (an `mtIncomingCallEvent` event) from the conference component. At this time, your application can use various conference component functions to determine whether to acknowledge the incoming call. For example, you might determine whether the remote user should be granted access to the local user, say, by implementing a password or some other authentication scheme. Further, your application might determine the level of protocol support offered by the remote user's system by negotiating message capabilities using the `MTConferenceGetMemberMessageCapabilities` function.

The `DoPhoneRinging` function, shown in Listing 2-8, shows how you might complete an outgoing call after you have called the `MTConferenceCall` function. This function implements a simple password as an authentication device. This function uses the `Requires` function to find out if the remote application requires password authentication.

Listing 2-8 Handling the phone ringing event and negotiating message capabilities

```
/*-----------------------------------------------------------------
   DoPhoneRinging: handle phone ringing event; get password
*-----------------------------------------------------------------*/
void DoPhoneRinging(MTConferenceEventPtr ce)
{
   Handle         ph;

   if((ce->bonus & mtRequiredMessagesModeMask) &&
    Requires(ce->who, PW_CLEAR) &&
    (ph = GetPassword(MTConferenceGetMemberName(gCC, ce->who))))
         MTConferenceSendMessageToMember(gCC, ce->who, ph,
                                         PW_CLEAR, 1);
}

/*-----------------------------------------------------------------
   Requires: message capability negotiation
*-----------------------------------------------------------------*/
Boolean Requires(MTConferenceMember who, OSType type)
{
   MTCapabilitiesHandle ch;
   MTCapabilitiesPtr    cp;
   int                  n;

   ch = MTConferenceGetMemberMessageCapabilities(gCC, who);

   for(n = (*ch)->count, cp = (*ch)->capability; --n >= 0; cp++)
      if(type == cp->type &&
       cp->desires == mtMessageRequiredCapability){
         DisposeHandle((Handle) ch);
         return(true);
      }

   DisposeHandle((Handle) ch);
   return(false);
}
```

When you receive an `mtConferenceReadyEvent` conference component event, your application must respond by activating your conference connection. You do so by calling the `MTConferenceActivateConference` function. This function enables the local user to send media data to the remote users in the conference. The `DoConfReady` function, shown in Listing 2-9, demonstrates how to do this. Note that this function also

handles adding new members to an existing conference by calling the `MTConferenceMerge` function.

Listing 2-9 Activating a conference

```
/*------------------------------------------------------------
   DoConfReady: handle conference ready event
*------------------------------------------------------------*/
void DoConfReady(MTConferenceEventPtr ce)
{
   if(gConference)                     // add to conference
      MTConferenceMerge(gCC, gConference, ce->who);
   else                                // start new conference
      MTConferenceActivateConference(gCC, gConference = ce->who,
                                     gMember[0].controller);
}
```

Any conference member can leave a conference at any time. The conference component detects when a remote user leaves a conference and stops sending media data to that user. Your application receives an `mtConferenceTerminatedEvent` conference component event when the last member leaves a conference. You can then perform any cleanup your application requires. The `DoConfTerminated` function, shown in Listing 2-10, shows some simple conference termination logic.

Listing 2-10 Terminating a conference

```
/*------------------------------------------------------------
   DoConfTerminated: handle conference terminated event
*------------------------------------------------------------*/
void DoConfTerminated(MTConferenceEventPtr ce)
{
   if(gConference == ce->who)
      gConference = 0;
}
```

Before a new member begins participating in a conference, your application receives an `mtMemberJoiningEvent` conference component event. This event tells you that a new member is about to enter the conference. Your application can perform any setup necessary in preparation. The `DoMemberJoining` function, shown in Listing 2-11, handles the simple setup needed by this application.

Listing 2-11 Setting up a new conference member

```
/*------------------------------------------------------------------
   DoMemberJoining: handle member joining event
*------------------------------------------------------------------*/
void DoMemberJoining(MTConferenceEventPtr ce)
{
   MemberPtr   mp;

   if(mp = FindMember(0))
      mp->member = ce->who;
   else
      /* process error */
}
```

When a remote user is capable of exchanging media data with other conference members, your application receives an `mtMemberReadyEvent` conference component event. At this time, you should allocate a controller that can present the remote user's media data to the local user and activate that member. The `DoMemberReady` function, shown in Listing 2-12, shows one way to do this.

Listing 2-12 Preparing to display a new member's media data

```
/*------------------------------------------------------------------
   DoMemberReady: handle member ready event
*------------------------------------------------------------------*/
void DoMemberReady(MTConferenceEventPtr ce)
{
   MemberPtr               mp;
   MTControllerComponent   mtc;


   if(ce->bonus & mtReceiveMediaModeMask){
      if(!(mp = FindMember(ce->who)))
         return;
      mtc = MTConferenceNewPreparedController(gCC, &mp->box,
                     mtWindowVisiblePrepMask +
                     mtControllerVisiblePrepMask +
                     mtEnableSnapshotPrepMask +
                     mtEnableRecordPrepMask,
                     MTConferenceGetMemberName(gCC, mp->member));
      mp->controller = mtc;
   } else
      mtc = 0;
```

```
   MTConferenceActivateMember(gCC, ce->who, mtc);
}
```

When a member leaves a conference, your application receives an `mtMemberTerminatedEvent` conference component event. You should dispose of any resources your application was using to handle that member's connection to the conference. This might include disposing of the controller that was presenting the member's media data to the local user, for example. The `DoMemberTerminated` function, provided in Listing 2-13, shows how this application cleans up after a departing conference member.

Listing 2-13 Cleaning up after a conference member departs

```
/*------------------------------------------------------------
   DoMemberTerminated: handle member terminated event
*------------------------------------------------------------*/
void DoMemberTerminated(MTConferenceEventPtr ce)
{
   MemberPtr   mp;

   if(!(mp = FindMember(ce->who)))
      return;

   if(mp->controller)
      MTConferenceDisposeController(gCC, mp->controller);

   mp->controller = 0;
   mp->member = 0;
}
```

A Conference Without Media Data

This section discusses some of the ways you would change the basic `See.c` program so that it supports conferences that don't carry any media data. Conferences of this type might be useful for text-only services, such as a messaging bulletin board or "chatline."

Besides the fact that the basic application would be completely different, the main differences with respect to how you interact with the conference component fall into two categories: setting the conference mode and using controllers.

Conference Mode Settings

The default conference mode setting accommodates fully connected conferences that carry media data. The `See.c` program uses this default setting, and it sets up a

conference where each participant can exchange both media data and control messages with every other participant.

If you are going to support conferences that do not carry media data, you should set a different conference mode. The following code fragment shows how to set up a conference so that each participant can interact with every other participant, but none can exchange media data with you.

```
MTConferenceSetMode(cc, mtJoinerModeMask+mtShareableModeMask);
```

The mode mask value indicates that others may join the conference (`mtJoinerModeMask`) and that your messages may be shared with any new participants (`mtShareableModeMask`).

In the `See.c` program, you would change the `Start` function to reflect the appropriate conference mode setting (note that the function does not call the `MTConferenceSetMode` function, because the program uses the conference component's default mode setting).

For more information about the `MTConferenceSetMode` function, see "Configuring the Conference Component" beginning on page 2-26.

Conferences Without Controllers

Controller components allow you to send and receive media data over conference connections. Because you are developing an application that does not carry media data, you do not need to use any controllers.

This simplifies some of what the `See.c` program does. For example, you don't need to allocate or dispose of controller components for each of the conference participants—this allows you to remove all of the references to the `MCConferenceNewPreparedController` function (see the `Start` and `DoMemberReady` functions earlier in this section).

In addition, when you start the conference or add a new participant, you do not specify a controller. The following code fragment shows how you might start a conference:

```
MTConferenceActivateConference(ph->cc, // component instance
        ph->conference = ce->who,     // conference identifier
        0);                           // no controller needed
```

To make a new member active, you might do the following after calling the `MTConferenceMerge` function to add the member to the conference:

```
MTConferenceActivateMember(ph->cc,     // component instance
        ce->who,                      // member identifier
        0);                           // no controller needed
```

A Conference With Auxiliary Sources

This section discusses how you might modify your basic conferencing application to include support for auxiliary media sources. Auxiliary media sources allow conference participants to provide more than one stream of media data to a conference. This can be useful when a participant wants to share a movie or some recorded sound with other conference participants.

To illustrate this feature, consider changing your basic conferencing application so that the sound is carried separately from the video media and the participants can turn their sound on and off.

Video-Only Controllers

First, you would need to change how the basic controller is created. In the `Start` function, the `See.c` program sets up your controller so that it sends and receives both audio and video data. In this case, you would allocate a video-only controller:

```
gMember[0].controller = MTConferenceNewPreparedController(gCC,
                            &gWindowBox,
                            mtMediaSourcePrepMask +
                            mtGrabVideoPrepMask +
                            mtWindowVisiblePrepMask +
                            mtControllerVisiblePrepMask +
                            mtEnableSnapshotPrepMask +
                            mtEnableRecordPrepMask,
                            vwTitle);
```

Similarly, in the `DoMemberReady` function, you would allocate a controller to each new participant. In this case, the controller examines the incoming data to determine the type of controller to use:

```
mtc = MTConferenceNewPreparedController(gCC, &gWindowBox,
                            mtAutoPositionPrepMask +
                            mtWindowVisiblePrepMask +
                            mtControllerVisiblePrepMask +
                            mtEnableSnapshotPrepMask +
                            mtEnableRecordPrepMask,
                            MTConferenceGetMemberName(gCC,
                               mp->member));
```

Attaching and Detaching Your Auxiliary Source

In this example, you are allowing the users to turn the sound on and off. When the sound is on, it is carried in the conference as an auxiliary media source. As a result, you need to be able to turn the auxiliary source on and off.

To turn the source on, you must create a source controller and then assign it to the conference as an auxiliary. To do so, you might do the following:

```
gVoice = MTConferenceNewPreparedController(gCC, &gVoiceBox,
                            mtMediaSourcePrepMask +
                            mtGrabAudioPrepMask +
                            mtAuxiliaryClosePrepMask +
                            mtWindowVisiblePrepMask +
                            mtControllerVisiblePrepMask +
                            mtEnableRecordPrepMask,
                            awTitle);

myAux = MTConferenceAttachAuxiliarySource(gCC,
                            gConference,
                            gVoice,
                            awTitle);
```

This code fragment allocates a sound-only controller, identified by the `gVoice` parameter, and then attaches that controller to the conference as an auxiliary media source.

When you turn off your sound, you must remove your auxiliary source from the conference:

```
MTConferenceDetachAuxiliarySource(gCC, myAux);
```

Terminating Incoming Auxiliary Sources

Finally, when you have detached your auxiliary source or the conference is ending, your application receives an `mtAuxiliaryTerminatedEvent` event telling you that the corresponding auxiliary source has been terminated. Your application should handle that event appropriately:

```
void DoAuxTerminated(void)
{
   MTConferenceDisposeController(gCC, gVoice);
   gVoice = 0;
}
```

A Conference Server

This section discusses some of the changes you might make to your application if you wanted to develop a send-only server application. Such an application might be useful for carrying an internal television broadcast or for broadcasting some stored video material. The conference component automatically takes advantage of the underlying network's inherent multicast facilities when sending the server's data.

The server application considered here differs from the basic videophone application in three main ways:

- First, it does not place calls. Users must call the server in order to join the conference.
- Second, its conference mode setting establishes a send-only conference.
- Third, it maintains only media connections with conference participants.

By requiring new participants to call your server, you are able to remove all of the program logic associated with establishing outgoing calls. This greatly simplifies the program's basic logic, eliminating a number of the functions in the `See.c` program.

Server Mode Settings

The default conference mode setting accommodates fully connected conferences that carry media data. If you are going to create a send-only server, you need to use a different conference mode. The following code fragment shows how to set up a conference so that your server sends media data and does not place outgoing calls:

```
MTConferenceSetMode(cc, mtSendMediaModeMask+mtShareableModeMask);
```

The mode mask value indicates that you will send media but not receive it (`mtSendMediaModeMask`) and that your messages may be shared with any new participants (`mtShareableModeMask`). The joiner attribute (`mtJoinerModeMask`) is not used because you do not want to allow conference participants to exchange data among themselves.

In the `See.c` program, you would change the `Start` function to reflect the appropriate conference mode setting (note that the function does not call the `MTConferenceSetMode` function, because the program uses the conference component's default mode setting).

For more information about the `MTConferenceSetMode` function, see "Configuring the Conference Component" beginning on page 2-26.

Terminating Control Connections

Once a user has established a call with the server, the server terminates the control connection in order to conserve system resources. The media connection remains, however, and is the conduit for the broadcast media data.

Establishing a new conference or adding a new participant becomes very straightforward then:

```
void DoConfReady(MTConferenceEventPtr ce)
{
   if(gConference)                     // new participant
      MTConferenceMerge(gCC, gConference, ce->who);
   else                                // new conference
      MTConferenceActivateConference(gCC, gConference = ce->who,
                        gMember[0].controller);
```

```
}

   void
DoMemberReady(
   MTConferenceEventPtr    ce
){
   MTConferenceActivateMember(gCC, ce->who, 0);
   MTConferenceDetachMember(gCC, ce->who);
}
```

The `MTConferenceDetachMember` function terminates the control channel while keeping the media channel open. When a participant decides to leave the conference, they simply terminate the conference at their end. Since the server is using multicast facilities to send its media data, it need not know when an individual participant leaves.

Conference Component Reference

This section provides detailed information about the functions supported by the conference component and the constants and data structures you use when working with those functions.

Constants

This section discusses important network component constants.

Conference Component Type and Subtype Values

The following constants identify the Component Manager component type and subtype values you must use to work with the conference component:

```
#define  kMTConferenceType              'conf'
#define  kMTMovieTalkSubType            'mtlk'
```

Functions

This section discusses the functions supported by the conference component. This section has been divided into the following topics:

- "Configuring the Conference Component" discusses functions that allow you to prepare the conference component to place or accept calls.
- "Managing Conference Events" discusses the events your application must support in order to use the conference component.

- "Managing Calls" describes how you place, accept, and terminate conference calls.
- "Activating Conferences" tells you how to start media data flowing across a new conference.
- "Working With Controllers" discusses the functions that allow you to create and manage controllers using the conference component.
- "Working With Auxiliary Sources" describes how to attach and detach auxiliary sources.
- "Exchanging Messages" discusses the functions that allow you to exchange control messages over a conference.
- "Getting Conference Information" tells you how to retrieve useful information about a conference and its members.

Configuring the Conference Component

The conference component provides a number of functions that allow you to prepare the component to place or receive conference calls. This section discusses those functions.

Use the `MTConferenceSetMode` function to establish the operating mode for any conferences you start or join. The `MTConferenceSetCallTimeout` function allows you to specify how long to wait for an outgoing call to complete. The `MTConferenceListen` function registers you on the network, allowing other potential conference members to call you. The `MTConferenceGetRegisteredNames` function retrieves the names under which you have been registered.

MTConferenceSetMode

The `MTConferenceSetMode` function sets your operating mode for all conferences you create subsequently.

```
pascal ComponentResult MTConferenceSetMode
                            (MTConferenceComponent cc,
                            MTConferenceModeFlags defaultMode);
```

`cc` Specifies the conference component instance for the operation. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`defaultMode`
Indicates the conference mode setting you desire. You provide a mask that specifies the settings for the following mode attributes (note that you may set more than one flag to 1):

`mtSendMediaModeMask`
: Indicates that you intend to send media data in your conferences. Set this flag to 1 if you want to be able to send media data. You must set this flag to 1 if you intend to provide media data to the conference.

`mtReceiveMediaModeMask`
: Indicates that you intend to receive media data in your conferences. Set this flag to 1 if you want to be able to receive media data from the conference.

`mtShareableModeMask`
: Indicates that you are willing to share your conference media data with new conference members. Set this flag to 1 if you want to allow new members to receive your conference data.

`mtJoinerModeMask`
: Indicates that all members of the conference may exchange messages with one another. Set this flag to 1 if all conference members are allowed to interact. Setting this flag to 0 results in a broadcast-type conference, where one member sends media data to other conference members, but the individual members do not exchange any media data among themselves.

DESCRIPTION

Use the `MTConferenceSetMode` function to specify the operating mode for any conferences you create subsequently. By default, the conference component establishes conferences that are fully media-data capable, shareable, and joinable (that is, the mode is set to `mtSendMediaModeMask + mtReceiveMediaModeMask + mtShareableModeMask + mtJoinerModeMask`). If you want to assert different conference characteristics, you must call this function before you call the `MTConferenceListen` function.

Any mode that results from combining the flag values is valid. For example, setting just the `mtJoinerModeMask` and `mtShareableModeMask` flags to 1 produces fully connected conferences that don't transmit any media data. While such a conference wouldn't carry any media data, its members could exchange messages using the `MTConferenceSendMessageToConference` function to implement a text-only "chatline" service.

A given conference component instance supports only a single conference mode at a time. If your application is managing different types of conferences with different modes, say, playing stored movies as well as managing multiparty conferences, you need to open different instances of the conference component to manage those conferences.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter value

MTConferenceSetCallTimeout

The `MTConferenceSetCallTimeout` function specifies the amount of time the conference component lets an outgoing call ring before considering it unanswered.

```
pascal ComponentResult MTConferenceSetCallTimeout
                        (MTConferenceComponent cc,
                        short callTimeout);
```

`cc` Specifies the conference component instance for the operation.

`callTimeout`
The number of ticks (1/60 second) to wait before giving up on an outgoing call.

DESCRIPTION

Use the `MTConferenceSetCallTimeout` function to specify how long you want to let your outgoing calls "ring" before hanging up. The default timeout is 30 seconds (corresponding to a `callTimeout` parameter value of 1800).

Your conference component passes this timeout value to the remote end during the negotiation process (the remote end gets this information in an `mtIncomingCallEvent` event). The remote end might use this value to select how long to wait for a person to answer a call before invoking some sort of an automated answering capability, such as an "answering machine" function.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter value

SEE ALSO

For more information about conference component events, see "Managing Conference Events" beginning on page 2-31.

MTConferenceListen

The `MTConferenceListen` function prepares your conference component to receive incoming calls. You must call this function exactly once before calling any other

conference component functions (except for the `MTConferenceSetMode`, `MTConferenceSetCallTimeout`, and `MTConferenceSetMessageCapabilities` functions).

```
pascal ComponentResult MTConferenceListen
                        (MTConferenceComponent cc,
                        MTCString63 userName,
                        MTCString32 serviceName,
                        MTCString serviceTypesList);
```

`cc` Specifies the conference component instance for the operation.

`userName` The user name by which you wish to be known, expressed as a C string. This name is visible to other conference members. If the name is not entirely in the Roman script, you should create the name using the `MTConferenceNameFromRString` function.

`serviceName`
A C string containing the name of the service you want to support. You determine the name that identifies the service. The name depends on the type of connection or service you are providing and should be meaningful to and recognizable by your application and other potential conference members.

Most network types (other than AppleTalk) ignore this field. In other cases, the value of this field corresponds to your transport's registered name (see the "Transport Components" chapter for more information about registerting names).

`serviceTypesList`
A C string containing entries for all the networks from which you wish to accept conference callers.

DESCRIPTION

Use the `MTConferenceListen` function to prepare your conference component to accept incoming calls. You must prepare your conference component to accept incoming calls before you can start a conference. You must do this even if you do not expect to receive incoming calls—the conference component uses this function to initialize much of its common call-handling support.

When you call this function, you specify all of the networks from which you are willing to accept calls. The conference component proceeds to register you on those networks (doing whatever is appropriate in those various network contexts). After each registration attempt, the conference component sends you an `mtListenerStatusEvent` event. This event tells you whether the registration attempt was successful. Once you have registered yourself, you can use the `MTConferenceGetRegisteredNames` function to retrieve a list of the names under which you have been registered.

You specify the networks from which you can accept calls using the `serviceTypesList` parameter. This parameter specifies a C string that consists of one

or more service type entries. Each entry specifies a transport interface and a service. The transport interface identifies the network medium over which the conference takes place. The service type identifies a type of conference that is meaningful to potential participants. For more information on service types, see "Service Types" beginning on page 2-5.

Note that a given conference component instance can support only a single service type at a time. However, you can register the same (or equivalent) service type on more than one network at one time.

You specify a transport interface by concatenating the component subtype values that identify the transport component and the network component that support the network. For example, a transport interface value that identifies a MovieTalk transport connection maintained over an AppleTalk network would be `"mtlkatlk"`. Similarly, MovieTalk over TCP/IP would be represented by `"mtlktcpi"`.

You specify the service information in a network-specific manner. On AppleTalk networks, services are identified by name. So, to specify an AppleTalk service, you would include the service's name (for example, `"VideoPhone"`). Other networks use other formats. For example, on TCP/IP networks services are typically represented by socket numbers. So, you would provide the number of the network socket on which you are accepting calls (say, `'458'`). However, in all cases, you must provide the specification as ASCII characters.

The service list is generally a constant for any given application and can be stored in a resource. Entries are terminated by new-line characters (`'\n'`). The two values are separated from one another by a tab character (`'\t'`). Here is a Rez definition for the examples from this discussion:

```
type 'srvt' {
   cstring;
};

resource 'srvt' (9911, "Service Types") {
   "mtlkatlk\tVideoPhone\nmtlktcpi\t458\n"
};
```

RESULT CODES

`noErr`	0	No error
`mtNoListenersErr`	–7743	Could not open any network type in your service list

SEE ALSO

For more information about conference component events, see "Managing Conference Events" beginning on page 2-31.

The `MTConferenceNameFromRString` function is discussed on page 2-62.

The `MTConferenceGetRegisteredNames` function is described next.

For more information about the `MTConferenceSetMode` function, see page 2-26.

The `MTConferenceSetCallTimeout` function is described on page 2-28.

See page 2-57 for more information about the `MTConferenceSetMessageCapabilities` function.

MTConferenceGetRegisteredNames

You can use the `MTConferenceGetRegisteredNames` function to retrieve a list of all the names you have successfully registered by calling the `MTConferenceListen` function.

```
pascal MTNameListPtr MTConferenceGetRegisteredNames
                        (MTConferenceComponent cc);
```

`cc` Specifies the conference component instance for the operation.

DESCRIPTION

Use this function to retrieve the names under which you have been registered to receive incoming conference calls. The function returns a structure containing an array of zero or more network-specific names. Each name identifies how you are presented on a given network. Note that, because the conference component works with a number of different networks, each of which may construct its identifiers differently, the various names may not appear similar to one another and may not correspond exactly to the way you specified your name when you called the `MTConferenceListen` function.

The `MTConferenceGetRegisteredNames` function returns a pointer to an `MTNameList` structure. You are responsible for disposing of the `MTNameList` structure when you are done with it; call the Memory Manager's `DisposePtr` function.

SEE ALSO

The `MTConferenceListen` function is described on page 2-28.

For a description of the `MTNameList` structure, see the "Browser Components" chapter, elsewhere in this book.

Managing Conference Events

As is the case with most Macintosh programs, conference components and the applications that use them are state-driven based on events. For example, conference components send events that allow your application to track the status of incoming and outgoing calls. Use the `MTConferenceGetNextEvent` function to retrieve the next conference event. This function returns the event information in an event structure. This structure is defined as follows:

```
struct MTConferenceEvent {
   MTConferenceEventType      what;
   OSErr                      err;
   long                       who;
   long                       bonus;
   long                       reserved;
   Handle                     surprise;
};
typedef struct MTConferenceEvent MTConferenceEvent,
*MTConferenceEventPtr;
```

Field descriptions

`what`	Indicates the type of conference event. These types are discussed later in this section.
`err`	Indicates an error, if any.
`who`	Meaning and use depend upon the event type.
`bonus`	Meaning and use depend upon the event type.
`reserved`	Meaning and use depend upon the event type.
`surprise`	Contains a handle to some event-specific data. The meaning and use of the data depend upon the event type.

EVENT TYPES

The `what` field of the event structure always indicates the type of event being reported, while the meaning of the other fields depends upon the particular event being reported. The `surprise` field, if not set to `nil`, contains a handle to some event-specific data. Your application is responsible for disposing of this handle even if you ignore the event. All unused fields are set to `nil` and should be ignored.

The conference component supports the following events:

`mtListenerStatusEvent`

This event provides you with the result of your request to register on a given network. You register by calling the `MTConferenceListen` function (discussed on page 2-28). Note that you may also receive this event if the network fails.

The following event structure fields are used:

`err`	Contains an error code. This field is set to `noErr` if the registration was successful.
`bonus`	Specifies the network type. This field contains the component subtype value of the network component you specified.
`reserved`	Specifies the transport type. This field contains the component subtype value of the transport component you specified.

`mtIncomingCallEvent`

This event tells you about an incoming conference call. In response to this event, you must accept or refuse the call using the `MTConferenceReply` function (see page 2-40 for more information). The following event structure fields are used:

`who` Identifies the calling party. This field contains an `MTConferenceMember` value.

`bonus` Contains the call's mode from your point of view. For example, in this field the `mtReceiveMediaModeMask` flag means that the other end wants to send data to you.

The value is derived by combining your desires with those expressed by the remote end. For example, the `mtReceiveMediaModeMask` flag will be set to 1 only if you allow reception (your mode value includes the `mtReceiveMediaModeMask` flag) and the caller wishes to send media (the `mtSendMediaModeMask` flag is set to 1 on the far end).

Besides the mode flags you can set using the `MTConferenceSetMode` function, there are two additional flags. These flags indicate that the remote end has special message-handling requirements. The `mtDesiredMessagesModeMask` flag indicates that the remote end has some messages it expects you to send at the beginning of the conference; the `mtRequiredMessagesModeMask` flag indicates that there are some messages that you must send. If either of these flags is set to 1, you can use the `MTConferenceGetMemberMessageCapabilities` function to get more information about the remote end's needs (see page 2-59 for more information about this function).

`reserved` Contains the caller's call timeout value. See the discussion of the `MTConferenceSetCallTimeout` function on page 2-28 for more information.

`mtMemberJoiningEvent`

This event lets you know that a caller is joining a conference because another party has merged two conferences. This event is similar to the `mtIncomingCallEvent` event, but you have no choice over accepting or rejecting the call. You use the `MTConferenceReply` function to accept the call (see page 2-40 for more information). The following event structure fields are used:

`who` Identifies the new conference member or the new auxiliary source. This field contains an `MTConferenceMember` value.

`bonus` Contains the call's mode from your point of view. For example, in this field the `mtReceiveMediaModeMask` flag means that the other end wants to send data to you.

The value is derived by combining your desires with those expressed by the remote end. For example, the `mtReceiveMediaModeMask` flag will be set to 1 only if you allow reception (your mode value includes the `mtReceiveMediaModeMask` flag) and the caller wishes to send media (the `mtSendMediaModeMask` flag is set to 1 on the far end).

`reserved` Identifies a new auxiliary source, if there is one. This field contains an `MTConferenceMember` value that, if not set to `nil`, indicates that this is an auxiliary source originating from the identified member.

`mtMemberMovedEvent`

Due to the merger of two conferences, a member now belongs to a different conference and is using a different conference identifier. If you are using the conference identifier in your application, you should obtain a new one for this member. The following event structure field is used:

`who` Identifies the conference member who has left. This field contains an `MTConferenceMember` value.

`mtMessageArrivedEvent`

You have been sent a message over the reliable control channel. The following event structure fields are used:

`who` Identifies the conference member who sent you the message. This field contains an `MTConferenceMember` value.

`bonus` Contains the version number of the message as specified by the sender.

`reserved` Contains the message type identifier.

`surprise` Contains a handle to the message data itself. The conference component places no restrictions on the contents of the handle. You may determine the message length using the `GetHandleSize` function. You must dispose of the handle when you have finished with it; use the Memory Manager's `DisposeHandle` function.

`mtPhoneRingingEvent`

A call you have placed using the `MTConferenceCall` function has progressed to the point where you may exchange messages. You may do so, for example, for authentication purposes prior to accepting or rejecting the call. At this point you may use the `MTConferenceGetMemberName` and `MTConferenceGetMemberMessageCapabilites` functions. The following event structure fields are used:

`who` Contains the `MTConferenceMember` value returned when you called the `MTConferenceCall` function.

`bonus` Contains the call's mode from your point of view. For example, in this field the `mtReceiveMediaModeMask` flag means that the other end wants to send data to you.

The value is derived by combining your desires with those expressed by the remote end. For example, the `mtReceiveMediaModeMask` flag will be set to 1 only if you allow reception (your mode value includes the `mtReceiveMediaModeMask` flag) and the caller wishes to send media (the `mtSendMediaModeMask` flag is set to 1 on the far end).

Besides the mode flags you can set using the `MTConferenceSetMode` function, there are two additional flags. These flags indicate that the remote end has special message-handling requirements. The `mtDesiredMessagesModeMask` flag indicates that the remote end has some messages it expects you to send at the beginning of the conference; the `mtRequiredMessagesModeMask` flag indicates that there are some messages that you must use. If either of these flags is set to 1, you can use the `MTConferenceGetMemberMessageCapabilities` function to get more information about the remote end's needs (see page 2-59 for more information about this function).

You should be sure to check the `mtRequiredMessagesModeMask` flag to determine if an authentication exchange is required.

`mtConferenceReadyEvent`

This event tells you that a conference has been established and the media streams may be started. You must call the `MTConferenceActivateConference` function in response to this event. The following event structure fields are used:

`who`	Identifies the new conference. This field contains an `MTConferenceToken` value.
`bonus`	Contains the conference's mode from your point of view. For example, in this field the `mtReceiveMediaModeMask` flag means that the other end wants to send data to you.

`mtConferenceTerminatedEvent`

All members in a conference have departed and the conference is being terminated. The following event structure fields are used:

`who`	Identifies the conference. This field contains an `MTConferenceToken` value.
`bonus`	Identifies the QuickTime Conferencing controller you attached to this conference using the `MTConferenceActivateConference` function (see page 2-44 for more information). The controller is now detached, and you may reuse or dispose of it. To dispose of the controller, either close the component or call the `MTConferenceDisposeController` function,

depending upon how you created the controller. If you did not assign a controller to the conference, the conference component sets this field to `nil`.

`mtAuxiliaryTerminatedEvent`

A conference's auxiliary source has been terminated. The following event structure fields are used:

`who` Identifies the auxiliary source. This field contains an `MTAuxiliaryToken` value.

`bonus` Identifies the QuickTime Conferencing controller you attached to this source using the `MTConferenceAttachAuxiliarySource` function (see page 2-52 for more information). The controller is now detached, and you may reuse or dispose of it. To dispose of the controller, either close the component or call the `MTConferenceDisposeController` function, depending upon how you created the controller. If you did not assign a controller to the source, the conference component sets this field to `nil`.

`mtConferenceNameChangedEvent`

Because another party has merged conferences, the name of this conference has changed. The following event structure field is used:

`who` Identifies the conference. This field contains an `MTConferenceToken` value.

`mtMemberReadyEvent`

This event informs you that a member is ready to receive a media stream. You must call the `MTConferenceActivateMember` function in response to this event (see page 2-45 for more information). The following event structure fields are used:

`who` Identifies the member. This field contains an `MTConferenceMember` value.

`bonus` Contains the conference's mode from your point of view, as it pertains to this member. For example, in this field the `mtReceiveMediaModeMask` flag means that the other end wants to send data to you.

The value is derived by combining your desires with those expressed by the remote end. For example, the `mtReceiveMediaModeMask` flag will be set to 1 only if you allow reception (your mode value includes the `mtReceiveMediaModeMask` flag) and the caller wishes to send media (the `mtSendMediaModeMask` flag is set to 1 on the far end).

`mtMemberTerminatedEvent`

This event tells you that a member has left the conference. You may still call the `MTConferenceGetMemberName` function in order to identify the member completely, but the conference component invalidates the

member identifier the next time you call the `MTConferenceGetNextEvent` function. The following event structure fields are used:

`who` Identifies the member who is leaving. This field contains an `MTConferenceMember` value.

`bonus` Identifies the QuickTime Conferencing controller you attached to this member using the `MTConferenceActivateMember` function (see page 2-45 for more information). The controller is now detached, and you may reuse or dispose of it. To dispose of the controller, either close the component or call the `MTConferenceDisposeController` function, depending upon how you created the controller. If you did not assign a controller to the member, the conference component sets this field to `nil`.

`mtRefusedEvent`

The remote end has refused your call. The following event structure fields are used:

`err` Gives the reason the remote end refused your call. This is the value that the remote application passed to the `MTConferenceReply` function.

`who` Identifies the conference member. This field contains an `MTConferenceMember` value.

`mtFailedEvent`

A call has failed as a result of a network communcations error. A separate `mtMemberTerminatedEvent` event will follow. The following event structure fields are used:

`err` Gives the system error code corresponding to the transport component error.

`who` Identifies the conference member you were calling. This field contains an `MTConferenceMember` value.

`mtSnapshotTakenEvent`

The user has captured a conference image using a controller you acquired from the `MTConferenceNewPreparedController` function (see page 2-46 for more information). You receive this event in response to a controller component's `mtControllerActionSnapshotData` action. The following event structure fields are used:

`who` Identifies the source of the controller's data. This field contains an `MTConferenceToken` value if the controller is attached to a conference, contains an `MTConferenceMember` value if it is attached to a member, or is set to `nil` if the controller is unattached.

`bonus` Contains the identifier of the controller that generated the image.

`surprise` Contains a picture handle. You must dispose of this handle; use the Memory Manager's `DisposeHandle` function.

`mtMovieRecordedEvent`

The user has recorded a movie from the live media data using a controller you acquired from the `MTConferenceNewPreparedController` function (see page 2-46 for more information). The following event structure fields are used:

`err`	Gives the system error code corresponding an error that occurred. In some cases, you may still get a partially full movie file.
`who`	Identifies the source of the controller's data. This field contains an `MTConferenceToken` value if the controller is attached to a conference, contains an `MTConferenceMember` value if it is attached to a member, or is set to `nil` if the controller is unattached.
`bonus`	Contains the identifier of the controller that generated the movie.
`surprise`	Contains a handle to an `FSSpec` structure that identifies the movie file. This file is currently in the Temporary Items folder. You should move it somewhere permanent and give it another name. If you aren't interested in saving the file, you must delete it. You are responsible for disposing of this handle; use the Memory Manager's `DisposeHandle` function.

MTConferenceGetNextEvent

The `MTConferenceGetNextEvent` function checks to see if any conference events need your attention and retrieves the next event, if one is queued.

```
pascal ComponentResult MTConferenceGetNextEvent
                            (MTConferenceComponent cc,
                            MTConferenceEventPtr theEvent);
```

`cc`	Specifies the conference component instance for the operation.
`theEvent`	A pointer to an `MTConferenceEvent` structure you have allocated to receive the next conference event. The conference component returns these events in the order they are received.

DESCRIPTION

You must call the `MTConferenceGetNextEvent` function frequently in order to receive events from the conference component. Your application is responsible for managing and responding to these events appropriately. The events supported by the conference component are discussed in "Managing Conference Events" beginning on page 2-31.

The `MTConferenceGetNextEvent` function returns a Boolean value indicating whether it has returned any event information. The function sets the returned value to

`true` if it has placed an event in the location you specify with the `theEvent` parameter; otherwise, it returns `false`.

Note
In order for the conference component to function properly, you must set the `doesActivateOnFGSwitch` flag in your application's `SIZE (-1)` resource to 0. ◆

Managing Calls

The conference component provides a number of functions that allow you to place, answer, and end conference calls. This section discusses those functions.

Use the `MTConferenceCall` function to call a member of a conference or to establish a new conference. The `MTConferenceReply` function allows you to answer or refuse an incoming call. You can combine two conferences by calling the `MTConferenceMerge` function. You end your involvement in a conference by calling the `MTConferenceTerminate` function. The `MTConferenceDetachMember` function is a special-purpose function that allows conference servers to shut down a conference's control channel while keeping the media channel open.

MTConferenceCall

The `MTConferenceCall` function calls a named network entity, thereby establishing a new conference.

```
pascal MTConferenceMember MTConferenceCall
                            (MTConferenceComponent cc,
                            MTCString63 confName,
                            MTNamePtr whoToCall);
```

`cc` Specifies the conference component instance for the operation.

`confName` The conference name. This is the name you wish to give to this conference, expressed as a C string. Use the `MTConferenceNameFromRString` function to construct this value if the conference name is not in the Roman script.

`whoToCall` The name of the user you wish to call. In general, you retrieve user names using a browser component.

DESCRIPTION

This function allows you to establish a conference.

The `MTConferenceCall` function returns an `MTConferenceMember` value. This value identifies the call in progress. Your application monitors the progress of this call by processing the conference component events that result.

The `MTConferenceCall` function will fail and return a value of 0 if the name you are attempting to call is on a network on which you are not registered. You register by calling the `MTConferenceListen` function. Other network errors encountered while establishing the call will be returned later using the `mtFailedEvent` conference event.

SEE ALSO

For more information about conference component events, see "Managing Conference Events" beginning on page 2-31.

The `MTConferenceNameFromRString` function is discussed on page 2-62.

See page 2-28 for information about the `MTConferenceListen` function.

MTConferenceReply

The `MTConferenceReply` function allows you to answer or refuse an incoming call.

```
pascal ComponentResult MTConferenceReply
                        (MTConferenceComponent cc,
                        MTConferenceMember callingParty,
                        OSErr response);
```

`cc` Specifies the conference component instance for the operation.

`callingParty`
Identifies the calling party. You obtain this value from the `mtIncomingCallEvent` conference event.

`response` Indicates whether you want to accept or refuse the call. Use a value of 0 to accept the call. Any nonzero value refuses the call. The conference component passes this value to the calling party in an `mtRefusedEvent` event.

DESCRIPTION

When your application receives an `mtIncomingCallEvent` event, it must call `MTConferenceReply` to answer or refuse the call. Before calling this function, your application may first exchange data using the `MTConferenceSendMessageToMember` function to gather information it needs to decide whether to accept the call. This information may include authentication data, such as passwords or accounting codes. Your application may also send a message giving a reason for refusing the call (for example, "I'm out to lunch until 3:15"). You can use the `MTConferenceSetMessageCapabilities` function to indicate that your application requires the exchange of certain messages before accepting a call.

RESULT CODES

`noErr`	0	No error

SEE ALSO

For more information about conference component events, see "Managing Conference Events" beginning on page 2-31.

The `MTConferenceSendMessageToMember` function is discussed on page 2-55.

See page 2-57 for information about the `MTConferenceSetMessageCapabilities` function.

MTConferenceMerge

The `MTConferenceMerge` function allows you to merge two existing conferences.

```
pascal ComponentResult MTConferenceMerge
                        (MTConferenceComponent cc,
                        MTConferenceToken staysAround,
                        MTConferenceToken goesAway);
```

`cc` Specifies the conference component instance for the operation.

`staysAround`
Identifies the conference that survives the operation.

`goesAway` Identifies the conference that is merged into the surviving conference. All the members of this conference are moved to the conference you specify with the `staysAround` parameter.

DESCRIPTION

The `MTConferenceMerge` function is the only way to create multiparty conferences. It does so by combining two existing single- or multiparty conferences.

You will get `mtMemberMovedEvent` events for each member of the conference you specify with the `goesAway` parameter, followed by an `mtConferenceTerminatedEvent` event for the conference itself.

If the `mtJoinerModeMask` flag is set to 1 in the destination conference's mode, the conference component establishes calls between all shareable participants in the new conference. Conversely, a server might merge calls together to take advantage of a network multicast feature and thus save bandwidth, but not connect the customers to one another by setting the `mtJoinerModeMask` flag to 0.

RESULT CODES

`noErr`	0	No error

SEE ALSO

For more information about conference component events, see "Managing Conference Events" beginning on page 2-31.

For more information about conference modes, see the discussion of the `MTConferenceSetMode` function on page 2-26.

MTConferenceTerminate

You use the `MTConferenceTerminate` function to withdraw from a conference.

```
pascal ComponentResult MTConferenceTerminate
                        (MTConferenceComponent cc,
                        MTConferenceToken theConference)
```

`cc` Specifies the conference component instance for the operation.

`theConference`
Identifies the conference you want to leave.

DESCRIPTION

Your calls to all members in this conference are disconnected, though connections between other members of the conference remain unaffected. You will receive `mtMemberTerminatedEvent` events for each member of the conference, `mtAuxiliaryTerminatedEvent` events for any auxiliary sources attached to this conference, and an `mtConferenceTerminatedEvent` event for the conference itself.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

SEE ALSO

For more information about conference component events, see "Managing Conference Events" beginning on page 2-31.

MTConferenceDetachMember

The `MTConferenceDetachMember` function is used by send-only servers to detach the control channels of conference members while those members continue to receive media data. This releases system resources consumed by the control channels.

```
pascal ComponentResult MTConferenceDetachMember
                            (MTConferenceComponent cc,
                            MTConferenceMember theMember);
```

`cc`	Specifies the conference component instance for the operation.
`theMember`	Specifies the conference member to detach.

DESCRIPTION

If the detach operation is successful, you will receive an `mtMemberTerminatedEvent` event after the control channel has been released. You may not send messages to a detached member.

RESULT CODES

`noErr`	0	No error
`mtNotDetachableErr`	–7746	Member not detachable because not on a network that supports multicast transmission

SEE ALSO

For more information about conference component events, see "Managing Conference Events" beginning on page 2-31.

Activating Conferences

After you have established a conference, you may then proceed to activate the media channel by enabling the flow of media data. You do so by attaching controllers to the conference. The `MTConferenceActivateConference` function allows you to send media data to a conference. The `MTConferenceActivateMember` function allows you to receive media data from a member of the conference.

The conference component provides the `MTConferenceNewPreparedController` function that creates a controller for you. This function is discussed in "Working With Controllers" beginning on page 2-46. Alternatively, you can create and prepare your own controllers, by working with other QuickTime Conferencing components.

MTConferenceActivateConference

The `MTConferenceActivateConference` function activates the send side of your conference connection.

```
pascal ComponentResult MTConferenceActivateConference
                        (MTConferenceComponent cc,
                        MTConferenceToken theConference,
                        MTControllerComponent selfController);
```

`cc` Specifies the conference component instance for the operation.

`theConference`
Identifies the conference. You obtain this `MTConferenceToken` value from the `mtConferenceReadyEvent` event.

`selfController`
Specifies the controller that provides the media data you are sending to the conference. You specify a controller only if you are a media sender (check the mode value in the `mtConferenceReadyEvent` event you received). If you are not a sender, set this parameter to `nil`.

DESCRIPTION

Soon after receiving an `mtConferenceReadyEvent` event, your application must call the `MTConferenceActivateConference` function to activate the send side of your conference. In response, the conference component attaches the controller you specify to the conference and starts sending the controller's data to other conference members. The conference component takes care of attaching the transport, stream director, and other low-level components and manages the stream-negotiation process.

After initially activating a conference, you may subsequently suspend transmission of media data by specifying `nil` for the `selfController` parameter. You may need to do this to answer another call if you have only a single video window, for example. You may later resume the original call by respecifying the original controller.

You obtain a controller by calling the `MTConferenceNewPreparedController` function or by creating and configuring your own controller using other QuickTime Conferencing components.

RESULT CODES

`noErr`	0	No error
`mtModeErr`	–7745	Controller specified when not needed or not specified when one was required

SEE ALSO

For more information about conference component events, see "Managing Conference Events" beginning on page 2-31.

The `MTConferenceNewPreparedController` function is discussed on page 2-46.

MTConferenceActivateMember

The `MTConferenceActivateMember` function allows you to activate the receive side of your conference connection.

```
pascal ComponentResult MTConferenceActivateMember
                        (MTConferenceComponent cc,
                        MTConferenceMember theMember,
                        MTControllerComponent memberController);
```

`cc` Specifies the conference component instance for the operation.

`theMember` Identifies the remote member who is to provide the data you will receive. You obtain the `MTConferenceMember` value from the `mtMemberReadyEvent` event.

`memberController`
Specifies the controller that is to receive the media data you receive from this member. You specify a controller only if you are to receive data from this member (check the mode value in the `mtMemberReadyEvent` event you received). If you are not going to receive media data from this member, set this parameter to `nil`.

DESCRIPTION

After receiving an `mtMemberReadyEvent` event, your application must call the `MTConferenceActivateMember` function to activate the receive side of your conference. The conference component sets up the necessary facilities to allow you to receive data from the member and begins sending data to this member, if appropriate.

You obtain a controller by calling the `MTConferenceNewPreparedController` function or by creating and configuring your own controller using other QuickTime Conferencing components.

RESULT CODES

`noErr`	0	No error
`mtModeErr`	–7745	Controller specified when not needed or not specified when one was required

SEE ALSO

For more information about conference component events, see “Managing Conference Events” beginning on page 2-31.

The `MTConferenceNewPreparedController` function is discussed on page 2-46.

Working With Controllers

In order to send or receive media data, you must use controllers. However, these controllers need not be visible to the user.

The conference component provides a number of functions that allow you to prepare controllers for use with a conference. The `MTConferenceNewPreparedController` function creates a new controller for you. The `MTConferenceDisposeController` function disposes of a controller when you are done with it. The `MTConferenceSetDefaultWindowProcID` function lets you control the type of window in which the controller is created. The `MTConferenceSetPreparationDefaults` function allows you to control the types of components that the conference component uses when it builds a controller for you. The `MTConferenceSetDefaultActionFilter` function allows you to assign an action filter function to controllers you create.

Alternatively, you may acquire controllers yourself, using the controller component. However, if you use the `MTConferenceNewPreparedController` function, the conference component handles many of the user interactions (such as taking snapshots and recording movies) and administrative details (such as handling controller events for your application) for you.

MTConferenceNewPreparedController

The `MTConferenceNewPreparedController` function creates and configures a new controller for you.

```
pascal MTControllerComponent MTConferenceNewPreparedController
                                (MTConferenceComponent cc,
                                const Rect *windowRect,
                                MTConferencePrepFlags prepFlags,
                                MTCString windowTitle);
```

`cc` Specifies the conference component instance for the operation.

`windowRect` A pointer to a rectangle giving the global coordinates specifying where you would like the controller's visible window placed. The controller will fill this rectangle.

`prepFlags` Flags indicating the controller features you are requesting. The following flags are defined—all these options are made active by setting the appropriate flag to 1 (you may set more than one flag to 1):

`mtMediaSourcePrepMask`
Indicates whether the controller supplies or recieves data. Set this flag to 1 for controllers that supply data and to 0 for controllers that receive data. Use controllers that supply data with the `MTConferenceActivateConference` and

`MTConferenceAttachAuxiliarySource` functions. Use controllers that receive data with the `MTConferenceActivateMember` function.

`mtGrabVideoPrepMask`
Indicates that you intend to capture video using this controller. This instructs the conference component to attach a sequence grabber component to the controller and to open a video channel.

`mtGrabAudioPrepMask`
Indicates that you intend to grab audio from this controller. This instructs the conference component to attach a sequence grabber component to the controller and to open an audio channel.

`mtGrabOtherPrepMask`
Indicates that you intend to grab media data, but that you will prepare the appropriate sequence grabber channels. The conference component attaches a sequence grabber component to the controller, but does not open any channels. You must open any media channels before passing this controller to the `MTConferenceActivateConference` function.

`mtControllerVisiblePrepMask`
Indicates that you want the controller to be made visible to the user. This flag affects the control portion of the window (that is, the buttons and other controls that manipulate the media presentation).

`mtWindowVisiblePrepMask`
Indicates that you want the controller's window to be made visible to the user initially. This flag determines whether the display window itself is visible.

`mtAutoPositionPrepMask`
Allows the conference component to reposition the controller. If the screen location specified by the `windowRect` parameter is occupied by another window in your application layer, the conference component will try to position it elsewhere on the screen without overlapping other windows.

`mtAuxiliaryClosePrepMask`
Indicates that you are going to use this controller to display an auxiliary source. You attach the controller to its source by calling the `MTConferenceAttachAuxiliarySource` function. The conference component creates a controller that has a close box and automatically calls the `MTConferenceDetachAuxiliarySource` function if the user clicks the close box.

`mtEnableSnapshotPrepMask`
: Allows the user to grab images from the controller. The conference component creates a controller that supports image grabbing and presents the appropriate controls to the user.

`mtEnableRecordPrepMask`
: Allows the user to capture movies from the controller. The conference component creates a controller that supports movie grabbing and presents the appropriate controls to the user.

`mtNoAutoResizePrepMask`
: Prevents a receiving controller from automatically resizing the display window when the size of the media source changes. If you enable this option, the controller remains whatever size the user selects, even if this is not an optimal size for the incoming media stream.

`windowTitle`
: Specifies the title of the controller's window. Set this parameter to `nil` if you want the controller's window to be untitled.

DESCRIPTION

The `MTConferenceNewPreparedController` function creates a window and opens a stream director component as well as a controller component. For controllers that supply data, it can also attach a sequence grabber component and configure the appropriate sequence grabber channels for you. Once the controller is ready for use, you can assign it to a conference by calling the `MTConferenceActivateConference` function. If you are using the controller to display incoming media data, you can assign it to a specific participant by calling the `MTConferenceActivateMember` function.

You may reconfigure the controller using the functions provided by the controller component, sequence grabber component, or stream director component before attaching the controller to a conference. However, you must not dispose of any of the pieces; the conference component gets rid of all the parts it acquired when you call the `MTConferenceDisposeController` function or you close the conference component instance.

The conference component creates a window with a kind of `kMTConferenceWindowKind`. The window's reference constant is set to the controller's reference constant; you can change this value if you want to.

This function sets the returned value to `NIL` if the operation fails. Call the Component Manager's `GetComponentInstanceError` function to retrieve the result code.

SEE ALSO

The `MTConferenceDisposeController` function is discussed next.

See page 2-52 for more information about the `MTConferenceAttachAuxiliarySource` function.

The `MTConferenceActivateConference` function is described on page 2-44. The `MTConferenceActivateMember` function is discussed on page 2-45.

For more information about controller components, see the "Controller Components" chapter. For more information about stream director components, see the "Stream Director Components" chapter.

For more information about sequence grabber components and sequence grabber channels, see *Inside Macintosh: QuickTime Components*.

MTConferenceDisposeController

The `MTConferenceDisposeController` function disposes of a controller you acquired by calling the `MTConferenceNewPreparedController` function.

```
pascal ComponentResult MTConferenceDisposeController
                        (MTConferenceComponent cc,
                        MTControllerComponent controller);
```

`cc` Specifies the conference component instance.

`controller` Identifies the controller to discard.

DESCRIPTION

The conference component disposes of the controller and all of its constituent parts, including sequence grabber components and sequence grabber channels, even if you added the channels yourself.

Do not dispose of any controller parts yourself if you create the controller using the conference component's `MTConferenceNewPreparedController` function.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTConferenceNewPreparedController` function is discussed on page 2-46.

MTConferenceSetDefaultWindowProcID

The `MTConferenceSetDefaultWindowProcID` function sets the type of window used by the `MTConferenceNewPreparedController` function.

```
pascal ComponentResult MTConferenceSetDefaultWindowProcID
                        (MTConferenceComponent cc,
                        short windowProcID);
```

`cc` Specifies the conference component instance for the operation.

`windowProcID`
The type of window to use on subsequent invocations of the `MTConferenceNewPreparedController` function.

DESCRIPTION

By default, the conference component creates a `documentProc` window. The value you set affects all controllers you create subsequently with the specified conference component instance.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTConferenceNewPreparedController` function is discussed on page 2-46.

MTConferenceSetPreparationDefaults

The `MTConferenceSetPreparationDefaults` function allows you to set the types of components the conference component uses to build controllers.

```
pascal ComponentResult MTConferenceSetPreparationDefaults
                        (MTConferenceComponent cc,
                        OSType controllerSubtype,
                        OSType compressorSubtype,
                        OSType sourceDirectorSubtype,
                        OSType sinkDirectorSubtype,
                        OSType recorderSubtype);
```

`cc` Specifies the conference component instance.

`controllerSubtype`
Specifies the type of controller. The default is `kMTMovieTalkSubType`.

`compressorSubtype`
Video compressor component type. The default is `'rpza'`.

`sourceDirectorSubtype`
Type of source stream director component. The default is `kMTGrabberSubType`.

`sinkDirectorSubtype`
Type of sink stream director component. The default is `kMTPlayerType`.

`recorderSubtype`
Type of recorder component opened when the user starts recording. The default is `kMTMovieSubType`.

DESCRIPTION

This function allows you to specify the types of components the conference component uses to create a new controller when you call the `MTConferenceNewPreparedController` function. This function affects all controllers you create subsequently. You may leave the current setting of any of these parameter unchanged by setting the parameter to `nil`.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTConferenceNewPreparedController` function is discussed on page 2-46.

MTConferenceSetDefaultActionFilter

The `MTConferenceSetDefaultActionFilter` function allows you to assign an action filter function to a controller.

```
pascal ComponentResult MTConferenceSetDefaultActionFilter
                    (MTConferenceComponent cc,
                    MTControllerActionFilterUPP actionProc,
                    long actionRefcon);
```

`cc` Specifies the conference component instance for the operation.

`actionProc` Specifies your action filter function.

`actionRefcon`
Specifies a reference constant to be passed to your action filter function.

DESCRIPTION

The conference component automatically handles most controller interactions for you. However, in order to support special user interactions, you may specify an action filter function. This function can override the default controller behavior. If your action filter function returns `false` for any action, the conference component will take its default action. The complete rules for action filter functions are described in the "Controller Components" chapter of this book.

Setting an action filter function affects all controllers you create subsequently.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTConferenceNewPreparedController` function is discussed on page 2-46.

For more information about stream controller components, see the "Stream Controller Components" chapter.

Working With Auxiliary Sources

Normally, each participant in a conference has one media source (if any). This source presents combined audio, video, and other streams to other participants in a single window on each system. On occasion, however, a user might wish to introduce other sources to a conference and have them appear in different windows on remote systems. These are called auxiliary sources and may be attached to an existing conference.

This section discusses the functions that allow you to work with auxiliary sources. Use the `MTConferenceAttachAuxiliarySource` function to attach an auxiliary source to a conference. Use the `MTConferenceDetachAuxiliarySource` function to remove an auxiliary source from a conference.

MTConferenceAttachAuxiliarySource

The `MTConferenceAttachAuxiliarySource` function attaches an auxiliary source to an existing conference.

```
pascal MTAuxiliaryToken MTConferenceAttachAuxiliarySource
                            (MTConferenceComponent cc,
                            MTConferenceToken theConference,
                            MTControllerComponent controller,
                            MTCString63 auxName);
```

`cc` Specifies the conference component instance for the operation.

`theConference`
Identifies the conference to which this auxiliary source is to be attached.

`controller` Specifies a prepared source controller.

`auxName` Provides the name of this auxiliary. Other conference participants can obtain the name by calling the `MTConferenceGetMemberName` function.

DESCRIPTION

After creating an auxiliary source with the `MTConferenceNewPreparedController` function, you call the `MTConferenceAttachAuxiliarySource` function to attach it to an existing conference.

Every other conference participant will receive an `mtMemberJoiningEvent` event for this new media stream. They may treat it exactly as another individual participant if they don't wish to take special action. You may attach more than one auxiliary source to a conference.

SEE ALSO

The `MTConferenceNewPreparedController` function is discussed on page 2-46.

The `MTConferenceGetMemberName` function is described on page 2-60.

MTConferenceDetachAuxiliarySource

You can remove an auxiliary source from a conference without terminating the conference by calling the `MTConferenceDetachAuxiliarySource` function.

```
pascal ComponentResult MTConferenceDetachAuxiliarySource
                        (MTConferenceComponent cc,
                        MTAuxiliaryToken theAuxiliary);
```

`cc` Specifies the conference component instance.

`theAuxiliary`
Identifies the auxiliary source to be detached. You obtain this identifier from the `MTConferenceAttachAuxiliarySource` function.

DESCRIPTION

All auxiliary sources are automatically detached when the conference is terminated.

RESULT CODES

noErr	0	No error

SEE ALSO

The MTConferenceAttachAuxiliarySource function is discussed on page 2-52.

Exchanging Messages

The conference component allows you to exchange both media and control messages with other conference members. This section discusses the functions that allow you to exchange control messages. You may use any of these functions before accepting or refusing an incoming call. As a result, a common use of control messages is to support user authentication prior to entering a conference.

You determine the format and content of these control messages. The message type identifies the format and content of the message. The version number allows you to discriminate between different versions of your control protocol. The conference component places no restrictions on the content of your messages and does not process the content in any way.

To be friendly, applications should always support earlier versions of a message.

The MTConferenceSendMessageToConference function sends a message to all of a conference's members. The MTConferenceSendMessageToMember function sends a message to a single member. If your application expects to receive any control messages, you must call the MTConferenceSetMessageCapabilities function to identify the messages you support. You should call this function before you call the MTConferenceListen function. The MTConferenceGetMemberMessageCapabilities function allows you to determine the message support required by a conference member.

MTConferenceSendMessageToConference

The MTConferenceSendMessageToConference function transmits a message to every member of a conference.

```
pascal ComponentResult MTConferenceSendMessageToConference
                        (MTConferenceComponent cc,
                        MTConferenceToken theConference,
                        Handle theMessage,
                        OSType messageType,
                        short version);
```

cc Specifies the conference component instance for the operation.

`theConference`
Identifies the conference.

`theMessage` A handle containing the message data you wish to send. The conference component disposes of this handle after transmitting the message to each member.

`messageType`
A four-character type that the conference component passes to the receiver in an `mtMessageArrivedEvent` event.

`version` An integer indicating the version number of this message. The conference component at the remote end passes this value to the receiving application in an `mtMessageArrivedEvent` event.

DESCRIPTION

The conference component does not forward messages to members that do not support a given message type. Members indicate their ability to support different types of messages by calling the `MTConferenceSetMessageCapabilities` function. You can determine a member's capabilities by calling the `MTConferenceGetMemberMessageCapabilities` function. However, it is up to each recipient to determine whether it understands a particular message version.

Because the conference component disposes of the message after transmission using the Memory Manager's `DisposeHandle` function, if the message comes from a resource you must detach it using the `DetachResource` function before calling the `MTConferenceSendMessageToConference` function.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTConferenceSetMessageCapabilities` function is discussed on page 2-57.

The `MTConferenceGetMemberMessageCapabilities` function is described on page 2-59.

MTConferenceSendMessageToMember

The `MTConferenceSendMessageToMember` function transmits a message to a single conference member.

```
pascal ComponentResult MTConferenceSendMessageToMember
                        (MTConferenceComponent cc,
                        MTConferenceMember theMember,
```

```
                        Handle theMessage,
                        OSType messageType,
                        short version);
```

`cc` Specifies the conference component instance for the operation.

`theMember` Identifies the member.

`theMessage` A handle containing the message data you wish to send. The conference component disposes of this handle after transmitting it to the member.

`messageType`
A four-character type that the conference component passes to the receiver in an `mtMessageArrivedEvent` event.

`version` An integer indicating the version number of this message. The conference component at the remote end passes this value to the receiving application in an `mtMessageArrivedEvent` event.

DESCRIPTION

This function is similar to the `MTConferenceSendMessageToConference` function, but it sends the message to a single member rather than to the entire conference. You can use the `MTConferenceGetMemberMessageCapabilities` function to determine the highest version number of a message type that a conference member supports.

Because the conference component disposes of the message after transmission using the Memory Manager's `DisposeHandle` function, if the message comes from a resource you must detach it using the `DetachResource` function before calling the `MTConferenceSendMessageToMember` function.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTConferenceSetMessageCapabilities` function is discussed next.

The `MTConferenceGetMemberMessageCapabilities` function is described on page 2-59.

MTConferenceSetMessageCapabilities

The `MTConferenceSetMessageCapabilities` function allows you to specify the messages you support.

```
pascal ComponentResult MTConferenceSetMessageCapabilities
                        (MTConferenceComponent cc,
                        MTCapabilitiesHandle myCapabilities);
```

`cc` Specifies the conference component instance for the operation.

`myCapabilities`
A handle containing an `MTCapabilitiesList` structure. The conference component disposes of the handle when it is done.

DESCRIPTION

You can use the `MTConferenceSetMessageCapabilities` function before you call the `MTConferenceListen` function to establish the set of message types you understand. Other potential conference members may then call the `MTConferenceGetMemberMessageCapabilities` function to determine the message support you offer.

You format your capability information in a capabilities list record with the `count` field indicating the number of `MTCapabilitiesEntry` items included. The capabilities list record is defined as follows:

```
struct MTCapabilitiesList {
   UInt32                    count;
   MTCapabilitiesEntry       capability[kMTVariableLengthArray];
};
typedef struct MTCapabilitiesList MTCapabilitiesList,
*MTCapabilitiesListPtr, **MTCapabilitiesHandle;
```

Field descriptions

`count` Indicates the number of capability entries in the list contained in the `capability` field.

`capability` An array of capability entries. The `count` field indicates the number of entries in the array.

A capabilities list record contains one or more capability entry records. There is one capability entry record for each message type you support. The capabilities entry record is defined as follows:

```
struct MTCapabilitiesEntry {
   OSType                    messageType;
   unsigned char             version;
   MTConferenceCapabilityType desires;
```

```
   short                       reserved;
};
typedef struct MTCapabilitiesEntry MTCapabilitiesEntry,
*MTCapabilitiesPtr;
```

Field descriptions

`messageType` Identifies the message type. This is a four-character value. Apple reserves all message types consisting of lowercase alphabetic characters.

`version` Contains an integer version number. Version numbers start at 1.

`desires` Indicates the level of support required by your application. The following values are valid:

`mtMessageOptionalCapability`
Indicates that you do not require this message to be exchanged before setting up a conference.

`mtMessageDesiredCapability`
Indicates that you expect to receive this message once at the beginning of each conference.

`mtMessageRequiredCapability`
Indicates that you require that this message be exchanged before setting up a conference.

`mtNegotiationMessageCapability`
A special option. This option allows your application to query the remote application using facilities offered by the conference component. In this case, the `messageType` field indicates a component type value. The conference component returns a message to you containing a list of all of the valid subtypes it finds on the remote system. The returned message has its `messageType` value set to the same value as the `messageType` value in this capability entry.

`reserved` Reserved for use by Apple. Set this field to 0.

Required messages might include any that are used for authentication prior to establishing a conference. A videophone application might desire a message containing a small picture of the calling person that can be displayed while ringing the phone.

Whenever you connect to a member who has indicated any required or desired messages, the `mtIncomingCallEvent` or `mtPhoneRingingEvent` event will have either the `mtDesiredMessagesModeMask` or `mtRequiredMessagesModeMask` flag set to 1 along with the other mode flags that are appropriate to the conference.

Note that the required category is only advisory in the case of the `mtMemberJoiningEvent` event. You many not refuse third parties who try to join a conference once you have allowed it by setting your `mtJoinerModeMask` mode flag to 1.

RESULT CODES

`noErr`	0	No error

SEE ALSO

For more information about conference component events, see "Managing Conference Events" beginning on page 2-31.

MTConferenceGetMemberMessageCapabilities

The `MTConferenceGetMemberMessageCapabilities` function returns a list of message types supported by a conference member.

```
pascal MTCapabilitiesHandle
                    MTConferenceGetMemberMessageCapabilities
                       (MTConferenceComponent cc,
                       MTConferenceMember theMember);
```

`cc`	Specifies the conference component instance for the operation.
`theMember`	Identifies the conference member whose message capabilities should be retrieved.

DESCRIPTION

The format of the data is the same as that supplied to the `MTConferenceSetMessageCapabilities` function. You are responsible for disposing of the handle when you are done with it.

SEE ALSO

The `MTConferenceSetMessageCapabilities` function, along with the format of the capabilities list, is discussed beginning on page 2-57.

Getting Conference Information

This section discusses a number of utility functions that return information about conferences or conference members.

The `MTConferenceGetMemberName` function returns a member's user name. The `MTConferenceGetMemberConference` function allows you to determine to which conference a member belongs. The `MTConferenceGetConferenceName` function returns a conference's name. The `MTConferenceGetReturnAddress` function returns a conference participant's network-independent name. The `MTConferenceNameFromRString` and `MTConferenceRStringFromName` functions convert data between C-string and `RString` formats.

MTConferenceGetMemberName

The `MTConferenceGetMemberName` function returns the user name of a conference member.

```
pascal MTCString MTConferenceGetMemberName
                  (MTConferenceComponent cc,
                  MTConferenceMember theMember);
```

`cc` Specifies the conference component instance for the operation.

`theMember` Identifies the conference member whose name you desire.

DESCRIPTION

This function returns the name that the remote user provided to the `MTConferenceListen` function. You might want to retrieve this, for instance, to display the name of a caller to the local user.

Use the `MTConferenceRStringFromName` function to convert names that may not be in the Roman script.

The returned string is valid until you leave the conference. Do not dispose of this string.

SEE ALSO

The `MTConferenceListen` function is discussed beginning on page 2-28.

For information about the `MTConferenceRStringFromName` function, see page 2-63.

MTConferenceGetMemberConference

The `MTConferenceGetMemberConference` function identifies the conference to which a member currently belongs.

```
pascal MTConferenceToken MTConferenceGetMemberConference
                          (MTConferenceComponent cc,
                          MTConferenceMember theMember);
```

`cc` Specifies the conference component instance for the operation.

`theMember` Identifies the member. This member identifier is returned by the `MTConferenceCall` function or with the `mtIncomingCallEvent` or `mtMemberJoiningEvent` event.

DESCRIPTION

Use this function to determine the conference identifier that corresponds to the conference to which a member belongs. Conference components create these identifiers as they need them. Consequently, when you establish a conference with a user (say, by calling the `MTConferenceCall` function), you need to call this function to retrieve the conference identifier.

SEE ALSO

For more information about conference component events, see "Managing Conference Events" beginning on page 2-31.

See page 2-39 for a description of the `MTConferenceCall` function.

MTConferenceGetConferenceName

The `MTConferenceGetConferenceName` function returns the name associated with a conference.

```
pascal MTCString MTConferenceGetConferenceName
                  (MTConferenceComponent cc,
                  MTConferenceToken theConference);
```

`cc` Specifies the conference component instance for the operation.

`theConference`
Indicates the conference whose name you desire.

DESCRIPTION

This function returns the name that the originator supplied to the `MTConferenceCall` function. Use the `MTConferenceRStringFromName` function to convert names that may not be in the Roman script.

The returned string is valid until you leave the conference. Do not dispose of this string.

SEE ALSO

The `MTConferenceCall` function is discussed beginning on page 2-39.

For information about the `MTConferenceRStringFromName` function, see page 2-63.

MTConferenceGetReturnAddress

The `MTConferenceGetReturnAddress` function returns a conference participant's network-independent name.

```
pascal MTNameListPtr MTConferenceGetReturnAddress
                        (MTConferenceComponent cc,
                        MTConferenceMember theMember);
```

`cc` Specifies the conference component instance for the operation.

`theMember` Identifies the conference member.

DESCRIPTION

Use this function to retrieve the network address you would use to call a conference member directly. This can be useful if you need to call a particular conference member at a later date. For example, if you were implementing an answering machine, you might want to use this function to record a caller's return address along with the message. You could use that address to provide a return call capability to your user.

The function returns the name in a name list structure. The name itself is suitable for use with the conference or transport component to initiate a network call.

You need to dispose of the name list structure when you are done with it; call the Memory Manager's `DisposePtr` function.

RESULT CODES

`noErr`	0	No error

SEE ALSO

See the "Browser Components" chapter for more information about the name list structure.

MTConferenceNameFromRString

For handling other than Roman script systems, the `MTConferenceNameFromRString` function converts an AOCE-format `RString` to the C-string format used for user and conference names throughout the conference component.

```
pascal ComponentResult MTConferenceNameFromRString
                        (MTConferenceComponent cc,
                        MTCString63 name,
                        MTRString60Ptr rString);
```

cc	Specifies the conference component instance for the operation.
name	Contains a pointer to the resultant name.
rString	Contains a pointer to the input RString.

DESCRIPTION

The resultant string may be manipulated as a standard C string.

RESULT CODES

noErr	0	No error

SEE ALSO

The MTConferenceRStringFromName function, described next, converts C strings into RString structures.

MTConferenceRStringFromName

For handling other than Roman script systems, the MTConferenceNameFromRString function converts a C string to an AOCE-format RString.

```
pascal ComponentResult MTConferenceRStringFromName
                        (MTConferenceComponent cc,
                        MTRString60Ptr rString,
                        MTCString63 name);
```

cc	Specifies the conference component instance for the operation.
rString	Specifies a pointer to the output RString.
name	Contains a pointer to the input name.

DESCRIPTION

This function sets the character set of the resulting RString to smRoman.

RESULT CODES

noErr	0	No error

SEE ALSO

The MTConferenceNameFromRString function, described on page 2-62, converts RString structures into C strings.

Summary of the Conference Component

C Summary

Constants

```
enum {
   kMTConferenceWindowKind =     621
};

/*----------------------------------------
   Conference selectors
----------------------------------------*/
enum {
   kMTConferenceListenSelect =                      1,
   kMTConferenceTerminateSelect =                   2,
   kMTConferenceCallSelect =                        3,
   kMTConferenceReplySelect =                       4,
   kMTConferenceMergeSelect =                       5,
   kMTConferenceDetachMemberSelect =                6,
   kMTConferenceGetNextEventSelect =                7,
   kMTConferenceGetMemberNameSelect =               8,
   kMTConferenceGetMemberConferenceSelect =         9,
   kMTConferenceGetReturnAddressSelect =            10,
   kMTConferenceGetRegisteredNamesSelect =          11,
   kMTConferenceGetConferenceNameSelect =           12,
   kMTConferenceActivateConferenceSelect =          13,
   kMTConferenceActivateMemberSelect =              14,
   kMTConferenceSendMessageToConferenceSelect =     15,
   kMTConferenceSendMessageToMemberSelect =         16,
   kMTConferenceSetModeSelect =                     17,
   kMTConferenceSetCallTimeoutSelect =              18,
   kMTConferenceSetDefaultWindowProcIDSelect =      19,
   kMTConferenceSetDefaultActionFilterSelect =      20,
   kMTConferenceSetPreparationDefaultsSelect =      21,
   kMTConferenceNewPreparedControllerSelect =       22,
   kMTConferenceDisposeControllerSelect =           23,
   kMTConferenceSetMessageCapabilitiesSelect =      24,
```

```
   kMTConferenceGetMemberMessageCapabilitiesSelect =  25,
   kMTConferenceAttachAuxiliarySourceSelect =         26,
   kMTConferenceDetachAuxiliarySourceSelect =         27,
   kMTConferenceNameFromRStringSelect =               28,
   kMTConferenceRStringFromNameSelect =               29
};
```

Data Types

```
typedef ComponentInstance        MTConferenceComponent;
typedef unsigned char            MTCString32[33], MTCString63[64];
typedef long                     MTConferenceToken;
typedef long                     MTAuxiliaryToken;
typedef long                     MTConferenceMember;

struct MTRString60 {
   short                         charSet;       // script code; see Script
   short                         dataLength;    // Manager for more
   Byte                          body[60];      // information
};
typedef struct MTRString60       MTRString60, *MTRString60Ptr;

/*----------------------------------------
   Conference mode bits
----------------------------------------*/
enum {
   mtSendMediaModeMask =         0x0001,
   mtReceiveMediaModeMask =      0x0002,
   mtShareableModeMask =         0x0004,
   mtJoinerModeMask =            0x0008,

   mtDesiredMessagesModeMask =   0x0100,     // only on
                                                mtPhoneRingingEvent,
                                                mtIncomingCallEvent, and
                                                mtMemberJoiningEvent
   mtRequiredMessagesModeMask=   0x0200      // only on
                                                mtPhoneRingingEvent, and
                                                mtIncomingCallEvent
};
typedef UInt16                   MTConferenceModeFlags;


/*----------------------------------------
```

```
   Conference prepared bits
----------------------------------------*/
enum {
   mtGrabVideoPrepMask =         0x00000001,
   mtGrabAudioPrepMask =         0x00000002,
   mtGrabOtherPrepMask =         0x00000008,
   mtAuxiliaryClosePrepMask =    0x00000100,
   mtMediaSourcePrepMask =       0x00008000,
   mtEnableSnapshotPrepMask =    0x00100000,
   mtEnableRecordPrepMask =      0x00200000,
   mtWindowVisiblePrepMask =     0x01000000,
   mtControllerVisiblePrepMask = 0x02000000,
   mtNoAutoResizePrepMask =      0x04000000,
   mtAutoPositionPrepMask =      0x08000000
};
typedef UInt32                   MTConferencePrepFlags;

/*----------------------------------------
   Conference event reasons
----------------------------------------*/
enum {
   mtListenerStatusEvent =          11,
   mtIncomingCallEvent =            12,
   mtMemberJoiningEvent =           13,
   mtMemberMovedEvent =             14,
   mtMessageArrivedEvent =          15,
   mtPhoneRingingEvent =            16,
   mtConferenceReadyEvent =         21,
   mtConferenceTerminatedEvent =    22,
   mtAuxiliaryTerminatedEvent =     23,
   mtConferenceNameChangedEvent =   24,
   mtMemberReadyEvent =             31,
   mtMemberTerminatedEvent =        32,
   mtRefusedEvent =                 33,
   mtFailedEvent =                  34,
   mtSnapshotTakenEvent =           41,
   mtMovieRecordedEvent =           42
};
typedef short                    MTConferenceEventType;



/*----------------------------------------
```

```
   Conference event structure
----------------------------------------*/

struct MTConferenceEvent {
   MTConferenceEventType            what;
   OSErr                            err;
   long                             who;
   long                             bonus;
   long                             reserved;
   Handle                           surprise;
};
typedef struct MTConferenceEvent MTConferenceEvent, *MTConferenceEventPtr;

/*----------------------------------------
   Conference capability types
----------------------------------------*/
enum {
   mtMessageOptionalCapability =    'O',
   mtMessageDesiredCapability =     'D',
   mtMessageRequiredCapability =    'R',
   mtNegotiationMessageCapability = 'N'
};
typedef char                        MTConferenceCapabilityType;

/*----------------------------------------
   Conference capabilities structure
----------------------------------------*/
struct MTCapabilitiesEntry {
   OSType                       messageType;
   unsigned char                version;
   MTConferenceCapabilityType   desires;
   short                        reserved;
};
typedef struct MTCapabilitiesEntry MTCapabilitiesEntry, *MTCapabilitiesPtr;

struct MTCapabilitiesList {
   UInt32                       count;
   MTCapabilitiesEntry          capability[kMTVariableLengthArray];
};
typedef struct MTCapabilitiesList MTCapabilitiesList, *MTCapabilitiesListPtr,
                                  **MTCapabilitiesHandle;
```

Functions

Configuring the Conference Component

```
pascal ComponentResult MTConferenceSetMode (MTConferenceComponent cc,
                            MTConferenceModeFlags defaultMode);
pascal ComponentResult MTConferenceSetCallTimeout (MTConferenceComponent cc,
                            short callTimeout);
pascal ComponentResult MTConferenceListen (MTConferenceComponent cc,
                            MTCString63 userName, MTCString32 serviceName,
                            MTCString serviceTypesList);
pascal MTNameListPtr MTConferenceGetRegisteredNames (MTConferenceComponent
                            cc);
```

Managing Conference Events

```
pascal ComponentResult MTConferenceGetNextEvent (MTConferenceComponent cc,
                            MTConferenceEventPtr theEvent);
```

Managing Calls

```
pascal MTConferenceMember MTConferenceCall (MTConferenceComponent cc,
                            MTCString63 confName, MTNamePtr whoToCall);
pascal ComponentResult MTConferenceReply (MTConferenceComponent cc,
                            MTConferenceMember callingParty, OSErr
                            response);
pascal ComponentResult MTConferenceMerge (MTConferenceComponent cc,
                            MTConferenceToken staysAround,
                            MTConferenceToken goesAway);
pascal ComponentResult MTConferenceTerminate (MTConferenceComponent cc,
                            MTConferenceToken theConference);
pascal ComponentResult MTConferenceDetachMember (MTConferenceComponent cc,
                            MTConferenceMember theMember);
```

Activating Conferences

```
pascal ComponentResult MTConferenceActivateConference (MTConferenceComponent
                            cc, MTConferenceToken theConference,
                            MTControllerComponent selfController);
pascal ComponentResult MTConferenceActivateMember (MTConferenceComponent cc,
                            MTConferenceMember theMember,
                            MTControllerComponent memberController);
```

Working With Controllers

```
pascal MTControllerComponent MTConferenceNewPreparedController
                            (MTConferenceComponent cc, const Rect
                            *windowRect, MTConferencePrepFlags prepFlags,
                            MTCString windowTitle);
pascal ComponentResult MTConferenceDisposeController (MTConferenceComponent
                            cc, MTControllerComponent controller);
pascal ComponentResult MTConferenceSetDefaultWindowProcID
                            (MTConferenceComponent cc, short windowProcID);
pascal ComponentResult MTConferenceSetPreparationDefaults
                            (MTConferenceComponent cc, OSType
                            controllerSubtype, OSType compressorSubtype,
                            OSType sourceDirectorSubtype, OSType
                            sinkDirectorSubtype, OSType recorderSubtype);
pascal ComponentResult MTConferenceSetDefaultActionFilter
                            (MTConferenceComponent cc,
                            MTControllerActionFilterUPP actionProc, long
                            actionRefcon);
```

Working With Auxiliary Sources

```
pascal MTAuxiliaryToken MTConferenceAttachAuxiliarySource
                            (MTConferenceComponent cc, MTConferenceToken
                            theConference, MTControllerComponent
                            controller, MTCString63 auxName);
pascal ComponentResultMTConferenceDetachAuxiliarySource
                            (MTConferenceComponent cc, MTAuxiliaryToken
                            theAuxiliary);
```

Exchanging Messages

```
pascal ComponentResult MTConferenceSendMessageToConference
                            (MTConferenceComponent cc, MTConferenceToken
                            theConference, Handle theMessage, OSType
                            messageType, short version);
pascal ComponentResult MTConferenceSendMessageToMember
                            (MTConferenceComponent cc, MTConferenceMember
                            theMember, Handle theMessage, OSType
                            messageType, short version);
pascal ComponentResult MTConferenceSetMessageCapabilities
                            (MTConferenceComponent cc,
                            MTCapabilitiesHandle myCapabilities);
pascal MTCapabilitiesHandle MTConferenceGetMemberMessageCapabilities
                            (MTConferenceComponent cc, MTConferenceMember
                            theMember);
```

Getting Conference Information

```
pascal MTCString MTConferenceGetMemberName (MTConferenceComponent cc,
                            MTConferenceMember theMember);
pascal MTConferenceToken MTConferenceGetMemberConference
                            (MTConferenceComponent cc, MTConferenceMember
                            theMember);
pascal MTCString MTConferenceGetConferenceName (MTConferenceComponent cc,
                            MTConferenceToken theConference);
pascal MTNameListPtr MTConferenceGetReturnAddress (MTConferenceComponent cc,
                            MTConferenceMember theMember);
pascal ComponentResult MTConferenceNameFromRString (MTConferenceComponent
                            cc, MTCString63 name, MTRString60Ptr rString);
pascal ComponentResult MTConferenceRStringFromName (MTConferenceComponent
                            cc, MTRString60Ptr rString, MTCString63 name)
```

Result Codes

`noErr`	0	No error
`paramErr`	–50	Invalid parameter value
`mtNoListenersErr`	–7743	Could not open any network type in your service list
`mtModeErr`	–7745	Controller specified when not needed or not specified when one was required
`mtNotDetachableErr`	–7746	Member not detachable because not on a network that supports multicast transmission

Browser Components

Contents

Browser components allow applications to find other QuickTime Conferencing services on a network. This chapter describes the standard QuickTime Conferencing browser components provided by Apple Computer, Inc. for PowerTalk, AppleTalk, ISDN, and TCP/IP networks.

About Browser Components

Browser components provide the user interface for locating and choosing QuickTime Conferencing services on a network. Your application can use browser components to present the user with a dialog box for browsing. When the user makes a selection, the browser component returns a MovieTalk network name data structure that can be passed on to the transport component to establish a connection.

The following standard browser components are provided by Apple Computer, Inc.:

- a PowerTalk browser that provides access to PowerTalk network browsing facilities as well as PowerTalk personal address books and directories
- an AppleTalk browser component that uses a zone/name interface, much like the Chooser
- a TCP/IP browser that accepts an Internet node name or domain address
- an ISDN browser that accepts ISDN telephone numbers

Other browser components may be supplied by Apple or third parties to provide new features or support additional network types.

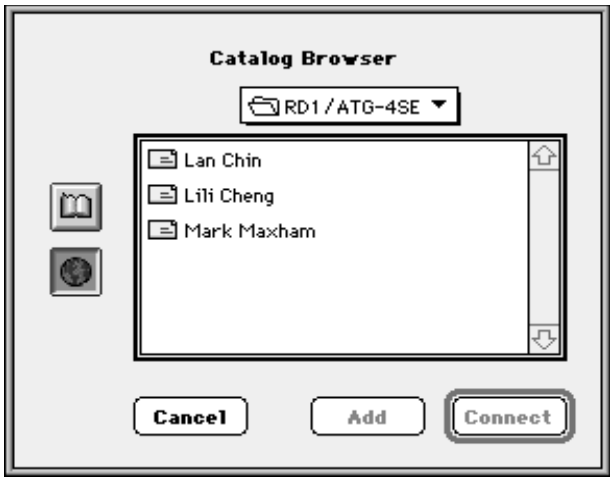
The PowerTalk Browser Component

The PowerTalk browser component provides access to PowerTalk network browsing facilities and the PowerTalk personal catalog. Your application should use the PowerTalk browser if PowerTalk is installed. Figure 3-1 shows the PowerTalk browser personal catalog window. Figure 3-2 shows the PowerTalk browser network catalog window.

For more information about PowerTalk see *Inside Macintosh: AOCE Application Interfaces*.

Figure 3-1 The PowerTalk browser personal catalog window

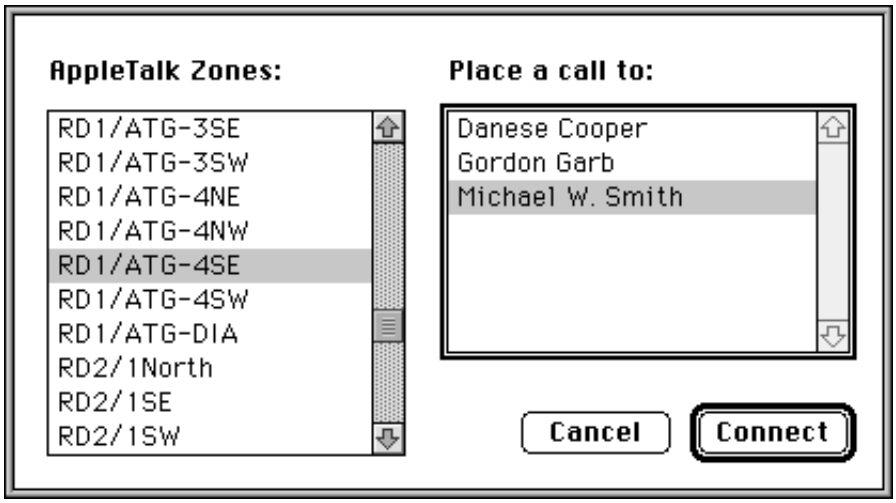
Figure 3-2 The PowerTalk browser network catalog window



The AppleTalk Browser Component

The AppleTalk browser component displays a modal dialog box that allows the user to select a service by its AppleTalk zone and name. The AppleTalk browser provides an interface similar to the Chooser. Figure 3-3 shows the AppleTalk browser dialog box.

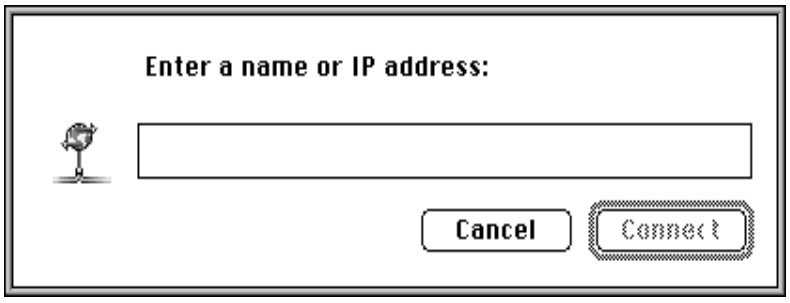
Figure 3-3 The AppleTalk browser dialog box



The TCP/IP Browser Component

The TCP/IP browser component displays a modal dialog box where the user can type an internet node name or IP address. If the user enters a node name, the TCP/IP network component searches for the corresponding IP address using TCP/IP directory services. Figure 3-4 shows the TCP/IP browser dialog box.

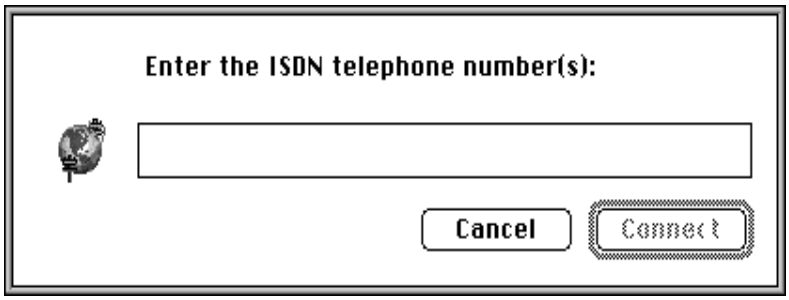
Figure 3-4 The TCP/IP browser dialog box



The ISDN Browser Component

The ISDN browser component displays a modal dialog box where the user can enter one or more ISDN telephone numbers. Multiple numbers are supported when more than one ISDN telephone number is used for a connection. Figure 3-5 shows the ISDN browser dialog box.

Figure 3-5 The ISDN browser dialog box



Using Browser Components

This section provides information you need to open and use the standard browser components.

Opening a Browser Component

You can open a browser component using the `OpenDefaultComponent` function:

```
browser = OpenDefaultComponent(kMTBrowserType, browserSubType);
```

where `browser` is a browser component instance and `browserSubType` specifies the type of browser to open.

All the standard browser components have a component type of `kMTBrowserType` (`'brsr'`) and one of the component subtypes shown in Table 3-1.

Table 3-1 Browser component subtypes

Subtype	Constant	Value
AppleTalk	`kMTAppleTalkSubType`	`'atlk'`
ISDN	`kMTISDNSubType`	`'isdn'`
PowerTalk	`kMTAOCESubType`	`'aoce'`
TCP/IP	`kMTTCPIPSubType`	`'tcpi'`

Finding a Specific Browser Component

QuickTime Conferencing registers browser components with the Component Manager using the following information in the component description record:

Browser	componentType	componentSubType	componentFlags
Chooser	'brsr'	'atlk'	kMTNativeBrowserFlag + kMTCanDoModalFlag
ISDN	'brsr'	'isdn'	kMTNativeBrowserFlag + kMTCanDoModalFlag
TCP/IP	'brsr'	'tcpi'	kMTNativeBrowserFlag + kMTCanDoModalFlag

The `kMTNativeBrowserFlag` specifies the native browser component for systems that do not have PowerTalk installed. The `kMTCanDoModalFlag` indicates that the browser has a modal dialog interface.

On systems with PowerTalk installed, the PowerTalk browser component is registered as:

Browser	componentType	componentSubType	componentFlags
PowerTalk	'brsr'	'aoce'	kMTCanDoModalFlag
PowerTalk	'brsr'	'atlk'	kMTCanDoModalFlag

The Component Manager returns a native browser component instance when you use the `OpenDefaultComponent` function on systems that do not have PowerTalk installed. On systems with PowerTalk installed, the PowerTalk browser component subtype replaces the Chooser browser component subtype (`'atlk'`), and the `OpenDefaultComponent` function returns a PowerTalk browser component instance.

You can use the Component Manager `FindNextComponent` and `OpenComponent` functions to find and open a specific browser component. Listing 3-1 shows how to use these functions to find a native browser component for a specified network type.

Listing 3-1 Selecting an alternate browser component

```
ComponentInstance GetMeANativeBrowser( OSType networkType )
{
   ComponentInstance    browser = nil;
   ComponentDescription nativeDescription;
   Component            nativeComponent;

   nativeDescription.componentType = kMTBrowserType;
   nativeDescription.componentSubType = networkType;
   nativeDescription.componentManufacturer = 0;
   nativeDescription.componentFlags = kMTNativeBrowserFlag;
   nativeDescription.componentFlagsMask = kMTNativeBrowserFlag;
```

```
    nativeComponent = FindNextComponent( nil, &nativeDescription );

    browser = OpenComponent( nativeComponent );

    return browser;
}
```

You can use the `GetMeANativeBrowser` function to select the Chooser browser component on a PowerTalk system by specifying `'atlk'` as the network type.

Specifying Filter Types

You can search for QuickTime Conferencing services that are using a specific transport or network type by specifying a filter type string when you call the `MTBrowserBrowse` function. You can also search for specific QuickTime Conferencing service types (for example, a particular QuickTime Conferencing application).

```
pascal ComponentResult MTBrowserBrowse(
                                MTBrowserComponent c,
                                Point where
                                ConstStr255Param prompt,
                                MTCString filterTypes,
                                ModalFilterProcPtr proc,
                                MTNameList *nameList)
```

The `filterTypes` parameter is a C string of type `MTCString`, in the following format:

OSTypeOSType`\t`*network-specific data*`\n`*[OSTypeOSType*`\t`*network-specific data*`\n`*] [...]*

This is the same string format used by the `serviceTypesList` parameter of the `MTConferenceListen` function. The first *OSType* value specifies the transport component subtype for which to search. The second *OSType* value specifies the network component subtype. You use the tab metacharacter (`\t`) to separate the transport and network subtypes from the network-specific data. You use the newline metacharacter (`\n`) to separate multiple filter types.

The *OSType* value for the default QuickTime Conferencing transport component subtype is `'mtlk'`. The network subtype string must be one of the defined subtypes shown in Table 3-1. The AppleTalk, TCP/IP, and ISDN browser components ignore network subtypes that they do not support, but the PowerTalk browser can search for any network subtype.

The network-specific data you supply may be an AppleTalk NBP type, such as `"VideoPhone"`, a TCP/IP port number, or any other information required by the network component. QuickTime Conferencing currently defines two service types using the AppleTalk Name Binding Protocol, `"VideoPhone"` and `"Multicaster"`. You can

use the following string to search for Apple VideoPhone or Multicaster services on an AppleTalk network:

```
"mtlkatlk\tVideoPhone\nmtlkatlk\tMulticaster\n"
```

The equivalent TCP/IP socket numbers for these service types are 458 for VideoPhone and 545 for Multicaster. You can use the following string to search for these service types on a TCP/IP network:

```
"mtlktcpi\t458\nmtlktcpi\t545\n"
```

You can define a unique service type for your application if you do not intend to interoperate with other media conferencing applications. For example, the following string filters for your unique NBP service type (`"MyOwnNBPType"`) on an AppleTalk network:

```
"mtlkatlk\tMyOwnNBPType\n"
```

IMPORTANT
Your application must use the VideoPhone or Multicaster service types to interoperate with the Apple Media Conference application. ▲

Displaying a Browser Dialog Box

You use the `MTBrowserBrowse` function to display a browser dialog box. Listing 3-2 shows an example of how to use the `MTBrowserBrowse` function.

Listing 3-2 Creating a browser dialog box

```
ComponentInstance  gBrowser = nil; // browser component instance
MTNameListPtr      nameList;
MTCString          filterTypes;
Point              *where = nil; // default
ConstStr255Param   prompt = nil; // default
ModalFilterProcPtr proc = MyModalFilterProc;

gBrowser = OpenDefaultComponent(kMTBrowserType, kMTAOCESubType);
if (gBrowser == nil) AlertUser(); // could not open browser
types = "mtlkatlk\tVideoPhone\n";
err = MTBrowserBrowse(gBrowser, where, prompt, types, proc, &nameList);
if (err == noErr) {
   // use nameList to make a connection
   DisposePtr((Ptr) nameList); // dispose memory
}
```

In this example, `gBrowser` is a PowerTalk browser component instance opened using the `OpenDefaultComponent` function. The `MTBrowserBrowse` function opens the browser dialog box.

The `where` parameter specifies the location of the dialog box. If `where` is `nil`, the dialog box is centered on the screen. The `prompt` parameter can contain an optional prompt string. If `prompt` is `nil`, the browser's default prompt string is displayed.

The `filterTypes` string instructs the browser component to display only `"VideoPhone"` services on an AppleTalk network.

When the user makes a selection from the browser dialog box, `MTBrowserBrowse` returns a pointer to an `MTNameList` structure. The `count` field of the `MTNameList` structure contains the number of `MTName` values in the list. Your application must release the memory used by the `MTNameList` structure.

Interpreting MovieTalk Names

When the user makes a selection in a browser dialog box, the browser component returns a MovieTalk name list record (`MTNameList`) containing one or more MovieTalk name records. Your application can then pass the MovieTalk name record to the `MTConferenceCall` function to place a call.

The MovieTalk name record returned by the browser component is defined by the `MTName` data type:

```
struct MTName {
   OSType   transportType;
   OSType   networkType;
   Byte     networkName[kMTMaxNameSize];
   struct {
      short charSet;
      short length;
      Byte  name[kMTMaxDisplayNameSize];
   } displayName;
   Byte     entityInfo[kMTMaxEmailAddressSize];
};
typedef struct MTName MTName, *MTNamePtr;
```

When the browser component returns an `MTName` structure, the `transportType` field is set to `kMTMovieTalkSubType` or another transport component subtype selected through the PowerTalk business card. The `networkType` field is set to one of the following: `kMTAppleTalkSubType`, `kMTTCPIPSubType`, `kMTISDNSubType`, or another network component subtype selected by the PowerTalk business card.

The `networkName` field is a C string in the format `"name:service type:address"`. Depending on the network type selected, `networkName` contains the following information:

Network	`networkName` field contains
AppleTalk	`"name:service type:zone"`
ISDN	`"name:0:ISDNAddress"`
TCP/IP	`"name:socket number:IPAddress"`

In most cases, your application does not need to interpret the information in the `networkName` field. It is up to the individual QuickTime Conferencing transport and network components to interpret and use these fields.

The `displayName` field contains the name of the service selected or typed by the user. The standard QuickTime Conferencing browsers fill in `displayName` as follows:

Browser	`displayName` field contains
AppleTalk (Chooser)	The NBP registered name of the selected service.
PowerTalk	If a PowerTalk business card is selected, `displayName` is the name of the business card. If the service is selected from the AppleTalk catalog, `displayName` is the NBP registered name of the service.
ISDN	The ISDN telephone number(s) specified by the user.
TCP/IP	The IP name or address specified by the user.

The `entityInfo` field is a structure large enough to hold an AOCE `PackedDSSpec` structure (described in *Inside Macintosh: AOCE Application Interfaces:*). The `PackedDSSpec` structure can be used to retrieve additional information about the selected AOCE record. It can also be used to send AOCE mail with AOCE's Standard Mail Package.

Browser Components Reference

This section describes the constants, data types, and functions you use with QuickTime Conferencing browser components.

Constants

```
/* browser component type */
enum {
   kMTBrowserType          = 'brsr'
};

/* component subtypes */
enum {
   kMTMovieTalkSubType     = 'mtlk',
   kMTAOCESubType          = 'aoce',
   kMTAppleTalkSubType     = 'atlk',
   kMTTCPIPSubType         = 'tcpi',
   kMTISDNSubType          = 'isdn'
};

/* browser selectors */
enum {
   kMTBrowserBrowseSelect        =  1,
   kMTBrowserGetSettingsSelect   =  2,
   kMTBrowserSetSettingsSelect   =  3
};

/* browser item hit values for modal dialog filter procs */
enum {
   kMTBrowserCancelItem = 1000,
   kMTBrowserOKItem     = 1001,
   kMTBrowserAddItem    = 1002
};

enum {
   kMTNativeBrowserFlag = 0x0002,
   kMTCanDoModalFlag    = 0x0004
};
```

Data Types

MovieTalk name list record

Browser components return network-independent names in a MovieTalk name list record. The MovieTalk name list record is defined by the `MTNameList` data type:

```
struct MTNameList {
   UInt32   count;
   MTName   list[kMTVariableLengthArray];
};
typedef struct MTNameList MTNameList, *MTNameListPtr;
```

Field descriptions

`count`	The number of elements in the `list` array.
`list`	An array of MovieTalk name records. The `MTName` structure is described in the chapter "Transport Components" in this book.

Functions

This section describes the functions that comprise the browser components API.

MTBrowserBrowse

You can use the `MTBrowserBrowse` function to display a modal browser dialog.

```
pascal ComponentResult MTBrowserBrowser
                              (MTBrowserComponent browser,
                               const Point *where,
                               ConstStr255Param prompt,
                               MTCString filterTypes,
                               ModalFilterProcPtr proc,
                               MTNameListPtr *nameList);
```

`browser`	Specifies the browser component instance for this request. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`where`	A point indicating the upper left of the browser window. If `where` is `nil`, the window is centered on the screen.
`prompt`	A prompt string to be displayed in the browser window. If `prompt` is `nil`, a default string is displayed.

`filterTypes`
: A null-terminated string of type `MTCString`, indicating the transport and network subtypes for which to search. Unsupported network subtypes are ignored by the AppleTalk, TCP/IP, and ISDN browser components. The format of this string is:

OSTypeOSType`\t`*network-specific data*`\n`*[...]*

The first *OSType* value specifies the transport component subtype for which to search. The second *OSType* value specifies the network component subtype. You use the tab metacharacter (`\t` or `0x09`) to separate the transport and network subtypes from the network-specific data. You use the newline metacharacter (`\n` or `0x0D`) to separate multiple filter types.

`proc`
: A pointer to an application-defined modal filter procedure. See *Inside Macintosh: Files* for information about writing a modal-dialog filter function.

`nameList`
: A pointer to an `MTNameListPtr` pointer. Your application must release the memory used by the `MTNameList` structure.

DESCRIPTION

The `MTBrowserBrowse` function displays a modal browser dialog. If the function returns `noErr`, the `nameList` parameter contains a list of `MTName` values selected by the user. The `count` field of the `MTNameList` structure contains the number of `MTName` values in the list.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameters
`permErr`	–54	Permission error (invalid PowerTalk password)
`userCanceledErr`	–128	User pressed Cancel button
`resNotFound`	–192	Error getting a resource
`mtConfigurationErr`	–7891	Improper network configuration (AppleTalk turned off)

MTBrowserGetSettings

You can use `MTBrowserGetSettings` to get preference settings from a browser component.

```
pascal ComponentResult MTBrowserGetSettings
                              (MTBrowserComponent browser,
                               Handle *prefs);
```

`browser`
: The browser component instance.

`prefs`
: A pointer to a handle that contains the preference data.

DESCRIPTION

The `MTBrowserGetSettings` function returns the preferences from a browser component. The size and format of the preference information is determined by the browser component.

SPECIAL CONSIDERATIONS

`MTBrowserGetSettings` allocates a handle and returns a pointer to it in `prefs`. Your application must dispose of this handle when it is no longer needed.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameters
`resNotFound`	–192	Error getting a resource

Additional error result codes may be returned by the Memory Manager or by PowerTalk services.

MTBrowserSetSettings

You can use `MTBrowserSetSettings` to set preferences for a browser component.

```
pascal ComponentResult MTBrowserSetSettings
                              (MTBrowserComponent browser,
                               Handle prefs);
```

`browser` The browser component instance.

`prefs` A handle to the preference data. Your application must dispose of this handle when it is no longer needed.

DESCRIPTION

The `MTBrowserSetSettings` function sets the preferences for a browser component. You can use this function to restore preference settings retrieved using the `MTBrowserGetSettings` function.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameters

Summary of Browser Components

C Summary

Constants and Data Types

```
/* browser component type */
enum {
   kMTBrowserType          = 'brsr'
};

/* component subtypes */
enum {
   kMTMovieTalkSubType     = 'mtlk',
   kMTAOCESubType          = 'aoce',
   kMTAppleTalkSubType     = 'atlk',
   kMTTCPIPSubType         = 'tcpi',
   kMTISDNSubType          = 'isdn'
};

/* browser selectors */
enum {
   kMTBrowserBrowseSelect        =  1,
   kMTBrowserGetSettingsSelect   =  2,
   kMTBrowserSetSettingsSelect   =  3
};

/* browser item hit values for modal dialog filter procs */
enum {
   kMTBrowserCancelItem = 1000,
   kMTBrowserOKItem     = 1001,
   kMTBrowserAddItem    = 1002
};

enum {
   kMTNativeBrowserFlag = 0x0002,
   kMTCanDoModalFlag    = 0x0004
};
```

```
/* transport independent name */
struct MTName {
   OSType   transportType;
   OSType   networkType;
   Byte     networkName[kMTMaxNameSize];
   struct {
      short charSet;
      short length;
      Byte  name[kMTMaxDisplayNameSize];
   } displayName;
   Byte     entityInfo[kMTMaxEmailAddressSize];
};
typedef struct MTName MTName, *MTNamePtr;

struct MTNameList {
   UInt32   count;
   MTName   list[kMTVariableLengthArray];
};
typedef struct MTNameList MTNameList, *MTNameListPtr;
```

Functions

```
pascal ComponentResult MTBrowserBrowse
                              (MTBrowserComponent browser,
                               const Point *where,
                               ConstStr255Param prompt,
                               MTCString filterTypes,
                               ModalFilterProcPtr proc,
                               MTNameListPtr *nameList);
pascal ComponentResult MTBrowserGetSettings
                              (MTBrowserComponent browser,
                               Handle *prefs);
pascal ComponentResult MTBrowserSetSettings
                              (MTBrowserComponent browser,
                               Handle prefs);
```

Result Codes

`noErr`	0	No error
`paramErr`	–50	Invalid parameters
`permErr`	–54	Permission error (invalid PowerTalk password)
`userCanceledErr`	–128	User pressed Cancel button
`resNotFound`	–192	Error getting a resource
`mtConfigurationErr`	–7891	Improper network configuration (AppleTalk turned off)

Additional error result codes may be returned by the Memory Manager or by PowerTalk services.

Stream Controller Components

Contents

This chapter describes stream controller components. **Stream controller components** allow your application to provide a user interface for QuickTime Conferencing connections quickly and easily. Stream controller components present visible and audible media data to a user. They supply controls to stop and start that presentation, to record the media data, to adjust volume and audio input gain levels, and to capture a picture of the current image.

Stream controller components remove much of the burden of presenting a human interface for QuickTime Conferencing connections, freeing your application to focus on the unique services it offers to users. They provide a consistent user interface. However, you also can write your own routines to handle some or all of the human interface tasks.

If your application uses a conference component (described in the chapter "Conference Components" in this book), the conference component configures and manages stream controller components on your behalf. The conference component allows you to override its default handling of stream controller components. If you need greater control of stream controller components or you want to implement application-specific user interface features, you need to read this chapter.

If you are creating a stream controller component, you must support the interface described in this chapter.

The chapter begins with a brief introduction to stream controller components, including an overview of the controls a controller provides, a controller's channel, and the spatial area you use in working with a stream controller. Then it describes how you can

- associate a stream controller with a stream director to manage a channel
- handle stream controller events
- customize stream controller behavior

All stream controller component functions are described in "Stream Controller Components Reference"—you should read the portions that are relevant to your application.

To use or to create a stream controller component, you need to be familiar with the Component Manager, described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

About Stream Controller Components

Stream controller components are similar to QuickTime movie controller components. Both provide controllers, a set of controls by which a user manipulates the data presented by the controller component. Movie controllers allow users to play and edit movies. **Stream controllers** are a set of controls that allow users to control the display and playback of audio-visual data transferred over QuickTime Conferencing connections.

You can think of stream controller components in terms of more familiar Macintosh controls. In addition to handling update, activate, and mouse-down events, stream

controller components know how to interact with the data that they control. Consequently, stream controller components can actually perform the actions requested by users (the controls handled by the Control Manager merely report user actions to your application). In this way, your application is relieved of much of the work of controlling connections. Furthermore, stream controller components can be updated to provide improved performance with no impact on your application.

Stream controller components have a component type value of `'mtcn'`. You can use the constant `kMTControllerType` to specify this value. The Apple-provided stream controller component has the subtype `'mtlk'`, specified by the constant `kMTMovieTalkSubType`.

A Note on Terminology

In this chapter, the terms *video* and *sound* are used generically—that is, they refer to any type of visible or audible media data. This contrasts with their use in some chapters in this book where video and sound refer to specific types of media data.

The term *presentation* is used inclusively to refer to both the display of visual data on the screen and the playback of audio data.

For the sake of brevity in the remainder of this chapter, stream controller components and stream controllers are often referred to as controller components and controllers. ◆

Controls

A stream controller component provides a controller, a set of controls that allow a user to regulate sound, stop and start presentation of a connection, record a connection, take a snapshot of a connection image, and resize the image. Your application can use any of these controls.

From a user's perspective, connections are displayed in onscreen windows—**connection windows**. A **source view** shows the local view—typically, the user's image. A **sink view** shows the view from the remote end of a connection. Each view is controlled by a stream controller.

In most cases, the source view and the sink views are located each in their own windows. Your application can also provide a connection window that contains multiple views.

Figure 4-1 and Figure 4-2 show the standard controllers for source views and sink views. In Figure 4-1, the source view controller is activated and the sink view controller is deactivated, while in Figure 4-2 the reverse is true. Note that controls in a deactivated controller are dimmed or invisible. The standard controls are described in the sections that follow.

Stream Controller Components

Figure 4-1 Activated stream controller for source view

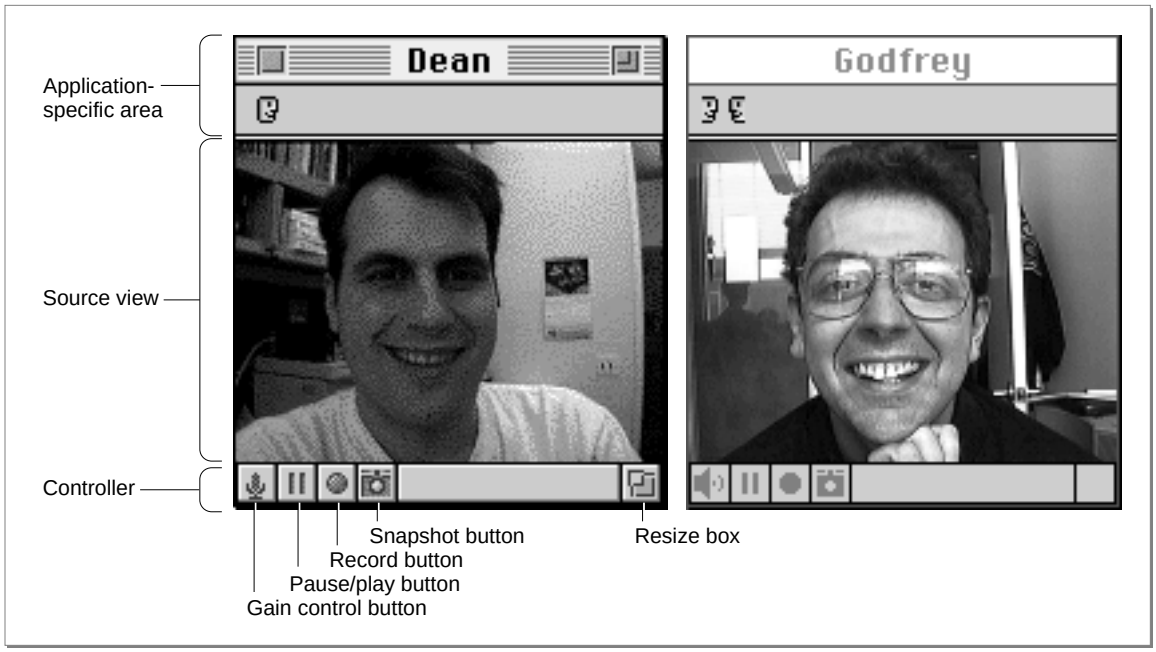
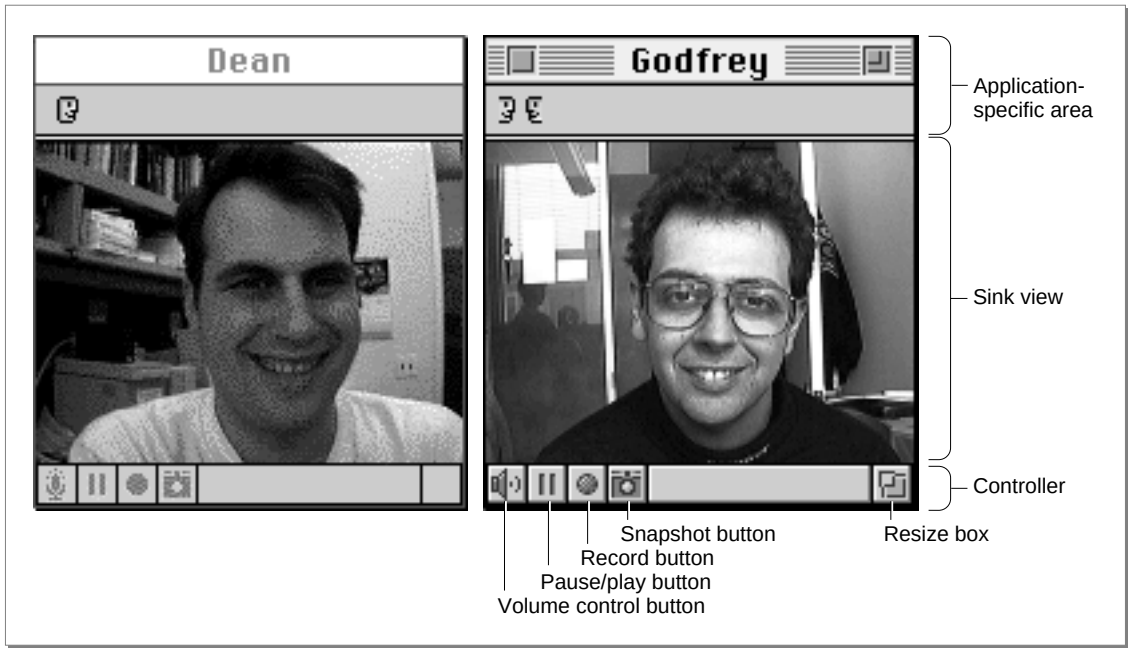


Figure 4-2 Activated stream controller for sink view



Volume Control Button

The volume control button allows a user to adjust the sound volume of incoming audio data. Holding down the mouse button while the cursor is on this button causes the controller to display a slider. The user can adjust the slider to change the sound volume for a sink view.

As a user adjusts the volume using the slider, the icon on the volume control button changes to indicate the amount of audio energy. For example, if a user sets the slider to its lowest level, the icon becomes a speaker with no sound waves. The volume level produced by setting the volume slider to its lowest level varies, depending on the sound settings made by the originating end, and does not necessarily mute the sound. To mute the sound, the user can Option-click the volume control button.

By default, if a channel has no audio streams, the controller component does not display a volume control button.

Gain Control Button

The gain control button allows a user to adjust the audio input gain of a connection. Holding down the mouse button while the cursor is on this button causes the controller to display a slider. The user can adjust the slider to change the gain for audio data to be transmitted to a remote connection end.

As a user adjusts the gain using the slider, the icon on the gain control button changes to indicate the live audio level of a channel by animating the microphone icon. For example, if a user sets the slider to its lowest level, the icon becomes a microphone with no sound waves. Option-clicking the gain button toggles the slider setting. Option-clicking it the first time sets the slider to its lowest level; Option-clicking it the second time restores the slider to its previous level.

If a channel has no sound, the controller component does not display a gain control button. If a channel has sound, but it is not possible to adjust the gain (for example, some audio devices use automatic gain adjustment), the gain control button is visible, but it does not act as a control. However, it does indicate the live audio level of a channel.

Pause/Play Button

The pause/play button lets a user stop and start presentation of a connection. Clicking the pause button for a sink view temporarily stops the presentation—the visual display freezes and the audio is silenced. For a source view, besides stopping the presentation, clicking the pause button also stops the transmission of visual and audio data to the remote connection end. In either case, clicking the pause button changes the pause button to a play button. Clicking the play button resumes the presentation (and transmission, for source views) of live connection data and changes the play button to a pause button.

(While the presentation of a sink view is temporarily stopped, data received from the remote connection end is discarded.)

The standard controller defines the space key as the keyboard equivalent of a pause/play button.

Record Button

The record button lets a user start or stop the recording of audio-visual data presented by a stream controller. Clicking the record button notifies an application or the conference component that a user wants to start recording and causes the record button to begin flashing. Clicking the flashing record button notifies an application or the conference component that a user wants to stop recording and usually (depending on the recorder component in use) saves the connection data to disk as a QuickTime movie.

Snapshot Button

The snapshot button allows a user to capture the image that is currently displayed in a channel. Clicking the snapshot button captures the live image, stores it in a QuickDraw picture, and causes the camera icon on the snapshot button to flash. Your application can display the picture in a new window, put it on the Clipboard, or make it available to the user in some other way. Your application is responsible for saving the picture.

Resize Box

The resize box lets a user change the size of a controller. Holding down the mouse button and moving the mouse while the cursor is on the resize box causes the controller to get larger or smaller.

Holding down both the Shift key and the mouse button and moving the mouse while the cursor is on the resize box constrains the new controller size to have the same aspect ratio as the original controller—its height and width are multiplied by the same value.

Holding down both the Option key and the mouse button and moving the mouse while the cursor is on the resize box constrains the new controller size to integral multiples of its original size.

(Resizing the controller does not cause the window containing the controller to be resized. Your application is responsible for resizing the window. You can detect when a user wants to resize a controller by watching for the `mtControllerActionControllerSizeChanged` action in your action filter function, described later in this chapter.)

If the user resizes the controller so that there is not enough space to display all the individual control elements, the stream controller component eliminates elements from the display.

Channels

In this chapter, a **channel** is defined as the real-time audio-visual data from a QuickTime Conferencing connection, as it is presented to a user by a stream controller component. In this context, you can think of a channel as similar to a television channel in that both typically present audio-visual data to a user.

A **stream** is a logical sequence of **media data**—time-based information, such as video or sound data—that flows between connection ends. A stream is composed of a single type of media data. Each channel consists of one or more streams managed by a stream director component.

For example, in a two-party conference call in which both video and sound are present, there are two channels and four streams at each connection end. One channel contains the data generated locally and sent to the remote party; it consists of two streams—one for video data and one for audio data. The second channel contains the data generated at the remote end and received locally. It also consists of one audio stream and one video stream.

Each channel needs its own controller. In the preceding example, the application needs two controllers. Each controller component instance provides a controller for one channel.

You assign a controller to a channel by passing a stream director component instance to a stream controller component instance. Stream director components manage groups of streams. In the preceding example, two stream director component instances exist at each connection end, one for the two streams flowing from the local end to the remote end, and one for the two streams flowing from the remote end to the local end.

You provide the stream director component instance when you call the `MTControllerNewAttachedController` function. The controller component then finds out from the stream director component the number and nature of the streams in its channel and interacts with the stream director to manage the streams.

(If you later want to change a controller component's channel, you can call the `MTControllerSetStreamDirector` function and specify a different stream director component instance.)

Spatial Elements

Stream controller components define several display regions that govern how a controller and its channel are displayed. In addition, stream controller components support a number of functions that allow your application to manipulate these regions and thereby control the display of a controller and its channel. This section discusses each of these regions and the stream controller component functions that your application can use to work with these regions.

A stream controller and its channel may be either attached or detached. When a controller and its channel are attached, the controller and its channel are contiguous. Your application cannot change the location of an attached controller relative to its channel. Furthermore, a controller and its attached channel are treated as a single unit by many of the functions that manipulate spatial areas.

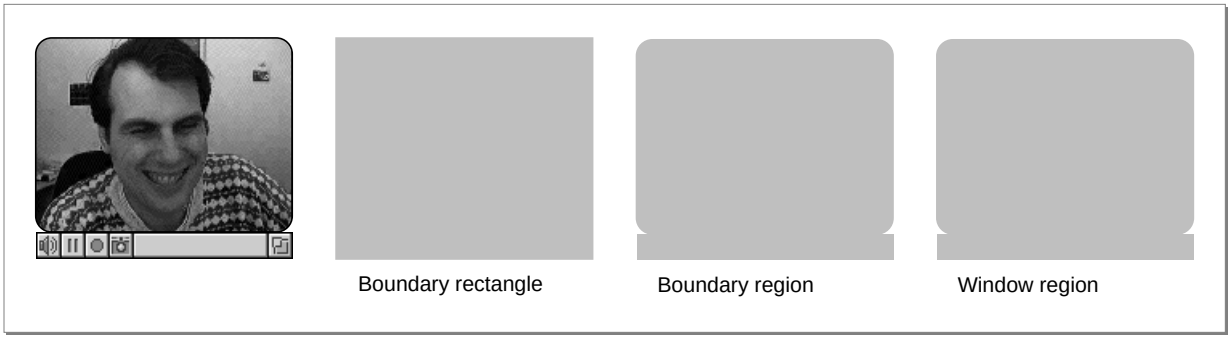
When a controller and its channel are detached, the channel and its controller may be contiguous, overlapping, or separate from each other. You can set and change the position of a detached channel and its controller independently of each other.

A new controller created by the `MTControllerNewAttachedController` function (page 4-31) is always attached to its channel. You can use the `MTControllerSetControllerAttached` function (page 4-37) to detach it.

Figure 4-1 and Figure 4-2 on page 4-5 show sample controllers. In these figures, there are two channels, each attached to its controllers. The channels, in this case, consist of the visible portion of the media streams that flow over a QuickTime Conferencing connection. The controllers consist of the sets of controls that allow a user to control the channels.

Stream controller components define several spatial elements that allow your application to control the display of a channel and its controller. Figure 4-3 shows the relationships between these spatial elements for attached controllers; Figure 4-4 shows the relationships between these spatial elements for detached controllers. You'll find it helpful to compare these figures as you read the following discussion.

Figure 4-3 Stream controller spatial elements for attached controllers

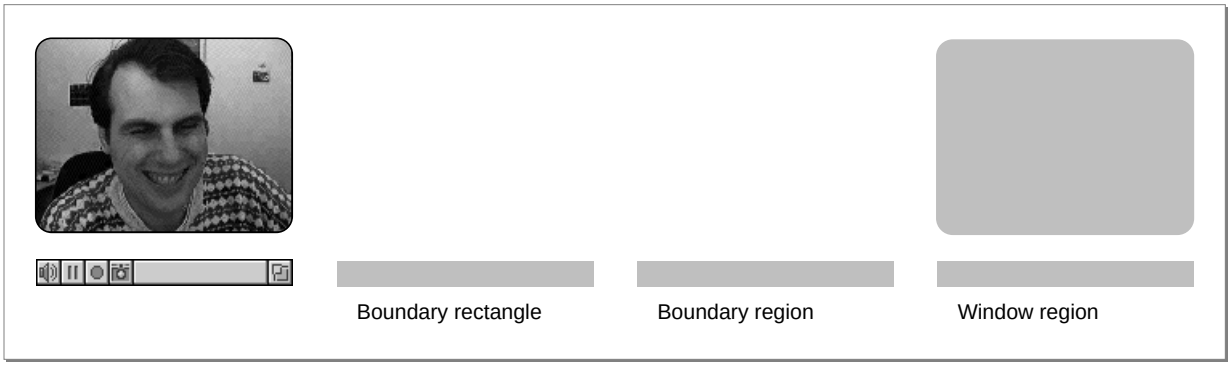


A controller's boundary rectangle is a rectangle that completely encloses the controller. If the controller is attached to its channel, the controller boundary rectangle also encloses the channel. The width of the boundary rectangle corresponds to the widest part of the displayed representation of the controller (and its attached channel). Similarly, its height is derived from the highest part of the controller (and its attached channel). You can use the `MTControllerSetControllerBoundsRect` function (page 4-40) to modify a controller's boundary rectangle. And you can retrieve a controller's boundary rectangle by calling the `MTControllerGetControllerBoundsRect` function (page 4-41).

A controller's boundary region defines the region occupied by a controller. If a controller is attached to its channel, the controller boundary region also includes the channel. A controller's boundary region corresponds exactly to the display footprint of the controller (and its attached channel). You can retrieve the boundary region of a controller by calling the `MTControllerGetControllerBoundsRgn` function (page 4-42).

Note
Figure 4-3, Figure 4-4, and Figure 4-5 show a channel in a rounded rectangle. This permits differentiation of the boundary region from the boundary rectangle in Figure 4-3. In QuickTime Conferencing 1.0, only rectangularly shaped channels are supported. Of course, you can change the shape of a channel by applying a clipping regions. ◆

Figure 4-4 Stream controller spatial elements for detached controllers

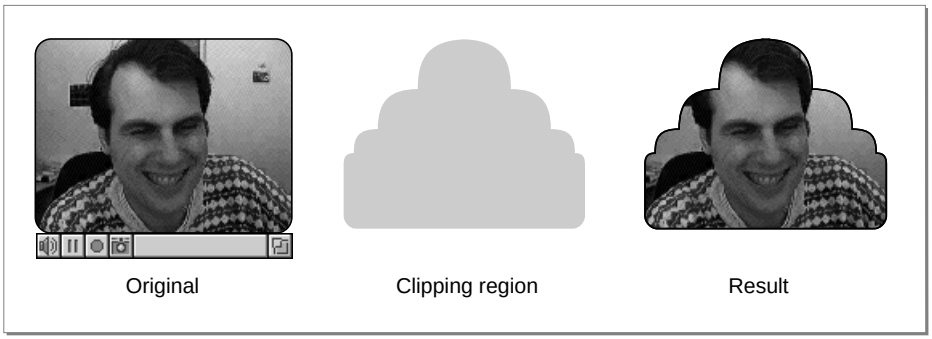


A controller's boundary rectangle and boundary region both are calculated from the unclipped display representation of the controller and its channel. You can manipulate a controller's clipping region by calling the `MTControllerSetClip` (page 4-43) and `MTControllerGetClip` functions (page 4-44). See *Inside Macintosh: Imaging With QuickDraw* for a discussion of clipping regions.

A controller's window region represents the portion of the controller and its channel that is actually displayed on the computer screen, after clipping by the controller clipping region. It is located within the visible region defined by QuickDraw. The controller window region always includes areas occupied by both the controller and its channel, whether the controller is attached or detached, if both are in the same window. (Because you can position a detached controller and its channel separately, you can place them in different windows.) You can retrieve a controller's window region by calling the `MTControllerGetWindowRgn` function (page 4-43).

Figure 4-5 illustrates the effect of applying a controller's clipping region to its boundary region.

Figure 4-5 Applying a clipping region



Using Stream Controller Components

This section discusses how you can

- assign a channel to a stream controller
- keep a stream controller component aware of events that it should handle
- customize the behavior of a stream controller

Assigning a Channel to a Stream Controller

After you have opened a stream controller component (by calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function), you need to create a stream controller and assign a channel to it.

You assign a channel to a stream controller component by providing the instance of the stream director component that manages the group of streams constituting the channel. The `MTControllerNewAttachedController` function (page 4-31) creates a controller and assigns its channel. You provide the function with the stream director component instance and the window in which the controller should appear, as shown here:

```
myErr = MTControllerNewAttachedController
            (myControllerCompInstance, myStreamDirectorInstance,
            theWindow, where);
```

The parameter `myStreamDirectorInstance` contains the stream director component instance that manages the group of streams that you want to assign to the new stream controller. In the `theWindow` parameter, you provide a pointer to the window in which you want the controller and its channel to appear. You set the `where` parameter to the upper-left corner of the channel within the window.

When your application uses the Apple-provided stream controller component, a new controller has the following characteristics:

- It is attached to its channel.
- It is visible.
- Keystroke processing is enabled.
- The record button is not displayed.
- The snapshot button is not displayed.

If you want to modify this default controller, you can do so. For example, you can call the `MTControllerDoAction` function (page 4-50) with the `mtControllerActionSetShowRecord` action to make the record button visible. Or you can call the `MTControllerSetControllerAttached` function (page 4-37) to detach the stream controller from its channel.

A stream controller can support one channel at a time. If you want to assign a different channel to an existing stream controller, you can call the `MTControllerSetStreamDirector` function (page 4-33) to replace an existing channel with a new one. You pass a new stream director component instance to the function to override the current stream director component instance. The streams managed by the new stream director become the new channel.

To find out the existing stream director instance, you can call the `MTControllerGetStreamDirector` function (page 4-34).

Handling Stream Controller Events

To handle the human interface tasks for a QuickTime Conferencing connection, a stream controller component must be aware of events that affect the user interface. The stream controller component API provides functions that allow your application to implement alternative methods to keep a stream controller component informed about events that it should handle.

The most convenient method for most applications is calling the `MTControllerIsControllerEvent` function (page 4-49) in your application's event loop, like this:

```
while (!gDoneFlag) {
   gResult = WaitNextEvent(everyEvent, &gEventRec, gSleep,
                              gMouseRgn);

   if (gResult) {

      /* pass non-null events to stream controller. handle the
         event if the stream controller doesn't
      */
      if (MTControllerIsControllerEvent(gMTControllerPlay,
```

```
                                        &gEventRec) == 0 ) {
         HandleEvent(gEventRec);
      }
   }
}
```

In this example, you pass all events except null events to the controller component. If the event pertains to the controller or the channel it is managing, the controller component handles the event and the `MTControllerIsControllerEvent` function returns the value 1. Otherwise, it returns 0.

If the controller component handles the event, your application should skip the rest of its event loop and wait for the next event. Otherwise, you should call `MTControllerIsControllerEvent` for each stream controller component you are using until the event is handled. If no controller component handles the event, your application must handle the event as part of its normal event processing.

With this method, your application looks at only those events that a controller component does not handle. While the sample code does not pass null events to the controller component, you may do so in your application.

An alternate method is for your application to look at every event and inform the controller component about those that the component should handle. You inform a controller component about events that it should handle by calling functions such as `MTControllerActivate`, `MTControllerClick`, `MTControllerDraw`, and `MTControllerKey`.

Regardless of the method you use to keep a controller component informed about events, the controller component tells your application about an action it is about to take in response to the event, before it actually takes the action. It does this by calling your action filter function, which is discussed in the next section.

(Note that if your application uses a conference component to acquire a controller, the conference component handles controller event processing for you.)

Customizing Stream Controller Behavior

The stream controller component API defines actions that a stream controller can perform. The actions are represented by action codes, which are integer constants. You'll find a complete list of the action codes and a description for each action starting on page 4-19.

An application can direct a stream controller component to perform an action by calling the `MTControllerDoAction` function (page 4-50) and specifying an action code.

A stream controller component notifies an application about an action it is about to take by calling the application's action filter function and specifying an action code.

Most action codes need to be accompanied by additional data that further describes the action. Both the `MTControllerDoAction` function and the action filter function require a pointer to parameter data for an action.

The next two sections, "Getting Actions From a Stream Controller Component" and "Sending Actions to a Stream Controller," provide more detail about the use of actions in your application.

Getting Actions From a Stream Controller Component

You can create an action filter function in your application. A stream controller component calls your action filter function before it performs an action. An action filter function gives your application an opportunity to monitor user actions, customize the behavior of a control, and modify default controller operations with application-specific processing.

You establish an action filter function by calling the `MTControllerSetActionFilter` function (page 4-51). You can provide a reference constant value that the stream controller component passes back to your action filter function when it calls your function.

Note
If your application uses a conference component to acquire a controller, the conference component automatically handles most controller actions for you. However, your application can still establish an action filter function to override the default controller response to actions by calling the `MTConferenceSetDefaultActionFilter` function, described in the chapter "Conference Component" in this book. ◆

The circumstances that cause a controller component to call your action filter function include the following:

- A user takes an action, such as clicking in the controller, resizing the controller, and so forth.
- Another component requests the controller component to perform an action.
- Your application calls the `MTControllerDoAction` function to tell a stream controller component to perform a specific action.

Your action filter function is called often—almost anything an application, a user, or another component does results in the controller component calling your filter function.

You decide what actions your application is interested in. For example, your filter function may filter for the `mtControllerActionControllerSizeChanged` action code so that it can change the window size if the user resizes the controller. Or it may filter for the `mtControllerActionSetVolume` action code so that it can save the volume setting for future connections. It may filter for the `mtControllerActionStartRecord` action code so it can perform an application-specific task before recording begins.

Many calls to your action filter function result from actions your application took. Consider this example: your application calls the `MTControllerDoAction` function with the `mtControllerActionSetFlags` action code to tell a controller component to set the controller flags. Before the component sets the flags, it calls your action filter function with the `mtControllerActionSetFlags` action code to notify you that it is about to set the flags—that is, before it implements the request, it notifies your

application so that you can intervene if you choose. In cases like this, you probably want to ignore the callback to your filter function.

When your filter function completely handles an action, it must return the Boolean value `true`. In this case, the stream controller component performs no additional processing for the action. Otherwise, your filter function must return `false`, which tells the stream controller component to perform appropriate processing for the action.

The sample code in Listing 4-1 demonstrates the use of an action filter function. This sample function handles the `mtControllerActionControllerSizeChanged` action code—the filter function resizes the window whenever the user hides the controller. Your application should include a similar action filter function so that you can determine when the user resizes the controller. This function supports only attached controllers.

Listing 4-1 Using a stream controller filter function

```
pascal Boolean MyMTControllerActionFilter
         (MTControllerComponent mtc, MTControllerActionType action,
         void *params, long myRefCon)
{
   RgnHandle            controllerRgn;
   Rect                 controllerBox;
   WindowPtr            connectionWindow;
   StandardFileReply    recordFileReply;
   Point                where = {-1,-1};
   Boolean              result = false;

   switch(action) {
      case mtControllerActionSetVolume:
         gCurrentVolume = *((short*)params);
      break;

      case mtControllerActionStartRecord:
         if ( *(Boolean*)params ) {
            /* put up a file dialog */
            CustomPutFile("\pSave recorded movie as",
                  "\pRecorded Movie", &recordFileReply, 0, where,
                  nil, nil, nil, nil, nil);

            /* call recorder component functions here to start recording */

            result = true;
         } else {

            /* call recorder component functions here to stop recording */
```

```
            }
      break;

      case mtControllerActionControllerSizeChanged:
      /* size of controller or media data has changed */
         connectionWindow =
                  (WindowPtr)MTControllerGetControllerPort(mtc);
         controllerRgn =
                  MTControllerGetWindowRgn(mtc, connectionWindow);
         if (controllerRgn != nil) {
            controllerBox = (**controllerRgn).rgnBBox;
            DisposeRgn (controllerRgn);
            SizeWindow (connectionWindow, controllerBox.right,
                controllerBox.bottom, true);
         }
       break;

      default:
         break;
   }
   return result;
}
```

Many action codes can be sent by either a controller component or an application. However, there are some that a controller component does not send to an application. A complete list of the action codes and a description for each action begins on page 4-19. Table 4-1 lists those action codes that a controller component does not send to an action filter function.

Table 4-1 Action codes not received by application action filter function

Action code constants
`mtControllerActionSetGrowBoxBounds`
`mtControllerActionGetDragEnabled`
`mtControllerActionSetDragEnabled`
`mtControllerActionSetEnableSnapshot`
`mtControllerActionSetEnableSound`
`mtControllerActionSetEnablePlay`
`mtControllerActionSetEnableRecord`
`mtControllerActionSetEnableGain`
`mtControllerActionGetEnableSnapshot`
`mtControllerActionGetEnableSound`
`mtControllerActionGetEnablePlay`
`mtControllerActionGetEnableRecord`
`mtControllerActionGetEnableGain`
`mtControllerActionSetShowSnapshot`
`mtControllerActionSetShowSound`
`mtControllerActionSetShowPlay`
`mtControllerActionSetShowRecord`
`mtControllerActionSetShowGain`
`mtControllerActionGetShowSnapshot`
`mtControllerActionGetShowSound`
`mtControllerActionGetShowPlay`
`mtControllerActionGetShowRecord`
`mtControllerActionGetShowGain`

Sending Actions to a Stream Controller

An application can tell a stream controller component to perform a specific action or to return some specific information by calling the `MTControllerDoAction` function (page 4-50).

When an application sends an action code to a controller component, the component is required to do something specific. (In contrast, actions sent from a controller component to an application are advisory—they give an application the opportunity to act on the action.)

There are some action codes that an application should not send to a controller component. Table 4-2 lists those action codes that an application should not send. See the section "Action Codes" beginning on page 4-19 for descriptions of all action codes.

Table 4-2 Action codes not sent by applications

Action code constants
`mtControllerActionClick`
`mtControllerActionControllerSizeChanged`
`mtControllerActionShowBalloon`
`mtControllerActionSnapshot`
`mtControllerActionSnapshotData`

Stream Controller Components Reference

This section describes the constants and functions in the API of the stream controller component.

Constants

This section describes the constants in the stream controller component API.

Component Type and Subtype

All stream controller components have the component type `'mtcn'`.

```
enum {
   kMTControllerType    = 'mtcn'
};
```

Apple defines the `'mtlk'` component subtype for the Apple-provided stream controller component.

```
enum {
   kMTMovieTalkSubType  = 'mtlk'
};
```

Request Codes

A request code specifies a function in the stream controller component API. The Component Manager passes the request code to a stream controller component to indicate which function an application called. You can use the following constants to refer to the request codes for the functions that a stream controller component must support.

```
enum {
   kMTControllerSetStreamDirectorSelect         = 1,
   kMTControllerGetStreamDirectorSelect         = 2,
   kMTControllerRemoveStreamDirectorSelect      = 3,
   kMTControllerIsControllerEventSelect         = 4,
   kMTControllerDoActionSelect                  = 5,
   kMTControllerSetControllerAttachedSelect     = 6,
   kMTControllerIsControllerAttachedSelect      = 7,
   kMTControllerSetControllerPortSelect         = 8,
   kMTControllerGetControllerPortSelect         = 9,
   kMTControllerSetVisibleSelect                = 10,
   kMTControllerGetVisibleSelect                = 11,
   kMTControllerSetControllerBoundsRectSelect   = 12,
   kMTControllerGetControllerBoundsRectSelect   = 13,
   kMTControllerGetControllerBoundsRgnSelect    = 14,
   kMTControllerGetWindowRgnSelect              = 15,
   kMTControllerChangedStreamsSelect            = 16,
   kMTControllerNewAttachedControllerSelect     = 17,
   kMTControllerDrawSelect                      = 18,
   kMTControllerActivateSelect                  = 19,
   kMTControllerKeySelect                       = 20,
   kMTControllerClickSelect                     = 21,
   kMTControllerSnapshotSelect                  = 22,
   kMTControllerPositionControllerSelect        = 23,
   kMTControllerGetControllerInfoSelect         = 24,
   kMTControllerSetClipSelect                   = 25,
   kMTControllerGetClipSelect                   = 26,
   kMTControllerSetActionFilterSelect           = 27
};
```

Action Codes

This section describes the action codes defined in the stream controller component API.

An application can send an action (and any necessary parameters to the action) to a stream controller component by calling the `MTControllerDoAction` function, described on page 4-50.

If your application includes an action filter function, a stream controller component sends actions to that function. You can set an action filter function by calling the `MTControllerSetActionFilter` function, described on page 4-51.

Many actions can be sent to either a controller component or an application. However, there are some that an application should not send to a controller component and others that a controller component does not send to an application. See Table 4-1 on page 4-17 and Table 4-2 on page 4-18 for lists of action codes that should be sent in one direction only.

Actions are represented by integer constant values of the `MTControllerActionType` data type.

```
enum {
   mtControllerActionDraw                  = 2,
   mtControllerActionActivate              = 3,
   mtControllerActionDeactivate            = 4,
   mtControllerActionMouseDown             = 5,
   mtControllerActionKey                   = 6,
   mtControllerActionPlay                  = 8,
   mtControllerActionSetVolume             = 14,
   mtControllerActionGetVolume             = 15,
   mtControllerActionSetGain               = 16,
   mtControllerActionGetGain               = 17,
   mtControllerActionSetGrowBoxBounds      = 25,
   mtControllerActionControllerSizeChanged = 26,
   mtControllerActionSetKeysEnabled        = 32,
   mtControllerActionGetKeysEnabled        = 33,
   mtControllerActionSetFlags              = 38,
   mtControllerActionGetFlags              = 39,
   mtControllerActionShowBalloon           = 43,
   mtControllerActionClick                 = 45,
   mtControllerActionSuspend               = 46,
   mtControllerActionResume                = 47,
   mtControllerActionGetDragEnabled        = 51,
   mtControllerActionSetDragEnabled        = 52,
   mtControllerActionStartRecord           = 60,
   mtControllerActionStreamsChanged        = 61,
   mtControllerActionSnapshot              = 62,
   mtControllerActionSnapshotData          = 63,
   mtControllerActionSetEnableSnapshot     = 70,
   mtControllerActionSetEnableSound        = 71,
   mtControllerActionSetEnablePlay         = 72,
   mtControllerActionSetEnableRecord       = 73,
   mtControllerActionSetEnableGain         = 74,
   mtControllerActionGetEnableSnapshot     = 75,
```

```
   mtControllerActionGetEnableSound          = 76,
   mtControllerActionGetEnablePlay           = 77,
   mtControllerActionGetEnableRecord         = 78,
   mtControllerActionGetEnableGain           = 79,
   mtControllerActionSetShowSnapshot         = 80,
   mtControllerActionSetShowSound            = 81,
   mtControllerActionSetShowPlay             = 82,
   mtControllerActionSetShowRecord           = 83,
   mtControllerActionSetShowGain             = 84,
   mtControllerActionGetShowSnapshot         = 85,
   mtControllerActionGetShowSound            = 86,
   mtControllerActionGetShowPlay             = 87,
   mtControllerActionGetShowRecord           = 88,
   mtControllerActionGetShowGain             = 89
};

typedef short MTControllerActionType;
```

Action Descriptions

`mtControllerActionDraw`

An application can use this action to notify a stream controller component that it received an update event.

A stream controller component sends this action to an action filter function after the controller receives an update event and before drawing the controller. Applications should not call the `MTControllerDraw` function in response to this action.

The parameter data is a pointer to the window to which the update event applies.

`mtControllerActionActivate`

An application can use this action to activate a stream controller.

A stream controller component sends this action to an action filter function after the controller receives an activate or resume event and before activating the controller. Applications should not call the `MTControllerActivate` function in response to this action.

There are no parameters for this action.

`mtControllerActionDeactivate`

An application can use this action to deactivate a stream controller.

A stream controller component sends this action to an action filter function after the controller receives a deactivate or suspend event and before deactivating the controller. Applications should not call the

`MTControllerActivate` function in response to this action.

There are no parameters for this action.

`mtControllerActionMouseDown`
An application can use this action to pass a mouse-down event to a stream controller.

A stream controller component sends this action to an action filter function when the controller receives a mouse-down event. When your application calls the `MTControllerClick` function, your action filter function gets this action.

The parameter data is a pointer to an event structure—the `message` field in the event structure specifies the window in which the user clicked.

`mtControllerActionKey`
An application can use this action to pass a key-down or auto-key event to a stream controller.

A stream controller component sends this action to an action filter function when the controller receives a key-down or auto-key event. When your application calls the `MTControllerKey` function, your action filter function gets this action.

The parameter data is a pointer to an event structure that describes the key event.

`mtControllerActionPlay`
An application can use this action to start or stop the presentation of a channel.

A stream controller component sends this action to an action filter function when the controller receives a request to start or stop the presentation of a channel. This can occur, for instance, when a user clicks the pause/play button or your application calls the `MTControllerDoAction` function with the `mtControllerActionPlay` action.

The parameter data is a pointer to a Boolean value. On sending the action to a controller component, set the Boolean value to `true` to start playing the channel. Set it to `false` to stop the channel. On receiving the action from a controller component, the value `true` indicates that the controller has received a request to start playing the channel; `false` indicates it has received a request to stop the channel.

`mtControllerActionSetVolume`
An application can use this action to set a channel's volume. The volume you set applies to all streams in the channel that contain audio data.

A stream controller component sends this action to an action filter function when the controller receives a request to set the channel's volume.

The parameter data is a pointer to a 16-bit, fixed-point number that indicates the relative volume of the channel. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Volume values range from –1.0 to 1.0. Negative values indicate no sound, but preserve the absolute value of the volume setting.

Volume values greater than 1.0 can be specified. Such values overdrive the volume and may result in decreased sound quality.

`mtControllerActionGetVolume`
An application can use this action to retrieve a channel's volume setting.

A stream controller component sends this action to an action filter function when the controller receives a request to retrieve the channel's volume setting.

The parameter data is a pointer to a 16-bit, fixed-point number that indicates the relative volume of the channel. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Volume values range from –1.0 to 1.0. Negative values indicate no sound, but preserve the absolute value of the volume setting.

Volume values greater than 1.0 can be specified. Such values overdrive the volume and may result in decreased sound quality.

`mtControllerActionSetGain`
An application can use this action to set a channel's audio gain.

A stream controller component sends this action to an action filter function when the controller receives a request to set the channel's audio gain.

The parameter data is a pointer to a 16-bit, fixed-point number that indicates the relative gain of the channel. Gain values range from –1.0 to 1.0. Negative values indicate a gain of 0, but preserve the absolute value of the gain setting.

`mtControllerActionGetGain`
An application can use this action to determine a channel's audio gain.

A stream controller component sends this action to an action filter function when the controller receives a request to retrieve the channel's audio gain setting.

The parameter data is a pointer to a 16-bit, fixed-point number that indicates the relative gain of the channel. Gain values range from –1.0 to 1.0. Negative values indicate a gain of 0, but preserve the absolute value of the gain setting.

`mtControllerActionSetGrowBoxBounds`
An application can use this action to set the limits for resizing a controller (and its channel, if attached). A user cannot use the controller's resize box

to resize the controller beyond the limits you set.

A stream controller component does not send this action to an action filter function.

The parameter data is a pointer to a `Rect` structure. The `Rect` structure contains the minimum size and maximum size, in pixels, of the controller and its attached channel. The top and left values specify the minimum height and width; the bottom and right values specify the maximum height and width.

If you provide an empty `Rect` structure (all values set to 0), the resize box is not displayed.

`mtControllerActionControllerSizeChanged`

An application should not send this action to a controller component. To tell a controller component to resize the stream controller, call the `MTControllerSetBoundsRect` function.

A stream controller component sends this action to an action filter function when it receives a request to resize the stream controller—the controller component issues this action before it updates the screen. This allows your application, before the user sees the resized controller, to change the controller's location or appearance, or to resize the window containing the controller, based on the new controller size.

The parameter data contains a pointer to a Boolean value. The stream controller component sets this value to `true` if the size change results from a change in a stream's media data format. It sets the value to `false` if the size change results from a user resizing a controller or your application calling the `MTControllerSetBoundsRect` function.

`mtControllerActionSetKeysEnabled`

An application can use this action to enable or disable keys for a controller. For example, a controller component can define the space key to act as a play/pause button—that is, a user can start and stop the real-time display of a channel by pressing the space key. The significance of particular key varies with the controller component.

A stream controller component sends this action to an action filter function when it receives a request to enable or disable keys for its controller.

The parameter data is a pointer to a Boolean value. On sending the action to a controller component, set the Boolean value to `true` to enable keys. Set it to `false` to disable keys. On receiving the action from a controller component, the value `true` indicates a request to enable keys; `false` indicates a request to disable keys.

By default, keys are disabled for the Apple-provided controller, and it defines the space key as a play/pause button.

`mtControllerActionGetKeysEnabled`
An application can use this action to determine whether keys are enabled for a stream controller.

A stream controller component sends this action to an action filter function when it receives a request to indicate whether keys are enabled for its controller.

The parameter data is a pointer to a Boolean value. The stream controller component sets this value to `true` if keys are enabled for its controller. Otherwise, it sets the value to `false`.

`mtControllerActionSetFlags`
An application can use this action to set a channel's control flags.

A stream controller component sends this action to an action filter function when it receives a request to set the channel's control flags.

The parameter data is a pointer to a long integer that contains the control flag values. The following flags are defined:

`mtControllerFlagSuppressChannelFrame`
Specifies whether the controller displays a frame around the channel. On sending this action to a controller component, set this flag to 0 if you want the controller to display a frame around the visible media data in the channel. Set it to 1 if you want to suppress the frame. On receiving this action from a controller component, if the flag is set to 0, the controller displays a frame around the channel; if the flag is set to 1, the controller does not display a frame.

By default, the Apple-provided controller does not display a frame.

`mtControllerFlagSuppressSpeakerButton`
Specifies whether the controller displays the volume button. The volume button allows the user to control the channel's sound. On sending this action to a controller component, set this flag to 0 if you want the controller to display the volume button. Set it to 1 if you want to suppress the volume button. On receiving this action from a controller component, if this flag is set to 0, the controller displays the volume button; if this flag is set to 1, the controller does not display the volume button.

You can also use the `mtControllerActionSetShowSound` action to display or hide the volume button.

By default, the Apple-provided controller displays the volume button if the channel has audio data.

`mtControllerFlagsUseWindowPalette`

On sending this action to a controller component, set this flag to 0 if you want the controller to use but not change the color palette associated with a window when the controller displays channel data in that window. Set this flag to 1 if you want the controller to actively manage the window's palette, including changing it to achieve optimal display. On receiving this action from a controller component, if this flag is set to 0, the controller uses the color palette associated with a window without changing it when displaying data in that window. If this flag is set to 1, the controller actively manages the window's palette, including changing it to achieve optimal display.

By default, the Apple-provided controller does not change the color palette associated with a window.

`mtControllerFlagsDontInvalidate`

On sending this action to a controller component, set this flag to 0 if you want the controller to invalidate areas of the window that change as a result of repositioning the channel or the controller. Set it to 1 if you don't want the controller to invalidate areas of the window that change. On receiving this action from a controller component, if this flag is set to 0, the stream controller component invalidates areas of the window that change as a result of repositioning the channel or the controller; if this flag is set to 1, the stream controller component doesn't invalidate areas of the window that change.

If you set this flag to 1, you must make sure that changed windows are redrawn properly.

By default, the Apple-provided controller invalidates areas of the window that change as a result of repositioning the channel or the controller.

`mtControllerActionGetFlags`

An application can use this action to retrieve a channel's control flags.

A stream controller component sends this action to an action filter function when it receives a request to retrieve the channel's control flags.

The parameter data is a pointer to a long integer. The stream controller places the channel's control flags into that long integer. The channel control flags are defined in the description of the `mtControllerActionSetFlags` action immediately above.

`mtControllerActionShowBalloon`

Applications don't send this action.

A stream controller component sends this action to an action filter function when it wants to display a help balloon. This action allows you

to override the stream controller's default Balloon Help behavior. Your filter function tells the controller whether it should display its help balloon.

The parameter data contains a pointer to a Boolean value. You set the value to `true` to cause the controller to display its own help balloon. Otherwise, set the value to `false`.

`mtControllerActionClick`
Applications don't send this action.

A stream controller component sends this action to an action filter function when a user clicks in the channel displayed on the screen. Note that when a user clicks in the channel, the controller component sends two actions to the action filter function: the `mtControllerActionMouseDown` action that it sends for every mouse-down event and the `mtControllerActionClick` action that it sends only when the mouse-down event occurred in the channel.

The parameter data contains a pointer to an event structure containing the mouse-down event.

`mtControllerActionSuspend`
An application can use this action to tell a controller that it has received a suspend event.

A stream controller component sends this action to an action filter function when it receives a suspend event via the `MTControllerIsControllerEvent` function.

There is no parameter for this action.

`mtControllerActionResume`
An application can use this action to tell a controller that it has received a resume event.

A stream controller component sends this action to an action filter function when it receives a resume event via the `MTControllerIsControllerEvent` function.

There is no parameter for this action.

`mtControllerActionGetDragEnabled`
An application can use this action to determine whether dragging of snapshots is enabled for a stream controller.

Stream controller components don't send this action.

The parameter data is a pointer to a Boolean value. The stream controller sets this value to `true` if dragging is enabled for the channel assigned to this controller. Otherwise, it sets the value to `false`.

`mtControllerActionSetDragEnabled`

An application can use this action to enable or disable dragging of snapshots from a channel. The Drag Manager must be installed to take advantage of this feature.

Stream controller components don't send this action.

The parameter data is a Boolean value. Set it to `true` to enable dragging of snapshots. Set it to `false` to disable dragging of snapshots. By default, the Apple-provided controller allows dragging of snapshots.

`mtControllerActionStartRecord`

An application can use this action to tell a controller to start or stop recording data from its channel.

A stream controller component sends this action to an action filter function when a user clicks an enabled record button.

The parameter data is a pointer to a Boolean value. On sending this action to a controller component, set it to `true` to start recording. Set it to `false` to stop recording. On receiving this action from a controller component, the value `true` indicates that a user clicked the record button to start recording the channel; `false` indicates the user clicked the flashing record button to stop recording.

`mtControllerActionStreamsChanged`

Applications don't send this action.

A stream controller component sends this action to an action filter function when a stream format negotiation has completed. This action allows your application to perform application-specific tasks when stream formats change. If your application does not use a conference component, you need to call the `MTControllerDoAction` function with the `mtControllerActionPlay` action after a successful negotiation to start the flow of media data in the connection. If a recording is in progress, you also need to call the `MTRecorderChangedStreams` function to advise the recorder component that a stream format change has occurred. If your application uses a conference component, the conference component does all this for you when your action filter function returns `false`.

The parameter data contains a pointer to a value of type `ComponentResult`. When set to `0`, the value indicates that a stream format negotiation completed successfully. Otherwise, it is the result code from a failed negotiation.

`mtControllerActionSnapshot`

Applications don't send this action.

A stream controller component sends this action to an action filter function when it receives a request to capture a picture of the image currently in its channel. This can happen, for instance, when a user presses the snapshot button in the controller, drags an image to the desktop, or copies the contents of a channel or when your application

calls the `MTControllerSnapshot` function.

(When a user clicks an enabled snapshot button, your filter function also receives the `mtControllerActionSnapshotData` action with identical parameter data.)

Rather than notifying you before performing the action as it does for other actions, the stream controller component first captures the image and then notifies your action filter function. Your application can then act on the picture data before the controller component returns it to the requestor. For example, you can compress the image. Your application should not dispose of the handle because it is in use.

The parameter data contains a pointer to a handle to a QuickDraw picture of the channel. The controller component provides the handle containing the picture. If an error occurs when taking the snapshot, the controller component sets the picture handle to `nil`.

`mtControllerActionSnapshotData`

Applications don't send this action.

A stream controller component sends this action to an action filter function when a user clicks an enabled snapshot button.

(When a user clicks an enabled snapshot button, your filter function also receives the `mtControllerActionSnapshot` action with identical parameter data.)

Rather than notifying you before performing the action as it does for other actions, the stream controller component first captures the image and then provides it to your action filter function. Your application is responsible for disposing of this handle.

The parameter is a pointer to a handle to a picture to a QuickDraw picture of the channel. The controller component provides the handle containing the picture. If an error occurs when taking the snapshot, the controller component sets the picture handle to `nil`.

`mtControllerActionSetEnableSnapshot`

`mtControllerActionSetEnableSound`

`mtControllerActionSetEnablePlay`

`mtControllerActionSetEnableRecord`

`mtControllerActionSetEnableGain`

Applications use these actions to enable and disable the standard controls provided by the stream controller.

Stream controller components don't send these actions.

The parameter data is a pointer to a Boolean value. Set it to `true` to enable the appropriate control and to `false` to disable it.

`mtControllerActionGetEnableSnapshot`

`mtControllerActionGetEnableSound`

`mtControllerActionGetEnablePlay`

`mtControllerActionGetEnableRecord`

`mtControllerActionGetEnableGain`

Applications use these actions to determine whether the standard controls are enabled or disabled.

Stream controller components don't send these actions.

The parameter data is a pointer to a Boolean value. The controller component returns `true` if the control is enabled and `false` if it is disabled.

`mtControllerActionSetShowSnapshot`

`mtControllerActionSetShowSound`

`mtControllerActionSetShowPlay`

`mtControllerActionSetShowRecord`

`mtControllerActionSetShowGain`

Applications use these actions to make the standard controls visible or invisible in the stream controller.

Stream controller components don't send these actions.

The parameter data is a pointer to a Boolean value. Set it to `true` to make the control visible and to `false` to make it invisible.

`mtControllerActionGetShowSnapshot`

`mtControllerActionGetShowSound`

`mtControllerActionGetShowPlay`

`mtControllerActionGetShowRecord`

`mtControllerActionGetShowGain`

Applications use these actions to determine whether the standard controls are visible or invisible.

Stream controller components don't send these actions.

The parameter data is a pointer to a Boolean value. The controller returns `true` if the control is visible and `false` if it is invisible.

Functions

This section describes the functions that are supported by stream controller components. It is divided into the following topics:

- "Assigning Channels to Stream Controllers" describes the functions that allow applications to assign a channel to a controller.

- "Getting and Setting Stream Controller Characteristics" describes the functions that allow applications to alter the display characteristics of the controller.
- "Capturing a Channel Image" describes the function that allows applications to capture images from channels.
- "Handling Events" describes the functions that applications use to handle channel actions.
- "Customizing Event Processing" describes the functions that allow applications to perform customized event processing.

These functions are described from the perspective of the developer of an application that uses stream controllers. If you are developing a stream controller component, your component must behave as described here.

Assigning Channels to Stream Controllers

You assign a channel to a stream controller by specifying the stream director component instance that manages the streams in that channel. The functions described in this section allow you to

- create a stream controller and assign its channel
- assign a new channel to an existing stream controller
- get a reference to a controller's channel
- remove a controller's channel

MTControllerNewAttachedController

The `MTControllerNewAttachedController` function creates a new controller and associates it with a stream director (channel) that you specify.

```
pascal ComponentResult MTControllerNewAttachedController
            (MTControllerComponent mtc, MTDirectorComponent sd,
            WindowPtr w, Point where);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`sd` The stream director component instance managing the group of streams that is to become the stream controller's channel. You provide this value.

`w` A pointer to the window in which you want the controller and its channel to be displayed. The stream controller component sets the graphics world of the controller and its channel to match this window. If you set the `w` parameter to `nil`, the component uses the current window.

`where` The upper-left corner of the channel within the window specified by the `w` parameter. You provide this value. The stream controller component uses the channel's boundary rectangle to determine the size of the channel.

DESCRIPTION

The function causes the Window Manager to generate an update event for the window that you specify. When it receives that event, the controller component calls your action filter function with the `mtControllerActionDraw` action to give your application a chance to modify the default controller. Then it draws the controller and its channel in the window.

SPECIAL CONSIDERATIONS

The `MTControllerNewAttachedController` function causes a stream controller component to install a negotiation callback function on the stream director component that you specify. Negotiations occur when a connection is established and whenever stream formats change during an active connection. The stream director calls the negotiation callback function to inform the controller component about the progress of stream format negotiations. The controller component informs your application when negotiations complete by sending the `mtControllerActionStreamsChanged` action to your action filter function. For most applications, this is sufficient. However, if your application needs more detailed information about the progress of stream format negotiations, you can install your own negotiation callback function on a stream director.

RESULT CODES

`noErr`	0	No error

SEE ALSO

For a description of the default controller, see the section "Channels" on page 4-7.

You can set an action filter function with the `MTControllerSetActionFilter` function, described on page 4-51.

You can find information about stream format negotiations and negotiation callback functions in the chapter "Stream Director Components" in this book.

If you want to change the channel for a controller, you must call the `MTControllerSetStreamDirector` function, described next.

MTControllerSetStreamDirector

The `MTControllerSetStreamDirector` function associates a stream director that you specify with a stream controller, thus assigning a particular channel to the controller.

```
pascal ComponentResult MTControllerSetStreamDirector
            (MTControllerComponent mtc, MTDirectorComponent sd,
            WindowPtr displayWindow, Point where);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`sd` The stream director component instance that you want to associate with the stream controller.

`displayWindow`
A pointer to the window in which you want the channel to be displayed. The stream controller component sets the channel's graphics world to match this window. If you set the w parameter to `nil`, the component uses the current window.

`where` The upper-left corner of the channel within the window specified by the `displayWindow` parameter. The stream controller component uses the channel's boundary rectangle to determine the size of the channel.

DESCRIPTION

Once you create a controller with the `MTControllerNewAttachedController` function, you can use the `MTControllerSetStreamDirector` function to associate a different stream director with that controller.

This function is not typically used, but it is included for flexibility.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTControllerNewAttachedController` function is described on page 4-31.

MTControllerGetStreamDirector

The `MTControllerGetStreamDirector` function returns the stream director component instance associated with the stream controller component that you specify.

```
pascal MTDirectorComponent MTControllerGetStreamDirector
            (MTControllerComponent mtc);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

DESCRIPTION

If there is no stream director associated with the controller, the function returns `nil`.

MTControllerRemoveStreamDirector

The `MTControllerRemoveStreamDirector` function removes the association between a stream director component and a stream controller component.

```
pascal ComponentResult MTControllerRemoveStreamDirector
            (MTControllerComponent mtc);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

DESCRIPTION

You call the `MTControllerRemoveStreamDirector` function to remove a currently existing association between a stream director component and the controller component that you specify.

It is not necessary to call `MTControllerRemoveStreamDirector` before calling the `MTControllerSetStreamDirector` function to associate a different stream director component with a controller component.

This function is not typically used, but it is included for flexibility.

RESULT CODES

`noErr`	0	No error

Getting and Setting Stream Controller Characteristics

The functions described in this section allow you to set certain characteristics of a stream controller and to find out the current settings of characteristics. Most of these characteristics pertain to the visual display of a controller and its channel. For a given stream controller, you can

- position a controller and its channel separately, allowing you to manage the controller and its channel as separate graphics entities
- attach and detach a stream controller and its channel
- determine if a controller is attached to its channel
- make a stream controller visible or invisible
- determine if a controller is visible
- get and set the controller's boundary rectangle
- get the controller's boundary region
- get the controller's window region
- get and set the controller's clipping region
- get and set the controller's graphics port
- get information about the controller component's capabilities and current status

The section "Spatial Elements" beginning on page 4-8 discusses several of the display characteristics.

MTControllerPositionController

The `MTControllerPositionController` function allows you to control the position of a controller and its channel on the screen.

```
pascal ComponentResult MTControllerPositionController
            (MTControllerComponent mtc, const Rect *imageRect,
            const Rect *controllerRect, long someFlags);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`imageRect` A pointer to a `Rect` structure that you set to the coordinates of the channel's boundary rectangle. If the channel is attached to its controller, set the boundary rectangle to enclose the controller also.

`controllerRect`

A pointer to a `Rect` structure that you set to the coordinates of a detached controller's boundary rectangle. If you are working with an attached controller, set this parameter to `nil`.

`someFlags` A set of bit flags that you can use to control how a channel and its controller are drawn. The following flags are defined:

`mtControllerTopLeftChannel`

If you set this flag to 1, the stream controller component places the channel into the upper-left corner of the rectangle specified by the `imageRect` parameter. Otherwise, the stream controller component centers the channel in the rectangle.

This flag has an effect only if the `mtControllerScaleChannelToFit` flag is set to 0.

`mtControllerScaleChannelToFit`

If you set this flag to 1 and the channel is smaller than the rectangle specified by the `imageRect` parameter, the stream controller component resizes the channel to fit the rectangle. Otherwise, it positions the channel based on the setting of the `mtControllerTopLeftChannel` flag.

(The Apple-provided stream controller component always scales a channel to fit when the channel is bigger than the rectangle specified by the `imageRect` parameter.)

`mtControllerNotVisible`

If you set this flag to 1, the stream controller component makes the controller invisible. Otherwise, the controller is visible.

`mtControllerWithFrame`

If you set this flag to 1, the stream controller component draws a frame around the channel. Otherwise, the frame is suppressed.

`mtControllerPositionDontInvalidate`

If you set this flag to 1, the stream controller component doesn't invalidate areas of the window that change as a result of repositioning the channel or the controller. Otherwise, the stream controller component invalidates areas of the window that change. This flag is useful if your application uses a stream controller as part of a larger document. In particular, if you scroll a document using the `ScrollRect` function, optimal redrawing occurs (that is, scrolled areas are not redrawn) if this flag is set.

DESCRIPTION

You call the `MTControllerPositionController` function to position a controller and its channel on the screen. If you are working with a detached controller and channel, you need to provide a boundary rectangle for each.

By using the `mtControllerTopLeftChannel` and `mtControllerScaleChannelToFit` flags, you can control how a controller and channel are drawn. Table 4-3 lists the visual results you get with the Apple-provided stream controller component, depending on how you set these flags.

Table 4-3 How flag settings affect the way the controller and channel are drawn

Top left flag	Scale to fit flag	Visual result
0	0	If the channel is larger than the boundary rectangle, the channel is scaled to fit within the boundary rectangle. Otherwise, the controller and channel are centered in the boundary rectangle.
1	0	If the channel is larger than the boundary rectangle, the channel is scaled to fit within the boundary rectangle. Otherwise, the controller and channel are aligned to the upper left corner of the boundary rectangle.
0	1	The controller and channel fill the boundary rectangle.
1	1	The controller and channel fill the boundary rectangle.

RESULT CODES

`noErr`	0	No error

SEE ALSO

For descriptions of the `Rect` structure and the `ScrollRect` function, see *Inside Macintosh: Imaging With QuickDraw*.

MTControllerSetControllerAttached

The `MTControllerSetControllerAttached` function attaches or detaches a stream controller and its channel.

```
pascal ComponentResult MTControllerSetControllerAttached
            (MTControllerComponent mtc, Boolean attach);
```

`mtc`	The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`attach`	A Boolean value that specifies whether to attach a controller to its channel. Set this parameter to `true` to attach the controller to its channel. Set it to `false` to detach the controller.

DESCRIPTION

By default, a new stream controller is attached to its channel. You can detach it and attach it again using the `MTControllerSetControllerAttached` function.

An attached channel is contiguous with its controller. A detached channel may be contiguous, but it need not be. An attached channel and controller are treated as a single unit by many functions in the stream controller API. Detached channels and controllers are manipulated separately.

SPECIAL CONSIDERATIONS

Your application should not make any assumptions about the location of an attached stream controller with respect to its channel. The controller may be above, below, or surrounding the channel.

RESULT CODES

`noErr`	0	No error

SEE ALSO

If you need to know the location of the controller, you can use the `MTControllerGetControllerBoundsRect` function, described on page 4-41, to obtain its boundary rectangle.

See the section "Spatial Elements" beginning on page 4-8 for information about attached and detached stream controllers relative to certain spatial elements.

MTControllerIsControllerAttached

The `MTControllerIsControllerAttached` function tells you whether a stream controller is attached to its channel.

```
pascal ComponentResult MTControllerIsControllerAttached
            (MTControllerComponent mtc);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

DESCRIPTION

The `MTControllerIsControllerAttached` function returns a value of 1 if the controller is attached. If the controller is not attached, the function returns 0.

SEE ALSO

You can use the `MTControllerSetControllerAttached` function, described on page 4-37, to attach or detach a stream controller and its channel.

MTControllerSetVisible

The `MTControllerSetVisible` function allows your application to control whether a stream controller is visible.

```
pascal ComponentResult MTControllerSetVisible
            (MTControllerComponent mtc, Boolean visible);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`visible` A Boolean value that specifies whether to make a controller visible. Set this parameter to `true` to make the controller visible. Set it to `false` to make the controller invisible.

DESCRIPTION

By default, a newly created controller is visible. You can call the `MTControllerSetVisible` function to make the controller visible or invisible.

While a controller is invisible, the stream controller component remains active and your application can call other functions to manage the controller and its channel.

A stream controller must be visible for a user to control the associated channel.

Making the controller invisible does not affect the visibility of the channel.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You create new stream controllers by calling the `MTControllerNewAttachedController` function, described on page 4-31.

You can use the `MTControllerGetVisible` function, described next, to determine whether a stream controller is visible.

MTControllerGetVisible

The `MTControllerGetVisible` function tells you whether a stream controller is visible.

```
pascal ComponentResult MTControllerGetVisible
            (MTControllerComponent mtc);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

DESCRIPTION

If the controller is visible, the `MTControllerGetVisible` function returns the value 1. If the controller is invisible, the function returns 0.

SEE ALSO

You can use the `MTControllerSetVisible` function, described on page 4-39, to make a stream controller visible or invisible.

MTControllerSetControllerBoundsRect

The `MTControllerSetControllerBoundsRect` function lets you change the position and size of a stream controller.

```
pascal ComponentResult MTControllerSetControllerBoundsRect
            (MTControllerComponent mtc, const Rect *bounds);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`bounds` A pointer to a `Rect` structure containing the boundary rectangle that you want to set for the stream controller.

DESCRIPTION

A controller's boundary rectangle encloses the controller. When a channel is attached to the controller, the boundary rectangle also encloses the channel. If the channel is attached to the controller, changing the size of the boundary rectangle may result in the channel being resized as well.

RESULT CODES

`noErr`	0	No error

SEE ALSO

To find the dimensions of the new boundary rectangle, call the `MTControllerGetControllerBoundsRect` function, described next.

MTControllerGetControllerBoundsRect

The `MTControllerGetControllerBoundsRect` function returns a stream controller's boundary rectangle.

```
pascal ComponentResult MTControllerGetControllerBoundsRect
            (MTControllerComponent mtc, Rect *bounds);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`bounds` A pointer to a `Rect` structure. The function sets the fields of the structure to the global coordinates of the stream controller's boundary rectangle.

DESCRIPTION

You call the `MTControllerGetControllerBoundsRect` function to get a stream controller's boundary rectangle. This rectangle reflects the size and location of the controller even if the controller is currently invisible. If the controller is detached from its channel, the rectangle encloses only the controller, not the channel. If the controller is attached to its channel, the rectangle encloses both the controller and the channel.

The returned rectangle is the boundary rectangle for the region occupied by the controller (and its channel, if the channel is attached to the controller), even if the controller is not rectangular.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can use the `MTControllerGetControllerBoundsRgn` function, described next, to obtain the boundary region for a controller.

The `MTControllerGetWindowRgn` function, described on page 4-43, tells you in what portion of a window a controller is visible.

MTControllerGetControllerBoundsRgn

The `MTControllerGetControllerBoundsRgn` function returns a handle to the boundary region enclosing a controller (and its channel, if attached to the controller).

```
pascal RgnHandle MTControllerGetControllerBoundsRgn
            (MTControllerComponent mtc);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

DESCRIPTION

You call the `MTControllerGetControllerBoundsRgn` function to get the actual region occupied by a controller and its channel, if the channel is attached to the controller. If the channel is not attached to its controller, the function returns the boundary region that encloses only the controller.

This function is useful with controllers that are not rectangular in shape. The `MTControllerGetControllerBoundsRgn` function returns a region that describes the actual size, shape, and location of the controller.

The function returns a region even if the controller is invisible.

Your application must dispose of the returned region handle.

The rectangle returned by the `MTControllerGetControllerBoundsRect` function bounds the region returned by `MTControllerGetControllerBoundsRgn`.

SEE ALSO

You can use the `MTControllerGetWindowRgn` function, described next, to determine the area of a window in which a controller, a channel, or both are visible.

If a controller is rectangular, the region returned by `MTControllerGetControllerBoundsRgn` describes the same area as the rectangle

returned by the `MTControllerGetControllerBoundsRect` function, described on page 4-41.

MTControllerGetWindowRgn

The `MTControllerGetWindowRgn` function returns a handle to a region. The region defines the area of a window in which a controller, a channel, or both are visible.

```
pascal RgnHandle MTControllerGetWindowRgn
            (MTControllerComponent mtc, WindowPtr w);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`w` A pointer to the window in which the stream controller and its channel are displayed. If the controller and its channel are detached and are located in separate windows, you must specify one of the windows.

DESCRIPTION

You call the `MTControllerGetWindowRgn` function to get the region, located within a window's visible region, that contains just the visible portions of a controller and its channel.

The function returns the region that defines the visible parts of a controller and its channel, whether they are attached or not, as long as they are in same window. If a controller is detached from and not contiguous with its channel, the returned region defines separate areas for the channel and the controller. If you want just the controller region, you must subtract the channel region from the returned region.

If a detached controller and its channel are located in separate windows, the function returns the region that defines the visible part of either the controller or its channel, depending on which window you specify.

Your application must dispose of the returned region.

MTControllerSetClip

The `MTControllerSetClip` function sets stream controller and channel clipping regions.

```
pascal ComponentResult MTControllerSetClip
            (MTControllerComponent mtc,
            RgnHandle theClip, RgnHandle imageClip);
```

`mtc`	The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`theClip`	A handle to a region that defines the clipping region of both the stream controller and its channel. To clear this clipping region, set this parameter to `nil`.
`imageClip`	A handle to a region that defines the clipping region of the controller's channel. This clipping region affects only the channel and not the stream controller. Set this parameter to `nil` to clear the channel clipping region.

DESCRIPTION

You can call the `MTControllerSetClip` function to set two clipping regions. The clipping region that you specify in the `theClip` parameter applies to both a controller and its channel, regardless of whether they are attached or detached. This is true even if you have placed a detached controller and its channel in different windows.

The clipping region that you specify in the `imageClip` parameter applies to the channel alone.

Your application is responsible for disposing of the regions you supply to the `MTControllerSetClip` function.

This function can return Memory Manager errors.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can retrieve controller and channel clipping regions by calling the `MTControllerGetClip` function, described next.

Clipping regions and the `Region` structure are discussed in *Inside Macintosh: Imaging With QuickDraw*.

MTControllerGetClip

The `MTControllerGetClip` function retrieves stream controller and channel clipping regions.

```
pascal ComponentResult MTControllerGetClip
            (MTControllerComponent mtc,
            RgnHandle *theClip, RgnHandle *imageClip);
```

`mtc`	The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`theClip`	A pointer to a region handle.The function allocates the memory for the `Region` structure and provides a handle to it. The function sets the structure fields to the clipping region of the stream controller and its channel. If you are not interested in this information, set this parameter to `nil`. If no clipping region has been set, the function sets this parameter to `nil`.
`imageClip`	A pointer to a region handle. The function allocates the memory for the `Region` structure and provides a handle to it. The function sets the structure fields to the clipping region of the controller's channel. If you are not interested in this information, set this parameter to `nil`. If no clipping region has been set, the function sets this parameter to `nil`.

DESCRIPTION

The `MTControllerGetClip` function returns two region handles:

- `theClip`: the region that defines the clipping region for both a controller and its channel, whether they are attached or not
- `imageClip`: the region that defines the clipping region for the channel alone

If you want just the controller's clipping region, you must subtract the channel clipping region from the combined clipping region.

Your application is responsible for disposing of the regions returned by the function.

This function can return Memory Manager errors.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can set controller and channel clipping regions by calling the `MTControllerSetClip` function, described on page 4-43.

Clipping regions and the `Region` structure are discussed in *Inside Macintosh: Imaging With QuickDraw*.

MTControllerSetControllerPort

The `MTControllerSetControllerPort` function allows your application to set the graphics port for a stream controller.

```
pascal ComponentResult MTControllerSetControllerPort
            (MTControllerComponent mtc, CGrafPtr gp);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`gp` A pointer to a new graphics port for the stream controller. Set this parameter to `nil` to use the current graphics port.

DESCRIPTION

You can use this function to place a stream controller and its channel in different graphics ports.

If you are using an attached controller, this function sets the graphics port of both the controller and the channel.

If you are using a detached controller, this function sets only the graphics port of the controller. To change the graphics port of a detached channel, you can

- call the `MTControllerSetControllerAttached` function to attach the controller to its channel
- call the `MTControllerSetControllerPort` function to set the graphics port of the attached controller and channel
- call the `MTControllerSetControllerAttached` function to detach the controller from its channel
- call the `MTControllerSetControllerPort` function to set the graphics port of the detached controller

When you call `MTControllerSetControllerPort`, some stream controller components may use the foreground and background colors from the graphics port to color the stream controller.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTControllerSetControllerAttached` function is described on page 4-37.

You can get a stream controller graphics port by calling the `MTControllerGetControllerPort` function, described next.

MTControllerGetControllerPort

The `MTControllerGetControllerPort` function returns a stream controller's graphics port.

```
pascal CGrafPtr MTControllerGetControllerPort
            (MTControllerComponent mtc);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

DESCRIPTION

You call the `MTControllerGetControllerPort` function to get the graphics port for a stream controller.

If you are using an attached controller, the returned graphics port applies to the controller and the channel.

If you are using a detached controller, the returned graphics port only applies to the controller.

SEE ALSO

To set the graphics port for a controller, use the `MTControllerSetControllerPort` function, described on page 4-46.

MTControllerGetControllerInfo

The `MTControllerGetControllerInfo` function returns information about the current status of a stream controller and its associated channel.

```
pascal ComponentResult MTControllerGetControllerInfo
            (MTControllerComponent mtc,
            MTControllerInfoType *someFlags);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`someFlags` A pointer to a set of flags that indicate the current status and capabilities of the controller. The controller component sets the flags. The following flags are defined (more than one flag may be set to 1):

`mtControllerInfoCopyAvailable`
If this flag is set to 1, the controller's channel currently contains visual data—that is, an image is available for capture.

`mtControllerInfoHasSound`
If this flag is set to 1, the controller can play a channel's sound.

`mtControllerInfoIsPlaying`
If this flag is set to 1, the controller is currently playing the channel.

`mtControllerInfoIsRecording`
If this flag is set to 1, the controller is currently recording the channel.

DESCRIPTION

You call the `MTControllerGetControllerInfo` function to obtain information about a controller's capabilities and to determine whether the stream controller is currently playing and recording its channel. You can use this information to control your application's menu highlighting.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can capture a channel's image by calling the `MTControllerSnapshot` function, described next.

Capturing a Channel Image

This section describes the function that your application can use to capture a current image of a channel.

MTControllerSnapshot

The `MTControllerSnapshot` function provides you with a handle to a QuickDraw picture containing the current image in a stream controller's channel.

```
pascal ComponentResult MTControllerSnapshot
            (MTControllerComponent mtc, PicHandle *picture);
```

`mtc`	The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`picture`	A pointer to a picture handle.

DESCRIPTION

You call the `MTControllerSnapshot` function to capture the most recent image from the channel assigned to the stream controller that you specify. The image may contain visual data from several streams.

The function provides you with a handle to a QuickDraw picture of this image. If there is no image available or if an error occurs while capturing the image, the function sets the picture handle to `nil`.

RESULT CODES

`noErr`	0	No error

Handling Events

The functions described in this section allow you to

- pass all events to a stream controller component for inspection. A stream controller component then handles all events that apply to its controller or channel.
- direct a stream controller component to perform a specific action
- set an action filter function. A stream controller component informs the action filter function about actions it is about to take. You then can perform application-specific processing or refer the action back to the stream controller component to be handled.
- inform a stream controller component that you are about to change or have changed the characteristics of a channel's data

MTControllerIsControllerEvent

The `MTControllerIsControllerEvent` function passes an event to a stream controller component.

```
pascal ComponentResult MTControllerIsControllerEvent
            (MTControllerComponent mtc, const EventRecord *e);
```

`mtc`	The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`e`	A pointer to the current event structure.

DESCRIPTION

You call the `MTControllerIsControllerEvent` function to pass an event to a stream controller component. Your application should call `MTControllerIsControllerEvent` in its main event loop. If the controller component handles the event, the function returns the value 1. Otherwise, it returns 0.

If the controller component handled the event, your application should skip the rest of its event loop and wait for the next event. Otherwise, it should continue calling `MTControllerIsControllerEvent` for each stream controller component used by your application until the event is handled. If no controller component handles the event, your application must handle the event as part of its normal event processing.

The `MTControllerIsControllerEvent` function frees your application from the need to look at every event. Because a stream controller component does everything necessary to support a stream controller and its channel, your application need only handle those events that are not related to the controller and its channel.

SPECIAL CONSIDERATIONS

If your application provides an action filter function, the stream controller component calls your filter function before it handles an event. This gives your application a chance to perform custom processing.

If your application uses a conference component to acquire controllers, you do not need to call this function. The conference component does it for you.

SEE ALSO

You set an action filter function with the `MTControllerSetActionFilter` function, described on page 4-51.

The function declaration for an action filter function is described on page 4-58.

For a discussion of alternative ways to handle controller events, see the section "Handling Stream Controller Events" beginning on page 4-12.

The conference component is described in the chapter "Conference Component" in this book.

MTControllerDoAction

The `MTControllerDoAction` function tells a stream controller component to perform an action that you specify.

```
pascal ComponentResult MTControllerDoAction
            (MTControllerComponent mtc,
            MTControllerActionType action, void *params);
```

`mtc`	The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`action`	The action code for the action you want the controller component to perform. The section"Action Codes" beginning on page 4-19 describes the actions supported by stream controller components.
`params`	A pointer to the parameter data for the action. See the action descriptions in the section "Action Codes" for information about the parameter data required for individual actions.

DESCRIPTION

Your application can use the `MTControllerDoAction` function to direct a stream controller component to perform a specified action. Say, for example, your application defines a menu item that stops the presentation of all channels. When the user selects this menu item, your application should call the `MTControllerDoAction` function with the `mtControllerActionPlay` action to tell each controller to stop playing the channel.

SPECIAL CONSIDERATIONS

Calling the `MTControllerDoAction` function with a given action often results in the stream controller component calling your action filter function with the same action. A stream controller component calls your action filter function prior to performing an action so that your application has the opportunity to perform custom processing.

RESULT CODES

`noErr`	0	No error

MTControllerSetActionFilter

The `MTControllerSetActionFilter` function establishes an action filter function for a stream controller component.

```
pascal ComponentResult MTControllerSetActionFilter
            (MTControllerComponent mtc,
            MTControllerActionFilterUPP actionFilter,
            long refCon);
```

`mtc`	The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`actionFilter`
A universal procedure pointer to your action filter function. If you want to remove your action filter function, set this parameter to `nil`.

`refCon` Reserved for your use. The stream controller component passes this reference constant to your action filter function each time it calls your function.

DESCRIPTION

Stream controller components allow your application to preempt stream controller actions. You define an action filter function in your application and assign it to a controller by calling the `MTControllerSetActionFilter` function.

A stream controller component calls your action filter function each time the component receives an action for its stream controller. Your filter function can either handle the action or to refer it back to the stream controller component. If you refer it back to the stream controller component, the component handles the action.

Your filter function can handle the action in any way appropriate to your application. Frequently, applications use the action filter function to monitor the actions of the controller (which can be triggered by a user, another component, or the controller component itself). For instance, you can use your filter function to detect when a user clicks the play button so that your application can enable appropriate menu selections. Or you could detect when a user resizes a controller so that your application can resize the window in which the controller is located.

RESULT CODES

`noErr`	0	No error

SEE ALSO

If you use any functions in your action filter function that modify the channel, be sure to call the `MTControllerChangedStreams` function, described next.

The function declaration for an action filter function is described on page 4-58.

See the section "Action Codes" beginning on page 4-19, for a description of the actions supported by stream controller components.

Universal procedure pointers are described in *Inside Macintosh: PowerPC System Software*.

MTControllerChangedStreams

The `MTControllerChangedStreams` function informs a stream controller component that characteristics of its channel are about to change or have changed.

```
pascal ComponentResult MTControllerChangedStreams
            (MTControllerComponent mtc, Boolean finished);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`finished` A Boolean value indicating whether the stream formats are about to change or have changed. Set this parameter to `false` to indicate that stream formats are about to change. Set it to `true` to indicate the stream formats have changed.

DESCRIPTION

You call the `MTControllerChangedStreams` function to inform a controller component that you are about to change or have changed the format of a stream in its channel.

Your application can make most stream format changes using the `MTControllerDoAction` function. In these cases, it is not necessary to call `MTControllerChangedStreams`.

However, if your application uses functions other than `MTControllerDoAction` to change the characteristics of a stream in a channel, you must inform the controller component so that it can update itself accordingly. For instance, if your application changes the size of the channel without informing the stream controller component, the controller may no longer be the proper size for the channel.

Prior to changing stream formats, call this function and set the `finished` parameter to `false`. Then, call whatever functions you need to change stream formats. After changing the stream formats, call `MTControllerChangedStreams` and set `finished` to `true` to tell the controller component that the formats have been changed.

This function can return Memory Manager errors.

SPECIAL CONSIDERATIONS

If your application works directly with a stream director, you typically need to call the `MTDirectorChangedStreamFormats` function to inform the stream director about stream format changes. However, if your application also uses a stream controller component, you don't need to call `MTDirectorChangedStreamFormats` because `MTControllerChangedStreams` does it for you.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTControllerDoAction` function is described on page 4-50.

The proper use of the `MTControllerChangedStreams` function is analogous to that of the `MTDirectorChangedStreamFormats` function. See the chapter "Stream Director Components" in this book for a discussion of changing stream formats.

Customizing Event Processing

If your application does not use the `MTControllerIsControllerEvent` function (described on page 4-49) to pass all events to a stream controller component for inspection and possible handling, you can use the functions described in this section to pass specific events to a stream controller component. The component then attempts to handle the event.

Your application obtains the values for many of the function parameters from the original event structure.

MTControllerActivate

The `MTControllerActivate` function tells a stream controller component to activate or deactivate its controller.

```
pascal ComponentResult MTControllerActivate
            (MTControllerComponent mtc, WindowPtr w,
            Boolean activate);
```

`mtc`	The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`w`	A pointer to the window in which the event occurred.
`activate`	A Boolean value that indicates the nature of the event. Set this parameter to `true` for activate and resume events. Set it to `false` for deactivate and suspend events.

DESCRIPTION

Your application can call the `MTControllerActivate` function in response to activate, deactivate, suspend, and resume events. If the controller is in the window to which the event applies, the stream controller component activates or deactivates the controller, depending on the setting of the `activate` parameter.

Activating or deactivating a controller causes its onscreen appearance to change. It also changes how the controller component responds to other events. For instance, when a controller is deactivated, the controller component ignores keystrokes.

The `MTControllerActivate` function returns the value 1 if the controller component handles the event. If it does not handle the event, the function returns 0. In this case, your application is responsible for handling the event.

If your application uses the `MTControllerIsControllerEvent` function that lets a controller component handle events that pertain to the controller and its channel, you don't need to call the `MTControllerActivate` function.

RESULT CODES

`noErr`	0	No error

MTControllerClick

The `MTControllerClick` function tells a controller component that a user clicked in a stream controller or its channel.

```
pascal ComponentResult MTControllerClick
            (MTControllerComponent mtc,
            WindowPtr w, Point where, long when, long modifiers);
```

`mtc`	The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`w`	A pointer to the window in which the click occurred.
`where`	The location of the click, expressed in the local coordinates of the window specified by the `w` parameter. Your application must convert the global coordinates returned in the event structure to local coordinates before passing this information to the function.
`when`	The time at which the user pressed the mouse button. You obtain this value from the event structure.
`modifiers`	Modifier flags for the event. You obtain this value from the event structure.

DESCRIPTION

Your application can call the `MTControllerClick` function when the user clicks in a controller or in a channel. Whether the controller and channel are attached or detached is irrelevant.

The `MTControllerClick` function returns the value 1 if the controller component handled the event. It returns 0 if it did not handle the event. In this case, your application is responsible for handling the event.

If your application uses the `MTControllerIsControllerEvent` function that lets a controller component handle events that pertain to the controller and its channel, you don't need to call the `MTControllerClick` function.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `EventRecord` structure is described in *Inside Macintosh: Macintosh Toolbox Essentials*.

MTControllerDraw

The `MTControllerDraw` function tells a stream controller component to draw its controller.

```
pascal ComponentResult MTControllerDraw
            (MTControllerComponent mtc, WindowPtr w);
```

`mtc` The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`w` A pointer to the window for which an update event occurred.

DESCRIPTION

You call the `MTControllerDraw` function in response to an update event. If the controller is in the window to which the update event applies, the stream controller component redraws the stream controller.

If the controller's channel is located in the window that received the event, the controller component also updates the channel.

The `MTControllerDraw` function returns the value 1 if the controller component handles the event. If it does not handle the event, the function returns 0. In this case, your application is responsible for handling the event.

If your application uses the `MTControllerIsControllerEvent` function that lets a controller component handle events that pertain to the controller and its channel, then you don't need to call the `MTControllerDraw` function.

RESULT CODES

`noErr`	0	No error

MTControllerKey

The `MTControllerKey` function tells a controller component that a user pressed a key.

```
pascal ComponentResult MTControllerKey (MTControllerComponent mtc,
         char key, long modifiers);
```

`mtc`	The stream controller component you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`key`	The virtual key code for the key that was pressed. You obtain this value from the event structure.
`modifiers`	Modifier flags for the event. You obtain this value from the event structure.

DESCRIPTION

You call the `MTControllerKey` function in response to a keyboard event only if you have enabled keystroke processing in the controller. By default, keystroke processing is disabled when you create a new stream controller.

The `MTControllerKey` function returns the value 1 if the controller component handles the event. If it does not handle the event, the function returns 0. In this case, your application is responsible for handling the event.

If your application uses the `MTControllerIsControllerEvent` function that lets a controller component handle events that pertain to the controller and its channel, then you don't need to call the `MTControllerKey` function.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can enable and disable keystroke processing by calling the `MTControllerDoAction` function with the `mtControllerActionSetKeysEnabled` action. The function is described on page 4-50. The action is described in the section "Action Codes" beginning on page 4-19.

Virtual key codes and event structures are described in *Inside Macintosh: Macintosh Toolbox Essentials*.

Application-Defined Function

This section describes the action filter function that you provide to a stream controller component. A stream controller component calls your filter function to keep you informed of actions it is about to take and to provide you with an opportunity to override its action with custom processing.

MyControllerActionFilter

When you call the `MTControllerSetActionFilter` function, you provide a pointer to an action filter function. A stream controller component calls your filter function to inform your application that it is about to take a specific action. Here's how you would declare an action filter function if it were a C function named `MyControllerActionFilter`:

```
Boolean MyControllerActionFilter (MTControllerComponent mtc,
            MTControllerActionType action,
            void *params, long refCon);
```

`mtc` The component instance of the stream controller component that calls this filter function.

`action` The action code. See the section "Action Codes" beginning on page 4-19 for a list of action codes and a description of the actions supported by stream controller components.

`params` A pointer to the parameter data for the action. See the descriptions of the actions in the section "Action Codes" for information about the parameter data for individual actions.

`refCon` A reference constant value. A stream controller component obtains this reference constant from the `MTControllerSetActionFilter` function and passes it to your action filter function.

DESCRIPTION

Your action filter function must return a Boolean value indicating whether it handled the action. Return the Boolean value `true` if your function completely handles the action. In this case, a stream controller component performs no additional processing for the action. Return `false` if your function does not handle the action. The stream controller component then handles the action.

If your action filter function calls any function that modifies the format of the stream data in the channel, be sure you call the `MTControllerChangedStreams` function (described on page 4-53) to inform the controller component.

SEE ALSO

You establish an action filter function with a stream controller component by calling the `MTControllerSetActionFilter` function, described on page 4-51.

Summary of Stream Controller Components

C Summary

Constants

```
enum { /* stream controller component type */
   kMTControllerType    = 'mtcn'
};

enum { /* stream controller component subtype */
   kMTMovieTalkSubType  = 'mtlk'
};

enum { /* stream controller component request codes */
   kMTControllerSetStreamDirectorSelect         = 1,
   kMTControllerGetStreamDirectorSelect         = 2,
   kMTControllerRemoveStreamDirectorSelect      = 3,
   kMTControllerIsControllerEventSelect         = 4,
   kMTControllerDoActionSelect                  = 5,
   kMTControllerSetControllerAttachedSelect     = 6,
   kMTControllerIsControllerAttachedSelect      = 7,
   kMTControllerSetControllerPortSelect         = 8,
   kMTControllerGetControllerPortSelect         = 9,
   kMTControllerSetVisibleSelect                = 10,
   kMTControllerGetVisibleSelect                = 11,
   kMTControllerSetControllerBoundsRectSelect   = 12,
   kMTControllerGetControllerBoundsRectSelect   = 13,
   kMTControllerGetControllerBoundsRgnSelect    = 14,
   kMTControllerGetWindowRgnSelect              = 15,
   kMTControllerChangedStreamsSelect            = 16,
   kMTControllerNewAttachedControllerSelect     = 17,
   kMTControllerDrawSelect                      = 18,
   kMTControllerActivateSelect                  = 19,
   kMTControllerKeySelect                       = 20,
```

```
   kMTControllerClickSelect                    = 21,
   kMTControllerSnapshotSelect                 = 22,
   kMTControllerPositionControllerSelect       = 23,
   kMTControllerGetControllerInfoSelect        = 24,
   kMTControllerSetClipSelect                  = 25,
   kMTControllerGetClipSelect                  = 26,
   kMTControllerSetActionFilterSelect          = 27
};

enum { /* stream controller component action codes */
   mtControllerActionDraw                    = 2,
   mtControllerActionActivate                = 3,
   mtControllerActionDeactivate              = 4,
   mtControllerActionMouseDown               = 5,
   mtControllerActionKey                     = 6,
   mtControllerActionPlay                    = 8,
   mtControllerActionSetVolume               = 14,
   mtControllerActionGetVolume               = 15,
   mtControllerActionSetGain                 = 16,
   mtControllerActionGetGain                 = 17,
   mtControllerActionSetGrowBoxBounds        = 25,
   mtControllerActionControllerSizeChanged   = 26,
   mtControllerActionSetKeysEnabled          = 32,
   mtControllerActionGetKeysEnabled          = 33,
   mtControllerActionSetFlags                = 38,
   mtControllerActionGetFlags                = 39,
   mtControllerActionShowBalloon             = 43,
   mtControllerActionClick                   = 45,
   mtControllerActionSuspend                 = 46,
   mtControllerActionResume                  = 47,
   mtControllerActionGetDragEnabled          = 51,
   mtControllerActionSetDragEnabled          = 52,
   mtControllerActionStartRecord             = 60,
   mtControllerActionStreamsChanged          = 61,
   mtControllerActionSnapshot                = 62,
   mtControllerActionSnapshotData            = 63,
   mtControllerActionSetEnableSnapshot       = 70,
   mtControllerActionSetEnableSound          = 71,
   mtControllerActionSetEnablePlay           = 72,
   mtControllerActionSetEnableRecord         = 73,
   mtControllerActionSetEnableGain           = 74,
   mtControllerActionGetEnableSnapshot       = 75,
   mtControllerActionGetEnableSound          = 76,
   mtControllerActionGetEnablePlay           = 77,
```

```
   mtControllerActionGetEnableRecord         = 78,
   mtControllerActionGetEnableGain           = 79,
   mtControllerActionSetShowSnapshot         = 80,
   mtControllerActionSetShowSound            = 81,
   mtControllerActionSetShowPlay             = 82,
   mtControllerActionSetShowRecord           = 83,
   mtControllerActionSetShowGain             = 84,
   mtControllerActionGetShowSnapshot         = 85,
   mtControllerActionGetShowSound            = 86,
   mtControllerActionGetShowPlay             = 87,
   mtControllerActionGetShowRecord           = 88,
   mtControllerActionGetShowGain             = 89
};

enum {      /* flags for MTControllerPositionController function */
   mtControllerTopLeftChannel          = 1 << 0,
   mtControllerScaleChannelToFit       = 1 << 1,
   mtControllerNotVisible              = 1 << 3,
   mtControllerWithFrame               = 1 << 4,
   mtControllerPositionDontInvalidate  = 1 << 5
};

enum {      /* flags for mtControllerActionSetFlags action */
   mtControllerFlagSuppressChannelFrame   = 1 << 0,
   mtControllerFlagSuppressSpeakerButton  = 1 << 2,
   mtControllerFlagsUseWindowPalette      = 1 << 3,
   mtControllerFlagsDontInvalidate        = 1 << 4
};

enum {      /* values of MTControllerInfoType */
   mtControllerInfoCopyAvailable = 1 << 2,
   mtControllerInfoWithFrame     = 1 << 4,
   mtControllerInfoHasSound      = 1 << 5,
   mtControllerInfoIsPlaying     = 1 << 6,
   mtControllerInfoIsRecording   = 1 << 7
};
```

Data Types

```
typedef short MTControllerActionType;

typedef long MTControllerInfoType;

typedef ComponentInstance MTControllerComponent;
```

Functions

Assigning Channels to Stream Controllers

```
pascal ComponentResult MTControllerNewAttachedController
                            (MTControllerComponent mtc,
                             ComponentInstance sd,
                             WindowPtr w, Point where);
pascal ComponentResult MTControllerSetStreamDirector
                            (MTControllerComponent mtc,
                             ComponentInstance sd,
                             WindowPtr displayWindow, Point where);
pascal ComponentInstance MTControllerGetStreamDirector
                            (MTControllerComponent mtc);
pascal ComponentResult MTControllerRemoveStreamDirector
                            (MTControllerComponent mtc);
```

Getting and Setting Stream Controller Characteristics

```
pascal ComponentResult MTControllerPositionController
                            (MTControllerComponent mtc,
                             const Rect *imageRect,
                             const Rect *controllerRect, long someFlags);
pascal ComponentResult MTControllerSetControllerAttached
                            (MTControllerComponent mtc, Boolean attach);
pascal ComponentResult MTControllerIsControllerAttached
                            (MTControllerComponent mtc);
pascal ComponentResult MTControllerSetVisible
                            (MTControllerComponent mtc, Boolean visible);
pascal ComponentResult MTControllerGetVisible
                            (MTControllerComponent mtc);
pascal ComponentResult MTControllerSetControllerBoundsRect
                            (MTControllerComponent mtc, const Rect *bounds);
pascal ComponentResult MTControllerGetControllerBoundsRect
                            (MTControllerComponent mtc, Rect *bounds);
pascal RgnHandle MTControllerGetControllerBoundsRgn
                            (MTControllerComponent mtc);
pascal RgnHandle MTControllerGetWindowRgn
                            (MTControllerComponent mtc, WindowPtr w);
pascal ComponentResult MTControllerSetClip
                            (MTControllerComponent mtc, RgnHandle theClip,
                             RgnHandle imageClip);
```

```
pascal ComponentResult MTControllerGetClip
                            (MTControllerComponent mtc, RgnHandle *theClip,
                             RgnHandle *imageClip);
pascal ComponentResult MTControllerSetControllerPort
                            (MTControllerComponent mtc, CGrafPtr gp);
pascal CGrafPtr MTControllerGetControllerPort
                            (MTControllerComponent mtc);
pascal ComponentResult MTControllerGetControllerInfo
                            (MTControllerComponent mtc,
                             MTControllerInfoType *someFlags);
```

Capturing a Channel Image

```
pascal ComponentResult MTControllerSnapshot
                            (MTControllerComponent mtc, PicHandle *picture);
```

Handling Events

```
pascal ComponentResult MTControllerIsControllerEvent
                            (MTControllerComponent mtc,
                             const EventRecord *e);
pascal ComponentResult MTControllerDoAction
                            (MTControllerComponent mtc,
                             MTControllerActionType action, void *params);
pascal ComponentResult MTControllerSetActionFilter
                            (MTControllerComponent mtc,
                             MTControllerActionFilterUPP actionFilter,
                             long refCon);
pascal ComponentResult MTControllerChangedStreams
                            (MTControllerComponent mtc);
```

Customizing Event Processing

```
pascal ComponentResult MTControllerActivate
                            (MTControllerComponent mtc, WindowPtr w,
                             Boolean activate);
pascal ComponentResult MTControllerClick
                            (MTControllerComponent mtc, WindowPtr w,
                             Point where, long when, long modifiers);
pascal ComponentResult MTControllerDraw
                            (MTControllerComponent mtc, WindowPtr w);
pascal ComponentResult MTControllerKey
                            (MTControllerComponent mtc, char key,
                             long modifiers);
```

Application-Defined Function

```
pascal Boolean MyControllerActionFilter
                            (MTControllerComponent mtc,
                             MTControllerActionType action,
                             void *params, long refCon);
```

Result Codes

noErr	0	No error

Stream Director Components

Contents

Stream Director Components

This chapter describes stream director components and how you can use the services they provide. A **stream director component** manages a group of incoming or outgoing streams of media data in a QuickTime Conferencing connection. The stream director component API provides a standard interface for managing streams of time-based data, independent of differences in the underlying data.

If your application calls stream director component functions, you should read this chapter. However, most applications do not call a stream director component directly. Instead, they use a conference component and stream controller components that manage stream director components on their behalf. If your application requires different features or greater control of stream director services than that available through the use of a conference component and a stream controller component, you need to read this chapter.

The information in this chapter is presented from the perspective of one who uses a stream director component, typically a conference component or a stream controller component.

To use a stream director component, you need to be familiar with the API described in this chapter as well as these elements of Macintosh system software:

- The Component Manager, described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*, provides functions that operate on all components.
- Some stream director components use QuickTime sequence grabber components to generate media data. If your application uses such a stream director, you need to know about sequence grabber components. They are discussed in *Inside Macintosh: QuickTime Components*.
- To work with streams that have visual characteristics, you should be familiar with QuickDraw concepts (see *Inside Macintosh: Imaging With QuickDraw*).
- To work with sound streams, you should know about the Sound Manager (see *Inside Macintosh: Sound*).

In addition to the above, if you want to create your own stream director component, you need to be familiar with the following types of QuickTime Conferencing components with which a stream director interacts:

- stream controller components
- stream player components
- flow control components
- recorder components
- transport components

These components are each discussed in their own chapters in this book.

This chapter begins with an introduction to streams and their constituent parts, followed by an introduction to stream director components. Then it discusses how you can

- open and close a stream director component
- set up and maintain the working environment for a stream director component

- use a stream director component to manage the flow of media data between other components
- get and set information about streams
- get and set characteristics of visual stream data
- get and set characteristics of audio stream data

Introduction to Streams

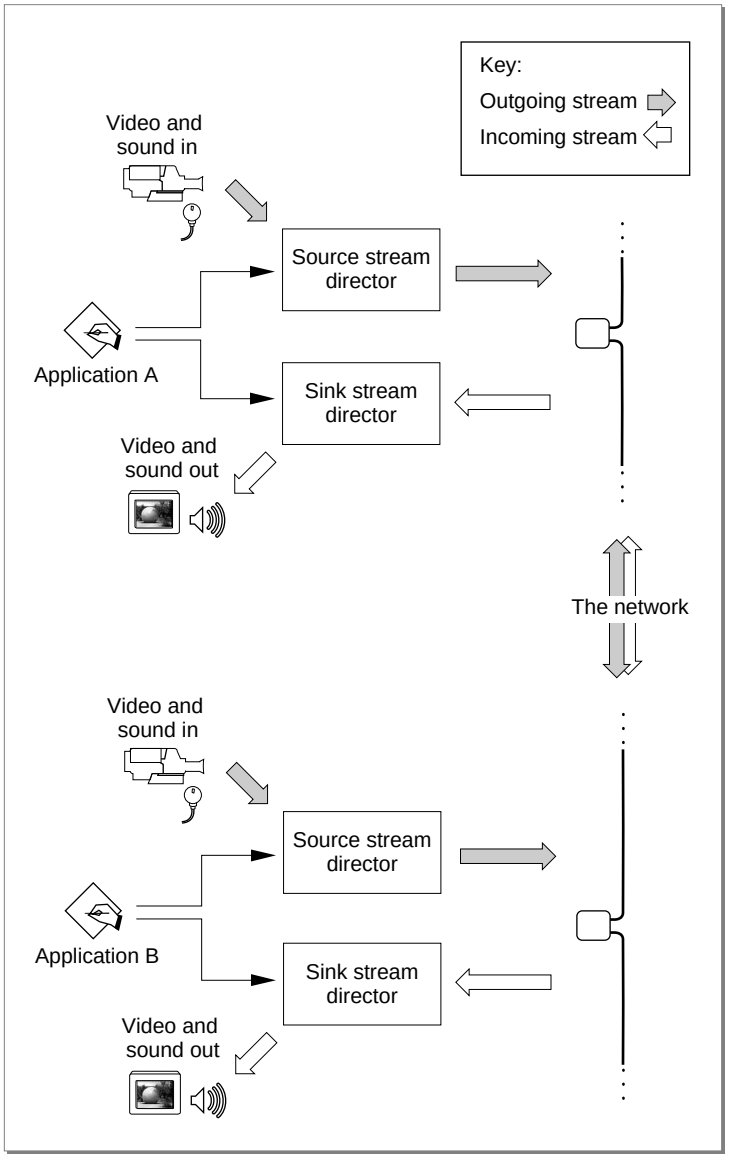
A **stream** is a logical sequence of **media data**—time-based information, such as video or sound data—that flows between QuickTime Conferencing connection ends. Each stream is composed of a single type of media data. The media data in a stream is organized as a sequence of chunks, with each chunk represented by a chunk record structure.

Streams are created and disposed of by a stream director during the course of a connection. A stream originates on the source side and flows to the sink side in the media channel of a QuickTime Conferencing connection. See the chapter "Transport Components" in this book for more information about QuickTime Conferencing connections and media channels.

A stream whose data is generated locally and sent out over a network to a remote connection end is referred to as an **outgoing stream**, whereas one whose data is received and played locally is referred to as an **incoming stream**.

Figure 5-1 shows the relationship between the incoming and outgoing streams within and between cooperating video conferencing applications exchanging media data. (Source and sink stream directors are defined in the next section, "About Stream Director Components.")

Figure 5-1 The relationship between incoming and outgoing streams



Each stream, when it is created, is assigned a unique stream ID. A stream ID is not a persistent identifier. It changes when the stream's format is renegotiated. Negotiation is discussed later in this chapter.

Chunk record structures are the constituent parts of streams. They contain chunks of media data as well as descriptive information about the chunk. If you are using the services of a stream director, you do not need to know about chunk record structures. However, if you are developing a stream director, you need to understand the contents of a chunk record structure. Relevant fields of the chunk record structure are described in

the section "Handling Media Data" beginning on page 5-32. The complete chunk record structure is described in the chapter "Transport Components" in this book.

A **chunk** is a logical unit of media data, such as a video frame or a sound sample. For example, a chunk of sound data may contain a collection of sampled sound, definitions of the characteristics of the sampled sound, and other relevant details about the sound. QuickTime Conferencing does not define the format or size of chunks because they vary with the type of media data.

If you are familiar with QuickTime concepts, you'll notice that a stream is somewhat analogous to a QuickTime track. Both streams and tracks consist of a set of time-ordered media data of a single type. And both contain descriptions of the format of that media data. Furthermore, streams can be recorded into tracks (using a recorder component), and tracks can be carried in a stream (if a source stream director taps a stored source of media data).

However, a stream differs from a QuickTime track in these significant ways:

- A stream travels over a network and is therefore subject to possible loss of data. A track comes from a storage medium not subject to data loss.
- A stream can carry live data—that is, the media data is generated in real time. The data in a track has been generated at a previous time and exists as stored data—in a file or in memory.
- A stream's data can be interpreted using stream description information and chunk record structures. To be interpreted, a track typically requires a QuickTime movie resource. Thus, a stream's media data is more self-contained.

A Note on Terminology

In this chapter, a stream containing data that can be displayed or played back is described as containing visual or audio data. The terms *video* and *sound* refer to the specific media types defined by QuickTime (`VideoMediaType` and `SoundMediaType`) and to the streams that carry video or sound media data. For example, MIDI and sound streams both contain audio data. These streams are generally described as *containing audio data*. They are specifically described as MIDI streams and sound streams—that is, they contain audio data in a specific format.

For the sake of brevity, stream director components are usually referred to as stream directors in this chapter.

In this chapter, the terms *media data* and *stream data* are used interchangeably. QuickTime Conferencing uses the term *media data* to refer to time-based information, such as video or sound. If you are familiar with QuickTime, you'll know that QuickTime uses the term *media* to refer to a data structure that describes the data in a movie track, including a reference to the actual data, which is stored on some device. ◆

About Stream Director Components

Media data flows through several components as it travels from the point of generation to the point of display. A stream director plays a key role in tying these various components together. This section provides introductory information about stream directors. More detailed information can be found in the section "Using a Stream Director Component" beginning on page 5-13.

A stream director component provides three main services:

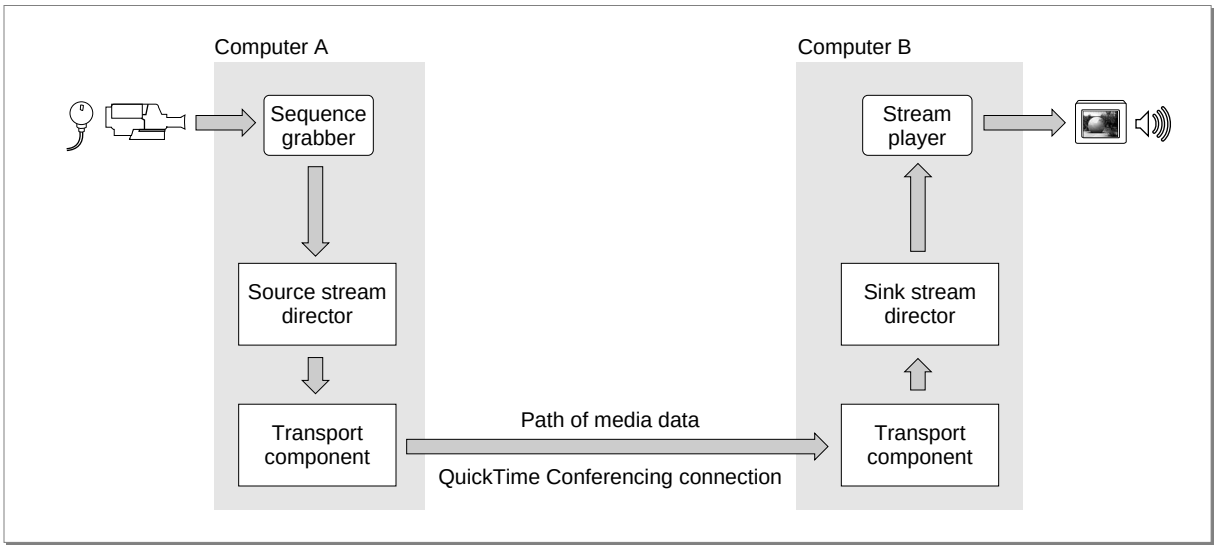
- It routes media data between other components. It also allows you to stop and start the flow of data in the streams that it manages and provides information about those streams.
- It negotiates stream formats to check that media data sent over a QuickTime Conferencing connection can be played at the remote connection end.
- It manages the playing of media data.

Because a single stream director manages either outgoing streams or incoming streams, but not both, two types of stream director components are defined. A **source stream director** manages one or more outgoing streams. A **sink stream director** manages one or more incoming streams.

A stream director uses other components to provide its services. A source stream director acquires media data from some source and uses a transport component to transmit the media data across a connection to a remote sink stream director. For example, the Apple-provided source stream director uses a QuickTime sequence grabber to capture real-time media data. A sink stream director receives media data from a remote source stream director via a transport component and is responsible for its disposition. Typically, it manages the display of the media data. For example, the Apple-provided sink stream director uses a stream player component for real-time display of media data it receives.

Figure 5-2 gives a simplified view of the path traversed by media data in a QuickTime Conferencing connection.

Figure 5-2 Path of media data in a QuickTime Conferencing connection



A sequence grabber component in computer A generates live audio-video data and forwards it to a source stream director. The source stream director passes the data to a transport component for transmission to the remote end of the connection. The transport component in computer B receives the incoming data and forwards it to a sink stream director. The sink stream director passes it to a stream player component that plays the data.

Applications that make full-duplex QuickTime Conferencing connections need both a source and a sink stream director. Applications that make half-duplex connections, such as for sending or for watching broadcast media, need either a source stream director or a sink stream director, but not both.

One stream director instance manages all of the outgoing or incoming streams carried on one connection. Typically, a connection carries multiple streams. For example, it is common for a connection to carry both video and sound data.

Note
In the chapter "Stream Controller Components" in this book, the term *channel* is used to refer to the group of streams managed by a given stream director. Once a stream controller component is associated with a specific stream director, the streams managed by that stream director become the controller component's channel, to be presented by the controller component to a user. The term *channel* is not used in this chapter to refer to a group of streams. ◆

Component Types and Subtypes

All source stream directors have the component type `'srsd'` (defined by the constant `kMTSourceStreamDirectorType`). All sink stream directors have the component type `'sksd'` (defined by the constant `kMTSinkStreamDirectorType`). Source and sink stream directors share the API described in this chapter.

The component subtype of a source stream director must match the component type of the component it uses to acquire media data. For example, Apple defines the component subtype `kMTGrabberSubType` for source stream directors that use QuickTime sequence grabbers as sources of media data.

The component subtype of a sink stream director must match the component type of the component it uses to play or otherwise process media data. Apple defines the component subtype `kMTPlayerSubType` for sink stream directors that use stream player components to play media data.

The sections that follow briefly describe a stream director's main services. You can find more specific information on these services in the section "Using a Stream Director Component" beginning on page 5-13.

Managing Streams

A stream director interacts with other components to acquire media data, to move it from the point of generation to the point of play, and to cause it to be displayed or played back. A typical sequence of source stream director actions consists of the following:

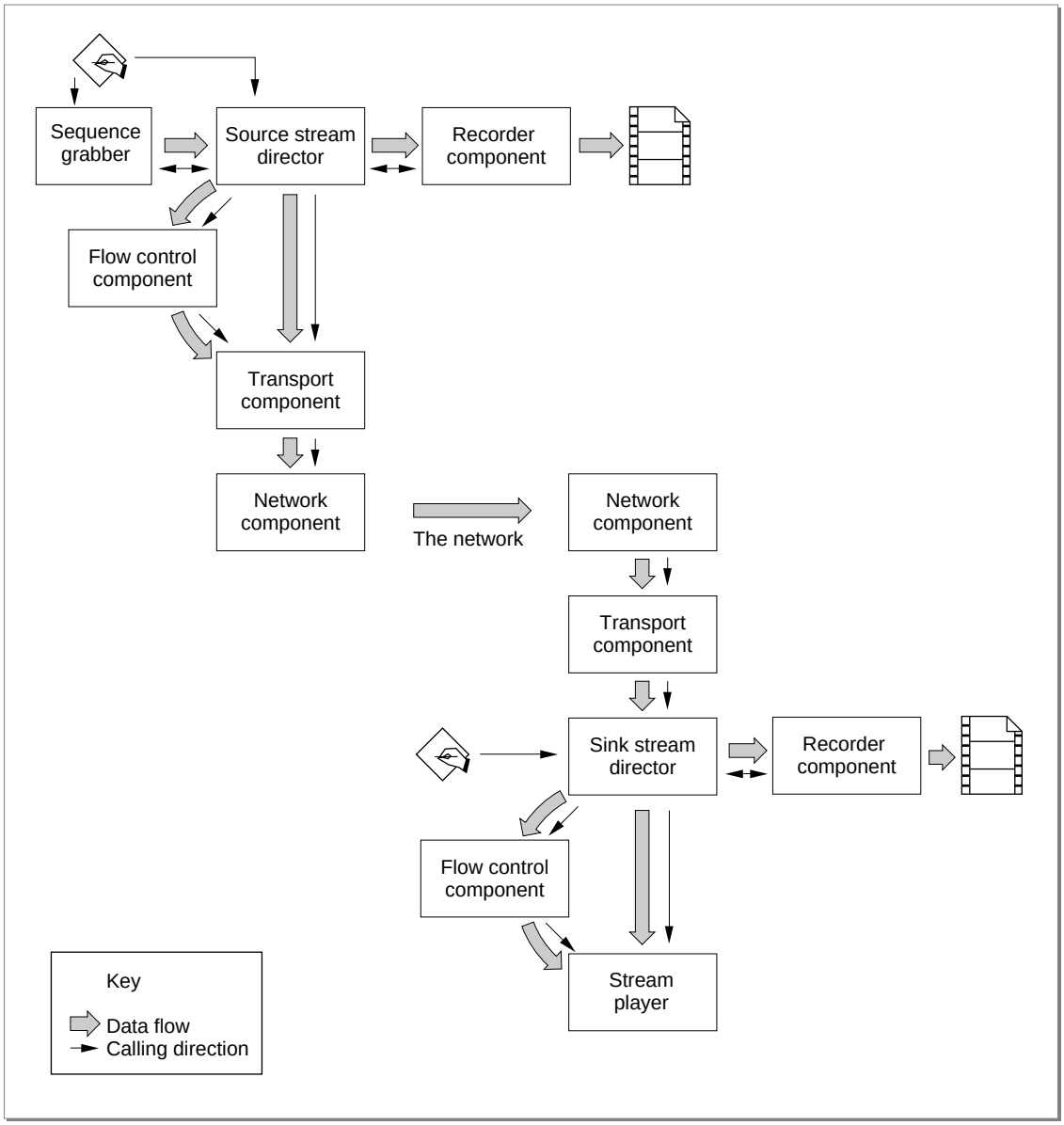
- A source stream director installs a media data callback function on a media component. When the media component generates media data, it calls the function to pass the data to the source stream director.
- If a recording function has been installed, the source stream director calls the recording function and passes it the media data to be recorded.
- The source stream director forwards the data to a transport component.

A typical sequence of sink stream director actions consists of the following:

- A sink stream director installs a media data callback function on a transport component. When the transport component receives incoming media data, it calls the function to deliver the data to the sink stream director.
- If a recording function has been installed, the sink stream director calls the function and passes it the media data to be recorded.
- The sink stream director forwards the data to a stream player component capable of playing the type of media data it received.

Figure 5-3 illustrates the standard relationships between a QuickTime Conferencing application, a stream director, and other components. The figure shows both the calling relationships and the path of media data as it flows between components. (An application that uses a conference component and a stream controller component does not need to know about these relationships.)

Figure 5-3 Relationships between an application, a stream director, and other components



A stream director allows your application to start and stop the flow of media data in streams. This is referred to as *enabling* or *disabling a stream*. An enabled stream is one whose data is processed—accepted, forwarded, recorded, and so forth. If you disable a stream, a stream director ignores all data for that stream. This is useful in many situations. For example, your application might want to disable a sound stream when the user turns the volume all the way off and to enable the stream when the user turns the volume back up.

You can get information about certain characteristics of a stream, its stream description, and the number of streams managed by a stream director through functions provided in the API.

Negotiating Stream Formats

Stream director components negotiate stream formats. A source stream director at one end of a QuickTime Conferencing connection and a sink stream director at the other end exchange information about the number of streams the source wants to transmit and the format of the media data for each stream. The process checks that the data sent from the source side can be played on the sink side. Negotiations occur as soon as a source stream director is informed that a connection exists, and whenever stream formats change during a connection.

A stream director typically performs the following sequence of actions:

- negotiate stream formats with the remote connection end
- send or receive media data
- renegotiate stream formats during the connection when source stream media data format changes

Managing the Playing of Media Data

A stream director manages the playing of media data. A source stream director controls the playing of local media data (such as the source view in a video conferencing application). A sink stream director typically manages stream player components that play media data received from a remote connection end.

For streams that contain visual data, the stream director API provides functions that allow you to modify a variety of visual display characteristics, such as the graphics world, the source rectangle, the transformation matrix, and the clipping region. For streams that contain audio data, it allows you to modify audio characteristics such as the volume and the audio input gain.

A stream director also provides functions you can use to inform it of changes in the display or application environment, such as the application going from the foreground to the background. You should call these functions when changes occur to assist a stream director in maintaining appropriate display characteristics and optimal use of system resources.

Using a Stream Director Component

The stream director API provides functions that allow applications and other components to control streams in QuickTime Conferencing applications. This section discusses some of the more common operations you can perform. In particular, it shows how you can

- open and close a stream director
- attach other components to a stream director
- monitor stream format negotiations
- keep a stream director informed of changing stream formats
- manage streams
- control streams that contain visual data
- control streams that contain audio data
- keep a stream director informed of changes in the display or application environment resulting from user events

Opening and Closing a Stream Director

You need to open a stream director component before you can use its services.

Listing 5-1 on page 5-15 shows how you open a source stream director of a given subtype when you do not want to specify any other characteristics of the component. You call the Component Manager function `OpenDefaultComponent`, setting the first parameter to the constant `kMTSourceStreamDirectorType` to specify a source stream director. You set the second parameter to the stream director subtype that you want to open. For example, if you want to open a stream director that uses a QuickTime sequence grabber as its source of media data, set the subtype to the constant `kMTGrabberSubType`. The function returns a stream director component instance.

To open a sink stream director, you set the first parameter to the constant `kMTSinkStreamDirectorType` and the second to the appropriate subtype.

If you want to open a specific stream director component, call the Component Manager `OpenComponent` function.

When you have finished using a stream director, close it by calling the `CloseComponent` function. The `OpenDefaultComponent`, `OpenComponent`, and `CloseComponent` functions are described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Attaching Components to a Stream Director

A stream director interacts with other components to provide its services. For example, it uses a transport component that sets up and maintains a QuickTime Conferencing connection. You are responsible for some of these other components—your application opens them, configures them, and closes them. The stream director API provides functions that allow you to inform a stream director about the existence of these other components. This is referred to as *attaching* a component to a stream director.

A stream director itself is responsible for other components—opening, configuring, and closing them. This is referred to as *owning* a component. You have to know about owned components only if you are developing a stream director. This topic is discussed in the section "Setting Up Other Components" beginning on page 5-34.

You attach media and transport components to a stream director. A **media component** generates real-time media data for a source stream director. The source stream director provided by QuickTime Conferencing 1.0 uses a QuickTime sequence grabber as a source of media data.

You attach a media component to a source stream director by calling the `MTDirectorSetMediaComponent` function (page 5-47). Listing 5-1 on page 5-15 shows how you call this function. You provide the instances of the stream director component you are using and the media component instance to attach.

You don't attach a media component to a sink stream director because sink stream directors deal only with media data received from a remote connection end. They have nothing to do with media data generated locally.

You can find out the component instance of a source stream director's attached media component by calling the `MTDirectorGetMediaComponent` function (page 5-48), like this:

```
myErr = MTDirectorGetMediaComponent
            (gStreamDirectorSourceInstance,
            &gMediaComponentInstance, &mediaComponentType,
            &mediaComponentSubType);
```

In addition to returning the media component instance, the function also returns the component type and subtype of the media component.

A stream director sends media data to or receives media data from a specific transport component. After you open a transport component, you need to attach it to a stream director by calling the `MTDirectorSetTransport` function (page 5-45). Listing 5-1 shows how you call this function. You set the `gStreamDirectorSourceInstance` parameter to the instance of the stream director component you are using and the `gTransportInstance` parameter to the transport component instance to attach.

Listing 5-1 Opening and setting up a source stream director component

```
ComponentResult OpenSourceStreamDirector()
{
   ComponentResult myErr = noErr;


   /* open source stream director component */

   gStreamDirectorSourceInstance =
         OpenDefaultComponent(kMTSourceStreamDirectorType,
                              kSequenceGrabberType);

   /* attach a sequence grabber component to the stream
      director component, so that the stream director knows
```

```
      where to get the media data */

   myErr = MTDirectorSetMediaComponent
                  (gStreamDirectorSourceInstance,
                  gMediaComponentInstance);

   /* attach a transport component to the stream director
      component, so that the stream director knows where to
      send media data */

   myErr = MTDirectorSetTransport(gStreamDirectorSourceInstance,
                                  gTransportInstance);

   /* install the negotiate function */
   myErr = MTDirectorSetNegotiateProc
               (gStreamDirectorSourceInstance,
               gSourceNegotiateProc, gMyWorld);

   return myErr;
}
```

You can find out the component instance of a stream director's attached transport component by calling the `MTDirectorGetTransport` function (page 5-46), as shown here:

```
myErr = MTDirectorGetTransport (directorInstance,
                                  &transportInstance);
```

If your application is using both a source and a sink stream director and both stream directors are using the same transport-level connection, you need to tell them about each other. You do this by calling the `MTDirectorJoin` function (page 5-54). Typically, you call `MTDirectorJoin` after you have established a connection and before you call the `MTDirectorConnected` function (page 5-53) to inform the source stream director about the connection.

A source and a sink stream director usually share a transport component when you make a full-duplex audio-visual connection. By joining them, you give each stream director access to the other's information about the connection. A stream director needs this information for certain types of operations in full-duplex audio-visual connections, such as managing the sound level for the connection when using certain echo-reduction techniques.

Stream director components are automatically separated when you close one of them.

Monitoring Stream Format Negotiations

The information in this section is presented from the perspective of an application. However, if your application uses a stream controller component, the stream controller component monitors negotiations for you. In that case, you do not need to be familiar with the material presented here. Furthermore, if your application uses a conference component, you can use the `mtNegotiationMessageCapability` option with the `MTConferenceSetMessageCapabilities` function to help ensure successful negotiations. See the chapter "Conference Component" in this book for a description of the `MTConferenceSetMessageCapabilities` function.

The material presented in this section will be of interest to you if your application does not use a stream controller or conference component or if it requires more detailed information about the progress of a negotiation process than that available through a stream controller or conference component. (Stream controller components are described in the chapter "Stream Controller Components" in this book.)

Stream format negotiation is the process by which a source stream director at one end of a connection and a sink stream director at the other end agree on the format of the streams that will flow between them. A source stream director tells a remote sink stream director what stream formats it wants to send. A sink stream director determines whether there are resources available to play the media data it is to receive. The negotiation process checks stream format compatibility between the connection ends.

Your application needs to know the outcome of a negotiation because you can send and receive media data only after stream formats are successfully negotiated. To stay informed, your application needs to provide a negotiation callback function. A stream director calls your function at certain times during a negotiation. This allows you to track the progress of the negotiation (and optionally to accept or reject proposed incoming streams).

You install your negotiation callback function by calling the `MTDirectorSetNegotiateProc` function (page 5-51), as shown in Listing 5-1 on page 5-15. The sample code installs a negotiation callback function for a source stream director. You set the parameter `gSourceNegotiateProc` to point to your negotiation function. You set the parameter `gMyWorld` to your reference constant.

A negotiation callback function has the following form:

```
pascal ComponentResult MyNegotiateProc
            (MTNegotiateStateType state,
            const MTMessageHeader* message, long refCon);
```

The parameter `state` contains the negotiation state; `message` points to the message that caused the transition to the current state; `refCon` contains your reference constant.

The `state` parameter may be set to one of the following:

- a positive value to indicate that a negotiation is in progress
- 0 (zero) to indicate that a negotiation has completed successfully
- a negative value to indicate an error in a negotiation progress

Most applications only need to examine the `state` parameter and need not look at the message. Once `state` is set to 0 (the constant `kMTNegotiationCompleted`), you know that the negotiation has successfully concluded, and you can prepare to receive or send data. For further details about the negotiation process, see the section "Negotiation in Depth" beginning on page 5-29. The negotiation callback function is described on page 5-86.

You can find out the currently installed negotiation function by calling the `MTDirectorGetNegotiateProc` function (page 5-52), like this:

```
myErr = MTDirectorGetNegotiateProc (directorInstance,
            &negotiateProc, &negotiateRefCon);
```

When Negotiations Occur

A source stream director always initiates a negotiation with a sink stream director on the remote side. It does this each time you do one of the following:

- establish a transport-level connection to a remote sink stream director and notify the source stream director of the connection
- add or remove a stream
- change an existing stream's format

You establish a connection by using a transport component. See the chapter "Transport Components" in this book for information on how to establish connections. Then, you inform a stream director that you have established a connection by calling the `MTDirectorConnected` function.

You keep a source stream director aware of stream format changes, new streams, and deleted streams by calling the `MTDirectorChangedStreamFormats` function. The section "Changing Stream Formats" beginning on page 5-19 describes how to do that. The source stream director begins a new negotiation as soon as you tell it you have changed the formats.

Monitoring Negotiations on the Source Side

Once you call the `MTDirectorConnected` function to inform a source stream director that you have established a transport-level connection between the connection ends, the source stream director sends a series of messages to the sink stream director at the remote connection end. The first message tells the sink stream director the number of streams the source side wants to send and the stream IDs. And the source stream director calls your negotiation function to indicate that it is beginning stream format negotiations.

Next, the source stream director sends a stream description message for each stream. When it receives an acknowledgment from the sink stream director, the source stream director calls your negotiation function. If the sink stream director rejects a stream, the source stream director sets `state` to a negative value to indicate an error in the negotiation.

(Note that when you use a multicast transport component for your connection, the source stream director does not receive acknowledgments from the sink stream director. See the chapter "Transport Components" in this book for information about multicast transport components.)

When the negotiation is complete, the source stream director calls your negotiation function, setting `state` to the constant `kMTNegotiationCompleted`.

At this point, your application should enable all streams so that media data can begin to flow to the sink stream director. You do this by calling the `MTDirectorStreamEnable` function (page 5-58) with the `stream` parameter set to `kMTAllStreams` and the `enableMode` parameter set to `true`.

Monitoring Negotiations on the Sink Side

A sink stream director first calls your negotiation function after the remote source stream director begins the negotiation, setting the `state` parameter to indicate that a new negotiation is beginning.

Subsequently, the sink stream director calls your negotiation function each time it receives a stream description message.

You can choose to inspect a stream description message and then decide if the negotiation should continue. If you want the negotiation to fail, return an error from your negotiation function. When your negotiation function returns the `noErr` result code, the sink stream director continues the negotiation.

When the negotiation is successfully completed, the stream director calls your negotiation function, setting `state` to `kMTNegotiationCompleted`.

At this point, you need to enable the streams so that you can start receiving data. You do this by calling the `MTDirectorStreamEnable` function with the `stream` parameter set to `kMTAllStreams` and the `enableMode` parameter set to `true`.

Changing Stream Formats

When an application or a user resizes a source view, modifies the audio sampling rate, selects a different audio or video compressor, or changes the format of a stream in some other way while a connection is active, the source stream director needs to know about it.

You don't need to inform a stream director about changes to stream formats when one of the following occurs:

- Your application uses a stream controller component. The controller component notifies the stream director for you.
- You call a stream director directly to modify some characteristic of a stream (for example, when you call the `MTDirectorSetClip` function to change the clipping region for a stream). In this case, the stream director already knows of the stream format change by virtue of your calling the function.
- You call the `MTDirectorSetGWorld` function to change local display characteristics.

You do need to tell the source stream director that stream formats are about to change any time you bypass a stream controller or stream director component to directly manipulate a source of media data, such as a QuickTime sequence grabber. You inform the source stream director that one or more stream formats are about to change by calling the `MTDirectorChangedStreamFormats` function (page 5-60) with the `finished` parameter set to `false`.

Then you call the functions necessary to make your changes. For example, if you are using a QuickTime sequence grabber component as a source of stream data, you can do one or more of the following:

- allow the user to change the current configuration of a sound or video channel by calling the `SGSettingsDialog` function (`SGSettingsDialog` makes the changes when the user closes the dialog box)
- add a new sound or video channel by calling the `SGNewChannel` function
- change the compressor used for a video stream by calling the `SGSetVideoCompressorType` function
- change the sound input device for a sound stream (for example, from microphone to CD player) by calling the `SGSetSoundInputDriver` function
- change the audio characteristics of a sound stream (for example, going from monophonic 8-bit sound to stereo 16-bit sound) by calling the `SGSetSoundInputParameters` function

(Sequence grabber functions are described in *Inside Macintosh: QuickTime Components*.)

After you've changed the stream formats, you need to call `MTDirectorChangedStreamFormats` again, this time setting the `finished` parameter to `true`. That tells the stream director that the stream formats are now changed. Listing 5-2 illustrates how to use the `MTDirectorChangedStreamFormats` function when using the sequence grabber settings dialog box.

Listing 5-2 Changing stream formats

```
ComponentResult MySetupChannelDialog (void)
{
   ComponentResult   myErr;

   myErr = MTDirectorChangedStreamFormats
                           (gStreamDirector, false);
   if (myErr == noErr) {
      myErr = SGSettingsDialog (gSGInstance, gSGChannelInstance,
                              0, nil, 0L,
                              myModalFilterProc, 0);
      if (myErr == noErr) {
         myErr = MTDirectorChangedStreamFormats
                  (gStreamDirector, true);
```

```
            if (myErr == noErr)
                SaveSequenceGrabberSettings();
        }
    }
    return myErr;
}
```

Once you tell a source stream director that stream formats have changed, it queries the component that generates the stream to find out exactly what has changed. (In the code example, the stream director would query a QuickTime sequence grabber.) Then the source stream director renegotiates the stream formats.

Because stream IDs do not persist across changes in stream format, you need to call the `MTDirectorGetNextStream` function (page 5-56) to get new stream IDs after changing stream formats.

Note
When a connection exists and a stream is added or removed or a stream format changes, a stream director renegotiates *all* stream formats. As a result, changing any stream format changes the stream IDs of all existing streams. It also means that stream format changes are expensive in terms of processing time. ◆

Managing Streams

The stream director API allows you to get information about streams and to start and stop the flow of media data for a particular stream or for all streams managed by a stream director.

To find out how many streams are managed by a stream director, you can call the `MTDirectorGetNumberOfStreams` function (page 5-56). The function returns the number of streams in the `numStreams` parameter, like this:

```
myErr = MTDirectorGetNumberOfStreams (directorInstance,
            &numStreams);
```

A stream director maintains a list containing the stream ID and stream type of each stream it manages. (The stream type indicates the type of media data carried by the stream—for instance, video or sound.)

You can use the `MTDirectorGetNextStream` function (page 5-56) to get the stream ID and stream type of each stream managed by a stream director.

To cycle through the list, set the stream ID to `nil` to start at the beginning of the list, as shown in Listing 5-3. The function returns information about the first stream in the list. Then, you can repeatedly call `MTDirectorGetNextStream`, passing it the stream ID and stream type that the function returned in the previous call. The function returns the stream ID and stream type of the next stream in the list. When

`MTDirectorGetNextStream` sets the stream ID to `nil`, you've reached the end of the list.

Listing 5-3 Getting all stream IDs from a stream director

```
void MyGetStreamIDsAndTypes(directorInstance)
   MTDirectorComponent        directorInstance;
{
   MTStreamID                 myStreamID;
   MTStreamType               myStreamType;
   ComponentResult            myErr;

   myStreamID = nil;

   do {
   myErr = MTDirectorGetNextStream (directorInstance,
            &myStreamID, &myStreamType);

      /* save the returned stream ID and stream type */

   } while (myStreamID != nil);
}
```

IMPORTANT

A stream ID is not a persistent identifier. All existing stream IDs change when a connection is established or torn down, when the format of any stream changes, and when a stream is added or removed. You should call the `MTDirectorGetNextStream` function to get a stream ID each time you need one. You should not assume that a stream ID remains valid over time. The `streamReference` field of the stream description structure contains a unique and persistent identifier for a stream. ▲

To get the stream description structure for a stream, you call the `MTDirectorGetStreamDescription` function (page 5-57). You specify the stream ID of the stream whose description you want, allocate memory for the stream description structure, and provide a handle to it. The function resizes the handle and returns a stream description structure containing descriptive information about the stream, including its stream ID, its stream type, its persistent stream reference value, its time scale, and characteristics that are specific to its media type. The stream description structure (`MTStreamDescription`) is described on page 5-42.

A stream is either enabled or disabled. If a stream is enabled, its data is acted on by a stream director—the data is routed to other components for transmission to the remote side, for local display, for recording, and so forth. If a stream is disabled, its data is ignored by a stream director and is not routed to any other components or processed in any way. You can find out if a stream is enabled by calling the `MTDirectorIsStreamEnabled` function (page 5-59), as shown here:

```
myErr = MTDirectorIsStreamEnabled (directorInstance, streamID,
                                   &streamState);
```

New streams are disabled by default. It is up to you to enable the stream so that data can flow. To enable a stream, you call the `MTDirectorStreamEnable` function (page 5-58), with the `enableMode` parameter set to `true`, like this:

```
myErr       = MTDirectorStreamEnable (directorInstance,streamID,
                                      true);
```

If you wanted to disable a stream, you would set the `enableMode` parameter to `false`.

You can also enable and disable, at one time, all of the streams managed by a stream director. To enable or disable all streams, call `MTDirectorStreamEnable` and specify the constant `kMTAllStreams` as the stream ID.

It's possible to determine the enable state of all streams by calling the `MTDirectorIsStreamEnabled` function and specifying `kMTAllStreams` as the stream ID. Be careful here. The function returns the setting that you previously specified *for all streams*.

If you enable all streams as a group and then disable one or more of them individually, the setting returned by the `MTDirectorIsStreamEnabled` function is inconsistent with the current state of one or more of the streams.

Consider the following example. You call the `MTDirectorStreamEnable` function with the `stream` parameter set to `kMTAllStreams` to enable all streams as a group. Then you call `MTDirectorStreamEnable` one or more times for an individual stream, setting `enableMode` to `false` to disable the stream. Later, you call `MTDirectorIsStreamEnabled` with `stream` set to `kMTAllStreams` to query the enable state of all streams as a group. The function reports that all streams are enabled, even though some streams are disabled.

Also take note of what happens if you call `MTDirectorIsStreamEnabled` and specify `kMTAllStreams` as the stream ID and you did not previously set an enable state for all streams. In this situation, the function reports that all streams are disabled because the default enable state for all streams, as a group, is disabled.

Managing Streams Containing Visual Data

When you work with streams containing visual data, you must tell a stream director how you want the data to be displayed. The stream director API allows you to control the following display characteristics:

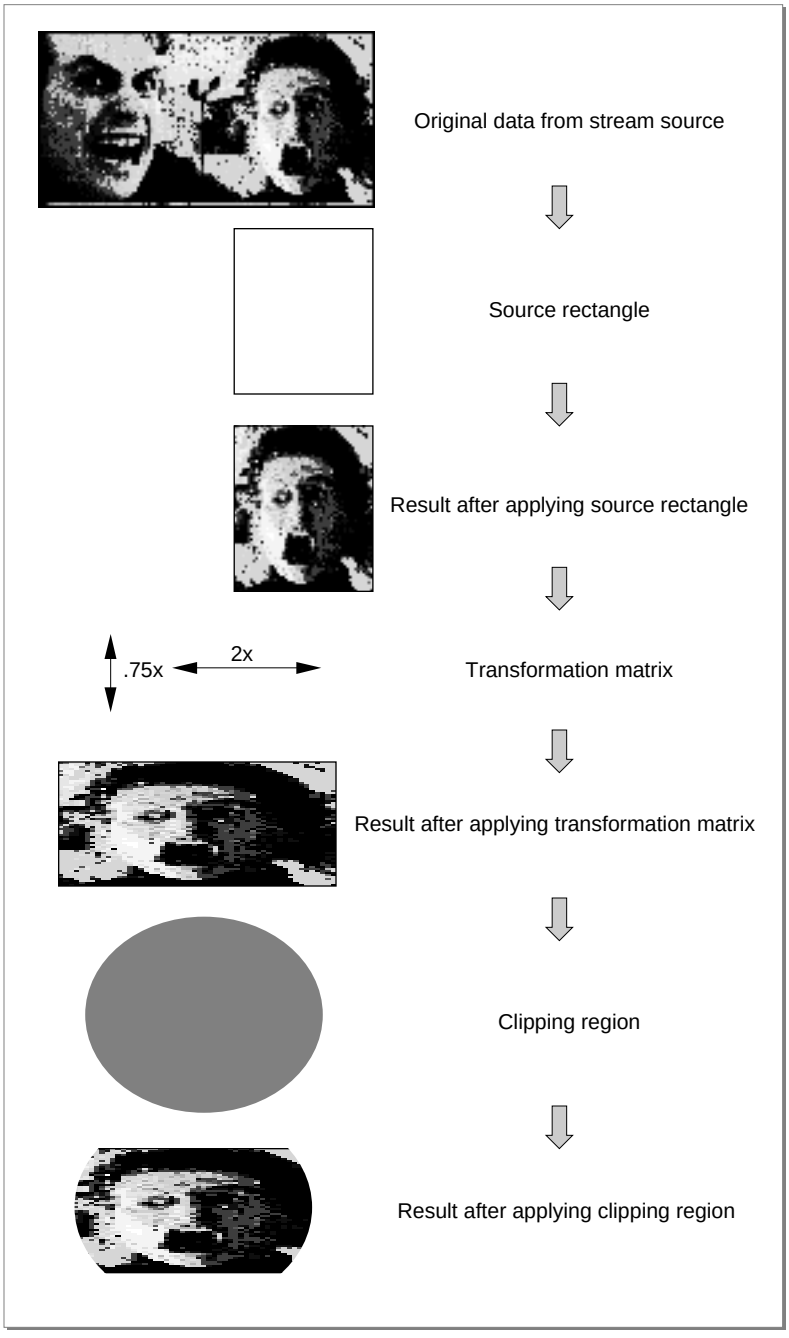
- the graphics world
- the source rectangle
- the display transformation matrix
- the clipping region
- the frame rate and maximum frame rate

When working with a stream that contains visual data, you can call these functions:

- the `MTDirectorSetRect` function (page 5-63) to set the source rectangle. The source rectangle is the base rectangle to which the transformation matrix and the clipping region are applied. The source rectangle must be smaller than or equal to the stream's boundary rectangle and located within or congruent to it. (A stream's boundary rectangle is set by the component that generates the stream data.)
- the `MTDirectorSetMatrix` function (page 5-65) to set the transformation matrix used for scaling, translating, and rotating images. (Transformation matrices are discussed in *Inside Macintosh: QuickTime*.)
- the `MTDirectorSetClip` function (page 5-67) to set the clipping region used to limit the area in which the visual data is displayed.

Figure 5-4 provides an example of visual media data, modified by applying a source rectangle, a transformation matrix, and a clipping region. A source stream director applies the source rectangle to the data coming from its attached media component. A sink stream director applies the source rectangle to media data coming from its attached transport component.

Figure 5-4 Modifying video data



After you attach a media component to a source stream director (by calling the `MTDirectorSetMediaComponent` function), you need to call the `MTDirectorSetGWorld` function (page 5-62) to set the graphics port and graphics

device for a stream or all streams. Both source and sink stream directors need to call `MTDirectorSetGWorld` before they enable streams.

The stream director API also allows you to retrieve the boundary rectangle that encloses the boundary rectangles of all the streams managed by a given stream director. This encompassing boundary rectangle is referred to as the *box*. The `MTDirectorGetBox` function (page 5-68) calculates the box for you.

When working with outgoing streams, you can set a frame rate and maximum frame rate.

Note
The term *frame* is often used when referring to video data. The term *sample* is used when referring to either sound or video data. A chunk of media data may contain a frame, a sample, or a group of samples. ◆

The frame rate refers to the number of frames of media data per second that you wish to transmit over a connection. Frame rates apply only to streams whose media data is organized into discrete samples, such as video data. The frame rate of such data is variable. In contrast, some types of media data, such as sound, are characterized by a fixed sample rate. You don't set a frame rate for this type of media data.

You can use the `MTDirectorSetFrameRate` function (page 5-69) to set a desired frame rate and the `MTDirectorSetMaxFrameRate` function (page 5-71) to set an upper limit on the transmission frame rate. The actual frame rate varies up to the maximum frame rate depending on network traffic conditions, CPU performance capabilities, and so forth.

When you work with incoming streams, setting a frame rate and maximum frame rate has no meaning.

The stream director API also provides functions to retrieve the source rectangle, transformation matrix, clipping region, frame rate, and maximum frame rate. These functions are described immediately after the companion function that sets a value. For instance, you'll find the description of the `MTDirectorGetRect` function immediately after that of `MTDirectorSetRect` in the reference section of this chapter.

Managing Streams Containing Audio Data

You can use a stream director to manage the audio characteristics of streams that contain audio data. As with the functions that control characteristics of visual streams, most of the functions that apply to streams containing audio data are part of a get/set pair—you use one to provide values for certain characteristics and the other to retrieve those values.

The term *sound* is used in this chapter to refer to a specific type of media data defined by QuickTime (`SoundMediaType`). The term *audio* is used to refer to any type of media data that can be heard. It is a general category containing any number of specific types of audio data. For example, MIDI streams and sound streams both contain audio data. These streams can be generally described as containing audio data. They are specifically described as MIDI streams and sound streams-—that is, they contain audio data in a specific format.

The stream director API allows you to control these audio characteristics:

- volume level, which is a relative measure of the loudness of the audio data when it is played. Both incoming and outgoing streams have a volume level. For an outgoing stream, the volume level affects the local play-through of the stream data. The `MTDirectorSetVolume` (page 5-75) and `MTDirectorGetVolume` (page 5-76) functions allow you to set and get the volume level for a stream.
- audio input gain, which is a relative measure of sensitivity. It applies only to outgoing streams. As the gain is increased, louder audio data is included in the stream, because of increased sensitivity. The `MTDirectorSetGain` (page 5-78) and `MTDirectorGetGain` (page 5-79) functions allow you to set and get the audio input gain for an outgoing stream.
- sound threshold, which specifies a sound level for an outgoing stream. It can be used by sound flow control components for a sound flow control algorithm. The `MTDirectorSetSoundThreshold` (page 5-80) and `MTDirectorGetSoundThreshold` (page 5-81) functions allow you to set and get the sound threshold for an outgoing stream.

In addition to setting and getting values for those characteristics, you can call the `MTDirectorGetSoundLevel` function (page 5-82) to retrieve the sound level of a stream. Sound level refers to the energy level (or signal strength) of sound detected by a sound input device.

Note
If you call the `MTDirectorSetGain` function and specify an incoming stream, the function sets a maximum volume level for the stream. Maximum volume level sets an upper limit on the loudness of an incoming stream. ◆

Volume level and audio input gain apply to any stream that contains audio data. Sound threshold and sound level apply only to sound streams, a specific type of audio stream.

Table 5-1 provides summary information about the audio characteristics.

Table 5-1 Audio stream characteristics

Characteristic	Applies to all audio streams	Applies to sound streams only	Applies to incoming streams	Applies to outgoing streams	Set/get
Volume	X		X	X	Set and get
Maximum volume	X		X		Set and get
Gain	X			X	Set and get
Sound threshold		X		X	Set and get
Sound level		X		X	Get

Advising of Environmental Changes

The stream director API provides functions that allow you to inform a stream director about changes in the display configuration and the application environment. A stream director uses the information you provide to control its streams. Many common user actions in an application can require adjustments in how a stream director manages its streams. Among these actions are

- dragging, selecting, and resizing windows
- moving the application from the foreground to the background and vice versa
- pulling down menus and making menu selections
- modifying the characteristics of a stream during a connection—for example, changing from mono to stereo sound

A window changes when a user moves, resizes, or closes it or when a user selects a different window. In response to these events, you should call the `MTDirectorChangedWindow` function (page 5-84) twice. The first call to `MTDirectorChangedWindow` tells a stream director that the window state is about to change—it's a hint to a stream director to manage the flow of streams in light of an impending window change. Then you should call the Window Manager functions needed to implement the window change. When the change is complete, call `MTDirectorChangedWindow` again to notify the stream director that the window state has changed. Listing 5-4 shows the sequence.

Listing 5-4 Telling a stream director about window changes

```
changedWindow = false;  /* window state is about to change */
myErr = MTDirectorChangedWindow (directorInstance, changedWindow);

/* call Window Manager functions to handle the window change */

changedWindow = true;   /* window state has changed */
myErr = MTDirectorChangedWindow (directorInstance, changedWindow);
```

You use the `MTDirectorPause` function (page 5-85) to make a stream director pause or resume processing of visual stream data. Prior to performing an operation during which visual data cannot be processed, such as when a user pulls down a menu and makes a menu selection, you should have the stream director pause. Then perform the user action and follow up by telling the stream director to resume processing of visual stream data. The code below shows the sequence:

```
pauseMode = true;    /* pause operations on visual data */
myErr = MMTStreamDirectorPause (directorInstance, pauseMode);

/* handle the user operation */
```

```
pauseMode = false;   /* resume operations on visual data */
myErr = MTDirectorPause (directorInstance, pauseMode);
```

Negotiation in Depth

This section provides more detailed information about the negotiation process. For most applications, the section "Monitoring Stream Format Negotiations" beginning on page 5-17 contains what you need to know about negotiation—you do not need to be familiar with the material presented here. However, this more detailed description will be of interest to you if you require more understanding and control of the negotiation process. This section assumes that you have already read "Monitoring Stream Format Negotiations."

A negotiation callback function has the following form:

```
pascal ComponentResult MyNegotiateProc
            (MTNegotiateStateType state,
            const MTMessageHeader* message, long refCon);
```

The parameter `state` contains the negotiation state; `message` points to the MovieTalk message that caused the transition to the current state; `refCon` contains your private data.

When a stream director calls your negotiation callback function, it sets the function's `state` parameter to a negative value or to one of the following values:

```
enum {
   kMTNegotiationCompleted          = 0,
   kMTNegotiateOpenChannel          = 1,
   kMTNegotiateOpenStream           = 2,
   kMTNegotiateAckOpenStream        = 3
};
```

Constant descriptions

`kMTNegotiationCompleted`
: Stream formats have been successfully negotiated with the remote stream director.

`kMTNegotiateOpenChannel`
: The source stream director is beginning negotiation of stream formats.

`kMTNegotiateOpenStream`
: Negotiation is in progress. The sink stream director received a stream description message.

`kMTNegotiateAckOpenStream`
: Negotiation is in progress. The source stream director received an acknowledgment for a particular stream description.

If your negotiation function is called with the parameter `state` set to a negative value, the negotiation has failed. The value of the `state` parameter is the error code that describes the failure, and the parameter `message` points to a structure containing the MovieTalk message that caused the failure.

Most applications need to examine only the `state` parameter and need not look at the MovieTalk message. Once `state` is set to `kMTNegotiationCompleted`, you know that the negotiation has successfully concluded, and you can prepare to receive or send data.

The message types for open channel, stream description, and acknowledgment messages are `kMTOpenChannelMessageType`, `kMTOpenStreamMessageType`, and `kMTAcknowledgeMessageType`. MovieTalk messages are described in the chapter "MovieTalk Protocol Messages" in this book.

The next two sections, "Source Side Negotiations" and "Sink Side Negotiations," provide a more detailed look at a negotiation from the source side and the sink side, respectively. Figure 5-5 on page 5-40 illustrates a typical point-to-point stream format negotiation, including the MovieTalk messages that are exchanged and the calls to your negotiation function that are triggered by those messages.

Source Side Negotiations

After you call the `MTDirectorConnected` function to inform the source stream director that you have established a transport-level connection, the source stream director sends an open channel message to the sink stream director. The open channel message tells the sink stream director the number of streams the source side wants to send, the stream types, and stream IDs. The source stream director calls your negotiation function, setting the `state` parameter to `kMTNegotiateOpenChannel` to indicate that it is beginning stream format negotiations; the `message` parameter points to the open channel message.

Next, the source stream director sends a stream description message for each stream. After receiving an acknowledgment from the sink stream director, the source stream director calls your negotiation function. If the sink stream director accepted a stream, the source stream director sets the `state` parameter to `kMTNegotiateAckOpenStream`. If the sink stream director rejected a stream, the source stream director sets `state` to a negative value to indicate an error in the negotiation. In either case, the `message` parameter points to the acknowledgment message received.

(Note that when you use a multicast transport component for your connection, the source stream director does not receive acknowledgments from the sink stream director.)

When the negotiation is complete, the source stream director calls your negotiation function, setting `state` to `kMTNegotiationCompleted` and `message` to point to the last acknowledgment message it received.

At this point, your application should enable the streams so that data can begin to flow to the sink stream director. You do this by calling the `MTDirectorStreamEnable` function with the `stream` parameter set to `kMTAllStreams` and the `enableMode` parameter set to `true`.

Sink Side Negotiations

A sink stream director first calls your negotiation function after the remote source stream director begins the negotiation, setting the parameter s`tate` to the value `kMTNegotiateOpenChannel` to indicate that a new negotiation is beginning. The `message` parameter points to the open channel message just received.

Subsequently, the sink stream director calls your negotiation function each time it receives a stream description message, setting `state` to `kMTNegotiateOpenStream` and `message` to point to the stream description message.

Your application can inspect any stream description message and then decide if the negotiation should continue. If you want the negotiation to fail, return an error from your negotiation function.

When the negotiation is successfully completed, the stream director calls your negotiation function, setting `state` to `kMTNegotiationCompleted` and `message` to point to the final acknowledgment message the stream director sends to the source stream director.

At this point, you need to enable the streams so that you can start receiving data. You do this by calling the `MTDirectorStreamEnable` function with the `stream` parameter set to `kMTAllStreams` and the `enableMode` parameter set to `true`.

Creating Stream Director Components

The following sections discuss topics you need to know about if you are creating a stream director component. If your application only uses the services of a stream director component, you do not need to read this section.

Your stream director component must support all of the functions in the stream director API. The request code constants that identify the functions are listed on page 5-41.

See the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox* for information on how to create any component.

Creating Streams

When you call the transport component function `MTTransportNewMediaStream` to create a new stream, a transport component generates and returns a unique stream ID for the new stream.

You must set the enable status of a new stream to disabled. Initially, no data is carried in a newly created stream. It is up to the application or the conference component to enable a stream at an appropriate time.

You need to maintain an enable status that applies globally to all the streams managed by your stream director. You return this global status when asked for the enable status of all streams. You must initialize the global enable status to disabled.

Handling Media Data

This section discusses how a stream director gets media data and what it does with the data after receiving it.

Figure 5-3 on page 5-12 shows the calling relationships between stream directors, applications, and other components and the path of media data as it flows between components. You may find it helpful to refer to the figure as you read this section.

Source and sink stream directors get media data by installing a media data callback function. A source stream director typically installs its data callback function on its attached media component—that is the behavior of the Apple-provided source stream director. A sink stream director installs its data callback function on its attached transport component.

If your source stream director uses a QuickTime sequence grabber component as its media component, the stream director needs to call the `SGSetDataProc` function (described in *Inside Macintosh: QuickTime Components*) to install its media data callback function.

An attached media component calls your data function whenever it has media data to pass to your source stream director. Each time a source stream director receives data from its media component, the stream director needs to create a chunk record structure, shown here:

```
struct MTStreamChunkRecord {
   Byte              *chunkPtr; /* pointer to chunk of media data */
   MTChunkSize       chunkSize; /* size of chunk in bytes */
   MTChunkTime       chunkTime; /* time of chunk */
   MTChunkPriority   chunkPriority; /* priority of chunk */
   MTStreamID        chunkStreamID; /* stream ID */
   ComponentResult   chunkError; /* error associated with chunk */
   long              chunkRefCon; /* refcon associated with chunk */
   UInt32            chunkReserved; /* reserved */
   long              chunkTimePlayed; /* time chunk was played */
   MTReleaseUPP      chunkReleaseProcCallBack; /* callback function */
   long              chunkPrivate; /* reserved */
   MTStreamOptions   chunkStreamOptions; /* media-specific information */
   UInt8             chunkReserved2; /* reserved */
   MTSequenceNum     chunkFrameNumber; /* sequence number of chunk */
   UInt16            chunkReserved3; /* reserved */
};
```

(The `MTStreamChunkRecord` structure is completely described in the chapter "Transport Components" in this book.)

A source stream director provides values for the `chunkPtr`, `chunkSize`, `chunkTime`, `chunkPriority`, `chunkStreamID`, and `chunkStreamOptions` fields of the structure.

After creating a chunk record structure, a source stream director passes it to an appropriate flow control component, if one exists, or to its attached transport component.

If a recorder component installed a recording callback function, your stream director must call it so that the data can be recorded. Recorder components provide a reference constant when they call the `MTDirectorSetRecordProc` function (page 5-49) to install a recording function. Your stream director needs to put it in the `chunkRefCon` field before passing the structure to the recorder component. That concludes a source stream director's processing of a given chunk.

A sink stream director installs a media data callback function on its attached transport component by calling the `MTTransportSetMediaDataProc` function (described in the chapter "Transport Components" in this book). The data function has this form:

```
pascal OSErr MyMediaProc (MTStreamChunkRecordPtr chunk);
```

After receiving media data from the network, a transport component calls your data function to pass the chunk record structure to your sink stream director.

The sink stream director should check the value of the `chunkError` field. A value of 0 in the field indicates a valid chunk of data. A nonzero value in the `chunkError` field indicates some problem with the chunk—for instance, some of the chunk data may have been lost during transmission over the network. If a valid chunk arrives too late to play, your sink stream director should set this field to the `mtChunkTimedOutErr` result code.

Checking the chunk error can also help a sink stream director synchronize the playing of different streams. For example, assume a sink stream director receives a video and a sound stream. Suppose that portions of the video stream arrive incomplete. In this situation, the sink stream director should account for the asymmetry of the sound and video data when passing it to stream players or to flow control components so that the streams are synchronized upon presentation to a user.

Then your sink stream director should pass the chunk record structure to an appropriate flow control component, if one exists, or to a stream player component. If a recording function has been installed, the sink stream director must pass the chunk to the recorder component also. Just like a source stream director, a sink stream director sets the `chunkRefCon` field to a recorder component's reference constant before calling a recording callback function.

There is one other field in the chunk record structure that a sink stream director can choose to set: `chunkReleaseProcCallBack`. This field contains a pointer to a function that is called by a transport component before it releases the memory occupied by the chunk record structure.

One reason you might want to provide a callback function here is that stream players vary in the time they take to play media data, depending on the decompression algorithm they use and other factors. If a stream director can stay aware of how quickly each of its players plays data, it can synchronize different streams more easily.

After playing a chunk of media data, a stream player sets the `chunkTimePlayed` field of the chunk record structure to the time that it finished playing the chunk data. Then it calls a release function to release the chunk memory. If you provide a callback function

to be called before the memory for chunk record structure is deallocated, you can read the value in the `chunkTimePlayed` field.

You can use this information to maintain smooth synchronization of streams you are managing by adjusting the rate at which you pass chunk data to the players. For example, suppose a given video stream should be played at 5 frames per second. For a given second, the play times are at 0.2, 0.2, 0.4, 0.6, and 0.8 seconds. If a stream player indicates in the `chunkTimePlayed` field that it plays a chunk in 0 seconds, a sink stream director should pass the chunks at the play times noted previously. However, if the player reports that it takes a tenth of a second to play a chunk, then the sink stream director should pass the chunk a tenth of a second before its play time—that is, pass it at 0.3 instead of 0.4.

Once a chunk record structure is appropriately filled out, the sink stream director passes the chunk record structure to a flow control component, if one exists, or to a stream player component. If a recording function has been installed, the stream director calls it to record the data. That concludes its processing of a given chunk.

Setting Up Other Components

Stream director components interact with several types of components. Media and transport components are the responsibility of an application or a conference component—the application or conference component opens, configures, and closes them. These types are said to be owned by the application or conference component. By calling the `MTDirectorSetMediaComponent` or `MTDirectorSetTransport` function, an application or a conference component tells a stream director which media or transport component instance it needs to work with.

However, a stream director itself owns other components. It is responsible for opening, maintaining, and closing them. A sink stream director owns stream player components that play media data. Source and sink stream directors own flow control components that control the flow of media data to transport components and to stream players.

The next two sections give a brief overview of how stream directors interact with flow control components and stream player components. If you are not already familiar with flow control components and stream players, you need to read the chapters “Flow Control Components” and “Stream Player Components” in this book.

Flow Control Components

Flow control components regulate the flow of media data in an effort to achieve optimal performance. They reduce or increase the data transmission rate in response to a variety of factors, including

- the level of network traffic and consequent delay
- the performance characteristics of the computers at the ends of a QuickTime Conferencing connection

A flow control component is specific to a given type of media data. With QuickTime Conferencing 1.0, Apple provides flow control components for video and sound data.

Once a stream director knows the type of media data to be carried in a given stream, it needs to find out if a flow control component for that type of media data is available locally.

To do this, you call the Component Manager function `OpenDefaultComponent` to open a flow control component whose subtype matches the type of media data in a stream. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for information on how to use this function.

The availability of appropriate flow control components determines where a stream director sends data. A source stream director passes the media data that it gets from its media component to a flow control component, rather than its attached transport component, if a flow control component is available for that type of media data. The flow control component, in turn, forwards the data to the transport component.

A sink stream director passes the media data that it gets from its transport component to a flow control component, rather than a stream player component, if a flow control component is available for that type of media data. The flow control component, in turn, forwards the data to the stream player.

Figure 5-3 on page 5-12 illustrates these alternate paths for the flow of media data.

If a flow control component is available for a stream of a given media data type, the stream director needs to open it and prepare to use it. For instance, you need to call the `MTTransportGetInfo` function for each of your attached transport components to determine if flow control should be enabled. Then, you need to call the `MTFlowControlEnable` function for each flow control component you attached to those transport components. The function allows you to enable or disable flow control for the streams you are managing. See the chapter "Flow Control Components" in this book to learn how you set up and use flow control components.

Stream Player Components

A stream player component plays a specific type of media data. Apple provides stream player components for video and sound data.

During stream format negotiations, a sink stream director receives stream description messages, each one describing the type of media data the source stream director wants to send in that stream. A sink stream director must determine if stream players for those types of media data are available locally.

The component subtype differentiates stream players on the basis of the type of media data that they support. For example, the video stream player has the subtype `kMTVideoSubType`. To find out if a stream player for a specific type of media data is available, you call the Component Manager function `OpenDefaultComponent` to open a stream player component whose subtype matches the type of media data in a given stream. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for information on how to use this function.

If the stream players are available, a stream director needs to open a stream player for each type of media data it will receive and prepare to use them. See the chapter "Stream

Player Components" in this book to learn how you set up and use stream player components.

Working With a Recorder Component

Stream directors do not own, nor are they attached to, recorder components. However, a recorder component can install a recording callback function on a stream director by calling the `MTDirectorSetRecordProc` function, like this:

```
myErr = MTDirectorSetRecordProc (directorInstance, recordProc,
                                 recordRefCon);
```

The `recordProc` parameter contains a universal procedure pointer to a recording function; the `recordRefCon` parameter contains a value private to a recorder component that you pass to the recording function when you call it.

A recording callback function that your stream director calls has the following form:

```
pascal OSErr MyRecordProc (MTStreamChunkRecordPtr chunk);
```

Once a recording function is installed, your stream director must call it whenever it has prepared a new chunk record structure. For instance, if a recorder component provided a reference constant when it call the `MTDirectorSetRecordProc` function, you need to set the `chunkRefCon` field to that reference constant. When the chunk record structure is ready, you pass it to the recording function.

See the section "Handling Media Data" beginning on page 5-32 for more information about how a stream director prepares a chunk record structure.

Getting Processor Time

A stream director needs periodic processor time to perform its tasks, such as negotiating stream formats. To make sure it gets that time, a stream director should use the QuickTime Conferencing idler component.

The idler component provides a convenient method to get periodic processing time. Furthermore, it provides the time when it is safe to move memory. This allows your stream director to call Memory Manager functions and Toolbox functions that might move memory.

Your stream director should register a callback function with the idler component by calling the idler component's `MTIdlerGimmeTime` function. You provide the address of the function to be called periodically, the minimum interval that should elapse between calls to that function, and a reference constant reserved for your use. The idler component thereafter calls your function periodically at a time it is safe to make Memory Manager requests.

For more information on how to use the idler component and the `MTIdlerGimmeTime` function, see the chapter "Utility Components" in this book.

Managing Memory

If your stream director dynamically allocates and releases memory, it should use the services of the Apple-provided memory component. Since all media data arrives from a transport component at interrupt time and must be buffered until it can be processed, your stream director is likely to need those services.

The memory component provides interrupt-safe memory management services. It manages a common pool of buffers in an application heap and makes the memory available to you at interrupt time. The buffers it provides are guaranteed to be in physical memory—they are not paged out as part of virtual memory management operations. The memory component also allows each component instance serving an application and the application itself to register a grow-zone function.

Your stream director first needs to open and initialize its connection with the memory component by calling the Component Manager's `OpenDefaultComponent` function and the `MTMemoryInit` function.

After that, a stream director should call the `MTMemoryGetOneChunk` and `MTMemoryPutOneChunk` functions to get and release memory as needed. You can call these functions at interrupt time.

The memory component is optimized for getting and releasing a series of similarly sized buffers. You should not use it for getting and releasing randomly sized buffers.

You should register a grow-zone function with the memory component because it allows for calling multiple grow-zone functions. This increases the chances of finding needed memory under low-memory conditions.

For complete information on how to use the memory component, see the chapter "Utility Components" in this book.

Negotiating Stream Formats

Your stream director must implement the stream format negotiation process. A stream format negotiation is an exchange of MovieTalk messages between a source and a sink stream director. A stream director must take certain actions in the proper sequence and correctly create and interpret MovieTalk messages.

The sections "Monitoring Stream Format Negotiations" beginning on page 5-17 and "Negotiation in Depth" beginning on page 5-29 look at negotiation from the perspective of an application using a stream director component. "Monitoring Stream Format Negotiations" provides an overview of the negotiation process, whereas "Negotiation in Depth" contains more detailed information. You should read those sections before reading this one—this section assumes you are familiar with that material.

MovieTalk messages are completely described in the chapter "MovieTalk Messages" in this book. They are referenced, but not explained, here.

A source stream director begins a negotiation by sending an open channel message (message type is `kMTOpenChannelMessageType`) to the sink stream director at the remote connection end. The open channel message tells a sink stream director the

number of streams the remote source stream director wants to send and the streams' types and IDs. (The channel referred to here is the media channel, part of a QuickTime Conferencing connection.) Then, the source stream director calls the installed negotiation function, setting the `state` parameter to `kMTNegotiateOpenChannel` to inform the application that negotiations have begun.

When the sink stream director at the remote connection end receives the open channel message, it also needs to call the installed negotiation function, setting the `state` parameter to `kMTNegotiateOpenChannel` to inform the application that negotiations have begun.

Next, a source stream director needs to send one open stream message (message type is `kMTOpenStreamMessageType`) for each stream in the channel.

A sink stream director must examine each incoming stream description message and determine whether there is a stream player available to play that type of stream. It also needs to call the installed negotiation function, setting the `state` parameter to `kMTNegotiateOpenStream`, to give the application a chance to accept or reject the stream format. Then, it must send an acknowledgment message (message type is `kMTAcknowledgeMessageType`) to either accept or reject the stream.

Note
Media data flows in one direction only from a single source to some number of sinks over a multicast connection. When the attached transport component for a source or a sink stream director is a multicast type, the sink stream director does not send acknowledgment messages to the source stream director, nor does the source stream director expect to receive them. For information on multicast transports, see the chapter "Transport Components" in this book. ◆

When the source stream director receives the acknowledgment message, it calls the installed negotiation function, setting the `state` parameter to `kMTNegotiateAckOpenStream` to inform the application about the acknowledgment.

A source stream director need not wait for an acknowledgment of one stream description message before sending another. However, it needs to use a timeout value to protect against excessive network delay or the loss of the message in the network. If the acknowledgment doesn't arrive before the timeout value expires, the negotiation fails. The stream director calls the negotiation function, setting `message` to `nil` and `state` to the `mtNegotiationTimedOutErr` error code.

Sink stream directors also need to use a timeout value to limit the amount of time it waits for a stream description message it knows should be coming. A sink stream director knows how many stream description messages to expect because the number of streams is defined in the open channel message.

A negotiation ends when one of the following occurs:

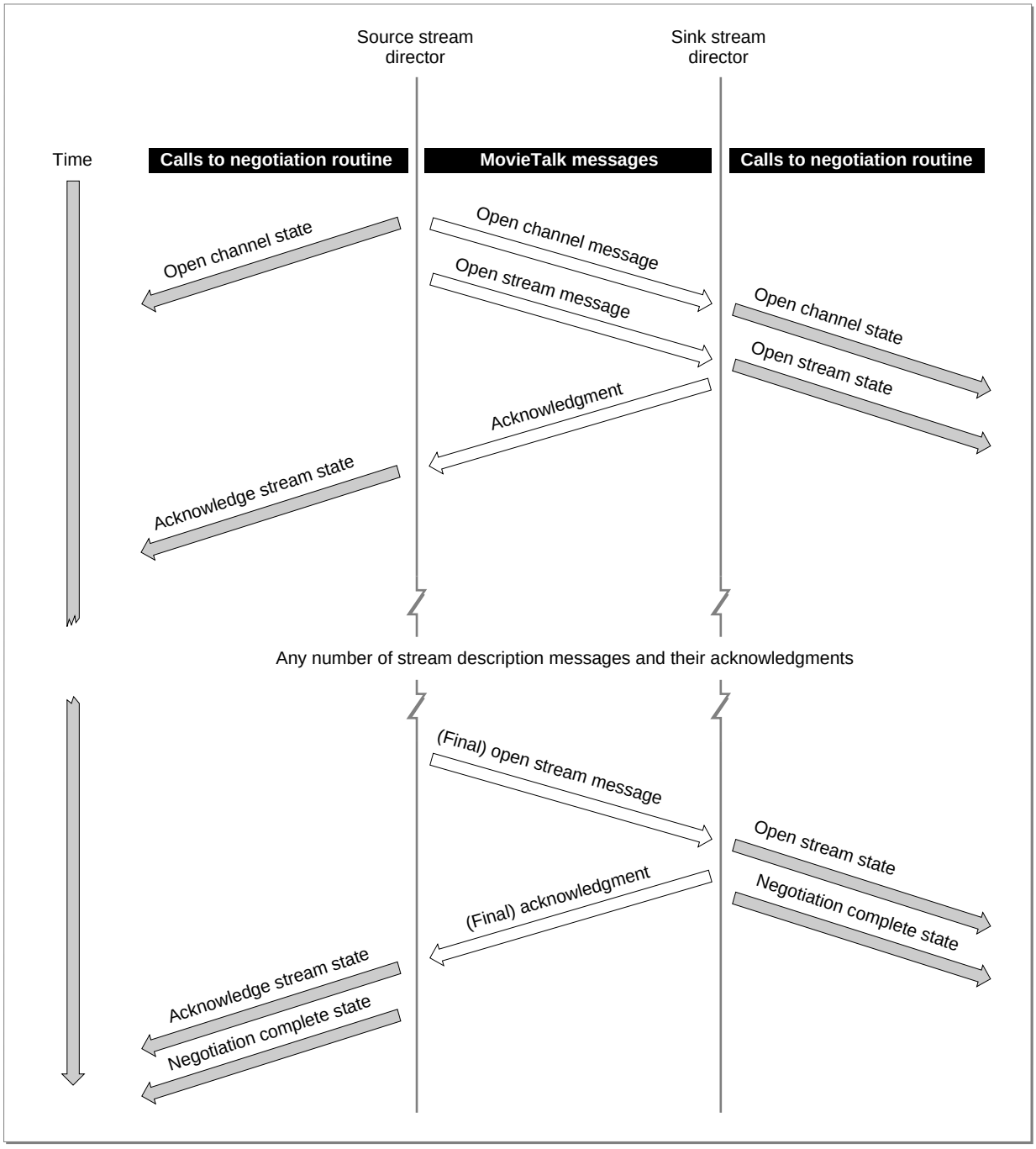
- a sink stream director receives the final stream description message, accepts the stream, and sends the acknowledgment message
- a sink stream director (or the application that's using it) rejects any stream and sends the acknowledgment message

- a timer expires

Note
In a multicast connection, a negotiation ends for a source stream director when the source stream director sends its final open stream message. For a sink stream director, it ends when the sink stream director receives final open stream message. ◆

Figure 5-5 illustrates the exchange of messages and calls to a negotiation function during a typical successful point-to-point negotiation.

Figure 5-5 A successful point-to-point stream format negotiation



Stream Director Components Reference

This section describes the constants, data structures, and functions that make up the API of stream director components.

Constants

This section describes the constants in the stream director API.

Component Type and Subtypes

There are two component types for stream directors, one for source stream directors and the other for sink stream directors. In addition, Apple defines two stream director subtypes to indicate the type of media component or stream player component utilized by the stream director.

```
enum {         /* stream director component types */
   kMTSinkStreamDirectorType     = 'sksd',   /* sink stream director */
   kMTSourceStreamDirectorType   = 'srsd'    /* source stream director */
};

enum {         /* stream director component subtypes */
   kMTGrabberSubType    = SeqGrabComponentType, /* uses sequence grabber */
   kMTPlayerSubType     = kMTPlayerType         /* uses stream player */
};
```

Request Codes

A request code specifies a function in the stream director API. The Component Manager passes the request code to a stream director to indicate which function was called.

```
enum {
   kMTDirectorConnectedSelect             = 1,
   kMTDirectorJoinSelect                  = 2,
   kMTDirectorGetNumberOfStreamsSelect    = 3,
   kMTDirectorGetNextStreamSelect         = 4,
   kMTDirectorStreamEnableSelect          = 5,
   kMTDirectorPauseSelect                 = 6,
   kMTDirectorIsStreamEnabledSelect       = 7,
   kMTDirectorSetNegotiateProcSelect      = 8,
   kMTDirectorGetNegotiateProcSelect      = 9,
   kMTDirectorChangedStreamFormatsSelect  = 10,
```

```
   kMTDirectorSetTransportSelect          = 11,
   kMTDirectorGetTransportSelect          = 12,
   kMTDirectorGetStreamDescriptionSelect  = 13,
   kMTDirectorUpdateSelect                = 14,
   kMTDirectorGetStreamInfoSelect         = 15,
   kMTDirectorSetGWorldSelect             = 16,
   kMTDirectorSetMatrixSelect             = 17,
   kMTDirectorGetMatrixSelect             = 18,
   kMTDirectorSetClipSelect               = 19,
   kMTDirectorGetClipSelect               = 20,
   kMTDirectorGetBoxSelect                = 21,
   kMTDirectorSetFrameRateSelect          = 22,
   kMTDirectorGetFrameRateSelect          = 23,
   kMTDirectorSetMaxFrameRateSelect       = 24,
   kMTDirectorGetMaxFrameRateSelect       = 25,
   kMTDirectorSetRectSelect               = 26,
   kMTDirectorGetRectSelect               = 27,
   kMTDirectorGetPictureSelect            = 28,
   kMTDirectorSetVolumeSelect             = 29,
   kMTDirectorGetVolumeSelect             = 30,
   kMTDirectorSetGainSelect               = 31,
   kMTDirectorGetGainSelect               = 32,
   kMTDirectorSetSoundThresholdSelect     = 33,
   kMTDirectorGetSoundThresholdSelect     = 34,
   kMTDirectorGetMediaComponentSelect     = 35,
   kMTDirectorSetMediaComponentSelect     = 36,
   kMTDirectorGetSoundLevelSelect         = 37,
   kMTDirectorChangedWindowSelect         = 38,
   kMTDirectorSetRecordProcSelect         = 39,
   kMTDirectorGetRecordProcSelect         = 40
};
```

Data Types

This section describes the data types and structures in the stream director component API.

The Stream Description Structure

A stream description structure describes a given stream. The structure includes fields specific to QuickTime Conferencing and a QuickTime sample description structure.

```
struct MTStreamDescription {
   MTStreamType         streamType;
   MTStreamID           streamID;
   MTStreamReference    streamReference;
   TimeScale            streamTimeScale;
   SampleDescription    streamSampleDescription;
};

typedef struct MTStreamDescription MTStreamDescription;

typedef MTStreamDescription **MTStreamDescriptionHandle;
```

Field descriptions

`streamType` The type of media data carried by this stream. Apple defines constants for the video (`VideoMediaType`), sound (`SoundMediaType`), and text (`TextMediaType`) media data types. (These constants are discussed in *Inside Macintosh: QuickTime*.)

`streamID` A unique, temporary identifier for a stream. A transport component assigns the stream ID when an application calls the `MTTransportNewMediaStream` function to create a new stream. It assigns a new stream ID to a given stream when the stream format is renegotiated.

`streamReference` A unique, persistent identifier for a stream. A source stream director component assigns the stream reference, and the reference never changes for that stream.

`streamTimeScale` The time scale for this stream, expressed as the number of time units per second.

`streamSampleDescription` A structure that specifies the characteristics of this stream's media data. The format of a sample description varies by media type. For example, if the stream contains video data, this field contains an `ImageDescription` structure. If the stream contains sound data, it contains a `SoundDescription` structure. If the stream contains text data, it contains a `TextDescription` structure. (All of these structures are defined in *Inside Macintosh: QuickTime*.)

The Negotiation State Type

A stream director uses the `MTNegotiateStateType` data type to inform a negotiation function about the state of a stream format negotiation. During a negotiation, a stream director calls a negotiation function to report on the progress of the negotiation. It passes the function a parameter that indicates the state of the negotiation. The `MTNegotiateStateType` data type defines values for the state of a stream format negotiation.

```
typedef ComponentResult MTNegotiateStateType;
```

A negative state value indicates a failure in the negotiation. Zero and positive values of the `MTNegotiateStateType` data type are defined as follows:

```
enum {            /* values of MTNegotiateStateType */
   kMTNegotiationCompleted          = 0,  /* negotiation complete state */
   kMTNegotiateOpenChannel          = 1,  /* open channel state */
   kMTNegotiateOpenStream           = 2,  /* open stream state */
   kMTNegotiateAckOpenStream        = 3   /* acknowledge stream state */
};
```

Constant descriptions

`kMTNegotiationCompleted`
: Stream formats have been successfully negotiated with the remote stream director.

`kMTNegotiateOpenChannel`
: The source stream director is beginning negotiation of stream formats.

`kMTNegotiateOpenStream`
: Negotiation is in progress. The sink stream director received a stream description message.

`kMTNegotiateAckOpenStream`
: Negotiation is in progress. The source stream director received an acknowledgment for a particular stream description.

Functions

This section describes the functions that are provided in the stream director component API. These functions are described from the perspective of one who calls a stream director component. This caller can be an application. Often, it is a conference component or a stream controller component. To use a stream director, you call the functions as described here. If you are developing a stream director component, your component must support the functions as they are described here.

You can specify a source stream director function for every function in the API. The following functions do not apply to sink stream directors:

- `MTDirectorSetMediaComponent`
- `MTDirectorGetMediaComponent`
- `MTDirectorChangedStreamFormats`
- `MTDirectorSetFrameRate`
- `MTDirectorGetFrameRate`
- `MTDirectorSetMaxFrameRate`
- `MTDirectorGetMaxFrameRate`
- `MTDirectorSetSoundThreshold`

- `MTDirectorGetSoundThreshold`

All of the functions take a component instance parameter. You can obtain a component instance by calling the Component Manager's `OpenDefaultComponent` function. See page 5-14 for a description of how to call this function.

This section describes the following groups of functions:

- "Attaching Other Components to a Stream Director" describes the functions you use to attach transport components and media components to a stream director.
- "Installing Callback Functions" describes the functions you use to install negotiation and recording callback functions.
- "Managing a Stream Director" describes the functions you use to inform a stream director about a QuickTime Conferencing connection and to inform local source and sink stream directors that share a transport-level connection about each other.
- "Managing Streams" describes the functions you use to get the number of streams managed by a stream director, get information about a stream, and enable or disable a stream.
- "Getting and Setting Characteristics of Visual Stream Data" describes the functions you use to control characteristics of a stream's visual display.
- "Taking a Picture of Visual Stream Data" describes the function you use to capture an image of a stream.
- "Getting and Setting Characteristics of Audio Stream Data" describes the functions you use to control characteristics of a stream's audio playback.
- "Advising of User Events" describes the functions that you use to inform a stream director of user events.

Attaching Other Components to a Stream Director

You use the functions described in this section to get and set a stream director's attached transport and media components.

MTDirectorSetTransport

The `MTDirectorSetTransport` function attaches a transport component to a stream director.

```
pascal ComponentResult MTDirectorSetTransport
            (MTDirectorComponent sdc,
            MTTransportComponent tc);
```

`sdc` The stream director component you are using.

`tc` The transport component instance to attach to the stream director. You obtain a component instance when you open the transport component by calling the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

DESCRIPTION

You call the `MTDirectorSetTransport` function to attach a transport component to a stream director. At any given time, a stream director can have one attached transport component. A stream director uses its transport component to exchange media data and control information across a connection.

Typically, a stream director uses the same transport instance over the lifetime of a connection. However, if a user changes the type of connection after it is under way (for example, from point-to-point mode to multicast), you need to attach a new transport component to the stream director. To do so, first call the `MTDirectorConnected` function. Set its `connected` parameter to `false`, indicating that the connection is no longer valid. Next, call `MTDirectorSetTransport` to attach a multicast transport component.

This function can return transport component errors.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTDirectorConnected` function is described page 5-53.

The `OpenDefaultComponent` and `OpenComponent` functions are described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

The chapter "Transport Components" in this book describes how your application can use transport components.

MTDirectorGetTransport

The `MTDirectorGetTransport` function gets the transport component instance for the stream director you specify.

```
pascal ComponentResult MTDirectorGetTransport
            (MTDirectorComponent sdc,
            MTTransportComponent *tc);
```

`sdc` The stream director component you are using.

`tc` A pointer to a component instance. The function returns the transport instance attached to the stream director that you specify.

DESCRIPTION

You call the `MTDirectorGetTransport` function to get the transport component instance attached to a stream director.

A stream director can have only one transport component attached at any given time. It uses its attached transport component to exchange media data and control information across a connection.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You attach a transport component to a stream director with the `MTDirectorSetTransport` function, which is described page 5-45.

MTDirectorSetMediaComponent

The `MTDirectorSetMediaComponent` function attaches a media component to a stream director that you specify.

```
pascal ComponentResult MTDirectorSetMediaComponent
            (MTDirectorComponent sdc,
            ComponentInstance media);
```

`sdc` The stream director component you are using.

`media` The media component instance to attach to the stream director.

DESCRIPTION

Media components are components capable of generating media data, such as a QuickTime sequence grabber component. You always attach a media component to a source stream director. You do not attach a media component to a sink stream director.

The component type of the media component that you are attaching must match the component subtype of the stream director you are using.

After calling this function, you must call the `MTDirectorSetGWorld` function to set the current graphics world, even if you already set the graphics world before you called `MTDirectorSetMediaComponent`.

SPECIAL CONSIDERATIONS

The source stream director provided by QuickTime Conferencing 1.0 supports QuickTime sequence grabbers as media components.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTDirectorSetGWorld` function is described on page 5-62.

MTDirectorGetMediaComponent

The `MTDirectorGetMediaComponent` function gets information about the media component used by a stream director that you specify.

```
pascal ComponentResult MTDirectorGetMediaComponent
            (MTDirectorComponent sdc,
            ComponentInstance *media, OSType *type,
            OSType *subtype);
```

`sdc`	The stream director component you are using.
`media`	A pointer to a component instance. The function returns the media component instance attached to the stream director that you specify.
`type`	A pointer to a component type. The function returns the component type of the attached media component instance.
`subtype`	A pointer to a component subtype. The function returns the component subtype of the attached media component instance.

DESCRIPTION

You call the `MTDirectorGetMediaComponent` function to get a stream director's media component instance and its type and subtype.

Media components are components, such as a QuickTime sequence grabber component, that are capable of generating media data.

SPECIAL CONSIDERATIONS

The source stream director provided by QuickTime Conferencing 1.0 supports QuickTime sequence grabbers as media components.

RESULT CODES

`noErr`	0	No error

Installing Callback Functions

Applications and other components can use the functions described in this section to get and set negotiation callback functions and recording callback functions.

During a negotiation, a stream director calls the negotiation callback function to keep you informed of its progress.

A stream director calls the recording callback function whenever it has media data, typically so that a recorder component can store the data into a file.

MTDirectorSetRecordProc

The `MTDirectorSetRecordProc` function installs a function for recording media data for a stream director that you specify.

```
pascal ComponentResult MTDirectorSetRecordProc
            (MTDirectorComponent sdc, MTMediaUPP proc,
            long refCon);
```

`sdc`	The stream director component you are using.
`proc`	A universal procedure pointer to your recording function. To remove a recording function that you previously installed, set this parameter to `nil`.
`refCon`	Reserved for your use. The stream director passes this value to your recording function each time it calls the function.

DESCRIPTION

You call the `MTDirectorSetRecordProc` function to install a recording function for a stream director. Typically, a recorder component calls `MTDirectorSetRecordProc`.

Once a recording function is installed, a stream director calls it every time that media data is available for any enabled stream that it manages.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The function declaration for a recording function is described in the chapter “Recorder Components” in this book.

See the section "Managing Streams" beginning on page 5-21 for a discussion of the enable status of streams.

Universal procedure pointers are described in *Inside Macintosh: PowerPC System Software*.

You can enable or disable a stream with the `MTDirectorStreamEnable` function, described on page 5-58.

MTDirectorGetRecordProc

The `MTDirectorGetRecordProc` function retrieves a universal procedure pointer to the installed recording function and the associated reference constant for a stream director that you specify.

```
pascal ComponentResult MTDirectorGetRecordProc
            (MTDirectorComponent sdc, MTMediaUPP *proc,
            long *refCon);
```

`sdc`	The stream director component you are using.
`proc`	The address of a universal procedure pointer. The function returns the universal procedure pointer to the recording function currently installed. If no recording function is installed, `MTDirectorGetRecordProc` sets the pointer to `nil`.
`refCon`	A pointer to a reference constant. The `MTDirectorGetRecordProc` function returns the reference constant used by the recording function.

DESCRIPTION

You can call this function to find out if a recording operation is under way—whenever a recording function is installed and media data is available, recording takes place.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can call the `MTDirectorSetRecordProc` function, described on page 5-49, to install a function for recording media data.

Universal procedure pointers are described in *Inside Macintosh: PowerPC System Software*.

MTDirectorSetNegotiateProc

The `MTDirectorSetNegotiateProc` function installs a negotiation function for a stream director that you specify.

```
pascal ComponentResult MTDirectorSetNegotiateProc
            (MTDirectorComponent sdc,
            MTNegotiateUPP negotiate, long refCon);
```

`sdc`	The stream director component you are using.
`negotiate`	A universal procedure pointer to your negotiation callback function. To remove a negotiation function that you previously installed, set this parameter to `nil`.
`refCon`	Reserved for your use. The stream director passes this value to your negotiation function each time it calls the function.

DESCRIPTION

You call the `MTDirectorSetNegotiateProc` function to install a negotiation function. A stream director calls your function at certain times during its negotiation with the stream director at the remote connection end.

SPECIAL CONSIDERATIONS

Stream controller components install a negotiation function after they are called with the `MTControllerSetStreamDirector` function. A stream director accepts only one installed negotiation function at any given time. Calling the `MTDirectorSetNegotiateProc` function when a negotiation function is already installed replaces the existing function with the new one.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The function declaration for your negotiation function is on page 5-86.

See the section “Monitoring Stream Format Negotiations” beginning on page 5-17 for a discussion on why you need to install a negotiation function and what actions you should take when it is called.

Universal procedure pointers are described in *Inside Macintosh: PowerPC System Software*.

The `MTControllerSetStreamDirector` function is described in the chapter “Stream Controller Components” in this book.

If you are creating a stream director component, see the section “Negotiating Stream Formats” beginning on page 5-37 for a discussion of the negotiation process itself.

MTDirectorGetNegotiateProc

The `MTDirectorGetNegotiateProc` function gets information about the installed negotiation function of a stream director.

```
pascal ComponentResult MTDirectorGetNegotiateProc
            (MTDirectorComponent sdc,
            MTNegotiateUPP *negotiate, long *refCon);
```

`sdc`	The stream director component you are using.
`negotiate`	A pointer to a universal procedure pointer. The function returns the universal procedure pointer to the negotiation callback function installed for this stream director.
`refCon`	A pointer to a reference constant. The function returns the reference constant used by the negotiation function.

DESCRIPTION

You call the `MTDirectorGetNegotiateProc` function to get the negotiation function currently installed for a stream director.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The function declaration for the negotiation function is on page 5-86.

You install a negotiation function by calling the `MTDirectorSetNegotiateProc` function, described on page 5-51.

See the section "Monitoring Stream Format Negotiations" beginning on page 5-17 for a discussion of why you need to install a negotiation function and what actions you should take when it is called.

Universal procedure pointers are described in *Inside Macintosh: PowerPC System Software*.

If you are creating a stream director component, see the section "Negotiating Stream Formats" beginning on page 5-37 for a discussion of the negotiation process itself.

Managing a Stream Director

You use the functions described in this section to

- inform a stream director of a change in connection status
- create a link between a source and a sink stream director that use the same transport-level connection on the local computer

MTDirectorConnected

The `MTDirectorConnected` function informs a stream director that you either set up a connection to a remote connection end or are about to terminate the existing connection.

```
pascal ComponentResult MTDirectorConnected
            (MTDirectorComponent sdc, Boolean connected);
```

`sdc` The stream director component you are using.

`connected` A Boolean value indicating whether a connection is active or not. Set this parameter to `true` to indicate that a connection is set up. Set it to `false` to indicate that a connection is about to end.

DESCRIPTION

You call the `MTDirectorConnected` function to inform a stream director about the status of its transport-level connection so that it can take appropriate action.

After you establish a transport-level connection (that is, after your call is answered or after you receive a call) and after you set your graphics world, you need to call `MTDirectorConnected`, setting `connected` to `true`. This tells a stream director that it can use its attached transport component to talk to the remote stream director. The local stream director then negotiates stream formats with the remote side, allocates the memory it needs, and performs any other operations necessary to prepare for the exchange of media data over the connection.

Before you terminate a transport-level connection, you need to call `MTDirectorConnected`, setting `connected` to `false`. This tells a stream director that its connection to the remote stream director is no longer valid. It stops exchanging media data. A sink stream director closes the stream player components it opened.

RESULT CODES

`noErr`	0	No error

SEE ALSO

For information about how your application sets up and terminates transport-level connections, see the chapter "Transport Components" in this book.

You set a graphics world by calling the `MTDirectorSetGWorld` function, described on page 5-62.

You attach a transport component to a stream director by calling the `MTDirectorSetTransport` function, described on page 5-45.

MTDirectorJoin

The `MTDirectorJoin` function joins two stream director components.

```
pascal ComponentResult MTDirectorJoin
            (MTDirectorComponent sdc,
            MTDirectorComponent other);
```

`sdc` The stream director component you are using.

`other` The stream director component instance to join to the one specified in the `sdc` parameter.

DESCRIPTION

You call the `MTDirectorJoin` function to join a sink stream director and a source stream director on the local computer. You can specify either stream director in the parameters `sdc` or `other`—the order does not matter.

You need to join stream directors when the source and sink stream directors use the same transport-level connection. For example, the source and sink stream directors typically share a transport component when you make a full-duplex audio-visual connection. You join them so that each stream director has access to the other's information about the connection. A stream director may need this information for certain types of operations in full-duplex audio-visual connections, such as managing the sound level for the connection.

Do not join stream directors when they use different transport-level connections. For example, if your user is sending a multicast transmission over one connection and watching a multicast transmission over another connection, your application's source and sink stream directors are using different transport components. If you join those two stream directors, the results are unpredictable.

You should join stream directors before the source stream director begins negotiations. A source stream director begins negotiations after you call the `MTDirectorConnected` function to tell it you've established a connection.

You can separate stream directors you have joined by calling `MTDirectorJoin` and setting the first parameter to either component and the second parameter to `nil`. Stream director components are automatically separated when you close one of the components.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTDirectorConnected` function is described on page 5-53.

Managing Streams

You can use the functions described in this section to

- get certain characteristics of a stream
- find out the number of streams managed by a stream director
- get the stream ID and stream type of each stream managed by a stream director
- get the stream description of a stream that you specify
- enable or disable a specific stream or all streams managed by a stream director
- find out whether a stream is enabled or disabled
- inform a stream director about changes to stream data formats

MTDirectorGetStreamInfo

The `MTDirectorGetStreamInfo` function tells you about certain characteristics of a stream that you specify.

```
pascal ComponentResult MTDirectorGetStreamInfo
            (MTDirectorComponent sdc, MTStreamID stream,
               MTStreamInfo *info);
```

`sdc` The stream director component you are using.

`stream` The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function.

`info` A pointer to a set of bit flags that describe certain characteristics of the stream. You allocate memory for the flags and provide a pointer to it; the function sets the flags. The following flags are defined:

`streamHasBounds`
Indicates whether a stream contains visual data. If the function sets this flag to 1, the stream contains visual data. In that case, you can call the stream director functions that pertain to visual data for this stream.

`streamHasVolume`
Indicates whether a stream contains audio data. If the function sets this flag to 1, the stream contains audio data. In that case, you can call the stream director functions that pertain to audio data for this stream.

`streamHasDiscreteSamples`
Indicates whether this stream's data is organized into discrete samples (such as a video frame). If the function sets this flag to 1, the media data is organized into discrete samples, meaning that no fixed sample rate applies to the data. If the function sets this flag to 0, the media data has a fixed sample rate.

`streamIsInterruptCapable`
Indicates whether a stream's data can be played at interrupt time. If the function sets this flag to 1, the data can be played at interrupt time.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

MTDirectorGetNumberOfStreams

The `MTDirectorGetNumberOfStreams` function tells you the number of streams managed by a stream director component.

```
pascal ComponentResult MTDirectorGetNumberOfStreams
            (MTDirectorComponent sdc, MTStreamCount *numStreams);
```

`sdc` The stream director component you are using.

`numStreams`
A pointer to the number of streams. The function returns the number of streams managed by the stream director you specify.

RESULT CODES

`noErr`	0	No error

MTDirectorGetNextStream

The `MTDirectorGetNextStream` function gets the stream identifier and stream type of one stream in a list of streams.

```
pascal ComponentResult MTDirectorGetNextStream
            (MTDirectorComponent sdc, MTStreamID *stream,
            MTStreamType *type);
```

`sdc` The stream director component you are using.

`stream` A pointer to a stream identifier. You set this parameter to tell the function where to start in the list of streams. To get the stream ID of the first stream in the list, set the stream identifier to `0`. If you set it to the stream ID of a known stream, the function returns the ID of the next stream in the list.

`type` A pointer to a stream type, which indicates the type of media data carried by a stream. The function returns the stream type of the stream it specifies in the `stream` parameter.

DESCRIPTION

The `MTDirectorGetNextStream` function returns the stream ID and the stream type of one stream each time you call it.

The stream director maintains a list of its active streams. An active stream is one whose formats have been successfully negotiated; it may be enabled or disabled.

You can use the `MTDirectorGetNextStream` function to get the stream ID and stream type of each currently active stream. To start at the beginning of the list, set the stream ID to `0` when you call `MTDirectorGetNextStream`. The function returns the stream ID and stream type of the first stream in the list. To cycle through the entire list, call the function repeatedly, each time passing it the stream ID that the function returned in the previous call.

When you pass `MTDirectorGetNextStream` the stream ID of the last stream on the list, the function returns a stream ID of 0.

RESULT CODES

`noErr`	0	No error

SEE ALSO

To determine if a given stream is enabled or disabled, call the `MTDirectorIsStreamEnabled` function, described on page 5-59.

The stream types `VideoMediaType` and `SoundMediaType` are described in *Inside Macintosh: QuickTime*.

MTDirectorGetStreamDescription

The `MTDirectorGetStreamDescription` function retrieves the stream description for a stream that you specify.

```
pascal ComponentResult MTDirectorGetStreamDescription
            (MTDirectorComponent sdc, MTStreamID stream,
            MTStreamDescriptionHandle streamDesc);
```

`sdc`	The stream director component you are using.
`stream`	The stream ID of the stream whose description you want. You can get a stream ID from the `MTDirectorGetNextStream` function.
`streamDesc`	A handle to a stream description structure. You allocate the memory for the stream description structure and passes the handle to the function. The function returns information in the structure and resizes the handle, based on the size of the stream description.

DESCRIPTION

You call the `MTDirectorGetStreamDescription` function to retrieve the stream description for a stream. A stream description contains both QuickTime Conferencing and QuickTime information.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

The `MTStreamDescription` stream description structure is described on page 5-42.

MTDirectorStreamEnable

The `MTDirectorStreamEnable` function enables or disables a stream.

```
pascal ComponentResult MTDirectorStreamEnable
            (MTDirectorComponent sdc, MTStreamID stream,
            Boolean enableMode);
```

`sdc`	The stream director component you are using.
`stream`	The stream ID of the stream you want to enable or disable. You can get a stream ID from the `MTDirectorGetNextStream` function. To enable or disable all currently active streams, set this parameter to the constant `kMTAllStreams`.
`enableMode`	A Boolean value indicating whether you want to enable or disable a stream. Set this parameter to `true` to enable a stream. Set it to `false` to disable a stream.

DESCRIPTION

You call the `MTDirectorStreamEnable` function to enable or disable either a single stream that you specify or all streams managed by a given stream director component. You can enable or disable both incoming and outgoing streams.

Once you disable an outgoing stream, a source stream director stops sending data for that stream to the remote connection end. The stream director resumes sending data for that stream when you enable the stream again.

When you disable an incoming stream, a sink stream director stops processing the data it receives from the remote connection end until you enable the stream again.

In addition to the result codes listed below, the `MTDirectorStreamEnable` function can return errors from its attached media component, such as a QuickTime sequence grabber, and from its attached transport component.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

You can find out the enable status of a stream by calling the `MTDirectorIsStreamEnabled` function, described next.

MTDirectorIsStreamEnabled

The `MTDirectorIsStreamEnabled` function tells you whether a stream is enabled or disabled.

```
pascal ComponentResult MTDirectorIsStreamEnabled
            (MTDirectorComponent sdc, MTStreamID stream,
            Boolean *enableMode);
```

`sdc` The stream director component you are using.

`stream` The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to know the enable status of all the streams managed by this stream director, specify the constant `kMTAllStreams`.

`enableMode` A pointer to a Boolean value that indicates whether the stream is enabled or disabled. The function sets this parameter to `true` if the stream is enabled. It sets it to `false` if the stream is disabled.

DESCRIPTION

You call the `MTDirectorIsStreamEnabled` function to determine if an individual stream or a group of streams is enabled or disabled.

The function always returns the correct enable status of an individual stream.

When you ask for the enable status of all streams managed by a stream director (that is, when you set `stream` to `kMTAllStreams`), the function returns the status setting that you provided the last time you called the `MTDirectorStreamEnable` function to set the status for all streams.

If you set the enable status for all streams as a group and then change the status of one or more of them individually, the status that the `MTDirectorIsStreamEnabled` function returns is inconsistent with the current status of one or more of the streams.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

You set the enable status of a stream by calling the `MTDirectorStreamEnable` function, described on page 5-58.

The section "Managing Streams" beginning on page 5-21 discusses the use of the `MTDirectorStreamEnable` function.

MTDirectorChangedStreamFormats

The `MTDirectorChangedStreamFormats` function informs a source stream director that outgoing stream formats are about to change or have changed.

```
pascal ComponentResult MTDirectorChangedStreamFormats
            (MTDirectorComponent sdc,
            Boolean finished);
```

`sdc` The stream director component you are using.

`finished` A Boolean value indicating whether the stream formats are about to change or have changed. Set this parameter to `false` to indicate that stream formats are about to change. Set it to `true` to indicate the stream formats have changed.

DESCRIPTION

You call the `MTDirectorChangedStreamFormats` function to inform a source stream director that you are about to change or have changed the format of one or more outgoing streams.

Prior to changing stream formats, call this function and set the `finished` parameter to `false`. Then call other functions to change the formats. After changing the stream formats, call `MTDirectorChangedStreamFormats` and set `finished` to `true` to tell the stream director that formats have been changed.

Note that you should not call the `MTDirectorChangedStreamFormats` function when you change local display characteristics with the `MTDirectorSetGWorld` function during an active connection. The stream looks exactly the same to the remote connection end. Therefore, it is not necessary to advise the remote connection end about these stream format changes.

This function can return result codes from a media component.

SPECIAL CONSIDERATIONS

Be aware that after you call the `MTDirectorChangedStreamFormats` function with the `finished` parameter set to `true`, the stream director renegotiates stream formats for all streams. As a result, it obtains a new stream ID for each stream it manages. To get the new stream IDs, call the `MTDirectorGetNextStream` function.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

For an example of changing stream formats, see the section "Changing Stream Formats" beginning on page 5-19.

Getting and Setting Characteristics of Visual Stream Data

You use the functions described in this section to get and set certain characteristics of streams that contain visual data. For a given stream, you can

- set its graphics world
- get and set its source rectangle—the base rectangle to which the transformation matrix and the clipping region are applied
- get and set the transformation matrix used for scaling, translating, and rotating images
- get and set the clipping region used to limit the area in which media data is displayed
- get the box—the rectangle that encloses the boundary rectangles of all the streams managed by a single stream director

- get and set the frame rate and the maximum frame rate

MTDirectorSetGWorld

The `MTDirectorSetGWorld` function sets the graphics port and graphics device for a stream that you specify.

```
pascal ComponentResult MTDirectorSetGWorld
            (MTDirectorComponent sdc, MTStreamID stream,
            CGrafPtr gp, GDHandle gd);
```

`sdc`	The stream director component you are using.
`stream`	The stream identifier of the stream whose graphics world you want to set. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to set the graphics world of all the streams managed by this stream director, specify the constant `kMTAllStreams`.
`gp`	A pointer to a `CGrafPort` structure that specifies the graphics port in which to display the stream's visual data.
`gd`	A handle to a `GDevice` structure that specifies the graphics device on which to display the stream's visual data.

DESCRIPTION

You call the `MTDirectorSetGWorld` function to set the graphics world for a stream that contains visual data. The stream director passes the information to the appropriate stream player or media component.

When working with a source stream director, you must call the `MTDirectorSetGWorld` function for all streams that contain visual data after calling the `MTDirectorSetMediaComponent` function. You can also call it before you call `MTDirectorSetMediaComponent`, but it's not required.

When working with a sink stream director, you must call the `MTDirectorSetGWorld` function for all streams that contain visual data before calling the `MTDirectorConnected` function to tell the sink stream director that a connection to a remote source stream director is ready for use.

Typically, you use a color graphics port, but the function also accepts a black-and-white graphics port.

SPECIAL CONSIDERATIONS

When you change the local display characteristics with the `MTDirectorSetGWorld` function during an active connection, you should not call the `MTDirectorChangedStreamFormats` function. Because the stream looks exactly the

same to the remote connection end, you don't need to advise the remote connection end about local stream format changes.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

The `MTDirectorSetMediaComponent` function attaches a media component to a source stream director. It is described on page 5-47.

The `MTDirectorConnected` function is described on page 5-53.

The `MTDirectorChangedStreamFormats` function is described on page 5-60.

The `MTDirectorGetNextStream` function is described on page 5-56.

For further information about `GDevice` and `CGrafPort` structures, see *Inside Macintosh: Imaging With QuickDraw*.

MTDirectorSetRect

The `MTDirectorSetRect` function sets the source rectangle of a stream.

```
pascal ComponentResult MTDirectorSetRect
            (MTDirectorComponent sdc,
            MTStreamID stream, Rect *r);
```

`sdc`	The stream director component you are using.
`stream`	The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to set the rectangle of all the streams managed by this stream director, specify the constant `kMTAllStreams`.
`r`	A pointer to the rectangle that you want to set for the stream.

DESCRIPTION

You call the `MTDirectorSetRect` function to set the source rectangle of a stream that contains visual data.

The source rectangle is a rectangle smaller than or equal to the boundary rectangle of a stream. (A stream's boundary rectangle is set by the component that generates the stream data.) The source rectangle is located within or congruent with the stream's boundary rectangle. Typically, the source rectangle is equal to the boundary rectangle.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

You can use the `MTPlayerGetDimensions` function, described in the chapter "Stream Player Components" in this book, to get the boundary rectangle of an incoming stream.

If the stream director's media component is a QuickTime sequence grabber, you can use the `SGGetSrcVideoBounds` function, described in *Inside Macintosh: QuickTime Components*, to determine the boundary rectangle for an outgoing stream.

You can retrieve the source rectangle of a stream by calling the `MTDirectorGetRect` function, described next.

MTDirectorGetRect

The `MTDirectorGetRect` function retrieves the source rectangle of a stream.

```
pascal ComponentResult MTDirectorGetRect
            (MTDirectorComponent sdc,
            MTStreamID stream, Rect *r);
```

`sdc`	The stream director component you are using.
`stream`	The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function.
`r`	A pointer to a rectangle structure. The function returns the source rectangle for the stream.

DESCRIPTION

You call the `MTDirectorGetRect` function to get the source rectangle of a stream that contains visual data.

The source rectangle is a rectangle smaller than or equal to the boundary rectangle of a stream. (A stream's boundary rectangle is set by the component that generates the stream data.) The source rectangle is located within or congruent with the stream's boundary rectangle. Typically, the source rectangle is equal to the boundary rectangle.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

You can use the `MTPlayerGetDimensions` function, described in the chapter "Stream Player Components" in this book, to find out the boundary rectangle of an incoming stream.

If the stream director's media component is a QuickTime sequence grabber, you can use the `SGGetSrcVideoBounds` function, described in *Inside Macintosh: QuickTime Components*, to determine the boundary rectangle for an outgoing stream.

You can set the source rectangle of a stream by calling the `MTDirectorSetRect` function, described on page 5-63.

MTDirectorSetMatrix

The `MTDirectorSetMatrix` function sets the display transformation matrix for a stream that you specify.

```
pascal ComponentResult MTDirectorSetMatrix
            (MTDirectorComponent sdc, MTStreamID stream,
            const MatrixRecord *m);
```

`sdc` The stream director component you are using.

`stream` The stream identifier of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to set the matrix of all the streams managed by this stream director, specify the constant `kMTAllStreams`.

`m` A pointer to a fully specified matrix structure for the stream. You provide the matrix structure. The function does not modify the matrix.

DESCRIPTION

You call the `MTDirectorSetMatrix` function to set the display transformation matrix for a stream that contains visual data. A transformation matrix defines how to map points from one coordinate space into another. A stream director uses it to scale, translate, and rotate images.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

Inside Macintosh: QuickTime discusses the use of transformation matrices.

MTDirectorGetMatrix

The `MTDirectorGetMatrix` function retrieves the display transformation matrix of a stream that contains visual data.

```
pascal ComponentResult MTDirectorGetMatrix
            (MTDirectorComponent sdc, MTStreamID stream,
            MatrixRecord *m);
```

`sdc`	The stream director component you are using.
`stream`	The stream identifier of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function.
`m`	A pointer to a matrix structure. You allocate the memory for the matrix structure. The function sets the structure fields to the stream's current matrix values.

DESCRIPTION

You call the `MTDirectorGetMatrix` function to retrieve the display transformation matrix of a stream that contains visual data. A transformation matrix defines how to map points from one coordinate space into another. A stream director uses it to scale, translate, and rotate images.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

You can set a stream's matrix values by calling the `MTDirectorSetMatrix` function, described on page 5-65.

See *Inside Macintosh: QuickTime* for a discussion of transformation matrices.

The `MTDirectorGetNextStream` function is described on page 5-56.

MTDirectorSetClip

The `MTDirectorSetClip` function sets the clipping region for a stream containing visual data.

```
pascal ComponentResult MTDirectorSetClip
            (MTDirectorComponent sdc,
            MTStreamID stream, RgnHandle theClip);
```

`sdc` The stream director component you are using.

`stream` The stream identifier of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to set the clipping region of all the streams managed by this stream director, specify the constant `kMTAllStreams`.

`theClip` A handle to the clipping region for the stream. You are responsible for disposing of the handle.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

Clipping regions and the `Region` structure are discussed in *Inside Macintosh: Imaging With QuickDraw*.

You can retrieve a stream's clipping region by calling the `MTDirectorGetClip` function, described next.

MTDirectorGetClip

The `MTDirectorGetClip` function retrieves the clipping region for a stream that contains visual data.

```
pascal ComponentResult MTDirectorGetClip
            (MTDirectorComponent sdc,
            MTStreamID stream, RgnHandle *theClip);
```

`sdc`	The stream director component you are using.
`stream`	The stream identifier of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function.
`theClip`	The address of a handle to a `Region` structure. The function allocates the memory for the `Region` structure and returns a handle to it in this parameter. The function sets the structure fields to the stream's current clipping region values. You are responsible for disposing of the `Region` structure. If no clipping region has been set, the function sets this parameter to `nil`.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

Clipping regions and the `Region` structure are discussed in *Inside Macintosh: Imaging With QuickDraw*.

You can set a stream's clipping region by calling the `MTDirectorSetClip` function, described on page 5-67.

MTDirectorGetBox

The `MTDirectorGetBox` function computes the rectangle that encloses all the boundary rectangles for streams displayed by a single stream director.

```
pascal ComponentResult MTDirectorGetBox
            (MTDirectorComponent sdc, Rect *box);
```

`sdc`	The stream director component you are using.
`box`	A pointer to a rectangle structure. The function returns the rectangle that encloses the boundary rectangles for all streams managed by the stream director you specify.

DESCRIPTION

You call the `MTDirectorGetBox` function to get the box—a rectangle that encloses all of the boundary rectangles for the streams managed by a given stream director. The box is in the coordinate system of the display area set by the `MTDirectorSetGWorld` function.

Each stream containing visual data has a boundary rectangle that is set by the media component that generates the data. When you ask for the box, a stream director calculates the rectangle that completely encloses the display area for all its streams.

You should not call this function if you have defined different graphics worlds for different streams because the result is undefined.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

The `MTDirectorSetGWorld` function is described on page 5-62.

MTDirectorSetFrameRate

The `MTDirectorSetFrameRate` function sets the frame rate for a stream that you specify.

```
pascal ComponentResult MTDirectorSetFrameRate
            (MTDirectorComponent sdc, MTStreamID stream,
            Fixed frameRate);
```

`sdc`	The stream director component you are using.
`stream`	The stream ID of the stream whose frame rate you want to set. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to set the frame rate of all the streams managed by this stream director, specify the constant `kMTAllStreams`.
`frameRate`	The desired frame rate for the stream, expressed as frames per second. Specify 0 to request the best available frame rate.

DESCRIPTION

You call the `MTDirectorSetFrameRate` function to set the frame rate of a stream. Frame rates apply only to streams whose media data is organized into discrete samples. You can determine if a stream's data is organized into discrete samples by examining the information returned by the `MTDirectorGetStreamInfo` function.

SPECIAL CONSIDERATIONS

You call the `MTDirectorSetFrameRate` function for a source stream director. This function has no meaning to a sink stream director.

A flow control component can dynamically adjust the frame rate up to the maximum frame rate to achieve the best results. Typically, on shared networks, the frame rate varies dynamically, with the variation in frame rate being proportional to the variation in traffic on the network. This function sets the desired frame rate; actual performance may be higher or lower.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

You can set a maximum frame rate by calling the `MTDirectorSetMaxFrameRate` function, described on page 5-71.

The `MTDirectorGetStreamInfo` function is described on page 5-55.

See page 5-26 for a brief discussion of frame rates.

MTDirectorGetFrameRate

The `MTDirectorGetFrameRate` function gets the frame rate for a stream.

```
pascal ComponentResult MTDirectorGetFrameRate
            (MTDirectorComponent sdc, MTStreamID stream,
            Fixed *frameRate);
```

`sdc`	The stream director component you are using.
`stream`	The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function.
`frameRate`	A pointer to the current frame rate for the stream.

DESCRIPTION

You call the `MTDirectorGetFrameRate` function to get the frame rate of a stream. Frame rates apply only to streams whose media data is organized into discrete samples. You can determine if a stream's data is organized into discrete samples by examining the information returned by the `MTDirectorGetStreamInfo` function.

Typically, on shared networks, the frame rate varies dynamically, with the variation in frame rate being proportional to the variation in traffic on the network.

You call the `MTDirectorGetFrameRate` function for a source stream director. This function has no meaning to a sink stream director.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

The `MTDirectorGetStreamInfo` function is described on page 5-55.

To set the frame rate of a stream, call the `MTDirectorSetFrameRate` function, described on page 5-69.

See page 5-26 for a brief discussion of frame rates.

MTDirectorSetMaxFrameRate

The `MTDirectorSetMaxFrameRate` function sets the maximum frame rate for a stream.

```
pascal ComponentResult MTDirectorSetMaxFrameRate
            (MTDirectorComponent sdc, MTStreamID stream,
            Fixed frameRate);
```

`sdc` The stream director component you are using.

`stream` The stream ID of the stream whose maximum frame rate you want to set. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to set the maximum frame rate of all the streams managed by this stream director, specify the constant `kMTAllStreams`.

`frameRate` The desired maximum frame rate for the stream.

DESCRIPTION

You call the `MTDirectorSetMaxFrameRate` function to set the maximum frame rate of a stream. Frame rates apply only to streams whose media data is organized into discrete samples. You can determine if a stream's data is organized into discrete samples by examining the information returned by the `MTDirectorGetStreamInfo` function.

The actual frame rate may drop below the maximum frame rate that you set, but it won't exceed the maximum.

SPECIAL CONSIDERATIONS

You call the `MTDirectorSetMaxFrameRate` function for a source stream director. This function has no meaning to a sink stream director.

A flow control component can dynamically adjust the frame rate up to the maximum frame rate to achieve the best results. Typically, on shared networks, the frame rate varies dynamically, with the variation in frame rate being proportional to the variation in traffic on the network.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

The `MTDirectorSetFrameRate` function is described on page 5-69.

The `MTDirectorGetStreamInfo` function is described on page 5-55.

See page 5-26 for a brief discussion of frame rates.

Functions that work with a sequence grabber's settings dialog box are described in *Inside Macintosh: QuickTime Components*.

To find out the maximum frame rate for a stream, you can call the `MTDirectorGetMaxFrameRate` function, described next.

MTDirectorGetMaxFrameRate

The `MTDirectorGetMaxFrameRate` function gets the maximum frame rate for a stream.

```
pascal ComponentResult MTDirectorGetMaxFrameRate
            (MTDirectorComponent sdc, MTStreamID stream,
            Fixed *frameRate);
```

`sdc`	The stream director component you are using.
`stream`	The stream ID of the steam of interest. You can get a stream ID from the `MTDirectorGetNextStream` function.
`frameRate`	A pointer to the current frame rate for the stream.

DESCRIPTION

You call the `MTDirectorGetMaxFrameRate` function to get the maximum frame rate of a stream. Frame rates apply only to streams whose media data is organized into discrete samples. You can determine if a stream's data is organized into discrete samples by examining the information returned by the `MTDirectorGetStreamInfo` function.

Typically, on shared networks, the frame rate varies dynamically, with the variation in frame rate being proportional to the variation in traffic on the network. The frame rate may drop below the maximum, but it won't exceed the maximum.

You call the `MTDirectorGetMaxFrameRate` function for a source stream director. This function has no meaning to a sink stream director.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

The `MTDirectorGetStreamInfo` function is described on page 5-55.

You can set the maximum frame rate of a stream by calling the `MTDirectorSetMaxFrameRate` function, described on page 5-71.

See page 5-26 for a brief discussion of frame rates.

Taking a Picture of Visual Stream Data

You can use the function described in this section to obtain a QuickDraw picture of a stream's visual data.

MTDirectorGetPicture

The `MTDirectorGetPicture` function provides a QuickDraw picture of a stream's visual data.

```
pascal ComponentResult MTDirectorGetPicture
            (MTDirectorComponent sdc, MTStreamID stream,
            PicHandle *p, const Rect *bounds,
            short offscreenDepth, long grabPictFlags);
```

`sdc` The stream director component you are using.

`stream` The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to get a picture of all the streams managed by this stream director, specify the constant `kMTAllStreams`.

`p` The address of a handle to a `Picture` structure. You define the handle and set it to `nil`. The function allocates memory for the `Picture` structure and returns a picture of the stream data. You are responsible for disposing of the memory for the `Picture` structure.

`bounds` A pointer to a `Rect` structure that defines the size and position of the boundary rectangle into which the stream director draws the picture.

`offscreenDepth`
The pixel depth of the offscreen graphics world. The value 0 specifies the best available depth. The function uses this parameter only when you set the `grabPictOffScreen` flag in the `grabPictFlags` parameter to 1. Otherwise, the function ignores this parameter.

`grabPictFlags`
A set of bit flags that you can use to control how the picture is drawn. The following flags are defined:

`grabPictOffScreen`
Set this flag to 0 to draw the picture onscreen. If you set this flag to 1, the stream director draws the picture in an offscreen graphics world. In that case, you use the `offscreenDepth` parameter to specify the pixel depth and the `bounds` parameter to specify the rectangle for the offscreen buffer.

`grabPictIgnoreClip`
Set this flag to 0 to use the existing clipping region for the image. If you set this flag to 1, the stream director ignores the current clipping region for the image.

Set all unused flags to 0.

DESCRIPTION

You call the `MTDirectorGetPicture` function to obtain a picture of a stream that contains visual data. The function draws the picture on the screen or in an offscreen buffer, depending on the flags you provide.

The function allocates the memory for a `Picture` structure, stores the picture in the structure, and returns a handle to it. You are responsible for disposing of the memory.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain visual data
`mtNoMediaComponentErr`	–8000	No currently active media component

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.
The `Picture` and `Rect` structures and offscreen graphics worlds are described in *Inside Macintosh: Imaging With QuickDraw*.

Getting and Setting Characteristics of Audio Stream Data

You use the functions described in this section to get and set certain characteristics of streams that contain audio data. For a given stream, you can

- set and get its volume
- set and get its maximum volume
- set and get its audio input gain
- set and get its sound threshold
- get its sound level

MTDirectorSetVolume

The `MTDirectorSetVolume` function sets the volume for a stream.

```
pascal ComponentResult MTDirectorSetVolume
            (MTDirectorComponent sdc, streamID stream,
            short volume);
```

`sdc` The stream director component you are using.

`stream` The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to set the volume of all the streams managed by this stream director, specify the constant `kMTAllStreams`.

`volume` The volume level that you want to set. Volume is expressed as a 16-bit fixed-point number in the range –1.0 to 1.0. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Positive values indicate sound that ranges from very soft to very loud. A value of 0 indicates no sound. Negative values indicate no sound but preserve the absolute value of the volume setting.

DESCRIPTION

If you call `MTDirectorSetVolume` for an incoming stream, you set the volume for the audio portion of that stream.

If you call it for an outgoing stream, `MTDirectorSetVolume` adjusts the audio playthrough for the stream. For example, if you digitize audio from a television source, you can set the volume of that source as it plays through the internal Macintosh speaker.

The value you pass in the `volume` parameter is a relative measure of loudness. It affects the audible level of the stream that you specify. If you set `volume` to 0, a user hears

nothing from that stream when it is played. Table 5-2 contains examples of selected volume levels and the resulting effect on the audio volume.

Table 5-2 Volume settings and their results

Volume as fixed-point value	Volume as hexadecimal value	Effect on volume
1.0	0x0100	Full volume
0.5	0x0080	Half volume
0.0	0x0000	Silence
–0.5	0xff7f	Silence
–1.0	0xff00	Silence

SPECIAL CONSIDERATIONS

The `MTDirectorSetVolume` function accepts volume values greater than 1.0. Such values overdrive the volume and may result in decreased sound quality.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain audio data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

You can set the maximum volume for an incoming stream with the `MTDirectorSetGain` function, described on page 5-78.

The section "Managing Streams Containing Audio Data" beginning on page 5-26 describes all of the audio characteristics you can manipulate.

You can retrieve the volume for a stream with the `MTDirectorGetVolume` function, described next.

MTDirectorGetVolume

The `MTDirectorGetVolume` function gets the volume of a stream.

```
pascal ComponentResult MTDirectorGetVolume
            (MTDirectorComponent sdc, streamID stream,
            short *volume);
```

`sdc`	The stream director component you are using.
`stream`	The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to get the volume of all the streams managed by this stream director, specify the constant `kMTAllStreams`.
`volume`	A pointer to the volume level of the stream. Volume is expressed as a 16-bit fixed-point number in the range –1.0 to 1.0. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Positive values indicate sound that ranges from very soft to very loud (1.0). A value of 0 indicates no sound. Negative values mean no sound but show the absolute value of the volume setting.

DESCRIPTION

You call the `MTDirectorGetVolume function` to get the volume level for a stream that contains audio data.

The value the function returns in the `volume` parameter is a relative measure of loudness.

The function always returns the correct volume of an individual stream. When you ask for the volume of all streams managed by a stream director (by setting the parameter `stream` to `kMTAllStreams`), the function returns the volume setting that you provided the last time you called the `MTDirectorSetVolume` function to set the volume for all streams. If you set the volume for all streams as a group and then change the volume of one or more of them individually, the volume that the `MTDirectorGetVolume` function returns is inconsistent with the current volume of one or more of the streams.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain audio data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

The section "Managing Streams Containing Audio Data" beginning on page 5-26 describes all of the audio characteristics you can manipulate.

To set the volume level for a stream, you call the `MTDirectorSetVolume` function, described on page 5-75. The function description includes a table of sample volume levels and the effect on volume.

MTDirectorSetGain

The `MTDirectorSetGain` function sets either the audio input gain for an outgoing stream or the maximum volume level for an incoming stream.

```
pascal ComponentResult MTDirectorSetGain
            (MTDirectorComponent sdc, streamID stream,
            short gain);
```

`sdc` The stream director component you are using.

`stream` The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to set the gain or maximum volume of all the streams managed by this stream director, specify the constant `kMTAllStreams`.

`gain` The gain or maximum volume level that you want to set for the stream. Gain or volume is expressed as a 16-bit fixed-point number in the range –1.0 to 1.0. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part.

DESCRIPTION

If you specify a source stream director, `MTDirectorSetGain` sets the gain of the audio input device available to the outgoing stream that you specify. For example, a microphone's ability to pick up sounds increases as the value of the `gain` parameter increases.

If you specify a sink stream director, `MTDirectorSetGain` sets the scaling factor of the audio output device used by the incoming stream that you specify. For example, it you set the `gain` parameter to 1.0, the maximum volume of the audio output device for that stream becomes 1.0. If a user then sets the volume control to the highest possible setting, the actual volume setting becomes 100% of 1.0. If the user sets the volume control to the midpoint, the actual volume setting becomes 50% of 1.0.

You can use the `MTDirectorSetVolume` function to set the volume level below or at the maximum volume you specify with `MTDirectorSetGain`.

Typically, a flow control component calls the `MTDirectorSetGain` function.

SPECIAL CONSIDERATIONS

When you call the `MTDirectorSetVolume` and `MTDirectorSetGain` functions, the stream directors provided by Apple translate the relative value that you provide into a value the Sound Manager can understand. When the Sound Manager plays a sound on the available hardware, it does so within the parameters you previously established.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain audio data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

The section "Managing Streams Containing Audio Data" beginning on page 5-26 describes all of the audio characteristics you can manipulate.

Use the `MTDirectorSetVolume` function, described on page 5-75, to set the user-adjustable volume level.

MTDirectorGetGain

The `MTDirectorGetGain` function gets the audio input gain for an outgoing stream or the maximum volume level for an incoming stream.

```
pascal ComponentResult MTDirectorGetGain
            (MTDirectorComponent sdc, streamID stream,
            short *gain);
```

`sdc`	The stream director component you are using.
`stream`	The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function. If you want to get the gain or maximum volume of all the streams managed by this stream director, specify the constant `kMTAllStreams`.
`gain`	A pointer to the gain or maximum volume level for the stream. Gain or volume is expressed as a 16-bit fixed-point number in the range –1.0 to 1.0. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part.

DESCRIPTION

Typically, a flow control component or a stream controller component calls the `MTDirectorGetGain` function.

If you specify an outgoing stream, `MTDirectorGetGain` provides the gain of the audio input device available to that stream.

If you specify an incoming stream, `MTDirectorGetGain` returns the maximum volume of the audio output device used by that stream. (You can call the `MTDirectorGetVolume` function to get the current volume level, which is always at the maximum volume level or below it.)

The function always returns the correct gain or maximum volume of an individual stream. When you ask for the gain or maximum volume of all streams managed by a stream director (by setting `stream` to `kMTAllStreams`), the function returns the setting that you provided the last time you called the `MTDirectorSetGain` function to set it for all streams.

If you set the gain or maximum volume for all streams as a group and then change the setting of one or more of them individually, the value that the `MTDirectorGetGain` function returns is inconsistent with the actual gain or maximum volume of one or more of the streams.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain audio data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

Use the `MTDirectorGetVolume` function, described on page 5-76, to get the current volume level.

To set the audio input gain or the maximum volume for a stream, call the `MTDirectorSetGain` function, described on page 5-78.

MTDirectorSetSoundThreshold

The `MTDirectorSetSoundThreshold` function sets the sound threshold for an outgoing sound stream.

```
pascal ComponentResult MTDirectorSetSoundThreshold
            (MTDirectorComponent sdc, streamID stream,
            short threshold);
```

`sdc`	The source stream director component you are using.
`stream`	The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function.
`threshold`	The sound threshold you want to set. The threshold level is expressed as a 16-bit fixed-point number in the range –1.0 to 1.0. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part.

DESCRIPTION

A source stream director sends sound data to a remote connection end only when the sound level is greater than the threshold level that you set. For example, suppose you set the threshold value to 0.6. As long as the sound level is at or below 0.6, the stream director does not send the audio data. This conserves network bandwidth by preventing the transmission of background noise. If you do not set a threshold value, the stream director transmits all sound.

This function has no meaning to a sink stream director.

SPECIAL CONSIDERATIONS

This function works only for a sound stream (type `SoundMediaType`). It does not work with streams that contain other types of audio data.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain sound data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

You can find out the current sound level by calling the `MTDirectorGetSoundLevel` function, described on page 5-82.

The section "Managing Streams Containing Audio Data" beginning on page 5-26 describes all of the audio characteristics you can manipulate.

To get the current threshold value, call the `MTDirectorGetSoundThreshold` function, described next.

MTDirectorGetSoundThreshold

The `MTDirectorGetSoundThreshold` function retrieves the sound threshold for an outgoing sound stream.

```
pascal ComponentResult MTDirectorGetSoundThreshold
            (MTDirectorComponent sdc, streamID stream,
            short *threshold);
```

`sdc`	The source stream director component you are using.
`stream`	The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function.

`threshold` A pointer to the sound level threshold. The function returns the threshold as a 16-bit fixed-point number in the range –1.0 to 1.0. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part.

DESCRIPTION

Typically, a flow control component calls this function.

You call the `MTDirectorGetSoundThreshold function` to find out the sound threshold for an outgoing sound stream.

A stream director captures and sends sound data to the remote connection end only when the sound level is at or above the threshold level that you previously set.

This function has no meaning to a sink stream director.

SPECIAL CONSIDERATIONS

This function works only for a sound stream (type `SoundMediaType`). It does not work with streams that contain other types of audio data.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain sound data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

You can set the sound threshold by calling the `MTDirectorSetSoundThreshold function`, described on page 5-80.

The section "Managing Streams Containing Audio Data" beginning on page 5-26 describes all of the audio characteristics you can manipulate.

You can find out the current sound level by calling the `MTDirectorGetSoundLevel function`, described next.

MTDirectorGetSoundLevel

The `MTDirectorGetSoundLevel` function gets the sound level detected by a sound input device for a stream.

```
pascal ComponentResult MTDirectorGetSoundLevel
            (MTDirectorComponent sdc, streamID stream,
            short *meterLevel);
```

`sdc`	The stream director component you are using.
`stream`	The stream ID of the stream of interest. You can get a stream ID from the `MTDirectorGetNextStream` function.
`meterLevel`	A pointer to the sound level for a stream. The function returns the sound level in a 16-bit fixed-point number in the range 0 to 1.0. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part.

DESCRIPTION

You call the `MTDirectorGetSoundLevel` function to get the sound level detected by a sound input device for a sound stream.

The sound level measures the energy level of the sound picked up by an input device such as a microphone. When you call this function, a device driver samples the current signal strength on the sound input device. The signal strength varies with the current noise in the local environment.

SPECIAL CONSIDERATIONS

This function works only for a sound stream (type `SoundMediaType`). It does not work with streams that contain other types of audio data.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream doesn't contain sound data

SEE ALSO

The `MTDirectorGetNextStream` function is described on page 5-56.

Advising of User Events

You can use the functions described in this section to alert a stream director to changes in the application environment, such as those that occur when a user opens or closes a window or changes a characteristic of a window such as its size or location. You advise a stream director of these changes so that it can appropriately manage its streams, its attached transport and sequence grabber components, and its stream players.

MTDirectorChangedWindow

The `MTDirectorChangedWindow` function informs a stream director that a window is about to change or has changed.

```
pascal ComponentResult MTDirectorChangedWindow
            (MTDirectorComponent sdc,
            Boolean changedWindow);
```

`sdc` The stream director component you are using.

`changedWindow`
A Boolean value that indicates whether a window is about to change or has changed. Set this parameter to `false` to indicate that a window is about to change. Set it to `true` to indicate that it has changed.

DESCRIPTION

You call the `MTDirectorChangedWindow` function to inform a stream director that a window is about to change or has changed. The function gives a hint to a stream director that may help it manage its visual display.

For example, if a user opens a new window, selects a different window, drags, resizes, closes, or otherwise changes a window, you should bracket your calls to the Window Manager with calls to `MTDirectorChangedWindow`. The first time you call `MTDirectorChangedWindow`, set the `changedWindow` parameter to `false` to allow the stream director to prepare for change. Then call the appropriate Window Manager functions. The second time you call `MTDirectorChangedWindow`, set `changedWindow` to `true`. This allows the stream director to finish adjusting its display of visual data in response to the user event. For example, a source stream director using a sequence grabber might call the `SGUpdate` function so that update and clipping regions are computed correctly.

This function can return result codes from a sequence grabber component.

RESULT CODES

`noErr`	0	No error

SEE ALSO

Sequence grabber components are discussed in *Inside Macintosh: QuickTime Components*.

MTDirectorPause

The `MTDirectorPause` function tells a stream director to pause or resume its operations on streams that contain visual data.

```
pascal ComponentResult MTDirectorPause
            (MTDirectorComponent sdc, Boolean pauseMode);
```

`sdc` The stream director component you are using.

`pauseMode` A Boolean value that indicates whether a stream director should resume or pause its operations on streams. Set this parameter to `true` to indicate that it should pause its operations. Set it to `false` to indicate that it should resume its operations.

DESCRIPTION

When you pause a stream director, it stops processing visual data—it does not pass incoming data to a stream player or outgoing data to a transport component. Typically, you call this function before starting an operation during which visual data cannot be processed. For example, you should call this function to pause a stream director before responding to a mouse-down event in the menu bar and then again after responding to tell the stream director to resume operations.

Pause operations can be nested. For example, if you call `MTDirectorPause` three times with `pauseMode` set to `true`, you need to call it three times with `pauseMode` set to `false` before the stream director once again processes visual data.

RESULT CODES

`noErr`	0	No error

MTDirectorUpdate

The `MTDirectorUpdate` function tells a stream director to refresh its display.

```
pascal ComponentResult MTDirectorUpdate
         (MTDirectorComponent sdc, RgnHandle updateRgn);
```

`sdc` The stream director component you are using.

`updateRgn` A handle to the update region. The update region defines the part of the window that has been changed. You can obtain this information by examining the appropriate window record. If you set this parameter to `nil`, the stream director uses the window's current visible region.

DESCRIPTION

You call the `MTDirectorUpdate` function to tell a stream director to update a window.

You should call `MTDirectorUpdate` after you receive an update event and before you call the Window Manager's `BeginUpdate` function.

RESULT CODES

`noErr`	0	No error

Application-Defined Function

This section describes the negotiation callback function that you provide to a stream director component. A stream director component calls your negotiation function to keep you informed of the progress of a stream format negotiation with a remote stream director.

MyNegotiateProc

When you call the `MTDirectorSetNegotiateProc` function, you provide a universal procedure pointer to a callback function. A stream director calls your function to inform you about the progress of stream format negotiation. Your negotiation function must support the following interface:

```
pascal ComponentResult MyNegotiateProc
            (MTNegotiateStateType state,
            const MTMessageHeader* message, long refCon);
```

`state` The current state of the negotiation.

`message` A pointer to a structure containing the MovieTalk message that triggered the transition to the current state.

`refCon` The reference constant that you passed in the `refCon` parameter to the `MTDirectorSetNegotiateProc` function. You can use this value for whatever purpose you choose.

DESCRIPTION

A stream director calls your negotiation function during stream format negotiations. It provides you with the current state of the negotiation and the MovieTalk message that triggered the transition to the current state. You can then take any actions that are appropriate for that state.

On the source side, you are interested in knowing when a negotiation has successfully completed or when it fails. On the sink side, besides finding out when a negotiation has successfully completed, you can accept or reject each stream format proposed by the

source side. A sink stream director calls your function each time it receives a stream description message, giving you the opportunity to examine the stream description and to accept or reject the stream. Note that a sink stream director determines if a stream player is available to play the stream before calling your negotiation function. If a stream player is not available, the sink stream director sets the `state` parameter to a negative value.

If you want a negotiation to continue, your function should return the `noErr` result code. If you want a negotiation to fail, return a nonzero result code.

SEE ALSO

You install your negotiation function by calling the `MTDirectorSetNegotiateProc` function, described on page 5-51.

The values that the `state` parameter can take are described with the `MTNegotiateStateType` data type on page 5-43.

See the sections "Monitoring Stream Format Negotiations" beginning on page 5-17 and "Negotiation in Depth" beginning on page 5-29 for discussions of why you need to install a negotiation function and what actions you should take when it is called.

For a description of the negotiation process itself, see "Negotiating Stream Formats" beginning on page 5-37.

The format of MovieTalk messages is described in the chapter "MovieTalk Protocol Messages" in this book.

Universal procedure pointers are described in *Inside Macintosh: PowerPC System Software*.

Summary of Stream Director Components

C Summary

Constants

```
enum {         /* stream director component types */
   kMTSinkStreamDirectorType     = 'sksd',   /* sink stream director */
   kMTSourceStreamDirectorType   = 'srsd'    /* source stream director */
};
```

```
enum {         /* stream director component subtypes */
   kMTGrabberSubType    = SeqGrabComponentType, /* uses sequence grabber */
   kMTPlayerSubType     = kMTPlayerType         /* uses stream player */
};

enum { /* stream director request codes */
   kMTDirectorConnectedSelect             = 1,
   kMTDirectorJoinSelect                  = 2,
   kMTDirectorGetNumberOfStreamsSelect    = 3,
   kMTDirectorGetNextStreamSelect         = 4,
   kMTDirectorStreamEnableSelect          = 5,
   kMTDirectorPauseSelect                 = 6,
   kMTDirectorIsStreamEnabledSelect       = 7,
   kMTDirectorSetNegotiateProcSelect      = 8,
   kMTDirectorGetNegotiateProcSelect      = 9,
   kMTDirectorChangedStreamFormatsSelect  = 10,
   kMTDirectorSetTransportSelect          = 11,
   kMTDirectorGetTransportSelect          = 12,
   kMTDirectorGetStreamDescriptionSelect  = 13,
   kMTDirectorUpdateSelect                = 14,
   kMTDirectorGetStreamInfoSelect         = 15,
   kMTDirectorSetGWorldSelect             = 16,
   kMTDirectorSetMatrixSelect             = 17,
   kMTDirectorGetMatrixSelect             = 18,
   kMTDirectorSetClipSelect               = 19,
   kMTDirectorGetClipSelect               = 20,
   kMTDirectorGetBoxSelect                = 21,
   kMTDirectorSetFrameRateSelect          = 22,
   kMTDirectorGetFrameRateSelect          = 23,
   kMTDirectorSetMaxFrameRateSelect       = 24,
   kMTDirectorGetMaxFrameRateSelect       = 25,
   kMTDirectorSetRectSelect               = 26,
   kMTDirectorGetRectSelect               = 27,
   kMTDirectorGetPictureSelect            = 28,
   kMTDirectorSetVolumeSelect             = 29,
   kMTDirectorGetVolumeSelect             = 30,
   kMTDirectorSetGainSelect               = 31,
   kMTDirectorGetGainSelect               = 32,
   kMTDirectorSetSoundThresholdSelect     = 33,
   kMTDirectorGetSoundThresholdSelect     = 34,
   kMTDirectorGetMediaComponentSelect     = 35,
   kMTDirectorSetMediaComponentSelect     = 36,
   kMTDirectorGetSoundLevelSelect         = 37,
   kMTDirectorChangedWindowSelect         = 38,
```

```
   kMTDirectorSetRecordProcSelect         = 39,
   kMTDirectorGetRecordProcSelect         = 40
};

enum {         /* flags for MTStreamInfo data type */
   streamHasBounds            = 0x00000001,  /* stream has visual data */
   streamHasVolume            = 0x00000002,  /* stream has audio data */
   streamHasDiscreteSamples   = 0x00000004,  /* data organized into discrete
                                                frames */
   streamIsInterruptCapable   = 0x00080000   /* data can be played at
                                                interrupt time */
};

enum {            /* values of MTNegotiateStateType */
   kMTNegotiationCompleted          = 0,  /* negotiation complete state */
   kMTNegotiateOpenChannel          = 1,  /* open channel state */
   kMTNegotiateOpenStream           = 2,  /* open stream state */
   kMTNegotiateAckOpenStream        = 3   /* acknowledge stream state */
};
```

Data Types and Structures

```
typedef ComponentInstance MTDirectorComponent;

typedef UInt32 MTStreamInfo;

struct MTStreamDescription {
   MTStreamType         streamType;
   MTStreamID           streamID;
   MTStreamReference    streamReference;
   TimeScale            streamTimeScale;
   SampleDescription    streamSampleDescription;
};

typedef struct MTStreamDescription MTStreamDescription;

typedef MTStreamDescription **MTStreamDescriptionHandle;

typedef ComponentResult MTNegotiateStateType;
```

Functions

Attaching Other Components to a Stream Director

```
pascal ComponentResult MTDirectorSetTransport
                            (MTDirectorComponent sdc,
                             MTTransportComponent tc);
pascal ComponentResult MTDirectorGetTransport
                            (MTDirectorComponent sdc,
                             MTTransportComponent *tc);
pascal ComponentResult MTDirectorSetMediaComponent
                            (MTDirectorComponent sdc,
                             ComponentInstance media);
pascal ComponentResult MTDirectorGetMediaComponent
                            (MTDirectorComponent sdc,
                             ComponentInstance *media, OSType *type,
                             OSType *subtype);
```

Installing Callback Functions

```
pascal ComponentResult MTDirectorSetRecordProc
                            (MTDirectorComponent sdc,
                             MTMediaUPP proc, long refCon);
pascal ComponentResult MTDirectorGetRecordProc
                            (MTDirectorComponent sdc,
                             MTMediaUPP *proc, long *refCon);
pascal ComponentResult MTDirectorSetNegotiateProc
                            (MTDirectorComponent sdc,
                             MTNegotiateUPP negotiate, long refCon);
pascal ComponentResult MTDirectorGetNegotiateProc
                            (MTDirectorComponent sdc,
                             MTNegotiateUPP *negotiate, long *refCon);
```

Managing a Stream Director

```
pascal ComponentResult MTDirectorConnected
                            (MTDirectorComponent sdc,
                             Boolean connected);
pascal ComponentResult MTDirectorJoin
                            (MTDirectorComponent sdc,
                             MTDirectorComponent other);
```

Managing Streams

```
pascal ComponentResult MTDirectorGetStreamInfo
                            (MTDirectorComponent sdc, MTStreamID stream,
                             MTStreamInfo *info);

pascal ComponentResult MTDirectorGetNumberOfStreams
                             (MTDirectorComponent sdc,
                             MTStreamCount *numStreams);

pascal ComponentResult MTDirectorGetNextStream
                            (MTDirectorComponent sdc,
                             MTStreamID *stream, MTStreamType *type);

pascal ComponentResult MTDirectorGetStreamDescription
                            (MTDirectorComponent sdc,
                             MTStreamID stream,
                             MTStreamDescriptionHandle streamDesc);

pascal ComponentResult MTDirectorStreamEnable
                            (MTDirectorComponent sdc,
                             MTStreamID stream, Boolean enableMode);

pascal ComponentResult MTDirectorIsStreamEnabled
                            (MTDirectorComponent sdc,
                             MTStreamID stream, Boolean *enableMode);

pascal ComponentResult MTDirectorChangedStreamFormats
                            (MTDirectorComponent sdc,
                             Boolean finished);
```

Getting and Setting Characteristics of Visual Stream Data

```
pascal ComponentResult MTDirectorSetGWorld
                            (MTDirectorComponent sdc,
                             MTStreamID stream, GrafPtr gp, GDHandle gd);

pascal ComponentResult MTDirectorSetRect
                            (MTDirectorComponent sdc,
                             MTStreamID stream, Rect *r);

pascal ComponentResult MTDirectorGetRect
                            (MTDirectorComponent sdc,
                             MTStreamID stream, Rect *r);

pascal ComponentResult MTDirectorSetMatrix
                            (MTDirectorComponent sdc,
                             MTStreamID stream, const MatrixRecord *m);

pascal ComponentResult MTDirectorGetMatrix
                            (MTDirectorComponent sdc,
                             MTStreamID stream, MatrixRecord *m);
```

```
pascal ComponentResult MTDirectorSetClip
                            (MTDirectorComponent sdc,
                             MTStreamID stream, RgnHandle theClip);
pascal ComponentResult MTDirectorGetClip
                            (MTDirectorComponent sdc,
                             MTStreamID stream, RgnHandle *theClip);
pascal ComponentResult MTDirectorGetBox
                            (MTDirectorComponent sdc, Rect *box);
pascal ComponentResult MTDirectorSetFrameRate
                            (MTDirectorComponent sdc,
                             MTStreamID stream, Fixed frameRate);
pascal ComponentResult MTDirectorGetFrameRate
                            (MTDirectorComponent sdc,
                             MTStreamID stream, Fixed *frameRate);
pascal ComponentResult MTDirectorSetMaxFrameRate
                            (MTDirectorComponent sdc,
                             MTStreamID stream, Fixed frameRate);
pascal ComponentResult MTDirectorGetMaxFrameRate
                            (MTDirectorComponent sdc,
                             MTStreamID stream, Fixed *frameRate);
```

Taking a Picture of Visual Stream Data

```
pascal ComponentResult MTDirectorGetPicture
                            (MTDirectorComponent sdc,
                             MTStreamID stream, PicHandle *p,
                             const Rect *bounds, short offscreenDepth,
                             long grabPictFlags);
```

Getting and Setting Characteristics of Audio Stream Data

```
pascal ComponentResult MTDirectorSetVolume
                            (MTDirectorComponent sdc,
                             MTStreamID stream, short volume);
pascal ComponentResult MTDirectorGetVolume
                            (MTDirectorComponent sdc,
                             MTStreamID stream, short *volume);
pascal ComponentResult MTDirectorSetGain
                            (MTDirectorComponent sdc, MTStreamID stream,
                             short gain);
pascal ComponentResult MTDirectorGetGain
                            (MTDirectorComponent sdc,
                             MTStreamID stream, short *gain);
```

```
pascal ComponentResult MTDirectorSetSoundThreshold
                            (MTDirectorComponent sdc,
                             MTStreamID stream, short threshold);
pascal ComponentResult MTDirectorGetSoundThreshold
                            (MTDirectorComponent sdc,
                             MTStreamID stream, short *threshold);
pascal ComponentResult MTDirectorGetSoundLevel
                            (MTDirectorComponent sdc,
                             MTStreamID stream, short *meterlevel);
```

Advising of User Events

```
pascal ComponentResult MTDirectorChangedWindow
                            (MTDirectorComponent sdc,
                             Boolean changedWindow);
pascal ComponentResult MTDirectorPause
                            (MTDirectorComponent sdc,
                             Boolean pauseMode);
pascal ComponentResult MTDirectorUpdate
                            (MTDirectorComponent sdc,
                             RgnHandle updateRgn);
```

Application-Defined Function

```
pascal ComponentResult MyNegotiateProc
                            (MTNegotiateStateType state,
                             const MTMessageHeader* message, long refCon);
```

Result Codes

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID or stream data type not valid with this function
`mtNoMediaComponentErr`	–8000	No currently active media component

Stream Player Components

Contents

This chapter describes stream player components. **Stream player components** display and play back streams of media data, such as video, sound, or other types of time-based data. The stream player component API provides a standard interface for playing time-based data, independent of differences in the underlying data. This not only frees a QuickTime Conferencing stream director component from the details of playing media data, but also makes it easily extensible to new media data formats.

Apple provides two stream player components: a video stream player and a sound stream player. The video stream player component decompresses video data and displays it on the screen. The sound stream player component configures the Macintosh sound output system and plays back sound data.

You can create new types of stream player components to support new types of streams (such as those containing text, H.320, or H.221 media data), or to handle visual or audio data in a manner different from the default behavior.

If your application calls a stream player component, you should read this chapter. However, most applications do not call a stream player component directly. Instead, they use a conference component or a stream director component, which manages one or more stream player components on their behalf. The information in this chapter is presented from the perspective of one who calls stream player component functions—typically, a sink stream director component.

If you want to create your own stream player component, you need to read this chapter to learn about the API a stream player needs to support. This chapter assumes you are familiar with streams as they are defined in QuickTime Conferencing. See the chapter "Stream Director Components" in this book for an introduction to streams.

To use or create a stream player component, you need to be familiar with the Component Manager, described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

This chapter begins with a brief introduction to stream player components. Then it discusses how you can

- open, initialize, and close a stream player component
- query the characteristics of an incoming stream
- enable or disable an incoming stream
- set and get characteristics of visual stream data
- set and get characteristics of audio stream data
- play stream data

About Stream Player Components

Stream player components display and play back streams of media data that come from a media stream source, typically a stream director component. The stream player

component API provides a standard interface for playing media data, independent of media type. Each stream player component supports a specific type of media data.

QuickTime 2.0 and QuickTime Conferencing Stream Players

QuickTime 2.0 and QuickTime Conferencing support a common stream player component API. However, some of the fields and functions may have different purposes. In general, this is due to the difference in their media sources. QuickTime Conferencing is designed to work with live networked media sources, such as a remote camera whose output can be digitized. QuickTime works with stored media sources such as QuickTime movies.

In a QuickTime Conferencing environment, a stream director component usually calls a stream player. In QuickTime, a media handler calls a stream player. For example, the QuickTime 2.0 MPEG media handler calls the MPEG stream player component.

This chapter documents QuickTime Conferencing stream player components.

Single-Media and Multiplexed Streams

A stream player component plays back a single stream. It gets information about the stream when its `MTPlayerInit` function is called.

A given stream's data may consist of a single media type, such as sound or video, or it may be some combination of basic media types. An MPEG stream, for example, consists of interleaved visual and audio data. Streams whose data consists of a single media type are referred to as single-media streams. Streams whose data consists of more than one media type are referred to as multiplexed streams.

Apple's QuickTime Conferencing 1.0 stream players each play back a single-media video or sound stream.

Some stream player component functions apply only to the visual or audio characteristics of a stream. When you call such a function, a stream player playing a multiplexed stream interprets the function to apply to the visual or audio portion of the stream. For example, if such a stream player receives an `MTPlayerSetVolume` function call, it adjusts the volume of the audio portion of its multiplexed stream.

Other stream player component functions apply to any type of stream. In these cases, a stream player playing a multiplexed stream interprets the function to apply to the multiplexed stream as a whole. For instance, it returns the stream description of the multiplexed stream when you call the `MTPlayerGetStreamDescription` function. Similarly, the stream player plays a piece of the multiplexed stream in response to the `MTPlayerPlayStream` function.

Component Type and Subtypes

All stream player components have a component type of `'sply'` (defined by the `kMTPlayerType` constant). The component subtype differentiates stream players on the

basis of the type of media data that they support. For example, the video stream player that decompresses video data and manages a screen display area for video playback has the subtype `'vide'` (defined by the constant `kMTVideoSubType`). A sound stream player that uses the computer's sound software and hardware to play sound data has the subtype `'soun'` (defined by the constant `kMTSoundSubType`).

If you create a stream player component that supports a new media type, you must give it a meaningful subtype. For example, a stream player that supports MPEG streams should have the subtype `'mpeg'`. Furthermore, if your stream player plays media data generated by a QuickTime sequence grabber channel component, then the stream player's component subtype must match the sequence grabber channel component subtype.

A Note on Terminology

In this chapter, a stream that contains data that a stream player can display or play back is described as containing visual or audio data. The terms *video* and *sound* refer either to the specific media types defined by QuickTime (`VideoMediaType` and `SoundMediaType`) or to specific stream types (those streams that carry video or sound media data). For example, MIDI and sound streams both contain audio data. These streams are more generally described as *containing audio data*. They are more specifically described as MIDI streams and sound streams—that is, they contain audio data in a specific format.

For the sake of brevity in this chapter, stream player components are often referred to as stream players or players. ◆

Using Stream Player Components

This section shows how you can use stream player component functions to

- open a stream player
- initialize a stream player
- get and set the characteristics of streams
- play stream data

Opening a Stream Player Component

You must open a component before you can use its services. If you are interested in any stream player component of a given subtype and do not need to specify any other characteristics of the component, you can call the Component Manager function `OpenDefaultComponent` as shown here:

```
myStreamPlayerInstance = OpenDefaultComponent
                  (kMTPlayerType, kMTVideoSubType);
```

Set the first parameter to the constant `kMTPlayerType` to specify a stream player component (value is `'sply'`). You set the second parameter to the stream player subtype that you want to open. For example, if you want to open a video stream player, set it to the constant `kMTVideoSubType`. The function returns a reference to your connection to the component (the stream player component instance).

If you want to open a specific stream player component, call the Component Manager `OpenComponent` function.

When you are done using a component, close it by calling the `CloseComponent` function. The `OpenDefaultComponent`, `OpenComponent`, and `CloseComponent` functions are described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Initializing a Stream Player Component

After opening the stream player and before calling any other stream player component function, you must call the `MTPlayerInit` function (page 6-13). The stream description that you provide to the function tells the stream player all it needs to know about the stream that it will play.

When you initialize a stream player component, you set the stream for the component for the duration of your connection to that component—you cannot change the stream that the component plays. If you want the component to play a different stream, you must first close the stream player, reopen it, and then call the `MTPlayerInit` function, passing it the description of the new stream.

Enabling and Disabling Streams

When its stream is enabled, a stream player plays the stream data. When its stream is disabled, a stream player does not play the data, even if it continues to receive data. You can enable or disable a stream by calling the `MTPlayerEnable` function (page 6-17), like this:

```
enableMode = true;
myErr = MTPlayerEnable(myStreamPlayerInstance, enableMode);
```

You enable a stream by setting the `enableMode` parameter to `true`. To disable the stream, set the `enableMode` parameter to `false`.

To determine the status of the stream, call the function `MTPlayerIsEnabled` (page 6-18), as shown here:

```
myErr = MTPlayerIsEnabled(myStreamPlayerInstance,&enableMode);
```

If the stream is enabled, the function sets `enableMode` to `true`; if the stream is disabled, the function sets `enableMode` to `false`.

The enable status setting applies to a stream as a whole. For a multiplexed stream, this means that all the types of media data it carries are either enabled or disabled simultaneously. A stream player does not discard the video part of an MPEG stream, for example, while continuing to play the audio part. Instead, the stream player either plays both the audio and video data or plays neither, depending on whether the stream is enabled or disabled.

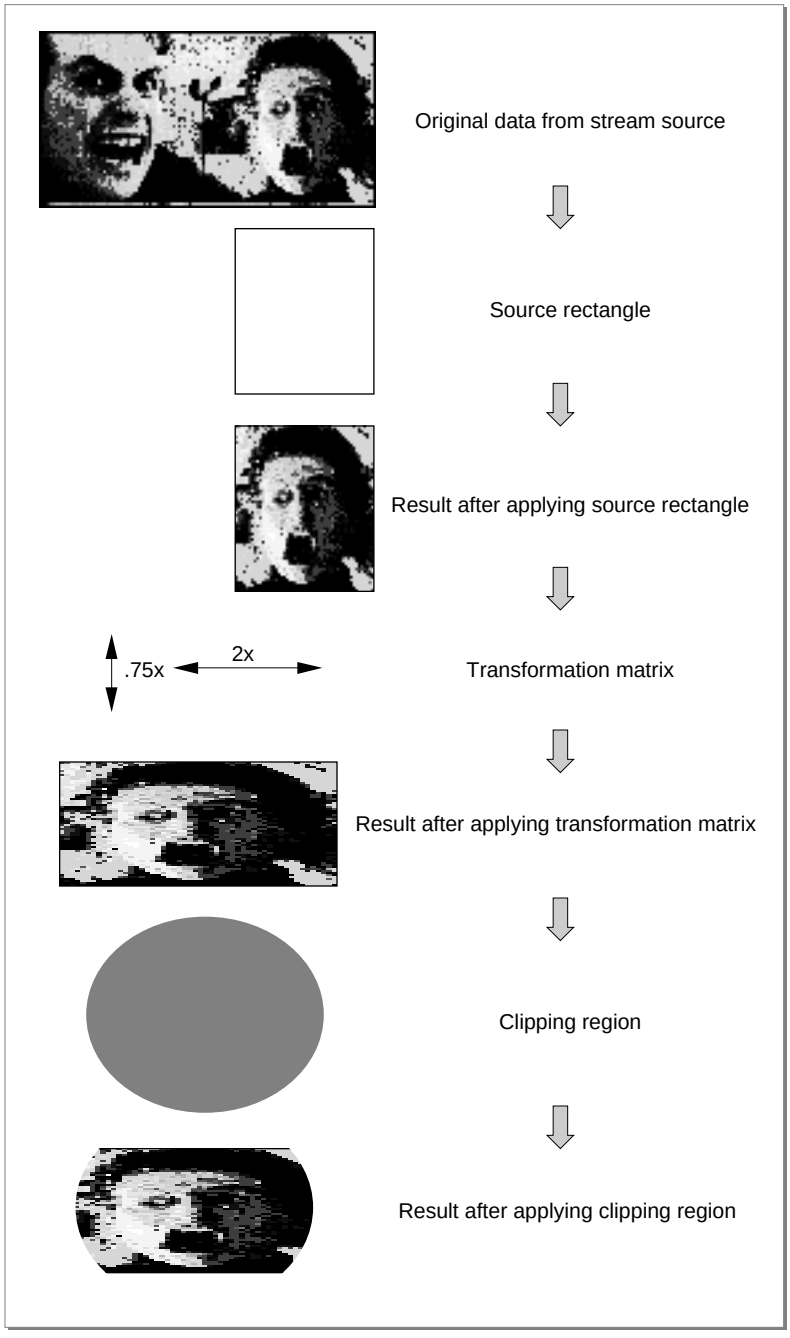
Getting and Setting Characteristics of Visual Stream Data

When you work with streams containing visual data, you must provide information that tells a stream player how to display the data. Before you pass visual stream data to a stream player for display, you need to call these functions:

- the `MTPlayerGetDimensions` function (page 6-29) to get the size of visual data in the stream.
- the `MTPlayerSetRect` function (page 6-22) to set the source rectangle. The source rectangle is the base rectangle to which the transformation matrix and the clipping region are applied. You must set the size of the source rectangle to be equal to or smaller than the size of visual data in the stream.
- the `MTPlayerSetMatrix` function (page 6-26) to set the transformation matrix used for scaling, translating, and rotating images.
- the `MTPlayerSetClip` function (page 6-24) to set the clipping region used to limit the area in which the stream player draws visual data.

When you pass visual stream data to a stream player, the stream player displays on the screen the image that results from applying the source rectangle, the transformation matrix, and the clipping region to the data. Figure 6-1 provides an example of visual stream data, modified by applying a source rectangle, a transformation matrix, and a clipping region.

Figure 6-1 Modifying visual stream data



The stream player API includes functions that allow you to retrieve the current settings for the source rectangle, the transformation matrix, and the clipping region and to set the graphics world for a stream.

You can provide hints for a stream player to help it manage the display by calling the `MTPlayerSetDisplayHints` function (page 6-30). In QuickTime Conferencing 1.0, one display hint is defined—it advises a stream player to update its display.

To capture the current image in a QuickDraw picture, you can call the `MTPlayerGetPicture` function (page 6-31). You provide a pointer to a picture handle; the function allocates memory and provides a handle to the picture.

Getting and Setting Characteristics of Audio Stream Data

For streams that contain audio data, you can query and set the volume level and the sound balance. The volume level is a measure of loudness. The sound balance indicates the mix of sound (the relative loudness) between two speakers or channels.

To set the volume level for a stream, call the `MTPlayerSetVolume` function (page 6-32), as shown here:

```
myErr = MTPlayerSetVolume(myStreamPlayerInstance, volume);
```

You can find out a stream's audio data volume level by calling the `MTPlayerGetVolume` function (page 6-34). You can get and set the sound balance with similar functions (`MTPlayerSetSoundBalance` and `MTPlayerGetSoundBalance` on page 6-34 and page 6-36, respectively).

Playing a Stream Chunk

An incoming stream of media data consists of a series of chunk record structures, each of which contains a chunk of media data and information about that chunk.

To play the incoming stream data, you call the `MTPlayerPlayStream` function once for each chunk. You provide a pointer to a chunk record structure each time you call the function. The chunk record structure looks like this:

```
struct MTStreamChunkRecord {
   Byte              *chunkPtr; /* pointer to chunk of media data */
   MTChunkSize       chunkSize; /* size of chunk in bytes */
   MTChunkTime       chunkTime; /* time of chunk */
   MTChunkPriority   chunkPriority; /* priority of chunk */
   MTStreamID        chunkStreamID; /* stream ID */
   ComponentResult   chunkError; /* error associated with chunk */
   long              chunkRefCon; /* refcon associated with chunk */
   UInt32            chunkReserved; /* reserved */
   long              chunkTimePlayed; /* time chunk was played */
   MTReleaseUPP      chunkReleaseProcCallBack; /* callback function */
   long              chunkPrivate; /* reserved */
   MTStreamOptions   chunkStreamOptions; /* media-specific information */
   UInt8             chunkReserved2; /* reserved */
```

```
   MTSequenceNum     chunkFrameNumber; /* sequence number of chunk */
   UInt16            chunkReserved3; /* reserved */
};
```

At a minimum, a chunk record structure must have values in these fields when you pass it to a stream player:

`chunkPtr`	A pointer to the chunk of media data to be played, such as a group of sound samples or one video frame.
`chunkSize`	The size of the chunk in bytes.
`chunkTime`	A cumulative value, expressed in the time scale for this stream, that represents the temporal position of this chunk in a sequence of chunks.
`chunkPriority`	The priority level of this chunk. The priority is relative to all other chunks in all the streams within a given media channel. Sound chunks have a higher priority than video chunks.
`chunkStreamID`	The stream ID of the stream that contains this chunk.
`chunkError`	An error code.

For more information about these fields that are essential to a stream player, see the description of the `MTPlayerPlayStream` function on page 6-37. The chunk record structure is handled by several QuickTime Conferencing components. For a complete description of the structure, see the chapter "Transport Components" in this book.

Creating Stream Player Components

If you are developing a stream player component, it must support these functions that are common to all stream types:

- `MTPlayerInit`
- `MTPlayerGetPlayerInfo`
- `MTPlayerGetStreamDescription`
- `MTPlayerEnable`
- `MTPlayerIsEnabled`
- `MTPlayerPlayStream`

In addition, if your stream player plays streams that contain visual data, you must support these functions:

- `MTPlayerSetGWorld`
- `MTPlayerGetMatrix`
- `MTPlayerSetMatrix`
- `MTPlayerGetClip`
- `MTPlayerSetClip`

- `MTPlayerGetRect`
- `MTPlayerSetRect`
- `MTPlayerGetPicture`
- `MTPlayerGetDimensions`
- `MTPlayerSetDisplayHints`

If your stream player plays streams that contain audio data, you must support these functions:

- `MTPlayerGetVolume`
- `MTPlayerSetVolume`
- `MTPlayerGetSoundBalance`
- `MTPlayerSetSoundBalance`
- `MTPlayerSamplesPerSecond`

Your stream player may optionally support these functions:

- `MTPlayerGetStreamTimeBase`
- `MTPlayerSetStreamTimeBase`
- `MTPlayerSetDimensions`
- `MTPlayerGetSrcRgn`

Stream Player Components Reference

This section describes the constants and functions that comprise the API of stream player components.

Constants

This section describes the constants in the stream player component API.

Component Type and Subtypes

All stream player components have the component type `'sply'`. Apple defines stream player component subtypes for video and sound data. The subtype of a video stream player component is equated to the QuickTime constant `VideoMediaType`; for a sound stream player component, the subtype is equated to the QuickTime constant `SoundMediaType`.

```
enum {
   kMTPlayerType        = 'sply'
};
```

```
enum {
   kMTVideoSubType      = VideoMediaType,
   kMTSoundSubType      = SoundMediaType
};
```

Stream Player Request Codes

A request code specifies a function in the stream player API. The Component Manager passes the request code to a stream player to indicate which function was called.

```
enum {
   kMTPlayerInitSelect                    = 1,
   kMTPlayerGetStreamDescriptionSelect    = 2,
   kMTPlayerGetPlayerInfoSelect           = 3,
   kMTPlayerPlayStreamSelect              = 4,
   kMTPlayerSetStreamTimeBaseSelect       = 5,
   kMTPlayerGetStreamTimeBaseSelect       = 6,
   kMTPlayerEnableSelect                  = 7,
   kMTPlayerIsEnabledSelect               = 8,
   kMTPlayerSetGWorldSelect               = 9,
   kMTPlayerSetClipSelect                 = 10,
   kMTPlayerGetClipSelect                 = 11,
   kMTPlayerSetRectSelect                 = 12,
   kMTPlayerGetRectSelect                 = 13,
   kMTPlayerSetMatrixSelect               = 14,
   kMTPlayerGetMatrixSelect               = 15,
   kMTPlayerGetPictureSelect              = 16,
   kMTPlayerSetVolumeSelect               = 17,
   kMTPlayerGetVolumeSelect               = 18,
   kMTPlayerSetSoundBalanceSelect         = 19,
   kMTPlayerGetSoundBalanceSelect         = 20,
   kMTPlayerSetDimensionsSelect           = 21,
   kMTPlayerGetDimensionsSelect           = 22,
   kMTPlayerGetSrcRgnSelect               = 23,
   kMTPlayerSamplesPerSecondSelect        = 24,
   kMTPlayerSetDisplayHintsSelect         = 25
};
```

Functions

This section describes the functions that are provided by a stream player component. These functions are described from the perspective of one who calls a stream player component. Usually, a stream director component calls stream player component

functions. Although applications can call stream player component functions, usually they don't. If you are developing a stream player component, your component must support the required functions as they are described here.

This section is organized into the following groups of functions:

- "Managing a Stream Player Component" describes the functions that you use to initialize a stream player and to get certain component-level characteristics, such as whether it can play a stream that contains audio data.
- "Getting and Setting Basic Stream Characteristics" describes the functions that you use to get information about a stream (such as its stream description and its sample rate), to get and set a stream's timebase, to enable or disable a stream, and to determine its enable status.
- "Getting and Setting Characteristics of Visual Stream Data" describes the functions that control characteristics of a stream's visual display.
- "Getting and Setting Characteristics of Audio Stream Data" describes the functions that control characteristics of a stream's audio playback.
- "Playing Stream Data" describes the function that you use to play stream data.

All of the functions take a component instance parameter. You obtain a component instance by calling the Component Manager's `OpenDefaultComponent` function. See page 6-5 for an example of how to call this function.

Managing a Stream Player Component

You use the functions in this section to initialize a stream player component and to find out about its capabilities.

MTPlayerInit

The `MTPlayerInit` function sets up a stream player component to control a stream that you specify.

```
pascal ComponentResult MTPlayerInit (MTPlayerComponent spc,
         MTStreamDescriptionHandle description,
         MTReleaseUPP proc);
```

`spc` The stream player component instance that you are calling.

`description`
A handle to a `MTStreamDescription` structure. You allocate the memory and provide a fully specified stream description. The function does not modify any values in the structure.

`proc` A universal procedure pointer to a release function that releases the memory for a chunk. If you are using Apple-provided QuickTime Conferencing components, you must provide a release function.

DESCRIPTION

You call the `MTPlayerInit` function to inform a stream player component about the characteristics of the particular stream you want it to play. You must call this function after you open the stream player and before you call any other stream player function.

The stream description structure contains a full description of the stream, which the stream player uses to initialize itself. The structure looks like this:

```
struct MTStreamDescription {
   MTStreamType         streamType;       /* type of media data in stream */
   MTStreamID           streamID;         /* temporary stream identifier */
   MTStreamReference    streamReference;  /* persistent stream indentifier */
   TimeScale            streamTimeScale;  /* stream's time scale */
   SampleDescription    streamSampleDescription; /* description of specific
                                                    media data type */
};
```

The stream player calls the release function to deallocate chunk memory after it has played a chunk.

This function can return errors from the Image Compression Manager and the Memory Manager.

RESULT CODES

`noErr`	0	No error

SEE ALSO

See page 6-39 for a description of the release function that you provide to release chunk memory.

The stream description structure is described in the chapter "Stream Director Components" in this book.

You can tell a stream player to play stream data by calling the `MTPlayerPlayStream` function, described on page 6-37.

You open a stream player component by calling either the `OpenDefaultComponent` or `OpenComponent` function. They are described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Universal procedure pointers are described in *Inside Macintosh: PowerPC System Software*.

MTPlayerGetPlayerInfo

The `MTPlayerGetPlayerInfo` function provides information about a stream player component.

```
pascal ComponentResult MTPlayerGetPlayerInfo
            (MTPlayerComponent spc, MTStreamInfo *playerFlags);
```

`spc` The stream player component instance about which you want information.

`playerFlags`
A pointer to a set of bit flags that describe the capabilities of the stream player. You allocate the memory for and provide the pointer to the flags, and the function sets the flags.

`streamHasBounds`
Indicates whether a stream player can play a stream that contains visual data. If the function sets this flag to 1, the stream player can play a stream that contains visual data. In that case, you can call the stream player functions that pertain to visual data.

`streamHasVolume`
Indicates whether a stream player can play a stream that contains audio data. If the function sets this flag to 1, the stream player can play a stream that contains audio data. In that case, you can call the stream player functions that pertain to audio data.

`streamHasDiscreteSamples`
Indicates whether the media data played by the stream player is organized into discrete samples. If the function sets this flag to 1, the media data is organized into discrete samples, meaning that the player receives the samples as they come, at no particular sample rate. If the function sets this flag to 0, the media data has a fixed sample rate.

`streamIsInterruptCapable`
Indicates whether a stream player is capable of playing data at interrupt time. If the function sets this flag to 1, the stream player is capable of playing data at interrupt time.

DESCRIPTION

You call the `MTPlayerGetPlayerInfo` function to find out certain characteristics of a stream player component and the data it plays.

RESULT CODES

`noErr`	0	No error

SEE ALSO

See page 6-10 for a list of the functions you can use with stream players that can play audio and visual data.

If the media data played by a stream player is not organized into discrete samples, then you can find out the stream's sample rate by calling the `MTPlayerSamplesPerSecond` function, described on page 6-18.

The `MTStreamInfo` data type is described in the chapter "Stream Director Components" in this book.

Getting and Setting Basic Stream Characteristics

You use the functions described in this section to get and set stream characteristics that are not limited to streams containing visual or audio data. These functions allow you to

- get the stream description of a stream that you specify
- enable or disable a stream
- find out whether a stream is enabled or disabled
- get the sample rate for a stream
- get and set a stream's timebase

MTPlayerGetStreamDescription

The `MTPlayerGetStreamDescription` function provides the stream description for a stream.

```
pascal ComponentResult MTPlayerGetStreamDescription
                    (MTPlayerComponent spc,
                    MTStreamDescriptionHandle streamDesc);
```

`spc`	The stream player component instance from which you want information.
`streamDesc`	A handle to a stream description structure. You must allocate the memory for a stream description structure and provide a handle to it. The function provides the stream description structure and resizes the memory block based on the size of the structure.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTPlayerInit` function description on page 6-13 shows the stream description structure declaration. The structure is described in the chapter "Stream Director Components" in this book.

MTPlayerEnable

The `MTPlayerEnable` function enables or disables a stream.

```
pascal ComponentResult MTPlayerEnable
            (MTPlayerComponent spc, Boolean enableMode);
```

`spc` The stream player component instance that controls the stream you want to enable or disable.

`enableMode` A Boolean value that indicates whether you want to enable or disable a stream. Set this parameter to `true` to enable the stream. Set it to `false` to disable the stream.

DESCRIPTION

When a stream is disabled, a stream player does not play any stream data it may receive.

SPECIAL CONSIDERATIONS

A stream player should immediately discard any data it receives for a disabled stream by calling the release function provided through the `MTPlayerInit` function. This increases the efficiency of memory management.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTPlayerInit` function is described on page 6-13.

You can check the enable status of a stream with the `MTPlayerIsEnabled` function, described next.

MTPlayerIsEnabled

The `MTPlayerIsEnabled` function tells you whether a stream is enabled or disabled.

```
pascal ComponentResult MTPlayerIsEnabled
            (MTPlayerComponent spc, Boolean *enableMode);
```

`spc` The stream player component instance that controls the stream whose status you are querying.

`enableMode` A Boolean value that indicates whether the stream is enabled or disabled. The function sets this parameter to `true` if the stream is enabled. It sets it to `false` if the stream is disabled.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can enable or disable a stream with the `MTPlayerEnable` function, described on page 6-17.

MTPlayerSamplesPerSecond

The `MTPlayerSamplesPerSecond` function gets the sample rate of the data in a stream.

```
pascal ComponentResult MTPlayerSamplesPerSecond
            (MTPlayerComponent spc, Fixed *samplesPerSecond);
```

`spc` The stream player component you are using.

`samplesPerSecond`
A pointer to the sample rate of the stream data played by the stream player.

DESCRIPTION

You call the `MTPlayerSamplesPerSecond` function to find out the sample rate of the stream data handled by a given stream player. The sample rate is a characteristic of a particular stream. For example, a sound stream player might play a sound stream whose sample rate is 22,545 samples per second at one time and a sound stream whose sample rate is 44,100 samples per second at another time.

Streams that do not have a fixed sample rate are said to be organized into discrete samples. If you call `MTPlayerSamplesPerSecond` and the player's stream does not have a fixed sample rate, the function returns the `badComponentSelector` result code.

In addition to the result codes listed below, this function may return Sound Manager result codes.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

You can find out if a player's stream data has discrete samples by calling the `MTPlayerGetPlayerInfo` function, described on page 6-15.

MTPlayerSetStreamTimeBase

The `MTPlayerSetStreamTimeBase` function sets the time base of a stream that carries media data from a stored source.

```
pascal ComponentResult MTPlayerSetStreamTimeBase
                    (MTPlayerComponent spc, TimeBase timebase);
```

`spc`	The stream player component instance that controls the stream.
`timebase`	A pointer to a `TimeBaseRecord` structure in which you specify the time base of the stream. You can obtain this value from the `NewTimeBase` and `GetMovieTimeBase` functions.

DESCRIPTION

You call the `MTPlayerSetStreamTimeBase` function to set the time base of a stream. This function applies only to streams that carry media data from stored sources, such as an MPEG stream. (The time base of a stream carrying live media data is set at the source connection end.)

For multiplexed streams, the time base is the same for each type of data carried in the stream. `MTPlayerSetStreamTimeBase` sets the time base for the multiplexed stream as a whole.

SPECIAL CONSIDERATIONS

This function is not supported by the sound and video stream players provided by Apple Computer, Inc. in QuickTime Conferencing 1.0.

All streams managed by a stream director should have the same time base. If you call the `MTStreamPlayerSetTimeBase` function and set the time base of a stream to something different from the time base shared by all streams managed by a given sink stream director, the results are unpredictable.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

The `NewTimeBase` and `GetMovieTimeBase` functions are described in *Inside Macintosh: QuickTime*.

For more information about multiplexed streams, see "Single-Media and Multiplexed Streams" on page 6-4.

MTPlayerGetStreamTimeBase

The `MTPlayerGetStreamTimeBase` function tells you the time base of a stream.

```
pascal ComponentResult MTPlayerGetStreamTimeBase
            (MTPlayerComponent spc, TimeBase *timebase);
```

`spc` The stream player component instance that controls the stream.

`timebase` The address of a pointer to a `TimeBaseRecord` structure. The function returns the stream's time base information in the structure.

DESCRIPTION

You call the `MTPlayerGetStreamTimeBase` function to get the time base of a stream.

For multiplexed streams, the time base is the same for each type of data carried in the stream. `MTPlayerGetStreamTimeBase` returns the time base for the multiplexed stream as a whole.

SPECIAL CONSIDERATIONS

This function is not supported by the sound and video stream players provided by Apple Computer, Inc. in QuickTime Conferencing 1.0.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

For more information about multiplexed streams, see "Single-Media and Multiplexed Streams" on page 6-4.

Getting and Setting Characteristics of Visual Stream Data

You use the functions described in this section to set certain characteristics of streams that contain visual data and to find out the current settings of those characteristics. For a stream containing visual data, you can

- set its graphics world
- get and set the rectangle within which its media data is displayed
- get and set its clipping region
- get and set its transformation matrix
- get its display region
- get and set the height and width of its visual data
- advise a stream player that it should update its display
- get a picture of the current visual image

MTPlayerSetGWorld

The `MTPlayerSetGWorld` function sets the graphics port and graphics device for a stream that contains visual data.

```
pascal ComponentResult MTPlayerSetGWorld
            (MTPlayerComponent spc, CGrafPtr gp, GDHandle gd);
```

`spc`	The stream player component instance that controls the stream whose graphics port and graphics device you want to set.
`gp`	A pointer to a `CGrafPort` structure that specifies the graphics port in which to display the stream's visual data. To use the current graphics port, you should set `gp` explicitly to the current graphics port.
`gd`	A handle to a `GDevice` structure that specifies the graphics device on which to display the stream's visual data. To use the current graphics device, you should set `gd` explicitly to the current graphics device.

DESCRIPTION

A stream player automatically sets its graphics port and graphics device when you call the `MTPlayerInit` function. You can call the `MTPlayerSetGWorld` function to provide a new graphics port or graphics device.

The stream player uses the pointer to the `CGrafPort` structure and the handle to the `GDevice` structure, but it does not set or examine any of the fields in the structures themselves.

Typically, you use a color graphics port, but the function also accepts a black-and-white graphics port.

You can call `MTPlayerSetGWorld` while passing a stream player a sequence of chunk record structures—that is, during a series of calls to the `MTPlayerPlay` function.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

For further information about working with `GDevice` and `CGrafPort` structures, see *Inside Macintosh: Imaging With QuickDraw*.

MTPlayerSetRect

The `MTPlayerSetRect` function sets the source rectangle of a stream.

```
pascal ComponentResult MTPlayerSetRect
                                (MTPlayerComponent spc, Rect *r);
```

`spc` The stream player component instance that controls the stream whose source rectangle you want to set.

`r` A pointer to the desired rectangle for the stream.

DESCRIPTION

You call the `MTPlayerSetRect` function to set the source rectangle of a stream that contains visual data.

The source rectangle is a rectangle smaller than or equal to the boundary rectangle of a stream. (A stream's boundary rectangle is set by the component that generates the stream data.) The source rectangle is located within or congruent with the stream's boundary rectangle. Typically, the source rectangle is equal to the boundary rectangle.

A stream player uses a transformation matrix to translate, scale, and rotate the source rectangle and produce a rectangle used for display (referred to as the *display rectangle*).

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

You can find out the boundary rectangle for a stream by calling the `MTPlayerGetDimensions` function, described on page 6-29.

You can set the transformation matrix for a stream with the `MTPlayerSetMatrix` function, described on page 6-26.

You can retrieve the source rectangle of a stream by calling the `MTPlayerGetRect` function, described next.

The section "Getting and Setting Characteristics of Visual Stream Data" on page 6-7 provides an overview of how to prepare a stream player to display visual data.

MTPlayerGetRect

The `MTPlayerGetRect` function retrieves the source rectangle of a stream.

```
pascal ComponentResult MTPlayerGetRect
                                (MTPlayerComponent spc, Rect *r);
```

`spc` The stream player component instance that controls the stream whose rectangle you want to query.

`r` A pointer to a rectangle structure. The function returns the source rectangle for the stream.

DESCRIPTION

You call the `MTPlayerGetRect` function to get the source rectangle of a stream.

The source rectangle is a rectangle smaller than or equal to the boundary rectangle of a stream. (A stream's boundary rectangle is set by the component that generates the stream data.) The source rectangle is located within or congruent with the stream's boundary rectangle. Typically, the source rectangle is equal to the boundary rectangle.

A stream player uses a transformation matrix to translate, scale, and rotate the source rectangle and produce a rectangle used for display (referred to as the *display rectangle*).

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

You set the source rectangle by calling the `MTPlayerSetRect` function, described on page 6-22.

You can set the transformation matrix for a stream with the `MTPlayerSetMatrix` function, described on page 6-26.

You can find out the boundary rectangle for a stream by calling the `MTPlayerGetDimensions` function, described on page 6-29.

MTPlayerSetClip

The `MTPlayerSetClip` function sets the clipping region for a stream that contains visual data.

```
pascal ComponentResult MTPlayerSetClip
            (MTPlayerComponent spc, RgnHandle theClip);
```

`spc` The stream player component instance that controls the stream whose clip region you want to set.

`theClip` A handle to a `Region` structure that specifies the clipping region for the stream. To remove a clipping region, set the parameter to `nil`.

DESCRIPTION

A stream player copies the clipping region that you pass to the `MTPlayerSetClip` function, so you are responsible for disposing of the handle.

In addition to the result codes listed, `MTPlayerSetClip` may return Image Compression Manager result codes.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

Clipping regions and the `Region` structure are discussed in *Inside Macintosh: Imaging With QuickDraw*. You can find a discussion of clipping QuickTime movie tracks in *Inside Macintosh: QuickTime*.

The section "Getting and Setting Characteristics of Visual Stream Data" on page 6-7 provides an overview of how to prepare a stream player to display visual data.

You can retrieve a stream's clipping region by calling the `MTPlayerGetClip` function, described next.

MTPlayerGetClip

The `MTPlayerGetClip` function retrieves the clipping region for a stream that contains visual data.

```
pascal ComponentResult MTPlayerGetClip
            (MTPlayerComponent spc, RgnHandle *theClip);
```

`spc` The stream player component instance that controls the stream whose clipping region you want to retrieve.

`theClip` The address of a handle to a `Region` structure. The function allocates the memory for the `Region` structure and returns a handle to it in this parameter. The function sets the structure fields to the stream's current clipping region values. You are responsible for disposing of the `Region` structure. If no clipping region has been set, the function sets this parameter to `nil`.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

Clipping regions and the `Region` structure are discussed in *Inside Macintosh: Imaging With QuickDraw*. You can find a discussion of clipping QuickTime movie tracks in *Inside Macintosh: QuickTime*.

You can set a stream's clipping region by calling the `MTPlayerSetClip` function, described in the previous section.

MTPlayerSetMatrix

The `MTPlayerSetMatrix` function sets the display transformation matrix for a stream.

```
pascal ComponentResult MTPlayerSetMatrix
            (MTPlayerComponent spc, MatrixRecord *m);
```

`spc` The stream player component instance that controls the stream whose matrix you want to set.

`m` A pointer to a fully specified transformation matrix for the stream. You provide the matrix structure. The stream player does not modify the matrix.

DESCRIPTION

You call the `MTPlayerSetMatrix` function to set the display transformation matrix of a stream that contains visual data. A transformation matrix defines how to map points from one coordinate space into another. You use it for scaling, translation, and rotation.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

For more information about transformation matrices, see the chapter "Movie Toolbox" in *Inside Macintosh: QuickTime*.

The section "Getting and Setting Characteristics of Visual Stream Data" on page 6-7 provides an overview of how to prepare a stream player to display visual data.

You can retrieve a stream's matrix by calling the `MTPlayerGetMatrix` function, described next.

MTPlayerGetMatrix

The `MTPlayerGetMatrix` function retrieves the display transformation matrix for a stream.

```
pascal ComponentResult MTPlayerGetMatrix
            (MTPlayerComponent spc, MatrixRecord *m);
```

`spc` The stream player component instance that controls the stream whose matrix you want to query.

`m` A pointer to a matrix structure. You allocate the memory for the matrix structure. The function sets the structure fields to the stream's current matrix values.

DESCRIPTION

You call the `MTPlayerGetMatrix` function to retrieve the display transformation matrix of a stream that contains visual data. A transformation matrix defines how to map points from one coordinate space into another. You use it for scaling, translation, and rotation.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

For more information about transformation matrices, see the chapter "Movie Toolbox" in *Inside Macintosh: QuickTime*.

You can set a stream's matrix values by calling the `MTPlayerSetMatrix` function, described in the previous section.

MTPlayerGetSrcRgn

The `MTPlayerGetSrcRgn` function retrieves the display region of a stream.

```
pascal ComponentResult MTPlayerGetSrcRgn
            (MTPlayerComponent spc, RgnHandle srcRgn,
            TimeValue requestTime);
```

`spc` The stream player component you are using.

`srcRgn` A handle to a `Region` structure. You initialize the region to the stream's boundary rectangle. If the boundary rectangle and the stream's display region are not equal, the function alters the region so that it corresponds to the boundary of the stream's display image.

`requestTime`
Reserved.

DESCRIPTION

You call the `MTPlayerGetSrcRgn` function to determine whether a stream has an irregularly shaped display region.

The display region exists within the stream's boundary rectangle.

SPECIAL CONSIDERATIONS

This function is not supported by the video stream player provided by Apple Computer, Inc. in QuickTime Conferencing 1.0.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

MTPlayerSetDimensions

The `MTPlayerSetDimensions` function sets the height and width of the visual data in a stream.

```
pascal ComponentResult MTPlayerSetDimensions
            (MTPlayerComponent spc, Fixed width,
            Fixed height);
```

`spc`	The stream player component you are using.
`width`	The width, in pixels, of the visual data in a stream.
`height`	The height, in pixels, of the visual data in a stream.

DESCRIPTION

You call the `MTPlayerSetDimensions` function to set the height and width of the visual data in a stream.

Height and width are expressed in pixels and are properties of the visual data that a stream player receives—that is, they are part of a stream's format description. The height and width values do not refer to any coordinate system—they are absolute values and don't describe how the data is represented on a screen.

If you know the format of a stream's media data, you can obtain initial dimension information from the media-specific `SampleDescription` part of a stream description structure.

SPECIAL CONSIDERATIONS

When this function is called, a stream player may need to update the stream description structure, depending on the stream type.

This function is not supported by the video stream player provided by Apple Computer, Inc. in QuickTime Conferencing 1.0.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

The stream description structure is described in the chapter "Stream Director Components" in this book.

To retrieve the dimensions of the visual data in a stream, you can call the `MTPlayerGetDimensions` function, described next.

MTPlayerGetDimensions

The `MTPlayerGetDimensions` function retrieves the height and width, expressed in pixels, of the visual data in a stream.

```
pascal ComponentResult MTPlayerGetDimensions
            (MTPlayerComponent spc, Fixed *width,
            Fixed *height);
```

`spc`	The stream player component you are using.
`width`	A pointer to a fixed-point value. The function sets this value to the width, in pixels, of the visual data in the stream.
`height`	A pointer to a fixed-point value. The function sets this value to the height, in pixels, of the visual data in the stream.

DESCRIPTION

You call the `MTPlayerGetDimensions` function to retrieve values for the height and width of the visual data in a stream. These values are set by the component that generated the data.

Height and width are properties of the visual data that a stream player receives—that is, they are part of a stream's format description. They do not describe how that data is represented on a screen.

You can use this function to determine the stream dimensions independent of the format of the stream's media data.

RESULT CODES

`noErr`	0	No error
`invalidSampleDescription`	–2041	Invalid sample description
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

The section "Getting and Setting Characteristics of Visual Stream Data" on page 6-7 provides an overview of how to prepare a stream player to display visual data.

Typically, stream dimensions are encoded in the media-specific `SampleDescription` part of a stream description. If you know the format of a stream's media data, you can get the dimensions by calling the `MTPlayerGetStreamDescription` function, described on page 6-16. For example, height and width are fields in an `ImageDescription` structure (described in *Inside Macintosh: QuickTime*).

The stream description structure is described in the chapter "Stream Director Components" in this book.

MTPlayerSetDisplayHints

The `MTPlayerSetDisplayHints` function tells a stream player to perform certain operations on its display of visual stream data.

```
pascal ComponentResult MTPlayerSetDisplayHints
            (MTPlayerComponent spc, MTPlayerHintFlags flags);
```

`spc`	The stream player component instance that you are using.
`flags`	A set of bit flags that tell a stream player what it should do to its display of visual stream data.

DESCRIPTION

You call the `MTPlayerSetDisplayHints` function to tell a stream player to take certain actions on its visual display.

One flag is currently defined:

```
enum {
   kMTDisplayUpdateHintMask = 0x00000001
};
```

To tell a stream player that it needs to refresh the screen (redraw the visual display), set the `kMTDisplayUpdateHintMask` flag to 1. You should set the rest of the bit flags to 0.

In addition to the result codes listed below, `MTPlayerSetDisplayHints` may return Image Compression Manager result codes.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

MTPlayerGetPicture

The `MTPlayerGetPicture` function provides a QuickDraw picture of a stream's visual data.

```
pascal ComponentResult MTPlayerGetPicture
            (MTPlayerComponent spc, PicHandle *p,
            const Rect *bounds, short offscreenDepth,
            long grabPictFlags);
```

`spc` The stream player component instance you are using.

`p` The address of a handle to a `Picture` structure. The function allocates memory for the `Picture` structure and provides a handle to it. You are responsible for disposing of the memory.

`bounds` A pointer to a `Rect` structure that defines the size and position of the boundary rectangle into which the stream player draws the picture. You must provide this value regardless of how you set the `grabPictOffScreen` flag.

`offscreenDepth`
The pixel depth of the offscreen graphics world. Set this to 0 to specify the best available depth. The function uses this parameter only when you set the `grabPictOffScreen` flag to 1. If the `grabPictOffScreen` flag is 0, the function ignores this parameter.

`grabPictFlags`
A set of bit flags that you can use to control how the picture is drawn. The following flags are defined:

`grabPictOffScreen`
Set this flag to 0 to draw the picture onscreen. If you set this flag to 1, the stream player draws the picture in an offscreen graphics world.

`grabPictIgnoreClip`
Set this flag to 0 to use the existing clipping region for the image. If you set this flag to 1, the stream player ignores any current clipping region for the image.

Set all unused flags to 0.

DESCRIPTION

You call the `MTPlayerGetPicture` function to get a picture of a stream that contains visual data.

The function allocates the memory for a `Picture` structure, stores the picture in the structure, and returns a handle to it. You are responsible for disposing of the memory.

The function draws the picture on the screen or in an offscreen buffer, depending on the flags you provide.

In addition to the result codes listed below, `MTPlayerGetPicture` may return Image Compression Manager result codes.

SPECIAL CONSIDERATIONS

In QuickTime Conferencing 1.0, the Apple-provided video stream player saves the most recent image it has drawn to the screen. When you call the `MTPlayerGetPicture` function, it retrieves that most recent image and puts it in the picture structure. However, when you use temporal compression, it saves only key video frames, not delta video frames. Therefore, the picture the function returns corresponds to what was in the most recent key frame, not necessarily exactly what is on the screen at the time you call `MTPlayerGetPicture`.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

The `Picture` and `Rect` structures and offscreen graphics worlds are described in *Inside Macintosh: Imaging With QuickDraw*.

Getting and Setting Characteristics of Audio Stream Data

You use the functions described in this section to set the volume and the sound balance of streams that contain audio data and to find out the current settings of those characteristics.

MTPlayerSetVolume

The `MTPlayerSetVolume` function sets the volume for a stream that contains audio data.

```
pascal ComponentResult MTPlayerSetVolume
            (MTPlayerComponent spc, short volume);
```

`spc` The stream player component instance that controls the stream whose volume you want to set.

`volume` The volume level that you want to set. Volume is expressed as a 16-bit fixed-point number in the range –1.0 to +1.0. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Positive values indicate sound that ranges from very soft to very loud. A value of 0 indicates no sound. Negative values play no sound, but preserve the absolute value of the volume setting.

DESCRIPTION

Table 6-1 contains examples of selected volume levels and the resulting effect on the audio volume played.

Table 6-1 Volume settings and their results

Volume as fixed-point value	Volume as hex value	Effect on volume played
1.0	0x0100	Full volume
0.0	0x0000	Silence
0.5	0x0080	Half volume
–1.0	0xff00	Silence
–0.5	0xff7f	Silence

In addition to the result codes listed, `MTPlayerSetVolume` may return result codes from the Sound Manager and the Memory Manager.

SPECIAL CONSIDERATIONS

The `MTPlayerSetVolume` function accepts volume values greater than 1.0. Such values overdrive the volume, and may result in decreased sound quality.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Volume parameter out of range
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

You can find out the volume setting for an audio stream by calling the `MTPlayerGetVolume` function, described next.

MTPlayerGetVolume

The `MTPlayerGetVolume` function gets the volume for a stream that contains audio data.

```
pascal ComponentResult MTPlayerGetVolume
            (MTPlayerComponent spc, short *volume);
```

`spc` The stream player component instance that controls the stream whose volume you want to query.

`volume` A pointer to the volume level of the stream. The function returns the volume level. Volume is expressed as a 16-bit fixed-point number in the range –1.0 to +1.0. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Positive values indicate sound that ranges from very soft to very loud. A value of 0 indicates no sound. Negative values indicate no sound, but show the absolute value of the volume setting.

DESCRIPTION

In addition to the result codes listed, `MTPlayerGetVolume` may return result codes from the Sound Manager and the Memory Manager.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

See the `MTPlayerSetVolume` function description on page 6-32 for examples of volume settings.

MTPlayerSetSoundBalance

The `MTPlayerSetSoundBalance` function sets the sound balance for a stream that contains audio data.

```
pascal ComponentResult MTPlayerSetSoundBalance
            (MTPlayerComponent spc, short balance);
```

`spc` The stream player component you are using.

`balance` The relative mix of sound between two speakers. Specify this as a 16-bit fixed-point value. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Valid balance values range from –1.0 to +1.0. Negative values emphasize the left sound channel; positive values emphasize the right. A value of 0 specifies a neutral balance.

DESCRIPTION

You call the `MTPlayerSetSoundBalance` function to set the sound balance for a stream.

If the sound source is monaural, the balance setting controls the relative loudness of each speaker.

If the sound source is stereo, the balance setting controls the mix of the right and left speaker. For example, 0 means both speakers are at full volume; –1 means the left speaker is at full volume and the right has no volume; +1 means the right speaker is at full volume and the left has no volume. As the balance value increases from 0.0 to 1.0, the volume of the left channel decreases. As the balance value changes from 0.0 to –1.0, the volume of the right channel decreases.

Table 6-2 contains examples of selected balance levels.

Table 6-2 Balance settings and their results

Fixed-point value	Hex value	Resulting balance
1.0	0x0100	Right channel full strength, left channel silent
0.0	0x0000	Left and right channels are equal
0.5	0x0080	Right channel full strength, left channel at half strength
–1.0	0xff00	Left channel full strength, right channel silent

In addition to the result codes listed, `MTPlayerSetSoundBalance` may return result codes from the Sound Manager and the Memory Manager.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

You can find out the balance setting for an audio stream by calling the `MTPlayerGetSoundBalance` function, described next.

MTPlayerGetSoundBalance

The `MTPlayerGetSoundBalance` function gets the balance for a stream that contains audio data.

```
pascal ComponentResult MTPlayerGetSoundBalance
            (MTPlayerComponent spc, short *balance);
```

`spc` The stream player component you are using.

`balance` A pointer to the relative mix of sound between two speakers. The function returns this as a 16-bit fixed-point value. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. The balance value ranges from –1.0 to +1.0. Negative values emphasize the left sound channel; positive values emphasize the right. A value of 0 specifies a neutral balance.

DESCRIPTION

In addition to the result codes listed, `MTPlayerGetSoundBalance` may return result codes from the Sound Manager and the Memory Manager.

RESULT CODES

`noErr`	0	No error
`badComponentSelector`	–32766	Function not supported by this component

SEE ALSO

See the `MTPlayerSetSoundBalance` function description on page 6-34 for examples of balance settings.

Playing Stream Data

You use the function in this section to play stream data.

MTPlayerPlayStream

The `MTPlayerPlayStream` function plays a chunk of stream data.

```
pascal ComponentResult MTPlayerPlayStream
            (MTPlayerComponent spc, MTStreamChunkRecordPtr chunk);
```

`spc` The stream player component instance that you are calling.

`chunk` A pointer to an `MTStreamChunkRecord` structure that describes the stream chunk that you want to play.

DESCRIPTION

You call the `MTPlayerPlayStream` function to play a chunk of stream data. The stream player plays the portion of the stream defined by the chunk record structure that you provide.

When you call the `MTPlayerPlayStream` function, the chunk record structure must contain values in the fields listed below. Typically, a transport component sets these fields before passing the chunk record structure to a sink stream director.

`chunkPtr` A pointer to the chunk of media data to be played, such as a group of sound samples or one video frame.

`chunkSize` The size of the chunk in bytes.

`chunkTime` A cumulative value, expressed in the time scale for this stream, that represents the temporal position of this chunk in a sequence of chunks. The time scale for the stream can be found in the stream description structure (described in the chapter "Stream Director Components" in this book). The chunk time starts accumulating (initial value is 0) when the media component at the source begins generating media data for the stream.

`chunkPriority` The priority level of this chunk. The priority is relative to all other chunks in all the streams within a given media channel. The following priorities are defined:

`mtChunkPrioritySound`
The priority level for sound data. It is the highest priority among the four defined priority levels.

`mtChunkPriorityMedium`
An intermediate priority level. It falls between the priorities for sound and video data.

`mtChunkPriorityVideo`
The priority for delta frames of video data. It is the lowest priority among the four defined priority levels. Delta frames are distinguished from key frames in that a delta frame contains only change information from a previous frame. A key frame contains complete information for the video frame.

`mtChunkPriorityKeyFrame`
A constant used to generate a priority for chunks containing key frames. You can use it with any media type to generate a priority that applies to a chunk on which you resynchronize.You generate the key frame priority value by performing a bitwise OR operation on `mtChunkPriorityKeyFrame` and another priority constant, such as `mtChunkPriorityVideo`.

`chunkStreamID` The stream ID of the stream that contains this chunk.

`chunkError` An error code. Transport, stream director, and flow control components can set this field for an incoming chunk. For example, a transport component sets it to indicate whether the chunk was received from the network without data loss. The value 0 indicates the chunk was received in full. If the field is set to `mtChunkIncompleteErr`, it indicates that the chunk was missing data when it arrived from the network. A stream director component can set this field to the `mtChunkTimedOutErr` result code if an intact chunk arrived too late to play.

After playing the chunk, a stream player calls the release function (that you provided to the `MTPlayerInit` function) to release the memory occupied by the chunk record structure. But before calling the release function, the stream player sets the `chunkTimePlayed` field of the chunk record structure to the time that it finished playing the chunk data. The time is expressed in the time scale for the stream.

If a stream player does not or cannot play a chunk immediately, it may queue the chunk.

The `MTPlayerPlayStream` function may return result codes from the Image Compression Manager and the Sound Manager.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The chunk record structure is handled by several QuickTime Conferencing components. For a complete description of the structure, see the chapter “Transport Components” in this book.

Application-Defined Function

This section describes the release function that you provide to a stream player component.

The release function is called by a stream player component to deallocate chunk memory when it no longer needs it.

MyReleaseProc

When you call the `MTPlayerInit` function, you provide a pointer to a release function that releases the memory for a stream chunk. The release function must support the following interface:

```
pascal void MyReleaseProc (MTStreamChunkRecordPtr chunk);
```

`chunk` A pointer to the chunk record structure to be released.

DESCRIPTION

The release function deallocates the memory for the chunk record structure.

A stream player calls the release function after it plays the chunk or when it receives data from a disabled stream.

SPECIAL CONSIDERATIONS

A transport component acquires memory for a chunk record structure as it builds the structure from packets it receives from a network component. You can call the `MTTransportGetReleaseProc` function to get the address of a transport component's release function and then pass that address to a stream player through the `MTPlayerInit` function.

SEE ALSO

The `MTPlayerInit` function is described on page 6-13.

The `MTStreamChunkRecord` structure and the `MTTransportGetReleaseProc` function are described in the chapter "Transport Components" in this book. (You can find a description of the chunk record structure fields that are essential to a stream player on page 6-37.)

Summary of Stream Player Components

C Summary

Constants

```
enum {      /* stream player component type */
   kMTPlayerType        = 'sply'
};

enum {      /* stream player component subtypes */
   kMTVideoSubType      = VideoMediaType,
   kMTSoundSubType      = SoundMediaType
};

enum {      /* stream player request codes */
   kMTPlayerInitSelect                    = 1,
   kMTPlayerGetStreamDescriptionSelect    = 2,
   kMTPlayerGetPlayerInfoSelect           = 3,
   kMTPlayerPlayStreamSelect              = 4,
   kMTPlayerSetStreamTimeBaseSelect       = 5,
   kMTPlayerGetStreamTimeBaseSelect       = 6,
   kMTPlayerEnableSelect                  = 7,
   kMTPlayerIsEnabledSelect               = 8,
   kMTPlayerSetGWorldSelect               = 9,
   kMTPlayerSetClipSelect                 = 10,
   kMTPlayerGetClipSelect                 = 11,
   kMTPlayerSetRectSelect                 = 12,
   kMTPlayerGetRectSelect                 = 13,
   kMTPlayerSetMatrixSelect               = 14,
   kMTPlayerGetMatrixSelect               = 15,
   kMTPlayerGetPictureSelect              = 16,
   kMTPlayerSetVolumeSelect               = 17,
   kMTPlayerGetVolumeSelect               = 18,
   kMTPlayerSetSoundBalanceSelect         = 19,
   kMTPlayerGetSoundBalanceSelect         = 20,
   kMTPlayerSetDimensionsSelect           = 21,
   kMTPlayerGetDimensionsSelect           = 22,
   kMTPlayerGetSrcRgnSelect               = 23,
```

```
   kMTPlayerSamplesPerSecondSelect        = 24,
   kMTPlayerSetDisplayHintsSelect         = 25
};

enum {      /* value for MTPlayerHintFlags type */
   kMTDisplayUpdateHintMask   = 0x00000001
};
```

Data Types

```
typedef ComponentInstance MTPlayerComponent;

typedef long MTPlayerHintFlags;
```

Functions

Managing a Stream Player Component

```
pascal ComponentResult MTPlayerInit
                           (MTPlayerComponent spc,
                            MTStreamDescriptionHandle description,
                            MTReleaseUPP proc);
pascal ComponentResult MTPlayerGetPlayerInfo
                           (MTPlayerComponent spc,
                            PlayerFlags *playerFlags);
```

Getting and Setting Basic Stream Characteristics

```
pascal ComponentResult MTPlayerGetStreamDescription
                           (MTPlayerComponent spc,
                            MTStreamDescriptionHandle streamDesc);
pascal ComponentResult MTPlayerEnable
                           (MTPlayerComponent spc, Boolean enableMode);
pascal ComponentResult MTPlayerIsEnabled
                           (MTPlayerComponent spc, Boolean *enableMode);
pascal ComponentResult MTPlayerSamplesPerSecond
                           (MTPlayerComponent spc,
                            Fixed *samplesPerSecond);
pascal ComponentResult MTPlayerGetStreamTimeBase
                           (MTPlayerComponent spc, TimeBase *timebase);
pascal ComponentResult MTPlayerSetStreamTimeBase
                           (MTPlayerComponent spc, TimeBase *timebase);
```

Getting and Setting Characteristics of Visual Stream Data

```
pascal ComponentResult MTPlayerSetGWorld
                            (MTPlayerComponent spc, CGrafPtr gp,
                             GDHandle gd);
pascal ComponentResult MTPlayerSetRect
                            (MTPlayerComponent spc, Rect *r);
pascal ComponentResult MTPlayerGetRect
                            (MTPlayerComponent spc, Rect *r);
pascal ComponentResult MTPlayerSetClip
                            (MTPlayerComponent spc, RgnHandle theClip);
pascal ComponentResult MTPlayerGetClip
                            (MTPlayerComponent spc, RgnHandle *theClip);
pascal ComponentResult MTPlayerSetMatrix
                            (MTPlayerComponent spc, const MatrixRecord *m);
pascal ComponentResult MTPlayerGetMatrix
                            (MTPlayerComponent spc, MatrixRecord *m);
pascal ComponentResult MTPlayerGetSrcRgn
                            (MTPlayerComponent spc,
                             RgnHandle srcRgn, TimeValue requestTime);
pascal ComponentResult MTPlayerSetDimensions
                            (MTPlayerComponent spc, Fixed width,
                             Fixed height);
pascal ComponentResult MTPlayerGetDimensions
                            (MTPlayerComponent spc, Fixed *width,
                             Fixed *height);
pascal ComponentResult MTPlayerSetDisplayHints
                            (MTPlayerComponent spc,
                             MTPlayerHintFlags flags);
pascal ComponentResult MTPlayerGetPicture
                            (MTPlayerComponent spc, PicHandle *p,
                             const Rect *bounds, short offscreenDepth,
                             long grabPictFlags);
```

Getting and Setting Characteristics of Audio Stream Data

```
pascal ComponentResult MTPlayerSetVolume
                            (MTPlayerComponent spc, short volume);
pascal ComponentResult MTPlayerGetVolume
                            (MTPlayerComponent spc, short *volume);
pascal ComponentResult MTPlayerSetSoundBalance
                            (MTPlayerComponent spc, short balance);
pascal ComponentResult MTPlayerGetSoundBalance
                            (MTPlayerComponent spc, short *balance);
```

Playing Stream Data

```
pascal ComponentResult MTPlayerPlayStream
                            (MTPlayerComponent spc,
                             MTStreamChunkRecordPtr chunk);
```

Application-Defined Function

```
pascal void MyReleaseProc (MTStreamChunkRecordPtr chunk);
```

Result Codes

`noErr`	0	No error
`paramErr`	–50	Volume parameter is out of range
`invalidSampleDescription`	–2041	Invalid sample description
`badComponentSelector`	–32766	Function not supported by this component

Flow Control Components

Contents

This chapter describes flow control components. A **flow control component** manages the rate of flow of a given type of time-based media data, such as video or sound sequences, between other components. The flow control component API provides a standard interface for regulating media data flow rates, independent of the type of media data.

Apple provides two flow control components: a video flow control component for regulating the flow of video data and a sound flow control component for regulating the flow of sound data.

You need to read this chapter if you are developing a stream director component. Stream director components configure and manage flow control components.

If you are developing an application that provides features derived from QuickTime Conferencing components, you do not need to read this chapter. Applications do not interact with flow control components.

If you want to develop a new flow control component, you need to read this chapter to understand the API that a flow control component must support. You can create new types of flow control components to support new types of media data (such as text, H.320, or H.221 media data). You could also provide a new video or sound flow control component to regulate the flow of video or sound data in a manner different from that used by the Apple-provided flow control components.

To use or to create a flow control component, you need to be familiar with the Component Manager, described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

This chapter is written from the perspective of a stream director developer. It assumes you are familiar with the material presented in the chapter "Stream Director Components" in this book. The chapter begins with a brief introduction to flow control components. Then it discusses how you can

- open a flow control component
- initialize a flow control component
- attach other components to a flow control component
- enable and disable flow control
- transfer data to a flow control component

About Flow Control Components

Flow control components manage the rate of flow of media data between QuickTime Conferencing components while a connection exists. They seek to conserve network bandwidth and CPU cycles. They also work to ensure the best result possible on playback, despite differences in CPU performance characteristics and processing loads at the local and remote connection ends, and the network traffic load.

Flow control components are specific to a type of media data. They free other components from the details of managing smooth data flow for a given media data type

by isolating these tasks and offering a common API. The type of algorithms used by flow control components can vary widely.

Apple provides a sound flow control component and a video flow control component. These flow control components control streams containing sound media data and video media data, as defined by QuickTime (types are `SoundMediaType` and `VideoMediaType`).

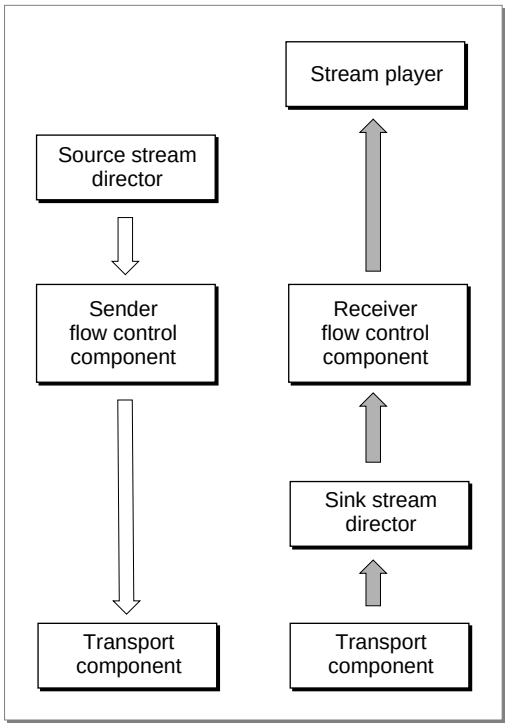
Each instance of a flow control component flow controls one stream. If that stream is an outgoing stream, the flow control component instance is referred to as a *sender flow control component*. If the stream is an incoming stream, the flow control component instance is referred to as a *receiver flow control component*.

Stream director components open, configure, manage, and close flow control components. Source stream directors attach transport component instances to sender flow control components. Sink stream directors attach stream player component instances to receiver flow control components.

When a source stream director gets media data from its media component, it passes the data to a sender flow control component, which in turn passes the data to its attached transport components.

When a sink stream director gets media data from its transport component, it passes the data to the receiver flow control component, which in turn passes the data to its attached stream player components. Figure 7-1 illustrates how data flows among flow control components, stream directors, stream players, and transport components. The white arrows represent outgoing streams; the gray arrows represent incoming streams.

Figure 7-1 Data flow among flow control components and stream directors, stream players, and transport components



Using Flow Control Components

This section discusses how to

- open and set up a flow control component
- attach other components to a flow control component
- enable and disable flow control for a stream
- transfer data to a flow control component

Opening a Flow Control Component

A stream director opens a flow control component to manage the data flow for a stream, based on the type of media data carried by that stream. For example, it opens a sound flow control component to flow control a sound stream and a video flow control component to flow control a video stream.

If you are interested in a flow control component of a given subtype and do not need to specify any other characteristics of the component, you can call the Component Manager

function `OpenDefaultComponent`. Set the first parameter to the constant `kMTFlowControlType` to specify a flow control component (value is `'flow'`). Set the second parameter to the subtype of the flow control component that you want to open. Apple defines the constants `kMTVideoSubType` (value is `'vide'`) and `kMTSoundSubType` (value is `'soun'`) for the subtypes of video and sound flow control components. For example, to open a video flow control component, you call the `OpenDefaultComponent` function like this:

```
myFlowControlInstance = OpenDefaultComponent
                  (kMTFlowControlType, kMTVideoSubType);
```

The function returns a reference to your connection to the component (that is, the flow control component instance).

If you want to open a specific flow control component, call the Component Manager `OpenComponent` function.

When a stream is torn down, a stream director should close the flow control component by calling the Component Manager `CloseComponent` function.

The `OpenDefaultComponent`, `OpenComponent`, and `CloseComponent` functions are described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Initializing a Flow Control Component

A stream director opens a flow control component after a QuickTime Conferencing connection is established. After you open a flow control component, you need to provide it with essential information that allows the flow control component to properly initialize itself. You do this by calling the `MTFlowControlInit` function (page 7-12).

The function takes a `type` parameter that tells the flow control component whether it is a receiver or a sender flow control component—that is, whether it is flow controlling an incoming or an outgoing stream. If you are initializing a sender flow control component, set this parameter to the constant `kMTFlowControlSender`; for a receiver flow control component, set it to `kMTFlowControlReceiver`.

You also need to tell the flow control component your stream director component instance—the flow control component needs to know the stream director component instance that opened it.

When you call the `MTFlowControlInit` function, you pass it a handle to a stream description structure. The `MTStreamDescription` structure contains information such as the stream ID of the stream that the flow control component instance is to manage and the type of media data carried by the stream. For a description of the `MTStreamDescription` structure, see the chapter "Stream Director Components" in this book.

Attaching Other Components to a Flow Control Component

Once you open and initialize a flow control component, you need to attach to it either a stream player component (if it's a receiver flow control component) or a transport component (if it's a sender flow control component). Attaching is the process by which you tell a flow control component where to send the data it receives from a stream director. The flow control component API provides two functions to do this.

To attach a stream player to a receiver flow control component, call the `MTFlowControlAttachPlayer` function (page 7-14) and pass in the component instance for the stream player. A receiver flow control component supports one attached stream player. If you call the `MTFlowControlAttachPlayer` function after you have already attached a stream player, the function replaces the existing stream player with the new one.

To attach a transport component to a sender flow control component, call the `MTFlowControlAttachTransport` function (page 7-13) and pass in the component instance for the transport component. A sender flow control component supports one attached transport component. If you call the `MTFlowControlAttachTransport` function after you have already attached a transport component, the function replaces the existing transport component with the new one.

Enabling and Disabling Flow Control

You can enable and disable flow control of a stream. While flow control is disabled, a flow control component forwards all the data it receives and makes no effort to regulate the rate of flow.

Flow control of a stream is disabled by default. The `MTTransportGetInfo` function (described in "Transport Components" in this book) provides information about the flow control needs of a stream. Calling this function may help you to determine if you should enable flow control.

To enable flow control, call the `MTFlowControlEnable` function (page 7-15) and set its `enableMode` parameter to `true`. To disable flow control of a stream, call `MTFlowControlEnable` and set its `enableMode` parameter to `false`.

You can check whether flow control for a stream is enabled or disabled at any time by calling the `MTFlowControlIsEnabled` function (page 7-15).

Transferring Data to a Flow Control Component

Both sender and receiver types of flow control components get their data from stream directors. Senders then forward the data to transport components. Receivers forward the data to stream players.

When a source stream director receives a chunk of data from a media component, it passes the data to the sender flow control component for that media type by calling the `MTFlowControlSndrProc` function, as shown in this example:

```
myErr = MTFlowControlSndrProc (flowInstance, chunk, async,
                                    completion, refCon);
```

The parameter `chunk` is a pointer to the chunk record structure containing the data. You set the `async` parameter to `true` or `false` depending on whether you want the function to execute asynchronously or synchronously. If you specify asynchronous execution, the `completion` parameter is a pointer to your completion routine and `refCon` is your private value.

(The sender flow control component then passes the data to its attached transport component to send the data out to the network.)

When a sink stream director receives a chunk of data from a transport component, it passes the data to the receiver flow control component for that media type by calling the `MTFlowControlRcvrProc` function, as shown in this example:

```
myErr = MTFlowControlRcvrProc (flowInstance, chunk);
```

(The receiver flow control component then passes the data to its attached stream player to play the data.)

Creating Flow Control Components

You can create a flow control component to implement your own algorithm for controlling the flow of sound or video media data, or to flow control a new type of media data. This section tells you what functions your flow control component must support and offers some thoughts on managing data flow in a flow control component.

Required Functions

Your flow control component must support these functions:

- `MTFlowControlInit`
- `MTFlowControlAttachTransport`
- `MTFlowControlAttachPlayer`
- `MTFlowControlIsEnabled`
- `MTFlowControlEnable`
- `MTFlowControlRcvrProc`
- `MTFlowControlSndrProc`
- `MTFlowControlSlowDown`
- `MTFlowControlSpeedUp`
- `MTFlowControlJoin`

If your flow control component works with streams that contain audio data, you must support these functions:

- `MTFlowControlSetVOXThreshold`
- `MTFlowControlGetVOXThreshold`

Managing Data Flow

Managing the end-to-end data flow across a QuickTime Conferencing connection is the essential task of a flow control component. In developing a flow control algorithm, you should consider such factors as network traffic load and differences in CPU performance capability at opposite ends of a connection, as well as factors that are specific to the type of media data with which you're working. For example, for sound streams, you might want to consider sound flow control algorithms.

What follows is a brief overview of a method that you can use to flow control a stream. It uses a video stream as an example.

Assume that a video conferencing application uses a QuickTime sequence grabber to generate video data. The application configures the sequence grabber with a current and a maximum frame rate. After a connection is established, the sequence grabber generates video data at the current frame rate.

The sender video flow control component at one end of the connection and the receiver video flow control component at the other end set up a feedback loop between themselves. The receiver flow control component calculates the frame rate that would give the best result on the receiving end, and it sends periodic messages about the frame rate to the sender flow control component. The sender flow control component receiving these messages calls the stream director function `MTDirectorSetFrameRate`. The stream director in turn tells the sequence grabber to adjust the rate at which it is generating video frames.

In this method, there is no buffering of data. Instead, the rate at which data is generated is adjusted up or down based on information exchanged via the feedback loop by the peer video flow control components at either end of the connection. The maximum frame rate limits how fast video data is generated, protecting a network against being flooded with video traffic.

Although the example uses a video flow control component, this method can be applied to other types of media data as well.

Flow Control Component Reference

This section describes the constants, data types, and functions that comprise the API of flow control components.

Constants

This section describes the constants in the flow control component API.

Component Type and Subtypes

All flow control components have the component type `'flow'`. Apple defines two flow control component subtypes, one for video and one for sound. The subtype of a video flow control component is equated to the QuickTime constant `VideoMediaType`; for a sound flow control component, the subtype is equated to the QuickTime constant `SoundMediaType`.

```
enum {
   kMTFlowControlType   = 'flow'
};

enum {
   kMTVideoSubType      = VideoMediaType,
   kMTSoundSubType      = SoundMediaType
};
```

Flow Control Component Request Codes

A request code specifies a function in the flow control API. The Component Manager passes the request code to a flow control component to indicate which function was called.

```
enum {
   kMTFlowControlInitSelect              = 1,
   kMTFlowControlEnableSelect            = 2,
   kMTFlowControlIsEnabledSelect         = 3,
   kMTFlowControlAttachTransportSelect   = 4,
   kMTFlowControlAttachPlayerSelect      = 5,
   kMTFlowControlRcvrProcSelect          = 6,
   kMTFlowControlSndrProcSelect          = 7,
   kMTFlowControlSlowDownSelect          = 8,
   kMTFlowControlSpeedUpSelect           = 9,
   kMTFlowControlSetVOXThresholdSelect   = 10,
   kMTFlowControlGetVOXThresholdSelect   = 11,
   kMTFlowControlJoinSelect              = 12
};
```

Data Types

This section describes the data type in the flow control component API.

The Flow Control Type

You use the `MTFlowControlType` data type to specify a flow control component as a sender type or a receiver type:

```
typedef short MTFlowControlType;
```

Apple defines the following constants to use with variables of type `MTFlowControlType`:

```
enum {
   kMTFlowControlReceiver     = 1,
   kMTFlowControlSender       = 2
};
```

Constant descriptions

`kMTFlowControlReceiver`
: Identifies a receiver type of flow control component. Receiver flow control components get their data from sink stream directors and forward it to stream players.

`kMTFlowControlSender`
: Identifies a sender type of flow control component. Sender flow control components get their data from source stream directors and forward it to transport components.

Functions

This section describes the functions provided by the flow control component API. These functions are described from the perspective of a stream director calling a flow control component. If you are developing a flow control component, your component must support the required functions as they are described here.

The functions are organized into the following sections:

- "Setting Up a Flow Control Component" describes the functions you use to initialize a flow control component and to attach it to the stream players and transport components to which it must pass data.
- "Getting and Setting Flow Control Status" describes the functions you use to enable and disable flow control of streams and to get the flow control status of a stream.
- "Transferring Data to a Flow Control Component" describes the functions you use to pass data to sender and receiver flow control components.
- "Adjusting Transmission Rates" describes the functions you use to advise a flow control component to adjust its transmission rate.

- "Joining Flow Control Components" describes the function you use to inform two local flow control components about each other.
- "Flow Controlling Sound Streams" describes the functions you use to set and get the threshold value for the sound flow control algorithm in use for a sound stream.

Setting Up a Flow Control Component

The functions described in this section allow you to initialize a flow control component and to inform it of the stream players or transport components to which it should forward media data.

MTFlowControlInit

The `MTFlowControlInit` function initializes a flow control component.

```
pascal ComponentResult MTFlowControlInit
            (MTFlowControlComponent fcc,
            MTFlowControlType type, MTDirectorComponent owner,
            MTStreamDescriptionHandle description);
```

`fcc` The flow control component you are using.

`type` The type of flow control component. Set this parameter to `kMTFlowControlReceiver` if the flow control component controls the flow of an incoming stream. Set it to `kMTFlowControlSender` if it controls an outgoing stream.

`owner` The stream director component instance that opened this flow control component and that manages the stream to be flow controlled. You provide this value.

`description`
A handle to the stream description structure that describes the stream to be flow controlled. You must provide a fully specified structure. The structure includes information such as the stream ID and stream type. The flow control component does not modify or dispose of this structure.

DESCRIPTION

You call the `MTFlowControlInit` function to initialize a flow control component to control the data flow for a particular stream.

In the `type` parameter, you tell the flow control component whether it is a sender or receiver type—that is, whether it flow controls an outgoing or an incoming stream.

You also provide your component instance—that of the stream director responsible for the stream.

The stream description structure that you provide gives the flow control component necessary information about the stream it is to control. The flow control component may copy information from the structure, but it leaves the original structure intact.

In addition to the result codes listed below, the `MTFlowControlInit` function may return result codes from the stream director component.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Wrong stream type for flow control component or invalid type parameter

SEE ALSO

The `MTStreamDescription` structure is described in the chapter "Stream Director Components" in this book.

MTFlowControlAttachTransport

The `MTFlowControlAttachTransport` function attaches a transport component to a sender flow control component.

```
pascal ComponentResult MTFlowControlAttachTransport
            (MTFlowControlComponent fcc, MTTransportComponent tc);
```

`fcc`	The flow control component you are using.
`tc`	The transport component to attach to this flow control component.

DESCRIPTION

You call the `MTFlowControlAttachTransport` function to attach a transport component to a sender flow control component (type `kMTFlowControlSender`).

Once the function completes successfully, a sender flow control component passes media data it receives from a source stream director to the transport component for transmission to the remote connection end.

A sender flow control component instance supports one attached transport component. If you call the `MTFlowControlAttachTransport` function after you've already attached a transport component to a flow control component, the flow control component replaces the old transport component with the new one.

The `MTFlowControlAttachTransport` function may return result codes from the transport component.

RESULT CODES

`noErr`	0	No error

MTFlowControlAttachPlayer

The `MTFlowControlAttachPlayer` function attaches a stream player component to a receiver flow control component.

```
pascal ComponentResult MTFlowControlAttachPlayer
            (MTFlowControlComponent fcc, MTPlayerComponent spc);
```

`fcc` The flow control component you are using.

`spc` The stream player component to attach to this flow control component.

DESCRIPTION

You call the `MTFlowControlAttachPlayer` function to attach a stream player component to a flow control component of type `kMTFlowControlReceiver`.

Once the function completes successfully, a receiver flow control component passes media data it receives from a sink stream director to the stream player component for playback.

A receiver flow control component instance can have only one attached stream player. If you call the `MTFlowControlAttachPlayer` function after you've already attached a stream player to a flow control component, the flow control component replaces the old stream player with the new one.

The `MTFlowControlAttachPlayer` function may return result codes from the stream player component.

RESULT CODES

`noErr`	0	No error

Getting and Setting Flow Control Status

The functions described in this section allow you to set and get the flow control status of a stream.

MTFlowControlEnable

The `MTFlowControlEnable` function enables or disables flow control for a stream.

```
pascal ComponentResult MTFlowControlEnable
            (MTFlowControlComponent fcc, Boolean enableMode);
```

`fcc` The flow control component you are using.

`enableMode` A Boolean value that indicates whether you want to enable or disable flow control. To enable flow control of the stream assigned to the flow control component, set this parameter to `true`. To disable flow control of the stream, set it to `false`.

DESCRIPTION

You call the `MTFlowControlEnable` function to enable or disable flow control of the stream assigned to the flow control component that you specify.

When you disable flow control, the flow control component continues to forward the data, but it stops making any attempt to regulate the rate of flow.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can get information about the flow control needs of a stream by calling the `MTTransportGetInfo` function, described in the chapter "Transport Components" in this book.

MTFlowControlIsEnabled

The `MTFlowControlIsEnabled` function tells you whether flow control is enabled or disabled for a flow control component that you specify.

```
pascal ComponentResult MTFlowControlIsEnabled
            (MTFlowControlComponent fcc, Boolean *enableMode);
```

`fcc` The flow control component you are using.

`enableMode` A pointer to a Boolean value that indicates whether flow control is enabled or disabled for the stream assigned to the flow control component. The function returns `true` if flow control is enabled. It returns `false` if flow control is disabled.

DESCRIPTION

You call the `MTFlowControlIsEnabled` function to determine whether flow control for a stream is enabled or disabled.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You enable or disable flow control for a stream by calling the `MTFlowControlEnable` function, described on page 7-15.

Transferring Data to a Flow Control Component

You use the functions described in this section to pass media data to a flow control component.

MTFlowControlSndrProc

The `MTFlowControlSndrProc` function passes data to a sender flow control component.

```
pascal ComponentResult MTFlowControlSndrProc
            (MTFlowControlComponent fcc,
            MTStreamChunkRecordPtr chunk, Boolean async,
            MTCompletionUPP completion, long refCon);
```

`fcc`	The flow control component you are using.
`chunk`	A pointer to a chunk record structure that contains the media data to be sent to the remote connection end.
`async`	A Boolean value that indicates whether you want the function to execute asynchronously. Set this to `true` if you do. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. The function calls this routine when an asynchronous operation completes. If you want to modify or dispose of the chunk record structure you pass to the function, you must provide a completion routine so that you can tell when it is safe to do so. If you don't provide a completion routine, set this parameter to `nil`. The function ignores this parameter if you call it synchronously.
`refCon`	Reserved for your use. The flow control component passes this value when it calls your completion routine. Set this to `nil` if you are not providing a completion routine.

DESCRIPTION

A source stream director calls the `MTFlowControlSndrProc` function to pass data to a sender flow control component. It calls the function when it receives data from a media component, such as a QuickTime sequence grabber component.

If you call this function asynchronously, you must not modify or dispose of the chunk record structure that you pass to the function until execution is complete. To find out when asynchronous execution is complete, you must provide a completion routine.

The flow control component forwards the data to its attached transport component for transmission to the remote connection end by calling the transport component's `MTTransportSendMediaData` function.

In addition to the result codes listed below, the `MTFlowControlSndrProc` function may return result codes from a transport component or a QuickTime sequence grabber component.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Flow control component is not sender type
`mtNoTransportErr`	–7910	No transport instance attached

SEE ALSO

The chunk record structure (`MTStreamChunkRecord`) is described in the chapter "Transport Components" in this book.

You attach transport components to a flow control component by calling the `MTFlowControlAttachTransport` function, described on page 7-13.

The `MTTransportSendMediaData` function is described in the chapter "Transport Components" in this book.

MTFlowControlRcvrProc

The `MTFlowControlRcvrProc` function passes data to a receiver flow control component.

```
pascal ComponentResult MTFlowControlRcvrProc
            (MTFlowControlComponent fcc,
            MTStreamChunkRecordPtr chunk);
```

`fcc` The flow control component you are using.

`chunk` A pointer to a chunk record structure that contains the media data to be passed to a stream player for playback.

DESCRIPTION

A sink stream director calls the `MTFlowControlRcvrProc` function to pass data to a receiver flow control component. It calls the function when it receives data from its attached transport component.

The flow control component forwards the data to its attached stream player component for playback by calling the `MTPlayerPlayStream` function.

In addition to the result codes listed below, the `MTFlowControlRcvrProc` function may return result codes from a transport component or a stream player component.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Flow control component is not receiver type

SEE ALSO

The chunk record structure (`MTStreamChunkRecord`) is described in the chapter "Transport Components" in this book.

The `MTPlayerPlayStream` function is described in the chapter "Stream Player Components" in this book.

You attach a stream player component to a flow control component by calling the `MTFlowControlAttachPlayer` function, described on page 7-14.

Adjusting Transmission Rates

The functions described in this section allow you to advise a flow control component to adjust its data transmission rate.

MTFlowControlSlowDown

The `MTFlowControlSlowDown` function advises a flow control component that it should reduce the rate of data flow if it can.

```
pascal ComponentResult MTFlowControlSlowDown
                        (MTFlowControlComponent fcc);
```

`fcc` The flow control component you are using.

DESCRIPTION

You call the `MTFlowControlSlowDown` function to inform a flow control component that an external event has occurred that may affect its data flow algorithm.

For example, a stream director might call `MTFlowControlSlowDown` after the application moves into the background.

The flow control component then takes whatever actions are appropriate, given the mechanism it uses to control data flow. For example, it might decrease its transmission rate.

The `MTFlowControlSlowDown` function may return result codes from a transport component.

RESULT CODES

`noErr`	0	No error

MTFlowControlSpeedUp

The `MTFlowControlSpeedUp` function advises a flow control component that it should increase the rate of data flow if it can.

```
pascal ComponentResult MTFlowControlSpeedUp
                            (MTFlowControlComponent fcc);
```

`fcc` The flow control component you are using.

DESCRIPTION

You call the `MTFlowControlSpeedUp` function to inform a flow control component that an external event has occurred that may affect its data flow algorithm.

For example, a stream director might call `MTFlowControlSpeedUp` after the application moves into the foreground.

The flow control component then takes whatever actions are appropriate, given the mechanism it uses to control data flow. For example, it might increase its data transmission rate.

The `MTFlowControlSpeedUp` function may return result codes from a transport component.

RESULT CODES

`noErr`	0	No error

Joining Flow Control Components

The function described in this section allows you to inform two local flow control components about the existence of the other.

MTFlowControlJoin

The `MTFlowControlJoin` function informs two flow control components that use the same transport component about each other.

```
pascal ComponentResult MTFlowControlJoin
            (MTFlowControlComponent fcc,
            MTFlowControlComponent otherfcc);
```

`fcc` The flow control component instance you are using.

`otherfcc` The component instance of the other local flow control component using the same transport-level connection. If the `fcc` parameter contains the instance of a sender flow control component, then this parameter contains the instance of a receiver flow control component, and vice versa.

DESCRIPTION

One of the flow control components that you specify must be a sender (type `kMTFlowControlReceiver`) and the other must be a receiver (type `kMTFlowControlSender`). The order in which you specify the flow control components does not matter.

You need to call this function for sound flow control components so that a sender sound flow control component has access to the receiver sound flow control component's information, and vice versa. A sound flow control component may need this information when it is using certain sound flow control techniques.

Whether you need to join other types of flow control components depends on the type of media data with which they work. You can call `MTFlowControlJoin` in all cases; if the components do not need to be aware of each other, the function returns `noErr`.

RESULT CODES

`noErr`	0	No error

Flow Controlling Sound Streams

The functions described in this section allow you to set and get a threshold value for the sound flow control algorithm in use for a given sound stream.

MTFlowControlSetVOXThreshold

The `MTFlowControlSetVOXThreshold` function sets a threshold value for a sound flow control algorithm used by a sender sound flow control component that you specify.

```
pascal ComponentResult MTFlowControlSetVOXThreshold
            (MTFlowControlComponent fcc, short threshold);
```

`fcc` The sender sound flow control component you are using.

`threshold` The threshold value you want to set. The threshold level is expressed as a 16-bit fixed-point number in the range –1.0 to 1.0. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part.

DESCRIPTION

Typically, a stream director calls `MTFlowControlSetVOXThreshold` after its `MTDirectorSetSoundThreshold` function is called.

SPECIAL CONSIDERATIONS

Once you set the threshold value and while flow control is enabled, the Apple-provided sound flow control component examines the sound level of each chunk of sound data that you forward to it. When the sound level exceeds the threshold value, the sound flow control component forwards the data to a transport component for transmission to the remote connection end.

If flow control is disabled, the Apple-provided sound flow control component ignores the threshold value and forwards all data to its attached transport components.

Only sender sound flow control components support this function. Receiver flow control components return the `paramErr` result code.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Flow control component not sender type

SEE ALSO

The `MTDirectorSetSoundThreshold` function is described in the chapter “Stream Director Components” in this book.

You can check the enable status of flow control by calling the `MTFlowControlIsEnabled` function, described on page 7-15.

MTFlowControlGetVOXThreshold

The `MTFlowControlGetVOXThreshold` function gets the threshold value for a sound flow control algorithm for a sound stream.

```
pascal ComponentResult MTFlowControlGetVOXThreshold
            (MTFlowControlComponent fcc, short *threshold);
```

`fcc` The flow control component you are using.

`threshold` A pointer to the threshold value. The functions returns the threshold value for a sound flow control algorithm. The threshold level is expressed as a 16-bit fixed-point number in the range –1.0 to 1.0. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part.

DESCRIPTION

Typically, a stream director calls `MTFlowControlGetVOXThreshold` after its `MTDirectorGetSoundThreshold` function is called.

SPECIAL CONSIDERATIONS

Only sender sound flow control components support this function.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Flow control component not sender type

SEE ALSO

The `MTDirectorGetSoundThreshold` function is described in the chapter "Stream Director Components" in this book.

Summary of Flow Control Components

C Summary

Constants

```
enum {         /* flow control component type */
   kMTFlowControlType   = 'flow'
};

enum {         /* flow control component subtype */
   kMTVideoSubType      = VideoMediaType,
   kMTSoundSubType      = SoundMediaType
};

enum {         /* flow control component request codes */
   kMTFlowControlInitSelect               = 1,
   kMTFlowControlEnableSelect             = 2,
   kMTFlowControlIsEnabledSelect          = 3,
   kMTFlowControlAttachTransportSelect    = 4,
   kMTFlowControlAttachPlayerSelect       = 5,
   kMTFlowControlRcvrProcSelect           = 6,
   kMTFlowControlSndrProcSelect           = 7,
   kMTFlowControlSlowDownSelect           = 8,
   kMTFlowControlSpeedUpSelect            = 9,
   kMTFlowControlSetVOXThresholdSelect    = 10,
   kMTFlowControlGetVOXThresholdSelect    = 11,
   kMTFlowControlJoinSelect               = 12
};

enum {         /* values for MTFlowControlType */
   kMTFlowControlReceiver  = 1,
   kMTFlowControlSender    = 2
};
```

Data Types

```
typedef ComponentInstance MTFlowControlComponent;
```

```
typedef short MTFlowControlType;
```

Functions

Setting Up a Flow Control Component

```
pascal ComponentResult MTFlowControlInit
                            (MTFlowControlComponent fcc,
                             MTFlowControlType type,
                             MTDirectorComponent owner,
                             MTStreamDescriptionHandle description);
pascal ComponentResult MTFlowControlAttachTransport
                            (MTFlowControlComponent fcc,
                             MTTransportComponent tc);
pascal ComponentResult MTFlowControlAttachPlayer
                            (MTFlowControlComponent fcc,
                             MTPlayerComponent spc);
```

Getting and Setting Flow Control Status

```
pascal ComponentResult MTFlowControlEnable
                            (MTFlowControlComponent fcc,
                             Boolean enableMode);
pascal ComponentResult MTFlowControlIsEnabled
                            (MTFlowControlComponent fcc,
                             Boolean *enableMode);
```

Transferring Data to a Flow Control Component

```
pascal ComponentResult MTFlowControlSndrProc
                            (MTFlowControlComponent fcc,
                             MTStreamChunkRecordPtr chunk, Boolean async,
                             MTCompletionUPP completion, long refCon);
pascal ComponentResult MTFlowControlRcvrProc
                            (MTFlowControlComponent fcc,
                             MTStreamChunkRecordPtr chunk);
```

Adjusting Transmission Rates

```
pascal ComponentResult MTFlowControlSlowDown
                            (MTFlowControlComponent fcc);
pascal ComponentResult MTFlowControlSpeedUp
                            (MTFlowControlComponent fcc);
```

Joining Flow Control Components

```
pascal ComponentResult MTFlowControlJoin
                            (MTFlowControlComponent fcc,
                             MTFlowControlComponent otherfcc);
```

Flow Controlling Sound Streams

```
pascal ComponentResult MTFlowControlSetVOXThreshold
                            (MTFlowControlComponent fcc, short threshold);
pascal ComponentResult MTFlowControlGetVOXThreshold
                            (MTFlowControlComponent fcc, short *threshold);
```

Result Codes

`noErr`	0	No error
`paramErr`	–50	Wrong stream type for flow control component or invalid flow control component type
`mtNoTransportErr`	–7910	No transport instance attached

Recorder Components

Contents

This chapter describes recorder components. **Recorder components** allow applications to capture the time-based media data as it is presented to a user. The recorder component provided by Apple stores captured media data into QuickTime movie files. Recorder components provided by third parties may store the media data in other types of files or redirect the data for further processing without storing the data.

If your application uses the Apple-provided conference component, you automatically get basic recording features in your application. The conference component manages stream controller components, which provide a user interface for making recordings. You need to read this chapter if you want to directly control the recording process in your application or record to a disk volume other than the system volume.

If you want to implement application-specific user interface features and still take advantage of the recording services provided through the conference component, you'll need to be familiar with the recorder component API covered in this chapter and the stream controller component's action filter function mechanism. The action filter function alerts your application to user actions before they are acted upon by the stream controller component, thereby giving you the opportunity to respond to those actions with application-specific features. Conference components and stream controller components are described in the chapters "Conference Component" and "Stream Controller Components" in this book.

This chapter begins with a brief introduction to recorder components. Then it discusses how you can

- specify the sources of the media data that you want to record
- specify a destination file in which to store the data
- start, stop, and pause the recording at any time
- query the pause state of a recording
- set a maximum recording time for a recording
- query certain characteristics of a recording, such as its maximum recording time, the remaining recording time, the QuickTime movie resource, the last movie resource ID, and the remaining storage space on a volume

If you are creating a recorder component, you need to support the API described in this chapter.

To use or to create a recorder component, you need to be familiar with the Component Manager, described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

About Recorder Components

Your application can use a recorder component to store one or more streams of media data into a file. The data can be played back by some other means at a later time.

A **stream** is a sequence of time-based media data, such as video or sound, that is carried over a QuickTime Conferencing connection. Each stream is composed of a single type of media data.

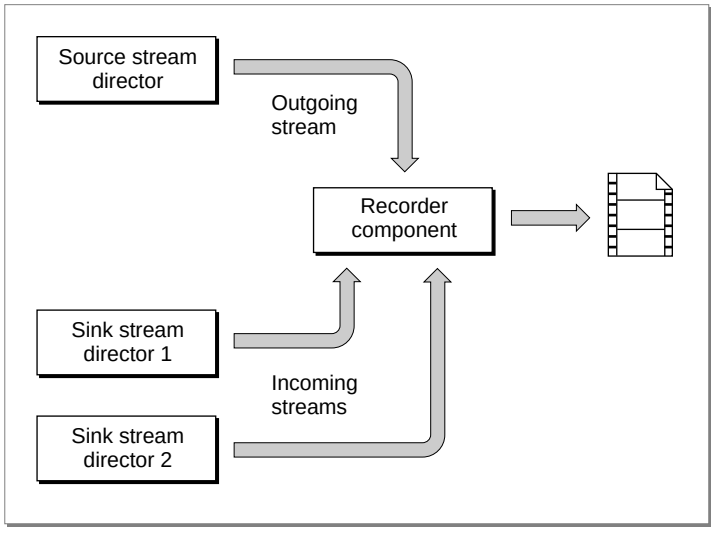
A stream whose data flows from the remote end of a connection to the local computer is referred to as an **incoming stream**; one whose data flows from the local computer to a remote end of a connection is referred to as an **outgoing stream**. Using a recorder component, you can record any number of incoming and outgoing streams simultaneously.

For example, if you are writing a video conferencing application, you can use recorder components to allow the user to record all sides of a multiparty conference call. You can record the digitized output of a locally attached camera and microphone, as well as audio and visual data coming from other conference participants.

When your application wants to record media data, it needs to tell the recorder component where to get the data. Stream director components serve as sources of media data for recorder components. There are two types of stream director components. A **source stream director** manages outgoing streams. It acquires media data from some source (for example, a QuickTime sequence grabber) and forwards the media data to the remote connection end. A **sink stream director** manages incoming streams. It receives media data from a remote connection end and displays it locally.

Do not confuse sources of media data for a recorder component with source stream directors. Both source stream directors and sink stream directors can be sources of media data for a recorder component. For example, you can record all sides of a three-party video conference. Figure 8-1 illustrates such an example.

Figure 8-1 Recording a three-way video conference



Recorder components have the component type `'recd'`, defined by the constant `kMTRecorderType`. Apple provides the component subtype `'moov'`, defined by the constant `kMTMovieSubType`, for recorder components that write data to QuickTime movie files. Third parties may provide recorder components that write to different file types. These recorder components should have a distinguishing subtype.

Using Recorder Components

This section shows how you can

- open a recorder component
- set up a recorder component
- make a recording
- query a recorder component for status information

Opening a Recorder Component

You need to open a component before you can use its services. To open a recorder component, call the Component Manager function `OpenDefaultComponent` as shown below. Set the first parameter to the constant `kMTRecorderType` to specify a recorder component (value is `'recd'`). Set the second parameter to the subtype of the component you want to open, typically to `kMTMovieSubType`. The function returns a reference to your connection to the component (that is, the recorder component instance).

```
myRecorderInstance = OpenDefaultComponent (kMTRecorderType,
                                kMTMovieSubType);
```

If you want to open a specific recorder component, call the Component Manager `OpenComponent` function.

When you are done using a component, close it by calling the `CloseComponent` function. The `OpenDefaultComponent`, `OpenComponent`, and `CloseComponent` functions are described in the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox*.

Setting Up a Recorder Component

Before you can actually record any media data, you have to set up a recorder component. Specifically, you need to identify the sources of the media data you want to record and a destination file to hold the recording.

Identifying Media Data Sources

You identify a source for a recorder component by calling the `MTRecorderAddSource` function (page 8-19) and specifying a stream director component instance that will provide media data to be recorded. A recorder component records media data from all of the streams managed by a stream director that you specify as a source.

In the simplest case, you can record one direction of a point-to-point connection. For example, to record the remote view, you need to call `MTRecorderAddSource` only once and specify your sink stream director component instance.

But you can also fully record a multiparty conference call by calling the `MTRecorderAddSource` function multiple times, each time specifying a different source until you have added the local source stream director and *n* sink stream directors, one for each party in the conference. With the Apple-provided recorder component, each stream becomes a track in a multitrack movie file.

Note
Performance of the recording operation is inversely proportional to the number of streams you record simultaneously. It's also dependent on the codec in use, the network traffic load, and CPU performance characteristics. ◆

You need to specify at least one source before you start to record (otherwise, nothing gets recorded). However, you can add new sources of media data for a recording after you've started to record.

If you need to know what sources you've set for a recorder component, you can retrieve them. Each time you call `MTRecorderAddSource`, a recorder component adds the stream director instance to a list of sources that it maintains. To find out what is on the list, call the `MTRecorderGetNextSource` function (page 8-21).

Listing 8-1 illustrates how you can get all of the media sources for a recorder component. To get the first source on the recorder's list, set the `sourceInstance` parameter to `nil`. The function returns the first source on its list in `sourceInstance`. After that, you can get each source on the list by calling `MTRecorderGetNextSource` repeatedly, each time setting `sourceInstance` to the source instance the function returned in the previous call. When you reach the end of the list, the function returns `nil` in `sourceInstance`.

Listing 8-1 Getting the sources of media data for a recording

```
void MyGetSources(gRecorderInstance)
   MTRecorderComponent  gRecorderInstance;
{
   ComponentInstance    sourceInstance;
   ComponentResult      myErr;

   sourceInstance = nil;
```

```
   do {
      myErr = MTRecorderGetNextSource (gRecorderInstance,
               &sourceInstance);

      /* handle this stream director source instance */

   } while (sourceInstance != nil);
}
```

You can delete a stream director component instance as a source of data for a recording by calling the `MTRecorderDeleteSource` function. You need to delete a source if it becomes invalid during a recording. Suppose, for example, that you are recording a multiparty conference call and one of the parties hangs up. You need to delete the stream director instance associated with that party from the list of sources being recorded. Otherwise, you are likely to encounter serious errors when you stop the recording. You need to call the `MTRecorderDeleteSource` function before you close the deleted stream director component.

You can delete all the sources at once—that is, cause the recorder to clear its list of sources—by calling the `MTRecorderAddSource` function and setting the `sourceInstance` parameter to `nil`.

Identifying the Destination File

A recorder component stores the data it records in a file. You tell it which file to record the data in by calling the `MTRecorderSetDataOutput` function and providing a file system specification.

The Apple-provided recorder component writes its data to a QuickTime movie file. Recorder components provided by others may write to a different type of file.

It's possible that third-party recorder components could redirect media data to another component for processing, rather than writing the data to a file. If your application uses such a component, you should set the file system specification to `nil` when you call the `MTRecorderSetDataOutput` function.

You also provide a set of flags that control certain aspects of the recorder component's operation. By setting the flags, you can control whether a recorder

- writes data directly to disk as it receives it or stores the data in memory until the recording is complete and then writes it to disk all at once.
- uses temporary memory during a recording. (By default, a recorder component uses temporary memory.)
- appends the recorded data to a destination movie file and creates a new movie resource in the file or deletes the movie file if it exists and creates a new file and movie resource. (By default, a recorder component that writes to QuickTime movie files deletes the movie file if it exists and creates a new file containing one movie and the corresponding movie resource.)

- adds a movie resource to a destination movie file. (By default, a recorder component creates a new movie resource and adds it to the movie file.)
- writes recorder data to a destination file.

These flags are named and described in the description of the `MTRecorderSetDataOutput` function beginning on page 8-16. (The flags are the same as those used by the `SGSetDataOutput` function, which is described in *Inside Macintosh: QuickTime Components*.) To set the destination file and the settings of the flags for a recording, call the `MTRecorderSetDataOutput` function like this:

```
myErr = MTRecorderSetDataOutput (recorderInstance,
                                 movieFile, whereFlags);
```

The parameter `recorderInstance` specifies the recorder component you are using, `movieFile` is the file system specification for the destination file, and `whereFlags` contains the control flags for the recording operation.

To find out the destination file and the settings of the flags for a recording, you call the `MTRecorderGetDataOutput` function (page 8-18).

If no recording is in progress when you call `MTRecorderGetDataOutput`, the function returns the destination file and the control flags for the last recording you made.

If you have not previously called the `MTRecorderSetDataOutput` function to set the destination file and flags, `MTRecorderGetDataOutput` returns the `noMovieFound` result code.

Making a Recording

Once you specify at least one source of media data and a destination file for the recording operation, you can start to record.

You start a recording by calling the `MTRecorderStartRecord` function. If you haven't previously specified a data source, `MTRecorderStartRecord` does *not* return an error, but nothing gets recorded, so be sure to set a source.

When you want to stop recording, call the `MTRecorderStopRecord` function. (The `MTRecorderStartRecord` and `MTRecorderStopRecord` functions are described on page 8-22 and page 8-23, respectively.)

Listing 8-2 illustrates how you would set up to record both sides of a point-to-point QuickTime Conferencing connection, and then start and stop the recording.

Listing 8-2 Setting up and making a recording

```
void MyDoRecording(movieFile, whereFlags)
   FSSpec   *movieFile;
   long     whereFlags;

{
```

```
    MTRecorderComponent  recorderInstance;
    ComponentInstance    sourceInstance;
    ComponentResult      myErr;

    /* set local recorder instance to global recorder instance */
    recorderInstance = gRecorderInstance;

    /* set source stream director as media data source */
    sourceInstance = gSourceStreamDirectorInstance;
    myErr = MTRecorderAddSource (recorderInstance, sourceInstance);

    /* set sink stream director as media data source */
    sourceInstance = gSinkStreamDirectorInstance;
    myErr = MTRecorderAddSource (recorderInstance, sourceInstance);

    myErr = MTRecorderSetDataOutput (recorderInstance,
                                     movieFile, whereFlags);

    myErr = MTRecorderStartRecord (recorderInstance);

    /* allow recording of data until time or space limits are
       reached or user indicated recording should stop */

    myErr = MTRecorderStopRecord (recorderInstance);

}
```

Pausing and Resuming a Recording

You can suspend and resume a recording operation at any time by calling the `MTRecorderPause` function (page 8-25) like this:

```
myErr = MTRecorderPause(recorderInstance, pause);
```

This function actually sets the pause mode or state of the recorder component, which then affects the recorder's behavior.

The `MTRecorderPause` function takes a Boolean value in its `pause` parameter. If you set `pause` to `true` and a recording operation is in progress, a recorder component temporarily stops recording media data. If you set `pause` to `false` and a recording operation is currently paused, a recorder component resumes recording media data.

The `MTRecorderPause` function does not release any system resources or temporary memory associated with the recording operation when you call it. As a result, it is typically much faster to suspend and resume recording using `MTRecorderPause` than it is to use the `MTRecorderStopRecord` and `MTRecorderStartRecord` functions for the same purpose.

If a recording is *not* in progress when you call `MTRecorderPause`, the recorder component simply sets its pause mode depending on the value you set in the `pause` parameter. If you set `pause` to `true`, the recorder puts itself in pause mode. If you set `pause` to `false`, the recorder takes itself out of pause mode.

You can't record while the recorder component is in pause mode. So if you put the recorder into pause mode when you are not recording and later you want to record, you need to call both the `MTRecorderStartRecord` function to start recording and the `MTRecorderPause` function, with `pause` set to `false`, to take the recorder component out of pause mode. It does not matter which function you call first.

You can determine the pause mode of a recorder component by calling the `MTRecorderGetPause` function (page 8-26). If the recorder component is paused, the function returns `true` in its `paused` parameter. Otherwise, it returns `false`.

Advising of Stream Format Changes

When a user resizes a window or changes settings that affect visual and audio data, the format of the media data in a stream changes.

A recorder component needs to know about stream format changes so that it can update its internal operations in a timely manner and continue to record media data from the affected streams without data loss. It also needs to know when streams are added or deleted.

To inform a recorder component about stream additions or deletions and about changes to the format of streams it is recording, call the `MTRecorderChangedStreams` function (page 8-24). If your application uses a stream controller component, you need to call `MTRecorderChangedStreams` when your action filter function is called with the `mtControllerActionStreamsChanged` action. (See the chapter "Stream Controller Components" for information about action filter functions and the actions they can receive.)

Monitoring Recording Status

The recorder component API provides a number of functions that allow you to monitor and control the conditions of a recording operation. You can limit the duration of a recording and monitor the amount of time and the amount of storage space remaining for a recording.

If you are using a recorder component that writes the data to a QuickTime movie file, you can call functions that provide you with the movie identifier and the most recently used movie resource ID.

Setting and Monitoring Time Limits

Your application can set a maximum time for a recording. You can do this simply to limit the length of a given recording. Or, because media data consumes a large amount of disk space, you can set a time limit to restrict the amount of disk space used in a recording operation.

To set a time limit on a recording, call the `MTRecorderSetMaximumRecordTime` function (page 8-27), as shown in this example:

```
myErr = MTRecorderSetMaximumRecordTime (recorderInstance, ticks);
```

In the `ticks` parameter, you specify the maximum amount of time to record data into the destination file in system ticks (60ths of a second).

You can retrieve the maximum recording time that was previously set by calling the `MTRecorderGetMaximumRecordTime` function.

To find out how much time is left for a given recording, call the `MTRecorderGetTimeRemaining` function (page 8-29). The function subtracts the amount of time the recording has been in progress from the maximum recording time you previously set and tells you how much time remains for the recording.

Monitoring Available Storage Space

The `MTRecorderGetStorageSpaceRemaining` function (page 8-28) tells you how much storage space, in bytes, remains available on the device that holds a recording's destination file.

The `MTRecorderGetStorageSpaceRemaining` function is useful in many situations, including the following:

- If you know the likely size of a destination file, you can check how much space is available before you start recording.
- If you need to keep a certain amount of disk space free, you can check periodically during the recording to see if you are approaching that limit.
- If you provide feedback to the user, such as a progress indicator showing the amount of space consumed by a recording, you can check periodically to keep the display up to date.

Checking for Recording Errors

An application should check for recorder component errors by calling the Component Manager's `GetComponentInstanceError` function in its event loop. In this way, you can detect if an error, such as running out of disk space, occurs during the recording process.

The Component Manager is documented in *Inside Macintosh: More Macintosh Toolbox*.

Creating Recorder Components

The sections that follow discuss topics you need to know about if you are writing a recorder component. If your application simply uses the services of a recorder component, you do not need to be familiar with the information in these sections.

Your recorder component must support the following functions in the recorder API:

- `MTRecorderAddSource`
- `MTRecorderGetNextSource`
- `MTRecorderSetDataOutput`
- `MTRecorderGetDataOutput`
- `MTRecorderStartRecord`
- `MTRecorderStopRecord`
- `MTRecorderChangedStreams`
- `MTRecorderPause`
- `MTRecorderGetPause`
- `MTRecorderSetMaximumRecordTime`
- `MTRecorderGetMaximumRecordTime`
- `MTRecorderGetStorageSpaceRemaining`
- `MTRecorderGetTimeRemaining`

In addition, if your recorder component writes its data to a QuickTime movie file, you must support these functions:

- `MTRecorderGetMovie`
- `MTRecorderGetLastMovieResID`

You can use the constants listed on page 8-15 to refer to the request codes for the functions that your recorder component supports.

See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for information on how to create any component.

Installing a Recording Function

Applications specify stream directors as sources of media data for recorder components. When an application calls the `MTRecorderAddSource` function, your recorder component needs to install a recording callback function on the specified stream director.

It does this by calling the `MTDirectorSetRecordProc` function, as shown in this example:

```
myErr = MTDirectorSetRecordProc(streamDirector, proc, refCon);
```

The `proc` parameter is a universal procedure pointer to your recording callback function; the `refCon` parameter is reserved for your use. When a stream director calls your function, it passes your reference constant in the `chunkRefCon` field in the chunk record structure.

Your recording callback function must have the following form:

```
pascal OSErr MyRecordProc (MTStreamChunkRecordPtr chunk);
```

The `chunk` parameter is a pointer to a chunk record structure that defines a chunk of media data. See the description of a recording function on page 8-32 for more complete information on this function.

Once your recorder component installs a recording function, a stream director calls it whenever it has media data. However, you should not actually record any data until the application calls the `MTRecorderStartRecord` function.

When an application calls the `MTRecorderStopRecord` function, your recorder component should call the `MTDirectorSetRecordProc` function again, this time setting the `proc` parameter to `nil`, to remove its recording function.

The `MTDirectorSetRecordProc` function is described in the chapter "Stream Director Components" in this book.

Getting Processor Time

A recorder component needs periodic processor time to perform its tasks, such as writing media data to a destination file. To make sure it gets that time, a recorder component should use the Apple-provided idler component.

The idler component provides a convenient method to get periodic processing time. Furthermore, it provides the time when it is safe to move memory. This allows your recorder component to call Memory Manager functions and Toolbox functions that might move memory.

Your recorder component should register a callback function with the idler component by calling the idler component's `MTIdlerGimmeTime` function. You provide the address of the function to be called periodically, the minimum interval that should elapse between calls to that function, and a reference constant reserved for your use. The idler component thereafter calls your function periodically at a time it is safe to make Memory Manager requests.

For more information on how to use the idler component and the `MTIdlerGimmeTime` function, see the chapter "Utility Components" in this book.

Managing Memory

If your recorder component dynamically allocates and releases memory, it should use the services of the Apple-provided memory component. Because some types of media data, such as audio data, arrive at interrupt time and must be buffered until they can be written to disk, your recorder component is likely to need those services.

The memory component allows for more efficient sharing of memory in the application heap than is likely if each QuickTime Conferencing component independently uses the Memory Manager. It does this by

- managing a common pool of buffers in an application heap and making the memory available to you at interrupt time. (The buffers are guaranteed to be in physical memory—they are not paged out as part of virtual memory management operations.)
- allowing each component instance serving an application and the application itself to register a grow-zone function.

Your recorder component first needs to open and initialize its connection with the memory component by calling the Component Manager's `OpenDefaultComponent` function and the `MTMemoryInit` function.

After that, a recorder component should call the `MTMemoryGetOneChunk` and `MTMemoryPutOneChunk` functions to get and release memory as needed. You can call these functions at interrupt time.

The memory component is optimized for handling relatively large amounts of data, like that needed for chunks of media data. You should not use it for getting and releasing small buffers.

You should register a grow-zone function with the memory component because it allows for calling multiple grow-zone functions. This increases the chances of finding needed memory under low-memory conditions.

For complete information on how to use the memory component, see the chapter "Utility Components" in this book.

Handling Stream Format Changes

An application calls the `MTRecorderChangedStreams` function to inform your recorder component that stream formats have changed.

To continue to record data without error, you may need to update your internal data structures to reflect new streams, deleted streams, or changed characteristics of existing streams. You can get the latest stream descriptions from a stream director. First, call the `MTDirectorGetNextStream` function to get stream IDs for all the streams managed by that stream director. Then call the `MTDirectorGetStreamDescription` function for each stream to get the stream description.

The `MTDirectorGetNextStream` and `MTDirectorGetStreamDescription` functions are described in the chapter "Stream Director Components" in this book.

Recorder Components Reference

This section describes the constants and functions that comprise the API for recorder components.

Constants

This section describes the constants in the recorder component API.

Component Type and Subtypes

All recorder components have the component type `'recd'`. Apple defines a recorder component subtype constant for those recorder components that write their data to QuickTime movie files. The constant `kMTMovieSubType` is equated to the QuickTime constant `MovieFileType`.

```
enum {
   kMTRecorderType      = 'recd'
};

enum {
   kMTMovieSubType      = MovieFileType
};
```

Request Codes

A request code specifies a function in the recorder component API. The Component Manager passes the request code to a recorder component to indicate which function an application called.

```
enum {
   kMTRecorderAddSourceSelect                  = 1,
   kMTRecorderDeleteSourceSelect               = 2,
   kMTRecorderGetNextSourceSelect              = 3,
   kMTRecorderSetDataOutputSelect              = 4,
   kMTRecorderGetDataOutputSelect              = 5,
   kMTRecorderStartRecordSelect                = 6,
   kMTRecorderStopRecordSelect                 = 7,
   kMTRecorderPauseSelect                      = 8,
   kMTRecorderGetPauseSelect                   = 9,
   kMTRecorderChangedStreamsSelect             = 10,
   kMTRecorderGetMovieSelect                   = 11,
   kMTRecorderSetMaximumRecordTimeSelect       = 12,
   kMTRecorderGetMaximumRecordTimeSelect       = 13,
   kMTRecorderGetStorageSpaceRemainingSelect   = 14,
   kMTRecorderGetTimeRemainingSelect           = 15,
   kMTRecorderGetLastMovieResIDSelect          = 16
};
```

Functions

This section describes the functions that are provided by recorder components. These functions are described from the perspective of one who calls a recorder component. This can be an application. Often, it is a stream controller component. To use a recorder

component, you call the functions as described here. If you are developing a recorder component, your component must support the functions as they are described here.

This section describes the following groups of functions:

- "Setting Up a Recording" describes the functions that you use to get and set both the destination file in which to store the recorded data and the one or more stream director instances that provide the data to be recorded.
- "Making a Recording" describes the functions that you use to start, stop, and pause a recording and to query the pause state of a recorder component.
- "Setting and Getting Recording Characteristics" describes the functions that you use to get certain characteristics of a recording, such as the maximum recording time, the recording time and storage space remaining, the movie identifier of a destination movie file, and the last movie resource ID used by a recorder component.

All of the functions take a recorder component instance parameter. You can obtain a component instance by calling the Component Manager's `OpenDefaultComponent` function. See page 8-5 for a description of how to call this function.

Setting Up a Recording

You use the functions described in this section to set and get a recorder component's sources of media data and the destination file for a recording.

MTRecorderSetDataOutput

The `MTRecorderSetDataOutput` function specifies a destination file for a recording and flags that control the record operation.

```
pascal ComponentResult MTRecorderSetDataOutput
         (MTRecorderComponent rc, FSSpec *movieFile,
         long whereFlags);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`movieFile` A pointer to the file system specification of a destination file for a recording. A recorder component stores the data it obtains from its sources in this file. The Apple-provided recorder component stores data in a QuickTime movie file. If you are using a recorder component that does not write the data to a file, set this parameter to `nil`.

`whereFlags` Flags that control the record operation. You must set either the `seqGrabToDisk` flag or the `seqGrabToMemory` flag to 1. Set all unused flags to 0.

`seqGrabToDisk`
: If you set this flag to 1, the recorder writes the data to the file specified by the `movieFile` parameter as the data is received. Set this flag to 0 if you set the `seqGrabToMemory` flag to 1 (only one of these two flags may be set to 1).

`seqGrabToMemory`
: Set this flag to 1 to store the data in memory until the recording process is complete. The recorder then writes all the data at once to the file specified by the `movieFile` parameter. This technique provides better performance than recording to a file as data is received, but it limits the amount of data you can record. Set this flag to 0 if you set the `seqGrabToDisk` flag to 1 (only one of these two flags may be set to 1).

`seqGrabDontUseTempMemory`
: Set this flag to 1 to prevent a recorder component from using temporary memory during the record operation. By default, a recorder component uses as much temporary memory as necessary to perform the record operation.

`seqGrabAppendToFile`
: Set this flag to 1 to cause a recorder component to append the recorded data to the data fork of the file specified by the `movieFile` parameter and to create a new movie resource in that file. By default, a recorder component that writes to QuickTime movie files deletes the movie file specified by the `movieFile` parameter, if it exists, and creates a new file containing one movie and the corresponding movie resource.

`seqGrabDontAddMovieResource`
: Set this flag to 1 to prevent a recorder component that writes to QuickTime movie files from adding the movie resource to the movie file specified by the `movieFile` parameter. You are then responsible for adding the resource to a file, if you so desire. By default, a recorder component creates a new movie resource and adds that resource to the movie file.

`seqGrabDontMakeMovie`
: If you set this flag to 1, the recorder component does not write any data to a destination file. You set this flag for recorder components that redirect the received data for further processing.

DESCRIPTION

You call the `MTRecorderSetDataOutput` function to specify the destination file for the recorded data and to set flags that control the behavior of a recorder component. You need to call this function before you start recording data.

If there isn't enough disk space to create a file before you start recording, the function returns the `dskFulErr` result code.

SPECIAL CONSIDERATIONS

The Apple-provided recorder component always writes received data directly to a QuickTime movie file. When you are using the Apple-provided recorder component, you must set the `seqGrabToDisk` flag to 1 and all other flags to 0.

In addition to the result codes listed below, the `MTRecorderSetDataOutput` function returns file system errors if a file is damaged or cannot be opened and QuickTime errors that pertain to creating movie files.

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	No more available disk space
`cantDoThatInCurrentMode`	–9402	Recording is currently in progress

MTRecorderGetDataOutput

The `MTRecorderGetDataOutput` function gets the file system specification of the destination file for a recording and the flags that control the record operation.

```
pascal ComponentResult MTRecorderGetDataOutput
         (MTRecorderComponent rc, FSSpec *movieFile,
         long *whereFlags);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`movieFile` A pointer to a file system specification. The function returns the specification.

`whereFlags` A pointer to flags that control the record operation. The function returns the flags. See the description of the `MTRecorderSetDataOutput` function for a description of the flags.

DESCRIPTION

You call the `MTRecorderGetDataOutput` function to find out the destination file of the recorded data and the flags that control the record operation.

If you haven't previously set the destination file and flags before calling `MTRecorderGetDataOutput`, the function returns the `noMovieFound` result code.

RESULT CODES

`noErr`	0	No error
`noMovieFound`	–2048	No destination file set

SEE ALSO

You set the destination file and the flags by calling the `MTRecorderSetDataOutput` function, described on page 8-16.

MTRecorderAddSource

The `MTRecorderAddSource` function specifies a stream director component instance as a source of data for a recording.

```
pascal ComponentResult MTRecorderAddSource
         (MTRecorderComponent rc,
         ComponentInstance sourceInstance);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`sourceInstance`
A stream director component instance that is a source of data for the recording. If you set this parameter to `nil`, the recorder component invalidates all sources you previously set.

DESCRIPTION

You call the `MTRecorderAddSource` function to specify a stream director component instance as a source of data for a recording. You can specify multiple sources to a recorder component.

Performance of the recording operation is inversely proportional to the number of streams you record simultaneously. It's also dependent on the codec in use, the network traffic load, and CPU performance characteristics.

To cause the recorder component to disregard all sources you have already set, set the `sourceInstance` parameter to `nil`.

In addition to the result codes listed below, the `MTRecorderAddSource` function may return result codes from the Memory Manager, the QuickTime Movie Toolbox, and the stream director component. If a connection does not exist when you call `MTRecorderAddSource`, it may also return QuickTime sequence grabber component result codes.

SPECIAL CONSIDERATIONS

The recorder component in QuickTime Conferencing 1.0 supports up to ten sources. If you attempt to add more than ten sources, the `MTRecorderAddSource` function returns the `mtOutOfResourcesErr` result code. Future versions of the software may allow you to add any number of sources, limited only by the available memory.

RESULT CODES

`noErr`	0	No error
`mtOutOfResourcesErr`	–7963	Cannot add any more sources

SEE ALSO

You can remove a single stream director component instance as a source of media data by calling the `MTRecorderDeleteSource` function, described next.

MTRecorderDeleteSource

The `MTRecorderDeleteSource` function removes a stream director component instance as a source of data for a recording.

```
pascal ComponentResult MTRecorderDeleteSource
         (MTRecorderComponent rc,
         ComponentInstance sourceInstance);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`sourceInstance`
A stream director component instance that is a source of data for the recording. You set this parameter to the stream director instance you want to remove.

DESCRIPTION

You call the `MTRecorderDeleteSource` function to remove a stream director component instance as a source of data for a recording.

You need to remove a stream director instance when it becomes invalid as a source of media data during a recording. Suppose, for example, that you are recording a multiparty conference call and one of the parties hangs up. You need to remove the stream director associated with the party that hung up from the list of sources being recorded. You should do this as soon as you know that a party has hung up. If your application uses a conference component, you can detect a hang-up by checking for the `mtMemberTerminatedEvent` event when you call the `MTConferenceGetNextEvent` function.

In addition to the result codes listed below, the `MTRecorderDeleteSource` function returns result codes from the QuickTime Movie Toolbox if an error occurs while closing a movie file.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Stream director source instance does not exist

SEE ALSO

You can add a stream director component instance as a source of media data by calling the `MTRecorderAddSource` function, described on page 8-19.

To simultaneously delete all sources you have previously set, call the `MTRecorderAddSource` function, described on page 8-19, and set its `sourceInstance` parameter to `nil`.

MTRecorderGetNextSource

The `MTRecorderGetNextSource` function gets the component instance of a stream director that is a data source for a recorder component.

```
pascal ComponentResult MTRecorderGetNextSource
         (MTRecorderComponent rc,
         ComponentInstance *sourceInstance);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`sourceInstance`
A pointer to a stream director component instance that is a source of data for the recorder component. To get the component instance of the first stream director source for the recorder component, set this parameter to `nil`. If you set it to a known component instance, the function returns the component instance of the next stream director in the list.

DESCRIPTION

You can use the `MTRecorderGetNextSource` function to get all of the sources that you've set for a recorder component. A recorder component maintains a list of stream directors that provide it data to record. The `MTRecorderGetNextSource` function returns the component instance of one stream director each time you call it. To start at the beginning of the list, set the source instance to `nil` when you call `MTRecorderGetNextSource`. The function returns the component instance of the first

stream director in the list. To cycle through the entire list, call the function repeatedly, each time passing it the component instance returned in the previous call.

When you reach the end of the list (that is, you pass `MTRecorderGetNextSource` the component instance of the last stream director on the list) or if no sources have been set, the function sets the source instance to `nil`.

You should not expect the sources to be stored in the list in any particular order.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can specify a stream director component instance as a source of data by calling the `MTRecorderAddSource` function, described on page 8-19.

Making a Recording

You use the functions described in this section to control a recording process. You can

- start and stop a recording
- inform a recorder component that stream data formats have changed
- pause a recording
- query the pause state of a recorder component

MTRecorderStartRecord

The `MTRecorderStartRecord` function tells a recorder component to start recording.

```
pascal ComponentResult MTRecorderStartRecord
                                    (MTRecorderComponent rc);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

DESCRIPTION

You call the `MTRecorderStartRecord` function to begin recording media data. The recorder component stores the data according to the flags you set with the `MTRecorderSetDataOutput` function.

You should have previously specified the sources of and destination file for the data. If you haven't previously specified a destination file, the function returns an error. If you

haven't specified a data source, the function does not return an error, but nothing gets recorded.

In addition to the result codes listed below, the `MTRecorderStartRecord` function returns QuickTime Movie Toolbox errors that pertain to creating movie tracks.

RESULT CODES

`noErr`	0	No error
`cantDoThatInCurrentMode`	–9402	Recording already in progress

SEE ALSO

You can add sources after you've started a recording as well as before you start. You specify a data source by calling the `MTRecorderAddSource` function, described on page 8-19.

To specify the destination file, call the `MTRecorderSetDataOutput` function, described on page 8-16.

MTRecorderStopRecord

The `MTRecorderStopRecord` function tells a recorder component to stop recording.

```
pascal ComponentResult MTRecorderStopRecord
                            (MTRecorderComponent rc);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

DESCRIPTION

You call the `MTRecorderStopRecord` function to stop recording media data.

The recorder releases any system resources it used during the recording operation, such as temporary memory. If it was writing the data to memory, it now writes it to disk. Then it closes the destination file and unsets the destination file and all the sources you previously specified.

If you want to record again after you call `MTRecorderStopRecord`, you need to call the `MTRecorderSetDataOutput` function to specify the destination file and the `MTRecorderAddSource` function to specify a data source.

In addition to the result codes listed below, the `MTRecorderStopRecord` function returns QuickTime Movie Toolbox errors that pertain to completing the creation of movie tracks and adding a movie resource, and stream director component errors.

RESULT CODES

`noErr`	0	No error
`cantDoThatInCurrentMode`	–9402	No recording is in progress

SEE ALSO

The `MTRecorderSetDataOutput` function is described on page 8-16.

The `MTRecorderAddSource` function is described on page 8-19.

MTRecorderChangedStreams

The `MTRecorderChangedStreams` function informs a recorder component that the format of streams has changed.

```
pascal ComponentResult MTRecorderChangedStreams
         (MTRecorderComponent rc,
         MTDirectorComponent sourceInstance);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`sourceInstance`
A stream director component instance whose stream formats have changed.

DESCRIPTION

You call the `MTRecorderChangedStreams` function to tell a recorder component that the format of one or more of the streams it is recording has changed. You should call this function as soon as a stream format negotiation is complete.

If your application uses a stream controller component, you can detect when stream format negotiations are complete by checking for the `mtControllerActionStreamsChanged` action in your action filter function. If your application does not use a stream controller component, you can monitor stream format negotiations through a negotiation callback function that you provide.

Keeping a recorder component informed of stream format changes allows it to update its internal operations in a timely manner and continue to record media data from the affected streams without data loss.

In addition to the result codes listed below, the `MTRecorderChangedStreams` function returns QuickTime Movie Toolbox errors and stream director component errors.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Stream director instance does not exist

SEE ALSO

See the chapter "Stream Controller Components" in this book for information about the `mtControllerActionStreamsChanged` action and action filter functions.

The chapter "Stream Director Components" in this book describes negotiation callback functions and how to monitor stream format negotiations.

MTRecorderPause

The `MTRecorderPause` function sets the pause state of a recorder component.

```
pascal ComponentResult MTRecorderPause (MTRecorderComponent rc,
            Boolean pause);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`pause` A Boolean value that indicates the pause state of the recorder component. Set this parameter to `true` to put the recorder into the pause state. Set it to `false` if you want the recorder component to be free to record media data.

DESCRIPTION

You call the `MTRecorderPause` function to set the pause state of a recorder component.

You can call `MTRecorderPause` at any time, regardless of whether you are currently recording media data. The recorder component remembers the pause state that you set.

If you are recording when you call `MTRecorderPause` with `pause` set to `true`, the recording pauses. If you set `pause` to `false`, recording resumes.

If you're not recording when you call `MTRecorderPause` with `pause` set to `true`, and if you later want to start recording, you need to call `MTRecorderPause` again, setting `pause` to `false` before you can actually start recording.

The function does not release any system resources or temporary memory associated with the record operation. As a result, it is typically much faster than using the `MTRecorderStopRecord` and `MTRecorderStartRecord` functions to suspend and resume a recording.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can use the `MTRecorderGetPause` function, described next, to find out the pause state of a recorder.

MTRecorderGetPause

The `MTRecorderGetPause` function tells you the pause state of a recorder component.

```
pascal ComponentResult MTRecorderGetPause
         (MTRecorderComponent rc, Boolean *paused);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`paused` A pointer to a Boolean value that indicates the pause state of the recorder component. The function sets this parameter to `true` if the recorder is paused. It sets it to `false` if the recorder is not paused.

DESCRIPTION

You call the `MTRecorderGetPause` function to get the pause state of a recorder component.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You set the pause state by calling the `MTRecorderPause` function, described on page 8-25.

Setting and Getting Recording Characteristics

You use the functions described in this section to get and set certain characteristics of a recording. You can

- get and set the maximum recording time
- find out how much storage space remains available for a recording
- find out how much recording time remains for a recording

- get the movie identifier of a destination movie file
- get the movie resource ID last used by a recorder component

MTRecorderSetMaximumRecordTime

The `MTRecorderSetMaximumRecordTime` function sets the maximum length of time for a recording.

```
pascal ComponentResult MTRecorderSetMaximumRecordTime
            (MTRecorderComponent rc, UInt32 ticks);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`ticks` The maximum number of Macintosh system ticks (60ths of a second) that a recording can continue. If you want the recording to continue indefinitely or until there is no more available disk space, set this parameter to 0.

DESCRIPTION

Before you start recording, you can call the `MTRecorderSetMaximumRecordTime` function to set the maximum length of time for recording data.

By default, there is no time limit on a record operation. If you don't set a limit, a record operation runs until it exhausts the Operating System resources or you call the `MTRecorderStopRecord` function. Memory and disk space are the two major limiting factors.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can monitor how much time is left for recording by calling the `MTRecorderGetTimeRemaining` function, described on page 8-29.

The `MTRecorderStopRecord` function is described on page 8-23.

MTRecorderGetMaximumRecordTime

The `MTRecorderGetMaximumRecordTime` function tells you the maximum length of time that you set for a recording.

```
pascal ComponentResult MTRecorderGetMaximumRecordTime
            (MTRecorderComponent rc, UInt32 *ticks);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`ticks` A pointer to the number of ticks (60ths of a second) allowed for a recording. If the function returns 0 in this parameter, the recording can continue forever or until disk space runs out.

DESCRIPTION

If you haven't previously set a maximum recording time, `MTRecorderGetMaximumRecordTime` returns 0.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You can set a maximum recording time by calling the `MTRecorderSetMaximumRecordTime` function, described on page 8-27.

MTRecorderGetStorageSpaceRemaining

The `MTRecorderGetStorageSpaceRemaining` function tells you the amount of disk space left for use during a recording.

```
pascal ComponentResult MTRecorderGetStorageSpaceRemaining
            (MTRecorderComponent rc, UInt32 *bytes);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`bytes` A pointer to the number of bytes remaining for use for a recording. The function returns the amount of storage space available on the device that holds the file.

DESCRIPTION

You call the `MTRecorderGetStorageSpaceRemaining` function to find out the number of bytes of available disk space on the device that holds the destination file for a recording.

The amount of disk space consumed by a recording can vary significantly, depending on the codec in use.

You can use the information returned by this function to monitor the amount of disk space being consumed during a record operation, to limit the amount of space consumed by a record operation, or to update a status display for the user.

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Out of disk space

SEE ALSO

You create a destination file and establish output conditions for a record operation by calling the `MTRecorderSetDataOutput` function, described on page 8-16.

The section "Monitoring Available Storage Space" on page 8-11 lists situations in which this function is useful.

MTRecorderGetTimeRemaining

The `MTRecorderGetTimeRemaining` function tells you the amount of time remaining for a recording.

```
pascal ComponentResult MTRecorderGetTimeRemaining
            (MTRecorderComponent rc, UInt32 *ticksLeft);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`ticksLeft` A pointer to the number of ticks (60ths of a second) that remain for a recording. This value has meaning only if you previously set a maximum record time by calling the `MTRecorderSetMaximumRecordTime` function.

DESCRIPTION

You call the `MTRecorderGetTimeRemaining` function to get the time remaining for the recording.

If you previously set the maximum time to be infinite or if you have not set any maximum time, the value returned in the `ticksLeft` parameter is undefined.

SEE ALSO

The `MTRecorderSetMaximumRecordTime` function is described on page 8-27.

MTRecorderGetMovie

The `MTRecorderGetMovie` function returns the movie identifier of the destination movie file during a recording operation.

```
pascal Movie MTRecorderGetMovie (MTRecorderComponent rc);
```

`rc` The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

DESCRIPTION

The `MTRecorderGetMovie` function returns a valid movie identifier while a recording is in progress. You must have previously specified a destination movie file to the `MTRecorderSetDataOutput` function and not yet called the `MTRecorderStopRecord` function to stop the recording operation.

If you haven't specified a destination movie file or if you stopped the record operation, the function returns `nil`.

You can use the movie identifier with Movie Toolbox functions. However, you should not dispose of the movie—it is owned by the recorder component. If you want to work with a movie containing the recorded data, use the Movie Toolbox's `NewMovieFromFile` function.

SEE ALSO

The `MTRecorderSetDataOutput` function is described on page 8-16.

The `MTRecorderStopRecord` function is described on page 8-23.

See the chapter "Movie Toolbox" in *Inside Macintosh: QuickTime* for information about the `NewMovieFromFile` function.

MTRecorderGetLastMovieResID

The `MTRecorderGetLastMovieResID` function gets the movie resource ID most recently used by a recorder component.

```
pascal ComponentResult MTRecorderGetLastMovieResID
            (MTRecorderComponent rc, short *resID);
```

`rc`	The recorder component instance you are using for this operation. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`resID`	A pointer to a resource ID. The function returns the movie resource ID most recently used for a recording. If you have not yet stopped a record operation, the function returns the `noMovieFound` result code instead of the resource ID.

DESCRIPTION

You call the `MTRecorderGetLastMovieResID` function to get the movie resource ID most recently used by a recorder component.

A recorder component assigns a new resource ID to each movie resource it creates. It creates the movie resource when you stop a record operation (unless you instructed the recorder not to add the movie resource to the movie file). Therefore, you must have stopped at least one recording operation before this function can return a valid movie resource ID.

The `MTRecorderGetLastMovieResID` function can return file system errors.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You tell a recorder whether or not to add a movie resource to a movie file when you call the `MTRecorderSetDataOutput` function, described on page 8-16.

You stop a recording by calling the `MTRecorderStopRecord` function, described on page 8-23.

Application-Defined Function

This section is of interest to developers who are writing new recorder components. It describes the recording callback function that a recorder component provides to a stream director. If your application is using a recorder component, you do not provide this function.

A stream director component calls the recording function when it has media data available so that a recorder component can record it to disk or otherwise handle the data.

MyRecordProc

When you call the `MTDirectorSetRecordProc` function, you provide a pointer to a callback function. A stream director calls your function when media data is available. Your recording function must support the following interface:

```
pascal OSErr MyRecordProc (MTStreamChunkRecordPtr chunk);
```

`chunk` A pointer to an `MTStreamChunkRecord` structure that defines a chunk of media data.

DESCRIPTION

Recorder components, rather than applications or other components, typically provide the recording callback function. A stream director calls the recording function every time it receives stream data and passes it a pointer to a chunk structure.

Usually, a recording function stores media data into a file so that it can be played back later. However, it can also redirect media data for other purposes. For example, it could display the data on the computer's screen, write it to a foreign file system, or perform some transformation on the data and store that information in a file. You can use a recording function to funnel the data to other locations for purposes that you determine.

Because data from audio or multiplexed streams can arrive at interrupt time, your recording function may be called at interrupt time. If this happens, you need to buffer the data until you can write it out to a disk file. You can use the memory component to request memory at interrupt time and the idler component to get periodic processing time for writing data to disk.

Your recording function needs to check the `chunkError` field of the chunk record structure. A nonzero value indicates a problem with the chunk. You should not record corrupted chunks.

If you are recording data from a sink stream director, you can check the `chunkFrameNumber` field of the chunk record structure to detect missing chunks.

The recording function should return the `noErr` result code when it successfully records the data and return a nonzero result code if it fails to record the data successfully.

SEE ALSO

You install your recording function by calling the `MTDirectorSetRecordProc` function, described in the chapter "Stream Director Components" in this book.

For a description of the chunk record structure `MTStreamChunkRecord`, see the chapter "Transport Components" in this book.

The memory and idler components are described in the chapter "Utility Components" in this book.

Summary of Recorder Components

C Summary

Constants

```
enum {         /* recorder component type */
   kMTRecorderType      = 'recd'
};

enum {         /* recorder component subtype */
   kMTMovieSubType      = MovieFileType
};

enum {         /* recorder component request codes */
   kMTRecorderAddSourceSelect                   = 1,
   kMTRecorderDeleteSourceSelect                = 2,
   kMTRecorderGetNextSourceSelect               = 3,
   kMTRecorderSetDataOutputSelect               = 4,
   kMTRecorderGetDataOutputSelect               = 5,
   kMTRecorderStartRecordSelect                 = 6,
   kMTRecorderStopRecordSelect                  = 7,
   kMTRecorderPauseSelect                       = 8,
   kMTRecorderGetPauseSelect                    = 9,
   kMTRecorderChangedStreamsSelect              = 10,
   kMTRecorderGetMovieSelect                    = 11,
   kMTRecorderSetMaximumRecordTimeSelect        = 12,
   kMTRecorderGetMaximumRecordTimeSelect        = 13,
   kMTRecorderGetStorageSpaceRemainingSelect    = 14,
   kMTRecorderGetTimeRemainingSelect            = 15,
   kMTRecorderGetLastMovieResIDSelect           = 16
};
```

Data Types

```
typedef ComponentInstance MTRecorderComponent;
```

Functions

Setting Up a Recording

```
pascal ComponentResult MTRecorderSetDataOutput
                            (MTRecorderComponent rc, FSSpec *movieFile,
                             long whereFlags);
pascal ComponentResult MTRecorderGetDataOutput
                            (MTRecorderComponent rc, FSSpec *movieFile,
                             long *whereFlags);
pascal ComponentResult MTRecorderAddSource
                            (MTRecorderComponent rc,
                             ComponentInstance sourceInstance);
pascal ComponentResult MTRecorderDeleteSource
                            (MTRecorderComponent rc,
                             ComponentInstance sourceInstance);
pascal ComponentResult MTRecorderGetNextSource
                            (MTRecorderComponent rc,
                             ComponentInstance *sourceInstance);
```

Making a Recording

```
pascal ComponentResult MTRecorderStartRecord
                            (MTRecorderComponent rc);
pascal ComponentResult MTRecorderStopRecord
                            (MTRecorderComponent rc);
pascal ComponentResult MTRecorderChangedStreams
                            (MTRecorderComponent rc,
                             MTDirectorComponent sourceInstance)
pascal ComponentResult MTRecorderPause
                            (MTRecorderComponent rc, Boolean pause);
pascal ComponentResult MTRecorderGetPause
                            (MTRecorderComponent rc, Boolean *paused);
```

Getting and Setting Recording Characteristics

```
pascal ComponentResult MTRecorderSetMaximumRecordTime
                            (MTRecorderComponent rc, UInt32 ticks);
pascal ComponentResult MTRecorderGetMaximumRecordTime
                            (MTRecorderComponent rc, UInt32 *ticks);
pascal ComponentResult MTRecorderGetStorageSpaceRemaining
                            (MTRecorderComponent rc, UInt32 *bytes);
pascal ComponentResult MTRecorderGetTimeRemaining
                            (MTRecorderComponent rc, UInt32 *ticksLeft);
```

```
pascal Movie MTRecorderGetMovie
                            (MTRecorderComponent rc);
pascal ComponentResult MTRecorderGetLastMovieResID
                            (MTRecorderComponent rc, short *resID);
```

Application-Defined Function

```
pascal OSErr MyRecordProc   (MTStreamChunkRecordPtr chunk);
```

Result Codes

`noErr`	0	No error
`dskFulErr`	–34	No more available disk space
`noMovieFound`	–2048	No destination file set
`mtInvalidStreamIDErr`	–7869	Stream director source instance does not exist
`mtOutOfResourcesErr`	–7963	Cannot add any more sources
`cantDoThatInCurrentMode`	–9402	Recording is currently in progress

Transport Components

Contents

This chapter describes transport components. **Transport components** allow other QuickTime Conferencing components to send and receive networked media data. This chapter is divided into the following major sections:

- "About Transport Components" introduces you to the features and functions of transport components and discusses their relationships with other QuickTime Conferencing components.
- "Using Transport Components" provides several examples of how you can use a transport component in your application.
- "Creating Transport Components" tells you what you need to know before you create your own transport component.
- "Transport Component Reference" contains detailed technical information about the interface supported by transport components.
- "Summary of Transport Components" provides a summary of the transport component interface.

Transport components are Component Manager components. As such, you need to be familiar with components and how to interact with them before you can use or create a transport component. If components are new to you, see the chapter that discusses the Component Manager in *Inside Macintosh: More Macintosh Toolbox*.

This chapter is aimed primarily at developers who plan to create their own transport components. By creating a new transport component, you can allow QuickTime Conferencing applications to take advantage of new networked media protocols. If you plan to create a transport component, you should read the entire chapter. Sections in the chapter describe the behavior your component should provide and the interface it must support.

If you are developing a QuickTime Conferencing application, you most likely do not need to work directly with a transport component. In most cases, your application can use the conference component to set up and use network calls. If, however, you have determined that you need to use a transport component in your application, this chapter describes how to do so and the interface supported by all transport components. You should skip the material that discusses how to create a transport component, though.

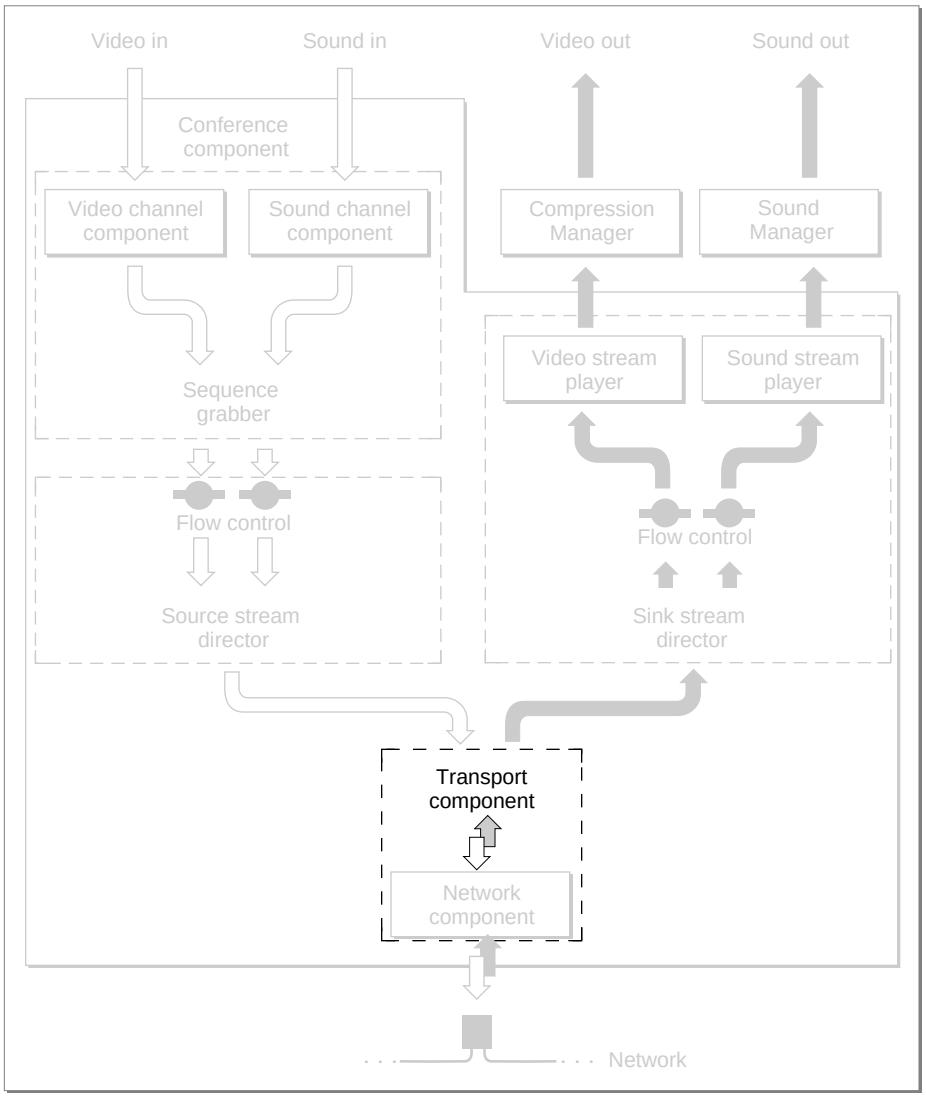
About Transport Components

Transport components, in tandem with network components, allow QuickTime Conferencing components to engage in networked multimedia communications. Network components, which are discussed in the "Network Components" chapter elsewhere in this book, support specific network protocols. Transport components use the services of network components to support a given networked-media protocol and thereby allow other QuickTime Conferencing components to exchange media data across a network.

QuickTime Conferencing is designed to support a variety of existing and future network technologies and to support cross-platform implementations. Transport components

provide an important part of that flexibility, by isolating higher-level components and applications from networked-media communications details. Figure 9-1 shows the relationships between transport components and other QuickTime Conferencing components.

Figure 9-1 The relationship between transport components and other QuickTime Conferencing components



A primary goal of the QuickTime Conferencing architecture is to support multimedia communications on any type of underlying network interface. To help achieve this goal, Apple has defined a transport component interface that is networked-media protocol independent.

Transport components provide five basic services:

- **Call setup**: Transport components use the metaphor of a call to describe the connection between network partners. You set up a connection by either calling a remote party or answering an incoming call.
- **Media and control data exchange**: Transport components assure that messages (control channel information) and media chunks (media stream information) get across the network. These components provide a full set of functions that allow you to exchange control and media data with remote applications.
- **Call management**: Transport components provide functions to determine a call's status, a remote entity's address, and other information about an existing call.
- **Stream management**: Multimedia data flows through the transport component as a set of distinct "streams." Transport components provide functions to create, attach, dispose, enable, disable, and control media streams.
- **Name service management**: Transport components use a network-independent address format (known as the MovieTalk address) to specify entities on the network for connection purposes. In addition, for those underlying protocols that provide some type of name service for managing network entities, these components provide functions that work with network names.

Following are definitions for some key terms that relate to transport components.

- **call**: Describes a connection between two QuickTime Conferencing components, where media data is sent over one channel and control information is sent over another channel. The media data is typically sent on a best effort high-speed channel, and the control data on a reliable, though higher-latency channel. However, a QuickTime Conferencing call can support media channels on whatever underlying network services are available.
- **stream**: A sequence of media data, such as video or sound, carried on a media channel in a QuickTime Conferencing call. Stream data is organized as a sequence of chunks. A QuickTime Conferencing call can carry several streams, including more than one stream of a given media type.
- **originator**: The end of a QuickTime Conferencing call that initiates communications by calling the other side.
- **answerer**: The end of a QuickTime Conferencing call that receives an incoming call from the originator.
- **point-to-point**: A point-to-point call has exactly one originator and one answerer. A point-to-point call, once started, cannot be converted to a multicast call.
- **multicast**: A multicast call supports more than one receiver of media data. In addition, new receivers may join a multicast call at any time, even after data has started flowing. This implies that media data format information is transmitted in-band, along with the media data. A multicast call, once started, cannot be converted to a point-to-point call.

Using Transport Components

This section provides examples that show you how you can use a transport component in your application. Each of the examples highlights a different aspect of the services provided by a transport component and includes sample code illustrating how to use those services. Taken together, these samples constitute much of the logic of a working, though simple, networked media data application.

This section contains the following examples:

- "Opening and Closing Transport Components" shows how to use the Component Manager to open and close an instance of a transport component.
- "Originating, Answering, and Hanging Up Calls" provides examples of placing and ending calls.
- "Sending and Receiving Control Messages" shows how to exchange control information using a transport component.
- "Sending and Receiving Media Data" shows how to exchange media data with another network entity.

Note that, while these examples give you much of what you need to start using transport components, there is much more that these components have to offer. For complete information about their functional interface, consult the "Transport Component Reference" later in this chapter.

The samples in this section use a number of global variables. Listing 9-1 provides the declarations for those global variables.

Listing 9-1 Global variables

```
MTReleaseUPP            gReleaseProc;
MTStreamID              gRecvStreamID;
MTStreamID              gSendStreamID;
MTChunkPriority         gSendStreamPriority;
MTStreamOptions         gSendStreamOptions;
MTSequenceNum           gFrameNumber;
```

Note

The samples in this section show the calling sequences you might employ when using transport components, but do not necessarily show the most efficient way to use each function. For example, these samples always call the transport component synchronously. You, on the other hand, would most likely call the component asynchronously and supply a completion routine. ◆

Opening and Closing Transport Components

As is the case with all Macintosh components, you must use Component Manager functions in order to gain access to a transport component. If you are not familiar with the Component Manager, you should take a look at the appropriate chapter in *Inside Macintosh: More Macintosh Toolbox*.

To open an instance of a transport component for your use, call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. The following code fragment uses the `OpenDefaultComponent` function:

```
instance = OpenDefaultComponent(kMTTransportType,
                                kMTMovieTalkSubType);
if (instance == nil) {
   /* process error */
}
```

This function requires you to identify the type and subtype of the component you desire. The `kMTTransportType` constant identifies the component type—in this case, the transport component.

Transport components use their subtype value to identify the networked-media protocol they support. The `kMTMovieTalkSubType` constant used here indicates that the network component you want to use supports the MovieTalk media protocol.

During your initialization processing, you might prepare your component to receive media and control message data. The code in Listing 9-2 sets up a transport component to receive incoming messages.

Listing 9-2 Setting up to receive incoming media data and control messages

```
err = MTTransportSetMessageProc(instance, SetUpRecvStream,
                                (long) instance);
if (err != noErr) {
   /* process error */
}

err = MTTransportGetReleaseProc(instance, &gReleaseProc);

err = MTTransportSetMediaDataProc(instance, ReceiveMedia,
                                (long) gReleaseProc);
if (err != noErr) {
   /* process error */
}
```

When you are done with the transport component, you should clean up. Typically, you will have registered a network name and set up some media streams. The code in Listing 9-3 cleans up after this sample application. If you don't plan to reuse any of these system resources, you may clean up by calling the Component Manager's `CloseComponent` function.

Listing 9-3 Cleaning up before exiting your application

```
err = MTTransportDisposeMediaStream(instance, gRecvStreamID);
if (err != noErr) {
   /* process error */
}

err = MTTransportDisposeMediaStream(instance, gSendStreamID);
if (err != noErr) {
   /* process error */
}

err = MTTransportRemoveName(instance, false, nil, nil);
if (err != noErr) {
   /* process error */
}

err = MTTransportDisconnect(instance, false, nil, nil);
if (err != noErr) {
   /* process error */
}
```

When you are done with the transport component instance, use the `CloseComponent` function to close the instance:

```
result = CloseComponent(instance);
```

Originating, Answering, and Hanging Up Calls

Transport components provide a number of functions that allow you to place and receive calls across the network. The samples in this section show you how to receive incoming calls and place outgoing calls.

First, consider an application that must receive an incoming network call. First, you must register your network name. Then you listen for incoming calls. Once you receive a call, you accept the call by answering it. The code in Listing 9-4 shows sample logic for all of this.

Listing 9-4 Setting up to receive an incoming call

```
/* set up and receive a transport call */

pascal ComponentResult ReceiveCall(MTTransportComponent
                              instance, OSType networkType,
                              ConstStr255Param localName,
                              ConstStr255Param localType)
{
   ComponentResult         err;
   MTAddress               remoteAddr;

   /* register name; listen for and answer incoming call */

   err = MTTransportRegisterName(instance, networkType, localName,
                              localType, false, nil, nil);
   if (err != noErr) {
      /* process error */
   }

   err = MTTransportListen(instance, &remoteAddr, false, nil,
                              nil);
   if (err != noErr) {
      /* process error */
   }

   err = MTTransportAnswer(instance, &remoteAddr, instance, false,
                              nil, nil);

   return (err);
}
```

On the other side of the network connection, the remote application must place a call. In order to do so, the application must resolve the desired network name to a valid network address and then place the call. The code in Listing 9-5 does this.

Listing 9-5 Setting up to place a call

```
/* set up and make a transport call */

pascal ComponentResult MakeCall(MTTransportComponent instance,
                              MTNamePtr remoteName)
{
```

```
   ComponentResult         err;
   MTAddress               remoteAddr;

   /* look up name and make outgoing call */

   err = MTTransportLookupName(instance, remoteName,
                               &remoteAddr, false, nil, nil);
   if (err != noErr) {
      /* process error */
   }

   err = MTTransportCall(instance, &remoteAddr, false, nil, nil);

   return (err);
}
```

In either case, you end a call by calling the `MTTransportDisconnect` function. Either end may call this function at any time.

Sending and Receiving Control Messages

Transport components allow you to send both control messages and media data. A common use for control messages is to exchange the information necessary to allow you to send or receive media data. The examples in this section demonstrate how to send and receive control messages.

In this example, the application uses the control channel to tell the remote application which stream to use for media data. First, the local application sets up its control channel and then sends a control message on that channel. This control message identifies the application's media stream. Note that the application defines the format and content of the control message. Listing 9-6 shows the local application's send routine.

Listing 9-6 Sending a control message

```
typedef struct {
   MTMessageHeader         header;
   MTStreamID              stream;
} StreamMessage, *StreamMessagePtr;

/* set up outgoing transport stream and inform partner */

pascal ComponentResult SetUpSendStream(MTTransportComponent
                               instance)
{
```

```
   ComponentResult         err;
   StreamMessage           streamMessage;

   /* allocate stream for sending data */

   err = MTTransportNewMediaStream(instance, &gSendStreamID, true,
                              nil);
   if (err != noErr) {
      /* process error */
   }

   /* set up structure to send message */

   streamMessage.header.msgType = 'MYID';
   streamMessage.header.msgSize = sizeof(StreamMessage);
   streamMessage.stream = gSendStreamID;

   /* send stream ID to partner */

   err = MTTransportSendMessage(instance, &streamMessage.header,
                              false, nil, nil);

   return (err);
}
```

On the other end of the call, the remote application receives the control message and opens its corresponding media stream, as shown in Listing 9-7. Note that the application's message routine receives the message.

Listing 9-7 Receiving a control message

```
/* get incoming transport stream from partner and attach */

pascal OSErr SetUpRecvStream(MTMessageHeader *message,
                              long refCon)
{
   ComponentResult         err;
   MTTransportComponent    instance;
   StreamMessagePtr        streamMessage;

   /* get stream ID from partner's message */

   instance = (MTTransportComponent) refCon;
```

```
   streamMessage = (StreamMessagePtr) message;

   /* attach stream for receiving data */

   err = MTTransportAttachMediaStream(instance,
                              streamMessage->stream,
                              true);
   if (err != noErr) {
      /* process error */
   }

   return (noErr);
}
```

Sending and Receiving Media Data

Transport components allow you to transmit media data between applications. Before you can do so, you must attach appropriate media streams. The sample code in "Sending and Receiving Control Messages" beginning on page 9-10 shows one way to do so. The routines listed there use the control connection to exchange media stream IDs in anticipation of exchanging media data.

The media data is structured as time-oriented streams. The transmitted data is composed of logical units called *chunks,* which correspond to video frames or collections of sound samples.

The routine in Listing 9-8 sends media data using a transport component.

Listing 9-8 Sending media data

```
/* send media data on a transport stream */

pascal ComponentResult SendMedia(MTTransportComponent instance,
                              Byte *mediaData, MTChunkSize
                              mediaSize, MTChunkTime mediaTime)
{
   ComponentResult         err;
   MTStreamChunkRecord     chunk;

   chunk.chunkPtr = mediaData;
   chunk.chunkSize = mediaSize;
   chunk.chunkTime = mediaTime;
   chunk.chunkStreamID = gSendStreamID;
   chunk.chunkPriority = gSendStreamPriority;
   chunk.chunkStreamOptions = gSendStreamOptions;
```

```
   err = MTTransportSendMediaData(instance, &chunk, false, nil,
                                  nil);

   return (err);
}
```

Receiving media data is not much more difficult, as shown in Listing 9-9. The `DeliverMedia` function passes the received data to a waiting application. The media data is first received by the application's media data routine.

Listing 9-9 Receiving media data

```
/* receive media data on a transport stream */

pascal OSErr ReceiveMedia(MTStreamChunkRecordPtr chunk)
{
   MTReleaseProcPtr     releaseProc;

   /* deliver media for processing */

   DeliverMedia(chunk->chunkPtr, chunk->chunkSize,
                                  chunk->chunkTime,
                                  chunk->chunkStreamID);

   /* release chunk */

   releaseProc = (MTReleaseProcPtr)chunk->chunkRefCon;
   (*releaseProc)(chunk);

   return (noErr);
}
```

Creating Transport Components

By creating a new transport component, you can open Apple's QuickTime Conferencing technology to new media protocols. This section discusses the topics you should consider if you plan to create your own transport component. If you plan to use, but not create, transport components, you need not read this section. More than likely, your transport component will use the services of a network component to send and receive network data. For more information about network components, see the "Network Components" chapter, elsewhere in this book.

If you are unfamiliar with creating Macintosh components in general, refer to the Component Manager chapter in *Inside Macintosh: More Macintosh Toolbox* for more information.

Functional Interface

Apple has defined a functional interface for transport components. Your transport component must support that interface. For a complete description of the interface your component must support, see "Functions" beginning on page 9-22.

You can use the following constants to identify the request codes your component receives from applications that use its services. Each of these constants corresponds to the similarly named transport component function described elsewhere in this chapter.

```
enum {
   kMTTransportListenSelect =               1,
   kMTTransportAnswerSelect =               2,
   kMTTransportCallSelect =                 3,
   kMTTransportDisconnectSelect =           4,
   kMTTransportSetNotificationProcSelect =  5,
   kMTTransportGetNotificationProcSelect =  6,
   kMTTransportSendMessageSelect =          7,
   kMTTransportSendMediaDataSelect =        8,
   kMTTransportSetMediaDataProcSelect =     9,
   kMTTransportSetMessageProcSelect =       10,
   kMTTransportGetMessageProcSelect =       11,
   kMTTransportGetReleaseProcSelect =       12,
   kMTTransportNewMediaStreamSelect =       13,
   kMTTransportDisposeMediaStreamSelect =   14,
   kMTTransportAttachMediaStreamSelect =    15,
   kMTTransportEnableStreamSelect =         16,
   kMTTransportIsStreamEnabledSelect =      17,
   kMTTransportGetStreamPerformanceSelect = 18,
   kMTTransportRegisterNameSelect =         19,
   kMTTransportRemoveNameSelect =           20,
   kMTTransportLookupNameSelect =           21,
   kMTTransportGetLocalNameSelect =         22,
   kMTTransportGetStatusSelect =            23,
   kMTTransportGetInfoSelect =              24,
   kMTTransportGetAddressSelect =           25
};
```

Component Type and Subtype Values

Apple has defined a type value for transport components. All transport components have a component type of `'tran'`. You can use the following constant to specify this component type value:

```
enum {                /* transport component type */
   kMTTransportType           = 'tran'
};
```

A transport component's component subtype value indicates the type of networked-media protocol that the component supports. Apple provides a transport component that implements the MovieTalk protocol. The `kMTMovieTalkSubType` constant identifies this Apple-provided transport component. If you develop a new type of transport component, you need to create a new subtype value that identifies the supported networked-media protocol.

```
enum {                /* MovieTalk transport component subtype */
   kMTMovieTalkSubType        = 'mtlk'
};
```

Required and Optional Functions

Your transport component must support all of the functions that constitute the transport component interface.

Interrupt-Time Processing

Applications must be able to call most transport component functions at interrupt time. This restricts how you implement your component. Except for the functions listed here, all transport component functions must be interrupt safe:

- `MTTransportAnswer`
- `MTTransportAttachMediaStream`
- `MTTransportCall`
- `MTTransportDisposeMediaStream`
- `MTTransportListen`
- `MTTransportNewMediaStream`
- `MTTransportRegisterName`
- `MTTransportSetMediaDataProc`
- `MTTransportSetMessageProc`
- `MTTransportSetNotificationProc`
- Your component's `Open` and `Close` functions

This means that you may not allocate memory except in these functions.

The QuickTime Conferencing memory component supplies memory management functions that you can use at interrupt time. Please refer to the "Utility Components" chapter of this book for more information about these functions.

Asynchronous Functions

Many transport component functions may execute asynchronously. Apple requires that all transport component functions that may access the underlying network be capable of asynchronous execution. All functions that support an `async` parameter must be capable of executing asynchronously. Conversely, all functions are required to execute synchronously when the client requests synchronous execution.

It is up to the client application to specify whether to execute asynchronously or synchronously. Client applications may supply completion routines and reference constants when they request asynchronous execution. Applications are allowed to supply a `nil` pointer for the completion routine if they do not desire notification of completion. You must call a supplied completion routine, unless you have completed the request due to an immediate error (for example, a bad parameter value). In this case, you cannot call the completion routine.

Your `MTTransportSendMediaData` and `MTTransportSendMessage` functions are required to support multiple outstanding asynchronous calls. All other functions must return an error if an asynchronous call is already in progress when the client requests an asynchronous operation.

Transport Component Reference

This section describes the constants, data types, and functions that constitute the transport component interface. This section covers the following topics:

- "Constants" discusses transport component constants.
- "Data Types" beginning on page 9-17 discusses transport component data types.
- "Functions" beginning on page 9-22 discusses transport component functions.

Constants

This section describes the constants used by the transport component.

Transport Component Type and Subtype

The Component Manager uses the component type value to locate all components of a given type. The `kMTTransportType` constant defines a transport component's type value.

A transport component's component subtype value indicates the type of networked-media protocol that the component supports. Apple provides a transport component that implements the MovieTalk protocol. The `kMTMovieTalkSubType` constant identifies this Apple-provided transport component. If you develop a new type of transport component, you need to create a new subtype that identifies the supported networked-media protocol.

```
enum {               /* transport component type and subtype */
   kMTTransportType           = 'tran',
   kMTMovieTalkSubType        = 'mtlk'
};
```

Name and Address Maximum Lengths

A number of transport component data structures and functions require network addresses and names. The following constants define the maximum sizes of these address and name fields:

```
enum {
   kMTMaxAddressSize       = 256,
   kMTMaxNameSize          = 248,
   kMTMaxEmailAddressSize  = 1832,
   kMTMaxDisplayNameSize   = 124
};
```

Constant descriptions

`kMTMaxAddressSize`
: The maximum size, in bytes, of the network-specific address contained in a MovieTalk address (`MTAddress` structure).

`kMTMaxNameSize`
: The maximum size, in bytes, of the network-specific name contained in a MovieTalk name (`MTName` structure).

`kMTMaxEmailAddressSize`
: The maximum size, in bytes, of the preferred electronic mail address contained in a MovieTalk name (the `entityInfo` field of an `MTName` structure.

`kMTMaxDisplayNameSize`
: The maximum size, in bytes, of the display name contained in a MovieTalk name (`MTName` structure).

Data Types

Many transport component functions and structures use component-specific data types to define parameters, fields, and structures. This section discusses those data types.

Chunk Record

The chunk record describes a chunk, a logical unit of data in a stream of media data. For example, in a video media stream, each chunk corresponds to a frame. You provide a chunk record to a transport component when you want to transmit media data using the `MTTransportSendMediaData` function.

The `MTStreamChunkRecord` structure defines the chunk record:

```
struct MTStreamChunkRecord {
   Byte                 *chunkPtr;
   MTChunkSize          chunkSize;
   MTChunkTime          chunkTime;
   MTChunkPriority      chunkPriority;
   MTStreamID           chunkStreamID;
   ComponentResult      chunkError;
   long                 chunkRefCon;
   UInt32               chunkReserved;
   long                 chunkTimePlayed;
   MTReleaseUPP         chunkReleaseProcCallback;
   long                 chunkPrivate;
   MTStreamOptions      chunkStreamOptions;
   UInt8                chunkReserved2;
   MTSequenceNum        chunkFrameNumber;
   UInt16               chunkReserved3;
};
typedef struct MTStreamChunkRecord MTStreamChunkRecord,
*MTStreamChunkRecordPtr;
```

Field descriptions

`chunkPtr` A pointer to a chunk of media data, such as a collection of sound samples or a video frame. The structure of a chunk varies with the type of media data carried in the stream and is not described by MovieTalk.

`chunkSize` The size of the chunk, in bytes.

`chunkTime` The time stamp of the media data, expressed in the stream's time scale. This is the time value associated with the media data in the chunk. The receiving system uses this time stamp to determine when to play the data contained in the chunk.

`chunkPriority` The priority of this chunk. The priority is relative to all other chunks in all the streams within a given media channel. The following values are supported:

`mtChunkPrioritySound`
The priority level for sound data. It is the highest priority among the defined priority levels.

`mtChunkPriorityMedium`
An intermediate priority level. It falls between the priorities for sound and video data.

`mtChunkPriorityVideo`
The priority for delta frames of video data. It is the lowest priority among the defined priority levels. Delta frames are distinguished from key frames in that a delta frame contains only changed information from a previous frame. A key frame contains complete information for the video frame.

`mtChunkPriorityKeyFrame`
A constant used to generate a priority for chunks containing key frames. You can use it with any media type to generate a priority that applies to a chunk on which you resynchronize. You generate the key frame priority value by performing a bitwise OR operation between `mtChunkPriorityKeyFrame` and another priority constant, such as `mtChunkPriorityVideo`. This priority value is always used for sound samples.

In the future, Apple and third parties may define additional priorities for new media types.

`chunkStreamID` The stream ID of the stream that contains this chunk. A transport component assigns the stream ID when your application calls the `MTTransportNewMediaStream` function to create a new stream.

`chunkError` A value that indicates transmission errors. Transport components set this field to indicate whether the chunk was received from the network without data loss. A value of 0 indicates that the chunk was received in full. A negative value indicates that something is wrong with the received data. Any QuickTime Conferencing result code is valid in this field.

`chunkRefCon` A reference constant. Transport components set this field appropriately, depending upon the recipient of the chunk record. For example, you provide a value for the `refCon` parameter when you assign a media data routine by calling the `MTTransportSetMediaDataProc` function. The transport component places that value in this field when it calls your media data routine. Note that the value of this field is not transmitted across the network; it is for local use only.

`chunkReserved` Reserved.

`chunkTimePlayed` This field is not used by transport components.

`chunkReleaseProcCallBack`
A pointer to a routine that a transport component's release routine calls before disposing of the chunk. Stream director components set this field so that they get notified before the memory associated with a chunk is released. Your application should not modify or use this field.

`chunkPrivate` Reserved.

`chunkStreamOptions`
: Additional information that is specific to a particular type of media data. Flow control components add this information for outgoing chunks and read it for incoming chunks. Your application should not modify or use this field. Transport components send this value through the network unchanged.

`chunkReserved2` Reserved.

`chunkFrameNumber`
: The frame sequence number corresponding to the media data. This field identifies the frame to which the media data belongs. The value contained in this field is incremented each time your application calls the transport component's `MTTransportSendMediaData` function.

`chunkReserved3` Reserved.

MovieTalk Message Header

MovieTalk protocol messages consist of a message header, which may be followed by additional message data.

```
struct MTMessageHeader {
   MTStandardID         msgType;
   MTMessageSize        msgSize;
};
typedef struct MTMessageHeader MTMessageHeader, *MTMessagePtr;
```

Field descriptions

`msgType` The type of message. In many cases, this information is sufficient and there is no additional message data.

`msgSize` The number of bytes in the entire message—that is, the message header plus any message data that may follow.

MovieTalk Address Record

Many transport component functions and MovieTalk protocol messages include network-independent address information. That information is always stored in a MovieTalk address record.

```
struct MTAddress {
   OSType               transportType;
   OSType               networkType;
   Byte                 networkAddress[kMTMaxAddressSize];
};
typedef struct MTAddress MTAddress, *MTAddressPtr;
```

Field descriptions

`transportType` The networked-media protocol. This field contains the component subtype of the transport component using this address. For example, a MovieTalk transport component sets this field to `kMTMovieTalkSubType` for any addresses it supports.

`networkType` The network protocol. This field contains the component subtype of the network component using this address. For example, a transport component that is using an AppleTalk network component sets this field to `kMTAppleTalkSubType` for any addresses it supports.

`networkAddress` A network-specific address. Its format varies according to the needs of the network supported by the network component.

MovieTalk Name Record

Many transport component functions and MovieTalk protocol messages include network-independent names. Network names are always stored in a MovieTalk name record.

```
struct MTName {
   OSType          transportType;
   OSType          networkType;
   Byte            networkName[kMTMaxNameSize];
   struct {
      short        charSet;
      short        length;
      Byte         name[kMTMaxDisplayNameSize];
   }               displayName;      /* in RString format */
   Byte            entityInfo[kMTMaxEmailAddressSize];
};
typedef struct MTName MTName, *MTNamePtr;
```

Field descriptions

`transportType` The networked-media protocol. This field contains the component subtype of the transport component using this address. For example, a MovieTalk transport component sets this field to `kMTMovieTalkSubType` for any addresses it supports.

`networkType` The network protocol. This field contains the component subtype of the network component using this address. For example, a transport component that is using an AppleTalk network component sets this field to `kMTAppleTalkSubType` for any addresses it supports.

`networkName` A network-specific name. Its format varies with the needs of the network supported by the network component.

The `networkName` value is a C string in the format `"name:service type:address"`. Depending on the network

type selected, the `networkName` field contains the following information:

Network	networkName field contains
AppleTalk	`"name:type:zone"`
TCP/IP	`"name:socket number:MTIPAddress"`
ISDN	`"name:0:MTISDNAddress"`

In most cases, your application does not need to interpret the information in the `networkName` field. It is up to the individual QuickTime Conferencing transport and network components to interpret and use this information.

`displayName` The user name in displayable form. This structure consists of the following fields:

- `charSet` The character set that applies to the text contained in the `name` field.
- `length` The length, in bytes, of the text contained in the `name` field.
- `name` The actual name text.

`entityInfo` The preferred electronic mail address.

Functions

This section discusses the functions supported by transport components. This section has been divided into the following topics:

- "Establishing and Terminating a QuickTime Conferencing Call" discusses how to set up and close a call.
- "Sending and Receiving Media Data" describes how to exchange media data with remote applications.
- "Sending and Receiving Call Control Messages" tells you how to exchange control messages with remote applications.
- "Managing Streams" discusses the functions that allow you to work with media streams.
- "Managing a Call" provides information about managing a call once you have established it.
- "Managing Network Names" discusses the name management functions supported by transport components.
- "Application-Defined Functions" describes the interfaces your callback routines must support.

Establishing and Terminating a QuickTime Conferencing Call

You use the functions described in this section to set up and tear down QuickTime Conferencing calls.

The MTTransportListen function allows you to receive incoming call requests. You accept an incoming call by calling the MTTransportAnswer function. You can place a call with the MTTransportCall function. You end or reject a call with the MTTransportDisconnect function.

MTTransportListen

The MTTransportListen function tells a transport component to listen for an incoming call.

```
pascal ComponentResult MTTransportListen (MTTransportComponent c,
                        MTAddress *remoteAddr, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

c	The transport component you are using.
remoteAddr	A pointer to a MovieTalk address record. The function returns the network address of a remote caller that is attempting to set up a call. You are responsible for allocating the memory for the address record and for disposing of it.
async	A Boolean value that indicates whether you want the function to execute asynchronously. Set this parameter to true if you want asynchronous operation. Otherwise, set it to false to specify synchronous execution.
completion	A pointer to your completion routine. The function calls this routine when an asynchronous operation completes. If you don't provide a completion routine, set this parameter to nil. The function ignores this parameter if you call it synchronously.
refCon	Reference constant for your use. The transport component passes this value to your completion routine.

DESCRIPTION

You call the MTTransportListen function to listen for a remote caller's attempt to make a call. You must register you endpoint's network name before calling this function. Use the MTTransportRegisterName function to register a network name.

Once another endpoint tries to reach you, you must either answer the incoming call by calling the MTTransportAnswer function or reject it by calling the MTTransportDisconnect function before you may listen for another incoming call.

When you call the MTTransportListen function asynchronously and you provide a completion routine, the transport component calls your completion routine when it receives an incoming call. This is the most efficient way to use the function.

If you don't provide a completion routine for an asynchronous call, you can call the MTTransportGetStatus function in your event loop to poll the call status. After

calling the `MTTransportListen` function asynchronously but before an incoming call is received, the `MTTransportGetStatus` function reports a status of `mtTransportListenPendingStatus`. When a listen operation completes successfully, the `MTTransportGetStatus` function reports a status of `mtTransportCalledStatus`.

Once you determine that you have received an incoming call (either by the transport component calling your completion routine or by checking the call's status), you should call either the `MTTransportAnswer` function to answer the call or the `MTTransportDisconnect` function to refuse it.

To cancel a listen operation, call the `MTTransportDisconnect` function.

SPECIAL CONSIDERATIONS

Calling the `MTTransportListen` function synchronously is not recommended. If you do so, no other operations can take place on the computer until the listen operation completes or fails.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous listen operation already in progress
`mtIncompatibleStateErr`	–7962	Cannot listen in the current state

In addition to these result codes, the `MTTransportListen` function returns network component errors.

SEE ALSO

The `MTTransportGetStatus` function is described on page 9-43. This section also discusses call state status constants, including `mtTransportCalledStatus` and `mtTransportListenPendingStatus`.

The `MTTransportAnswer` function is described next.

The `MTTransportDisconnect` function is described on page 9-28.

The `MTTransportRegisterName` function is described on page 9-53.

Completion routines are discussed on page 9-57.

MTTransportAnswer

The `MTTransportAnswer` function answers a call initiated by a remote caller. You listen for an incoming call using the `MTTransportListen` function.

```
pascal ComponentResult MTTransportAnswer (MTTransportComponent c,
                        const MTAddress *remoteAddr,
                        MTTransportComponent listener,
                        Boolean async, MTCompletionUPP
                        completion, long refCon);
```

`c`	The transport component you are using.
`remoteAddr`	A pointer to the MovieTalk address describing the network address of the remote caller. You get this value from the `MTTransportListen` function.
`listener`	The transport component that listened for the call. If you are using the same transport component to answer the call and listen for the call, set this parameter to `nil`. If you used a different transport component to listen for the call, set this parameter to the listener's component instance.
`async`	A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. If you set the `async` parameter to `true`, the transport component calls this routine when it has answered the incoming call. If you set the `async` parameter to `false`, the transport component ignores this parameter. If you do not provide a completion routine, set this parameter to `nil`.
`refCon`	Reference constant for your use. The transport component passes this value to your completion routine.

DESCRIPTION

You call the `MTTransportAnswer` function to answer a remote caller's attempt to make a call. Once another endpoint tries to reach you, you must either answer the incoming call by calling this function or reject it by calling the `MTTransportDisconnect` function before you may listen for another incoming call.

The most efficient way to answer a call is to specify asynchronous operation and provide a completion routine. In that case, the `MTTransportAnswer` function calls your completion routine when it has answered the call or detected an error.

If you don't provide a completion routine for an asynchronous call, you can call the `MTTransportGetStatus` function in your event loop to poll the call status. After calling the `MTTransportAnswer` function asynchronously but before it completes execution, the `MTTransportGetStatus` function reports a status of `mtTransportAnswerPendingStatus`. When the `MTTransportAnswer` function completes successfully (that is, the call is established), the `MTTransportGetStatus` function reports a status of `mtTransportOpenedStatus`. If the `MTTransportAnswer`

function fails to complete successfully, the `MTTransportGetStatus` function reports a status of `mtTransportFailedStatus`.

The `listener` parameter allows you to respond to incoming calls using a different transport component than you use to answer the calls. This can be useful in some networked environments. For example, socket-based TCP/IP networks define several well-known sockets to receive incoming calls. However, the traffic that results from these calls is typically carried over different sockets. In such an environment, you might want to have one transport component listen for incoming calls and another support the call once it is established.

SPECIAL CONSIDERATIONS

Calling the `MTTransportAnswer` function synchronously is not recommended. If you do so, no other operations can take place on the computer until the answer operation completes or fails.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous answer operation already in progress
`mtIncompatibleStateErr`	–7962	Cannot answer in the current state

In addition to these result codes, the `MTTransportAnswer` function can return network component errors.

SEE ALSO

The `MTTransportGetStatus` function is described on page 9-43.

The `MTTransportDisconnect` function is discussed on page 9-28.

Completion routines are discussed on page 9-57.

MTTransportCall

The `MTTransportCall` function initiates a call to a remote listener.

```
pascal ComponentResult MTTransportCall (MTTransportComponent c,
                        const MTAddress *remoteAddr,
                        Boolean async, MTCompletionUPP
                        completion, long refCon);
```

`c` The transport component you are using.

`remoteAddr` A pointer to the remote address you want to call. You can get a MovieTalk address from the `MTTransportLookupName` function.

`async`	A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. If you set the `async` parameter to `true`, the transport component calls this routine when the function completes. If you set the `async` parameter to `false`, the transport component ignores this parameter. If you do not provide a completion routine, set this parameter to `nil`.
`refCon`	Reference constant for your use. The transport component passes this value to your completion routine.

DESCRIPTION

You call the `MTTransportCall` function to initiate a call with a remote endpoint.

The most efficient way to issue a call is to specify asynchronous operation and provide a completion routine. In that case, the `MTTransportCall` function calls your completion routine when it completes execution.

If you don't provide a completion routine for an asynchronous call, you can call the `MTTransportGetStatus` function in your event loop to poll the call status. After calling the `MTTransportCall` function asynchronously but before it completes, the `MTTransportGetStatus` function reports a status of `mtTransportOpenPendingStatus`. When the operation completes successfully, the `MTTransportGetStatus` function reports either a status of `mtTransportOpenedStatus` (if the called party accepted the call) or `mtTransportFailedStatus` (if the called party refused the call). If the `MTTransportCall` function fails to complete successfully, the `MTTransportGetStatus` function returns a status of `mtTransportFailedStatus`.

SPECIAL CONSIDERATIONS

Calling the `MTTransportCall` function synchronously is not recommended. If you do that, no other operations can take place on the computer until the operation completes or fails.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous call operation already in progress
`mtIncompatibleStateErr`	–7962	Cannot call in the current state

In addition to these result codes, the `MTTransportCall` function can return network component errors.

SEE ALSO

You can obtain a MovieTalk address from the `MTTransportLookupName` function, described on page 9-55.

The `MTTransportGetStatus` function is described on page 9-43.

Completion routines are discussed on page 9-57.

MTTransportDisconnect

The `MTTransportDisconnect` function disconnects an existing call, cancels a listen operation, or rejects an incoming call.

```
pascal ComponentResult MTTransportDisconnect
                        (MTTransportComponent c, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`c`	The transport component you are using.
`async`	A Boolean value that indicates whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. If you set the `async` parameter to `true`, the transport component calls this routine when the function completes. If you set the `async` parameter to `false`, the transport component ignores this parameter. If you do not provide a completion routine, set this parameter to `nil`.
`refCon`	Reference constant for your use. The transport component passes this value to your completion routine.

DESCRIPTION

You call the `MTTransportDisconnect` function to disconnect an existing call or to cancel a listen operation you initiated with the `MTTransportListen` function.

Either endpoint can disconnect an existing call.

When you call this function to disconnect an existing call, the transport component sends a CloseConnection message to the remote end, signaling your wish to close down the call.

The most efficient way to disconnect a call is to specify asynchronous operation and provide a completion routine. In that case, the `MTTransportDisconnect` function calls your completion routine when it completes execution.

If you don't provide a completion routine for an asynchronous call, you can call the `MTTransportGetStatus` function in your event loop to poll the call status. After calling the `MTTransportDisconnect` function asynchronously but before it completes,

the `MTTransportGetStatus` function reports a status of `mtTransportClosePendingStatus`. When the operation completes successfully, the `MTTransportGetStatus` function reports a status of `mtTransportClosedStatus`. If the `MTTransportDisconnect` function fails to complete successfully, the `MTTransportGetStatus` function returns a status of `mtTransportFailedStatus`.

SPECIAL CONSIDERATIONS

Calling the `MTTransportDisconnect` function synchronously is not recommended. If you do so, no other operations can take place on the computer until the operation completes or fails.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous disconnect operation already in progress
`mtIncompatibleStateErr`	–7962	Cannot disconnect in the current state

In addition to these result codes, the `MTTransportDisconnect` function can return network component errors.

SEE ALSO

For more information about the CloseConnection message, see the chapter "MovieTalk Protocol Messages" in this book.

The `MTTransportGetStatus` function is described on page 9-43.

The `MTTransportListen` function is described on page 9-23.

Completion routines are discussed on page 9-57.

Sending and Receiving Media Data

This section describes the functions that transport components provide that allow you to send and receive media data.

You send media data by calling the `MTTransportSendMediaData` function. You provide a media data routine to the transport component by calling the `MTTransportSetMediaDataProc` function; the transport component calls your media data routine whenever it receives media data for you. Once you are done with the received media data, you need to free the buffers that contain the data. Call the `MTTransportGetReleaseProc` function to obtain a pointer to the transport component routine that frees media data buffers.

MTTransportSendMediaData

The `MTTransportSendMediaData` function sends media data across a QuickTime Conferencing call.

```
pascal ComponentResult MTTransportSendMediaData
                        (MTTransportComponent c,
                        const MTStreamChunkRecord *chunk,
                        Boolean async, MTCompletionUPP
                        completion, long refCon);
```

`c`	The transport component you are using.
`chunk`	A pointer to a chunk record. You provide the structure. In it, you set the stream ID, a pointer to the chunk data, and information about the data, such as its length in bytes, its priority level, its timestamp, and so forth. You are responsible for disposing of the record when the transport component is done with it.
`async`	A Boolean value that indicates whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. If you set the `async` parameter to `true`, the transport component calls this routine when the function completes. If you set the `async` parameter to `false`, the transport component ignores this parameter. If you do not provide a completion routine, set this parameter to `nil`.
`refCon`	Reference constant for your use. The transport component passes this value to your completion routine.

DESCRIPTION

You call the `MTTransportSendMediaData` function to send media data to a remote endpoint.

If you call the `MTTransportSendMediaData` function asynchronously, you should not modify the media data or free its memory until the function completes.

SPECIAL CONSIDERATIONS

The `MTTransportSendMediaData` function does not guarantee that the data will be delivered to the receiving end. If the underlying network uses a best effort protocol for media data, some or all of the data that was transmitted may be lost in the network.

The receiving end can determine if data loss has occurred by examining the `chunkError` and `chunkFrameNumber` fields in the chunk record. The remote transport component passes a chunk record to the receiving application's media data routine after it assembles a chunk from data it receives from the network.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Too many asynchronous send-data operations already in progress

In addition to these result codes, the `MTTransportSendMediaData` function can return network component errors.

SEE ALSO

You can install a media data routine by calling the `MTTransportSetMediaDataProc` function, described next.

Completion routines are discussed on page 9-57.

MTTransportSetMediaDataProc

The `MTTransportSetMediaDataProc` function installs or removes a media data routine.

```
pascal ComponentResult MTTransportSetMediaDataProc
                        (MTTransportComponent c,
                        MTMediaUPP proc, long refCon);
```

`c`	The transport component you are using.
`proc`	A pointer to your media data routine. Set this parameter to `nil` to remove a media data routine you installed previously.
`refCon`	Reference constant for your use. The transport component passes this value to your media data routine.

DESCRIPTION

You call the `MTTransportSetMediaDataProc` function to install a media data routine. This routine is valid only for the specified call. The transport component calls your media data routine at interrupt time.

The transport component calls your routine when it assembles a chunk of media data from an incoming network stream. The transport component passes your routine a pointer to a chunk record. It is up to you to present the incoming media data appropriately.

You can remove a previously installed media data routine by setting the `proc` parameter to `nil`.

This function is typically called by sink stream director components.

RESULT CODES

`noErr`	0	No error

SEE ALSO

Media data routines are described on page 9-59.

MTTransportGetReleaseProc

The `MTTransportGetReleaseProc` function provides you with a pointer to a transport component routine that releases the memory allocated to chunks of incoming media data.

```
pascal ComponentResult MTTransportGetReleaseProc
                        (MTTransportComponent c,
                        MTReleaseUPP *proc);
```

`c`	The transport component you are using.
`proc`	Address of a location to receive a function pointer. The transport component returns a pointer to its routine that releases media data memory.

DESCRIPTION

You call the `MTTransportGetReleaseProc` function to get the address of a routine that you need to call when you want to release the memory assigned to a chunk of incoming media data. The release routine is a regular C routine belonging to the transport component that you specify. The release routine supports the following interface (the name is arbitrary):

```
pascal void ReleaseProc(MTStreamChunkRecordPtr chunk);
```

When a transport component receives incoming media data from a network component, it allocates the memory necessary to assemble a chunk of media data. Then it calls your media data routine (installed by calling the `MTTransportSetMediaDataProc` function) and passes the chunk to your routine.

When you no longer need the media data, you should call the transport component's release routine to dispose of the memory.

Typically, sink stream director components call the `MTTransportGetReleaseProc` function.

RESULT CODES

`noErr`	0	No error

SEE ALSO

The `MTTransportSetMediaDataProc` function is described on page 9-31.

Sending and Receiving Call Control Messages

This section discusses the transport component routines that allow you to send and receive control messages.

Use the `MTTransportSendMessage` function to send control messages. You supply one or more message routines to receive incoming control messages. You can manage these message routines with the `MTTransportSetMessageProc` and `MTTransportGetMessageProc` functions.

MTTransportSendMessage

The `MTTransportSendMessage` function sends a fixed-length message to a remote endpoint via a reliable control channel.

```
pascal ComponentResult MTTransportSendMessage
                        (MTTransportComponent c,
                        const MTMessageHeader *message,
                        Boolean async, MTCompletionUPP
                        completion, long refCon);
```

`c`	The transport component you are using.
`message`	A pointer to the message that you want to send to the remote end.
`async`	A Boolean value that indicates whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. If you set the `async` parameter to `true`, the transport component calls this routine when the function completes. If you set the `async` parameter to `false`, the transport component ignores this parameter. If you do not provide a completion routine, set this parameter to `nil`.
`refCon`	Reference constant for your use. The transport component passes this value to your completion routine.

DESCRIPTION

You call the `MTTransportSendMessage` function to send a fixed-length message to a remote endpoint.

Your message should be formatted as a MovieTalk message. A MovieTalk message consists of a MovieTalk message header followed by the message data.

SPECIAL CONSIDERATIONS

The maximum length of a MovieTalk message is determined by the underlying network protocol. You can determine the maximum message length supported by the current call by calling the `MTTransportGetInfo` function and using the `mtControlSizeSelector` selector value.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Too many asynchronous send-message operations already in progress

In addition to these result codes, the `MTTransportSendMessage` function can return network component errors.

SEE ALSO

For more information about the format of MovieTalk messages, see page 9-20.

The `MTTransportGetInfo` function is discussed on page 9-48.

Completion routines are discussed on page 9-57.

MTTransportSetMessageProc

The `MTTransportSetMessageProc` function installs or removes a message routine.

```
pascal ComponentResult MTTransportSetMessageProc
                        (MTTransportComponent c,
                        MTMessageUPP proc, long refCon);
```

`c`	The transport component you are using.
`proc`	A pointer to your message routine. Set this parameter to `nil` to remove a message routine you installed previously.
`refCon`	Reference constant for your use. The transport component passes this value to your message routine. If you are removing a message routine, you must set this parameter to the same value you provided when you installed the routine.

DESCRIPTION

You call the `MTTransportSetMessageProc` function to install a message routine. This routine supports the current call.

The transport component calls your routine when it receives a MovieTalk message on the control channel. It passes a pointer to the message to your routine. Make a copy of the message if you need to. Note that the transport component calls your routine at idle time (not at interrupt time). As a result, you are free to allocate or move memory in your routine.

You can install any number of message routines by calling the `MTTransportSetMessageProc` function repeatedly, each time providing a pointer to a different routine. The transport component maintains a list of installed message routines.

When you install more than one message routine, the transport component calls them in the order in which you installed them until one of them handles the message. If a message routine does not handle a message passed to it, it should return the `mtMessageNotHandledErr` result code. If no routine handles the message, the transport component disposes of the message.

RESULT CODES

`noErr`	0	No error

SEE ALSO

Message routines are described on page 9-60.

The format of MovieTalk messages is described on page 9-20.

You can retrieve a pointer to each installed message routine with the `MTTransportGetMessageProc` function, described next.

MTTransportGetMessageProc

The `MTTransportGetMessageProc` function gets one message routine and its associated reference constant from a list of message routines.

```
pascal ComponentResult MTTransportGetMessageProc
                        (MTTransportComponent c,
                        MTMessageUPP *proc, long *refCon);
```

`c`	The transport component you are using.
`proc`	Address of a location to receive a function pointer. The transport component returns a routine's address here. When you reach the end of the list, it sets the location to `nil`.

You can use this location to tell the transport component where to start in its list of message routines. To get the first message routine in the list, set the location to `nil`. If you set it to point to a message routine that you previously installed, the transport component uses that as the starting point. It returns the address of the next message routine you installed.

`refCon` A pointer to a reference constant. The transport component returns the value associated with the routine it returns in the `proc` parameter.

If you are using the `proc` parameter to specify a message routine, you must provide the appropriate reference constant value, too.

DESCRIPTION

The `MTTransportGetMessageProc` function returns a pointer to a message routine and its associated reference constant.

The transport component maintains a list of message routines that you have installed. You can use the `MTTransportGetMessageProc` function to get a pointer to each installed message routine and its associated reference constant. To start at the beginning of the list, set the locations referred to by the `proc` parameter and the `refCon` parameter to `nil`. The function returns a pointer to the first installed message routine in the list and its associated reference constant. To cycle through the entire list, call the function repeatedly, leaving the previous routine's address and reference constant in the appropriate locations.

When you reach the end of the list, the function sets the locations to `nil`.

Since you can install the same routine more than once, using different reference constants, you need to provide values in both the `proc` and `refCon` parameters to uniquely identify an installed message routine. The function uses the pair of values to identify a message routine.

Stream director components, conference components, and flow control components often install message routines on the same transport component.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid procedure pointer

SEE ALSO

Message routines are described on page 9-60.

You use the `MTTransportSetMessageProc` function, described on page 9-34, to install or remove a message routine.

Managing Streams

Transport components provide a number of functions that allow you to work with media streams.

You can create a new media stream by calling the `MTTransportNewMediaStream` function; you dispose of a stream when you are done with it by calling the `MTTransportDisposeMediaStream` function. The `MTTransportAttachMediaStream` function attaches a stream to a call, allowing you to receive media data. The `MTTransportEnableStream` and `MTTransportIsStreamEnabled` functions allow you to control media streams. The `MTTransportGetStreamPerformance` function returns information about a stream's throughput.

MTTransportNewMediaStream

The `MTTransportNewMediaStream` function creates a new stream to carry outgoing media data.

```
pascal ComponentResult MTTransportNewMediaStream
                        (MTTransportComponent c,
                        MTStreamID *stream, Boolean enable,
                        MTReservationPtr resources);
```

`c`	The transport component that is to create the new stream.
`stream`	Location to receive the new stream's unique identifier.
`enable`	A Boolean value indicating whether the stream should be enabled or disabled when it is created. Set this parameter to `true` to enable the new stream; set it to `false` to disable the stream.
`resources`	Specifies bandwidth reservation parameters. The transport component updates some of the fields in this structure to reflect the result of your request. Set this paramater to `nil` if you are not reserving bandwidth.

DESCRIPTION

You call the `MTTransportNewMediaStream` function to create a new stream that is to carry outgoing media data. The transport component creates a new media data stream and returns that stream's unique identifier in the location referred to by the `stream` parameter.

Typically, source stream director components call this function.

You can use this function to reserve network bandwidth when you are using a network that supports bandwidth reservation. You can determine whether a network supports bandwidth reservation by calling the `MTTransportGetInfo` function and using the `mtHasReservationSelector` selector. When you reserve network bandwidth, you do so using flow specifications. Transport components encapsulate flow specifications in records called reservation blocks.

The reservation block record is defined as follows:

```
struct MTReservation {
   MTReserveType        flowSpecType;
   MTBitsPerSecond      bandwidth;
   MTReservationID      identity;
   MTChannel            channel;
};
```

Field descriptions

`flowSpecType` Specifies the type of flow specification. The value of this field allows the transport component to discriminate between different flow specifications. This must be the first field in any flow specification used with a transport component. You must set this field to `kMTReserveType`.

`bandwidth` Bandwidth, in bits per second. You use this field to indicate the amount of bandwidth you desire. In response, the transport component returns a value indicating the amount of bandwidth remaining after your request. This returned value is valid even after failed reservation requests.

`identity` Identifies a reservation. The component assigns a unique identifier to each reservation request you make. The transport component supplies this value when it returns to your application.

`channel` Network-specific media channel identifier. This value identifies a network-specific media channel that is carrying your bandwidth. This is not the same as a MovieTalk channel; it is relevant only to the transport component.

In response to your reservation request, the transport component returns this value.

RESULT CODES

`noErr`	0	No error
`mtIncompatibleStateErr`	–7962	No call is set up

SEE ALSO

The `MTTransportGetInfo` function is discussed beginning on page 9-48.

To dispose of a media stream, you call the `MTTransportDisposeMediaStream` function, described next.

MTTransportDisposeMediaStream

The `MTTransportDisposeMediaStream` function disposes of a media data stream.

```
pascal ComponentResult MTTransportDisposeMediaStream
                        (MTTransportComponent c, MTStreamID id);
```

c	The transport component you are using.
id	The stream ID of the stream you want to delete. You get a stream ID from the MTTransportNewMediaStream function.

DESCRIPTION

You call the MTTransportDisposeMediaStream function to dispose of a stream. If you reserved network bandwidth when you created the stream, the transport component releases that bandwidth for you.

You should dispose of both incoming and outgoing streams when you are done with them. The Component Manager's CloseComponent function disposes of all of your streams for you.

RESULT CODES

noErr	0	No error
mtInvalidStreamIDErr	–7869	Invalid stream ID

SEE ALSO

You create an outgoing stream by calling the MTTransportNewMediaStream function, described on page 9-37.

You attach an incoming steam by calling the MTTransportAttachMediaStream function, described next.

MTTransportAttachMediaStream

The MTTransportAttachMediaStream function attaches an incoming stream to a transport component. This allows you to receive media data.

```
pascal ComponentResult MTTransportAttachMediaStream
                        (MTTransportComponent c, MTStreamID id,
                        Boolean enable);
```

c	The transport component you are using.
id	The stream ID of the stream you want to attach. You get the stream ID from the remote end via the control channel.
enable	A Boolean value indicating whether the stream should be enabled or disabled when it is attached. Set this to true to enable the stream; set it to false to disable the stream.

DESCRIPTION

You call the `MTTransportAttachMediaStream` function to attach an incoming stream to a transport component.

Typically, a sink stream director component calls this function.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Stream ID is in use

MTTransportEnableStream

The `MTTransportEnableStream` function enables or disables one or more streams.

```
pascal ComponentResult MTTransportEnableStream
                        (MTTransportComponent c, MTStreamID id,
                        Boolean enable);
```

`c`	The transport component you are using.
`id`	The stream's unique identifier. You get a stream ID when you use the `MTTransportNewMediaStream` function to create a stream. To enable or disable all streams, set this parameter to `kMTAllStreams`.
`enable`	A Boolean value indicating whether you want to enable or disable a stream. Set this parameter to `true` to enable a stream. Set it to `false` to disable a stream.

DESCRIPTION

You call the `MTTransportEnableStream` function to enable or disable either a single stream that you specify or all streams managed on your behalf by a given transport component.

When you disable an outgoing stream, the transport component stops sending data for that stream to the remote endpoint. The transport component starts sending data again when you enable the stream.

When you disable an incoming stream, the transport component discards data it receives from the remote endpoint. The transport component continues to discard the data until you reenable the stream.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID

SEE ALSO

You can determine a stream's status by calling the `MTTransportIsStreamEnabled` function, described next.

MTTransportIsStreamEnabled

The `MTTransportIsStreamEnabled` function returns a stream's enabled status.

```
pascal ComponentResult MTTransportIsStreamEnabled
                        (MTTransportComponent c, MTStreamID id,
                        Boolean *enabled);
```

`c`	The transport component you are using.
`id`	The stream's unique identifier. You get a stream ID when you use the `MTTransportNewMediaStream` function to create a new stream. If you want to know the status of all the streams managed on your behalf by this transport component, specify the constant `kMTAllStreams`.
`enabled`	Address of the location to receive a Boolean indicating whether the specified stream is enabled.

DESCRIPTION

You call the `MTTransportIsStreamEnabled` function to determine if an individual stream or all streams (both incoming and outgoing) are enabled or disabled.

The function returns `true` if the stream is enabled. It returns `false` if the stream is disabled.

When you ask for the status of all streams (that is, you set the `id` parameter to `kMTAllStreams`), the function returns `true` if at least one stream is enabled.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID

SEE ALSO

You enable and disable streams by calling the `MTTransportEnableStream` function, described on page 9-40.

MTTransportGetStreamPerformance

The `MTTransportGetStreamPerformance` function provides you with information about a stream's performance.

```
pascal ComponentResult MTTransportGetStreamPerformance
                        (MTTransportComponent c, MTStreamID id,
                        MTStreamPerformance *performance);
```

`c` The transport component you are using.

`id` The stream's unique identifier.

`performance`
A pointer to an `MTStreamPerformance` structure. The function fills in the fields of the structure with its performance information. You allocate the structure and must dispose of it when you are done.

DESCRIPTION

You call the `MTTransportGetStreamPerformance` function to determine a stream's performance characteristics, including the number of packets received, sent, and lost.

The function returns running totals for all performance values. If you want periodic information (for example, the number of packets lost every 5 seconds), you need to call `MTTransportGetStreamPerformance` periodically and compute the differences.

The `MTTransportGetStreamPerformance` function returns the performance information in a performance information record:

```
struct MTStreamPerformance {
   UInt32               numPacketsReceived;
   UInt32               numPacketsLost;
   UInt32               numPacketsSent;
   UInt32               numBytesReceived;
   UInt32               numBytesSent;
};
typedef struct MTStreamPerformance MTStreamPerformance;
```

Field descriptions

`numPacketsReceived`
For an incoming stream, the number of packets received since this stream was created. It is always 0 for an outgoing stream.

`numPacketsLost` For an incoming stream, the number of packets sent by the remote endpoint but not received locally since this stream was created. For an outgoing stream, this is always 0.

`numPacketsSent` For an outgoing stream, the number of packets sent to the remote endpoint since this stream was created. For an incoming stream, this is always 0.

`numBytesReceived`
: For an incoming stream, the number of bytes received since this stream was created. For an outgoing stream, this is always 0.

`numBytesSent`
: For an outgoing stream, the number of bytes sent since this stream was created. For an incoming stream, this is always 0.

RESULT CODES

`noErr`	0	No error
`mtInvalidStreamIDErr`	–7869	Invalid stream ID

Managing a Call

Once you have established a call, you need to manage it. Transport components provide a number of functions that help you do so.

The `MTTransportGetStatus` function allows you to retrieve status information about a call. The `MTTransportGetAddress` function retrieves network address information about either of a call's endpoints. You can retrieve information about the transport component and the network component it is using by calling the `MTTransportGetInfo` function. You can manage your notification routine with the `MTTransportSetNotificationProc` and `MTTransportGetNotificationProc` functions.

MTTransportGetStatus

The `MTTransportGetStatus` function retrieves status information about a QuickTime Conferencing call.

```
pascal ComponentResult MTTransportGetStatus
                        (MTTransportComponent c,
                        MTTransportStatusType *status);
```

`c`
: The transport component you are using.

`status`
: A pointer to a location that is to receive the status information. The function returns one of the following values:

  `mtTransportErrorStatus`
  : No network services are available—the network is offline or disconnected. For example, this state occurs while an AppleTalk network connection is switched from TokenTalk to EtherTalk. A call may enter this state at any time.

When a call enters the error state, the transport component disposes of all previous setup information; any ongoing operations are terminated. Upon leaving the error state, you need to restore all network-related operations, such as registering a network name or activating a listen operation.

The transport component calls your notify routine with a network offline event (`mtTransportOfflineEvent`) before putting your call into the error state. When network services are restored, the component calls your notify routine with a network online event (`mtTransportOnlineEvent`).

`mtTransportFailedStatus`
An error occurred that has caused the call to be broken, but network services remain available. For example, this state can occur if the remote endpoint has a hardware or software failure, but the network itself remains operational. A call may enter this state at any time.

In order to recover from this state, you must call the `MTTransportDisconnect` function.

`mtTransportClosedStatus`
No call-related operation is in progress, but network services are available. A call may enter this state at any time.

`mtTransportOpenedStatus`
A call is established and operational. A call may enter this state at any time.

`mtTransportCalledStatus`
An incoming call is detected. A component that is listening for a call (that is, it is in the `mtTransportListenPendingStatus` state) enters this state when the transport component detects a call.

`mtTransportOpenPendingStatus`
The local endpoint is waiting for the remote end to answer a call. A call may enter this state after you have originated the call.

`mtTransportListenPendingStatus`
The local endpoint is listening for an incoming call. An endpoint may enter this state after you instruct the transport component to listen for an incoming call.

`mtTransportAnswerPendingStatus`
The local endpoint has answered an incoming call, but the call is not yet established. A component may enter this state when it receives a call.

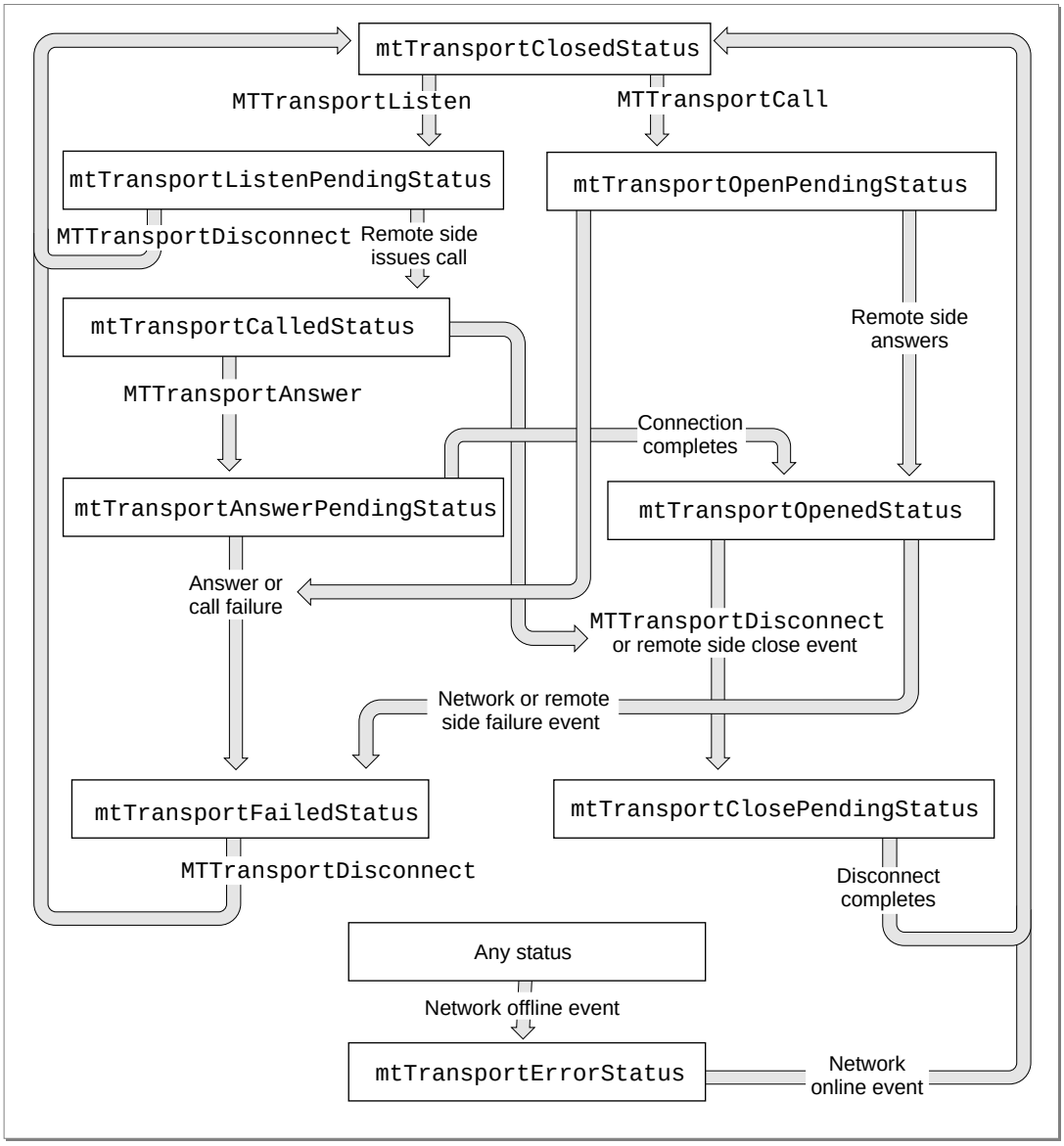
`mtTransportClosePendingStatus`
The call is closing. Either side may close a call. A call may enter this state at any time.

DESCRIPTION

Transport components maintain state information for every QuickTime Conferencing call. You call the `MTTransportGetStatus` function to get the state of the current call. In some cases, the transport component also calls your notify routine with network-related events that cause state transitions.

At first, when no call is established or pending, the state is closed (`mtTransportClosedStatus`). After that, the state changes when you call any of the call management functions (such as the `MTTransportCall` function). If you use a call management function asynchronously, another state transition occurs when the function completes. Figure 9-2 shows how your requests and network events affect the transport component's status.

Figure 9-2 Transport component status transitions



Both the originating and answering endpoints can call this function.

RESULT CODES

noErr	0	No error

SEE ALSO

The call management functions are described in the section "Establishing and Terminating a QuickTime Conferencing Call" beginning on page 9-22.

MTTransportGetAddress

The MTTransportGetAddress function obtains the network address of either a local or remote transport component.

```
pascal ComponentResult MTTransportGetAddress
                        (MTTransportComponent c,
                        MTTransportAddressType addressType,
                        MTAddress *address);
```

c The transport component you are using.

addressType
The type of address you want the function to return. You specify the address types with one of these constants:

mtTransportRemoteMediaAddress
The remote endpoint's media address.

mtTransportRemoteControlAddress
The remote endpoint's control address.

mtTransportLocalMediaAddress
The local endpoint's media address.

mtTransportLocalControlAddress
The local endpoint's control address.

mtTransportLocalListenerAddress
The local endpoint's listener address. This address is used by the MTTransportRegisterName function.

address A pointer to a MovieTalk address record. The function returns the appropriate network address in this structure.

DESCRIPTION

You call the MTTransportGetAddress function to get a network address. You can get one of three address types—media, control, or listener—for a call's local or remote end. You specify the type of address by setting the addressType parameter appropriately.

If no active call exists when you call the MTTransportGetAddress function and ask for a remote endpoint address, the function returns a result code of mtNoTransportErr.

A transport component's addresses are the same as those of the network component it is using.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad address type
`mtNoTransportErr`	–7910	No network component

MTTransportGetInfo

The `MTTransportGetInfo` function provides information about a transport component and the network component it is using.

```
pascal ComponentResult MTTransportGetInfo
                        (MTTransportComponent c,
                        MTInfoSelector whichInfo,
                        long *result);
```

`c` The transport component you are using.

`whichInfo` The type of information to retrieve. The following values are valid:

`mtNetworkAttachedSelector`
The transport component tells you whether it detects a functional network. The transport component returns a Boolean value. If the returned value is `true`, then the transport component detects a functioning network connection.

`mtPayloadSizeSelector`
The transport component tells you the maximum payload size it can accommodate. The transport component returns an `MTMessageSize` value that specifies the maximum acceptable media payload size, in bytes. This size value indicates the maximum number of data bytes you can expect to transmit in a single media data packet.

`mtFlowBandwidthSelector`
The transport component tells you the maximum effective throughput of your call's media stream. The transport component returns an `MTBitsPerSecond` value that specifies the effective throughput, in bits per second.

`mtFlowControlSelector`
The transport component tells you whether the network component is requesting flow control on your call's media stream. The transport component returns a Boolean value. The returned value is set to `true` if the network

component wants flow control on the stream. Flow control components use this value to determine when to enable and disable flow control.

`mtIsBidirectionalSelector`
The transport component tells you whether the media call is bidirectional. The transport component returns a Boolean value. If the call is bidirectional, the transport component returns a value of `true`.

`mtIsMulticastSelector`
The transport component tells you whether the media call is multicast. The transport component returns a Boolean value. If the call is multicast, the transport component returns a value of `true`.

`mtIsUnsegmentedSelector`
The transport component tells you whether it is using a unsegmented mode network connection. An unsegmented mode connection does not segment media data. The transport component returns a Boolean value. If your call is running over an unsegmented mode connection, the transport component returns a value of `true`.

`mtHasReservationSelector`
The transport component tells you whether the underlying network supports bandwidth reservation. The transport component returns a Boolean value. If the network supports bandwidth reservation, the transport component returns a value of `true`. If the network supports bandwidth reservation, you can reserve network bandwidth when you call the `MTTransportNewMediaStream` function.

`mtControlSizeSelector`
The transport component tells you the maximum message buffer size you can use on your call's control channel. The transport component returns an `MTMessageSize` value. This returned value specifies the maximum control message buffer size, in bytes. You send control messages by calling the `MTTransportSendMessage` function.

`mtMediaSizeSelector`
The transport component tells you the maximum packet size you can use on your call's media channel. The transport component returns an `MTMessageSize` value. This returned value specifies the maximum media packet size, in bytes. This size value includes both header and payload information.

`mtMulticastTypeSelector`
The transport component tells you the component subtype value assigned to its multicast twin. A given transport component supports either multicast or point-to-point traffic, but not both. If you are working with a

point-to-point component and need to use a multicast version of the same transport protocol, you can use this option to locate an appropriate multicast component. The transport component returns an `OSType` value. This value contains the component subtype assigned to transport components that support a multicast version of the point-to-point networked-media protocol that is used by the current transport component.

`mtUnicastTypeSelector`
The transport component tells you the component subtype value assigned to its point-to-point twin. As with the `mtMulticastTypeSelector` type, the transport component returns an `OSType` value. This value contains the component subtype assigned to transport components that support a point-to-point version of the multicast networked-media protocol that is used by the current transport component.

`result` The address of the location to receive the resulting information. The transport component returns different types of data, based on your request.

DESCRIPTION

You call the `MTTransportGetInfo` function to get information about the capabilities and characteristics of the transport component you are using to support a call.

The `MTTransportGetInfo` function takes a parameter that allows you to specify the type of information you want to retrieve. In response, the transport component returns the information to a location you specify.

Depending upon the nature of your request, the result may contain information about the features the component supports with the assistance of an appropriate network component. Such features include multicast and unsegmented data transmission, and flow control. In other cases, the result may indicate characteristics of the network component itself, such as its maximum bandwidth and the maximum sizes of control messages, media packets, and media payloads it can send or receive.

When you call this function, the transport component responds if it can. If the option is meant for a network component and there is no network component attached, the transport component returns an error value of `mtUnsupportedMessageErr`. Otherwise, the transport component passes the request on to the network component and passes the network component's results back to your application.

RESULT CODES

`noErr`	0	No error
`mtUnsupportedMessageErr`	–7747	Invalid option type or no network component attached

MTTransportSetNotificationProc

The `MTTransportSetNotificationProc` function installs a notify routine. The transport component calls your notify routine when certain call-related events occur.

```
pascal ComponentResult MTTransportSetNotificationProc
                        (MTTransportComponent c,
                        MTNotificationUPP notify,
                        long refCon);
```

`c` The transport component you are using.

`notify` A pointer to your notify routine. Set this parameter to `nil` to remove a notify routine you previously installed.

`refCon` Reserved for your use. The transport component passes this value when it calls your notify routine.

DESCRIPTION

You call the `MTTransportSetNotificationProc` function to install a notify routine for a call. The transport component calls your notify routine when a call is broken for any reason and when a network goes online or offline.

The transport component passes your routine an event indicating what happened. This allows you to do whatever processing is appropriate for your application. For example, your response to an orderly disconnect may be different from your response to an unexpected network failure.

RESULT CODES

`noErr`	0	No error

SEE ALSO

Notify routines are discussed on page 9-58. This section also discusses the events that cause the transport component to call your notify routine.

MTTransportGetNotificationProc

The `MTTransportGetNotificationProc` function tells you the notify routine installed for a call.

```
pascal ComponentResult MTTransportGetNotificationProc
                        (MTTransportComponent c,
                        MTNotificationUPP *notify,
                        long *refCon);
```

`c` The transport component you are using.

`notify` Address of a location to receive a pointer to the notify routine you installed previously. The function returns a pointer to the routine. If you haven't installed a notify routine, the function returns a value of `nil`.

`refCon` A pointer to a location to receive the routine's reference constant. The function returns the value. If you haven't installed a notify routine, the function returns a value of `nil`.

DESCRIPTION

You call the `MTTransportGetNotificationProc` function to get the notify routine installed on a transport component.

RESULT CODES

`noErr`	0	No error

SEE ALSO

Notify routines are discussed on page 9-58.

You can install a notify routine with the `MTTransportSetNotificationProc` function, described on page 9-51.

Managing Network Names

You use the following functions to work with network names.

The `MTTransportRegisterName` function registers a name on the underlying network. The `MTTransportRemoveName` function removes a previously registered name. The `MTTransportLookupName` function allows you to retrieve the MovieTalk network address corresponding to a network name. The `MTTransportGetLocalName` function retrieves the network name under which you are registered.

MTTransportRegisterName

The `MTTransportRegisterName` function registers a name on a network.

```
pascal ComponentResult MTTransportRegisterName
                        (MTTransportComponent c,
                        OSType networkType, ConstStr255Param name,
                        ConstStr255Param serviceType,
                        Boolean async, MTCompletionUPP completion,
                        long refCon);
```

`c` The transport component you are using.

`networkType`
The type of network on which to register the name. The value of this parameter should correspond to the component subtype value assigned to the network component that supports the network type.

`name` A pointer to a string containing the network name you want to register. The name can be a user's name or a name that identifies the local computer. Its format is up to you, though it must comply with any naming rules imposed by the underlying network.

`serviceType`
A pointer to a string containing the type of network service you want to support. You determine the type of service and the string that identifies the service. The type depends on the type of call or service you are providing and should be meaningful to and recognizable by your application.

`async` A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.

`completion` A pointer to your completion routine. If you set the `async` parameter to `true`, the transport component calls this routine when the function completes. If you set the `async` parameter to `false`, the transport component ignores this parameter. If you do not provide a completion routine, set this parameter to `nil`.

`refCon` Reference constant for your use. The transport component passes this value to your completion routine.

DESCRIPTION

You call the `MTTransportRegisterName` function to register a name on a network and associate it with the local endpoint's listener address. This allows someone who wants to call you to look up your name and find out the address on which you're listening.

The transport component constructs a name for you, based on the information you specify. Then, if possible and appropriate, it registers the name on the underlying network. You can retrieve the resulting name by calling the `MTTransportGetLocalName` function.

A given transport component instance cannot support more than one registered name at a time. If you previously registered a network name and call the `MTTransportRegisterName` function to register another, the transport component removes the first name before registering the second.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous register name operation already in progress

In addition to the result codes listed here, the `MTTransportRegisterName` function can return network component and Memory Manager errors.

SEE ALSO

You can look up registered names with the `MTTransportLookupName` function, described on page 9-55.

The `MTTransportGetLocalName` function is described on page 9-56.

To remove a name you have registered, call the `MTTransportRemoveName` function, described next.

Completion routines are discussed on page 9-57.

MTTransportRemoveName

The `MTTransportRemoveName` function removes a name you previously registered on a network.

```
pascal ComponentResult MTTransportRemoveName
                        (MTTransportComponent c, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`c`	The transport component you are using.
`async`	A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. If you set the `async` parameter to `true`, the transport component calls this routine when the function completes. If you set the `async` parameter to `false`, the transport component ignores this parameter. If you do not provide a completion routine, set this parameter to `nil`.
`refCon`	Reference constant for your use. The transport component passes this value to your completion routine.

DESCRIPTION

You call the `MTTransportRemoveName` function to remove a previously registered network name.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous remove name operation already in progress

In addition to the result codes listed here, the `MTTransportRemoveName` function can return network component result codes.

SEE ALSO

You register a network name by calling the `MTTransportRegisterName` function, described on page 9-53.

Completion routines are discussed on page 9-57.

MTTransportLookupName

The `MTTransportLookupName` function looks up a name on a network and returns the entity's MovieTalk address.

```
pascal ComponentResult MTTransportLookupName
                        (MTTransportComponent c,
                        const MTName *remoteName,
                        MTAddress *remoteAddr, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`c` The transport component you are using.

`remoteName` A pointer to the network name you want to look up.

`remoteAddr` A pointer to a MovieTalk address record. The function returns the network independent address corresponding to the name you specified.

`async` A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.

`completion` A pointer to your completion routine. If you set the `async` parameter to `true`, the transport component calls this routine when the function completes. If you set the `async` parameter to `false`, the transport component ignores this parameter. If you do not provide a completion routine, set this parameter to `nil`.

`refCon` Reference constant for your use. The transport component passes this value to your completion routine.

DESCRIPTION

You call the `MTTransportLookupName` function to get the MovieTalk address of a named QuickTime Conferencing entity on a network. You can also use this function to determine the address associated with a "well-known" network name. For example, you might use this function to access a well-known named socket on a TCP/IP network. On networks without an inherent name service, the `MTTransportLookupName` function may apply a simple transformation to convert the MovieTalk name to a MovieTalk address.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous name lookup operation already in progress
`mtNameNotFoundErr`	–7972	Name not found

SEE ALSO

Completion routines are discussed on page 9-57.

MTTransportGetLocalName

The `MTTransportGetLocalName` function obtains the network name under which you are currently registered.

```
pascal ComponentResult MTTransportGetLocalName
                        (MTTransportComponent c,
                        MTName *localName);
```

`c` The transport component you are using.

`localName` A pointer to a MovieTalk name record. The function returns your network name.

DESCRIPTION

You call the `MTTransportGetLocalName` function to obtain your network name.

This function returns the network name that you create when you call the `MTTransportRegisterName` function. If you have not registered a network name, the `MTTransportGetLocalName` function returns the `mtNameNotFoundErr` result code.

A transport component's name is the same as that of the network component it is using.

RESULT CODES

`noErr`	0	No error
`mtNameNotFoundErr`	–7972	Name not found

SEE ALSO

The `MTTransportRegisterName` function is described on page 9-53.

Application-Defined Functions

This section describes the application-defined routines that you may provide to a transport component. Transport components call all of these routines at interrupt time, except for your message routine.

A transport component calls your completion routine when a function you called asynchronously completes execution.

A transport component calls your notify routine in response to certain network events.

A transport component calls your media data routine when it has assembled a chunk of media data from a media channel.

A transport component calls your message routine when it receives a MovieTalk message on its control channel.

MyCompletionRoutine

When you call a function asynchronously, you can provide a completion routine for the transport component to call when it completes execution.

Your routine must support the following interface (the name is arbitrary):

```
pascal void MyCompletionRoutine (ComponentResult err,
                                 long refCon);
```

`err`	The function's result code.
`refCon`	The reference constant that you provided to the function that calls your completion routine.

DESCRIPTION

You can provide a completion routine to any transport component function that you can call asynchronously. All of call management functions can be called asynchronously. The functions to send media data and control messages and to use or query network name services can also be called asynchronously.

When a function that you called asynchronously completes execution, the transport component calls your completion routine and passes its result code in the `err` parameter. It also passes your reference constant value.

SPECIAL CONSIDERATIONS

Because your completion routine may be called at interrupt time, you must not make calls to the Memory Manager, either directly or indirectly. If your completion routine needs to access global variables, you can pass a reference to your application's A5 register value or to your component's global variables in the `refCon` parameter when you issue the asynchronous function call.

SEE ALSO

For more information about the A5 register, see *Inside Macintosh: Memory*.

MyNotifyProc

When you call the `MTTransportSetNotificationProc` function, you provide a pointer to a notify routine. The transport component calls this routine in response to certain network events, such as an orderly disconnect initiated by the remote endpoint.

Your routine must support the following interface (the name is arbitrary):

```
pascal void MyNotifyProc (MTTransportComponent c,
                          MTTransportEventType event,
                          long refCon);
```

`c`	The transport component calling your notify routine.
`event`	The event that triggered the call to your routine.
`refCon`	The reference constant that you provided in the `refCon` parameter to the `MTTransportSetNotificationProc` function. You can use this value for whatever purpose you choose.

DESCRIPTION

In order to maintain QuickTime Conferencing calls, your application needs to be able to receive notification of network events. Transport components allow you to define a notify routine by calling the `MTTransportSetNotificationProc` function.

Transport components use the `event` parameter to tell you what has happened. The following constants define the events that may cause a transport component to call your notify routine.

Constant descriptions

`mtTransportFailEvent`
: Indicates some failure in the remote side hardware or software or in the network or some other unexpected reason for a disconnect. An existing call is broken.

`mtTransportCloseEvent`
: Indicates an orderly disconnect. An orderly disconnect is originated by the user at the remote end.

`mtTransportOfflineEvent`
: Indicates that no network services are available—the network is offline or disconnected. For example, when a user switches the computer's AppleTalk network connection from TokenTalk to EtherTalk, the network is not available during the transition.

`mtTransportOnlineEvent`
: Indicates that network services are again available.

A transport component calls your notify routine when a call is broken for any reason and when network services become available or unavailable. It passes you information about the event that caused the component to call your routine. Your application can then take whatever actions are appropriate to handle that event.

Because more than one transport component instance may call your notify routine, the transport instance making the call identifies itself in the `c` parameter. For example, a conference component may have multiple transports open and use the same notify routine for all of them.

SPECIAL CONSIDERATIONS

The notify routine is distinct from the completion routine you provide to the `MTTransportDisconnect` function. The transport component calls that completion routine upon the completion of a local disconnect operation, irrespective of the state of the network connection.

SEE ALSO

You install a notify routine by calling the `MTTransportSetNotificationProc` function, described on page 9-51.

MyMediaProc

When you call the `MTTransportSetMediaDataProc` function, you provide a pointer to a media data routine. The transport component calls this routine when it receives data on the media channel.

Your routine must support the following interface (the name is arbitrary):

```
pascal OSErr MyMediaProc (struct MTStreamChunkRecord *chunk);
```

`chunk` A pointer to a chunk record. This structure contains a pointer to a chunk of media data along with certain descriptive information about the data, such as its stream ID, the size of the chunk, its time stamp, and so forth.

DESCRIPTION

A transport component calls your media data routine after it assembles a chunk from media data it received from the remote endpoint. A chunk is a logical unit of media data, such as a video frame. The transport component provides you with a pointer to a chunk record. You then take whatever actions are appropriate to handle the data, such as passing it to a stream player component to present the data.

You install your media data routine on a transport component by calling the `MTTransportSetMediaDataProc` function. At that time you supply a reference constant value that the transport component puts in the `chunkRefCon` field of the chunk record it passes to your routine.

You can determine if a chunk is complete by examining the value of the `chunkError` and `chunkFrameNumber` fields in the chunk record. If the `chunkError` field is set to `noErr`, the chunk is complete—the transport component received all of the network packets containing its media data. If the `chunkError` field is set to `mtChunkIncompleteErr`, the chunk is incomplete; that is, some network packets were lost.

You can detect lost frames by tracking the value of the `chunkFrameNumber` field. The transport component increments the value of this field each time your application calls the `MTTransportSendMediaData` function. If the receiving endpoint detects discontinuous values in this field, the network has dropped a frame.

SEE ALSO

The `MTTransportSetMediaDataProc` function is described on page 9-31.

The chunk record is described earlier in this chaper, in “Chunk Record” on page 9-18.

MyMessageProc

When you call the `MTTransportSetMessageProc` function, you provide a pointer to a message routine. A transport component calls this routine when it receives a message on the control channel.

Unlike the other application-defined functions, the transport component calls your message routine at idle time, not at interrupt time.

Your routine must support the following interface (the name is arbitrary):

```
pascal OSErr MyMessageProc (MTMessageHeader *message,
                            long refCon);
```

`message`	A pointer to a structure containing the MovieTalk message.
`refCon`	The reference constant that you passed in the `refCon` parameter to the `MTTransportSetMessageProc` function. You can use this value for whatever purpose you choose.

DESCRIPTION

A transport component calls your message routine when it receives a message on the control channel. The transport component provides you with a pointer to the MovieTalk message. Your application can then take whatever actions are appropriate to handle that message.

Do not delete the message in your message routine. If you need the message after your routine completes, make a copy of it—the transport component deletes the message after your routine returns.

If your routine does not handle the type of message it is passed, it should return the `mtMessageNotHandledErr` result code. Otherwise, it should return the `noErr` result code.

SEE ALSO

You install your message routine by calling the `MTTransportSetMessageProc` function, described on page 9-34.

The format of MovieTalk messages is described in the chapter "MovieTalk Protocol Messages" elsewhere in this book.

Summary of Transport Components

C Summary

Constants

```
enum {
   kMTAllStreams =                    0
};

/*----------------------------------------
   Transport selectors
----------------------------------------*/
enum {
```

```
   kMTTransportListenSelect =               1,
   kMTTransportAnswerSelect =               2,
   kMTTransportCallSelect =                 3,
   kMTTransportDisconnectSelect =           4,
   kMTTransportSetNotificationProcSelect =  5,
   kMTTransportGetNotificationProcSelect =  6,
   kMTTransportSendMessageSelect =          7,
   kMTTransportSendMediaDataSelect =        8,
   kMTTransportSetMediaDataProcSelect =     9,
   kMTTransportSetMessageProcSelect =       10,
   kMTTransportGetMessageProcSelect =       11,
   kMTTransportGetReleaseProcSelect =       12,
   kMTTransportNewMediaStreamSelect =       13,
   kMTTransportDisposeMediaStreamSelect =   14,
   kMTTransportAttachMediaStreamSelect =    15,
   kMTTransportEnableStreamSelect =         16,
   kMTTransportIsStreamEnabledSelect =      17,
   kMTTransportGetStreamPerformanceSelect = 18,
   kMTTransportRegisterNameSelect =         19,
   kMTTransportRemoveNameSelect =           20,
   kMTTransportLookupNameSelect =           21,
   kMTTransportGetLocalNameSelect =         22,
   kMTTransportGetStatusSelect =            23,
   kMTTransportGetInfoSelect =              24,
   kMTTransportGetAddressSelect =           25
};
```

Data Types

```
typedef ComponentInstance        MTTransportComponent;
typedef OSType                   MTStandardID;
typedef UInt32                   MTMessageSize;
typedef UInt32                   MTChannel;
typedef UInt32                   MTReservationID;

struct MTMessageHeader {
   MTStandardID                  msgType;       // message identifier
   MTMessageSize                 msgSize;       // size of entire message
};
typedef struct MTMessageHeader   MTMessageHeader, *MTMessagePtr;

/*----------------------------------------
   Transport event types
```

```
-----------------------------------------*/
enum {
   mtTransportFailEvent =              0x0001,
   mtTransportCloseEvent =             0x0002,
   mtTransportOfflineEvent =           0x0004,
   mtTransportOnlineEvent =            0x0008
};
typedef UInt32                         MTTransportEventType;

/*-----------------------------------------
   Transport status types
-----------------------------------------*/
enum{
   mtTransportErrorStatus =            -2,
   mtTransportFailedStatus =           -1,
   mtTransportClosedStatus =           0,
   mtTransportOpenedStatus =           1,
   mtTransportCalledStatus =           2,
   mtTransportOpenPendingStatus =      3,
   mtTransportListenPendingStatus =    4,
   mtTransportAnswerPendingStatus =    5,
   mtTransportClosePendingStatus =     6
};
typedef SInt32                         MTTransportStatusType;

/*-----------------------------------------
   Transport/network information
-----------------------------------------*/
enum {
   mtNetworkAttachedSelector =         'ntwk',
   mtPayloadSizeSelector =             'pyld',
   mtFlowBandwidthSelector =           'fbps',
   mtFlowControlSelector =             'flow',
   mtIsBidirectionalSelector =         'bdir',
   mtIsMulticastSelector =             'mcst',
   mtIsUnsegmentedSelector =           'unsg',
   mtHasReservationSelector =          'resv',
   mtControlSizeSelector =             'csiz',
   mtMediaSizeSelector =               'msiz',
   mtMulticastTypeSelector =           'mtyp',
   mtUnicastTypeSelector =             'utyp'
};
typedef OSType                         MTInfoSelector;
```

```
/*----------------------------------------
   Transport address types
----------------------------------------*/
enum {
   mtTransportRemoteMediaAddress =     1,
   mtTransportRemoteControlAddress =   2,
   mtTransportLocalMediaAddress =      3,
   mtTransportLocalControlAddress =    4,
   mtTransportLocalListenerAddress =   5
};
typedef SInt16                         MTTransportAddressType;

/*----------------------------------------
   Transport callback functions
----------------------------------------*/

typedef pascal OSErr (*MTMessageProcPtr) (MTMessageHeader *message,
                                long refCon);

typedef pascal OSErr (*MTMediaProcPtr) (struct MTStreamChunkRecord *chunk);

typedef pascal void (*MTCompletionProcPtr) (ComponentResult err,
                                long refCon);

typedef pascal void (*MTNotificationProcPtr) (MTTransportComponent c,
                                MTTransportEventType event, long refCon);

typedef pascal void (*MTReleaseProcPtr) (struct MTStreamChunkRecord *chunk);

typedef MTMessageProcPtr               MTMessageUPP;
typedef MTMediaProcPtr                 MTMediaUPP;
typedef MTCompletionProcPtr            MTCompletionUPP;
typedef MTNotificationProcPtr          MTNotificationUPP;
typedef MTReleaseProcPtr               MTReleaseUPP;

/*----------------------------------------
   Transport stream data types
----------------------------------------*/

typedef UInt32                         MTChunkSize;
typedef CompTimeValue                  MTChunkTime;
typedef OSType                         MTStreamType;
```

```
typedef UInt32                          MTStreamReference;
typedef UInt16                          MTStreamID;
typedef UInt32                          MTStreamOptions;
typedef UInt32                          MTStreamCount;
typedef UInt8                           MTSequenceNum;
typedef UInt32                          MTBitsPerSecond;

enum {
   mtChunkPrioritySound =               0x0800,
   mtChunkPriorityMedium =              0x0400,
   mtChunkPriorityVideo =               0x0100,
   mtChunkPriorityKeyFrame =            0x0001
};
typedef UInt16                          MTChunkPriority;

struct MTStreamChunkRecord {
   Byte                                 *chunkPtr;
   MTChunkSize                          chunkSize;
   MTChunkTime                          chunkTime;
   MTChunkPriority                      chunkPriority;
   MTStreamID                           chunkStreamID;
   ComponentResult                      chunkError;
   long                                 chunkRefCon;
   UInt32                               chunkReserved;
   long                                 chunkTimePlayed;
   MTReleaseUPP                         chunkReleaseProcCallback;
   long                                 chunkPrivate;
   MTStreamOptions                      chunkStreamOptions;
   UInt8                                chunkReserved2;
   MTSequenceNum                        chunkFrameNumber;
   UInt16                               chunkReserved3;
};
typedef struct MTStreamChunkRecord      MTStreamChunkRecord,
                                        *MTStreamChunkRecordPtr;

struct MTStreamPerformance {
   UInt32                               numPacketsReceived;
   UInt32                               numPacketsLost;
   UInt32                               numPacketsSent;
   UInt32                               numBytesReceived;
   UInt32                               numBytesSent;
};
typedef struct MTStreamPerformance      MTStreamPerformance;
```

```
enum {
   kMTReserveType =                    'mtrs'
};
typedef OSType                         MTReserveType;

struct MTReservation {
   MTReserveType                       flowSpecType;
   MTBitsPerSecond                     bandwidth;
   MTReservationID                     identity;
   MTChannel                           channel;
};
/*----------------------------------------
   Transport independent address
----------------------------------------*/
enum {
   kMTMaxAddressSize =                 256,
   kMTMaxNameSize =                    248,
   kMTMaxEmailAddressSize =            1832,    // should hold a PowerTalk
                                                packed MailRecipient
   kMTMaxDisplayNameSize =             124
};

struct MTAddress {
   OSType                              transportType;
   OSType                              networkType;
   Byte                                networkAddress[kMTMaxAddressSize];
};
typedef struct MTAddress               MTAddress, *MTAddressPtr;

/*----------------------------------------
   Transport independent name
----------------------------------------*/

struct MTName {
   OSType                              transportType;
   OSType                              networkType;
   Byte                                networkName[kMTMaxNameSize];
   struct {
      short                            charSet;
      short                            length;
      Byte                             name[kMTMaxDisplayNameSize];
```

```
   } displayName;                                  // in RString format
   Byte                                 entityInfo[kMTMaxEmailAddressSize];
};
typedef struct MTName                   MTName, *MTNamePtr;
```

Functions

Establishing and Terminating a QuickTime Conferencing Call

```
pascal ComponentResult MTTransportListen (MTTransportComponent c, MTAddress
                            *remoteAddr, Boolean async, MTCompletionUPP
                            completion, long refCon);
pascal ComponentResult MTTransportAnswer (MTTransportComponent c, const
                            MTAddress *remoteAddr, MTTransportComponent
                            listener, Boolean async, MTCompletionUPP
                            completion, long refCon);
pascal ComponentResult MTTransportCall (MTTransportComponent c, const
                            MTAddress *remoteAddr, Boolean async,
                            MTCompletionUPP completion, long refCon);
pascal ComponentResult MTTransportDisconnect (MTTransportComponent c,
                            Boolean async, MTCompletionUPP completion,
                            long refCon);
```

Sending and Receiving Media Data

```
pascal ComponentResult MTTransportSendMediaData (MTTransportComponent c,
                            const MTStreamChunkRecord *chunk, Boolean
                            async, MTCompletionUPP completion, long
                            refCon);
pascal ComponentResult MTTransportSetMediaDataProc (MTTransportComponent c,
                            MTMediaUPP proc, long refCon);
pascal ComponentResult MTTransportGetReleaseProc (MTTransportComponent c,
                            MTReleaseUPP *proc);
```

Sending and Receiving Call Control Messages

```
pascal ComponentResult MTTransportSendMessage (MTTransportComponent c, const
                            MTMessageHeader *message, Boolean async,
                            MTCompletionUPP completion, long refCon);
pascal ComponentResult MTTransportSetMessageProc (MTTransportComponent c,
                            MTMessageUPP proc, long refCon);
pascal ComponentResult MTTransportGetMessageProc (MTTransportComponent c,
                            MTMessageUPP *proc, long *refCon);
```

Managing Streams

```
pascal ComponentResult MTTransportNewMediaStream (MTTransportComponent c,
                            MTStreamID *stream, Boolean enable,
                            MTReservationPtr resources);
pascal ComponentResult MTTransportDisposeMediaStream (MTTransportComponent
                            c, MTStreamID id);
pascal ComponentResult MTTransportAttachMediaStream (MTTransportComponent c,
                            MTStreamID id, Boolean enable);
pascal ComponentResult MTTransportEnableStream (MTTransportComponent c,
                            MTStreamID id, Boolean enable);
pascal ComponentResult MTTransportIsStreamEnabled (MTTransportComponent c,
                            MTStreamID id, Boolean *enabled);
pascal ComponentResult MTTransportGetStreamPerformance (MTTransportComponent
                            c, MTStreamID id, MTStreamPerformance
                            *performance);
```

Managing a Call

```
pascal ComponentResult MTTransportGetStatus (MTTransportComponent c,
                            MTTransportStatusType *status);
pascal ComponentResult MTTransportGetAddress (MTTransportComponent c,
                            MTTransportAddressType addressType, MTAddress
                            *address);
pascal ComponentResult MTTransportGetInfo (MTTransportComponent c,
                            MTInfoSelector whichInfo, long *result);
pascal ComponentResult MTTransportSetNotificationProc (MTTransportComponent
                            c, MTNotificationUPP notify, long refCon);
pascal ComponentResult MTTransportGetNotificationProc (MTTransportComponent
                            c, MTNotificationUPP *notify, long *refCon);
```

Managing Network Names

```
pascal ComponentResult MTTransportRegisterName (MTTransportComponent c,
                            OSType networkType, ConstStr255Param name,
                            ConstStr255Param serviceType, Boolean async,
                            MTCompletionUPP completion, long refCon);
pascal ComponentResult MTTransportRemoveName (MTTransportComponent c,
                            Boolean async, MTCompletionUPP completion,
                            long refCon);
pascal ComponentResult MTTransportLookupName (MTTransportComponent c, const
                            MTName *remoteName, MTAddress *remoteAddr,
                            Boolean async, MTCompletionUPP completion,
                            long refCon);
```

```
pascal ComponentResult MTTransportGetLocalName (MTTransportComponent c,
                              MTName *localName);
```

Application-Defined Functions

```
pascal void MyCompletionRoutine (ComponentResult err, long refCon);
pascal void MyNotifyProc (MTTransportComponent c, MTTransportEventType
                              event, long refCon);
pascal OSErr MyMediaProc (struct MTStreamChunkRecord *chunk);
pascal OSErr MyMessageProc (MTMessageHeader *message, long refCon);
```

Result Codes

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`mtUnsupportedMessageErr`	–7747	Invalid option type or no network component attached
`mtPendingAsyncCallErr`	–7860	Asynchronous listen operation already in progress
`mtInvalidStreamIDErr`	–7869	Invalid stream ID
`mtNoTransportErr`	–7910	No network component
`mtIncompatibleStateErr`	–7962	Cannot listen in the current state
`mtNameNotFoundErr`	–7972	Name not found

Network Components

Contents

This chapter describes network components. **Network components** allow other QuickTime Conferencing components (generally transport components) to access an underlying network and use its communication services. This chapter is divided into the following major sections:

- "About Network Components" introduces you to the features and functions of network components and discusses their relationships with other QuickTime Conferencing components.
- "Using Network Components" provides several examples of how you can use a network component in your application.
- "Creating Network Components" tells you what you need to know before you create your own network component.
- "Network Components Reference" contains detailed technical information about the interface supported by network components.
- "Summary of Network Components" provides a summary of the network component interface.

Network components are Component Manager components. As such, you need to be familiar with components and how to interact with them before you can use or create a network component. If components are new to you, see the chapter that discusses the Component Manager in *Inside Macintosh: More Macintosh Toolbox*.

This chapter is aimed primarily at developers who plan to create their own network components. By creating a new network component, you can allow QuickTime Conferencing applications to work in new network environments. If you plan to create a network component, you should read the entire chapter. Sections in the chapter describe the behavior your component should provide and the interface it must support.

If you are developing a QuickTime Conferencing application, you most likely do not need to work directly with a network component. In most cases, your application should interact with a conference component in order to work with networked media data. If, however, you have determined that you need to use a network component, this chapter describes how to do so and the interface supported by all network components. You should skip the material that discusses how to create a network component, though.

About Network Components

Network components, in tandem with transport components, allow QuickTime Conferencing components to engage in networked multimedia communications. Transport components, which are discussed in the "Transport Components" chapter elsewhere in this book, support specific networked-media protocols. Network components allow transport components, as well as other QuickTime Conferencing components and even standalone applications, to communicate over a given underlying communications network.

QuickTime Conferencing is designed to support a variety of existing and future network technologies and to support cross-platform implementations. Network components

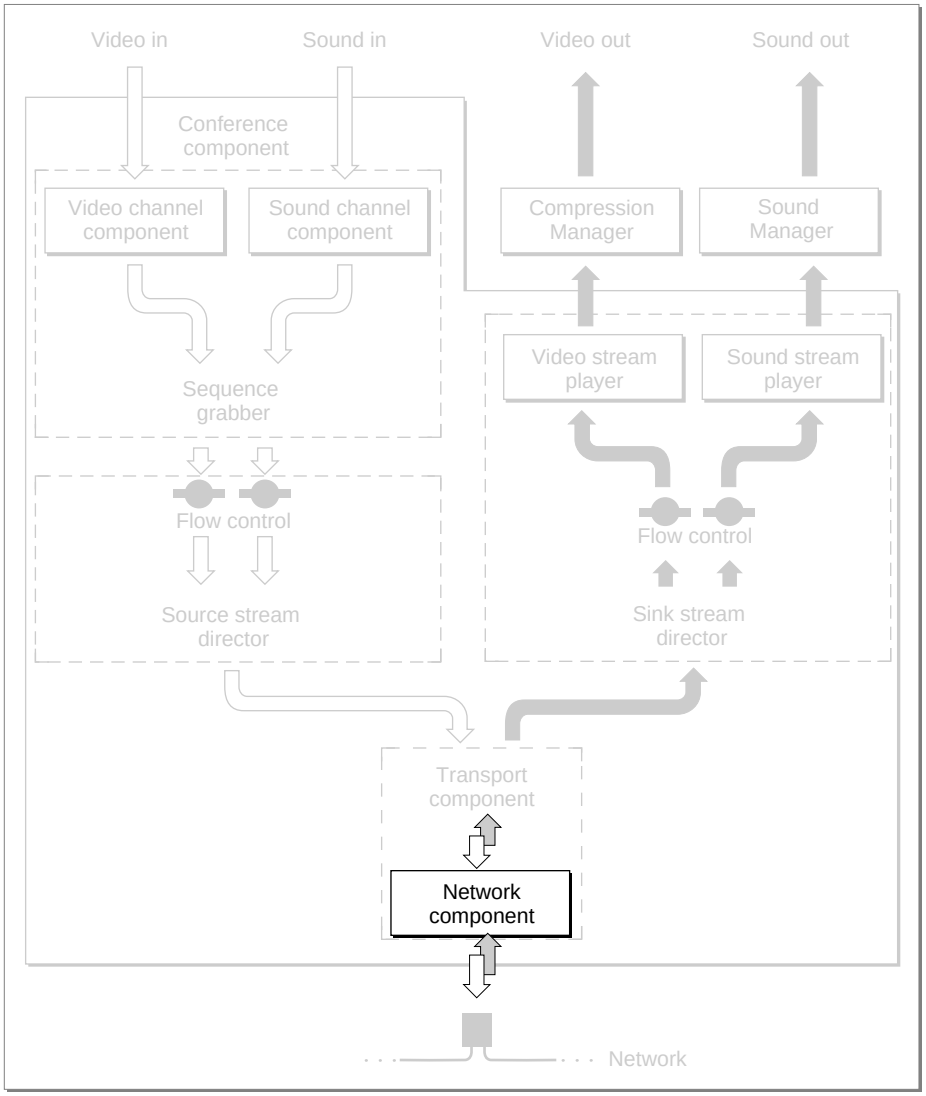
provide an important part of that flexibility, by isolating higher-level components and applications from network communications details. Figure 10-1 shows the relationships between network components and other QuickTime Conferencing components.

QuickTime Conferencing network components perform four major functions:

- **Media data management:** Network components allow you to exchange media data with remote network endpoints. Network components generally use a best effort, connectionless network interface for transferring media data. Media data is packetized for transmission.
- **Control data management:** Network components also allow you to send and receive control messages with remote systems. Network components generally use a reliable, connection-oriented network interface for transferring control data across the network. The control data path provides applications a byte stream with no forced message boundaries.
- **Name management:** Network components use the concept of a network-independent address (known as the MovieTalk address) to specify entities on the network for connection purposes. In addition, for those network protocols that provide some type of name service for managing network entities, these components provide functions that work with network names.
- **Component management:** As Component Manager components, network components support the standard component functions. In addition, they provide several management functions that allow you to control, monitor, and configure the component.

Media data endpoints and control data endpoints are separate from one another. Network components provide different functions that allow you to manipulate each different endpoint type. In addition, it's up to you to manage any implied relationship your application maintains between the different types of endpoints. For example, your application might use the control data path to exchange media endpoint addresses with remote systems in anticipation of transmitting media data.

Figure 10-1 The relationship between network components and other QuickTime Conferencing components



Network components encapsulate the protocol support necessary to work over a network. As such, network components are typically developed specifically for a network. For example, a given network component may provide AppleTalk or TCP/IP services to transport components or other programs. Note, however, that in some cases a single network component may support more than one protocol; an example of this would be an Open Transport network component that adheres to the XOpen Transport Interface (XTI) standard. You must configure such components for a specific protocol before using them. The network component functional interface supports such configuration.

Using Network Components

This section provides examples that show you how you can use a network component in your application. Each of the examples highlights a different aspect of the services provided by a network component and includes sample code illustrating how to use those services. Taken together, these samples constitute much of the logic of a working, though simple, networked media data application.

This section contains the following examples:

- "Opening and Managing Network Components" shows how to use the Component Manager to open an instance of a network component.
- "Establishing and Using Control Data Connections" provides examples of working with reliable data connections.
- "Sending and Receiving Media Data" presents examples of working with best effort data connections.

Note that, while these examples give you much of what you need to start using network components, there is much more that these component have to offer. For complete information about their functional interface, consult the "Network Components Reference" later in this chapter.

The samples in this section use a number of global variables. Listing 10-1 provides the declarations for those global variables.

Listing 10-1 Global variables

```
MTAddress               gLocalListenerAddress;
MTAddress               gLocalCallerAddress;
MTAddress               gLocalMediaAddress;

MTAddress               gRemoteListenerAddress;
MTAddress               gRemoteCallerAddress;
MTAddress               gRemoteMediaAddress;

ComponentInstance       gMemoryInstance;
long                    gMediaSize;
long                    gSendFrameNumber;
long                    gRecvFrameNumber;
long                    gRecvBytesLeft;
Byte                    *gRecvBuffer;
```

Note
The samples in this section show the calling sequences you might employ when using network components, but do not necessarily show the most efficient way to use each function. For example, these samples always call the network component synchronously. You, on the other hand, would most likely call the component asynchronously and supply a completion routine. ◆

Opening and Managing Network Components

As is the case with all Macintosh components, you must use Component Manager functions in order to gain access to a network component. If you are not familiar with the Component Manager, you should take a look at the appropriate chapter in *Inside Macintosh: More Macintosh Toolbox*.

To open an instance of a network component for your use, call the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. The following code fragment uses the `OpenDefaultComponent` function:

```
instance = OpenDefaultComponent(kMTNetworkType,
                                kMTAppleTalkSubType);
if (instance == nil) {
   /* process error */
}
```

This function requires you to identify the type and subtype of the component you desire. The `kMTNetworkType` constant identifies the component type—in this case, the network component.

Network components use their subtype value to identify the network protocol they support. The `kMTAppleTalkSubType` constant used here indicates that the network component you want to use supports the AppleTalk network protocol.

When you are done with the network component instance, use the `CloseComponent` function to close the instance:

```
result = CloseComponent(instance);
```

During your initialization processing, you might establish your network endpoints for establishing and receiving control connections. The code in Listing 10-2 sets up both local endpoints for an application.

Listing 10-2 Setting up local network endpoints

```
   err = MTNetworkBindListener(instance, &gLocalListenerAddress,
                               false, nil, nil);
   if (err != noErr) {
      /* process error */
```

```
}

err = MTNetworkBindCaller(instance, &gLocalCallerAddress,
                          false, nil, nil);
if (err != noErr) {
   /* process error */
}
```

In addition, you may want to prepare your application to receive media data. Listing 10-3 shows how you might define your media endpoint.

Listing 10-3 Preparing to receive media data

```
err = MTNetworkBindLocalMedia(instance, &gLocalMediaAddress,
                          false, nil, nil);
if (err != noErr) {
   /* process error */
}

err = MTNetworkSetReceiveMediaProc(instance, ReceiveMedia,
                          (long) globals, sizeof(MediaHeader));
if (err != noErr) {
   /* process error */
}
```

Finally, you may want to retrieve some basic information about the underlying network. The `MTNetworkGetInfo` function allows you to do so. The code fragment in Listing 10-4 shows how to determine the maximum amount of media data you can send in a single packet.

Listing 10-4 Retrieving information about the underlying network

```
err = MTNetworkGetInfo(instance, mtMediaSizeSelector,
                          &gMediaSize);
if (err != noErr) {
   /* process error */
}
```

Establishing and Using Control Data Connections

Network components allow you to establish control data connections as well as media channels. The samples in this section demonstrate how to establish and use control data connections to reliably exchange data with a remote endpoint.

First, consider the case where you want to be able to accept incoming connections. You must first register your endpoint's name on the network and then listen for an incoming connection. When you receive an incoming connection, you must accept the connection before you can use it to transfer data. Listing 10-5 shows one way to do this. This sample also formats and sends a control message containing the network address of this application's media endpoint. This logic shows how to use the network component's buffer records to hold a control message.

Listing 10-5 Receiving and using an incoming connection

```
/* set up and receive a network connection */

pascal ComponentResult ReceiveConnection (MTNetworkComponent
                              instance, Ptr name, Ptr type)
{
   ComponentResult   err;
   MTName            localName;
   MTBuffer          buffer;
   MTSegment         segment;
   MTBlock           block;

   /* register name; listen for and accept incoming connection */

   err = MTNetworkRegisterName(instance, name, type, &localName,
                              false, nil, nil);
   if (err != noErr) {
      /* process error */
   }

   err = MTNetworkListen(instance, &gRemoteCallerAddress, false,
                              nil, nil);
   if (err != noErr) {
      /* process error */
   }

   err = MTNetworkAccept(instance, &gRemoteCallerAddress,
                              instance, false, nil, nil);
   if (err != noErr) {
      /* process error */
   }

   /* set up buffer/segment/block records to send data */
```

```
   buffer.nextBuffer = nil;
   buffer.firstSegment = &segment;
   buffer.flags = 0;
   segment.nextSegment = nil;
   segment.thisBlock = &block;
   block.data = (Byte *) &gLocalMediaAddress;
   block.length = sizeof(gLocalMediaAddress);
   block.offset = 0;

   /* send local media data address on connection */
   err = MTNetworkSend(instance, &buffer, false, nil, nil);

   return (err);
}
```

Now consider the remote application that establishes the connection. Listing 10-6 shows logic to establish a connection and then receive the remote endpoint's media address. This routine first obtains the remote endpoint's network name, then establishes the connection. Once the connection is established, this routine expects the remote application to send its media address in a control message. Finally, the routine establishes a media channel with that remote media endpoint.

Listing 10-6 Establishing a connection and receiving a control message

```
/* set up and make a network connection */

pascal ComponentResult MakeConnection(MTNetworkComponent instance,
                              MTNamePtr remoteName)
{
   ComponentResult         err;
   MTRcvBlock              rcvBlock;

   /* look up and extract name and make outgoing connection */

   err = MTNetworkLookupName(instance, remoteName, false, nil,
                              nil);
   if (err != noErr) {
      /* process error */
   }

   err = MTNetworkExtractName(instance, &gRemoteListenerAddress);
   if (err != noErr) {
      /* process error */
   }
```

```
   err = MTNetworkConnect(instance, &gRemoteListenerAddress,
                              false, nil, nil);
   if (err != noErr) {
      /* process error */
   }

   /* set up block record to receive data */

   rcvBlock.buffer = (Byte *) &gRemoteMediaAddress;
   rcvBlock.size = sizeof(gRemoteMediaAddress);

   /* receive remote media data address on connection */

   err = MTNetworkReceive(instance, &rcvBlock, false, nil, nil);
   if (err != noErr) {
      /* process error */
   }

   /* bind remote media data address */

   err = MTNetworkBindRemoteMedia(instance,
                              &gRemoteMediaAddress, false, nil,
                              nil);

   return (err);
}
```

Sending and Receiving Media Data

Network components allow you to transmit media data from one network endpoint to another. Before you can do so, you must bind the endpoints into a media channel. The sample code in "Establishing and Using Control Data Connections" beginning on page 10-8 shows one way to do so. The routines listed there use the control connection to exchange media endpoint addresses in anticipation of exchanging media data.

The media data is structured as time-oriented streams. The transmitted data is composed of logical units called *chunks*, which correspond to video frames or collections of sound samples. When sending media data, you must properly format the buffer data structures so that they correspond to an appropriate chunk organization. This may involve creating a linked list of buffers in cases where you are sending more than one chunk of data in a single operation. You must also take into account the maximum packet size allowed by the underlying network.

The routine in Listing 10-7 shows an example of sending media data using a network component.

Listing 10-7 Sending media data

```
typedef struct {
   long        packetSize;
   long        packetNumber;
   long        frameSize;
   long        frameNumber;
} MediaHeader, *MediaHeaderPtr;

typedef struct {
   MTBuffer    buffer;
   MTSegment   hdrSegment;
   MTBlock     hdrBlock;
   MediaHeader header;
   MTSegment   dataSegment;
   MTBlock     dataBlock;
} MediaBuffer, *MediaBufferPtr;


/* send media data on a network connection */

pascal ComponentResult SendMedia(MTNetworkComponent instance,
                                 Byte *chunk, long chunkSize,
                                 MTChunkPriority chunkPriority)
{
   ComponentResult      err;
   short                payloadSize, dataSize, residualSize;
   short                numPackets, packet;
   MediaBufferPtr       mbp;
   MTBufferPtr          buffer;

   /* calculations for segmentation of media data */

   payloadSize = gMediaSize - sizeof(MediaHeader);
   numPackets = (chunkSize - 1) / payloadSize + 1;
   residualSize = chunkSize;
   gSendFrameNumber++;

   /* get memory for buffer records */

   mbp = (MediaBufferPtr)MTMemoryGetOneChunk(gMemoryInstance,
                                 numPackets * sizeof(MediaBuffer),
                                 false);
   buffer = &mbp->buffer;
```

```
/* set up buffer for each packet with two segments for packet
   header and packet payload */

for(packet = 0; packet < numPackets; packet++, mbp++) {

   /* set up packet header for this segment */

   dataSize = residualSize > payloadSize ?
                           payloadSize : residualSize;
   mbp->header.packetSize = sizeof(MediaHeader) + dataSize;
   mbp->header.packetNumber = packet;
   mbp->header.frameSize = chunkSize;
   mbp->header.frameNumber = gSendFrameNumber;

   /* set up buffer/segment/block records */

   mbp->buffer.flags = mtBufferPendingMask;
   mbp->buffer.priority = chunkPriority;
   mbp->buffer.firstSegment = &mbp->hdrSegment;
   mbp->hdrSegment.thisBlock = &mbp->hdrBlock;
   mbp->hdrSegment.nextSegment = &mbp->dataSegment;
   mbp->hdrBlock.data = (Ptr) &mbp->header;
   mbp->hdrBlock.length = sizeof(MediaHeader);
   mbp->hdrBlock.offset = 0;
   mbp->dataSegment.thisBlock = &mbp->dataBlock;
   mbp->dataSegment.nextSegment = 0;
   mbp->dataBlock.data = chunk + packet * payloadSize;
   mbp->dataBlock.length = dataSize;
   mbp->dataBlock.offset = 0;

   /* link to next buffer if not last buffer */

   if(residualSize -= dataSize)
      mbp->buffer.nextBuffer = (MTBufferPtr) (mbp + 1);
   else
      mbp->buffer.nextBuffer = nil;
}

/* send linked list of buffers  */
err = MTNetworkSendMedia(instance, buffer, false, nil, nil);

/* release memory for buffer records */
```

```
   MTMemoryPutOneChunk(gMemoryInstance, (Ptr) buffer);

   return (err);
}
```

At the remote endpoint, receiving the media data involves providing receive buffers to the network component and correctly dequeuing the incoming packets. Listing 10-8 contains a routine that receives media data and delivers it to a waiting application.

Listing 10-8 Receiving media data

```
/* receive media data on a network connection */

Byte *ReceiveMedia(MTRcvBlock *rcvBlockPtr)
{
   MediaHeaderPtr       mhp;
   Byte                 *buffer;
   GlobalsPtr           gp;

   gp = (GlobalsPtr)rcvBlockPtr->refcon;
   mhp = (MediaHeaderPtr) rcvBlockPtr->header;

   /* process received data only after read completes */

   if (rcvBlockPtr->flags == kMTRcvBlockCompleteMask) {

      if ((gp->gRecvBytesLeft -= mhp->packetSize -
                              sizeof(MediaHeader)) <= 0)
         DeliverMedia(gp->gRecvBuffer);

      return(nil);
   }

   /* process header before read completes */

   if(mhp->frameNumber != gp->gRecvFrameNumber) {
      DeliverMedia(gp->gRecvBuffer);
      gp->gRecvFrameNumber = mhp->frameNumber;
      gp->gRecvBytesLeft = mhp->frameSize;
      gp->gRecvBuffer = MTMemoryGetOneChunk(gp->gMemoryInstance,
                              mhp->frameSize, false);
```

```
      if(!gp->gRecvBuffer)
         return(nil);
   }

   /* set flag for callback after read completes */

   rcvBlockPtr->flags = kMTRcvBlockCompleteMask;

   /* set buffer offset to read into for resegmenting media
      chunk */

   buffer = gp->gRecvBuffer + mhp->packetNumber *
            (gp->gMediaSize - sizeof(MediaHeader));

   return (buffer);
}
```

Creating Network Components

By creating a new network component, you can open Apple's QuickTime Conferencing technology to new network protocols. This benefits users of those networks by allowing them to participate in media-rich conferences using standard network protocols. This also allows those users to interact with users on other types of networks. This section discusses the topics you should consider if you plan to create your own network component. If you plan to use, but not create, network components, you need not read this section.

If you are unfamiliar with creating Macintosh components in general, refer to the Component Manager chapter in *Inside Macintosh: More Macintosh Toolbox* for more information.

Functional Interface

Apple has defined a functional interface for network components. Your network component must support that interface. For a complete description of the interface your component must support, see "Functions" beginning on page 10-29.

You can use the following constants to identify the request codes your component receives from applications that use its services. Each of these constants corresponds to the similarly named network component function described elsewhere in this chapter.

```
enum {
   kMTNetworkGetInfoSelect =              1,
   kMTNetworkOptionsSelect =              2,
```

```
   kMTNetworkBindLocalMediaSelect =       3,
   kMTNetworkUnbindLocalMediaSelect =     4,
   kMTNetworkBindRemoteMediaSelect =      5,
   kMTNetworkUnbindRemoteMediaSelect =    6,
   kMTNetworkSendMediaSelect =            7,
   kMTNetworkSetReceiveMediaProcSelect =  8,
   kMTNetworkBindListenerSelect =         9,
   kMTNetworkUnbindListenerSelect =       10,
   kMTNetworkBindCallerSelect =           11,
   kMTNetworkUnbindCallerSelect =         12,
   kMTNetworkListenSelect =               13,
   kMTNetworkAcceptSelect =               14,
   kMTNetworkConnectSelect =              15,
   kMTNetworkDisconnectSelect =           16,
   kMTNetworkSendSelect =                 17,
   kMTNetworkReceiveSelect =              18,
   kMTNetworkGetNextEventSelect =         19,
   kMTNetworkSetNotifyProcSelect =        20,
   kMTNetworkRegisterNameSelect =         21,
   kMTNetworkRemoveNameSelect =           22,
   kMTNetworkLookupNameSelect =           23,
   kMTNetworkExtractNameSelect =          24
};
```

Component Type and Subtype Values

Apple has defined a type value for network components. All network components have a component type of `'ntwk'`. You can use the following constant to specify this component type value:

```
enum {
   kMTNetworkType          = 'ntwk'
};
```

Network components use the subtype value to identify the type of network they support. Apple has defined the following subtype values. Third parties may define other network component subtypes in order to support new types of networks (Apple Computer reserves values consisting of all lower-case letters).

```
enum {
   kMTAppleTalkSubType     = 'atlk',
   kMTTCPIPSubType         = 'tcpi',
   kMTATMulticastSubType   = 'atmc',
```

```
   kMTIPMulticastSubType   = 'ipmc',
   kMTISDNSubType          = 'isdn'
};
```

Required and Optional Functions

All network components must support the following functions:

- `MTNetworkGetInfo`
- `MTNetworkBindListener`
- `MTNetworkUnbindListener`
- `MTNetworkListen`
- `MTNetworkAccept`
- `MTNetworkDisconnect`
- `MTNetworkBindCaller`
- `MTNetworkUnbindCaller`
- `MTNetworkConnect`
- `MTNetworkSend`
- `MTNetworkReceive`
- `MTNetworkBindLocalMedia`
- `MTNetworkUnbindLocalMedia`
- `MTNetworkBindRemoteMedia`
- `MTNetworkUnbindRemoteMedia`
- `MTNetworkSendMedia`
- `MTNetworkSetReceiveMediaProc`
- `MTNetworkRegisterName`
- `MTNetworkRemoveName`
- `MTNetworkLookupName`
- `MTNetworkExtractName`

Your network component must support one of the following two functions (preferably the `MTNetworkSetNotifyProc` function):

- `MTNetworkGetNextEvent`
- `MTNetworkSetNotifyProc`

You may choose to support the following optional function:

- `MTNetworkOptions`

Interrupt-Time Processing

Applications must be able to call most network component functions at interrupt time. This restricts how you implement your component. The following functions are guaranteed not to be called at interrupt time:

- `MTNetworkExtractName`
- `NTNetworkOptions`
- `MTNetworkSetNotifyProc`
- `MTNetworkSetReceiveMediaProc`
- your component's `Open` and `Close` functions

Other than these exceptions, your component must be interrupt safe. This means that you may not allocate memory except in these functions.

Your component's clients typically allocate memory for send and receive media data buffers, except for media data headers.

The QuickTime Conferencing memory component supplies memory management functions that you can use at interrupt time. Please refer to the "Utility Components" chapter of this book for more information about these functions.

Asynchronous Functions

Many network component functions must execute asynchronously. Apple requires that all network component functions that access the underlying network be capable of asynchronous execution. At a minimum, any function that has an `async` parameter must be able to execute asynchronously. Conversely, all functions are required to execute synchronously when the client requests synchronous execution.

It is up to the client application to specify whether to execute asynchronously or synchronously. Client applications may supply completion routines and reference constants when they request asynchronous execution. Applications are allowed to supply a `nil` pointer for the completion routine if they do not desire notification of completion. Even if a network component function is inherently synchronous, you are still required to call the supplied completion routine if the client application requests asychronous execution.

Only the `MTNetworkReceive` function provides a method for returning more than an error code to the client's completion procedure. Your other functions must return data in the locations specified by the client when it calls your component.

Your `MTNetworkSendMedia` and `MTNetworkSend` functions are required to support multiple outstanding asynchronous calls. All other functions must return an error if an asynchronous call is already in progress when the client calls asynchronously.

Network Component States

While Apple does not require a specfic implementation of network components, you should expect to have to do a certain amount of state management in your component. It is up to you to decide whether you want to implement your component as a state machine. For your guidance, this section discusses the states that Apple's network components manage and the network and application events that can cause state transitions.

The states fall into three major categories:

- **Name mapping:** Only a single name may be registered for the network component at a time. A new name may not be registered until an existing registered name is removed.
- **Media data:** Media data is generally sent over a connectionless, best effort protocol. The network component states for media data are a superset of the Open Transport connectionless states.
- **Control data:** Control data is generally sent over a connection-oriented, reliable protocol. Apple's network components maintain both an active and a passive connection endpoint. The network component states for control data are based on the XTI connection-oriented states of these two endpoints.

The following three sections discuss each of these categories in more detail.

Name-Mapping States

Apple's network components manage name-mapping states in response to application requests to register, remove, look up, or extract network endpoint names. The components use the following two Boolean state variables to manage these states:

- `nameRegistered`
- `nameLookupDone`

Table 10-1 shows how application requests cause name-mapping state transitions.

Table 10-1 Name-mapping state transitions

State variable	Application request	Initial value	Result value
`nameRegistered`	`MTNetworkRegisterName`	`false`	`true`
`nameRegistered`	`MTNetworkRemoveName`	`true`	`false`
`nameLookupDone`	`MTNetworkLookupName`	`false`	`true`
`nameLookupDone`	`MTNetworkExtractName`	`true`	`false`

Media Data States

Apple's network components maintain four states that relate to media data transfer. These states are a superset of the corresponding Open Transport connectionless data

transfer states. Table 10-2 lists these states, mapping them to their XTI equivalents, and noting any valid data transfers that can occur in each state.

Table 10-2 Media data states and corresponding XTI states

Network component state	XTI state	Data transfer
`neitherBound`	`T_UNBND`	None
`remoteBound`	`T_UNBND`	None
`localBound`	`T_IDLE`	Receive media data
`bothBound`	`T_IDLE`	Send and receive media data

Transitions between these states are triggered by application requests. Table 10-3 lists each state transition, showing the application request that causes the transition to occur.

Table 10-3 Media data state transitions

Initial state	Application request	Result state
`neitherBound`	`MTNetworkBindLocalMedia`	`localBound`
`neitherBound`	`MTNetworkBindRemoteMedia`	`remoteBound`
`remoteBound`	`MTNetworkBindLocalMedia`	`bothBound`
`remoteBound`	`MTNetworkUnbindRemoteMedia`	`neitherBound`
`localBound`	`MTNetworkBindRemoteMedia`	`bothBound`
`localBound`	`MTNetworkUnbindLocalMedia`	`neitherBound`
`localBound`	`MTNetworkSetReceiveMediaProc`	`localBound`
`bothBound`	`MTNetworkSetReceiveMediaProc`	`bothBound`
`bothBound`	`MTNetworkSendMedia`	`bothBound`
`bothBound`	`MTNetworkUnbindLocalMedia`	`remoteBound`
`bothBound`	`MTNetworkUnbindRemoteMedia`	`localBound`

Control Data States

The network component keeps states for control data transfer for both an active and a passive endpoint. The active endpoint corresponds to the endpoint that initiates a connection; the passive one receives that connection attempt. Only one of these endpoints is used for data transfer at a time.

Table 10-4 lists these states, mapping them to their XTI equivalents.

Table 10-4 Control data states and corresponding XTI states

Network component state	XTI passive state	XTI active state
`neitherBound`	`T_UNBND`	`T_UNBND`
`listenBound`	`T_IDLE`	`T_UNBND`
`callerBound`	`T_UNBND`	`T_IDLE`
`bothBound`	`T_IDLE`	`T_IDLE`
`listenPend`	`T_IDLE`	`T_IDLE`
`listenDone`	`T_INCON`	`T_IDLE`
`acceptPend`	`T_INCON`	`T_IDLE`
`dataTransfer` (passive endpoint)	`T_DATAXFER`	`T_IDLE`
`connectPend`	`T_IDLE`	`T_OUTCON`
`dataTransfer` (active endpoint)	`T_IDLE`	`T_DATAXFER`

Both application requests and network events trigger state transitions. Table 10-5 lists the possible state transitions and the requests or events that cause them to occur.

Table 10-5 Control data state transitions

Initial state	Application request or network event	Result state
neitherBound	MTNetworkBindListener	listenBound
neitherBound	MTNetworkBindCaller	callerBound
listenBound	MTNetworkBindCaller	bothBound
listenBound	MTNetworkUnbindListener	neitherBound
callerBound	MTNetworkBindListener	bothBound
callerBound	MTNetworkUnbindCaller	neitherBound
bothBound	MTNetworkUnbindListener	callerBound
bothBound	MTNetworkUnbindCaller	listenBound
bothBound	MTNetworkConnect	connectPend
bothBound	MTNetworkListen	listenPend
bothBound	MTNetworkAccept*	acceptPend
listenPend	Listen completes	listenDone
listenPend	MTNetworkDisconnect†	bothBound
listenDone	MTNetworkAccept	acceptPend
listenDone	MTNetworkDisconnect	bothBound
acceptPend	Accept completes	dataTransfer
acceptPend	MTNetworkDisconnect	bothBound
connectPend	Connect completes	dataTransfer
connectPend	MTNetworkDisconnect	bothBound
dataTransfer	MTNetworkSend	dataTransfer
dataTransfer	MTNetworkReceive	dataTransfer
dataTransfer	MTNetworkDisconnect	bothBound

* This is an accept on behalf of another component instance that listened for the incoming connection.

† The disconnect request may originate at either endpoint; this applies to all disconnect request state transitions.

Network Components Reference

This section provides detailed information about the functions supported by network components and the constants and data structures you use when working with those functions.

Constants

This section discusses important network component constants.

Network Component Subtypes

The following constants identify a variety of network component subtypes. Network components use the subtype value to identify the type of network they support. Apple has defined the following subtypes. Third parties may define other network component subtypes in order to support new types of networks.

```
enum {
   kMTAppleTalkSubType     = 'atlk',
   kMTTCPIPSubType         = 'tcpi',
   kMTATMulticastSubType   = 'atmc',
   kMTIPMulticastSubType   = 'ipmc',
   kMTISDNSubType          = 'isdn'
};
```

Constant descriptions

`kMTAppleTalkSubType`
: Describes a network component that provides an AppleTalk network interface.

`kMTTCPIPSubType`
: Describes a network component that provides a TCP/IP network interface.

`kMTATMulticastSubType`
: Describes a network component that provides an AppleTalk multicast network interface.

`kMTIPMulticastSubType`
: Describes a network component that provides an IP multicast network interface.

`kMTISDNSubType`
: Describes a network component that provides an ISDN network interface.

Data Structures

This section discusses several key data structures that are used by network components and by applications that use network components.

Receive Block

Network components use the receive block record to describe incoming network data, whether it contains control messages or media data. For more information about how you use this data structure to receive media data or messages, see "Application-Defined Functions" beginning on page 10-65. Network components also use this data structure to request a buffer from your application or component.

```
struct MTRcvBlock {
   UInt32               refcon;
   UInt32               size;
   MTRcvBlockFlags      flags;
   ComponentResult      error;
   Byte                 *header;
   Byte                 *buffer;
   MTChannel            channel;
};
typedef struct MTRcvBlock MTRcvBlock, *MTRcvBlockPtr;
```

Field descriptions

`refcon` A reference constant value for your use. The network component sets this field to the reference constant value you specify when you call the `MTNetworkSetReceiveMediaProc` function to install a receive media data routine or when you call the `MTNetworkReceive` function asynchronously.

`size` The amount of received data, in bytes. This is not necessarily the same as the size of the buffer.

If the network component is requesting a buffer from your media data receive procedure (in this case, the `kMTRcvBlockGetBufferMask` flag is set to 1), this field specifies the desired buffer size.

`flags` Control flags. The following flag values are defined (note that more than one flag may be set to 1):

`kMTRcvBlockStartMask`
Indicates that this buffer contains the header for an incoming packet.

`kMTRcvBlockCompleteMask`
Indicates that all of the media data for an incoming packet has been received.

Your application can use this flag to cause the network component to call your notify routine when the chunk has been received completely. Set this flag to 1 when you return the receive block to the network component in order to receive this notification.

`kMTRcvBlockGetBufferMask`
Indicates that the network component is requesting a new buffer from your media data receive procedure. In this case, the `size` field contains the desired buffer size. The `refcon` field is set to the value you specified when you defined your media data receive procedure. Your function returns a pointer to the buffer. This flag is valid only in unsegmented mode.

`kMTRcvBlockInterruptMask`
Indicates that it is interrupt time. Your application or component must be careful not to move memory while servicing the receive block. For example, if the network component is requesting a buffer during interrupt time, your program can return only preallocated buffers. This flag is valid only in unsegmented mode.

`error` Indicates any network-related error. Any network component result code is valid.

`header` Pointer to the header associated with the incoming data. If there is no header information, the network component sets this field to `nil`.

`buffer` Pointer to the incoming data.

`channel` Channel on which to send the data. This field specifies the channel on which the network component is to send the media data. This field is valid only when the network supports bandwidth reservation. You obtain the channel identifier when you reserve network bandwidth using the `MTNetworkOptions` function (discussed on page 10-32).

Data Buffer Structures

In order to minimize the copying of outbound media and message data, network components use a group of data structures to define buffers of outgoing data. This section discusses those data structures. Network components use a different structure, the receive block record, to receive incoming data; that record is discussed on page 10-24.

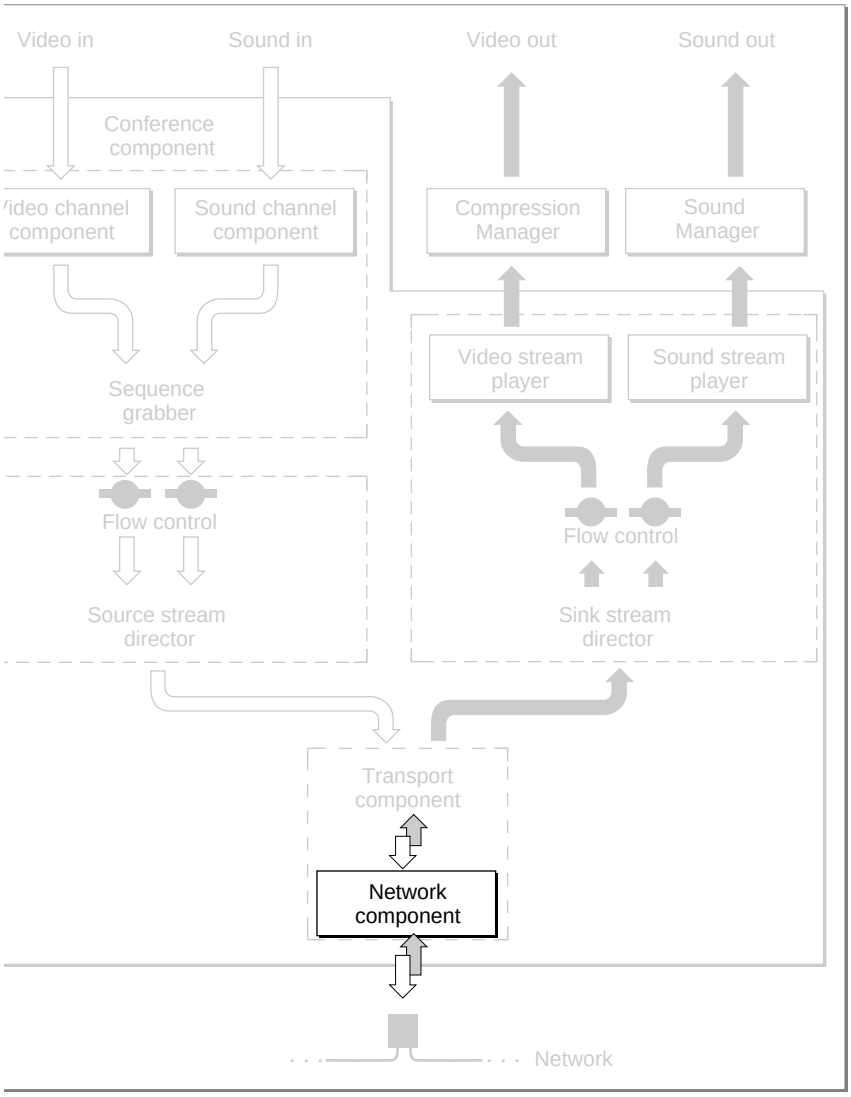
Figure 10-2 shows the relationships between these records and how they relate to the actual buffers of media data. This figure depicts an example of how Apple's network component uses these records to describe media data. At the highest level, there is one buffer record for each media data packet or control message buffer. When they describe media data, buffer records may be chained together, allowing network components to support multiply buffered operations. The `MTBuffer` structure defines the buffer record.

Each buffer record, in turn, refers to one or more segment records. Each segment record designates a piece of a media data packet or control message buffer. As with buffer

records, segment records may be chained together, allowing the media data packet or control message buffer to be dealt with in pieces. The `MTSegment` structure defines the segment record.

Finally, each segment record refers to a single block record. The fields in a block record designate the piece of a media data packet or control message buffer that corresponds to a given segment. Note that those pieces may be in any order in the buffer, as well, based on the data references in the block records. The `MTBlock` structure defines the block records, and describes an arbitrary block of memory and the data within it. The data of interest need not start at the beginning of that memory block.

Figure 10-2 Relationships between data buffer structures



```
struct MTBuffer {
   struct MTBuffer      *nextBuffer;
   MTSegmentPtr         firstSegment;
   MTBufferFlags        flags;
   ComponentResult      error;
   MTChunkPriority      priority;
   MTChannel            channel;
};
typedef struct MTBuffer MTBuffer, *MTBufferPtr;
```

Field descriptions

`nextBuffer` Pointer to the next `MTBuffer` structure in the chain. If this is the last `MTBuffer` structure in the chain, set this field to `nil`.

`firstSegment` Pointer to this buffer's first `MTSegment` structure.

`flags` Control flags. The following value is defined:

`mtBufferPendingMask`
In-use indicator. When you supply a buffer to the network component, you must set this flag to 1. When the network component is done with the buffer, it sets the flag to 0. You can they reuse the buffer. This allows you to reuse a buffer more quickly than you could if you waited for the network component to call your completion routine—the network component calls your completion routine only after it is done with all of the buffers for an operation. This flag is valid only when you call the network component asynchronously.

`error` Error code.

While processing an `MTBuffer` chain, a network component may encounter a network transmission error. When this happens, the network component gives up on the send operation, clears the pending mask (the `mtBufferPendingMask` flag in the `flags` field) in all of the remaining `MTBuffer` structures in the chain, and sets an appropriate error code in the `MTBuffer` structure that caused the error. The component also returns this error code to your completion routine, if you called the function asynchronously. The component sets the `error` field in the remaining `MTBuffer` structures to `mtOperationAbandonedErr`.

`priority` Media data priority value. This value indicates the relative priority of the data to be sent. The following values are valid (these are QuickTime Conferencing priority values; network components must translate these values to appropriate values for the underlying network protocols):

`mtChunkPrioritySound`
The priority level for sound data. It is the highest priority among the defined priority levels.

`mtChunkPriorityMedium`
An intermediate priority level. It falls between the priorities for sound and video data.

`mtChunkPriorityVideo`
The priority for delta frames of video data. It is the lowest priority among the defined priority levels. Delta frames are distinguished from key frames in that a delta frame contains only change information from a previous frame. A key frame contains complete information for the video frame.

`mtChunkPriorityKeyFrame`
A constant used to generate a priority for chunks containing key frames. You can use it with any media type to generate a priority that applies to a chunk you use to resynchronize.You generate the key frame priority value by performing a bitwise OR operation between `mtChunkPriorityKeyFrame` and another priority constant, such as `mtChunkPriorityVideo`.

In the future, Apple and third parties may define additional priorities for new media types.

`channel` Channel on which to send the data. This field specifies the channel on which the network component is to send the media data. This field is valid only when the network supports bandwidth reservation. You obtain the channel identifier when you reserve network bandwidth using the `MTNetworkOptions` function (discussed on page 10-32).

A single `MTBuffer` structure may refer to one or more `MTSegment` structures.

```
struct MTSegment {
   struct MTSegment     *nextSegment;
   MTBlockPtr           thisBlock;
};
typedef struct MTSegment MTSegment, *MTSegmentPtr;
```

Field descriptions

`nextSegment` Pointer to this buffer's next segment. If this is the last segment in the buffer, set this field to `nil`.

`thisBlock` Pointer to the `MTBlock` structure corresponding to this segment.

Each `MTSegment` structure corresponds to a single `MTBlock` structure.

```
struct MTBlock {
   Byte                 *data;
   UInt16               maxLength;
   UInt16               length;
   UInt16               offset;
```

```
   UInt16              count;
};
typedef struct MTBlock MTBlock, *MTBlockPtr;
```

Field descriptions

`data`	Pointer to the block. This pointer identifies the start of the memory block that holds the media or control data. The data itself starts at a location specified by the `offset` field.
`maxLength`	Size of the block. This value specifies the size of the entire block, including the media data as well as any other data in the block.
`length`	The length of the media data, in bytes.
`offset`	The offset from the block's base address to the media data. The network component adds this value to the address you specify in the `data` field in order to obtain the address of the media data. If the media data starts at the beginning of the block, set this field to 0. You can use the value of this field to reserve some space in front of your media data. You can later use that space to prepend a media data header without having to copy the media data itself to make room.
`count`	Use count. This field indicates how many segments are pointing to this `MTBlock` structure.

Functions

This section discusses the functions supported by network components. This section has been divided into the following topics:

- "Managing Network Components" discusses functions that allow you to control and configure network components.
- "Managing Connections" describes the functions that allow you to receive incoming network connections and establish outgoing connections.
- "Exchanging Control Messages" discusses the functions that allow you to send and receive data over reliable network connections.
- "Setting Up Media Channels" tells you how to set up media data connections.
- "Exchanging Media Data" describes the functions that let you send and receive media data.
- "Managing Network Names" discusses the functions that provide network name services.
- "Application-Defined Functions" describes the various callback routines your application can provide to a network component.

Managing Network Components

This section discusses functions that allow you to control, configure, and monitor a network component, including negotiating for network bandwidth.

The `MTNetworkGetInfo` function retrieves information describing the capabilities of the network component and the underlying network. The `MTNetworkOptions` function allows you to read and write a network component's option settings. These options may include bandwidth reservation requests, where appropriate. The `MTNetworkGetNextEvent` and `MTNetworkSetNotifyProc` functions allow you to receive notification of certain network events.

MTNetworkGetInfo

The `MTNetworkGetInfo` function provides information about a network component and the underlying communications network.

```
pascal ComponentResult MTNetworkGetInfo (MTNetworkComponent nc,
                        MTInfoSelector whichInfo, long *result);
```

`nc` The network component you are using.

`whichInfo` The type of information to retrieve. The following values are valid:

`mtFlowBandwidthSelector`
The network component tells you the maximum throughput of your connection's media stream. The network component returns an `MTBitsPerSecond` value that specifies the maximum throughput, in bits per second.

`mtFlowControlSelector`
The network component tells you whether it is requesting flow control on your call's media stream. The network component returns a Boolean value. The returned value is set to `true` if the network component wants flow control on the stream. Flow control components use this value to determine when to enable and disable flow control.

`mtIsBidirectionalSelector`
The network component tells you whether the media connection is bidirectional. The network component returns a Boolean value. If the connection is bidirectional, the network component returns a value of `true`.

`mtIsMulticastSelector`
The network component tells you whether the media connection is multicast. The network component returns a Boolean value. If the connection is multicast, the network component returns a value of `true`.

`mtIsUnsegmentedSelector`
: The network component tells you whether it expects to send and receive media data in unsegmented mode. The network component returns a Boolean value. If your connection is running in unsegmented mode, the network component returns a value of `true`. For more information about unsegmented mode, see the discussion of the media data receive routine in "Application-Defined Functions" beginning on page 10-65.

`mtHasReservationSelector`
: The network component tells you whether the underlying network supports bandwidth reservation. The network component returns a Boolean value. If the network supports bandwidth reservation, the network component returns a value of `true`. In this case, you can use the `MTNetworkOptions` function to reserve and release network bandwidth.

`mtControlSizeSelector`
: The network component tells you the maximum message buffer size you can use on your connection's control channel. The network component returns an `MTMessageSize` value. This returned value specifies the maximum control message buffer size, in bytes.

`mtMediaSizeSelector`
: The network component tells you the maximum packet size you can use on your connection's media channel. The network component returns an `MTMessageSize` value. This returned value specifies the maximum media packet size, in bytes. This size value includes both header and payload information.

`mtMulticastTypeSelector`
: The network component tells you the component subtype value assigned to its multicast twin. A given network component supports either multicast or point-to-point traffic, but not both. If you are working with a point-to-point component and need to use a multicast version of the same protocol, you can use this option to locate an appropriate multicast component.

  The network component returns an `OSType` value. This value contains the component subtype assigned to network components that support a multicast version of the point-to-point networked-media protocol that is used by the current network component.

`mtUnicastTypeSelector`
: The network component tells you the component subtype value assigned to its point-to-point twin. As with the `mtMulticastTypeSelector` type, the network component returns an `OSType` value. This value contains the component subtype assigned to network components

that support a point-to-point version of the multicast networked-media protocol that is used by the current network component.

`result` The address of the location to receive the resulting information. The network component returns different types of data, based on your request.

DESCRIPTION

You call the `MTNetworkGetInfo` function to get information about the capabilities and characteristics of the network component you are using to support a connection.

The `MTNetworkGetInfo` function takes a parameter that allows you to specify the type of information you want to retrieve. In response, the network component returns the information to a location you specify.

This function corresponds to the `MTTransportGetInfo` function supported by transport components. In many cases, a transport component responds to that function by calling the network component it is using for a given connection and retrieving the appropriate information.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid selector value
`mtConfigurationErr`	–7891	Unable to retrieve information

SEE ALSO

You can negotiate network bandwidth reservations by calling the `MTNetworkOptions` function, described next.

MTNetworkOptions

The `MTNetworkOptions` function allows you to retrieve detailed information about the network component and to negotiate some network parameters. You interact with the network component by exchanging options block and reservation block records; these records are described in this section.

```
pascal ComponentResult MTNetworkOptions (MTNetworkComponent nc,
                        MTOptionType optionType,
                        const MTOptions *setOpts,
                        MTOptions *getOpts);
```

`nc` The network component you are using.

`optionType` Request type. This parameter specifies the type of request you are making. The following values are valid:

`kMTOpenTransportType`

Indicates that this is an Open Transport options request. This option allows you to use XTI-compliant options requests, in cases where the network component supports Open Transport requests. In such cases, the network component passes your options block record through to the underlying transport and passes the resulting response block back to your application.

You supply the Open Transport request in the options block you specify with the `setOpts` parameter. The `networkType` field must identify the type of network in use. Use the `flags` field to specify the appropriate XTI flags for the request. The network component ignores the `maxlen` field. The `length` field indicates the length of the Open Transport options request record pointed to by the `buffer` field.

The network component returns the results of your request in the location you identify with the `getOpts` parameter. The component does not use the `networkType` field or the `flags` field. You set the `maxlen` field to indicate the maximum reply message you can accept at the location referred to by the `buffer` field. The component sets the `length` field to indicate the amount of data it returns to the location you specify with the `buffer` field.

`kMTRequestReservationType`

Reserve network bandwidth. This option allows you to reserve some amount of network bandwidth for the current connection. You specify the details of your request using the `setOpts` parameter. Set the `networkType` field to identify the type of network in use. The network component ignores the `flags` and `maxlen` fields. The `length` field indicates the size of the reservation block pointed to by the `buffer` field. This option can support a wide range of reservation blocks (also known as flow specifications). Apple's reservation block is discussed later in this section.

The network component reports the results of your request in the location you specify with the `getOpts` parameter. The component ignores the `networkType` field and the `flags` field. You set the `maxlen` field to indicate the maximum reply message you can accept at the location referred to by the `buffer` field. The component sets the `length` field to indicate the amount of data it returns to the location you specify with the `buffer` field.

This option is valid only when you are using a network that supports bandwidth reservation. You can determine whether a network supports bandwidth reservation by calling the `MTNetworkGetInfo` function.

Components that support bandwidth reservation must release any reserved bandwidth when an application closes its component instance.

`kMTReleaseReservationType`
Release reserved bandwidth. This option allows you to release some network bandwidth that you reserved previously (using the `kMTRequestReservationType` option).

You specify the details of your request using the `setOpts` parameter. Set the `networkType` field to identify the type of network in use. The network component ignores the `flags` and `maxlen` fields. The `length` field indicates the size of the reservation block you specify using the `buffer` parameter. This option can support a wide range of reservation blocks (also known as flow specifications). Apple's reservation block is discussed later in this section.

The network component reports the results of your request in the location you specify with the `getOpts` parameter. The component ignores the `networkType` field and the `flags` field. You set the `maxlen` field to indicate the maximum reply message you can accept at the location referred to by the `buffer` field. The component sets the `length` field to indicate the amount of data it returns to the location you specify with the `buffer` field.

`setOpts` Options to set. This parameter contains the address of an options block record. You set the fields of this record appropriately for your request.

`getOpts` Retrieved options. This parameter contains the address of a location to receive an options block record from the network component. The component reports the results of your request in this location. It is your application's responsibility to release this memory when you are done with it.

DESCRIPTION

The `MTNetworkOptions` function allows you to set and retrieve network component options and to negotiate network bandwidth reservations. You can set and retrieve options information at the same time. This function also allows you to exchange Open Transport compliant options messages with network components that implement the XTI standard.

You interact with the network component by exchanging options block records. The `setOpts` parameter allows you to pass an options block record to the network component. You set fields in that record appropriately for the given request. The `getOpts` parameter allows you to specify where the component is to return its response. You set some of the fields in that record in order to apply constraints to the reply.

The options block record is defined as follows:

```
struct MTOptions {
   OSType               networkType;
   MTOptionFlags        flags;
   UInt16               maxlen;
   UInt16               length;
   Byte                 *buffer;
};
typedef struct MTOptions MTOptions, *MTOptionsPtr;
```

Field descriptions

`networkType` Specifies the type of network. You must set this field to correspond to the component subtype value of the network component you are using.

`flags` Control flags. This field is used only when you are exchanging Open Transport options messages. In these cases, you use this field to specify the XTI flags that are appropriate to the operation.

`maxlen` Maximum acceptable reply. This field is valid only in options blocks supplied using the `getOpts` parameter. It indicates the size of the reply buffer you are supplying at the location you specify with the `buffer` field.

`length` Request or reply length. In request records (those provided using the `setOpts` parameter), this field indicates the size of the request block you are supplying at the location specified by the `buffer` field. In response records (those supplied using the `getOpts` parameter), this field indicates the actual amount of data returned by the component.

`buffer` Message buffer. This field points to a request or reply record that you supply to the component. For Open Transport requests, this field specifies the XTI options record. For bandwidth reservation requests, this field points to a reservation block record.

You can use this function to negotiate network bandwidth when you are using a network that supports bandwidth reservation. You can determine whether a network supports bandwidth reservation by calling the `MTNetworkGetInfo` function. When you negotiate for network bandwidth, you do so using flow specifications. Network components encapsulate flow specifications in records called reservation blocks.

The reservation block record is defined as follows (Table 10-6 summarizes how the fields are used):

```
struct MTReservation {
   MTReserveType        flowSpecType;
   MTBitsPerSecond      bandwidth;
   MTReservationID      identity;
   MTChannel            channel;
};
```

Field descriptions

`flowSpecType` Specifies the type of flow specification. The value of this field allows the network component to discriminate between different flow specifications. This must be the first field in any flow specification used with a network component. You must set this field to `kMTReserveType`.

`bandwidth` Bandwidth, in bits per second. For reservation requests, you use this field to indicate the amount of bandwidth you desire (in the options block specified by the `setOpts` parameter). In response, the network component returns a value indicating the amount of bandwidth remaining after your request (in the options block specified by the `getOpts` parameter). This returned value is valid even after failed reservation requests.

For release requests, the network component ignores this field in the input options block. In response, the component returns a value indicating the amount of bandwidth remaining after your request (in the options block specified by the `getOpts` parameter).

`identity` Identifies a reservation. The component assigns a unique identifier to each reservation request you make. This allows you to identify the reservation when you release the bandwidth.

In response to your reservation request, the network component returns this value in the options block specified by the `getOpts` parameter. You must supply this value when you release that bandwidth.

In all other cases, this field is ignored.

`channel` Network-specific media channel identifier. This value identifies a network-specific media channel that is carrying your bandwidth. This is not the same as a MovieTalk channel; it is relevant only to the network component.

In response to your reservation request, the network component returns this value. You then use this value when you set up buffers for sending and receiving media data.

In all other cases, this field is ignored.

Table 10-6 Reservation block field usage

Field	Reserve, setOpts	Reserve, getOpts	Release, setOpts	Release, getOpts
type	kMTReserveType	kMTReserveType	kMTReserveType	kMTReserveType
bandwidth	Bandwidth requested	Bandwidth remaining	Not used	Bandwidth remaining
identity	Not used	Returned; use for release	From reserve	Not used
channel	Not used	Returned; use for media data	Not used	Not used

RESULT CODES

noErr	0	No error
paramErr	–50	Unsupported option type
mtConfigurationErr	–7891	Unable to set or get options
mtUnsupportedNetworkErr	–7892	Unable to set network type
mtIncompatibleStateErr	–7962	Wrong state for specified option
mtOutOfResourcesErr	–7963	Unable to reserve resource

SEE ALSO

You can determine whether a network supports bandwidth reservation by calling the `MTNetworkGetInfo` function, described on page 10-30.

Network component subtype values are discussed in "Network Component Subtypes" on page 10-23.

The data structures used to send and receive media data are discussed in "Data Buffer Structures" on page 10-25.

MTNetworkSetNotifyProc

The `MTNetworkSetNotifyProc` function installs a notify routine. The network component calls your notify routine when certain network-related events occur.

```
pascal ComponentResult MTNetworkSetNotifyProc
                        (MTNetworkComponent nc,
                        MTNotifyUPP notifyProc, long refCon);
```

`nc` The network component you are using.

`notifyProc` A pointer to your notify routine. Set this parameter to `nil` to remove a notify routine you previously installed.

`refCon` Reserved for your use. The network component passes this value when it calls your notify routine.

DESCRIPTION

You call the `MTNetworkSetNotifyProc` function to install a notify routine for a connection. The network component calls your notify routine for a variety of network events, including when a connection is broken for any reason and when the network goes online or offline.

The network component passes your routine an event indicating what happened. This allows you to do whatever processing is appropriate for your application. For example, your response to an orderly disconnect is probably different from your response to an unexpected network failure.

You can remove a previously installed notify routine by setting the `notifyProc` parameter to `nil`.

Not all network components support notify routines. If the network component you are using does not support notify routines, use the `MTNetworkGetNextEvent` function to retrieve network events. Note, however, that installing a notify routine is the preferred method of receiving event notification.

RESULT CODES

`noErr`	0	No error
`mtConfigurationErr`	–7891	Unable to assign notify procedure
`badComponentSelector`	0x80008002	Unsupported function

SEE ALSO

The `MTNetworkGetNextEvent` function is described next.

Notify procedures, along with the network events they may receive, are discussed starting on page 10-68.

MTNetworkGetNextEvent

The `MTNetworkGetNextEvent` function allows you to receive event notification from a network component that does not support notify procedures directly.

```
pascal ComponentResult MTNetworkGetNextEvent
                        (MTNetworkComponent nc,
                        MTNetworkEventType *event,
                        Boolean async, MTNotifyUPP completion,
                        long refCon);
```

`nc` The network component you are using.

`event` A pointer to a location to receive an event. This parameter is ignored if you set the `async` parameter to `true`. The events that you may receive are discussed starting on page 10-68.

`async`	A Boolean value that indicates whether you want the function to execute asynchronously. Set this parameter to `true` if you want asynchronous operation. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your notify routine. The function calls this routine when an asynchronous operation completes. The function ignores this parameter if you call it synchronously.
`refCon`	Reference constant for your use. The network component passes this value to your notify routine.

DESCRIPTION

The `MTNetworkGetNextEvent` function is one of two ways that you can receive event notification from a network component. Most network components include direct support for event notification routines. When you are using one of these components, call the `MTNetworkSetNotifyProc` function to define your notify routine. If the network component you are using does not support notification routines directly, you must retrieve network events yourself by calling the `MTNetworkGetNextEvent` function.

Call this function from within your event loop. You should call the function repeatedly until the function returns an event value of `kMTNetworkNullEvent`.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned

SEE ALSO

The `MTNetworkSetNotifyProc` function is described on page 10-37.

Notify procedures, along with the network events they may receive, are discussed starting on page 10-68.

Managing Connections

This section discusses functions that allow you to receive incoming connections and establish outgoing connections.

The `MTNetworkBindListener` function defines your listener endpoint on the network. You remove that endpoint by calling the `MTNetworkUnbindListener` function. You enable your endpoint to receive incoming connections by calling the `MTNetworkListen` function. You accept an incoming connection with the `MTNetworkAccept` function. You reject or terminate a connection with the `MTNetworkDisconnect` function.

You define your calling endpoint with the `MTNetworkBindCaller` function. You remove that endpoint with the `MTNetworkUnbindCaller` function. You place a

connection with the `MTNetworkConnect` function. You terminate a connection with the `MTNetworkDisconnect` function.

MTNetworkBindListener

The `MTNetworkBindListener` function defines a listening endpoint.

```
pascal ComponentResult MTNetworkBindListener
                        (MTNetworkComponent nc,
                        MTAddress *localAddr, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc`	The network component you are using.
`localAddr`	A pointer to a MovieTalk address record. The function returns the network address of your local listening endpoint. You are responsible for allocating the memory for the address record and for disposing of it.
`async`	A Boolean value that indicates whether you want the function to execute asynchronously. Set this parameter to `true` if you want asynchronous operation. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. The function calls this routine when an asynchronous operation completes. The function ignores this parameter if you call it synchronously.
`refCon`	Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You use the `MTNetworkBindListener` function to define a listening endpoint for receiving incoming connections and to associate a MovieTalk address with that endpoint. The component sets up a listening endpoint that is appropriate for the underyling network and returns an `MTAddress` structure that contains the endpoint's address. You may then enable the endpoint by calling the `MTNetworkListen` function. When you no longer want a listening endpoint, you can remove it by calling the `MTNetworkUnbindListener` function.

This function is part of the process you must go through in order to receive incoming connections. You define a listening endpoint by calling this function. You enable that endpoint by calling the `MTNetworkListen` function. When you receive a connection, you can accept it by calling the `MTNetworkAccept` function or reject it by calling the `MTNetworkDisconnect` function.

When you call `MTNetworkBindListener` asynchronously, you provide a completion routine. The network component calls your completion routine when it has set up your listening endpoint. If you call this function synchronously, no other operations take place

on the computer until this function completes. While setting up the listening endpoint does not typically take a long time, calling the function asynchronously is the most efficient way to use it.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to bind this endpoint
`mtUnsupportedNetworkErr`	–7892	Unsupported network type
`mtIncompatibleStateErr`	–7962	Endpoint already bound
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned

SEE ALSO

The `MTNetworkUnbindListener` function is described next.

The `MTNetworkListen` function is discussed on page 10-42.

The `MTNetworkAccept` function is described on page 10-44.

See page 10-45 for a discussion of the `MTNetworkDisconnect` function.

Completion routines are discussed on page 10-69.

The `MTAddress` structure is described in the chapter "Transport Components," elsewhere in this book.

MTNetworkUnbindListener

The `MTNetworkUnbindListener` function removes a listening endpoint.

```
pascal ComponentResult MTNetworkUnbindListener
                        (MTNetworkComponent nc, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc`	The network component you are using.
`async`	A Boolean value that indicates whether you want the function to execute asynchronously. Set this parameter to `true` if you want asynchronous operation. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. The function calls this routine when an asynchronous operation completes. The function ignores this parameter if you call it synchronously.
`refCon`	Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You use the `MTNetworkUnbindListener` function to remove a listening endpoint for receiving incoming connections. You establish the listening endpoint by calling the `MTNetworkBindListener` function.

When you call `MTNetworkUnbindListener` asynchronously, you provide a completion routine. The network component calls your completion routine when it has removed your listening endpoint. If you call this function synchronously, no other operations take place on the computer until this function completes. While removing the listening endpoint does not typically take a long time, calling the function asynchronously is the most efficient way to use it.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to unbind this endpoint
`mtIncompatibleStateErr`	–7962	Endpoint not bound
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned

SEE ALSO

The `MTNetworkBindListener` function is described on page 10-40.

Completion routines are discussed on page 10-69.

MTNetworkListen

The `MTNetworkListen` function enables your listening endpoint to receive incoming connections.

```
pascal ComponentResult MTNetworkListen (MTNetworkComponent nc,
                        MTAddress *remoteAddr, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc` The network component you are using.

`remoteAddr` A pointer to a MovieTalk address record. The function returns the network address of a remote endpoint that is attempting to set up a connection. You are responsible for allocating the memory for the address record and for disposing of it.

`async` A Boolean value that indicates whether you want the function to execute asynchronously. Set this parameter to `true` if you want asynchronous operation. Otherwise, set it to `false` to specify synchronous execution.

`completion`	A pointer to your completion routine. The function calls this routine when an asynchronous operation completes. The function ignores this parameter if you call it synchronously.
`refCon`	Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You use the `MTNetworkListen` function to listen for a remote endpoint's attempt to connect with your listening endpoint. You establish the listening endpoint by calling the `MTNetworkBindListener` function. When the network component detects an incoming connection, it returns a MovieTalk address record identifying the control endpoint.

This function is part of the process you must go through in order to receive incoming calls. You define a listening endpoint by calling the `MTNetworkBindListener` function. You enable that endpoint by calling this function. When you receive a connection, you can accept it by calling the `MTNetworkAccept` function or reject it by calling the `MTNetworkDisconnect` function.

When you call `MTNetworkBindListener` asynchronously, you provide a completion routine. The network component calls your completion routine when it has set up your listening endpoint. This is the most efficient way to use this function.

SPECIAL CONSIDERATIONS

Calling `MTNetworkListen` synchronously is not recommended. If you do so, no other operations can take place on the computer until an incoming connection is received.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtIncompatibleStateErr`	–7962	Listener endpoint not bound
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned
`mtBadEndPointErr`	–7980	Unable to listen using this endpoint

SEE ALSO

The `MTNetworkAccept` function is described next.

The `MTNetworkBindListener` function is described on page 10-40.

The `MTNetworkUnbindListener` function is discussed on page 10-42.

See page 10-45 for a discussion of the `MTNetworkDisconnect` function.

Completion routines are discussed on page 10-69.

The `MTAddress` structure is described in the chapter "Transport Components," elsewhere in this book.

MTNetworkAccept

The `MTNetworkAccept` function accepts an incoming connection.

```
pascal ComponentResult MTNetworkAccept (MTNetworkComponent nc,
                        const MTAddress *remoteAddr,
                        MTNetworkComponent listener,
                        Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc`	The network component you are using.
`remoteAddr`	A pointer to the MovieTalk address describing the network address of the remote endpoint. You get this value from the `MTNetworkListen` function.
`listener`	The network component that listened for the connection. If you are using the same network component to answer a connection and listen for a connection, set this parameter to `nil`. If you used a different network component to listen for the connection, set this parameter to the listener's component instance.
`async`	A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. If you set the `async` parameter to `true`, the network component calls this routine when it has answered the incoming connection. If you set the `async` parameter to `false`, the network component ignores this parameter.
`refCon`	Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You call the `MTNetworkAccept` function to answer a remote endpoint attempting to make a connection.

This function is part of the process you must go through in order to receive incoming connections. You define a listening endpoint by calling the `MTNetworkBindListener` function. You enable that endpoint by calling the `MTNetworkListen` function. When you receive a connection, you can accept it by calling this function or reject it by calling the `MTNetworkDisconnect` function.

The most efficient way to answer a connection is to specify asynchronous operation and provide a completion routine. In that case, the `MTNetworkAccept` function calls your completion routine when it has answered the connection or detected an error.

The `listener` parameter allows you to respond to incoming connections using a different network component than you use to answer the connections. This can be useful in some networked environments. For example, socket-based TCP/IP networks define

several well-known sockets to receive incoming connections. However, the traffic that results from these connections is typically carried over different sockets. In such an environment, you might want to have one network component listen for incoming connections and another to support the connection once it is established.

SPECIAL CONSIDERATIONS

Calling `MTNetworkAccept` synchronously is not recommended. If you do so, no other operations can take place on the computer until the answer operation succeeds or fails.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConnectionFailedErr`	–7895	Unable to accept
`mtIncompatibleStateErr`	–7962	No incoming request is pending
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned

SEE ALSO

The `MTNetworkBindListener` function is described on page 10-40.

The `MTNetworkListen` function is discussed on page 10-42.

See page 10-45 for a discussion of the `MTNetworkDisconnect` function.

Completion routines are discussed on page 10-69.

The `MTAddress` structure is described in the chapter "Transport Components," elsewhere in this book.

MTNetworkDisconnect

The `MTNetworkDisconnect` function terminates a connection. You can use this function to terminate connections you established by calling the `MTNetworkConnect` function and connections you received by calling the `MTNetworkAccept` function.

```
pascal ComponentResult MTNetworkDisconnect
                        (MTNetworkComponent nc, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc` The network component you are using.

`async` A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.

`completion` A pointer to your completion routine. If you set the `async` parameter to `true`, the network component calls this routine when the function completes. If you set the `async` parameter to `false`, the network component ignores this parameter.

`refCon` Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You call the `MTNetworkDisconnect` function to disconnect an existing connection.

The most efficient way to disconnect a connection is to specify asynchronous operation and provide a completion routine. In that case, the `MTNetworkDisconnect` function calls your completion routine when it completes execution.

SPECIAL CONSIDERATIONS

Calling the `MTNetworkDisconnect` function synchronously is not recommended. If you do so, no other operations can take place on the computer until the operation succeeds or fails.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned
`mtBadEndPointErr`	–7980	Cannot disconnect this endpoint

SEE ALSO

See page 10-44 for a description of the `MTNetworkAccept` function.

The `MTNetworkConnect` function is discussed on page 10-49.

Completion routines are discussed on page 10-69.

MTNetworkBindCaller

The `MTNetworkBindCaller` function establishes a calling endpoint.

```
pascal ComponentResult MTNetworkBindCaller
                        (MTNetworkComponent nc,
                        MTAddress *localAddr, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc` The network component you are using.

`localAddr`	A pointer to a MovieTalk address record. The function returns the network address of your local calling endpoint. You are responsible for allocating the memory for the address record and for disposing of it.
`async`	A Boolean value that indicates whether you want the function to execute asynchronously. Set this parameter to `true` if you want asynchronous operation. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. The function calls this routine when an asynchronous operation completes. The function ignores this parameter if you call it synchronously.
`refCon`	Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You use the `MTNetworkBindCaller` function to define a calling endpoint for placing outgoing connections and to associate a MovieTalk address with that endpoint. The component sets up a calling endpoint that is appropriate for the underyling network and returns an `MTAddress` structure that contains the endpoint's address. You may then place a connection using the endpoint by calling the `MTNetworkConnect` function. When you no longer want a calling endpoint, you can remove it by calling the `MTNetworkUnbindCaller` function.

This function is part of the process you must go through in order to establish outgoing connections. You define a calling endpoint by calling this function. You place a connection using that endpoint by calling the `MTNetworkConnect` function.

When you call the `MTNetworkBindCaller` function asynchronously, you provide a completion routine. The network component calls your completion routine when it has set up your calling endpoint. If you call this function synchronously, no other operations take place on the computer until this function completes. While setting up the calling endpoint does not typically take a long time, calling the function asynchronously is the most efficient way to use it.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to bind this endpoint
`mtUnsupportedNetworkErr`	–7892	Unsupported network type
`mtIncompatibleStateErr`	–7962	Endpoint already bound
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned

SEE ALSO

The `MTNetworkUnbindCaller` function is described next.

The `MTNetworkConnect` function is described on page 10-49.

Completion routines are discussed on page 10-69.

The `MTAddress` structure is described in the chapter "Transport Components," elsewhere in this book.

MTNetworkUnbindCaller

The `MTNetworkUnbindCaller` function removes a calling endpoint.

```
pascal ComponentResult MTNetworkUnbindCaller
                        (MTNetworkComponent nc, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc`	The network component you are using.
`async`	A Boolean value that indicates whether you want the function to execute asynchronously. Set this parameter to `true` if you want asynchronous operation. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. The function calls this routine when an asynchronous operation completes. The function ignores this parameter if you call it synchronously.
`refCon`	Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You use the `MTNetworkUnbindCaller` function to remove a calling endpoint for placing outgoing connections. You establish the calling endpoint by calling the `MTNetworkBindCaller` function.

When you call the `MTNetworkUnbindCaller` function asynchronously, you provide a completion routine. The network component calls your completion routine when it has removed your calling endpoint. If you call this function synchronously, no other operations take place on the computer until this function completes. While removing the calling endpoint does not typically take a long time, calling the function asynchronously is the most efficient way to use it.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to unbind this endpoint
`mtIncompatibleStateErr`	–7962	Endpoint not bound
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned

SEE ALSO

The `MTNetworkBindCaller` function is described on page 10-46.

Completion routines are discussed on page 10-69.

MTNetworkConnect

The `MTNetworkConnect` function allows you to place an outgoing connection.

```
pascal ComponentResult MTNetworkConnect (MTNetworkComponent nc,
                        const MTAddress *remoteAddr,
                        Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc`	The network component you are using.
`remoteAddr`	A pointer to the remote address with which you want to connect. You can get a MovieTalk address from the `MTNetworkLookupName` function.
`async`	A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. If you set the `async` parameter to `true`, the network component calls this routine when the function completes. If you set the `async` parameter to `false`, the network component ignores this parameter.
`refCon`	Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You call the `MTNetworkConnect` function to initiate a connection with a remote connection end. This function places a connection from your calling endpoint.

This function is part of the process you must go through in order to establish outgoing connections. You define a calling endpoint by calling the `MTNetworkBindCaller` function. You establish a connection using that endpoint by calling this function.

You end a connection by calling the `MTNetworkDisconnect` function.

The most efficient way to establish a connection is to specify asynchronous operation and provide a completion routine. In that case, the `MTNetworkConnect` function calls your completion routine when it completes execution.

SPECIAL CONSIDERATIONS

Calling the `MTNetworkConnect` function synchronously is not recommended. If you do so, no other operations can take place on the computer until the connection operation completes or fails.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConnectionFailedErr`	–7895	Unable to connect
`mtIncompatibleStateErr`	–7962	Calling endpoint not bound
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned

SEE ALSO

See page 10-46 for a description of the `MTNetworkBindCaller` function.

The `MTNetworkLookupName` function is discussed on page 10-63.

The `MTNetworkDisconnect` function is described on page 10-45.

Completion routines are discussed on page 10-69.

The `MTAddress` structure is described in the chapter "Transport Components," elsewhere in this book.

Exchanging Control Messages

Network components provide two functions that allow you to exchange control messages with other entities on the network. These messages travel over the network's most reliable services and are generally guaranteed to arrive at the intended destination.

You send a message by calling the `MTNetworkSend` function; you receive a message with the `MTNetworkReceive` function.

MTNetworkSend

The `MTNetworkSend` function sends a control message to a remote connection end via a reliable control channel.

```
pascal ComponentResult MTNetworkSend (MTNetworkComponent nc,
                        MTBuffer *buf, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc`	The network component you are using.
`buf`	A pointer to an `MTBuffer` structure that describes the message.
`async`	A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. If you set the `async` parameter to `true`, the network component calls this routine when the function completes. If you set the `async` parameter to `false`, the network component ignores this parameter.

`refCon` Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You can use the `MTNetworkSend` function to send control messages to the remote end of the connection. The network component uses a reliable network connection to send this data.

The most efficient way to send a message is to specify asynchronous operation and provide a completion routine. In that case, the `MTNetworkSend` function calls your completion routine when it completes execution.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtIncompatibleStateErr`	–7962	No active connection
`mtOutOfResourcesErr`	-7963	Too many asychronous operations already in progress
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned
`mtBadEndPointErr`	–7980	Unable to send to this endpoint
`mtInvalidDataLengthErr`	–7981	Buffer too large

SEE ALSO

See "Data Buffer Structures" on page 10-25 for a description of the `MTBuffer` structure.

Completion routines are discussed on page 10-69.

MTNetworkReceive

You can use the `MTNetworkReceive` function to receive a control message from the other side of a connection via a reliable control channel.

```
pascal ComponentResult MTNetworkReceive (MTNetworkComponent nc,
                        MTRcvBlock *receiveBlock,
                        Boolean async,
                        MTReceiveUPP receiveProc,
                        long refCon);
```

`nc` The network component you are using.

`receiveBlock`
A pointer to a receive block record describing where the network component is to place the received message and the exact size of the message you expect to receive. You must supply values for the `size` and `buffer` fields of this structure.

`async` A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.

`receiveProc`
A pointer to your message receive routine. If you set the `async` parameter to `true`, the network component calls this routine when the function completes. If you set the `async` parameter to `false`, the network component ignores this parameter.

`refCon` Reference constant for your use. The network component passes this value to your message receive routine.

DESCRIPTION

You can use the `MTNetworkReceive` function to receive control messages from the remote end of the connection. The network component receives this message over a reliable control channel and returns the message into a location you specify.

The most efficient way to receive a message is to specify asynchronous operation and provide a message receive routine. In that case, the `MTNetworkReceive` parameter calls your message receive routine when it has received an incoming message.

The `MTNetworkReceive` function does not complete until it successfully receives the exact number of bytes you have specified. If the remote endpoint sends more bytes than you have specified, you will receive them the next time you call this function.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtIncompatibleStateErr`	–7962	No connection active
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned
`mtBadEndPointErr`	–7980	Unable to receive from this endpoint

SEE ALSO

See “Receive Block” on page 10-24 for a description of the receive block record.

Message receive routines are discussed on page 10-67.

Setting Up Media Channels

Before you can exchange media data with a remote endpoint, you must set up the media data channels. Network components provides several routines for this purpose.

You prepare yourself to receive media data by calling the `MTNetworkBindLocalMedia` function. You drop a media channel with the `MTNetworkUnbindLocalMedia` function. You attach to a remote entity for sending data by calling the `MTNetworkBindRemoteMedia` function. You detach by calling the `MTNetworkUnbindRemoteMedia` function.

Note that media data is typically sent over the fastest protocol supported by the underlying network. This is often a "best effort" protocol, which does not necessarily guarantee data delivery.

MTNetworkBindLocalMedia

The `MTNetworkBindLocalMedia` function enables a local endpoint for receiving media data.

```
pascal ComponentResult MTNetworkBindLocalMedia
                        (MTNetworkComponent nc,
                        MTAddress *localAddr, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc`	The network component you are using.
`localAddr`	A pointer to a MovieTalk address record. The function returns the network address of your local endpoint for receiving media data. You are responsible for allocating the memory for the address record and for disposing of it.
`async`	A Boolean value that indicates whether you want the function to execute asynchronously. Set this parameter to `true` if you want asynchronous operation. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. The function calls this routine when an asynchronous operation completes. The function ignores this parameter if you call it synchronously.
`refCon`	Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You use the `MTNetworkBindLocalMedia` function to define an endpoint for receiving incoming media data and to associate a MovieTalk address with that endpoint. The component sets up a media data endpoint that is appropriate for the underlying network and returns an `MTAddress` structure that contains the endpoint's address. Your media data routine may then begin receiving incoming media data. You define your media data routine by calling the `MTNetworkSetReceiveMediaProc` function. When you no longer want a listening endpoint, you can remove it by calling the `MTNetworkUnbindLocalMedia` function.

When you call the `MTNetworkBindLocalMedia` function asynchronously, you provide a completion routine. The network component calls your completion routine when it has set up your listening endpoint. If you call this function synchronously, no other operations take place on the computer until this function completes. While setting up the

endpoint does not typically take a long time, calling the function asynchronously is the most efficient way to use it.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to bind this endpoint
`mtUnsupportedNetworkErr`	–7892	Unsupported network type
`mtIncompatibleStateErr`	–7962	Endpoint already bound

SEE ALSO

The `MTNetworkUnbindLocalMedia` function is described next.

The `MTNetworkSetReceiveMediaProc` function is discussed on page 10-60.

Completion routines are discussed on page 10-69.

Media data routines are discussed on page 10-65.

The `MTAddress` structure is described in the chapter "Transport Components," elsewhere in this book.

MTNetworkUnbindLocalMedia

You can use the `MTNetworkUnbindLocalMedia` function to disable an endpoint for receiving media data.

```
pascal ComponentResult MTNetworkUnbindLocalMedia
                        (MTNetworkComponent nc, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc`	The network component you are using.
`async`	A Boolean value that indicates whether you want the function to execute asynchronously. Set this parameter to `true` if you want asynchronous operation. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. The function calls this routine when an asynchronous operation completes. The function ignores this parameter if you call it synchronously.
`refCon`	Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You use the `MTNetworkUnbindLocalMedia` function to disable an endpoint for receiving incoming media data. You establish the listening endpoint by calling the `MTNetworkBindLocalMedia` function.

When you call the `MTNetworkUnbindLocalMedia` function asynchronously, you provide a completion routine. The network component calls your completion routine when it has disabled your endpoint. If you call this function synchronously, no other operations take place on the computer until this function completes. While disabling the endpoint does not typically take a long time, calling the function asynchronously is the most efficient way to use it.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to unbind this endpoint
`mtUnsupportedNetworkErr`	–7892	Unsupported network type
`mtIncompatibleStateErr`	–7962	Endpoint not bound

SEE ALSO

The `MTNetworkBindLocalMedia` function is described on page 10-53.

Completion routines are discussed on page 10-69.

MTNetworkBindRemoteMedia

You can use the `MTNetworkBindRemoteMedia` function to specify the address of the remote endpoint to which you want to send media data.

```
pascal ComponentResult MTNetworkBindRemoteMedia
                        (MTNetworkComponent nc,
                        const MTAddress *remoteAddr,
                        Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc`	The network component you are using.
`remoteAddr`	A pointer to the remote address to which you want to send media data.
`async`	A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.

`completion` A pointer to your completion routine. If you set the `async` parameter to `true`, the network component calls this routine when the function completes. If you set the `async` parameter to `false`, the network component ignores this parameter.

`refCon` Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You call the `MTNetworkBindRemoteMedia` function to establish a media data connection with a specified network endpoint and to associate a MovieTalk address with that endpoint. Once you have established a media data connection, you can send media data by calling the `MTNetworkSendMedia` function. You drop the media data connection by calling the `MTNetworkUnbindRemoteMedia` function.

The most efficient way to set up a media data connection is to specify asynchronous operation and provide a completion routine. In that case, the `MTNetworkBindRemoteMedia` function calls your completion routine when it completes execution.

SPECIAL CONSIDERATIONS

Calling the `MTNetworkBindRemoteMedia` function synchronously is not recommended. If you do so, no other operations can take place on the computer until the connection is established or the attempt fails.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to bind this endpoint
`mtUnsupportedNetworkErr`	–7892	Unsupported network type
`mtIncompatibleStateErr`	–7962	Endpoint already bound
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned

SEE ALSO

The `MTNetworkUnbindRemoteMedia` function is described next.

See page 10-58 for information about the `MTNetworkSendMedia` function.

The `MTNetworkLookupName` function is discussed on page 10-63.

Completion routines are discussed on page 10-69.

The `MTAddress` structure is described in the chapter "Transport Components," elsewhere in this book.

MTNetworkUnbindRemoteMedia

The `MTNetworkUnbindRemoteMedia` function allows you to disassociate yourself from a remote media endpoint.

```
pascal ComponentResult MTNetworkUnbindRemoteMedia
                        (MTNetworkComponent nc, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc`	The network component you are using.
`async`	A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. If you set the `async` parameter to `true`, the network component calls this routine when the function completes. If you set the `async` parameter to `false`, the network component ignores this parameter.
`refCon`	Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You call the `MTNetworkUnbindRemoteMedia` function to detach from a remote endpoint. You establish media data channels by calling the `MTNetworkBindRemoteMedia` function.

The most efficient way to detach media channels is to specify asynchronous operation and provide a completion routine. In that case, the `MTNetworkUnbindRemoteMedia` function calls your completion routine when it completes execution.

SPECIAL CONSIDERATIONS

Calling the `MTNetworkUnbindRemoteMedia` function synchronously is not recommended. If you do so, no other operations can take place on the computer until the operation succeeds or fails.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to unbind this endpoint
`mtIncompatibleStateErr`	–7962	Endpoint not bound
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned

SEE ALSO

See page 10-55 for a description of the `MTNetworkBindRemoteMedia` function.

Completion routines are discussed on page 10-69.

Exchanging Media Data

Network components provide two routines that allow you to exchange media data with remote endpoints. You send media data by calling the `MTNetworkSendMedia` function. You install your media data receive routine by calling the `MTNetworkSetReceiveMediaProc` function

MTNetworkSendMedia

The `MTNetworkSendMedia` function allows you to send media data to a remote endpoint on the network.

```
pascal ComponentResult MTNetworkSendMedia (MTNetworkComponent nc,
                        MTBuffer *buf, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc`	The network component you are using.
`buf`	A pointer to an `MTBuffer` structure that describes the media data.
`async`	A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.
`completion`	A pointer to your completion routine. If you set the `async` parameter to `true`, the network component calls this routine when the function completes. If you set the `async` parameter to `false`, the network component ignores this parameter.
`refCon`	Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You can use the `MTNetworkSendMedia` function to send media data to the remote end of the connection. The network component uses the most efficient network service to send this data. Often, this is a "best effort" service that does not guarantee data delivery.

The `buf` parameter points to a linked list of `MTBuffer` structures. The network component uses the layout of the structures to determine how it sends the media data. The component sends all of the data associated with each `MTBuffer` structure as a single data packet. As a result, you should make sure that the buffer is not larger than the network's maximum packet size. You can determine the maximum packet size by calling the `MTNetworkOptions` function. However, if you are operating in unsegmented mode, there is no maximum packet size.

The most efficient way to send a message is to specify asynchronous operation and provide a completion routine. In that case, the `MTNetworkSend` function calls your completion routine when it completes execution. If you call the function asynchronously, you must not modify or free the data referred to by the `buf` parameter until the send operation has completed.

SPECIAL CONSIDERATIONS

Sending media data by calling the `MTNetworkSendMedia` function does not guarantee delivery. If the underlying network uses a best effort packet mechanism, then it is possible that some or all of the media data you send across the network may not be delivered.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtIncompatibleStateErr`	–7962	Local or remote endpoint not bound
`mtOutOfResourcesErr`	–7963	Too many asynchronous operations already in progress
`mtBadEndPointErr`	–7980	Unable to send using this endpoint
`mtInvalidDataLengthErr`	–7981	Buffer too large

SEE ALSO

See "Data Buffer Structures" on page 10-25 for a description of the `MTBuffer` structure.

The `MTNetworkOptions` function is described on page 10-32.

Completion routines are discussed on page 10-69.

MTNetworkSetReceiveMediaProc

The `MTNetworkSetReceiveMediaProc` function allows you to install a media data receive routine. The network component calls this routine whenever it receives incoming media data.

```
pascal ComponentResult MTNetworkSetReceiveMediaProc
                        (MTNetworkComponent nc,
                        MTReceiveMediaUPP receiveProc,
                        long refCon, MTMessageSize headerSize);
```

`nc` The network component you are using.

`receiveProc` A pointer to your media data receive routine. Set this parameter to `nil` to remove a routine you previously installed.

`refCon` Reserved for your use. The network component passes this value when it calls your media data receive routine.

`headerSize` A long value specifying the number of bytes to read before calling your media data receive routine.

DESCRIPTION

You use the `MTNetworkSetReceiveMediaProc` function to install your media data receive routine. The network component calls your media data receive routine whenever it receives incoming media data. You enable your connection to receive media data by calling the `MTNetworkBindLocalMedia` function.

The `headerSize` parameter allows you to specify the minimum number of bytes you need in order to make sense out of the incoming data. Typically, this value corresponds to the number of bytes in any headers that precede your media data.

RESULT CODES

`noErr`	0	No error
`mtConfigurationErr`	–7891	Unable to assign routine
`mtIncompatibleStateErr`	–7962	Local endpoint not bound

SEE ALSO

See page 10-53 for information on the `MTNetworkBindLocalMedia` function.

Media data receive routines are discussed starting on page 10-65.

Managing Network Names

Network components provide a number of routines that allow you to define and resolve network names.

Use the MTNetworkRegisterName function to register your name on the network. The MTNetworkRemoveName function allows you to remove your network name. You can resolve a remote endpoint's name by calling the MTNetworkLookupName function. You can obtain a remote address by calling the MTNetworkExtractName function.

MTNetworkRegisterName

The MTNetworkRegisterName function registers a name on a network.

```
pascal ComponentResult MTNetworkRegisterName
                        (MTNetworkComponent nc,
                        ConstStr255Param name,
                        ConstStr255Param serviceType,
                        MTName *localName, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

nc — The network component you are using.

name — A pointer to a string containing the network name you want to register. The name can be a user's name or a name that identifies the local computer. Its format is up to you, though it must comply with any naming rules imposed by the underlying network.

serviceType — A pointer to a string containing the type of network service you want to support. You determine the type of service and the string that identifies the service. The type depends on the type of connection or service you are providing and should be meaningful to and recognizable by your application.

localName — A pointer to a MovieTalk name record that is to receive the new name.

async — A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to true. Otherwise, set it to false to specify synchronous execution.

completion — A pointer to your completion routine. If you set the async parameter to true, the network component calls this routine when the function completes. If you set the async parameter to false, the network component ignores this parameter.

refCon — Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You call the MTNetworkRegisterName function to register a name on a network. This allows someone who wants to call you to look up your name and find your address.

The network component constructs a MovieTalk name record for you, based on the information you specify. Then, if possible and appropriate, it registers the name on the underlying network. The component returns the name in the location you specify with the `localName` parameter.

You cannot have more than one registered name at a time. If you registered a network name previously, you must call the `MTNetworkRemoveName` function to remove that name before you register a new name.

The most efficient way to register a name is to specify asynchronous operation and provide a completion routine. In that case, the `MTNetworkRegisterName` parameter calls your completion routine when it completes execution.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to register this name
`mtIncompatibleStateErr`	–7962	You already have a registered name
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned
`mtDuplicateNameErr`	–7971	Name already in use

SEE ALSO

The `MTNetworkRemoveName` function is discussed next.

MovieTalk names are discussed in the "Browser Components" chapter of this book.

Completion routines are discussed on page 10-69.

MTNetworkRemoveName

You can use the `MTNetworkRemoveName` function to remove a previously registered name from the network.

```
pascal ComponentResult MTNetworkRemoveName
                        (MTNetworkComponent nc, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc` The network component you are using.

`async` A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.

`completion` A pointer to your completion routine. If you set the `async` parameter to `true`, the network component calls this routine when the function completes. If you set the `async` parameter to `false`, the network component ignores this parameter.

`refCon` Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

Use the `MTNetworkRemoveName` function to remove a name you have registered previously. You register network names by calling the `MTNetworkRegisterName` function. You must remove an old name before you can register a new one.

The most efficient way to remove a name is to specify asynchronous operation and provide a completion routine. In that case, the `MTNetworkRemoveName` function calls your completion routine when it completes execution.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to remove name
`mtIncompatibleStateErr`	–7962	Component not registered
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned

SEE ALSO

See page 10-61 for information on the `MTNetworkRegisterName` function.

Completion routines are discussed on page 10-69.

MTNetworkLookupName

The `MTNetworkLookupName` function allows you to find a specified name on the network. Together with the `MTNetworkExtractName` function, you can use this function to determine a remote endpoint's address.

```
pascal ComponentResult MTNetworkLookupName
                        (MTNetworkComponent nc,
                        const MTName *remoteName, Boolean async,
                        MTCompletionUPP completion,
                        long refCon);
```

`nc` The network component you are using.

`remoteName` A pointer to a network name record that contains the name for this request.

`async` A Boolean value that specifies whether you want the function to execute asynchronously. If you do, set this parameter to `true`. Otherwise, set it to `false` to specify synchronous execution.

`completion` A pointer to your completion routine. If you set the `async` parameter to `true`, the network component calls this routine when the function completes. If you set the `async` parameter to `false`, the network component ignores this parameter.

`refCon` Reference constant for your use. The network component passes this value to your completion routine.

DESCRIPTION

You can use the `MTNetworkLookupName` function, along with the `MTNetworkExtractName` function, to resolve a network name to its MovieTalk address. When you call this function, the network component resolves the name and determines the corresponding network address. You must then call the `MTNetworkExtractName` function to retrieve the network address corresponding to the name.

The most efficient way to resolve a name is to specify asynchronous operation and provide a completion routine. In that case, the `MTNetworkLookupName` function calls your completion routine when it completes execution.

RESULT CODES

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to lookup name
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned
`mtNameNotFoundErr`	–7972	Name not found

SEE ALSO

The `MTNetworkExtractName` function is discussed next.

Completion routines are discussed on page 10-69.

The `MTName` structure is described in the chapter "Transport Components," elsewhere in this book.

MTNetworkExtractName

Once you have resolved a network name by calling the `MTNetworkLookupName` function, you can use the `MTNetworkExtractName` function to retrieve the MovieTalk address corresponding to that name.

```
pascal ComponentResult MTNetworkExtractName (MTNetworkComponent
                        nc, MTAddress *remoteAddr);
```

`nc` The network component you are using.

`remoteAddr` A pointer to a MovieTalk address record that is to receive the network address corresponding to the name you just resolved.

DESCRIPTION

You can use the `MTNetworkLookupName` function, along with the `MTNetworkExtractName` function, to resolve a network name to its MovieTalk address. When you call the `MTNetworkLookupName` function, the network component resolves the name and determines the corresponding network address. You must then call the `MTNetworkExtractName` function to retrieve the network address corresponding to the name.

RESULT CODES

`noErr`	0	No error
`mtConfigurationErr`	-7891	Unable to extract name
`mtIncompatibleStateErr`	–7962	No lookup operation performed

SEE ALSO

The `MTNetworkLookupName` function is discussed on page 10-63.

The `MTAddress` structure is described in the chapter "Transport Components," elsewhere in this book.

Application-Defined Functions

This section describes the completion routine and callback routines that you may provide to a network component. Network components call all of these routines at interrupt time.

A network component calls your media data receive routine when it receives media data.

A network component calls your message routine when it receives a message on its control channel.

A network component calls your notify routine in response to certain network events.

A network component calls your completion routine when a function you called asynchronously completes execution.

MyMediaDataProc

When you call the `MTNetworkSetReceiveMediaProc` function, you provide a pointer to a media data receive routine. The network component calls this routine when it receives media data.

Your routine must support the following interface (the name is arbitrary):

Note
This application-defined function does not use Pascal calling conventions. ◆

```
Byte *MyMediaDataProc (MTRcvBlock *rbp);
```

`rbp` A pointer to a receive block record. This record describes the received media data.

DESCRIPTION

A network component calls your media data receive routine whenever it receives your media data. It is up to your routine to process the received media data.

There are two main services your media data receive routine provides: buffer management and data reception. In all cases, the network component uses the receive block record to exchange information with your routine.

Network components operate in two modes: segmented and unsegmented. Unsegmented mode allows network components to exchange arbitrarily large pieces of media data without segmentation. When receiving data, the network component uses the receive block record, and your routine, differently depending upon the type of network connection. The network component preserves the arbitrary packet boundaries that you establish when you send the data.

For segmented connections, your routine may be called once or twice for each packet. In addition, the incoming packet does not necessarily correspond to a media chunk.

The network component first calls your routine when it has assembled a valid packet header (you specify the size of the header when you call the `MTNetworkSetReceiveMediaProc` function). The component sets the `kMTRcvBlockStartMask` flag in the `flags` field to 1, indicating that it is returning the header only. The `header` field points to the header itself. The `refcon` field is set appropriately. The `channel`, `size`, and `error` fields are not used. When you return control to the network component, set the `kMTRcvBlockCompleteMask` flag to 1 if you want the component to call your component again after it has received the media data following the header in the packet. Set your returned pointer to the buffer where the component is to place the media data.

The second time the component calls your routine, the network component returns the media data (the component does not call your routine a second time unless you set the `kMTRcvBlockCompleteMask` flag to 1 in response to the previous call). The `size` field refers to the size of the media data. The `buffer` field points to the media data. The `header` field still points to the packet header. The `refcon` and `error` fields are set appropriately. The `channel` field is not used. When your routine is called the second time, set your returned value to `nil`.

For connections that are operating in unsegmented mode, the component calls your routine to obtain buffers from you and to return media data.

When the network component calls your routine to retrieve buffers for incoming data, it sets the `kMTRcvBlockGetBufferMask` flag in the `flags` field to 1. The `size` field

indicates the desired buffer size. The `refcon` field contains your reference constant. All other fields are not used. Return the buffer's address as your routine's returned value. The network component now controls that buffer until it returns the buffer of media data to you. Be aware that the network component may stockpile buffers and may return buffers in any order. In any case, the network component returns a given buffer only once per use.

Note that, if the `kMTRcvBlockInterruptMask` flag is 1, it is interrupt time, and your routine must not move memory. If you haven't already allocated a buffer, use the memory component's interrupt safe memory allocation routine. If you cannot return a buffer, set your returned value to `nil` and wait for the component to call you again.

After receiving a packet, the component returns a buffer of media data to you. The `size` field indicates the amount of data returned. The `buffer` field points to the data buffer. The `refcon` and `channel` fields are set appropriately. The `kMTRcvBlockCompleteMask` flag in the flags field is set to 1, indicating that the receive operation is complete. If the component sets the `kMTRcvBlockGetBufferMask` flag to 1, be sure to return a new buffer when you return control to the component. The `error` field indicates any receive errors. The `header` field is not used.

SEE ALSO

The receive block record is discussed in "Receive Block" on page 10-24.

See page 10-60 for information about the `MTNetworkSetReceiveMediaProc` function.

MyMessageProc

When you call the `MTNetworkReceive` function, you can provide a pointer to a message routine. A network component calls this routine when it receives a control message and you have called the function asynchronously.

Your routine must support the following interface (the name is arbitrary):

```
pascal void MyMessageProc (const MTRcvBlock *rbp);
```

`rbp` A pointer to a receive block record. This record describes the received message.

DESCRIPTION

The network component delivers the message to your routine in a receive block record. The component allocates the memory for the message buffer and sets the fields in the receive block appropriately. The `refcon` field contains the reference constant you supplied when you called the `MTNetworkReceive` function. The `error` field indicates any receive error; if there is an error, then only the `refcon` and `error` fields are valid.

The `size` field indicates the number of bytes in the message. The `buffer` field points to the message. The `flags`, `header`, and `channel` fields are not used.

SEE ALSO

The receive block record is discussed in "Receive Block" on page 10-24.

See page 10-51 for information about the `MTNetworkReceive` function.

MyNotifyProc

When you call the `MTNetworkSetNotifyProc` function, you provide a pointer to a notify routine. The network component calls this routine in response to certain network events, such as an orderly disconnect initiated by the remote connection end. This function is also the completion routine for the `MTNetworkGetNextEvent` function.

Your routine must support the following interface (the name is arbitrary):

```
pascal void MyNotifyProc (MTNetworkEventType event, long refCon);
```

`event` The event that triggered the call to your routine.

`refCon` The reference constant that you provided in the `refCon` parameter to the `MTNetworkSetNotifyProc` function. You can use this value for whatever purpose you choose.

DESCRIPTION

In order to maintain network connections, your application needs to be able to receive notification of certain network events. Some network components allow you to define a notify routine by calling the `MTNetworkSetNotifyProc` function. Others require you to manage the events yourself by calling the `MTNetworkGetNextEvent` function.

In either case, network components use the `event` parameter to tell you what has happened. The following constants define the events that may cause a network component to call your notify routine.

Constant descriptions

`kMTNetworkNullEvent`
Indicates that there is no event pending. You receive this event only when you call the `MTNetworkGetNextEvent` function.

`kMTNetworkFailEvent`
Indicates some failure in the remote side hardware or software or in the network or a network disconnect for some other unexpected reason. The existing connection is broken.

`kMTNetworkCloseEvent`
Indicates an orderly disconnect. An orderly disconnect is originated by the user at the remote end.

`kMTNetworkOfflineEvent`
: Indicates that no network services are available—the network is offline or disconnected. For example, when a user switches the computer's AppleTalk network connection from TokenTalk to EtherTalk, the network is not available during the transition.

`kMTNetworkOnlineEvent`
: Indicates that network services are again available.

A network component calls your notify routine when a connection is broken for any reason and when network services become available or unavailable. It passes you the event that triggered the call to your routine. Your application can then take whatever actions are appropriate to handle that event.

SEE ALSO

The `MTNetworkSetNotifyProc` function is discussed on page 10-37.

The `MTNetworkGetNextEvent` function is discussed on page 10-38.

MyCompletionProc

When you call a function asynchronously, you can provide a completion routine for the network component to call when it completes execution.

Your routine must support the following interface (the name is arbitrary):

```
pascal void MyCompletionRoutine (ComponentResult err,
                                 long refCon);
```

`err` The function's result code.

`refCon` The reference constant that you provided to the function that calls your completion routine.

DESCRIPTION

You provide a completion routine to any network component function that you can call asynchronously.

When a function that you called asynchronously completes execution, the network component calls your completion routine and passes its result code in the `err` parameter. It also passes your reference constant value.

SPECIAL CONSIDERATIONS

Because your completion routine may be called at interrupt time, you must not make calls to the Memory Manager, either directly or indirectly. If your completion routine needs to access global variables, you can pass a reference to your application's A5

register value or to your component's global variables in the `refCon` parameter when you issue the asynchronous function call.

SEE ALSO

For more information about the A5 register, see *Inside Macintosh: Memory.*

Summary of Network Components

C Summary

Constants

```
/*----------------------------------------
   Network selectors
----------------------------------------*/

enum {
   kMTNetworkGetInfoSelect =             1,
   kMTNetworkOptionsSelect =             2,
   kMTNetworkBindLocalMediaSelect =      3,
   kMTNetworkUnbindLocalMediaSelect =    4,
   kMTNetworkBindRemoteMediaSelect =     5,
   kMTNetworkUnbindRemoteMediaSelect =   6,
   kMTNetworkSendMediaSelect =           7,
   kMTNetworkSetReceiveMediaProcSelect = 8,
   kMTNetworkBindListenerSelect =        9,
   kMTNetworkUnbindListenerSelect =      10,
   kMTNetworkBindCallerSelect =          11,
   kMTNetworkUnbindCallerSelect =        12,
   kMTNetworkListenSelect =              13,
   kMTNetworkAcceptSelect =              14,
   kMTNetworkConnectSelect =             15,
   kMTNetworkDisconnectSelect =          16,
   kMTNetworkSendSelect =                17,
   kMTNetworkReceiveSelect =             18,
   kMTNetworkGetNextEventSelect =        19,
   kMTNetworkSetNotifyProcSelect =       20,
   kMTNetworkRegisterNameSelect =        21,
```

```
   kMTNetworkRemoveNameSelect =          22,
   kMTNetworkLookupNameSelect =          23,
   kMTNetworkExtractNameSelect =         24
};
```

Data Types

```
typedef ComponentInstance              MTNetworkComponent;
typedef UInt32                         MTChannel;
typedef UInt32                         MTOptionFlags;
typedef UInt32                         MTReservationID;

enum {
   kMTRcvBlockStartMask =              0x0001,
   kMTRcvBlockCompleteMask =           0x0002,
   kMTRcvBlockGetBufferMask =          0x0004,
   kMTRcvBlockInterruptMask =          0x0008
};
typedef UInt32                         MTRcvBlockFlags;

struct MTRcvBlock {
   UInt32                              refcon;
   UInt32                              size;
   MTRcvBlockFlags                     flags;
   ComponentResult                     error;
   Byte                                *header;
   Byte                                *buffer;
   MTChannel                           channel;
};
typedef struct MTRcvBlock              MTRcvBlock, *MTRcvBlockPtr;

enum {
   kMTOpenTransportType =              1,
   kMTRequestReservationType =         3,
   kMTReleaseReservationType =         4
};
typedef UInt32                         MTOptionType;

struct MTOptions {
   OSType                              networkType;
   MTOptionFlags                       flags;
   UInt16                              maxlen;
   UInt16                              length;
```

```
   Byte                                *buffer;
};
typedef struct MTOptions               MTOptions, *MTOptionsPtr;

enum {
   kMTReserveType =                    'mtrs'
};
typedef OSType                         MTReserveType;

struct MTReservation {
   MTReserveType                       flowSpecType;
   MTBitsPerSecond                     bandwidth;
   MTReservationID                     identity;
   MTChannel                           channel;
};

struct MTBlock {
   Byte                                *data;
   UInt16                              maxLength;
   UInt16                              length;
   UInt16                              offset;
   UInt16                              count;
};
typedef struct MTBlock                 MTBlock, *MTBlockPtr;

struct MTSegment {
   struct MTSegment                    *nextSegment;
   MTBlockPtr                          thisBlock;
};
typedef struct MTSegment               MTSegment, *MTSegmentPtr;


enum {
   mtBufferPendingMask =               0x0001
};
typedef UInt32                         MTBufferFlags;

struct MTBuffer {
   struct MTBuffer                     *nextBuffer;
   MTSegmentPtr                        firstSegment;
   MTBufferFlags                       flags;
   ComponentResult                     error;
   MTChunkPriority                     priority;
```

```
   MTChannel                        channel;
};
typedef struct MTBuffer             MTBuffer, *MTBufferPtr;


enum {
   kMTNetworkNullEvent =            0x0000,
   kMTNetworkFailEvent =            0x0001,
   kMTNetworkCloseEvent =           0x0002,
   kMTNetworkOfflineEvent =         0x0004,
   kMTNetworkOnlineEvent =          0x0008
};
typedef UInt32                      MTNetworkEventType;


/*----------------------------------------
   Network callback functions
----------------------------------------*/
typedef Byte * (*MTReceiveMediaProcPtr) (MTRcvBlock *rbp);

typedef pascal void (*MTReceiveProcPtr) (const MTRcvBlock *rbp);

typedef pascal void (*MTNotifyProcPtr) (MTNetworkEventType event,
                                        long refCon);

typedef MTReceiveMediaProcPtr       MTReceiveMediaUPP;
typedef MTReceiveProcPtr            MTReceiveUPP;
typedef MTNotifyProcPtr             MTNotifyUPP;
```

Functions

Managing Network Components

```
pascal ComponentResult MTNetworkGetInfo (MTNetworkComponent nc,
                            MTInfoSelector whichInfo, long *result);
pascal ComponentResult MTNetworkOptions (MTNetworkComponent nc, MTOptionType
                            optionType, const MTOptions *setOpts,
                            MTOptions *getOpts);
pascal ComponentResult MTNetworkSetNotifyProc (MTNetworkComponent nc,
                            MTNotifyUPP notifyProc, long refCon);
pascal ComponentResult MTNetworkGetNextEvent (MTNetworkComponent nc,
                            MTNetworkEventType *event, Boolean async,
                            MTNotifyUPP completion, long refCon);
```

Managing Connections

```
pascal ComponentResult MTNetworkBindListener (MTNetworkComponent nc,
                            MTAddress *localAddr, Boolean async,
                            MTCompletionUPP completion, long refCon);
pascal ComponentResult MTNetworkUnbindListener (MTNetworkComponent nc,
                            Boolean async, MTCompletionUPP completion,
                            long refCon);
pascal ComponentResult MTNetworkListen (MTNetworkComponent nc, MTAddress
                            *remoteAddr, Boolean async, MTCompletionUPP
                            completion, long refCon);
pascal ComponentResult MTNetworkAccept (MTNetworkComponent nc, const
                            MTAddress *remoteAddr, MTNetworkComponent
                            listener, Boolean async, MTCompletionUPP
                            completion, long refCon);
pascal ComponentResult MTNetworkDisconnect (MTNetworkComponent nc, Boolean
                            async, MTCompletionUPP completion, long
                            refCon);
pascal ComponentResult MTNetworkBindCaller (MTNetworkComponent nc, MTAddress
                            *localAddr, Boolean async, MTCompletionUPP
                            completion, long refCon);
pascal ComponentResult MTNetworkUnbindCaller (MTNetworkComponent nc, Boolean
                            async, MTCompletionUPP completion, long
                            refCon);
pascal ComponentResult MTNetworkConnect (MTNetworkComponent nc, const
                            MTAddress *remoteAddr, Boolean async,
                            MTCompletionUPP completion, long refCon);
```

Exchanging Control Messages

```
pascal ComponentResult MTNetworkSend (MTNetworkComponent nc, MTBuffer *buf,
                            Boolean async, MTCompletionUPP completion,
                            long refCon);
pascal ComponentResult MTNetworkReceive (MTNetworkComponent nc, MTRcvBlock
                            *receiveBlock, Boolean async, MTReceiveUPP
                            receiveProc, long refCon);
```

Setting Up Media Channels

```
pascal ComponentResult MTNetworkBindLocalMedia (MTNetworkComponent nc,
                            MTAddress *localAddr, Boolean async,
                            MTCompletionUPP completion, long refCon);
pascal ComponentResult MTNetworkUnbindLocalMedia (MTNetworkComponent nc,
                            Boolean async, MTCompletionUPP completion,
                            long refCon);
```

```
pascal ComponentResult MTNetworkBindRemoteMedia (MTNetworkComponent nc,
                            const MTAddress *remoteAddr, Boolean async,
                            MTCompletionUPP completion, long refCon);
pascal ComponentResult MTNetworkUnbindRemoteMedia (MTNetworkComponent nc,
                            Boolean async, MTCompletionUPP completion,
                            long refCon);
```

Exchanging Media Data

```
pascal ComponentResult MTNetworkSendMedia (MTNetworkComponent nc, MTBuffer
                            *buf, Boolean async, MTCompletionUPP
                            completion, long refCon);
pascal ComponentResult MTNetworkSetReceiveMediaProc (MTNetworkComponent nc,
                            MTReceiveMediaUPP receiveProc, long refCon,
                            MTMessageSize headerSize);
```

Managing Network Names

```
pascal ComponentResult MTNetworkRegisterName (MTNetworkComponent nc,
                            ConstStr255Param name, ConstStr255Param
                            serviceType, MTName *localName, Boolean async,
                            MTCompletionUPP completion, long refCon);
pascal ComponentResult MTNetworkRemoveName (MTNetworkComponent nc, Boolean
                            async, MTCompletionUPP completion, long
                            refCon);
pascal ComponentResult MTNetworkLookupName (MTNetworkComponent nc, const
                            MTName *remoteName, Boolean async,
                            MTCompletionUPP completion, long refCon);
pascal ComponentResult MTNetworkExtractName (MTNetworkComponent nc,
                            MTAddress *remoteAddr);
```

Application-Defined Functions

```
Byte *MyMediaDataProc (MTRcvBlock *rbp);
pascal void MyMessageProc (const MTRcvBlock *rbp);
pascal void MyNotifyProc (MTNetworkEventType event, long refCon);
pascal void MyCompletionRoutine (ComponentResult err, long refCon);
```

Result Codes

`noErr`	0	No error
`mtPendingAsyncCallErr`	–7860	Asynchronous operation in progress
`mtConfigurationErr`	–7891	Unable to support request at this time

`mtUnsupportedNetworkErr`	–7892	Unsupported network type
`mtConnectionFailedErr`	–7895	Unable to connect
`mtIncompatibleStateErr`	–7962	Inappropriate state for the operation
`mtOutOfResourcesErr`	–7963	Out of resources
`mtOperationAbandonedErr`	–7964	Asynchronous operation abandoned
`mtDuplicateNameErr`	–7971	Network name already in use
`mtNameNotFoundErr`	–7972	Network name not found
`mtBadEndPointErr`	–7980	Invalid endpoint for this operation
`mtInvalidDataLengthErr`	–7981	Buffer too large
`badComponentSelector`	0x80008002	Unsupported function

MovieTalk Protocol Messages

Contents

This chapter discusses the MovieTalk protocol. This network-independent protocol supports QuickTime Conferencing. If you are developing a QuickTime Conferencing application, you do not need to be familiar with the material in this chapter. If you are planning to support QuickTime Conferencing in a non-Macintosh environment or if you need to understand MovieTalk protocol flows, you should understand the material presented here. In order to understand this material fully, you should also be familiar with transport, stream director, and conference components. Consult the appropriate chapters elsewhere in this book.

About the MovieTalk Protocol

QuickTime Conferencing consists of two major elements:

- A suite of programming interfaces provided by a number of Macintosh components. These components allow applications to send and receive time-based media streams across local and wide area networks.
- A protocol for exchanging control and media data across a network.

In both cases, QuickTime Conferencing was designed to be as open and flexible as possible. The programming interfaces provided by the various conferencing components are described elsewhere in this book. This chapter describes the format and content of the protocol messages that constitute the MovieTalk protocol.

The MovieTalk protocol messages fit into the following major categories:

- **Managing connections**: These messages are used by transport components to set up and manage MovieTalk connections.
- **Managing channels and streams**: These messages are used by stream director components to open and manage media data channels and streams.
- **Conferencing**: These messages are sent and received by conference components to establish and manage conferences among applications.

The MovieTalk protocol is a high-level protocol that uses lower-level network services in a number of ways. The MovieTalk protocol uses two distinct links to transmit and receive conferencing data. The Connection Control Channel (CCC) carries exclusively control messages. This channel runs over reliable network protocols. As a result, the MovieTalk protocol assumes a reliable data link and does not provide for explicit acknowledgment messages in most cases.

The Media Stream Channel (MSC) carries media data and, in some cases, some control messages. This channel may run over either an unreliable (best effort) link or a reliable link, depending upon the available network services. As a result, the MovieTalk protocol assumes that any given message on the MSC may be lost. Applications must be able to handle lost media data messages.

The remainder of this chapter discusses the protocol in more detail and is divided into the following sections:

- "Sequences" discusses several interesting MovieTalk protocol sequences at a high level, showing how the control message and media data packets interact.
- "Data Types" provides information about how data is represented in the MovieTalk protocol.
- "Messages" describes the format and content of the various MovieTalk protocol messages.

Sequences

This section discusses a number of MovieTalk message sequences. During the life of a MovieTalk connection, most of the data flow carries the conference's media data. The media data is sent using best effort facilities without acknowledgment. When sent, the data is properly sequenced by media chunk and packetized for transmission on the underlying network. It is the responsibility of the recipient to reorder the packets and to reconstruct the transmitted chunks.

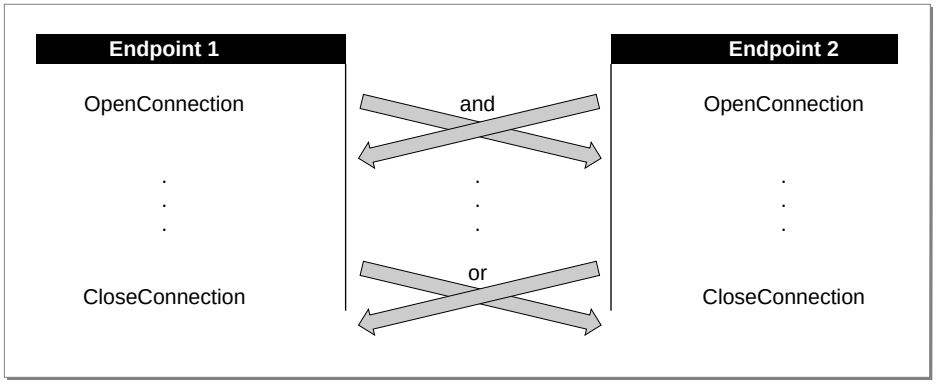
There are also a number of other sequences that have to do with opening, closing, and maintaining connections, negotiating channels and streams, and setting up and managing conference component conferences. These sequences are more complex and more protocol specific than media data exchange and are discussed in the following sections.

Opening and Closing a Connection (Point-to-Point)

Transport components open point-to-point connections between two endpoints by exchanging OpenConnection messages. That connection is closed whenever either endpoint sends a CloseConnection message (or simply drops the link over which the control connection is operating).

Figure 11-1 shows the message exchanges that open and close a point-to-point connection.

Figure 11-1 Opening and closing point-to-point connections



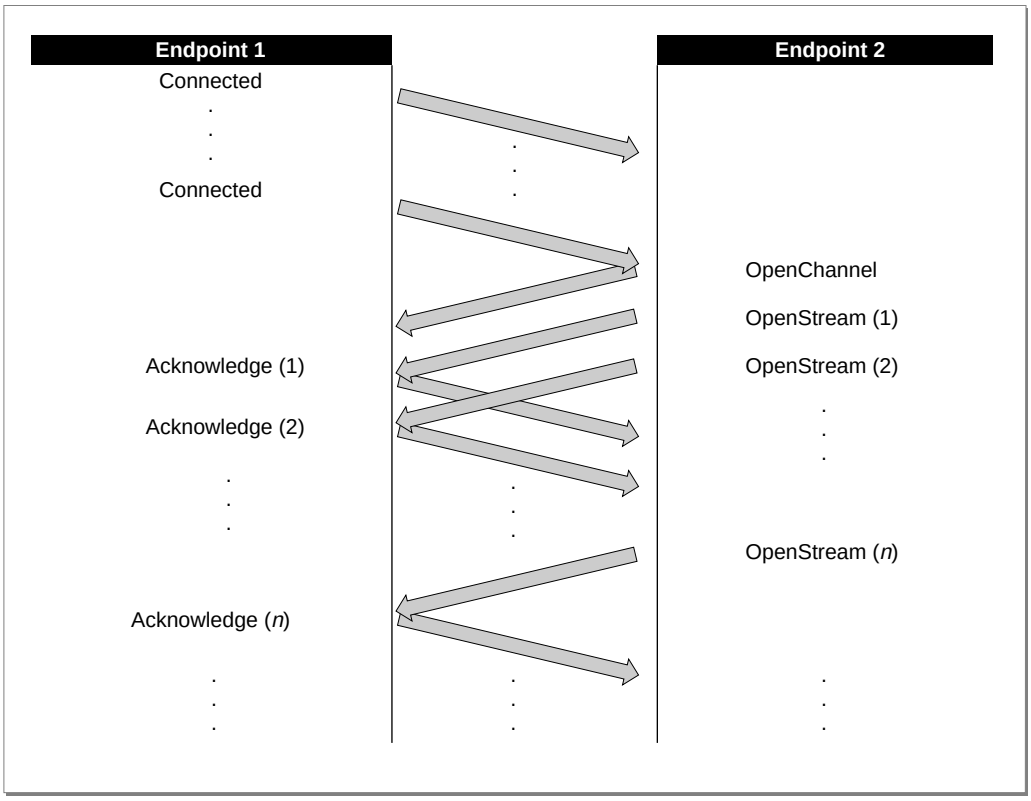
Negotiating a Channel (Point-to-Point)

Stream director components negotiate a channel by sending an OpenChannel message, followed by the appropriate number of OpenStream messages. The receiving endpoint acknowledges each of the OpenStream messages by sending an appropriately formatted Acknowledge message.

Note that the negotiation process may recur at any time during the life of a connection. During such a renegotiation, neither side sends a Connected message; the renegotiation starts with an OpenChannel message.

Figure 11-2 shows the message exchanges that negotiate a point-to-point channel.

Figure 11-2 Negotiating a point-to-point channel



Opening and Closing a Connection (Multicast)

MovieTalk multicast connections are inherently simplex; that is, data flow is unidirectional from the data source to the various data sinks. A transport component opens a multicast connection between two endpoints whenever a receiving endpoint "taps into" the media data stream.

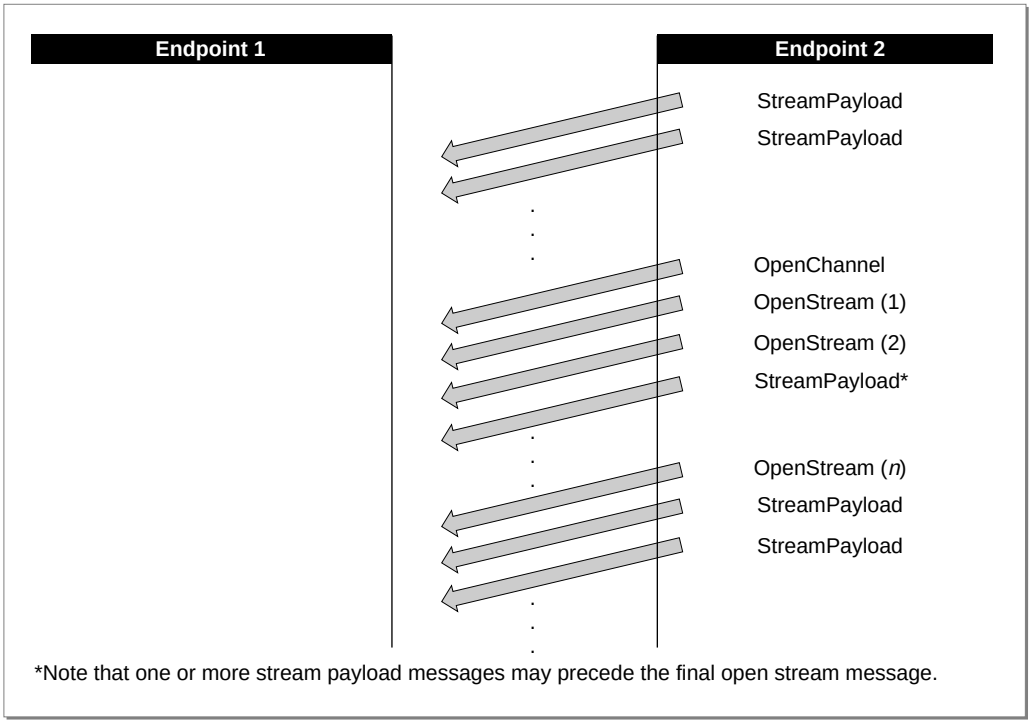
That connection is closed whenever the data source sends a CloseConnection message (or simply drops the link over which the media data is flowing). A multicast data sink should handle link timeouts by closing its connection.

Negotiating a Channel (Multicast)

For multicast connections, the data source stream director component periodically sends the appropriate negotiation messages in the media data stream. New recipients can use the negotiation messages to establish an appropriate conference environment. Existing recipients monitor these messages in order to determine when to renegotiate the connection.

Figure 11-3 shows the message exchanges that negotiate a multicast channel.

Figure 11-3 Negotiating a multicast channel

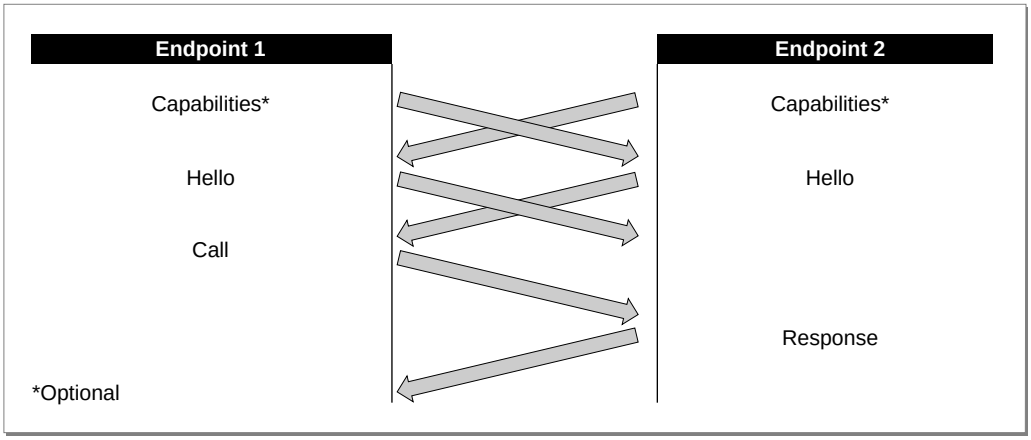


Setting Up a Conference

In order to set up a conference, the conference components at each end may exchange capability information. The calling member then causes a Call message to be sent. When the receiving user answers the call, the conference component at that end sends a Response message and the conference is established. If the receiving user does not answer before the specified timeout interval, the caller terminates the call.

Figure 11-4 shows the message exchanges that open a conference component conference.

Figure 11-4 Opening a conference

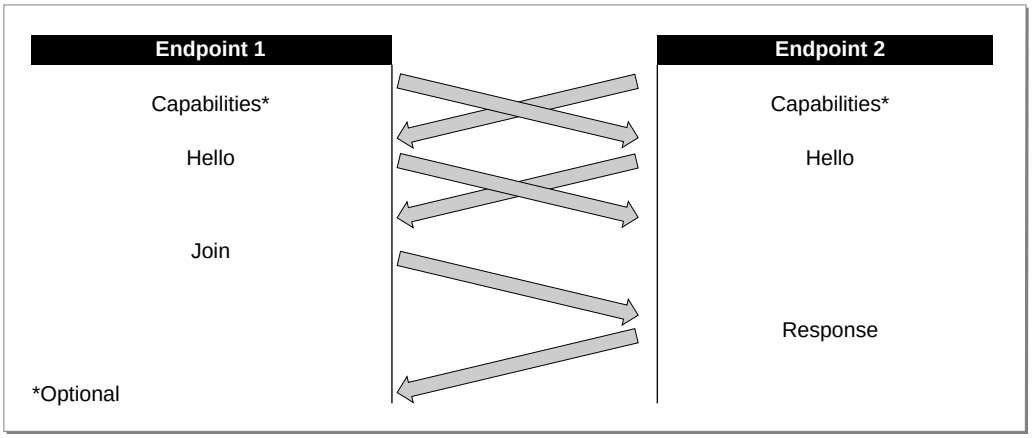


Joining an Existing Conference

Before joining an existing conference, the new member must exchange capability information with a member of the conference. In order to enter the conference, the new member sends a Join message (rather than a Call message) to each member of the conference. As with a new conference, the existing members send an appropriately formatted Response message, welcoming the new member to the conference. Note that the new member may be bringing other new members as well. In that case, each of the new members send their own Join messages in response to Merge messages they receive.

Figure 11-5 shows the message exchanges that allow a new member to join an existing conference component conference.

Figure 11-5 Joining an existing conference

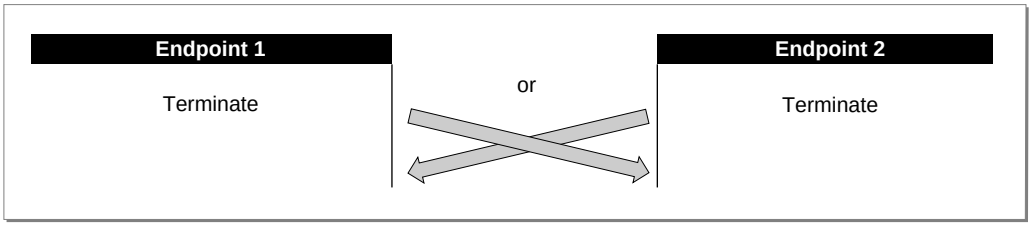


Leaving a Conference

A conference member may leave at any time by sending a Terminate message to other conference members. One message must be sent to each member.

Figure 11-6 shows the message exchanges that allow a member to leave a conference.

Figure 11-6 Leaving a conference



Data Types

This section discusses the ways in which data is represented in the MovieTalk protocol. This section is divided into the following topics:

- "Data Representation" discusses data representation issues that stem from the Macintosh machine architecure and that persist in the MovieTalk protocol.
- "Transport Data Types" discusses the data types that are used in transport component messages.
- "Stream Director Data Types" discusses the data types that are used in stream director component messages.
- "Conferencing Data Types" discusses how data is represented in conference component messages.

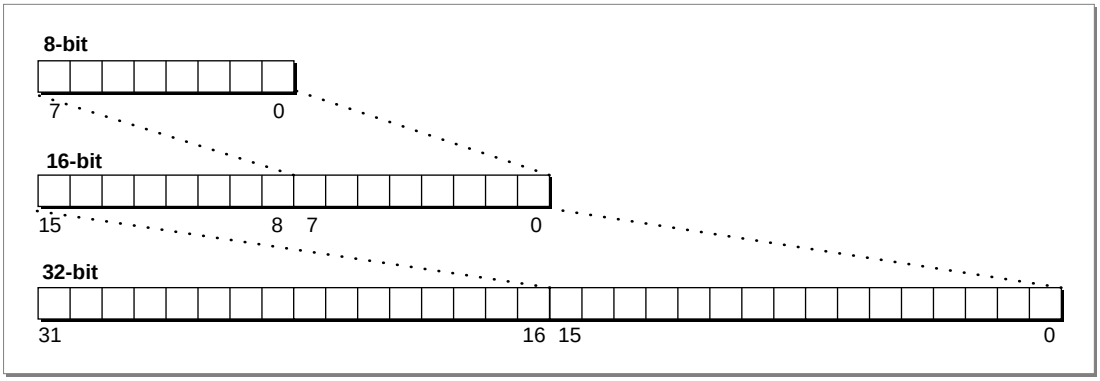
Data Representation

There are several aspects of MovieTalk protocol data representation that are driven by the machine architecture of the Macintosh family of computers. Specifically, this affects bit numbering within binary data types, byte-ordering within binary data types, and the widths of the various canonical data types. This section discusses these issues.

Bits within binary data types are numbered in a "big-endian" fashion, sequentially from right to left. That is, the low-order bit in a field is the rightmost bit and is always bit 0. The high-order bit is always the leftmost bit and is numbered one less than the number of bits in the field (7 for a byte, 31 for a long).

Figure 11-7 shows how binary values of various sizes are represented in the MovieTalk protocol.

Figure 11-7 Binary data representation in the MovieTalk protocol



Bytes within binary data types are arranged similarly to bits. That is, the low-order byte of a multibyte binary field is the rightmost byte and is considered byte 0. The high-order byte is the leftmost byte and is numbered one less than the number of bytes in the field (for example, 3 for a long). Figure 11-7 also shows how the bytes are represented in a long (a 32-bit signed numeric). This is not the way that bytes are arranged in multibyte binary fields on some systems. In order to exchange MovieTalk messages with such systems, bytes within binary values must be "swapped" appropriately so that the target machine can understand the values.

The widths of a number of data types depend upon the architecture of the Macintosh family of computers. The following table lists the data types used in the MovieTalk protocol. Note that many of these data types are named in such a way that you can determine their width from their name.

Data type	Width (in bits)
`Byte`	8 bits, signed
`UInt8`	8 bits, unsigned
`UInt16`	16 bits, unsigned
`UInt32`	32 bits, unsigned
`OSType`	4 bytes
`Int64Bit`	64 bits, signed

Transport Data Types

This section discusses data types that are used by transport components in the messages they send and receive. If you are developing a transport component, you may need to

know about these data types (depending upon the type of transport component you are developing and the protocol you plan to support). If you are using a transport component, you need not know about how these data types are represented; the transport component hides this detail from you.

MTAddress

The `MTAddress` data type defines the format of MovieTalk network addresses. These addresses identify transport-level network endpoints. For example, the OpenConnection message includes the MovieTalk address of the intended recipient. For a more detailed discussion of the MovieTalk address, see the "Transport Components" chapter elsewhere in this book.

```
enum {
   kMTMaxAddressSize =  256,
};

struct MTAddress {
   OSType      transportType;
   OSType      networkType;
   Byte        networkAddress[kMTMaxAddressSize];
};
```

`transportType`
: Type of transport. This is an `OSType` that specifies the type of transport protocol implemented by the transport component. On a Macintosh computer, the value of this field corresponds to the transport component's component subtype.

`networkType`
: Network type. This is an `OSType` that specifies the underlying network protocol. On a Macintosh computer, the value of this field corresponds to the network component's component subtype.

`networkAddress`
: A network-specific address. This is an array of bytes containing an address that is appropriate to the network in use. The size of this array must be `kMTMaxAddressSize`.

MTMessageSize

The `MTMessageSize` data type defines a 32-bit unsigned numeric value. Typically, this data type is used to represent the number of bytes in a MovieTalk message. Every MovieTalk message has a size field in its header.

```
typedef  UInt32         MTMessageSize;
```

MTProtocolVersion

The `MTProtocolVersion` data type defines a 32-bit unsigned numeric value. Typically, this data type is used to represent the version of a component's MovieTalk protocol implementation. Apple assigns version numbers as fixed-point values; the high-order 16 bits represent the whole-number portion of the value; the low-order 16 bits carry the fractional part.

```
typedef  UInt32         MTProtocolVersion;
```

MTStandardID

The `MTStandardID` data type defines a 4-byte alphanumeric value. Typically, this data type is used to contain a value that uniquely identifies a MovieTalk message's type. Every MovieTalk message includes a message type field. This type value determines the format and content of the remainder of the message. Apple assigns message type values that consist of four lowercase characters.

```
typedef  OSType         MTStandardID;
```

MTChunkPriority

The `MTChunkPriority` data type defines a 16-bit unsigned numeric value. Typically, this data type is used to contain a value that specifies the relative priority of a media chunk (that is, a piece of media data that may span one or more messages). See the "Transport Components" chapter, elsewhere in this book, for valid values.

```
typedef  UInt16         MTChunkPriority;
```

MTChunkTime

The `MTChunkTime` data type defines a 64-bit signed numeric value. Typically, this data type is used to contain a value that specifies a media chunk's starting time, in the media's time base.

```
typedef  Int64Bit          CompTimeValue;
typedef  CompTimeValue     MTChunkTime;
```

MTSequenceNum

The `MTSequenceNum` data type defines an 8-bit unsigned numeric value. Typically, this data type is used to contain a value that specifies a packet's sequence number in the context of a stream (allowing the recipient to detect lost or out-of-order stream packets).

```
typedef  UInt8          MTSequenceNum;
```

MTStreamID

The `MTStreamID` data type defines a nonzero 16-bit unsigned numeric value. Typically, this data type is used to identify a stream within the context of a connection.

```
typedef  UInt16         MTStreamID;
```

MTStreamOptions

The `MTStreamOptions` data type defines a 32-bit unsigned numeric value. Typically, this data type is used to specify stream type specific stream-processing options.

```
typedef  UInt32         MTStreamOptions;
```

MTStreamSequence

The `MTStreamSequence` data type defines a 16-bit unsigned numeric value. Typically, this data type is used to contain a value that specifies a packet's sequence number in the context of a media chunk (allowing the recipient to assemble complete media chunks).

```
typedef  UInt16         MTStreamSequence;
```

Stream Director Data Types

This section discusses data types that are used by stream director components in the messages they send and receive. If you are developing a stream director component, you will need to be familiar with these data types. You do not need to know about how these data types are represented in order to use a stream director component.

MTStreamCount

The `MTStreamCount` data type defines a 32-bit unsigned numeric value. Typically, this data type is used to contain a value that specifies the number of streams to be requested in a connection.streams:

```
typedef  UInt32         MTStreamCount;
```

MTStreamReference

The `MTStreamReference` data type defines a 32-bit unsigned numeric value. Typically, this data type is used to specify a stream's persistent reference (this reference is valid for the life of a connection, even if several different sets of streams are created and destroyed in the meantime).

```
typedef  UInt32         MTStreamReference;
```

MTStreamType

The `MTStreamType` data type defines a 4-byte alphanumeric value. Typically, this data type is used to specify the type of data carried by a stream. On a Macintosh computer, this field is set with QuickTime media type values (such as `'vide'` for video and `'soun'` for sound).

```
typedef  OSType         MTStreamType;
```

TimeScale

The `TimeScale` data type defines a 32-bit signed numeric value. Typically, this data type is used to specify a stream's time scale (that is, the number of time units that pass per second in the stream's time coordinate system).

```
typedef  long           TimeScale;
```

Conferencing Data Types

This section discusses data types that are used by conference components in the messages they send and receive. If you plan to develop a conference component, you will need to be familiar with these data types. You need not know about these data types if you are developing an application that uses a conference component.

As is the case with all conference component messages, these data types all consist of ASCII data and represent special ways of organizing and interpreting that ASCII data.

Throughout this section, separator and delimiter characters are represented by the following value names:

Field	Value	Value name
separator	`0x20`	space
separator	`':'`	colon
separator	`'.'`	period
delimiter	`0x09`	tab
delimiter	`0x0D`	newline

Capabilities List

A capabilities list contains information about the messages supported by a conferencing application. The list consists of zero or more lines of information; each line specifies a single capability and consists of the following fields:

Field	Size	Value
type	4 bytes	(alphanumeric)
delimiter	1 byte	tab
version	1–*n* bytes	(numeric)
delimiter	1 byte	tab
desires	1 byte	(alphanumeric)
delimiter	1 byte	newline

type — Conference component message type. This field contains the message's unique type value. Apple assigns values that consist of lowercase alphabetic characters.

version — Message version number. Specifies the most recent version of the message that the application supports.

desires — Support level. This field indicates the level of support required by the application. Possible values are

`mtMessageOptionalCapability`
: Optional. Indicates that the message is optional. The value corresponding to this option is `'O'`.

`mtMessageDesiredCapability`
: Desired. This value indicates that the application expects to receive this message once at the beginning of each conference. The value corresponding to this option is `'D'`.

`mtMessageRequiredCapability`
: Required. The application must receive this message. The value corresponding to this option is `'R'`.

`mtNegotiationMessageCapability`
: Negotiate. The application wants to learn about the recipient's facilities. The value corresponding to this option is `'N'`.

  This option allows an application to query the remote application using facilities offered by the conference component. In this case, the type field contains a component type value. The remote conference component returns a message containing a list of all of the valid component subtypes it finds on the remote system. The returned message has its type value set to the same value that was sent.

Conference Identifier

Conference identifiers uniquely identify a conference endpoint. Each endpoint represents a data source or sink for the conference. Note that a single user may have more than one conference endpoint in a given conference and may have more than one conference active at a time.

A conference endpoint consists of the following fields:

Field	Size	Value
uniqueID	1–*n* bytes	(numeric)
separator	1 byte	space
name	1–*n* bytes	(alphanumeric)

uniqueID
: Unique numeric identifier. This field contains a unique numeric endpoint identifier. Each conference component assigns its own identifiers in order to guarantee uniqueness within the context of a given MovieTalk name.

name
: MovieTalk name. Identifies the system on the MovieTalk network. The name is unique within the context of a given transport interface. See the discussion of the name data type, later in this section, for more information about how MovieTalk names are represented in the conference component protocol.

Member List

A member list is an array of zero or more conference identifiers. Each entry in the array is terminated with a newline character.

Multicast Address

A multicast address specifies a source for broadcast conference data. This address is network independent and unique in the context of a given transport interface.

A conference component protocol multicast address consists of the following fields:

Field	Size	Value
transportType	4 bytes	(alphanumeric)
networkType	4 bytes	(alphanumeric)
separator	1 byte	':'
networkAddress	1–*n* bytes	(alphanumeric)

transportType Type of transport. This is an `OSType` that specifies the type of transport protocol implemented by the transport component. On a Macintosh computer, the value of this field corresponds to the transport component's component subtype.

networkType Network type. This is an `OSType` that specifies the underlying network protocol. On a Macintosh computer, the value of this field corresponds to the network component's component subtype.

networkAddress
A network-specific address. This is an array of bytes containing an address that is appropriate to the network in use. This address may consist of several different address elements, each of which must be numeric and expressable as a 16-bit value. Adjacent elements are separated by periods (`'.'`).

Note
The networkAddress field contains the ASCII representation of a binary network address, expressed as a series of 16-bit values separated by periods. Be sure to convert the entire nonzero address. Pad the address with bytes set to `0x00`. ◆

Name

MovieTalk names identify entities on a MovieTalk network. The name uniquely identifies the entity on the network and is used as part of the conference identifier that specifies a conference endpoint.

MovieTalk names consist of the following fields:

Field	Size	Value
transportType	4 bytes	(alphanumeric)
networkType	4 bytes	(alphanumeric)
address	1–*n* bytes	(alphanumeric)

transportType
: Type of transport. This is an `OSType` that specifies the type of transport protocol implemented by the transport component. On a Macintosh computer, the value of this field corresponds to the transport component's component subtype.

networkType
: Network type. This is an `OSType` that specifies the underlying network protocol. On a Macintosh computer, the value of this field corresponds to the network component's component subtype.

address
: A network-specific address. This is an array of bytes containing an address that is appropriate to the network in use.

  The address value is a C string in the format `"name:service type:address"`. Depending on the network type selected, the address field contains the following information:

Network	networkName field contains
AppleTalk	`"name:type:zone"`
TCP/IP	`"name:socket number:MTIPAddress"`
ISDN	`"name:0:MTISDNAddress"`

Messages

This section discusses the format and content of MovieTalk protocol messages. These messages are exchanged by several of the QuickTime Conferencing components. This section is divided into topics that reflect the type of component that uses the messages, as follows:

- "Transport Component Messages" discusses the messages used by transport components to set up and use media connections between two parties.
- "Stream Director Component Messages" describes the messages that stream director components use to set up channels and streams for media data.
- "Conference Component Messages" provides information about the messages that the conference component uses to set up and manage conferences on behalf of other applications.

Note
Throughout the remainder of this chapter, many messages are presented in the form of C-language structure definitions. These structure definitions are correct for Macintosh computers, and result in message definitions that contain no pad or slack bytes. These structure definitions may not work, unchanged, on other types of computers. ◆

Transport Component Messages

Transport components open, use, and close network connections in order to move QuickTime conferencing data through local and wide area networks. The MovieTalk protocol defines a number of messages that the MovieTalk transport component uses to establish and maintain these network connections. Except for the StreamPayload message, all of these messages flow through the Connection Control Channel. This section discusses those messages. For more information about transport components, see the "Transport Components" chapter, elsewhere in this book.

The OpenConnection message establishes a transport-level connection between two systems. The CloseConnection message closes a connection. StreamPayload messages carry a connection's media data. MovieTalk transport components send KeepAlive messages from time to time in order to verify the continued validity of a connection.

OpenConnection

The OpenConnection message establishes a transport-level media connection between two systems.

```
struct MTOpenConnectionMessage {
   MTStandardID         msgType;
   MTMessageSize        msgSize;
   MTProtocolVersion    protocolVersion;
   MTAddress            address;
};
```

`msgType` The message type. This field must be set to `kMTOpenConnectionMessageType('opnc')`.

`msgSize` The size of the message, in bytes, including the message header. This field must be set to `sizeof(MTOpenConnectionMessage)`, or 276 bytes.

`protocolVersion`
The MovieTalk protocol version. This field indicates the protocol version supported by the sender. All transport components should be backward compatible; that is, transport components should support a given protocol level and all of its predecessors. This chapter describes version 6.0 of the MovieTalk protocol.

`address` The recipient's address. This field contains the MovieTalk address of the intended recipient's media connection. Transport components obtain these addresses from network components.

DESCRIPTION

Transport components use this message to establish a media connection with a remote system. Before a bidirectional media connection exists, the sending component must also receive an OpenConnection message from the receiving system.

For multicast connections, there is no OpenConnection message. New endpoints simply begin accessing the media stream.

Transport components send this message by calling the `MTNetworkSendMessage` network component function.

RESPONSE MESSAGE

None

CloseConnection

The CloseConnection message closes a transport-level media connection between two systems. Either system may close the connection by sending this message at any time.

```
struct MTCloseConnectionMessage {
   MTStandardID         msgType;
   MTMessageSize        msgSize;
};
```

`msgType` The message type. This field must be set to `kMTCloseConnectionMessageType('clsc')`.

`msgSize` The size of the message, in bytes, including the message header. This field must be set to `sizeof(MTCloseConnectionMessage)`, or 8 bytes.

DESCRIPTION

Transport components use this message to close a media connection with a remote system. Closing a connection closes all open channels and streams. Note that this message may be sent or received at any time during a connection's lifetime.

Alternatively, either end may close a point-to-point MovieTalk connection by disconnecting the transport-level control link.

For multicast connections, the CloseConnection message is optional and, if sent, is sent only by the data source.

Transport components send this message by calling the `MTNetworkSendMessage` network component function.

RESPONSE MESSAGE

None

StreamPayload

The StreamPayload message carries media data through a connection. The media data immediately follows the `MTStreamPayloadMessage` structure.

```
struct MTStreamPayloadMessage {
   MTStandardID      msgType;
   MTMessageSize     msgSize;

   MTChunkTime       streamTime;
   MTStreamID        streamID;
   MTChunkPriority   streamPriority;
   MTStreamOptions   streamOptions;
   MTMessageSize     streamChunkSize;
   MTSequenceNum     packetNumber;
   MTSequenceNum     frameNumber;
   MTStreamSequence  streamSequenceNum;
};
```

`msgType` The message type. This field must be set to `kMTStreamPayloadMessageType('strm')`.

`msgSize` The size of the message, in bytes, including the message header and the media data. This field must be set to `sizeof(MTStreamPayloadMessage)`, or 32 bytes, plus the size, in bytes, of the media data itself. The total size of the message must be less than or equal to the transport protocol's maximum packet size.

`streamTime` The time stamp of the media data, expressed in the stream's time scale. This is the time value associated with the media chunk to which the message's media data belong. This time value is included in each message, even if a single chunk spans several messages.

The receiving system uses this time stamp to determine when to play the media data. The stream's time scale is established when the stream is opened (see the discussion of the OpenStream message, later in this chapter).

`streamID` The stream's unique identifier. The stream's ID is established when the stream is opened (see the discussion of the OpenStream message, later in this chapter).

`streamPriority`
The priority of the media data, relative to other streams and to other data in this stream. Setting the chunk priority value correctly allows network elements to intelligently drop data in low-bandwidth situations. The priority value applies to all of the data in the media chunk to which the message's media data belong.

Priority values typically differ for sound and video data (sound is usually higher priority). Similarly, key frame data typically has a higher priority than delta frame data.

See the "Transport Components" chapter for valid chunk priority values.

`streamOptions`
Media-specific control information. This field contains control information that is defined differently for different stream types. MovieTalk transport components set this field from the contents of the `chunkStreamOptions` field of the `MTStreamChunkRecord` structure.

`streamChunkSize`
The size (in bytes) of the media chunk to which this media data belongs.

`packetNumber`
The sequence number of the packet within the stream. This number uniquely identifies each packet sent in a stream, allowing the receiver to detect lost or out-of-order packets between chunks (as distinct from the `streamSequenceNum` field, which identifies a packet within a given chunk). Packet numbers are also useful in situations where a media chunk can be transmitted in a single packet. Packet numbers start at 1 and increment by 1 for each packet sent until the stream is closed.

`frameNumber`
The frame sequence number corresponding to the media data. This field identifies the chunk to which the media data belongs. The value contained in this field is incremented each time an application calls the transport component's `MTTransportSendMediaData` function.

`streamSequenceNum`
The sequence number of a packet within a media chunk. The sequence number allows the receiver to detect lost or out-of-order packets within a given media chunk. Sequence numbers start at 0 and increment by 1 until a chunk is completely transmitted.

The high-order bit of this field is used to indicate the last message in a chunk. If the bit is set to 1, the message is the last one for a chunk. The `kMTStreamSequenceLast` constant allows you to test this bit.

DESCRIPTION

Transport components send all of a connnection's media data using the StreamPayload message. These messages flow through the Media Stream Channel. StreamPayload messages are typically transmitted using unreliable, low-overhead protocols (such as datagram services). As a result, the receiving component must be prepared to skip lost data and to reorder data that is received out of order.

In addition, many control fields that are not necessarily specific to the media data in a given message are sent in each StreamPayload message. For example, the chunk-specific fields in this message are set in each of the messages that contain data from a chunk. This allows the receiver to interpret the data in case one or more of a chunk's messages are lost.

Transport components send this message by calling the `MTNetworkSendMedia` network component function.

RESPONSE MESSAGE

None. Note that StreamPayload messages are typically transmitted over unreliable high speed transports and may therefore be lost.

KeepAlive

Transport components use the KeepAlive message to verify that a connection is still open.

```
struct MTKeepAliveMessage {
   MTStandardID      msgType;
   MTMessageSize     msgSize;
};
```

`msgType` The message type. This field must be set to `kMTKeepAliveMessageType` (`'keep'`).

`msgSize` The size of the message, in bytes, including the message header. This field must be set to `sizeof(MTKeepAliveMessage)`, or 8 bytes.

DESCRIPTION

Transport components send KeepAlive messages to one another in order to maintain an open connection. Each end of the connection regularly sends a KeepAlive message to the other end. If either end of a connection does not receive a single KeepAlive message within an agreed-upon period, the connection is to be considered broken (the connection enters the unexpected disconnect state).

On multicast connections, only the data source sends KeepAlive messages. Receiving endpoints should handle link timeouts by closing their connection.

Transport components send this message by calling the `MTNetworkSendMessage` network component function.

RESPONSE MESSAGE

None

Stream Director Component Messages

Stream director components open and close channels and streams on behalf of conferencing applications. These components use a number of MovieTalk protocol messages to set up and manage channels and streams. All of these messages flow through the Connection Control Channel. For more information about stream director components, see the “Stream Director Components” chapter, elsewhere in this book.

The Connected message allows a stream director to indicate its willingness to accept an open channel request, once a connection has been established. The OpenChannel message opens and closes a channel; the OpenStream message opens a new stream. Stream director components use the Acknowledge message to respond to each OpenStream message.

Connected

Once a media connection has been established, a sink stream director component sends the Connected message periodically until the component receives an OpenChannel message.

```
struct MTConnectedMessage {
   MTStandardID      msgType;
   MTMessageSize     msgSize;
};
```

`msgType` The message type. This field must be set to `kMTConnectedMessageType('sdcn')`.

`msgSize` The size of the message, in bytes, including the message header. This field must be set to `sizeof(MTConnectedMessage)`, or 8 bytes.

DESCRIPTION

The Connected message essentially indicates that the stream director is willing to begin the negotiation process, in anticipation of exchanging media data. The OpenChannel message starts the negotiation process.

Stream director components send this message by calling the `MTTransportSendMessage` transport component function.

RESPONSE MESSAGE

None, although the data sink stream director expects to receive an OpenChannel message eventually. The stream director component continues to resend the Connected message until it receives an OpenChannel message.

OpenChannel

Source stream directors send the OpenChannel message in order to start a stream negotiation process with sink stream directors.

```
struct MTOpenChannelMessage {
   MTStandardID         msgType;
   MTMessageSize        msgSize;
   MTStreamCount        numStreams;
   MTStreamID           streamID[kMTVariableLengthArray];
};
```

`msgType` The message type. This field must be set to `kMTOpenChannelMessageType` (`'ochn'`).

`msgSize` The size of the message, in bytes, including the message header. This field must be set to `sizeof(MTOpenChannelMessage)`, or 12 bytes plus 2 bytes for each stream ID.

`numStreams` The number of streams within a channel. This field specifies the number of streams the source stream director expects to open in this channel. The source stream director later sends one OpenStream message for each stream in the channel.

`streamID` An array of stream IDs. There is one entry in the array for each stream in the channel—the number of streams is specified with the `numStreams` field. The transport component assigns the `streamID` values.

DESCRIPTION

The OpenChannel message starts a stream negotiation process, specifies the number of streams to be established for a channel, and assigns stream identifiers to each stream. Receipt of this message always signals that a remote connection end wishes to establish a new set of streams for transmission. This message may be sent at any time during the life of a connection. When a stream director receives this message, it closes any existing streams before requesting the new streams from the transport component.

Sending an OpenChannel message that specifies zero streams shuts down an existing channel, closing the media data path between two systems.

For multicast connections, the data source periodically sends OpenChannel and OpenStream messages in the media data stream. This allows new recipients to configure their environment properly for the media stream. Ongoing recipients should monitor these control messages in order to determine when they need to renegotiate, based on new stream configurations. For example, recipients detect changes in stream ID values.

Stream director components send this message by calling the `MTTransportSendMessage` transport component function.

RESPONSE MESSAGE

None

OpenStream

Source stream directors send the OpenStream message in order to establish a media data stream with sink stream directors.

```
struct MTOpenStreamMessage {
   MTStandardID         msgType;
   MTMessageSize        msgSize;
   MTStreamType         streamType;
   MTStreamID           streamID;
   MTStreamReference    streamReference;
   TimeScale            streamTimeScale;
// the rest of the message is a SampleDescription
};
```

`msgType` The message type. This field must be set to `kMTOpenStreamMessageType` (`'ostm'`).

`msgSize` The size of the message, in bytes, including the message header. This field must be set to `sizeof(MTOpenStreamMessage)`, or 22 bytes, plus the size of the sample description that follows the `MTOpenStreamMessage` structure.

`streamType` The media type of the stream, such as video (`'vide'`) or sound (`'soun'`).

`streamID` The stream's unique ID. The transport component assigns each stream's ID when it sends the OpenChannel message. Note that the stream ID is unique to both endpoints.

`streamReference` The stream's unique reference number. This value identifies a stream for the life of a connection, in essence providing a logical name for the stream. This value is handy in the event that a channel gets renegotiated. Under such circumstances, you can use the stream reference to relate the new streams to those that existed prior to the renegotiation.

`streamTimeScale` The stream's time scale, expressed in units per second.

DESCRIPTION

The OpenStream message describes the format of each stream in a channel. Besides the control information shown in the `MTOpenStreamMessage` structure, the OpenStream message provides media-specific information. That media-specific information immediately follows the `MTOpenStreamMessage` structure and is contained in a QuickTime sample description structure.

The format and content of the sample description data structure depend on the stream's media type. For sound streams, use a QuickTime sound description structure; for video streams, use a QuickTime image description structure. For more information about these data structures, refer to *Inside Macintosh: QuickTime*.

For multicast connections, the data source periodically sends OpenChannel and OpenStream messages in the media data stream. This allows new recipients to configure their environment properly for the media stream. Ongoing recipients should monitor these control messages in order to determine when they need to renegotiate, based on new stream configurations. Note that multicast recipients do not send an Acknowledge message in response.

Stream director components send this message by calling the `MTTransportSendMessage` transport component function.

RESPONSE MESSAGE

Acknowledge

Acknowledge

Sink stream directors send the Acknowledge message in reply to a point-to-point OpenStream request.

```
struct MTAcknowledgeMessage {
   MTStandardID         msgType;
   MTMessageSize        msgSize;

   MTStandardID         ackMessageType;
   MTMessageSize        ackErrorCode;
   MTMessageSize        ackID;
};
```

`msgType` The message type. This field must be set to `kMTAcknowledgeMessageType` (`'ack!'`).

`msgSize` The size of the message, in bytes, including the message header. This field must be set to `sizeof(MTAcknowledgeMessage)`, or 20 bytes.

`ackMessageType`
The type of the message being acknowledged. Set this field to `kOpenStreamMessageType` to acknowledge an OpenStream message.

`ackErrorCode`
The status of the message. If an error occurred, the sender sets this field to an appropriate error code. If the message was successful, the sender sets this field should to `noErr`.

`ackID` Additional message-specific data. When acknowledging receipt of an OpenStream message, stream director components place the corresponding stream ID value in this field.

DESCRIPTION

The Acknowledge message contains a message's response, including an error code value, if appropriate.

Multicast recipients do not send Acknowledge messages in response to OpenStream messages.

Stream director components send this message by calling the `MTTransportSendMessage` transport component function.

RESPONSE MESSAGE

None

Conference Component Messages

Conference components exchange a number of messages in order to establish, maintain, and terminate conferences. Conference components also send messages that encapsulate user data, that is, the data that is exchanged by the programs that are using the conference. For more information about conference components, see the "Conference Component" chapter, elsewhere in this book.

Conference components send all of their messages by formatting MovieTalk protocol messages and calling the `MTTransportSendMessage` transport component function. All of these messages flow through the Connection Control Channel. Control messages all have a message type value of `'conf'`; user data messages all have a message type value of `'conm'`. The message length field reflects the size of the message, including the header and the conference component or user data.

The following structure defines the message header that precedes each conference component message:

```
struct MTOpenStreamMessage {
   MTStandardID         msgType;
   MTMessageSize        msgSize;
};
```

All conference component control message data is formatted as ASCII text and is null terminated. Fields within messages are variable length and separated by delimiter characters. Depending upon how the message is defined, these delimiters may be either tab (`0x09`) or newline (`0x0D`) characters. Given how these messages are constructed, you might consider building conference component messages using the `sprintf()` standard run-time function.

The format and content of data in user data messages is defined by the conferencing applications.

The conference's media data travels through a separate media channel. Applications establish the media channels using conference component functions intended for that purpose.

The remainder of this section discusses each of the defined conference component control messages. This discussion defines version 3 of the conference component protocol.

Prior to establishing a conference channel with a remote system, a conference member may send either a Capabilities message or an Auxiliary message. The member then sends a Hello message to identify itself, followed by a Call message (to set up a conference) or a Join message (to add an auxiliary media data source). The remote member sends a Response message in answer to the Call or Join message.

Once a conference is established, a member can add other members by sending a Merge message.

Conference members may send or receive Terminate messages to end a conference.

The conference component uses the BroadcastRequest and BroadcastAck messages to negotiate the transition from point-to-point to multipoint connections.

The UserData message allows the conferencing applications to exchange application-specific data over the conference's control channel.

Throughout this section, separator, delimiter, and terminator characters are represented by the following value names:

Field	Value	Value name
separator	`':'`	colon
delimiter	`0x09`	tab
delimiter	`0x0D`	newline
terminator	`0x00`	`NULL`

Capabilities

The Capabilities message allows a potential member to tell other potential conference members what it can and cannot do in a conference. Each member optionally sends this message before sending the Hello message.

Field	Size	Value
type	2 bytes	`'K '`
delimiter	1 byte	newline
capabilitiesList	0–*n* bytes	(alphanumeric)
terminator	1 byte	`NULL`

type The conference message type. This 2-byte field must be set to `'K '`.

capabilitiesList
The member's capabilities. This field is optional. If specified, it contains a list of the member's capabilities. See "Data Types" beginning on page 11-9

for a discussion of the format and content of the capabilities list. An application specifies its capabilities by calling the conference component's `MTConferenceSetMessageCapabilities` function.

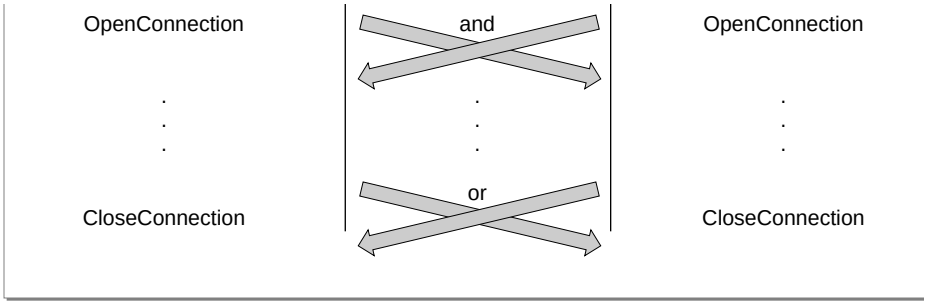
DESCRIPTION

The Capabilities message informs other potential conference participants about a member's capabilities. These capabilities relate directly to messages the member either supports or requires. For example, Apple's conferencing component defines one capability for each conferencing message type that the component supports.

The messages themselves are defined by the type of application the member is running. For example, videophone applications and chatlines deliver different services and use different messages to do so. As a result, the capabilities a member requires will change in light of the application that is participating in the conference.

Figure 11-8 shows a sample capabilities message. In this message, the sending endpoint indicates that it can support version 1 of its `'paus'` and `'shar'` messages, and that it requires that the `'paus'` message be sent.

Figure 11-8 A sample Capabilities message



Entries in the capabilitiesList field may request information from the remote system. By setting an entry's desires field to `mtNegotiationMessageCapability('N')`, a conference member asks for a list of subtypes for a given component type. The type field contains the component type value (for example, `'play'` for a stream player components).

In response, the remote member formats a UserData message, setting the userType field to the component type and putting an array containing the available component subtypes into the userData field. Note that this list may contain duplicate entries and entries with a value of `NULL`. To parse this array, a member must determine the array's size.

After sending a Capabilities message, the member sends a Hello message to establish a conference.

Note that the Capabilities message is optional.

RESPONSE MESSAGE

None

Auxiliary

The Auxiliary message allows one member to alert other members of a conference that it is about to provide an auxiliary media data source. The member must send this message before sending the Hello and Join messages that precede adding an auxiliary data source to the conference.

Field	Size	Value
type	2 bytes	`'A '`
parentConfID	1–*n* bytes	(alphanumeric)
delimiter	1 byte	newline
terminator	1 byte	`NULL`

type The conference message type. This 2-byte field must be set to `'A '`.

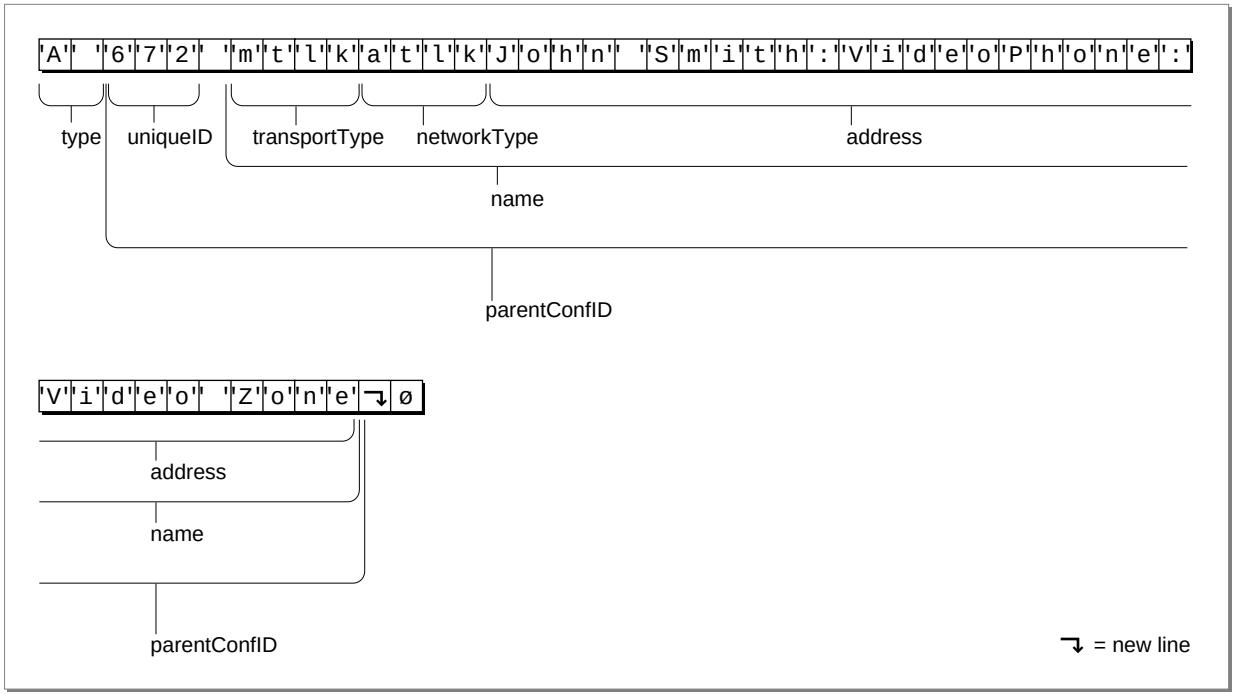
parentConfID The member's conference endpoint identifier. This field identifies the member's existing conference endpoint (the conference identifier the member supplied in the Call message when it first joined the conference). This allows other conference participants to identify the source of the auxiliary data. See "Data Types" beginning on page 11-9 for information about the format and content of conference identifers.

DESCRIPTION

The Auxiliary message informs other conference participants that a member is about to provide an auxiliary conference data source. For auxiliary data sources, this message replaces the Capabilities message during early interactions. The member must send this message to each conference participant. The member then sends a Hello and a Join message to each participant. It is up to those other participants to accept the new data source by accommodating the Join request.

Figure 11-9 shows a sample auxiliary message. With this message, "John Smith" tells the rest of the conference that he is about to supply an auxiliary media data source.

Figure 11-9 A sample Auxiliary message



RESPONSE MESSAGE

None

Hello

Conference components exchange Hello messages at the start of a conference.

Field	Size	Value
type	2 bytes	'H '
minimumVersion	1–*n* bytes	(numeric)
separator	1 byte	colon
maximumVersion	1–*n* bytes	(numeric)
delimiter	1 byte	newline
conferenceMode	1–*n* bytes	(numeric)
delimiter	1 byte	newline
name	1–*n* bytes	(alphanumeric)
delimiter	1 byte	newline
terminator	1 byte	NULL

type The conference message type. This 2-byte field must be set to `'H '`.

minimumVersion
The oldest version of the conference protocol supported by the sending component.

maximumVersion
The newest version of the conference protocol supported by the sending component.

conferenceMode
The desired conference operating mode. This field contains a value that corresponds to the operating mode the sender desires for this conference. Applications specify their desired mode by calling the `MTConferenceSetMode` conference component function.

Conference mode values are discussed in the "Conference Component" chapter, elsewhere in this book.

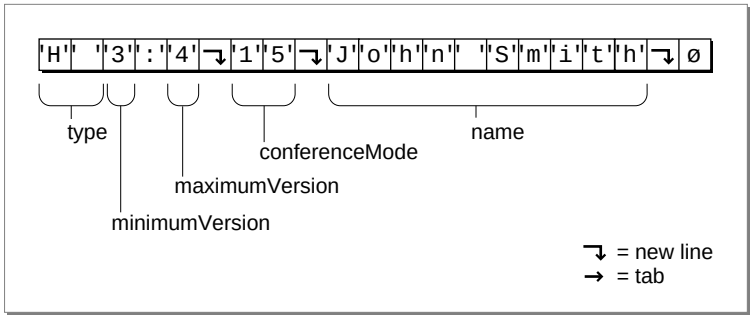
name The name of the prospective conference member. This name identifies the entity that wants to join the conference and may represent either an auxiliary data source or a new user. Applications specify a user name by calling the `MTConferenceListen` conference component function; they specify an auxiliary data source name by calling the `MTConferenceAttachAuxiliarySource` function.

DESCRIPTION

The Hello message is the first step in establishing a conference. This message allows conference components on different systems to exchange basic identification and capability information.

Figure 11-10 contains a sample Hello message. In this message, the sending endpoint indicates that it supports versions 3 and 4 of its protocol. The endpoint proposes opening a conference that is fully media-capable, shareable, and joinable (a mode value of 15). Finally, the endpoint identifies the user as "John Smith."

Figure 11-10 A sample Hello message



Before sending a Hello message, a conference component may send either a Capabilities or an Auxiliary message. The type of message sent depends upon the role the member anticipates playing in the conference. If the member is looking to join or start a conference, the conference component sends a Capabilities message. If the member is setting up an auxiliary media data source, the component sends an Auxiliary message.

Following this message, the conference component can enter the call setup phase by sending the Call message. If the member wants to provide an auxiliary data source to an existing conference, the component sends the Join message.

RESPONSE MESSAGE

None

Call

The Call message begins the process of establishing a conference connection between two possible participants. This is akin to dialing a number from a phone directory.

Field	Size	Value
type	2 bytes	`'C '`
callTimeout	1–*n* bytes	(numeric)
delimiter	1 byte	tab
conferenceName	1–*n* bytes	(alphanumeric)
delimiter	1 byte	newline
callingConfID	1–*n* bytes	(alphanumeric)
delimiter	1 byte	newline
terminator	1 byte	`NULL`

type The conference message type. This 2-byte field must be set to `'C '`.

callTimeout The amount of time the calling component is willing to wait for an answer. This field specifies the number of ticks (60ths of a second) the calling component will wait before deciding that the call has not been answered. Called components must respond within this time period. Applications set a timeout value by calling the `MTConferenceSetCallTimeout` function.

conferenceName The conference name. If the caller is establishing a conference, this is the name the caller has assigned to the conference. If the caller is connecting to a conference server, this is the name of the server's conference. Applications set the conference name by calling the `MTConferenceCall` function.

callingConfID

The caller's unique conference identifier. This field uniquely identifies the caller's conference endpoint on the network. Conference components create conference identifiers on behalf of calling applications. See "Data Types" beginning on page 11-9 for a discussion of the format of conference identifiers.

DESCRIPTION

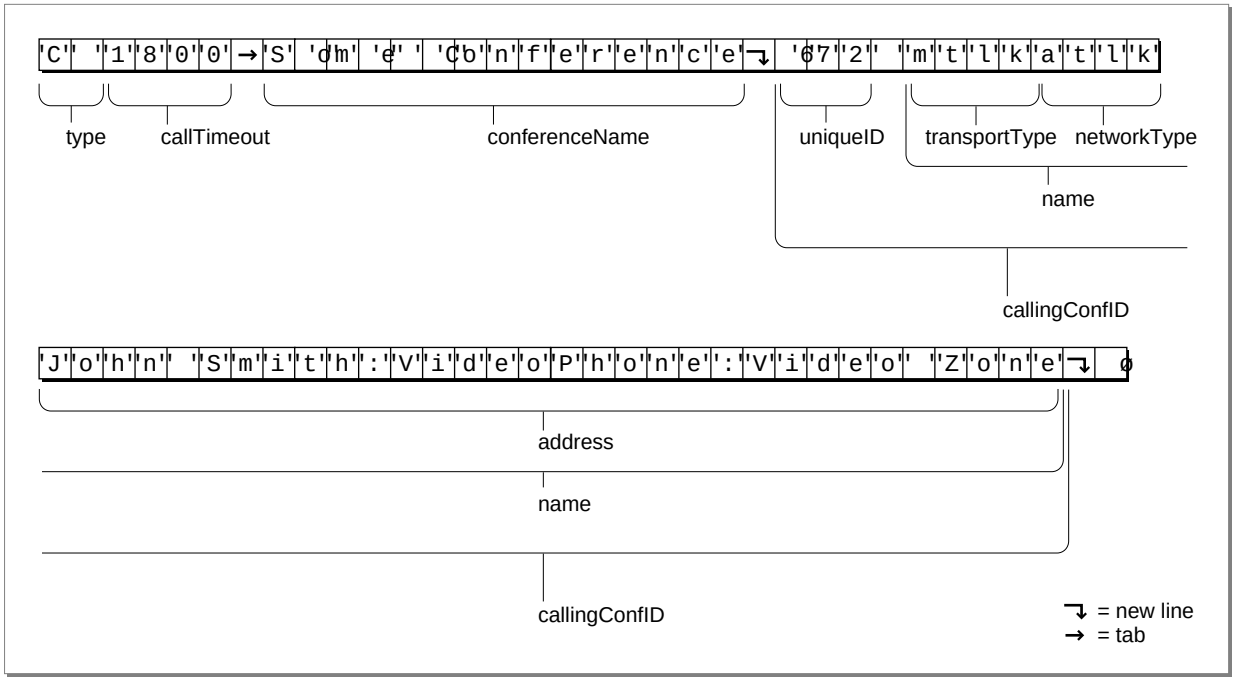
The Call message begins the process of establishing a conference between two participants. This message can be used in two ways. First, this message can create a conference between two participants. In this scenario, the caller assigns a name to the conference, so that other possible participants may join later.

Alternatively, this message can allow the caller to join a conference that is managed by a conference server on a network. In this situation, the caller obtains the server's conference name by some other means.

Once the call is set up, the caller can begin exchanging user data with other conference participants.

Figure 11-11 shows a sample Call message. The sending endpoint employs a call timeout of 30 seconds (1800 ticks) and proposes a conference name of "Some Conference."

Figure 11-11 A sample Call message



RESPONSE MESSAGE

Response

Join

The Join message allows a member to join an existing conference, given that conference's identifier. This message is useful for adding auxiliary data sources to an existing conference and for merging two existing conferences.

Field	Size	Value
type	2 bytes	`'J '`
destinationConfID	1–*n* bytes	(alphanumeric)
delimiter	1 byte	newline
callingConfID	1–*n* bytes	(alphanumeric)
delimiter	1 byte	newline
memberList	0–*n* bytes	(alphanumeric)
terminator	1 byte	`NULL`

type The conference message type. This 2-byte field must be set to `'J '`.

destinationConfID
The remote end's unique conference identifier. This field identifies the recipient. See "Data Types" beginning on page 11-9 for a discussion of the format of conference identifiers.

callingConfID
Unique conference identifier. This field uniquely identifies the caller's conference endpoint on the network. Conference components create conference identifiers on behalf of calling applications. If the message is adding an auxiliary media data source, this is the auxiliary's identifier. See "Data Types" beginning on page 11-9 for a discussion of the format of conference identifiers.

memberList A list of other conference participants. This list identifies all other known conference participants that are willing to exchange data with new participants (that is, they have `mtJoinerModeMask` set to 1 in their conference mode). You as the recipient then connect with each participant with whom you are not already connected.

If the message is adding an auxiliary, this list contains the endpoint identifier of each participant to which the caller is making the auxiliary available. It is up to each of them to respond.

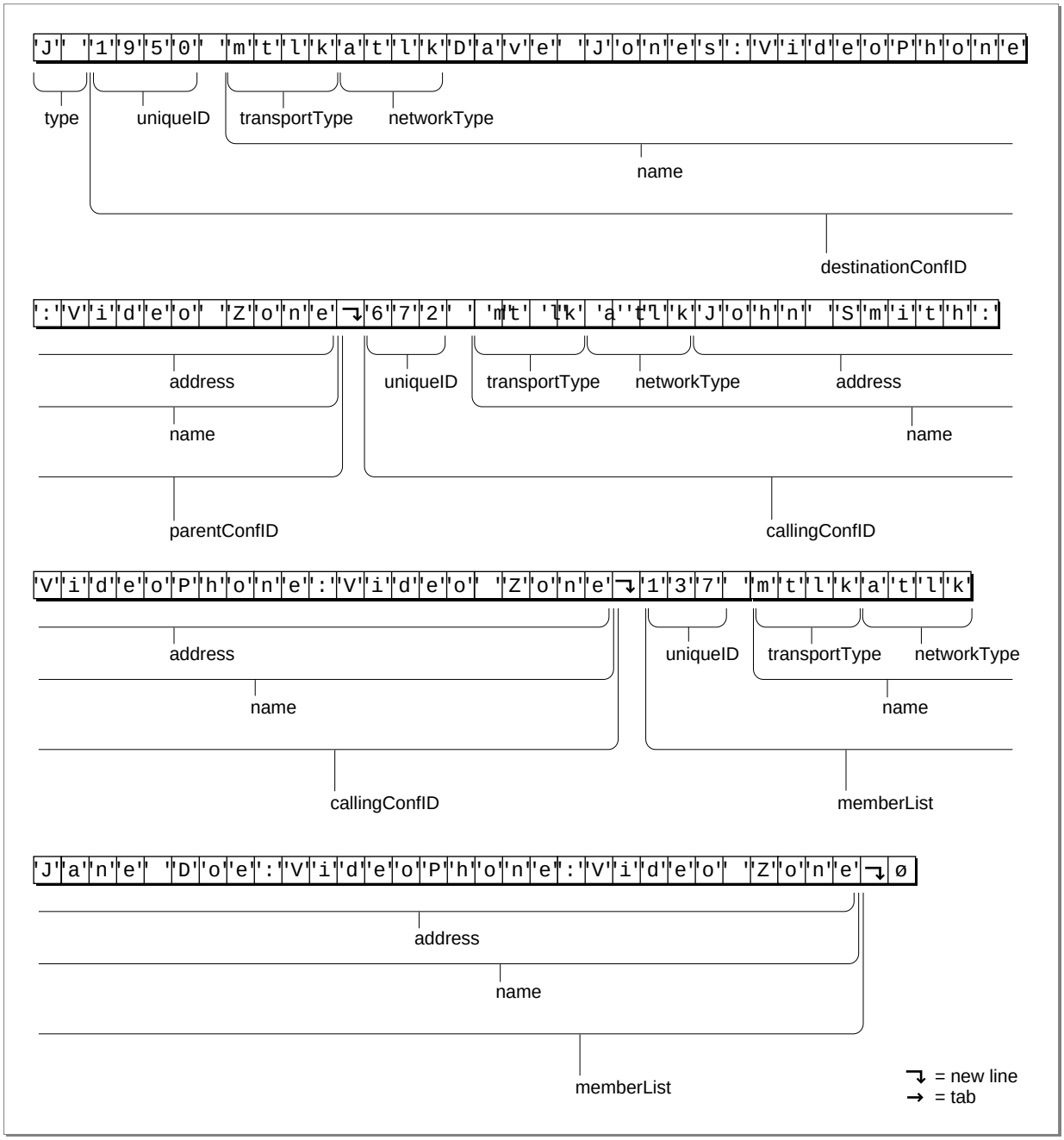
This is a list of conference endpoint identifiers; each element in the list is followed by a newline character.

DESCRIPTION

The Join message allows a member to add an auxiliary data source to an existing conference or to merge two existing conferences. The caller sends this message to each new member of the conference in response to a merge request.

Figure 11-12 shows a sample Join message. With this message, “John Smith” proposes to add “Jane Doe” to his existing conference with “Dave Jones.”

Figure 11-12 A sample Join message



RESPONSE MESSAGE

Response

Merge

The Merge message combines two conferences. Recipients of this message connect with the listed members with whom they are not already connected.

Field	Size	Value
type	2 bytes	`'M '`
conferenceName	1–*n* bytes	(alphanumeric)
delimiter	1 byte	newline
myConfID	1–*n* bytes	(alphanumeric)
delimiter	1 byte	newline
memberList	0–*n* bytes	(alphanumeric)
terminator	1 byte	`NULL`

type The conference message type. This 2-byte field must be set to `'M '`.

conferenceName
The conference name. This is the name that was assigned to the conference when the conference was established. Applications specify the conference name by calling the `MTConferenceCall` function.

myConfID Unique conference identifier. This field uniquely identifies the caller's conference endpoint in the target conference. Conference components create conference identifiers on behalf of calling applications. See "Data Types" beginning on page 11-9 for a discussion of the format of conference identifiers.

memberList A list of other new conference participants. This list identifies other current conference participants. This list contains the endpoint identifier of each participant.

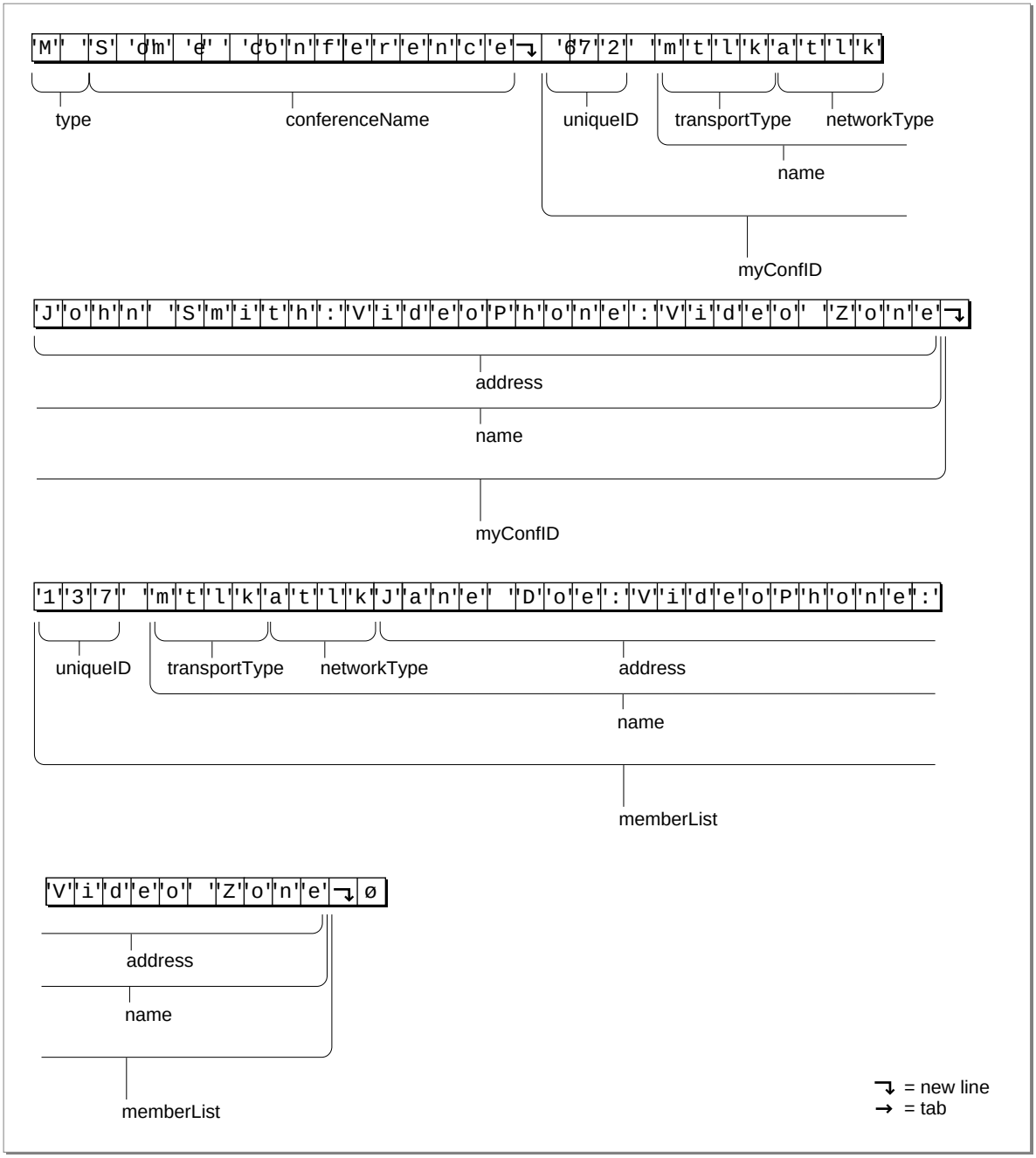
This is a list of conference endpoint identifiers; each element in the list is followed by a newline character.

DESCRIPTION

The Merge message combines two conferences. This is the mechanism for creating conferences with more than two participants.

Figure 11-13 shows a sample Merge message. With this message, "John Smith" adds "Jane Doe" to the conference named "Some Conference."

Figure 11-13 A sample Merge message



The caller sends this message to each existing conference member, specifying the conference's name. Each endpoint establishes communications with any new endpoints. By convention, the endpoint with the higher conference identifier value establishes the

connection (to avoid duplicate or missed connections). Members of the conference receive a Join message instead of a Call message when contacted by each new member.

RESPONSE MESSAGE

None

Response

The Response message is sent in response to a Call or a Join request.

Field	Size	Value
type	2 bytes	`'R '`
callResponse	1–*n* bytes	(signed numeric)
delimiter	1 byte	newline
destinationConfID	1–*n* bytes	(alphanumeric)
delimiter	1 byte	newline
terminator	1 byte	`NULL`

type The conference message type. This 2-byte field must be set to `'R '`.

callResponse The result. This field indicates the result of the preceding Call request. This field is set to `'0'` if the request was successful. Otherwise, it should contain an appropriate result code.

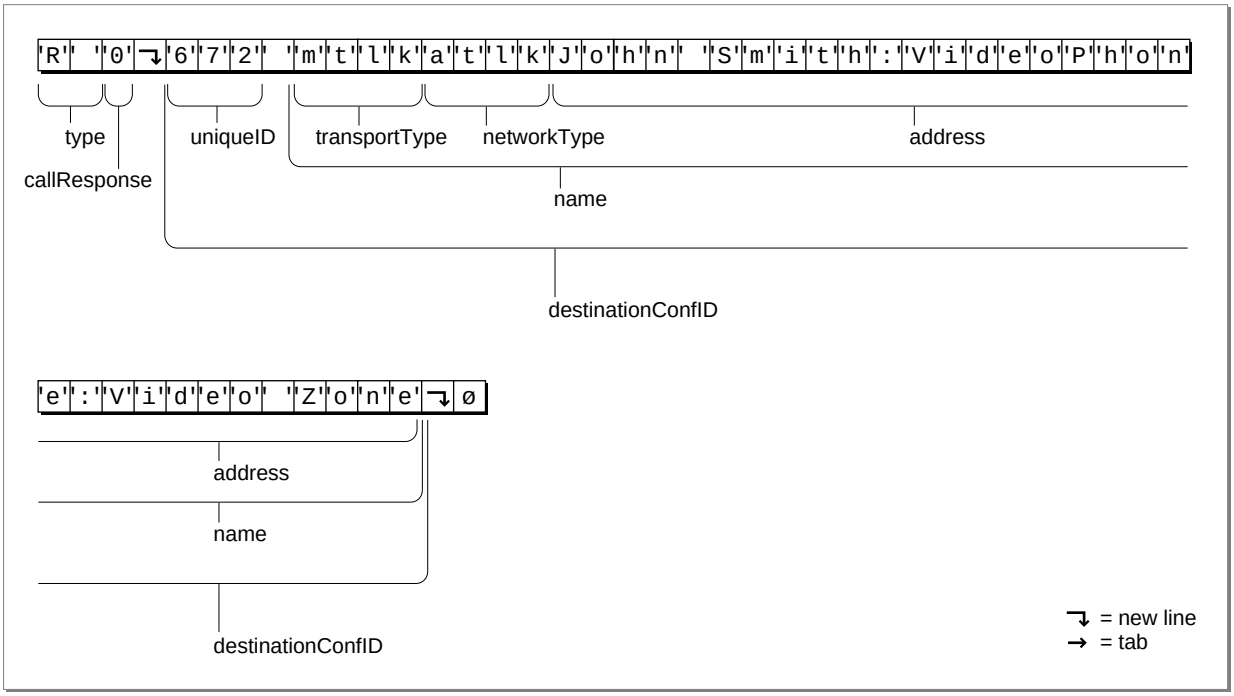
destinationConfID
The other endpoint's unique identifier. This field identifies the other participant in the conference. See "Data Types" beginning on page 11-9 for a discussion of the format of conference identifiers.

DESCRIPTION

The Response message allows the caller to determine the success of a Call or Join request. The Response message indicates how the user at the remote system reacted to the call (for example, whether the user answered the call). The callResponse field contains the request's result code. If nonzero, an error occurred and the request was not successful. Otherwise, the destinationConfID field identifies the endpoint with which the caller may now communicate.

Figure 11-14 shows a sample Response message. In this message, the sending endpoint indicates that the caller may now communicate with "John Smith."

Figure 11-14 A sample Response message



RESPONSE MESSAGE

None

Terminate

The Terminate message ends a conference, closing a member's communications with the participants to which it sends the message.

Field	Size	Value
type	2 bytes	`'T '`
delimiter	1 byte	newline
terminator	1 byte	`NULL`

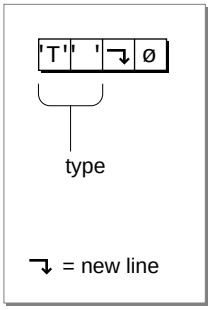
type The conference message type. This 2-byte field must be set to `'T '`.

DESCRIPTION

The Terminate message is the last step in ending a member's participation in a conference. This message ends the member's conference communication with all participants to which it sends the message. If a member is leaving a conference

altogether, it must send this message to each conference participant. Figure 11-15 shows a sample Terminate message.

Figure 11-15 A sample Terminate message



RESPONSE MESSAGE

None

Detach

The Detach message allows conference servers to close the control channel without affecting the user (or media) channel.

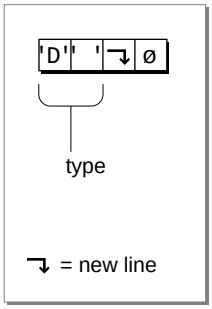
Field	Size	Value
type	2 bytes	`'D '`
delimiter	1 byte	newline
terminator	1 byte	`NULL`

type The conference message type. This 2-byte field must be set to `'D '`.

DESCRIPTION

The Detach message is a special-purpose message that allows a conference server to close its control channel with a participant while leaving the user (media) channel open. This allows the participant to continue to receive media data. This message is especially useful to send-only conference servers; it allows them to conserve processing resources while maintaining user connections. Figure 11-16 shows a sample Detach message.

Figure 11-16 A sample Detach message



RESPONSE MESSAGE

None

BroadcastRequest

The BroadcastRequest message allows a member to find out if another conference member can accept broadcast messages.

Field	Size	Value
type	2 bytes	`'B '`
subtype	1 byte	`'R'`
delimiter	1 byte	tab
multicastAddress	1–*n* bytes	(alphanumeric)
delimiter	1 byte	newline
terminator	1 byte	`NULL`

type The conference message type. This 2-byte field must be set to `'B '`.

subtype The broadcast message subtype. This field must be set to `'R'`.

multicastAddress
The proposed multicast address. This field contains the multicast address on which the member proposes to send broadcast messages. See "Data Types" beginning on page 11-9 for information about the format and content of MovieTalk multicast addresses.

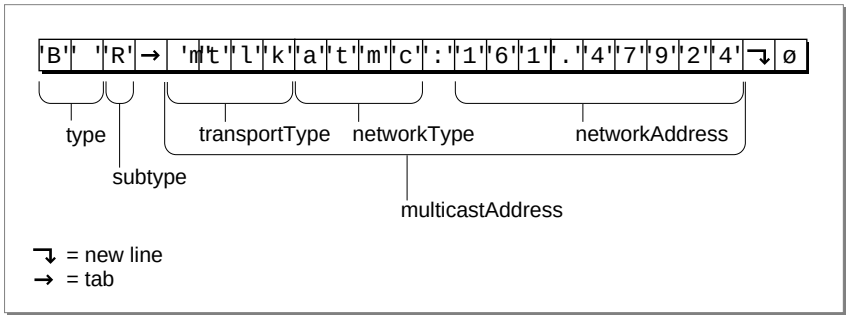
DESCRIPTION

The BroadcastRequest message allows a member to determine whether another conference member can accept broadcast messages over a given multicast network address. The recipient indicates its ability to support the broadcast messages by sending a BroadcastAck message (described next). If the recipient cannot support broadcast

messaging, the calling member should continue to send point-to-point messages to the recipient.

Figure 11-17 shows a sample BroadcastRequest message. With this message, the sending endpoint proposes using the AppleTalk multicast address of `'161.47924'` to send media data.

Figure 11-17 A sample BroadcastRequest message



This message is typically used as part of the negotiation process that follows merging two conferences. Conference participants may choose to set up broadcast capabilities as a more efficient alternative to maintaining several different point-to-point connections.

RESPONSE MESSAGE

BroadcastAck

BroadcastAck

The BroadcastAck message allows a member to respond to a BroadcastRequest message.

Field	Size	Value
type	2 bytes	`'B '`
subtype	1 byte	`'A'`
delimiter	1 byte	tab
broadcastResponse	1–*n* bytes	(numeric)
delimiter	1 byte	newline
terminator	1 byte	`NULL`

type The conference message type. This 2-byte field must be set to `'B '`.

subtype The broadcast message subtype. This field must be set to `'A'`.

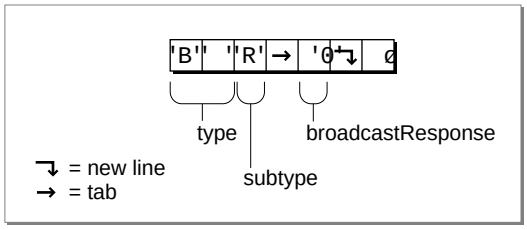
broadcastResponse
: The result. This field indicates whether the member can support the multicast address proposed in the BroadcastRequest message.

DESCRIPTION

The BroadcastAck message allows a member to indicate whether it can receive messages over a proposed multicast address. Another conference participant proposes a multicast address by sending a BroadcastRequest message (discussed earlier in this chapter). If the recipient can support that address, it sets the broadcastResponse field to `'0'`. Otherwise, the broadcastResponse field contains an appropriate nonzero result code.

Figure 11-18 shows a sample BroadcastAck message. This message indicates that the sending endpoint can receive media data using the proposed multicast address.

Figure 11-18 A sample BroadcastAck message



This message is typically used as part of the negotiation process that follows merging two conferences. Conference participants may choose to set up broadcast capabilities as a more efficient alternative to maintaining several different point-to-point connections.

RESPONSE MESSAGE

None

UserData

The UserData message allows a member to send application-specific data through the conference's control channel.

UserData messages all have a message type value of `'conm'`. The message length field reflects the size of the message, including the header and the user data.

The following structure defines the message header that precedes each UserData message:

```
struct MTOpenStreamMessage {
   MTStandardID         msgType;
   MTMessageSize        msgSize;
};
```

Field	Size	Value
userType	4 bytes	(alphanumeric)
userVersion	4 bytes	(numeric)
userData	0–*n* bytes	(alphanumeric)

userType An application-defined type field.

userVersion An application-defined message version field.

userData Application-defined message data.

RESPONSE MESSAGE

None

Utility Components

Contents

This chapter describes the QuickTime Conferencing idler and memory components collectively referred to as *utility components*. The **idler component** provides you with periodic processing time when it is safe to allocate and move memory. The **memory component** provides memory management services optimized for processing streams of media data.

The idler component is typically used by other QuickTime Conferencing components, so if you are developing a QuickTime Conferencing component, you need to know how to use the idler component. If you are developing an application that uses QuickTime Conferencing components, you don't need to know about the idler component.

The memory component also is typically used by other QuickTime Conferencing components, particularly transport components. If you are developing a QuickTime Conferencing component, you should understand the services of the memory component and how to use them.

If you are developing a QuickTime Conferencing application, you need to know that the memory component overrides your application's memory grow-zone function. You should read the section "Registering Multiple Grow-Zone Functions" on page 12-6. Other memory management services provided by the memory component are not relevant to most applications.

This chapter assumes you are familiar with the Component Manager and the Memory Manager and that you have a general understanding of the architecture of QuickTime Conferencing components and media data with which they work. The Component Manager is discussed in *Inside Macintosh: More Macintosh Toolbox*; the Memory Manager is discussed in *Inside Macintosh: Memory*. See the chapter "Introduction to QuickTime Conferencing," earlier in this book, for an overview of the architecture of QuickTime Conferencing components.

About Utility Components

Utility components provide services to all other QuickTime Conferencing components and, in some cases, to applications as well.

The Idler Component

The idler component provides a convenient method by which your component can get periodic processing time when it is safe to move memory. This allows your component to call Memory Manager functions and Toolbox functions that might move memory.

If your component queues data or tasks for later processing or performs some processing not associated with a particular request, then it probably needs the services of the idler component. For example, a stream director component might need to queue an amount of media data before passing it to a stream player component. Or a recorder component may need to buffer data it receives at interrupt time before it can write the data to a disk file.

Components that don't have such requirements do not need to use the idler component. For instance, a flow control component doesn't need to get memory-safe processing time if each time it's called, it gets data to work with, it completes its processing of the data, and it passes the data along. In this case, it doesn't queue any data for future processing.

An idler component has the component type `'cabg'` (defined by the `kMTIdlerType` constant) and the component subtype `'mtlk'` (defined by the constant `kMTMovieTalkSubType`).

If your component does not need to take actions that might move memory, you can use Time Manager functions to get processing time at interrupt time. For information about the Time Manager, see *Inside Macintosh: Processes*.

The Memory Component

The memory component allows you to request memory at interrupt time, and it provides a mechanism for registering multiple grow-zone functions for a single application heap.

A memory component has the component type `'bufl'` (defined by the constant `kMTMemoryComponentType`) and the component subtype `'mtlk'` (defined by the constant `kMTMovieTalkSubType`).

Using Utility Components

This section briefly discusses how you can

- get periodic processing time
- get memory at interrupt time
- set a grow-zone function

Getting Periodic Processing Time

To get memory-safe periodic processing time, you call the `MTIdlerGimmeTime` function, specifying the address of a function to be called periodically, the minimum interval that should elapse between calls to that function, and a reference constant. The idler component thereafter calls your function periodically at a time it is safe to make Memory Manager requests.

For complete description of the `MTIdlerGimmeTime` function, see page 12-8.

Getting Memory at Interrupt Time

The memory component provides interrupt-safe memory management services. It uses the services of the Memory Manager, but gives you the added flexibility of getting and releasing memory at interrupt time.

While the interrupt-safe memory services of the memory component are particularly useful to transport and network components, you can consider using the memory component with any QuickTime Conferencing component that needs to manage memory at interrupt time.

In considering whether to use the memory component in your component, you should look at the pattern of buffer requests your component makes. The memory component is optimized for getting and releasing a series of similarly sized buffers. A transport component, for instance, needs same-size buffers to build chunk record structures as it receives media data. The memory component provides efficient memory management in such situations.

To acquire and release randomly sized buffers, you should use the Memory Manager if possible. The memory component is not a general memory management tool.

The memory component manages a single memory pool in an application's heap. Within a given application, all QuickTime Conferencing components that use the memory component share the available memory in the memory pool. Getting buffers from and releasing them to a common memory pool usually is more efficient than maintaining private buffer pools that limit the use of the memory, and it reduces the likelihood of encountering low-memory conditions.

Before you can get buffers from the memory component, you need to call the `MTMemoryInit` function (page 12-9), like this:

```
myErr = MTMemoryInit (myMemoryInstance, extraBytes);
```

The function allows the memory component to initialize itself. It also requires you to provide a value for the number of extra bytes that the memory component will add to your subsequent requests for memory. For example, if you typically need 1024-byte buffers to hold chunks of media data and you need to add a 22-byte header to each chunk, you should set the `extraBytes` parameter to 22. Then, when you request a buffer, you can request a 1024-byte buffer instead of a 1046-byte buffer. This allows the memory component to perform certain optimizations in managing the memory pool that are not possible otherwise. You could also set the `extraBytes` parameter to 0. In that case, the memory component would return a buffer of exactly the size you specify when you later make a request for memory.

To request memory from the shared pool, you call the `MTMemoryGetOneChunk` function (page 12-10), like this:

```
myBufPtr = MTMemoryGetOneChunk (myMemoryInstance, bufSize,
                isMemorySafeTime);
```

This function first adds the number of extra bytes that you specified to the `MTMemoryInit` function to the value you specify in the `bufSize` parameter to get the total number of bytes for the memory request. Then it attempts to satisfy your request.

If it does not have enough memory in the pool to service your request and you set the `isMemorySafeTime` parameter to `true` to indicate that it's safe to move memory, the memory component attempts to get more memory from the Memory Manager. If you set

the `isMemorySafeTime` parameter to `false`, the memory component is limited to the memory currently available in the shared memory pool.

The memory component returns a pointer to the requested buffer (or `nil` if it cannot satisfy the request). A buffer provided by the memory component is guaranteed to be in physical memory—that is, it is not paged out in virtual memory operations.

To release a buffer, you call the `MTMemoryPutOneChunk` function (page 12-11).

Registering Multiple Grow-Zone Functions

The memory component provides enhanced handling of low memory conditions by allowing multiple grow-zone functions to be called.

The Memory Manager allows an application to set a grow-zone function. When it can't find enough contiguous memory to service a memory allocation request, the Memory Manager calls the grow-zone function, if one has been set. The Memory Manager remembers only one grow-zone function at a time. Setting a second overrides the first.

Under these conditions, only one software entity, either an application or a single component, can try to release memory under low-memory conditions. Other QuickTime Conferencing components that may be able to release buffers at a critical time cannot be tapped.

To maximize the probability of getting memory when it is most needed, the memory component permits the registration of multiple grow-zone functions without automatically overriding one previously registered with the Memory Manager. Thus, the application and any number of components can release nonessential buffers under low-memory conditions.

The memory component itself calls the Memory Manager's `SetGrowZone` function, overriding any set by the application. When the Memory Manager calls the memory component's grow-zone function, the grow-zone function first releases as much memory as it can from the memory component's own memory pool. If still more memory is needed, it then calls grow-zone functions that have been registered with the memory component.

If you are writing a QuickTime Conferencing component or an application that uses a QuickTime Conferencing component, you should register a grow-zone function with the memory component. You do this by calling the `MTMemorySetGrowZoneProc` function, described on page 12-13.

If you need information about grow-zone functions, see *Inside Macintosh: Memory*.

Utility Components Reference

This section describes the constants and functions that comprise the API of the idler and memory components.

Constants

This section describes the constants in the idler and memory component APIs.

Component Types and Subtype

The idler component has the component type `'cabg'`. The memory component has the component type `'bufl'`. Apple defines the component subtype `'mtlk'` for use with the utility components.

```
enum {      /* component types */
   kMTIdlerType   = 'cabg',      /* idler component */
   kMTMemoryType  = 'bufl'       /* memory component */
};

enum {      /* component subtype */
   kMTMovieTalkSubType  = 'mtlk' /* MovieTalk component subtype */
};
```

Request Codes

A request code specifies a function in a component's API. The Component Manager passes the request code to a component to indicate which function was called.

```
enum {      /* idler component request code */
   kMTIdlerGimmeTimeSelect= 1
};

enum {      /* memory component request codes */
   kMTMemoryInitSelect           = 1,
   kMTMemoryGetOneChunkSelect    = 2,
   kMTMemoryPutOneChunkSelect    = 3,
   kMTMemorySetGrowZoneProcSelect= 4
};
```

Functions

This section describes the functions provided by idler and memory components.

You need to open a utility component before you can use its services. You do this by calling the Component Manager function `OpenDefaultComponent`, specifying the component type and subtype that you want to open.

When you are done using a utility component, close it by calling the `CloseComponent` function. The `OpenDefaultComponent` and `CloseComponent` functions are described in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Idler Component Function

You use the function described in this section to request processing time for your component at a time when it is safe to move memory.

MTIdlerGimmeTime

The `MTIdlerGimmeTime` function installs an idler callback function on an idler component.

```
pascal ComponentResult MTIdlerGimmeTime
            (ComponentInstance ic, long interval,
            MTIdlerUPP proc, long refcon);
```

`ic` The idler component instance you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`interval` The minimum interval, in milliseconds, that should elapse between successive calls to your idler function by the idler component.

`proc` A universal procedure pointer to your idler function. If you set this parameter to `nil`, the function removes any previously installed idler function.

`refcon` Reserved for your use. The idler component passes this value when it calls your idler function.

DESCRIPTION

You call the `MTIdlerGimmeTime` function to install an idler callback function on an idler component.

The idler component then calls your function periodically to give processing time to your component. The idler component always calls the function at a time when it is safe to make Memory Manager requests.

The time period you specify in the `interval` parameter is a minimum value. The actual interval that elapses between calls to your idler function can be larger. For example, while a user holds down the mouse button, an idler component cannot call an idler function.

You can change the interval period or the idler function at any time by calling `MTIdlerGimmeTime` again. You cannot have more than one installed function at a given time. If you call `MTIdlerGimmeTime` and specify a different function, the function removes the existing function before installing the new one.

You can remove a previously installed function by setting the `proc` parameter to `nil`. Usually, however, if you no longer want your function to be called, you should just close the idler component.

SPECIAL CONSIDERATIONS

If you want to get processing time at more than one interval, you can create additional instances of the idler component. For each instance of the idler component, you can install an idler callback function with the desired interval.

RESULT CODES

`noErr`	0	No error

SEE ALSO

See page 12-14 for a description of an idler callback function.

Memory Component Functions

You use the functions described in this section to

- initialize your connection to the memory component
- get and release memory
- set a grow-zone function

MTMemoryInit

The `MTMemoryInit` function initializes the memory component and sets the number of bytes the memory component will add to your subsequent requests for memory.

```
pascal ComponentResult MTMemoryInit (ComponentInstance mc,
                                     long extraBytes);
```

`mc` The memory component instance you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`extraBytes` A number of bytes. When you later call the `MTMemoryGetOneChunk` function, it increases the size of every buffer you request by the value you specify here. If you do not want a fixed number of bytes automatically added to your future requests for memory, you can set this parameter to 0. Negative values are not valid.

DESCRIPTION

You call the `MTMemoryInit` function to allow the memory component to initialize itself and to tell it the number of bytes that it should add to your future requests for memory from the shared memory pool.

The value that you specify in the `extraBytes` parameter typically is the size of a fixed header. Each time you request a buffer by calling the `MTMemoryGetOneChunk` function, the memory component automatically adds that number of bytes to the size of the buffer you are requesting. The memory component can manage its memory pool more efficiently if you specify a fixed number of overhead bytes to the `MTMemoryInit` function instead of adding it to each request made with `MTMemoryGetOneChunk`.

You are free to set the `extraBytes` parameter to whatever value you like—there is no need for components requesting memory from the same memory pool to specify the same number of bytes to be added to their subsequent requests for memory.

RESULT CODES

`noErr`	0	No error
`cantOpenHandler`	–2004	Can't open idler component

SEE ALSO

You request buffers from the memory component by calling the `MTMemoryGetOneChunk` function, described next.

MTMemoryGetOneChunk

You call the `MTMemoryGetOneChunk` function to request a memory buffer.

```
pascal Ptr MTMemoryGetOneChunk (ComponentInstance mc,
                long size, short memSafe);
```

`mc`	The memory component instance you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`size`	The number of bytes you are requesting for this buffer.
`memSafe`	A Boolean value indicating whether you are calling this function at a time when it is safe to issue Memory Manager requests. Set this parameter to `true` if you are calling the function when it's safe to move memory. Otherwise, set this parameter to `false`.

DESCRIPTION

Before attempting to service your request, the `MTMemoryGetOneChunk` function adds the number of extra bytes that you specified in the `extraBytes` parameter of the `MTMemoryInit` function to the value you specify in the `size` parameter to get the total number of bytes for this memory request.

When you set the `memSafe` parameter to `true`, the memory component attempts to get additional memory from the Memory Manager if it does not have enough memory in the

pool to service your request. When you set the `memSafe` parameter to `false`, the memory component is limited to the memory currently available in the shared memory pool.

The function returns a pointer to the buffer. If sufficient memory is not currently available, the function returns `nil`.

The memory component holds its memory pool in physical memory—it does not allow it to be paged out by the Virtual Memory Manager's paging mechanism. At interrupt time, you can safely transfer data to or from buffers acquired from the `MTMemoryGetOneChunk` function.

SPECIAL CONSIDERATIONS

`MTMemoryGetOneChunk` is best used to get relatively large amounts of memory, such as that needed for a chunk of media data and in situations where you need a series of similarly sized buffers. You should not use it to get small buffers or randomly sized buffers—it's not efficient for that type of operation.

SEE ALSO

To release memory you get with `MTMemoryGetOneChunk`, you call the `MTMemoryPutOneChunk` function, described next.

For information about the Virtual Memory Manager, see *Inside Macintosh: Memory*.

MTMemoryPutOneChunk

The `MTMemoryPutOneChunk` function releases a buffer you previously allocated with the `MTMemoryGetOneChunk` function.

```
pascal ComponentResult MTMemoryPutOneChunk
            (ComponentInstance mc, Ptr buffer);
```

`mc` The memory component instance you are using. Typically, you obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. You can also get it from the `MTMemoryGetInstance` function.

`buffer` The address of the buffer you want to release. You must have previously allocated the buffer with the `MTMemoryGetOneChunk` function.

DESCRIPTION

You call the `MTMemoryPutOneChunk` function to release a buffer you previously allocated with the `MTMemoryGetOneChunk` function. `MTMemoryPutOneChunk` returns the buffer to the available memory pool.

The memory component automatically releases all buffers after the last memory component instance opened for a given application is closed.

RESULT CODES

`noErr`	0	No error

SEE ALSO

You acquire memory by calling the `MTMemoryGetOneChunk` function, described on page 12-10.

You can obtain the memory component instance you need for the `MTMemoryPutOneChunk` function by calling the `MTMemoryGetInstance` function, described next.

MTMemoryGetInstance

The `MTMemoryGetInstance` function returns the instance of the memory component that allocated a buffer that you specify.

```
pascal ComponentInstance MTMemoryGetInstance (Ptr chunk);
```

`chunk` A pointer to a buffer that you obtained by calling the `MTMemoryGetOneChunk` function.

DESCRIPTION

You call the `MTMemoryGetInstance` function to get the instance of a memory component. You provide a pointer to a buffer that you obtained with the `MTMemoryGetOneChunk` function. The function returns the instance of the memory component that allocated that buffer.

SEE ALSO

The `MTMemoryGetOneChunk` function is described on page 12-10.

You need to provide a memory component instance for a given buffer when you release the buffer with the `MTMemoryPutOneChunk` function, described on page 12-11.

MTMemorySetGrowZoneProc

The `MTMemorySetGrowZoneProc` function registers a grow-zone function with a memory component.

```
pascal ComponentResult MTMemorySetGrowZoneProc
            (ComponentInstance mc, MTMemoryGZUPP proc,
            long refcon);
```

`mc` The memory component instance you are using. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`proc` A universal procedure pointer to your grow-zone function. If you set this parameter to `nil`, it removes a previously registered function.

`refcon` Reserved for your use. A memory component passes this value when it calls your grow-zone function.

DESCRIPTION

You call the `MTMemorySetGrowZoneProc` function to register a grow-zone function with a memory component.

When the memory component cannot find enough contiguous memory to satisfy a memory allocation request from the Memory Manager, it calls the grow-zone functions registered with it until one frees some amount of memory or it has called all of the functions.

RESULT CODES

`noErr`	0	No error

SEE ALSO

For an explanation of why the memory component provides for registering multiple grow-zone functions, see the section "Registering Multiple Grow-Zone Functions" on page 12-6.

See page 12-14 for a description of a grow-zone function.

The Memory Manager's `SetGrowZone` function is described in *Inside Macintosh: Memory*.

Application-Defined Functions

This section describes the callback functions that you provide to an idler and memory component.

The idler callback function is called periodically by an idler component to give your component processing time during which it can move memory.

A memory component calls your grow-zone function under low-memory conditions.

MyIdlerProc

When you call the `MTIdlerGimmeTime` function, you provide a universal procedure pointer to an idler callback function. An idler component calls this function periodically to give your component processing time. It's always called at a time when it's safe to move memory. Here's how you would declare this function if it were a C function named `MyIdlerProc`:

```
pascal void MyIdlerProc (long refcon);
```

`refcon` The reference constant that you provided to the `MTIdlerGimmeTime` function.

DESCRIPTION

An idler component calls your idler function periodically. The minimum interval between two successive calls to `MyIdlerProc` is determined by the value you provided in the `interval` parameter of the `MTIdlerGimmeTime` function. The actual elapsed time is affected by user actions, such as holding down the mouse button or pulling down menus, and it can be much longer.

When an idler component calls your idler function, you can do whatever processing is appropriate to your component, knowing that it is safe to call Memory Manager functions and Toolbox functions that may move memory.

You install your idler function on an idler component by calling the `MTIdlerGimmeTime` function. If you supplied a reference constant to `MTIdlerGimmeTime`, the idler component passes it to your callback function in the `refcon` parameter.

SEE ALSO

The `MTIdlerGimmeTime` function is described on page 12-8.

Universal procedure pointers are described in *Inside Macintosh: PowerPC System Software*.

MyGrowZoneFunc

When you call the `MTMemorySetGrowZoneProc` function, you provide a universal procedure pointer to a grow-zone function. A memory component calls this function

when memory is low. Here's how you would declare the function if it were a C function named `MyGrowZoneFunc`:

```
pascal long MyGrowZoneFunc (long refcon, Size cbNeeded);
```

`refcon` The reference constant that you provided to the `MTMemorySetGrowZoneProc` function.

`cbNeeded` The size, in bytes, of the needed block of memory, including the block header. The grow-zone function should attempt to create a free block of at least this size.

DESCRIPTION

A memory component calls your grow-zone function under low-memory conditions.

If you supplied a reference constant to the `MTMemorySetGrowZoneProc` function, the memory component passes it to your callback function in the `refcon` parameter.

Your function should be a standard Macintosh grow-zone function. It can do whatever is necessary to create free space in the application heap. Your function should return a nonzero value that equals the number of bytes of memory it freed, or zero if it is unable to free any.

SEE ALSO

You install your grow-zone function on a memory component by calling the `MTMemorySetGrowZoneProc` function, described on page 12-13.

For a complete description of a grow-zone function, see *Inside Macintosh: Memory*.

Universal procedure pointers are described in *Inside Macintosh: PowerPC System Software*.

Summary of Utility Components

C Summary

Constants

```
enum {      /* component types */
   kMTIdlerType   = 'cabg',         /* idler component */
   kMTMemoryType  = 'bufl'          /* memory component */
};
```

```
enum {      /* component subtype */
   kMTMovieTalkSubType  = 'mtlk'    /* MovieTalk component subtype */
};

enum {      /* idler component request code */
   kMTIdlerGimmeTimeSelect= 1
};

enum {      /* memory component request codes */
   kMTMemoryInitSelect           = 1,
   kMTMemoryGetOneChunkSelect    = 2,
   kMTMemoryPutOneChunkSelect    = 3,
   kMTMemorySetGrowZoneProcSelect= 4
};
```

Functions

Idler Component Function

```
pascal ComponentResult MTIdlerGimmeTime
                           (ComponentInstance ic, long interval,
                            MTIdlerUPP proc, long refcon);
```

Memory Component Functions

```
pascal ComponentResult MTMemoryInit
                           (ComponentInstance mc, long extraBytes);
pascal Ptr MTMemoryGetOneChunk
                           (ComponentInstance mc,
                            long size, short memSafe);
pascal ComponentResult MTMemoryPutOneChunk
                           (ComponentInstance mc, Ptr buffer);
pascal ComponentInstance MTMemoryGetInstance
                           (Ptr chunk);
pascal ComponentResult MTMemorySetGrowZoneProc
                           (ComponentInstance mc,
                            MTMemoryGZUPP proc, long refcon);
```

Application-Defined Functions

```
pascal void MyIdlerProc     (long refcon);
pascal long MyGrowZoneFunc  (long refcon, Size cbNeeded);
```

Result Codes

`noErr`	0	No error
`cantOpenHandler`	–2004	Can’t open idler component

Glossary

answerer In a QuickTime Conferencing connection, the QuickTime Conferencing application that receives a connection request from the caller.

AppleTalk Datagram Delivery Protocol (DDP) A connectionless AppleTalk protocol that provides best-effort delivery. DDP, which is implemented at the network level, transfers datagrams between sockets over an AppleTalk internetwork with each packet carrying its destination socket address.

AppleTalk Data Stream Protocol (ADSP) A connection-oriented protocol that provides a reliable full-duplex byte-stream service between any two sockets in an AppleTalk internetwork. This protocol appears to its clients to maintain an open pipeline between two entities on an AppleTalk internetwork. Either entity can write a stream of bytes to the pipeline or read data bytes from the pipeline. ADSP is a symmetrical protocol.

AppleTalk Multicast Datagram Protocol (AMDP) A connectionless AppleTalk protocol that provides best effort delivery. AMDP is implemented at the transport level and provides point-to-multipoint delivery of datagrams on an individual AppleTalk network with a single packet for each datagram carrying the multicast address.

AppleTalk network component A network component provided by Apple Computer, Inc. that uses the AppleTalk Data Stream Protocol for control messages and the AppleTalk Datagram Delivery Protocol for media data. It provides network services for QuickTime Conferencing on AppleTalk networks. See also **network component**.

application program interface (API) The set of routines that applications use to access the services provided by system software.

attached components Components that know about each other by reference. No hierarchy is implied with attached components; one component does not open, control, or close the other.

browser component A component that provides the user interface for locating and initiating QuickTime Conferencing connections with other users on a network. The standard browser component provides AppleTalk, PowerTalk, TCP/IP, and ISDN browsing services.

caller The QuickTime Conferencing application that originates a QuickTime Conferencing connection.

call management Originating a QuickTime Conferencing connection, accepting or refusing such a connection, terminating a connection, and other call control functions.

channel (1) Real-time visible and audible data from a QuickTime Conferencing connection as it is presented to a user by a stream controller component. In this context, a channel is similar to a television channel in that it provides sound and images to a user. A channel consists of a group of streams. (2) Part of a QuickTime Conferencing connection. A channel carries either MovieTalk protocol messages or media data. See also **control channel, media channel.**

chunk A logical unit of media data, such as a video frame. Data flowing in a stream across a QuickTime Conferencing connection is organized into chunks. On a storage device, chunks allow optimized data access. A chunk may contain one or more samples of media data. Chunks may have different sizes, and the samples within a chunk may have different sizes. In the Sound Manager, a chunk may refer to a collection of sampled sound and definitions of the characteristics of sampled sound and other relevant details about the sound.

component A piece of code, managed by the Component Manager, that provides a defined set of services to one or more clients. Applications, system extensions, as well as other components can use the services of a component.

component instance A value that identifies a component connection. Each instance of a component can maintain separate storage and error information and manage its A5 world.

conference component The component that provides a high-level API for managing QuickTime Conferencing connections. Applications that use the conference component do not need to interact with lower-level components such as the transport or stream director components. Applications must use the conference component to interoperate with the Apple Media Conference application and other applications that use the conference component.

connection See **QuickTime Conferencing connection**.

connection window From a user's point of view, a window in which one or more parties in a QuickTime Conferencing connection are displayed. See also **sink view, source view**.

control channel That part of a MovieTalk connection that carries messages for setting up, maintaining, and terminating the connection. The control channel typically requires a reliable protocol. For example, on an AppleTalk network, the control channel uses the AppleTalk Data Stream Protocol to assure reliable message transfers.

controller See **stream controller, stream controller component**.

datagram service A type of network delivery in which each packet is sent and received without reference to any other packet. Such a service is often called a "best effort" service because it does not provide error recovery mechanisms when a packet is lost, corrupted, or received out of sequence.

delta frame A video frame that contains only change information—the differences between the current frame and the previous frame. See also **key frame**.

disabled stream A stream whose data is either not sent by the transmitting component or discarded without processing by the receiving component.

enabled stream A stream whose data is sent by the transmitting component or processed appropriately by the receiving component.

flow control component A component that manages the rate of flow of media data between QuickTime Conferencing components while a connection exists. In regulating the rate of flow, a flow control component takes into account such things as CPU performance, local processing load, and network traffic. Apple provides flow control components for sound and video data streams.

frame rate The rate at which visual data is displayed or transmitted, expressed as the number of frames per second.

gain A measure of audio input sensitivity. By adjusting the gain, an application or a user can change the volume of the sound captured by an audio input device, such as a microphone.

H.320 An international standard for video and audio telephony and conferencing established by the International Telegraph and Telephone Consultative Committee (CCITT).

idler component A component that provides to other components a convenient method for getting periodic processing time when it is safe to move memory. A component using the services of the idler component can call Memory Manager functions and Macintosh Toolbox functions that might move memory when it gets its periodic time.

incoming stream A stream whose media data is generated remotely, received from a QuickTime Conferencing connection, and played locally.

Integrated Services Digital Network (ISDN) An international standard for digital telephony. ISDN connections provide one or more data channels operating at 64 kilobits per second.

ISDN network component A network component provided by Apple Computer, Inc. that sends and receives media data and control

information across a reliable ISDN channel. The ISDN network component is used primarily for communicating across wide area networks.

key frame A video frame that contains complete information about the video image—that is, it does not rely on other frames for any of its information. See also **delta frame, temporal compression**.

live connection A QuickTime Conferencing connection in which media data is captured "live" at the sender's side and sent immediately. Live connections are useful for telephony and broadcast applications.

media channel That part of a MovieTalk connection that carries one or more streams of media data. Exchanging data on the media channel requires only a best effort protocol. For example, on an AppleTalk network, the media channel uses the AppleTalk Datagram Delivery Protocol for efficient transfer of media data.

media chunk See **chunk**.

media component A component capable of generating media data, such as a QuickTime sequence grabber component. Stream director components use media components as sources of media data.

media data Time-based information, such as video or sound, that flows across a QuickTime Conferencing connection.

media payload That part of a transport layer packet datagram that carries media data, as distinct from the headers added by a transport component and other network-related headers and trailers.

media stream See **stream**.

memory component A component that allows other components to request memory at interrupt time. It also provides a mechanism for registering multiple grow-zone functions for a single application heap.

MovieTalk address A network-independent address that identifies a network endpoint to the MovieTalk protocol and, thereby, to QuickTime Conferencing components.

MovieTalk message A protocol message sent over the control channel. These messages set up, maintain, and tear down connections.

MovieTalk name A network-independent name used to identify QuickTime Conferencing applications for the purpose of establishing and maintaining connections.

MovieTalk protocol A set of rules for setting up, maintaining, and breaking down a connection and for delivering media data. It consists of a media stream protocol and a connection control channel protocol. The MovieTalk protocol is platform and media independent.

MPEG A standard set by the Motion Picture Experts Group for compressing interleaved video and sound data that is used for the playback of stored data.

multicast connection A connection having two or more stations in which only one station transmits data. Data flows one way from the sender to any number of receiver stations requesting the data. Lecture and broadcast applications use multicast connections.

multicast network component A network component that is optimized for multicast connections.

multipoint connection A connection having two or more stations communicating with each other. Each station has a direct link to every other station in a multipoint connection. Data and control information flows in both directions across those links. Conference telephone calls and collaborative work group applications use multipoint connections.

network component A component that sends and receives media data and control information across a network. It provides a simple interface to either traditional or Open Transport network services.

Open Transport A transport-independent communications architecture that can be used to implement any number of networking and communications systems. Open Transport provides a consistent API across different underlying network protocols.

outgoing stream A stream whose media data is generated locally and sent out over a QuickTime Conferencing connection for playback at a remote connection end.

point-to-point connection A connection having precisely two stations communicating with each other. Data and control information flows in both directions across a point-to-point connection. Private telephone calls and client-server applications use point-to-point connections.

QuickTime Conferencing A set of functions and network protocols, implemented as components, that applications can use to send time-based media data across local and wide area networks.

QuickTime Conferencing connection A connection between QuickTime Conferencing components over which media data is exchanged. The connection consists of two channels: a control channel and a media channel. Transport components manage such connections.

Real-time Transport Protocol (RTP) An Internet protocol for transmitting real-time data. The specification is a product of the Audio/Video Transport working group within the Internet Engineering Task Force.

recorder component A component that stores media data from streams into files for playback by some other means. The Apple-provided recorder component stores media data in QuickTime movie files.

sink view An area within a connection window in which media data arriving from a remote connection end is displayed. See also **connection window, source view**.

source view An area within a connection window in which media data generated at the local connection end is displayed. Typically, it contains a view of the user. See also **connection window, sink view**.

sequence grabber component A QuickTime component that allows applications to obtain digitized data from sources that are external to a Macintosh computer. Your application can request that the sequence grabber store the captured video data in a QuickTime movie. You can also use sequence grabber components to obtain and display data from external sources, without saving the captured data in a movie—for example, when you want to send live video over a QuickTime Conferencing connection.

Simple Multicast Routing Protocol (SMRP) A routing protocol that adds internetwork-wide capability to multicast protocols such as AMDP. SMRP, which is executed by routers, transfers multicast datagrams between networks with each packet carrying the multicast address.

sink stream director A stream director component that manages incoming streams. It uses stream player components to play streams of different types of media data and to manage the stream data's visual and audio characteristics. See also **source stream director, stream director component**.

sound stream player A stream player component that plays back a stream that consists of sound data.

source stream director A component that manages outgoing streams. It uses QuickTime sequence grabber components to capture media data. See also **sink stream director, stream director component**.

stream A sequence of a given type of media data, such as video or sound, carried in a media channel in a QuickTime Conferencing connection. Stream data is organized as a sequence of chunks. A QuickTime Conferencing connection can carry several streams, including more than one stream of a given media type, such as sound or video. A stream is uniquely identified by a stream identifier.

stream chunk See **chunk**.

stream controller A set of controls by which a user manipulates the audio/visual characteristics of a channel. Stream controllers allow users to control the presentation and behavior of QuickTime Conferencing connections. A stream controller typically contains a volume control, a play button, a record button, and so forth. See also **stream controller component**.

stream controller component A component that presents visible and audible media data to a user and supplies user controls for such actions as pausing and resuming the presentation of

media data, making a recording, changing the sound volume and microphone gain levels, and capturing an image. See also **stream controller**.

stream data A sequence of one or more chunks.

stream director component A component that manages any number of either incoming or outgoing streams. It often uses other components to provide services. There are two types: source stream directors and sink stream directors. Both types manage user events and negotiate stream formats with a stream director at the remote end. See also **sink stream director, source stream director**.

stream player component A component that plays media data from a stream that it manages.

TCP/IP network component A network component provided by Apple Computer, Inc. that uses the Transmission Control Protocol (TCP) for control messages and the User Datagram Protocol (UDP) for media data. You can use the TCP/IP network component for communicating across local area networks and the Internet.

temporal compression Image compression that is performed between frames in a sequence. This compression technique takes advantage of redundancy between adjacent frames in a sequence to reduce the amount of data that is required to accurately represent each frame in the sequence.

transport component A component that sets up, maintains, and tears down a QuickTime Conferencing connection across a network. A transport component provides two channels to implement the connection: a control channel and a media channel. While the connection exists, a transport component segments outgoing chunks into packets, reassembles chunks from incoming packets, and uses a network component to send and receive media data and control messages across a network.

video stream player A stream player component that displays a video stream.

Index

A

B

C

D

E

F

G

H

I

J

K

M

N

O

P

Q

R

S

T

U

V

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe Illustrator™ and Adobe Photoshop™.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

LEAD WRITER
Mark Turner

WRITERS
Dee Eduardo, Doug Engfer

EDITOR
Beverly Zegarski

ART DIRECTOR
Bruce Lee

PRODUCTION EDITOR
Gerri Gray

ONLINE PRODUCTION EDITOR
Alex Solinsky

Special thanks to Dean Blackketter, Ramiro Calvo, Lan Chin, Godfrey DiGiorgi, Kevin Gong, Mark Green, Eric Hoffert, Antonio Padial, Murali Ranganathan, and Guy Riddle