Mwave™ System
Application Programmer’s Guide

Version 3.0
for
OS/2 and Windows

**** This Document has been converted to Acrobat format.****
To view document in Acrobat, select view from the tool bar and select “bookmarks and pages” (cntrl + 7). The list of contents will display in left hand column. **Click on the triangle beside each heading to view the subheading.**

NOTE

Before using this information and the product it supports, be sure to read the information under "Notices" on page iii.

Third Edition (July 1995)

This edition is prepared and maintained by IBM Microelectronics. For more information contact:

United States
and Canada
IBM Microelectronics Division
1580 Route 52, Bldg. 504
Hopewell Junction, NY 12533-6531
Tel: (800) IBM-0181 ext. 500

Japan
IBM
800, Ichimiyake,
Yasu-cho, Yasu-gun
Shiga-ken, Japan 520-23
Tel: (81) 775-87-4745
Fax: (81) 775-87-4735

Europe
IBM
La Pompignane BP 1021
34006 Montpellier
France
(33) 6713-5757 (French)
(33) 6713-5756 (Italian)
Fax: (33) 6713-5750*
*from Paris add 16

Europe
Informations Systeme
GmbH
Laatzener Str. 1
30539 Hanover
Germany
(49) 511-516-3444 (English)
(49) 511-516-3555 (German)
Fax: (49) 511-516-3888

Notices

Any reference to an International Business Machines (IBM) licensed program in this document is not intended to state or imply that only IBM's program may be used.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, New York, 10577.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make them available in all countries in which IBM operates.

This document represents a complete rewrite and replacement of previous versions of the Application Programmer's Guide. There are no indications of text or content changes. All previous versions of this document should be considered obsolete.

This document is not intended for production use and is furnished "as is" without any warranties of any kind, and all warranties are hereby disclaimed including the warranties of mechantability and fitness for a particular product.

Trademarks

Some trademarks of IBM and other companies are used in this document. The trademarks used and how they are identified are described here.

The following are trademarks of IBM in the United States, in other countries, or both. These IBM trademarks are identified by an asterisk (*) where they are first used later in this document.

IBM	Mwave

The following are trademarks of companies other than IBM. These trademarks are identified by two asterisks (**) where they are first used later in this document.

Trademark	*Company Owning Trademark*
Microsoft Windows	Microsoft Corporation
MS-Windows	Microsoft Corporation

Table Of Contents

Preface

Before You Begin

This manual describes how to develop OS/2* and Microsoft Windows** 3.1 applications which take advantage of the Multimedia capabilities of Mwave* hardware. It assumes you are familiar with developing applications in 'C' for OS/2 and Microsoft Windows 3.1, and are familiar with the Multimedia services and the Media Control Interface (MCI) provided by these products.

Before attempting to develop an Mwave Multimedia application, install the respective Mwave hardware and software components. In addition, appropriate OS/2 MMPM or Microsoft Windows 3.1 development software must be installed on your system.

Contents of this Manual

This manual is divided into two parts:

- Chapters 2-5 provide a "how-to" introduction to Mwave audio, FAX and TAM applications. Programming examples are used to illustrate important concepts.

- Chapters 6-8 provide a complete reference guide for the Mwave driver Application Programming Interfaces (API), including command messages, data and structure types, and error messages.

Sample Applications

The *Applications Programmer’s Guide* includes example Mwave applications. These example applications, including complete source code, are provided on the companion diskette. They illustrate how to call the Application Programming Interfaces to access the FAX, and Telephone/Answering Machine (TAM) Multimedia capabilities of Mwave compliant hardware. Each application is described in detail in later chapters of this manual.

Mwave audio application services are identical to those described in existing reference material for OS/2 and Windows. For OS/2, this includes the MMPM Application Programming Guide and MMPM Programming Reference. Sample audio applications are included in the Programmer’s Guide and the MMPM/2 Toolkit.

For Windows, this includes the Microsoft Windows Software Development Kit Multimedia Programmer’s Guide and the Microsoft Windows Software Development Kit Multimedia Programmer’s Reference. Sample audio applications are included in the Programmer’s Guide and the SDK.

Related Documentation

This manual describes how to develop applications which take advantage of the Multimedia capabilities of Mwave hardware. The following manuals provide additional information pertaining to the Mwave system and developing Microsoft Windows or OS/2 Multimedia applications.

- The *Mwave Technical Brief* describes the Mwave system, providing an overview of the Mwave processor, Operating System, DSP tasks, Microsoft Windows manager, application drivers, and the DSP development tools.

- The *Microsoft Windows Software Development Kit Multimedia Programmer's Guide* describes how to develop Multimedia applications for Microsoft Windows.

- The *Microsoft Windows Software Development Kit Multimedia Programmer's Reference* provides a summary of the Microsoft Windows Multimedia API, including function and message descriptions, data types and structures, and Multimedia file formats.

- The *Microsoft Windows Software Development Kit Programmer's Reference, Volumes 1-4* describe the complete Microsoft Windows API in detail.

- The *MMPM Application Programming Guide* and the *MMPM Programming Reference* provide information for developing applications in OS/2.

Mwave Developer’s Toolkit

The Mwave Developers Toolkit (MDK) provides a software development environment for programming the Mwave DSP and documentation supporting the development of host device drivers. It provides the following in addition to the material in this Application Programmer’s Guide.

- APIs for Mwave Manager, Mwave External I/O (MEIO) services, and Data Mover Services
- A variety of programming and debugging tools including an Mwave Assembler, debugger, C compiler, linker
- Library support for both C and DSP

The MDK includes the following documentation:

- *Getting Started with the Mwave Developers Toolkit*
- *Application Developer’s Guide*
- *DSP Task Developer’s Guide*
- *DSP Toolkit User’s Guide*
- *Assembly Language Reference Manual*
- *Debugger User’s Guide*

The Mwave Developer’s Toolkit can be purchased from IBM Microelectronics.

Documentation Conventions

Most of the Application Programmer's Guide documentation is applicable to both OS/2 and Windows. In those cases where a difference exists, the text will indicate this explicitly or the OS/2-specific test will be shaded.

Chapter 1 - Introduction

This chapter provides an overview of the Mwave multimedia system and software environment, and briefly describes how to get started integrating Mwave multimedia capabilities into your OS/2 or Microsoft Windows 3.1 application.

Most of the Application Programmer's Guide is applicable to both OS/2 and to Windows. In those cases where a difference exists, the text will say so explicitly or the OS/2-specific text will be shaded.

Mwave System Overview

The Mwave system is a programmable DSP (Digital Signal Processor) based hardware and software platform designed specifically to handle the demands of multimedia in the desktop PC environment. A single Mwave system can integrate a variety of multimedia capabilities such as audio, speech, FAX, modem, and Telephone Answering Machine (TAM).

OS/2 and Microsoft Windows provide high-level and low-level services which enable an application developer to take advantage of the extended capabilities of a multimedia PC. By providing Mwave Application Programming Interfaces (APIs) which are compatible with OS/2 MMPM and Microsoft Windows multimedia services, the application developer is able to develop powerful, portable applications which utilize the multimedia capabilities offered by a wide range of Mwave products.

This manual describes the Mwave APIs and how to use them to develop Mwave multimedia applications for OS/2 and Microsoft Windows 3.1.

Notice

This material is being made available to enable software developers to produce digital signal processing applications, device drivers, and tasks of Mwave™ Technology Platforms. It is not intended to enable others to provide the services of the applications interfaces described herein, rather to enable others to interface to these services.

The following figure illustrates the Mwave runtime software environment:

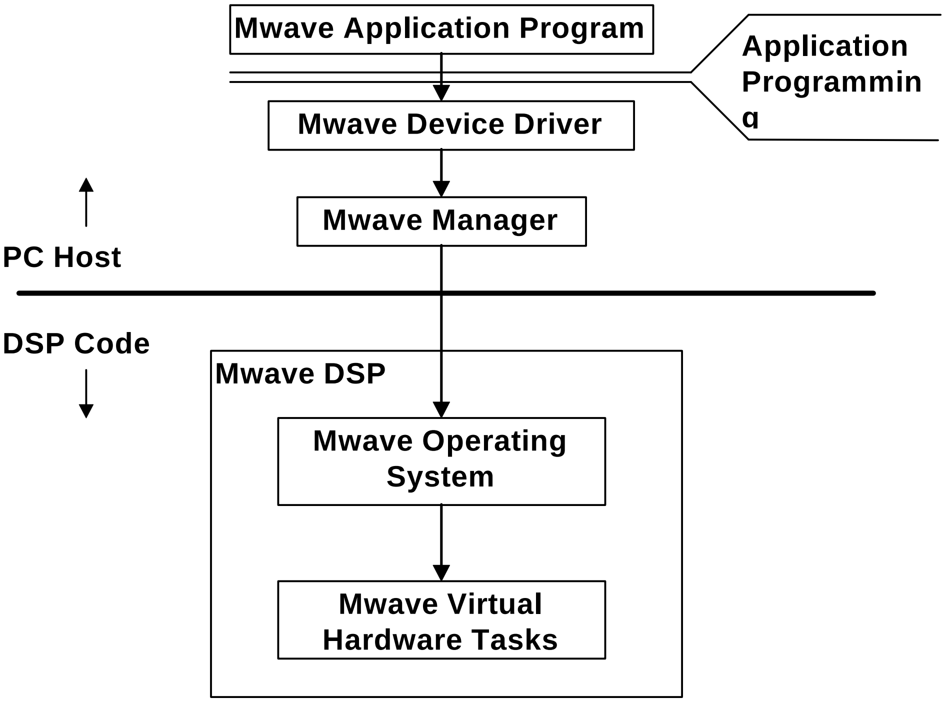

Figure 1-1: Mwave Runtime Software Environment

The following functional blocks comprise the Mwave runtime software environment.

Mwave Application Program

The application program communicates to a device driver through standardized APIs, thus performing a variety of multimedia tasks on the Mwave platform.

Mwave Device Drivers

These MCI compliant drivers provide the software interface between the application program and the Mwave manager, enabling a single Mwave application to run on a variety of Mwave platforms.

Mwave Manager

This host-based software manages DSP resources and provides a hardware-independent interface layer between the Mwave Device Drivers and the underlying Mwave hardware.

Mwave Operating System

Real-time, multitasking DSP kernel that allows concurrent processing of virtual hardware tasks.

Mwave DSP

High-performance DSP optimized for the demands of multimedia applications.

Mwave Virtual Hardware Tasks

DSP-optimized software library that emulates multimedia hardware components such as audio, speech, FAX, and communications.

Developing an Mwave Application

Before writing an Mwave multimedia application, you should have a working knowledge of the following:

- Programming in the Microsoft Windows environment
- Understand the high-level and low-level multimedia services provided in OS/2 Multimedia Presentation Manager/2 (MMPM/2) or Microsoft Windows 3.1.
- Be able to develop programs which use the C-language interface to MMPM’s or Microsoft's Media Control Interface (MCI)

The Multimedia Presentation Manager Toolkit/2 and Microsoft Windows 3.1 Software Development Kit provide documentation and program examples to help you understand these concepts.

If you are planning on adding audio multimedia capabilities to your application, consult Chapter 2, "Audio Services" in this manual for additional information.

If you are developing an application which will utilize the FAX and/or TAM capabilities of the Mwave system, then start out by reviewing Chapter 3, "Telephony Services". This chapter provides an overview of Mwave telephony features and how to access these capabilities from your application program. Next, consult one of the following chapters for specific code examples illustrating how to call the FAX and TAM APIs from your application program:

- For FAX application program examples, see Chapter 4, "FAX Services". The complete FAX API reference is located in Chapter 6, "FAX API Reference".
- For TAM application program examples, see Chapter 5, "TAM Services". The complete TAM API reference is located in Chapter 7, "TAM API Reference".

Chapter 2 - Audio Services

The audio services available to application developers for the purpose of developing Mwave compatible audio applications are identical to those described in the OS/2 publications: *MMPM Application Programming Guide* and *MMPM Programming Reference*, and the Windows publications: *Microsoft Windows Software Development Kit Multimedia Programmer's Guide* and the *Microsoft Windows Software Development Kit Multimedia Programmer's Reference*.

This chapter provides a brief overview of the Mwave Audio Architecture, the Mwave Audio Device Driver, the Mwave Windows Sound System 2.0 audio architecture and implementation description, and the facilities available for developing Mwave audio applications.

Mwave Audio Architecture

Application developers can access the audio capabilities of Mwave compliant audio hardware through the high-level and low-level audio services provided in OS/2 MMPM/2 and Microsoft Windows 3.1.

The host high-level and low-level audio services provide a device-independent software interface, which enables a multimedia application to take advantage of different levels of audio support on a wide range of audio hardware. The Mwave Audio device driver provides the link between the device-independent high-level and low-level audio services of the host PC and the Mwave system software and audio hardware.

Windows Sound System 2.0 Implementation

The Windows Sound System 2.0 API is a standardized, low level, Microsoft developed mixer API that provides applications and other PC based code the ability to gather information, setup, and remain informed about the audio sources, destinations, and controls that exist on a particular hardware platform. It provides developers with a central repository of information and an easy way to get that information in the form of a standardized Windows API. It also allows developers to share a particular piece of hardware in that each registered user is informed of any changes made by any other registered user

The Mwave Sound System audio subsystem was architected to implement the Microsoft Windows Sound System 2.0 API and still retain the flexibility of Mwave, which can dynamically MAKE and BREAK connections, as well as mix digital and analog connections in a single stream to a specific destination. The design point included the following requirements:

- Manage in a single functional module all audio sources that utilize common subsystem components. These components consist of such things as SPEAKER (CDDAC), MICROPHONE (ADC), master and source volumes, etc.
- Allow ISV’s and ourselves to easily add controls and/or source/destinations to suit new hardware and application requirements.
- A desire to separate hardware specific code from non hardware specific code, in order to minimize the amount of code that would have to be rewritten between hardware platforms

NOTE: Not all Mwave subsystems that utilize audio are completely integrated with Sound System. For example, TAM currently uses the SPEAKER and MICROPHONE connections, but is not tied in with the Sound System driver. Therefore, when TAM changes volume, gains, etc., the only user that knows about those changes is TAM. Future releases of Mwave software will include further audio interaction with Sound System.

The following diagram illustrates the relationship between the host PC high-level and low-level audio services, the Mwave Audio device driver, and the Mwave system software and audio hardware:

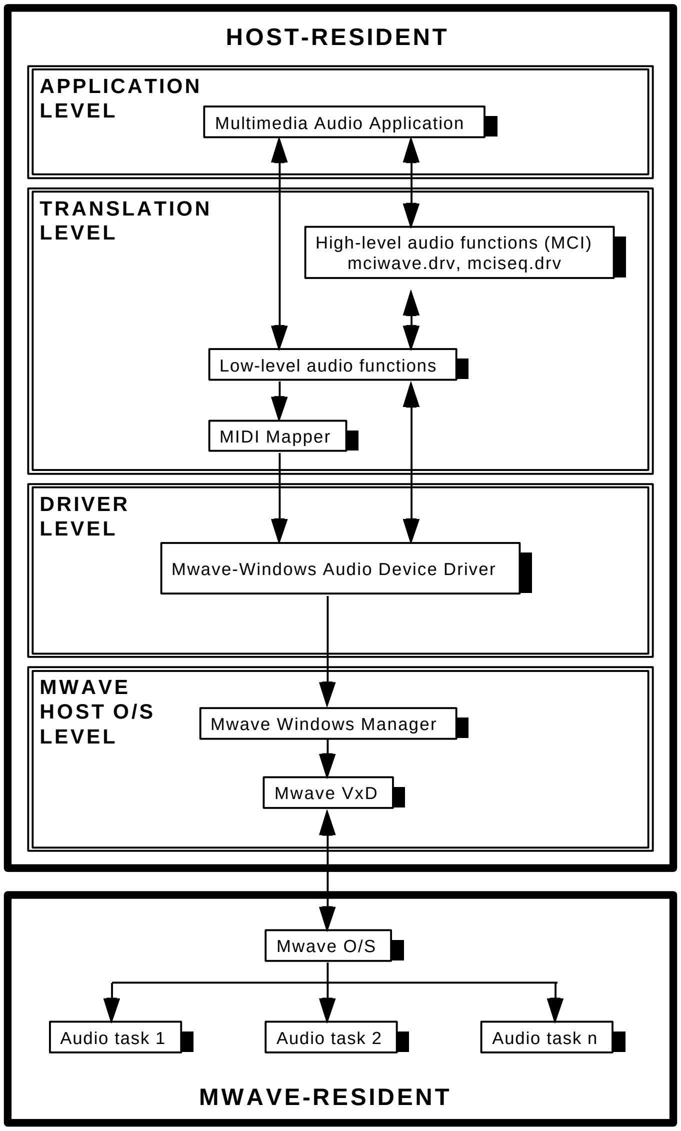

Figure 2-1: Mwave Audio Architecture
(Example for Microsoft Windows)

A multimedia application can access the audio capabilities of Mwave hardware in one of two ways:

Use the high-level audio services, the host PC's Media Control Interface (MCI) provides a high-level command interface to control the audio capabilities of Mwave hardware. As the diagram above illustrates, MCI uses the low-level audio functions to provide high-level audio services to a multimedia audio application.

The Mwave Audio Device Drivers

The Mwave Audio Sound System architecture is architected as shown in the following diagram:

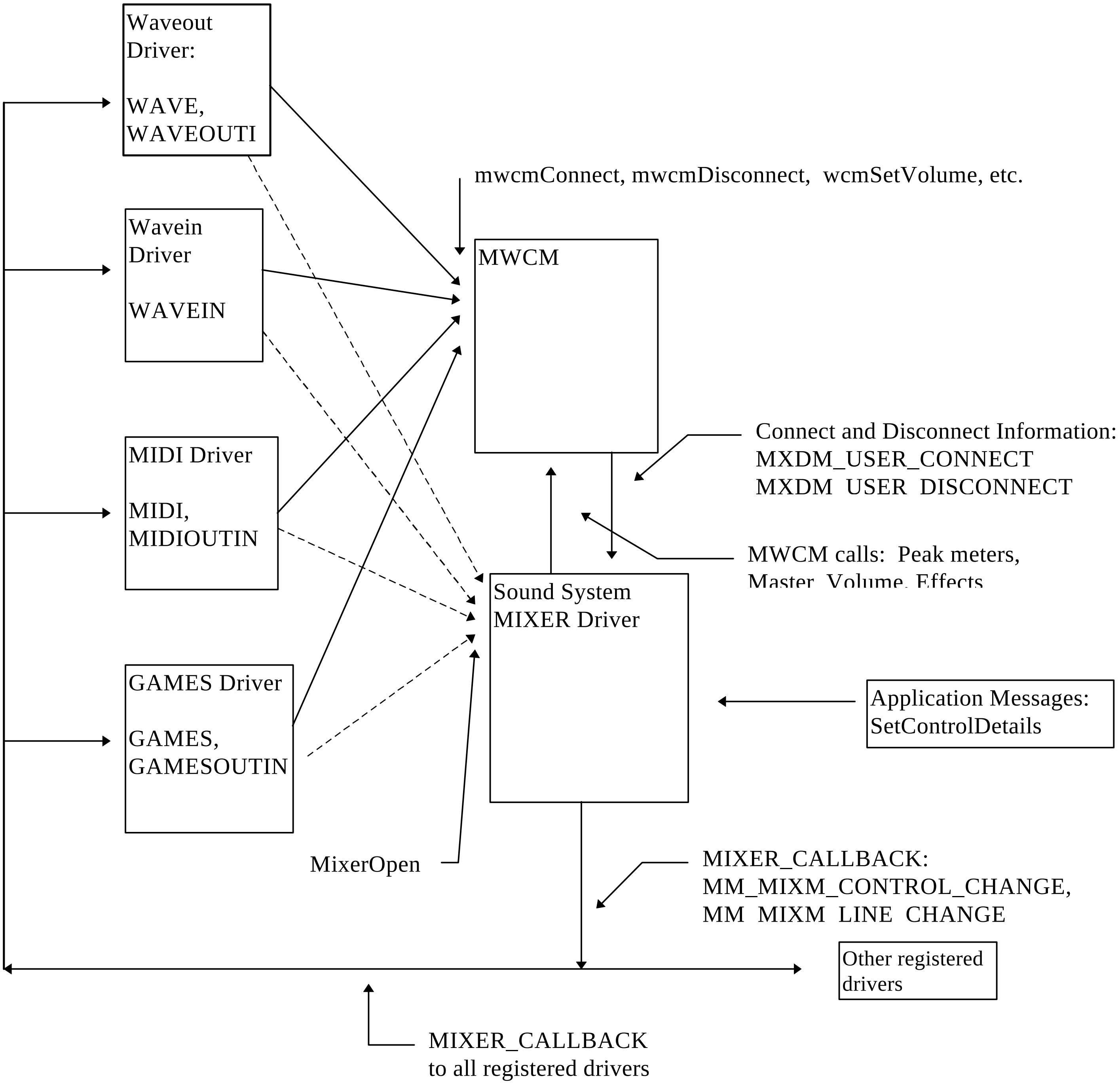

Figure 2-2: Host Drivers Diagram

Component Description

The following paragraphs provide a brief description of each of the components in the audio architecture.

MWCM

MWCM is an acronym for the Mwave Connection Manager.   MWCM provides drivers with an API that allows drivers to connect sources to destinations at driver specified source and destination data rates. MWCM determines, based on the driver specified data rates, what interpolators, decimators, and/or mixers are required for a particular audio connection. If any of these connections already exist on the DSP for a different stream, it will hook into that connection, thereby saving DSP resources. It will load and activate all required tasks, connect up all connections including the connections to the source and destination provided by the driver, and set all volumes on the stream initially to a maximum.

MWCM also provides volume set and get API's for both individual streams (set on the input of the first mixer in the stream attached to the source) and master settings (set on the output of the LAST mixer in a stream before a specified destination).

MWCM provides API's to insert, delete, and control effects in a stream based upon the destination specified. Currently effects destined for the SPEAKER destination are inserted into the last mixer in the stream prior to connecting to the SPEAKER destination. Effects destined for the RECORD destination are placed after the first mixer connected to the analog input source (the 44K mixer connected to MIC, LINE, and/or CD).

MWCM also provides an API to read the peak meters on all of its active streams. This peak meter information is read from the input of the first mixer (the source mixer) attached to the specified source type. There are no MASTER peak meters; a master peak is calculated in the mixer driver by reading the individual peaks and summing their result.

Note: This has the effect of showing a master peak even when the master output is MUTED, because the master peak meter is a sum of all the input sources.

Multimedia Drivers

WAVE, MIDI, AUX, and WAVEIN are the standard Windows 3.1 drivers that interface to the Microsoft multimedia software layer called MMSYSTEM. The GAMES driver is a VDD that emulates the SoundBlaster audio driver.

The following standard MCI drivers, provided in the OS/2 MMPM and Microsoft Windows 3.1, are used for processing high-level MCI calls to Mwave audio hardware:

Device Type	Driver Filename	Description
cdaudio	MCICDA.DRV	An MCI device driver for playing CDDA format files
sequencer	MCISEQ.DRV	An MCI device driver for playing standard MIDI and RIFF MIDI (RMID) files
waveaudio	MCIWAVE.DRV	An MCI device driver for playing and recording waveform audio files

Table 2-1: MCI Drivers

These functions provide a device-independent interface which enable applications to communicate directly with the Mwave Audio device driver.

Low-level audio functions provide additional control over the multimedia device, and as a result, require more programming and are usually more complicated than using high-level services.

Mixer Device Driver

The Mixer Device Driver interfaces to MWCM, to the multimedia drivers, and to the Microsoft Mixer Manager. The Mixer Manager provides applications with a low level interface to/from the mixer device driver.

Mwave Audio Operations

The following paragraphs describe the interaction between the audio components and Sound System applications.

Connect and Disconnect

The interaction starts when a new audio stream is added to the system. This occurs when an application starts to initiate a stream, say for example Media Player starts playing a WAVE file. The app (Media Player) opens the WAVE driver, causing the WAVE driver to load its driver specific DSP code, and make a connection to MWCM. When MWCM receives the connection, it loads and/or modifies existing DSP tasks in order to connect the supplied source GPC and data rate (specified by the driver in the MWCM connection) to the supplied destination GPC and data rate. In the example provided the source would be the output of the WAVE PCM task, and the destination would be the CDDAC BIOS input GPC. Once the connections are made (post connection) MWCM will inform the mixer driver that a new connection was made.

The mixer driver, when it receives connection information, checks the connection information against its “active map”. The active map is an internal data structure in the mixer driver used to describe which sources and destinations are currently ACTIVE (a signal is flowing through them), MUTED, or DISCONNECTED. If a line changes state (in the above example the line would go from DISCONNECTED to ACTIVE) the mixer driver sends a callback to all registered devices indicating this change. Then it returns control to MWCM, who returns control to the multimedia driver. The connection has now been established.

The reverse occurs on a disconnect. On a disconnect, the multimedia application tells the multimedia driver to close. The multimedia driver sends a DISCONNECT message to MWCM. MWCM, prior to unloading any DSP code, sends a DISCONNECT message to the mixer driver (predisconnect). The mixer driver checks the line status of the disconnecting line against its “active map”, changes the status of the source to DISCONNECTED, and, if the destination has no active sources, changes the status of the destination to DISCONNECTED. It then sends a callback to all registered devices indicating a change in line status, and returns control to MWCM, which unloads the stream, and returns control to the multimedia driver. The disconnect sequence is now complete.

Handling Mixer Callbacks

Mwave has a distinct advantage over fixed, hardware only based audio platforms in that the audio streams are mixed digitally on the DSP. It then becomes very useful to provide control over this mix, and allow multiple destinations to receive all possible sources. With the advent of Sound System, control over these streams is relatively simple, and it just becomes necessary to manage stream mixing. This management of stream mixing also uses the Sound System facilities and requires the drivers that are Sound System enabled to follow a few simple rules. When the drivers implement these rules, play destinations can be easily connected to the record destination (and any other destination that the drivers choose), to provide digital mixing of analog (MIC, CD, LINE) and digital streams into the final output (record file).

The rules for drivers to hook into the record destination are as follows.

1. Each driver has a new exported function entry point, in a FIXED code segment (Microsoft requirement) to handle the mixer callbacks. This function must handle the two callbacks: line status changes, and control status changes. It is up to each driver to determine what they will do when they receive these callbacks.

2. Each driver must open the mixer, and register the callback function with the mixer driver on the mixerOpen.

3. Once the mixerOpen has completed, drivers will then receive notice of all line and control status changes that occur.

Control (e.g. Volume Status Change)

For control status changes, drivers are primarily interested in changes to their volumes that occur as a result of some other mixer client changing volume. An example of another client may be a mixer application where a user just adjusted the volume slider. When the slider changes the mixer driver will callback the multimedia driver which can update its own volume parameters accordingly. In addition, if an app sends a volume message to the driver, the driver should make a Sound System call (SetControlDetails) to set the volume on a stream so that other applications will also receive callback notice of the change and can update their own volume data.

Line status changes

For line status changes, drivers can use this information to hook into the record stream if they so choose. For example, if WAVE play is in progress, and the RECORD destination becomes active as the result of someone starting a recording, then WAVE play can make a second MWCM connection to this new destination. The WAVE play stream will now be digitally mixed into the RECORD stream. In order to do this, the conditions described below must be properly handled by the driver:

1. If WAVE is playing and RECORD starts, when the callback occurs, WAVE calls MWCM to connect to the new RECORD destination.

2. If RECORD is recording and WAVE starts, WAVE, as a part of its open sequence, calls MWCM to connect to both the PLAY and RECORD destinations. All line status callbacks are ignored.

3. If WAVE is playing and RECORD stops, when the callback occurs, WAVE disconnects the RECORD destination MWCM connection.

4. If RECORD is recording and WAVE stops, as a part of the WAVE close sequence, WAVE disconnects both of its PLAY and RECORD destinations from MWCM. All line status callbacks are ignored.

Mixer Driver Description

The following discusses the Sound System mixer driver architecture and implementation. This discussion is included to give an overview of both the generic architecture, which can be updated and modified to include new sources, destinations, and controls, and the specific implementation available today for a set of customers based off of the Mwave WHALE DSP reference design.

Architecture

The mixer driver is architected to handle the standard WSS 2.0 messages, and to interface into the existing Mwave multimedia drivers (MIDI, WAVE, AUX, GAMES). It was partitioned to separate hardware specific functions from generic functions, and to allow modification of controls, sources and destinations.

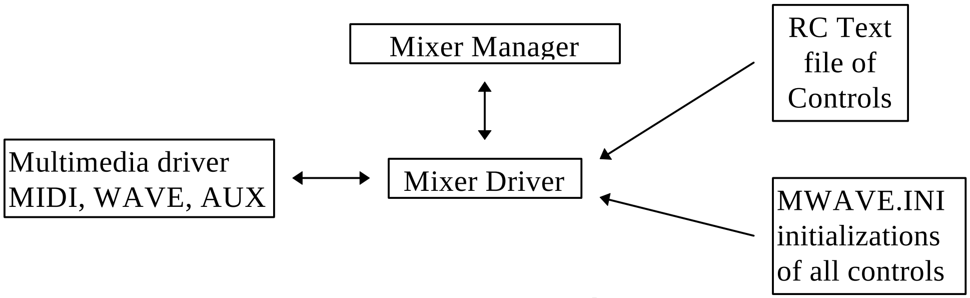
Mixer Manager
RC Text file of Controls
Multimedia driver MIDI, WAVE, AUX
Mixer Driver
MWAVE.INI initializations of all controls

FIGURE 2-3: Mixer Architecture

The controls, sources, and destinations are defined in an RC text file that gets loaded and parsed at Windows start. The initial values of all controls are contained in the MWAVE.INI file, and each control is initialized at Windows start to the value contained in the INI file. These values are written by the driver at Windows exit back into the INI file.

Special note: Because the MSMIXMGR driver loads after MMSYSTEM loads, the mixer controls are not available to the AUX, WAVE, and MIDI drivers at windows start because MMSYSTEM loads the AUX and WAVE drivers and the MSMIXMGR has not yet been initialized. Therefore, the AUX and WAVE drivers must read their initial volumes out of the MWAVE.INI file, they cannot query them from the mixer driver at Windows start. Once Windows is up however, they must get their control values from the mixer driver.

In order to manage controls, the mixer driver has a hardware specific piece of code, CONTROLS.C, which contains all the hardware specific functions necessary to set the controls defined in the RC text file. A table exists in the mixer driver based off of control type that calls each hardware specific function (VOLUME, MUTE, etc.).

NOTE: In the future, this control specific piece of code, along with the table lookup, may exist in a separate DLL in order to facilitate adding or changing controls.

The RC text file is written to be human readable and changeable to allow users to change, add, or delete destinations, sources, and controls. An INI file entry (SSRC=) in the PCMWAVE section of MWAVE.INI tells the mixer driver to read from an RC file or directly from the text file to get its setup information. This means that a user can modify the text file, try it, change it, and when they are satisfied with the results they can recompile the text file in the RC file, and tell the driver to read from RC file again. To read from the RC file, set SSRC equal to 1. To read directly from the text file, set SSRC equal to 0.

It is envisioned that in the future, applications may exist that provide the user with a set of control objects (VOLUME, MUTE, REVERB, etc.), and a set of sources and destinations. Users could design their own mixer driver by visually placing the objects on a template. Once satisfied with the design, they could save the results, which would cause the application to write to the mixer text definition file, which would then be picked up by the mixer driver when Windows is restarted.

The following diagrams describe the current set of controls that have been designed to suit the Mwave hardware reference design for MDSP2780.

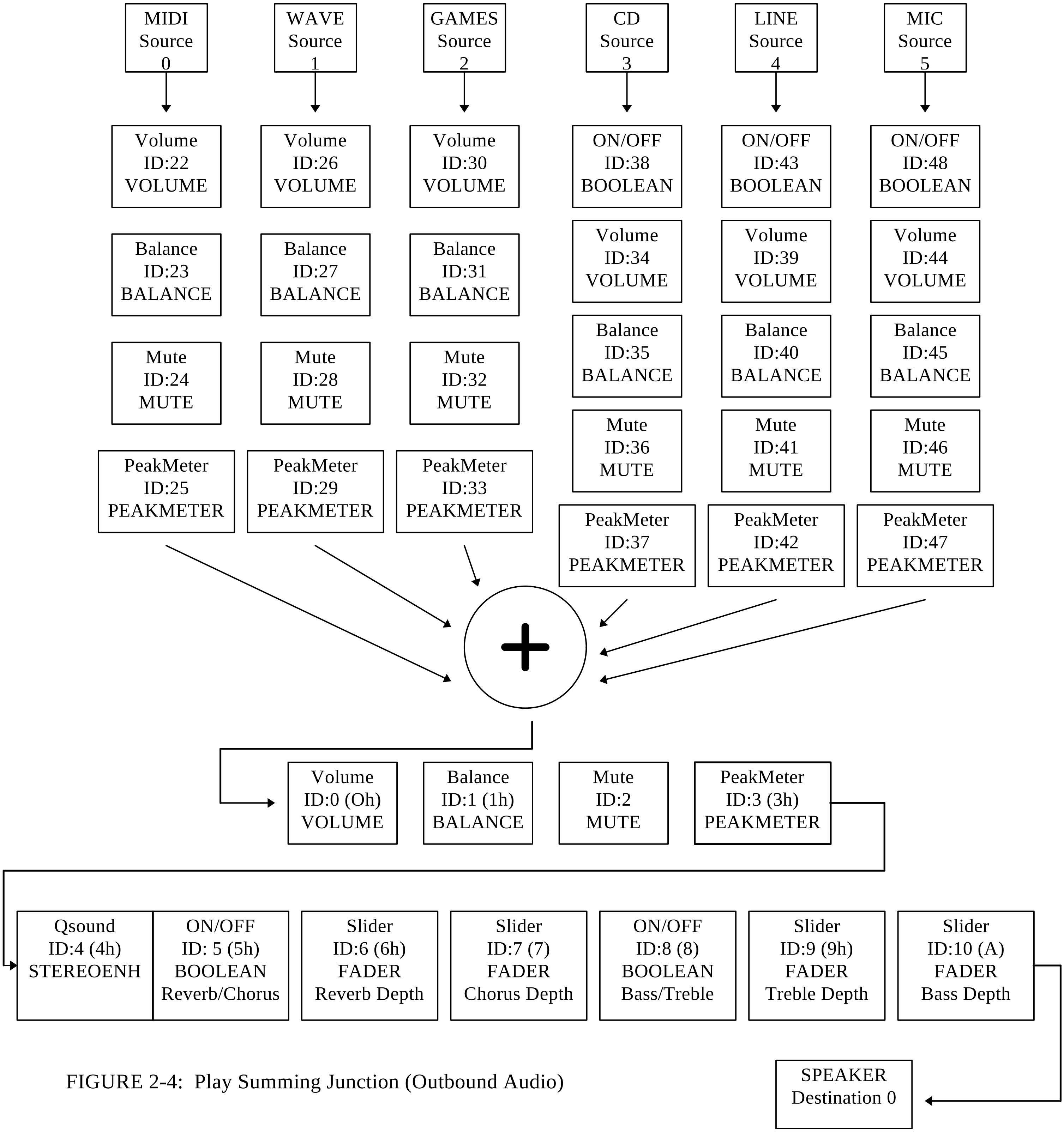

FIGURE 2-4: Play Summing Junction (Outbound Audio)

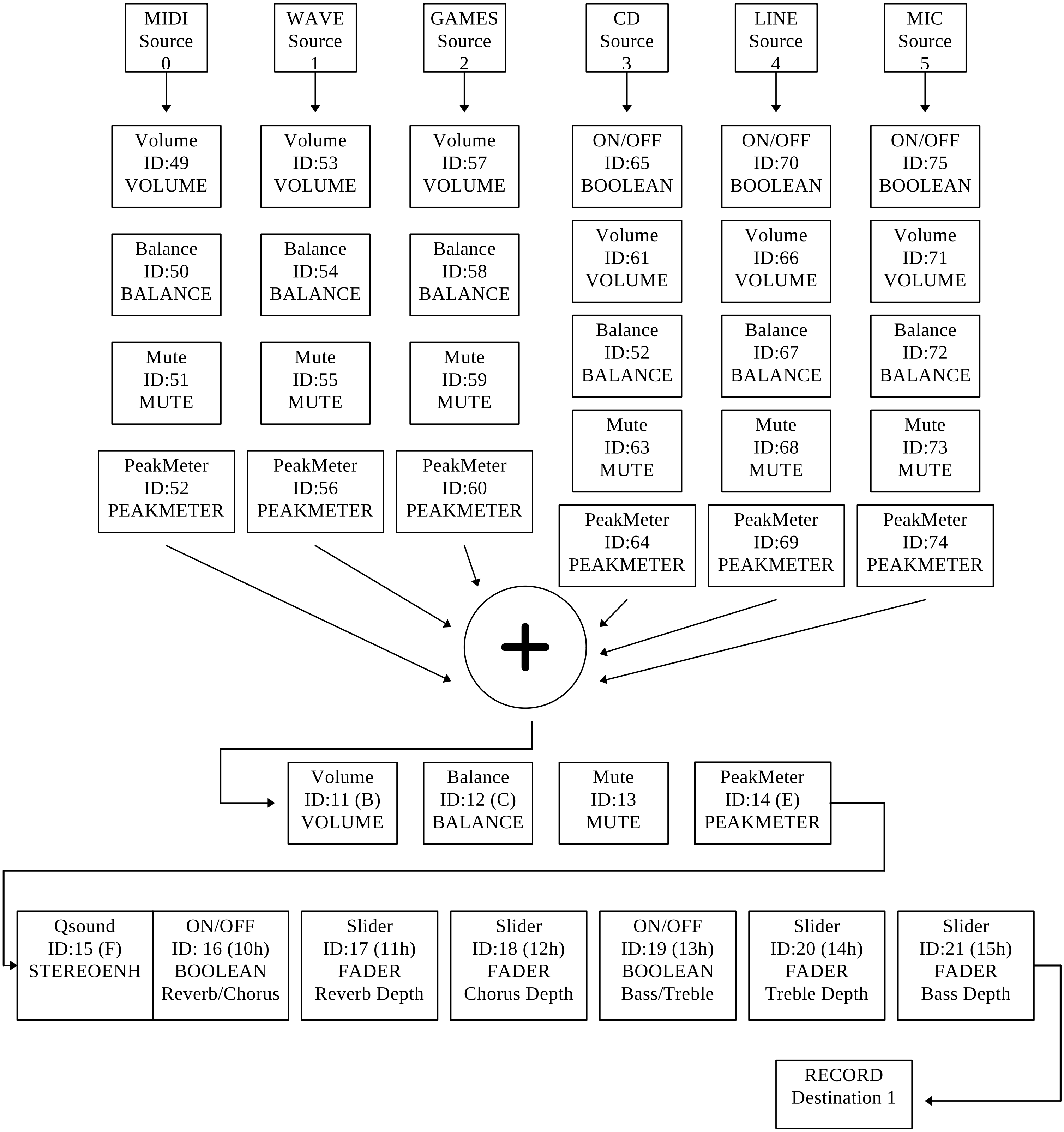

FIGURE 2-5: Record Summing Junction (Inbound Audio)

Developing an Mwave Audio Application

The Mwave Audio device driver is 100% compliant with the low-level command interface of the OS/2 MMPM and Microsoft Windows 3.1 audio device driver specifications. As a result, any OS/2 MMPM or Microsoft Windows 3.1 application which calls functions provided in the high or low level audio services of these systems Windows will operate correctly on Mwave compliant audio hardware.

Mwave audio applications utilize the high-level and low-level audio services provided in OS/2 and Microsoft Windows 3.1.

The following manuals, provided in the Microsoft Windows 3.1 Software Development Kit, describe these services in detail, and also explain how to use these services to add multimedia audio capabilities to your Microsoft Windows 3.1 application:

- The *Microsoft Windows Software Development Kit Multimedia Programmer's Guide* describes how to develop Multimedia applications for Microsoft Windows 3.1. Chapters 2-5 describe the programming interface and audio services provided in Microsoft Windows 3.1.

- The *Microsoft Windows Software Development Kit Multimedia Programmer's Reference* provides a summary of the Microsoft Windows Multimedia API, including function and message descriptions, data types and structures, and Multimedia file formats.

The following manuals, provided in the Multimedia Presentation Manager Toolkit/2 , describe these services in detail, and also explain how to use these services to add multimedia audio capabilities to your OS/2 application:

- The MMPM *Application Programming Guide* describes how to develop Multimedia applications for OS/2. Chapters 2-5 describe the programming interface and audio services provided in OS/2.

- The MMPM *Programming Reference* provides a summary of the MMPM API, including function and message descriptions, data types and structures, and multimedia file formats.

All audio capabilities described in the above documentation are available to Mwave audio application developers.

Audio Mixer API Reference

The following describes each of the Windows Sound System 2.0 messages and the Mwave Sound System driver's implementation of each of the messages provided by the Sound System API.

MXDM_OPEN

The calling client is added to the mixer driver's list of registered clients. The calling client will be notified, via callback, of changes in controls or line status.

Note that Microsoft does not require a client to open the mixer driver in order to use the mixer driver. A client can access the mixer driver independent of the open/close message. The only service that open and close provides is the ability of the client to receive a callback on any change to control or line status.

MXDM_CLOSE

The calling client is removed from the mixer driver's list of registered clients. The calling client will no longer be notified, via callback, of changes in controls or line status.

MXDM_GETDEVCAPS

Returns the Mwave mixer device driver capabilities copied into the passed in MIXERCAPS structure. Currently the returned values are as follows:

wMid:	MM_MICROSOFT;
wPid:	MM_MSFT_WSS_MIXER;
vDriverVersion:	0x200
fdwSupport:	NULL
cDestinations:	2
szPname:	"Mwave Mixer Audio Driver"

MXDM_GETNUMDEVS

Returns 1, only 1 mixer device supported.

MXDM_GETLINEINFO

Returns information about a specific source or destination. For queries of TARGETTYPE, valid targettypes are WAVEOUT, WAVEIN, and MIDIOUT.

MXDM_GETLINECONTROLS

Returns information about a specific set of controls. This driver only supports the standard three queries: ALL, ONEBYID, and ONEBYTYPE.

MXDM_GETCONTROLDETAILS

Returns the current setting(s) of a specific control.

MXDM_SETCONTROLDETAILS

Sets the control to the specified state, updates the driver's internal tables, and notifies all registered users (those that OPENED the mixer driver via MXD_OPEN) of the change in the control. The mixer driver retains this information while Windows is running in its own internal data structures, and when Windows is shut down in INI file entries in the PCMWAVE section of the MWAVE.INI file.

MXDM_USER_CONNECT

Notification from MWCM (Mwave Connection Manager) of the connection of a line. The following parameters are expected in this call:

dwUser:	Specific instance data (none used)
dwParam1:	MWCM Connection name
dwParam2:	MWCM connection handle, type HMWCM.

MXDM_USER_DISCONNECT

Notification from MWCM (Mwave Connection Manager) of the disconnection of a line. The following parameters are expected in this call:

dwUser:	Specific instance data (none used)
dwParam1:	MWCM Connection name
dwParam2:	MWCM connection handle, type HMWCM.

Mixer Callbacks API Reference

Callbacks are passed to a function or a window handle based on the type of callback specified in the clients call to mixerOpen. Note that only those devices that have opened the mixer driver will receive callbacks. The callback messages notify clients of changes in line status (ACTIVE, MUTED, DISCONNECTED), and changes in control values. Source lines are ACTIVE when they have data flowing through them (which, on Mwave, is when they have a MWCM connection), and destination lines are ACTIVE when any source line is ACTIVE.

Callbacks are the mechanism used by Sound System enabled drivers (MIDI, WAVE, etc.) to implement Record what you play, which allows users to record both digital and analog input streams, and to update and modify global volume parameters.

MM_MIXM_LINE_CHANGE

This callback occurs when a line is connected or disconnected from MWCM, or changes state (MUTE, UNMUTE).

MM_MIXM_CONTROL_CHANGE

This callback occurs when a control changes state.

Chapter 3 - Telephony Services

This chapter describes the telephony services available to OS/2 and Microsoft Windows 3.1 application developers for the purpose of developing Mwave compatible telephony based applications.

Mwave Telephony Architecture

The Mwave system hardware has the ability to play and record data to and from a telephone line, an external microphone/speaker, a telephone handset (just the ear piece and microphone), or a telephone deskset (standard analog telephone). For the purposes of this document, no distinction is made between a telephone handset and a telephone deskset. The term "handset" refers to either one. The data obtained from these devices can be in a variety of formats (voice data, fax data, etc.) as supported by corresponding DSP code tasks. Along with the ability to record and play telephony media, the system (with the required software tasks) has the ability to decode touch-tone type key presses coming either from the telephone line or handset.

Building a complete application from a library of various DSP functionality would be tedious at best. For this reason, the DSP tasks have been grouped into categories of applications, and have been integrated with software device drivers. The types of telephone formats addressed by the current device drivers include voice and fax carrier data transmission. In future software releases, data transmission may be integrated with voice, e.g. voice and data.

The following figure shows the Mwave telephony architecture:

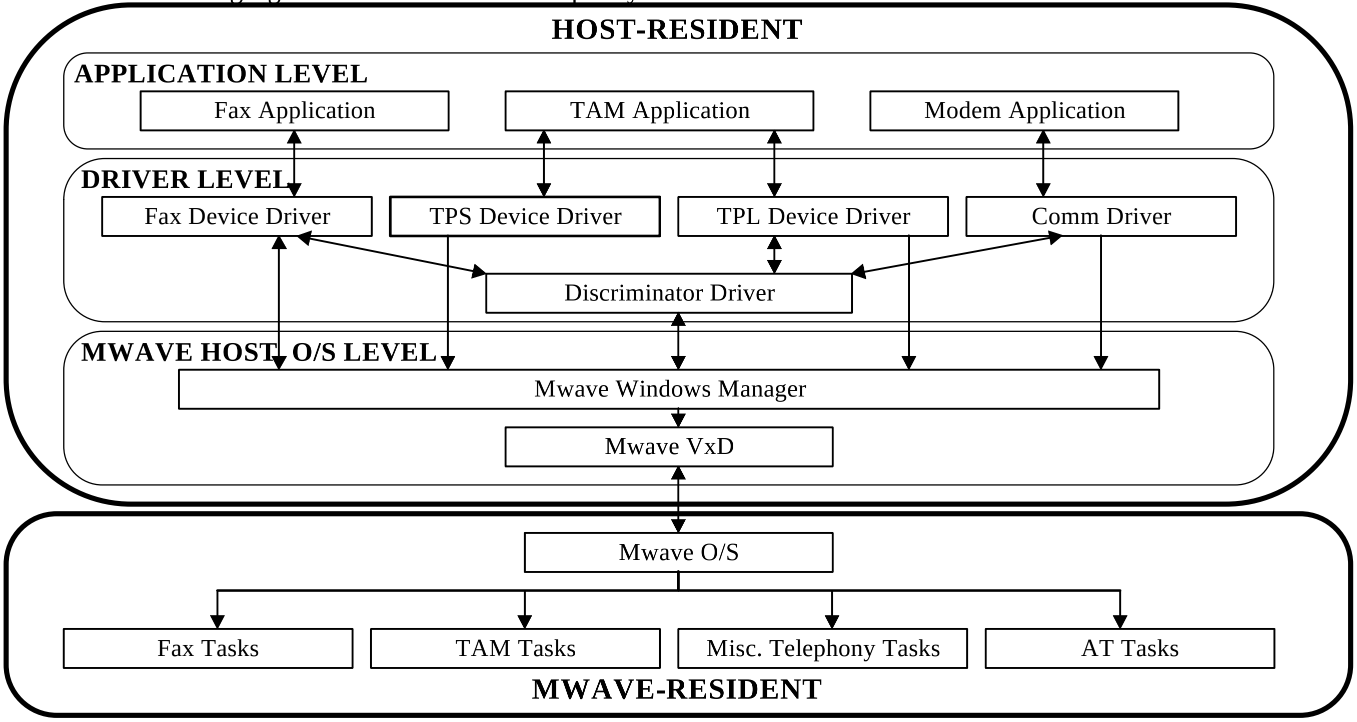

Figure 3-1: Mwave Telephony Architecture

The following standard MCI drivers, provided in the OS/2 MMPM and Microsoft Windows 3.1, are used for processing high-level MCI calls to Mwave telephony hardware:

Device Type	Driver Filename	Description
fax	mcifax.drv fax.dll (OS/2)	An MCI device driver for sending and receiving faxes
tps	mcimsg.drv tps.dll (OS/2)	An MCI device driver for local message record and playback
tpl	mciphone.drv tpl.dll (OS/2)	An MCI device driver for message record and play through the phone line only

Table 3-1: Telephony MCI Driver Summary

The Telephony Device Drivers
Two types of telephony drivers are supplied with the Mwave subsystem. These include fax send and receive (FAX) and telephone answering machine (TAM). They are designed to comply with standard MCI type command protocols, and are implemented as MCI extensions.

Common Telephone Interface

Because both drivers make use of the telephone device, potential conflicts arise when multiple applications are active at once. Although the telephone can be physically used by only one type of driver at a time, the Mwave telephony drivers were designed with the ability to share control of the telephone device, and in essence, to virtualize the telephone line. This virtualization process has been integrated into the MCI telephone interface used by both drivers, and is performed transparently to the application.

Using a common telephone interface has some significant advantages. The common interface presents a standardized view of the telephone device to the application programmer. This allows the programmer to create a personal library of telephony functions, and use them in a variety of applications. Migrating from a FAX device to a TAM device does not require any retraining on programming the telephone.

Most importantly in today's multi-tasking environment, having a common telephone interface allows for transparent implementation of a virtual telephone device. Although multiple applications cannot use the telephone simultaneously, they can constantly monitor for incoming calls. The job of sharing the telephone device, discriminating between calls, and signaling the corresponding waiting application is performed transparently by the device driver. This allows the application programmer to treat the telephone device as a sharable resource, and does not require inter-communication between separate FAX, TAM, and Modem applications. An application can wait for a call, and know that it will gain control of the telephone when the call arrives. An application that owns the telephone knows that it can complete the call without fear that another application will try to "steal" the telephone line. The common telephone interface greatly simplifies any environment where the telephone line is shared.

Multiple applications can monitor for an incoming telephone call at any time. However, only one application may monitor for any one particular type of call. For example, two applications can not be simultaneously waiting for a voice call, but one application can wait for a voice call, and another application can wait for a fax or modem call. Only one application can own the telephone line at any given time. Ownership of the telephone is the ability to actually use the telephone line (make a call).

An application takes ownership of the phone line in three ways:

- The application receives an incoming phone call.
- The application executes a command to take the phone off-hook.
- The application executes a dial command.

Once an application takes ownership of the phone line, other applications can continue to wait for incoming telephone calls, or wait for the phone line to free up, but these other applications are not allowed to use the telephone device. If an application is just waiting for a call, active use of the telephone by other applications is transparent to the waiting application. Any application can still execute MCI commands to control the driver environment, even when the telephone is in use by another application. The telephone line is owned by an application until it places the telephone on-hook, or until the device driver detects that the call has been completed.

MCI Event Handler

One difficulty in using a telephone device is the random nature of telephone events. At any one time, one or more applications might be waiting for a call, determining a data transmission baud rate, checking to see if the handset is on-hook, and looking for touch tone key presses from either the handset or the telephone line. Obviously, it is impractical to constantly poll for these types of events, especially in a non-real-time environment such as Microsoft Windows. Ideally, an application would be notified (via messages) when any of these randomized real-time events occur. Under MCI for OS/2 and Microsoft Windows, an application notification message, **MM_MCINOTIFY,** is used to notify an application when a function call has been completed, but unfortunately, no mechanism exists to signal an application when a defined event occurs.

To handle the need for an on-demand messaging system, the MCI drivers for Fax and TAM include a message posting system, which when combined with an application supplied message handler, can signal telephony applications when a defined event occurs. The receiving application might or might not act on this message. Some of the message events defined include:

- Receiving a telephone call
- Detecting call termination
- Incoming caller identification string
- Handset hook status
- Handset touch-tone key press
- Telephone line hook status
- Telephone line touch-tone key press
- Telephone ring detected

There are additional messages defined for the Fax and TAM drivers which notify an application about more application specific events. The Microsoft Windows message chosen to signal these events is **MM_MCIEVENT**, because of the function's similarity to its *OS/2* counterpart. "Initializing the Application" on page 1-24 describes the **MM_MCIEVENT** message in detail.

By using the supplied event handler, a telephony application need not poll the status of any of the important telephony peripherals.

Developing an Mwave Telephony Application

The Mwave telephony device drivers are compliant with the guide-lines of the MCI command interface of MMPM and Microsoft Windows. The drivers implement all required MCI commands, and include additional MCI extensions to provide a simple yet comprehensive interface to the telephone device. Information specific to the functionality of the individual drivers is available in their corresponding chapters.

Before developing an Mwave telephony application, you need to become familiar with the procedures involved in using the MCI interface. This document assumes you are familiar with MCI. The following manuals, provided in the Microsoft Windows *3.1* Software Development Kit, describe MCI command execution in detail, and also explain the various commands and messages involved in writing an MCI-based Microsoft Windows application:

- The *Microsoft Windows Software Development Kit Multimedia Programmer's Guide* contains an excellent overview of MCI, and provides code examples detailing the use of the MCI interface.

- The *Microsoft Windows Software Development Kit Multimedia Programmer's Reference* provides a summary of the Microsoft Windows Multimedia API, including function and message descriptions, data types and structures, and Multimedia file formats.

The following manuals, provided in the Multimedia Presentation Manager Toolkit/2, describe this information for OS/2.

- The *MMPM/2 Sample Application Programming Guide* (S71G-2221*)* contains an excellent overview of MCI, and provides code examples detailing the use of the MCI interface.

- The *MMPM/2 Programming Reference* (S71G-2222*)* provides a summary of the MMPM API, including function and message descriptions, data types and structures, and multimedia file formats.

This section illustrates the use of the common telephone interface, which is integrated into every telephony device. The information supplied here applies to all of the supplied telephony device drivers, although the code examples have references to specific drivers. More information on programming each individual device driver is supplied in a separate chapter.

Initializing the Application

The most significant difference between traditional MCI devices and the telephony drivers supplied here is the use of an event handing routine. Communication of real-time status information from the MCI device to the application is performed through this application event handler. The handler should be able to service messages posted by the MCI device, which contain real-time status information about the device. The message, MM_MCIEVENT, is not a standard MCI message under Microsoft Windows. Thus, a Microsoft Windows application must call the **RegisterWindowMessage** function with the string "MM_MCIEVENT" to obtain the numeric value of the notification message.

MM_MCIEVENT

In addition to the message itself, *wParam* and *lParam* are used to pass information to the application.

WPARAM *wParam*
Contains a device specific event message ***wEvent***.

LPMCI_EVENT_PARMS *lParam*

Specifies a far pointer to the following MCI_EVENT_PARMS structure:

```
typedef struct {
   DWORD dwDataParam1;
   DWORD dwEventData;
} MCI_EVENT_PARMS;
```

The data parameters are defined as follows:

DWORD *dwDataParam1*

The low-order word specifies the device specific event message ***wEvent*** (same as *wParam*). The high-order word specifies the device ID of the device initiating the message.

DWORD *dwEventData*

Contains a data parameter, which is dependent on the message type. This parameter is usually an on / off indication, or a pointer to string data.

The message type contained in ***wEvent***, and the message data (or pointer to data) contained in ***dwEventData***, comprise the event message. The value and meaning of the message data varies according to the individual message. The individual telephony device messages for each device are detailed in their corresponding MCI command reference chapters of this document.

The following code example illustrates the initialization of a Microsoft Windows application, including the MCI event handler. Modifications to the code include the implementation of three distinct functions to handle the initialization of the MCI driver environment. These functions have been isolated in the example for the sake of clarity, and could be integrated into the main program logic of an actual Microsoft Windows application. The functions are as follows:

InitDriverEnv() Initializes the driver environment by opening the driver and installing the event handler.

UninitDriverEnv() Uninitializes the driver, by simply closing the device.

EventHandler() Receives messages from MCI (both **MM_MCINOTIFY** and **MM_MCIEVENT**), and acts on these messages. This routine is shown as a separate window procedure, but the code could easily be merged into the main window procedure.

An entire OS/2 and Microsoft Windows startup example is shown below. Some of the modifications made to the generic application startup routines have been highlighted for easier reference.

```
//-----------------------------------------------------------------------
// Windows Sample Code                                          GENAPP.C
//-----------------------------------------------------------------------
//
// This example shows how t o open the MCI device, and initialize the
// event handler.  The event handler routine shown in this example,
//  receives MM_MCINOTIFY messages as well as MM_MCIEVENT messages.
//
// WinMain()    - Invokes initialization & contains message loop
// InitApplication() - Register window classes
// InitInstance() - Create application & event handler windows
// InitDriverEnv() - Initialize MCI driver & register event handler
// UninitDriverEnv() - Close the MCI driver
// EventHandler() - Process incoming M CI messages
// MainWndProc()   - Main program window proc
//-------------------------------------------------------------------
#include <windows.h>
#include <mmsystem.h>
#include <mciftdd.h>
#include <stdio.h>

HANDLE hInst;
HWND   hMainWnd,hEventHandler;
static UINT wOurDeviceID = 0;

//
// WinMain - Program entry point
//
int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow)
HANDLE hInstance;
HANDLE hPrevInstance;
LPSTR lpCmdLine;
int nCmdShow;
{
MSG msg;

// Register the window classes i f first time through, else abort
if( hPrevInstance || !InitApplication(hInstance) )
       return (FALSE);

// Create the main windows and event handler
if( !InitInstance(hInstance, nCmdShow) )
       return (FALSE);

// Initialize the MCI driver and begin execution
if( InitDriverEnv() )
      {
       while (GetMessage(&msg,NULL,NULL,NULL))
            {
             TranslateMessage(&msg);
             DispatchMessage(&msg);
            }
// Close and clean up the MCI driver environment
UninitDriverEnv();
      }
return (msg.wParam);
}

//
// InitApplication - Register the window classes to be used
//
BOOL InitApplication(hInstance)
HANDLE hInstance;
{
WNDCLASS  wc;
BOOL  bTmp;

wc.style           = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
wc.lpfnWndProc  = MainWndProc;
wc.cbClsExtra   = 0;
wc.cbWndExtra   = 0;
wc.hInstance    = hInstance;
wc.hIcon           = LoadIcon(hInstance,"AppIcon");
wc.hCursor      = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground   = GetStockObject(BLACK_BRUSH);
wc.lpszMenuName = "AppMenu";
wc.lpszClassName   = "AppWClass";
bTmp = RegisterClass(&wc);

wc.style          = 0;
wc.lpfnWndProc = EventHandler;
wc.cbClsExtra  = 0;
wc.cbWndExtra  = 0;
wc.hInstance   = hInstance;
wc.hIcon          = 0;
wc.hCursor     = 0;
wc.hbrBackground  = 0;
wc.lpszMenuName= 0;
wc.lpszClassName  = "HandlerWClass";
return( RegisterClass(&wc) && bTmp );
}

//
// InitInstance - Create the main window and the event handler window
```

```
//
BOOL InitInstance(hInstance, nCmdShow)
HANDLE hInstance;
intnCmdShow;
{
hInst = hInstance;

hMainWnd = CreateWindow("AppWClass","Event Handler Examp  le",
                        WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,
                        CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,
                        NULL,NULL,hInstance,NULL );

hEventHandler = CreateWindow("HandlerWClass",0,0,0,0,0,0,
                             NULL,NULL,hInstance,NULL);

if (!hMainWnd || !hEventHandler )
       return (FALSE);

ShowWindow(hMainWnd, nCmdShow);
UpdateWindow(hMainWnd);

return (TRUE);
}

//
// InitDriverEnv - Initialize the MCI driver environment
//
UINT InitDriverEnv()
{
MCI_OPEN_PARMS  mciOpenParms;
MCI_FAX_SET_PARMS  mciSetParms;

// Open the MCI Driver of choice (in this case FAX), and register
// the message handler using MCI_SET...
mciOpenParms.dwCallback = hEventHandler; //set handle in dwCallback for open
mciOpenParms.lpstrDeviceType = "Mwavefax";
if( mciSendCommand(0,MCI_OPEN,MCI_OPEN_TYPE,
                                  (DWORD)(LPVOID)&mciOpenParms) )
       MessageBox(hMainWnd,"MCI Open Error","MCI_OPEN",MB_OK);
   else
      {
       wOurDeviceID = mciOpenP arms.wDeviceID;
       mciSetParms.dwItem= MCI_FAX_SET_EVENT_HANDLER;
       mciSetParms.dwSetData = hEventHandler;
       mciSendCommand(wOurDeviceID,MCI_SET,
                      MCI_SET_ITEM,(DWORD)(LPVOID)&mciSetParms);
      }
return(wOurDeviceID);
}

//
// UninitDriverEnv - Close down the MCI driver
//
void UninitDriverEnv()
{
MCI_GENERIC_PARMS mciGenericParms;
// Here we'll simply close the driver...
     mciGenericParms.dwCallback = hEventHandler; //Set handle in dwCallback for Close
mciSendCommand( wOurDeviceID, MCI_CL OSE, MCI_WAIT, &mciGenericParms );
}
//
// EventHandler - Handle messages from MCI
//
long FAR PASCAL EventHandler(hWnd, message, wParam, lParam)
HWND hWnd;
unsigned message;
WPARAM wParam;
LPARAM lParam;
{
static UINT  uMCIMessage = 0xffff;
unsigned short wDeviceID;
unsigned short wEvent;
  DWORD         dwEventData;
   char             tmpstr[80];

switch (message)
      {
       case WM_CREATE:
             // Register the new event message to be received
             uMCIMessage = RegisterWindowMessage("MM_MCIEVENT");
             break;

       case MM_MCINOT IFY:
             // *** Received a NOTIFY message ***
```

```
            // Get DeviceID of the driver which is sending the message
            wDeviceID = LOWORD( lParam );

            // Check the message...
            switch( wParam )
               {
                  .
                Handle MM_MCINOTIFY messages here.
                  .
               }
            break;

       default:
            if( message == uMCIMessage )
               {
                LPMCI_EVENT_PARMS mep = (LPMCI_EVENT_PARMS)lParam;
                // *** Received an EVENT message ***

                // Get DeviceID of the driver which is sending the message
                wDeviceID = HIWORD( mep->dwDataParam1 ) ;

                // Get message being sent (wEvent). We could simply assign
                // wEvent = wParam;, but for illustration we'll use...
                wEvent = LOWORD( mep->dwDataParam1 );

                // Get the data associated with the message (dwEventData)
                dwEventData = mep->dwEventData;

                // Check the message...
                switch( wEvent )
                  {
                      .
                   Handle MM_MCIEVENT messages here.
                      .
                  }
               }
            else
                return (DefWindowProc(hWnd, message, wParam, lParam));
      }
return (NULL);
}
//
// MainWndProc - This is the window procedure for our main window
//
long FAR PASCAL MainWndProc(hWnd, message, wParam, lParam)
HWND hWnd;
unsigned message;
WPARAM wParam;
LPARAM lParam;
{
      .
Standard Window Procedure
      .
}
```

```
// OS/2 MMPM Sample Code                                          genapp.c
//
// This example shows how to open an MCI device and initialize the
// event handler.
//
// main
// MyWindowProc()      Process messages from MCI
// InitDriverEnv()     Initialize the MCI driver and register event handler
// UninitDriverEnv()   Close the MCI driver
// main()              Main program window procedure
//

#include <os2.h>                        // PM header file
#include "tam.h"                        // also includes mciftdd.h
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

#define STRINGLENGTH 80                 // Length of string

// Function prototypes
MRESULT EXPENTRY MyWindowProc( HWND hwnd, ULONG msg, MPARAM mp1, MPARAM
mp2 );
static int InitDriverEnv(void);
static void UninitDriverEnv(void);

char *lpAppName    = "Mwave TAM";•
char *lpIniName    = "MWTAM.INI";
char *AnnounceFile = "\\announce.tam";
char *AnnounceTmp  = "\\announce.tmp";
long dwBFE;•
                                        // Define parameters by type
HAB  hab;                               // PM anchor block handle
CHAR szTAM[11] = "TAM Sample";          // String parameters, set in
CHAR szString[STRINGLENGTH];            // procedure.
PSZ  pszErrMsg;
HWND hwndClient = NULLHANDLE;           // Client area window handle
HWND hwndFrame = NULLHANDLE;            // Frame window handle
HWND hwndMenu;                          // Handle for the Menu bar
WORD mci_cmd_ctr = 1;
HWND    hEventHandler;
UINT    wTplDeviceID = 0;
UINT    wTpsDeviceID = 0;

//
//  lines omitted for clarity - see tam.c for complete code
//

// ----------------------------------------------------------------
// Main window procedure

INT main (VOID)
{
  HMQ  hmq;                             // Message queue handle
  QMSG qmsg;                            // Message from message queue
  ULONG flCreate;                       // Window creation control flags

  if ((hab = WinInitialize(0)) == 0L)   // Initialize PM
     AbortTam(hwndFrame, hwndClient);   // Terminate the application

  if ((hmq = WinCreateMsgQueue( hab, 0 )) == 0L)// Create a msg queue
     AbortTam(hwndFrame, hwndClient);   // Terminate the application

  if (!WinRegisterClass(                // Register window class
     hab,                               // Anchor block handle
     (PSZ)"MyWindow",                   // Window class name
     (PFNWP)MyWindowProc,               // Address of window procedure
     CS_SIZEREDRAW,                     // Class style
     0                                  // No extra window words
     ))
     AbortTam(hwndFrame, hwndClient);   // Terminate the application


   flCreate = FCF_STANDARD &            // Set frame control flags to
              ~FCF_SHELLPOSITION &
              ~FCF_MAXBUTTON &
              ~FCF_SIZEBORDER &
              ~FCF_ACCELTABLE | FCF_DLGBORDER;

  if ((hwndFrame = WinCreateStdWindow(
               HWND_DESKTOP,            // Desktop window is parent
               WS_VISIBLE,              // STD. window styles
               &flCreate,               // Frame control flag
               "MyWindow",              // Client window class name
               szTAM,                   // No window text
               0,                       // No special class style
               (HMODULE)0L,             // Resource is in .EXE file
               ID_WINDOW,               // Frame window identifier
               &hwndClient              // Client window handle
               )) == 0L)
     AbortTam(hwndFrame, hwndClient);   // Terminate the application

  if (!WinSetWindowPos( hwndFrame,      // Shows and activates frame
                   HWND_TOP,            // window at position 100, 100,
                   100, 100, 550, 80,   // and size 550,  70.
                   SWP_SIZE | SWP_MOVE | SWP_ACTIVATE | SWP_SHOW
                 ))
     AbortTam(hwndFrame, hwndClient); // Terminate the application


   hEventHandler = hwndFrame;
// Get and dispatch messages from the application message queue
// until WinGetMsg returns FALSE, indicating a WM_QUIT message.

  if (InitDriverEnv()) {
    hwndMenu = WinWindowFromID( hwndFrame, FID_MENU );
    while( WinGetMsg( hab, &qmsg, 0L, 0, 0 ) )
      WinDispatchMsg( hab, &qmsg );
  }
  UninitDriverEnv();
  WinDestroyWindow(hwndFrame);           // Tidy up...
  WinDestroyMsgQueue( hmq );             // Tidy up...
  WinTerminate( hab );                   // Terminate the application
} // End of main


// --------------------------------------------------------
// InitDriverEnv  -  initialize the MCI driver environment


static int InitDriverEnv(void)
{
    // Open the MCI driver (in this case, Mwavetpl)

    mciOpenParms.dwCallback = hEventHandler;
    mciOpenParms.lpstrDeviceType = (INT *) "Mwavetpl";
    dwBFE = mciSendCommand(0,MCI_OPEN,MCI_WAIT | MCI_OPEN_TYPE,
                           (DWORD)&mciOpenParms, mci_cmd_ctr++);
    if( dwBFE )
    {
        error_box();
        return(0);
    }

    // Get the device ID & register the Event Handler

    wTplDeviceID = mciOpenParms.wDeviceID;
    mciSetParms.dwCallback = hEventHandler;
    mciSetParms.dwItem     = MCI_TAM_SET_EVENT_HANDLER;
    mciSetParms.dwSetData  = hEventHandler;
    mciSendCommand( wTplDeviceID,MCI_SET,MCI_WAIT | MCI_SET_ITEM,
                    (DWORD)&mciSetParms, mci_cmd_ctr++);

    // Open the MCI Driver (in this case, Mwavetps)

    mciOpenParms.dwCallback = hEventHandler;
    mciOpenParms.lpstrDeviceType = (INT *)"Mwavetps";
    dwBFE = mciSendCommand(0,MCI_OPEN,MCI_WAIT | MCI_OPEN_TYPE,
                           (DWORD)&mciOpenParms, mci_cmd_ctr++);
    if( dwBFE )
    {
        error_box();
        return(0);
    }

    // Get the device ID & register the Event Handler

    wTpsDeviceID = mciOpenParms.wDeviceID;
    mciSetParms.dwCallback = hEventHandler;
    mciSetParms.dwItem     = MCI_TAM_SET_EVENT_HANDLER;
    mciSetParms.dwSetData  = hEventHandler;
    mciSendCommand( wTpsDeviceID,MCI_SET,MCI_WAIT | MCI_SET_ITEM,
                    (DWORD)&mciSetParms, mci_cmd_ctr++);

    // Set to receive TAM phone calls
    mciSetParms.dwItem    = MCI_TAM_SET_CALL_FILTER;
    mciSetParms.dwSetData = 1;
    mciSendCommand( wTplDeviceID,MCI_SET,MCI_WAIT | MCI_SET_ITEM,
                    (DWORD)&mciSetParms, mci_cmd_ctr++);

    return(wTplDeviceID);
}


// -------------------------------------------------------
// UninitDriverEnv  - close the MCI driver•
//
```

```
static void UninitDriverEnv(void)
{
    mciGenericParms.dwCallback = hEventHandler;

    dwBFE = mciSendCommand( wTplDeviceID, MCI_CLOSE, MCI_WAIT,
              (DWORD)&mciGenericParms, mci_cmd_ctr++);
    if( dwBFE )
       error_box();

    dwBFE = mciSendCommand( wTpsDeviceID, MCI_CLOSE, MCI_WAIT,
              (DWORD)&mciGenericParms, mci_cmd_ctr++);
    if( dwBFE )
       error_box();
}



// --------------------------------------------------
// MyWindowProc - event handler for messages from MCI
//

MRESULT EXPENTRY MyWindowProc( HWND hwnd, ULONG msg, MPARAM mp1, MPARAM
mp2 )
{
  HDC hdc;
  static int          InitEnv    = 0;
  static short        wKeys[3];              // Last 3 keys entered
  static short        wQuiet;                // Count for QUIET messages
  static short        wKeysPressed;          // Count for 3 key command
  static short        wCmdKey;               // Flag for 5-x play ctrl
  unsigned short      wEvent;
  unsigned long       dwEventData;
  static int          FlashState = 0;

  switch( msg )
  {
    case MM_MCINOTIFY:
        switch( SHORT1FROMMP(mp1) )
        {
          case MCI_NOTIFY_FAILURE:
          case MCI_NOTIFY_SUCCESSFUL:
          case MCI_NOTIFY_SUPERSEDED:
          case MCI_NOTIFY_ABORTED:

            switch( wTamState )
            {
              case TS_COMMAND_MODE:
              case TS_PLAY_MESSAGE:
                PlayComplete();
                break;

              case TS_REMOTE_PLAY:
                ContinueRemote();
                wQuiet = 0;
                break;

              case TS_PLAY_ANNOUNCEMENT:
                RecordMessage();
                wQuiet = 0;
                break;

              case TS_RECORD_MESSAGE:
                SaveMessage();
                break;

              case TS_ARCHIVE_PLAY:
                PlayComplete();
                break;

              default:
                break;
            }
            break;
        }
      break;
    case MM_MCIEVENT:
```

```
mep = (LPMCI_EVENT_PARMS)mp2;
wEvent      = LOWORD( mep->dwDataParam1 );  // or wParam
dwEventData = mep->dwEventData;
if (dwEventData >= '0')
  dwEventData -= '0';
else if (dwEventData == '#')
  dwEventData = 35;
else if (dwEventData == '*')
  dwEventData == 42;

switch(wEvent)
{
  case PHONE_EVENT_CALL_TAM:
    wKeysPressed = 0;
    AnswerCall();
    break;

  case PHONE_EVENT_CALL_TERMINATED:•
    CallTerminated();
    break;

  case PHONE_EVENT_CALL_PROGRESS:
    if( wTamState == TS_RECORD_MESSAGE ||
        (wTamState==TS_REMOTE_PLAY && wRemoteState==RS_WAITING) )
        switch( dwEventData )
        {
          case DIALTONE:
          case SLOWBUSY:•
          case FASTBUSY:
            CallTerminated();
            break;
        }
    break;

  case PHONE_EVENT_LINE_KEY:
    if( wTamState == TS_REMOTE_PLAY )
    {
        if( wCmdKey == 5 )  // Check for play ctrl sequence
        {
            wCmdKey = -1;
            switch( dwEventData )
            {
              case 1:
                SeekMessage(TB_BACK);
                break;
              case 2:•
                if(!(wPause^=1))
                    mciSendCommand( wTpsDeviceID, MCI_RESUME, MCI_WAIT,
                                   (DWORD)&mciGenericParms, mci_cmd_ctr++);
                else
                    mciSendCommand( wTpsDeviceID, MCI_PAUSE, MCI_WAIT,
                                    (DWORD)&mciGenericParms, mci_cmd_ctr++);
                break;
              case 3:
                SeekMessage(TB_FORWARD);
                break;
            }
        }
        else                // Standard Remote Play command
        {
            switch( dwEventData)
            {
              case 1:
                RemoteNext();
                break;
              case 2:
                RemoteRemove();
                break;
              case 3:
                RemoteRepeat();
                break;
              case 4:
                RemoteArchive();
                break;
              case 5:       // Initiate play ctrl sequence
                wCmdKey = (short)dwEventData;
                break;
```

```
                        }
                    }
                }
                else                    // Check for 3 digit command code
                {
                    wKeys[2] = wKeys[1];
                    wKeys[1] = wKeys[0];
                    wKeys[0] = (short)dwEventData;
                    if( ++wKeysPressed > 2 )
                    {
                        if((wKeys[2]*100+wKeys[1]*10+wKeys[0])==wCommandCode)
                        {
                            BeginRemote();
                            wCmdKey = -1;
                        }
                    }
                }
                break;

              default:
                break;
            }

      break;
    case WM_CREATE:
        WinStartTimer(hab, hwnd, 1000,1000UL);
        break;

    case WM_TIMER:
        if( !wNewMessages )
        {
            if( FlashState )
            {
                WinSetWindowText(hwndFrame, szTAM);
                FlashState = 0;
            }
        }
        else if( FlashState ^= 1 )
            WinSetWindowText(hwndFrame," ");
        else
            WinSetWindowText(hwndFrame, szTAM);
        break;

    case WM_COMMAND:
      //
      // When the user chooses option 1, 2, or 3 from the Options pull-
      // down, the text string is set to 1, 2, or 3, and
      // WinInvalidateRegion sends a WM_PAINT message.
      // When Exit is chosen, the application posts itself a WM_CLOSE
      // message.

      {
      USHORT command;                   // WM_COMMAND command value
      command = SHORT1FROMMP(mp1);      // Extract the command value
      switch (command)
      {
        case ID_RECANNOUNCE:
          WinDlgBox( HWND_DESKTOP,      // Place anywhere on desktop
                     hwndFrame,         // Owned by frame
                     RecordAnnounce,    // Address of dialog procedure
                     (HMODULE)0,        // Module handle
                     RECANNOUNCE,       // Dialog identifier in resource
                     NULL);             // Initialization data

          WinInvalidateRegion( hwnd, NULLHANDLE, FALSE ); // Force a repaint
          break;

        case ID_SETRING:
          WinDlgBox( HWND_DESKTOP,      // Place anywhere on desktop
                     hwndFrame,         // Owned by frame
                     SetRingCount,      // Address of dialog procedure
                     (HMODULE)0,        // Module handle
                     RINGCOUNT,         // Dialog identifier in resource
                     NULL);             // Initialization data

          WinInvalidateRegion( hwnd, NULLHANDLE, FALSE ); // Force a repaint
          break;
        case ID_COMMANDCODE:
          WinDlgBox( HWND_DESKTOP,      // Place anywhere on desktop
                     hwndFrame,         // Owned by frame
                     SetCommandCode,    // Address of dialog procedure
                     (HMODULE)0,        // Module handle
                     COMMANDCODE,       // Dialog identifier in resource
                     NULL);             // Initialization data

          WinInvalidateRegion( hwnd, NULLHANDLE, FALSE ); // Force a repaint
          break;


        case ID_RESET:
          mp1 = (MPARAM)ID_PHONE;
        case ID_PHONE:
        case ID_TAM:
        case ID_HANDSET:
        case ID_SPEAKER:
        case ID_FIRST:
        case ID_PREVIOUS:
        case ID_AGAIN:
        case ID_NEXT:
        case ID_ERASE:
        case ID_REVERSE:
        case ID_PAUSE:
        case ID_FORWARD:
        case ID_FAST:
        case ID_NORMAL:
        case ID_SLOW:
          hdc = WinOpenWindowDC( hwnd );
          ButtonAction(hdc,Menu2Button[SHORT1FROMMP(mp1)-ID_PHONE]);
          break;

        case ID_V0:
        case ID_V1:
        case ID_V2:
        case ID_V3:
        case ID_V4:
        case ID_V5:
        case ID_V6:
        case ID_V7:
        case ID_V8:
        case ID_V9:
            SetVolume(SHORT1FROMMP(mp1)-ID_V0);
          break;
        case ID_QUIT:
          WinPostMsg( hwnd, WM_QUIT, (MPARAM)0,(MPARAM)0 );// Cause termination
          break;
        default:
          return WinDefWindowProc( hwnd, msg, mp1, mp2 );
      }

      break;
      }
    case WM_ERASEBACKGROUND:
      //
      // Return TRUE to request PM to paint the window background
      // in SYSCLR_WINDOW.

      return (MRESULT)( TRUE );
    case WM_PAINT:

      // Window contents are drawn here in WM_PAINT processing.

      {
      HPS    hps;                       // Presentation Space handle
      RECTL  rc;                        // Rectangle coordinates
      POINTL pt;                        // String screen coordinates
                                        // Create a presentation space
      if( !InitEnv ) {
         InitEnv = 1;
         InitTamState();
         ButtonAction(hdc,TB_TELEPHONE);
         SetVolume(wVolume);
      }
      hps = WinBeginPaint( hwnd, 0L, &rc );
      pt.x = 1; pt.y = 5;             // Set the text coordinates,
      GpiSetColor( hps, CLR_NEUTRAL );         // colour of the text,
      GpiSetBackColor( hps, CLR_BACKGROUND );  // its background and
      GpiSetBackMix( hps, BM_OVERPAINT );      // how it mixes,
                                               // and draw the string...
      GpiCharStringAt( hps, &pt, (LONG)strlen( szString ), szString );
      WinEndPaint( hps );                      // Drawing is complete
      break;
      }
    case WM_CLOSE:
      //
      // This is the place to put your termination routines
      //

      sprintf(szString,"%d", wVolume);
      PrfWriteProfileString(hini, lpAppName,"VOL",szString);
      sprintf(szString,"%d", wMsgOut);
      PrfWriteProfileString(hini, lpAppName,"MSGOUT",szString);
      sprintf(szString,"%d", dwMsgIndex);
      PrfWriteProfileString(hini, lpAppName,"MSGIDX",szString);

      PrfCloseProfile(hini);
      WinPostMsg( hwnd, WM_QUIT, (MPARAM)0,(MPARAM)0 );// Cause termination
      break;
    default:
      //
      // Everything else comes here.  This call MUST exist
      // in your window procedure.


      return WinDefWindowProc( hwnd, msg, mp1, mp2 );
  }
  return (MRESULT)FALSE;
} // End of MyWindowProc


// ----------------------------------------------
// AbortTam

VOID AbortTam(HWND hwndFrame, HWND hwndClient)
{
   PERRINFO  pErrInfoBlk;
   PSZ       pszOffSet;

   DosBeep(100,10);
   if ((pErrInfoBlk = WinGetErrorInfo(hab)) != (PERRINFO)NULL)
   {
      pszOffSet = ((PSZ)pErrInfoBlk) + pErrInfoBlk->offaoffszMsg;
      pszErrMsg = ((PSZ)pErrInfoBlk) + *((PSHORT)pszOffSet);
      if((INT)hwndFrame && (INT)hwndClient)
         WinMessageBox(HWND_DESKTOP,         // Parent window is desk top
                       hwndFrame,            // Owner window is our frame
                       (PSZ)pszErrMsg,       // PMWIN Error message
                       "Error Msg",          // Title bar message
                       MSGBOXID,             // Message identifier
                       MB_MOVEABLE | MB_CUACRITICAL | MB_CANCEL ); // Flags
      WinFreeErrorInfo(pErrInfoBlk);
   }
   WinPostMsg(hwndClient, WM_QUIT, (MPARAM)NULL, (MPARAM)NULL);
} // End of AbortTam
```

The switch statements in the event handler routine are of special interest. As further code examples are provided in the Fax and TAM sections of this document, the case code for the switch statements will be filled in with code specific to the operation of the application. The event handler is the key section to this example. First, it is new to even the experienced MMPM programmer, and secondly, it is the foundation on which to build message driven applications.

Chapter 4 - Fax Services

This chapter describes the telephony services available to application developers for the purpose of developing Mwave compatible Fax based applications.

Mwave Fax Device Driver Architecture

In order to develop a functional OS/2 MMPM or Microsoft Windows fax application, three items are required: 1) hardware capable of providing fax send/receive functions, 2) a fax device driver to control the fax hardware based on inputs from the application, and 3) a device-independent programming interface between the application and the fax device driver to isolate the application from specific device driver and hardware differences.

The following block diagram illustrates this architecture as provided by the Mwave system:

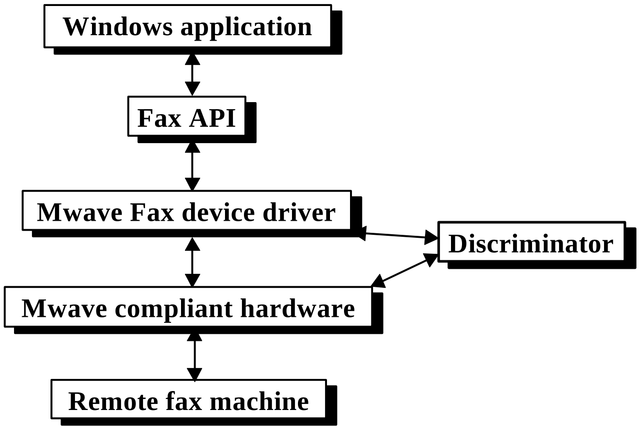

Figure 4-1

This section provides an overview of the FAX Application Programming Interface and the Mwave Fax device driver used to develop an OS/2 or Microsoft Windows Mwave fax application.

The Fax Application Programming Interface (API)

The FAX Application Programming Interface (API), described fully in Chapter 6 of this manual, provides the interface between an OS/2 MMPM or Microsoft Windows application and a fax device driver compliant with the FAX API. The Mwave system provides such a fax device driver, enabling any application calling the FAX API to access the fax capabilities of Mwave compliant hardware.

The FAX API was designed to be very similar to the Media Control Interface (MCI) standard used in MMPM and Windows. The fax specific extensions to the MCI API were designed to provide a 'hands-off' interface to a fax driver's send and receive capabilities, while supplying a rich set of features and options. The MCI command message format is ideal for setting and tracking device status in an orderly manner, and with a couple of command extensions, allows the applications programmer to easily incorporate fax send and receive capabilities into an application.

The FAX API provides support for the following basic operations:

- Receiving a Fax Document File
- Sending a Fax Document File
- Converting from Fax Document File format to Device Independent Bitmap (DIB) format and visa-versa

The following sections describe various aspects and features of the FAX API, and how they relate to the Mwave Fax device driver implementation.

Fax Document File Format

For simplicity, the FAX API command extensions to the standard MCI functions include file send and file receive commands. Fax documents (single and multiple pages) are treated as a single Fax Document File to simplify the process of communicating with the fax device driver. Under the single file scheme, the application need only call a single send or receive command to send or receive an entire fax document, allowing the application to monitor the progress of the operation, but not requiring constant maintenance of the transmission procedure.

The native Fax Document File format used by the Mwave Fax device driver to send and receive fax image data is TIFF Class F. This format was chosen because it provides efficient, multiple-page storage capability.

An Mwave fax application can work directly with fax files in their native TIFF Class F format. Alternatively, the FAX API provides commands to convert from device independent Fax Document File format to Device Independent Bitmap (DIB) format and visa-versa. Thus, the application is able to use DIB format when displaying and printing fax images, but use the more efficient Fax Document File format to store the fax image data to disk.

The file format conversion commands provided by the Fax API enable an application to construct Fax Document Files from DIB files, and extract a DIB format file from a Fax Document File. A Fax Document File is composed of multiple pages of fax image data, while a DIB file represents a single page from a multi-page Fax Document File. Using the file format conversion commands allows the extraction, insertion, and/or replacement of any page within a Fax Document File.

Command Message Summary

The main design goals when defining the MCI commands for the FAX API were ease of use and hands-off operation. The following table provides a summary of the MCI commands available through the FAX API and Mwave Fax device driver. For complete details on these commands, see the FAX API Reference, Chapter 6 of this document.

MCI Command	Description
MCI_CLOSE	Close the device driver
MCI_CONVERT	Convert to / from device dependent file data
MCI_DIAL	Dial the telephone
MCI_GETDEVCAPS	Get the capabilities of the device
MCI_INFO	Get device string identifier
MCI_OPEN	Open the device driver
MCI_RECEIVE	Receive a fax file
MCI_SEND	Send a fax file
MCI_SET	Configure the device
MCI_STATUS	Query device configuration

Table 4-1: MCI Command Summary

Programmers familiar with the standard MCI specification will note the addition of the following MCI commands to the FAX API:

- MCI_CONVERT
- MCI_DIAL
- MCI_RECEIVE
- MCI_SEND

The MCI_CONVERT command is required to convert device independent Fax Document Files (TIFF Class F in the case of the Mwave Fax device driver) to/from DIB files. The MCI_DIAL command is required to dial the telephone device. The other two new commands, MCI_RECEIVE and MCI_SEND, provide generic multi-page file receive and send capability.

Event Message Summary

The Mwave Fax device driver uses event messages to inform an application when various telephony-related events occur. The following table provides a summary of the MCI event messages which an application can receive from the Mwave Fax device driver. For complete details on these event messages, see Chapter 6 of this manual.

Event Message	Description
PHONE_EVENT_CALL_PROGRESS	Call progress state has changed
PHONE_EVENT_CALL_FAX	An incoming fax call has been received
PHONE_EVENT_CALL_TERMINATED	Call terminated (supplies termination code)
PHONE_EVENT_CALLER_ID	Caller ID string detected (supplies string pointer)
PHONE_EVENT_FAX_CONNECT	Returns connection parameters
PHONE_EVENT_FAX_HEADER	Supplies fax header from calling machine
PHONE_EVENT_FAX_PAGE_COMPLETE	Signals that a fax page has been completed
PHONE_EVENT_FAX_PAGE_STATUS	Supplies individual page completion status
PHONE_EVENT_FAX_POLL	Request to poll received
PHONE_EVENT_HANDSET	Change in handset status (supplies status)
PHONE_EVENT_HANDSET_KEY	Keypad press from handset (supplies character)
PHONE_EVENT_LINE	Change in hook status (supplies status)
PHONE_EVENT_LINE_KEY	Keypad press from line (supplies character)
PHONE_EVENT_RING	Telephone ring status (supplies ring on/off)

Table 4-2: MCI Event Message Summary

Developing an Mwave Fax Application

This section describes how to develop an application which calls the FAX API to access the Mwave Fax device driver, providing fax send and receive capabilities.

Throughout this section, code snippets are used to illustrate the basic concepts of how to use the Mwave Fax device driver. These code snippets are part of a complete fax application example, fax.exe (referred to as FAXAPP throughout the remainder of this section). included on the companion diskette. One example is provided for OS/2 and one for Microsoft Windows. The source code is provided for reference, and can also be used as a starting point from which you can develop your own Mwave fax application.

The code snippets in this book are accompanied by the filename and function name (indicated by ***[filename: function()]***) of the FAXAPP source module where the corresponding source code can be located.

FAXAPP Application Definition

FAXAPP demonstrates the basic concepts required to add fax send/receive/view capability to an application through the use of the Mwave Fax device driver. These concepts are illustrated by providing support for the following capabilities:

- Send a Fax Document File or DIB file to a remote fax machine via a user specified phone number. The send operation runs in the background, allowing other operations to occur.

- Receive a Fax Document File from a remote fax machine. The receive operation runs in the background, allowing other operations to occur.

- View a user specified page from a Fax Document File. The view operation runs in the foreground.

The purpose of FAXAPP is to provide an example of using the basic fax send and receive capabilities of the FAX API and the Mwave Fax device driver. As a result, there are many other capabilities provided by the FAX API and Mwave Fax device driver which are not demonstrated in FAXAPP, but might be useful for your particular application. Additionally, although the sample is a functional fax machine, it doesn't contain the error handling or feature set of a complete robust application.

How to run FAXAPP

This section provides a brief overview of how to run the example fax application FAXAPP provided in the Mwave system.

Starting FAXAPP

Copy the \fax subdirectory (for OS/2 or Windows) from the companion diskette to a \fax subdirectory (or other convenient subdirectory) on your system. Add a fax icon to the desktop if you wish. To start FAXAPP, simply double-click on the Fax program icon (or start from an OS/2 command line or the Windows File|Run menu). When the fax application starts, a screen appears to tell you that its initialization has completed. Click OK to continue the operation.

Sending a fax

Facsimile machines send and receive files in a format called TIFF Class F format. Many PC programs generate and manipulate image files in a different format called BMP (bitmapped). The fax application supports both these formats BMP files are converted to TIFF format before being sent The Mwave FAX driver supports conversion of BMP files to TIFF (and vice-versa), and the application uses this conversion support. The application supports only black and white (monochrome) files. Color BMP files are not supported by the Mwave FAX device driver.

To send a file:

1. Select the Send command from the Options menu.

2. Select the file (either a BMP or TIFF file) to be sent. If the selected file is a BMP file, the application will convert the file from BMP format to TIFF format before being sent. The application prompts you to enter the destination filename where the converted TIFF file is to be stored. Be sure and specify a .TIF filename extension for the destination file. Also, make sure the BMP file is monochrome. Color BMP files will not be sent correctly.

3. Enter the phone number of the remote fax machine when prompted. The specified file is then sent, in the background, to the remote fax machine, allowing you to continue to use other applications. A message box is displayed after successful completion or call termination due to an error.

Receiving a fax

The application automatically receives fax data from incoming fax calls.

The receive operation proceeds in the background, allowing you to continue working with other applications. Message boxes are used to notify you that a fax call has been received, to display the receive operation's completion status (either success or failure), and to indicate the name of the received TIFF file.

Viewing a fax

FAXAPP enables you to view fax data from a BMP file or a single page from a TIFF FAX Document File. To view a fax:

1. Select the View command from the Options menu.

2. Select the file (either a BMP or TIFF file) to be viewed. TIFF files will be converted to BMP format before being viewed. For TIFF files, the application prompts you to enter the destination filename where the converted BMP file will be stored. Be sure and specify a .BMP filename extension for the destination file. If there are multiple pages of fax data within the specified TIFF file, you are prompted to select the number of the page you wish to view.

3. When the conversion completes (if conversion was required...if you asked to view a TIFF file), the fax application displays the file. (The application actually displays a negative image of the file. Where the original image is dark, the displayed image is light and vice-versa).

Other commands

Two additional commands are available from the FAXAPP Options menu. They are:

Hang up
This command hangs up the phone (places the phone device on hook). You can

hang up the phone any time.

Clear screen
This command clears the fax image currently displayed (if any) in the fax application window. Select the View command if you want to view another fax.

FAXAPP Code Model Design

This section briefly describes a few of the design considerations used to develop FAXAPP.

The primary goal of FAXAPP is to illustrate the operation of a fax device, which is event (message) driven, and thus is able to execute as a background task. There are some functions which are performed in the foreground, but these include only those functions (such as viewing a BMP file) invoked when the user is using FAXAPP in the foreground.

Because FAXAPP is designed to be message and event driven, a brief review of the types of messages and events which can be sent to FAXAPP by the host PC and the Mwave Fax device driver is useful.

MM_MCIEVENT
The MM_MCIEVENT event message is sent by the Mwave Fax device driver as a direct result of an external telephony event. All event messages are for notification purposes only, and the application is not required to perform any action to handle any of these events. The event messages are very useful however for writing event driven applications. The messages that are handled in FAXAPP are:

- PHONE_EVENT_CALL_FAX
  An incoming fax call has been received. The application must begin receiving the incoming fax data.

- PHONE_EVENT_CALL_TERMINATED
  An active call (send or receive) has been terminated, either successfully, or due to some error condition.

MM_MCINOTIFY (Windows only)
The MCI notification message MM_MCINOTIFY is the standard method for MCI to notify an application that an MCI command has been completed. This message is sent to an application whenever an MCI command is called with the MCI_NOTIFY flag specified. In FAXAPP, the MCI_NOTIFY flag is used instead of the MCI_WAIT flag for those MCI commands (MCI_DIAL and MCI_RECEIVE) which can take a substantial amount of time to complete, thus freeing the Microsoft Windows system to respond to other actions.

The MM_MCINOTIFY message is handled in the ***WndProc()*** function in **fax.c**. Receipt of MM_MCINOTIFY messages cause a change in the FAXAPP state machine.

The FAXAPP State Machine (Windows only)

FAXAPP's send and receive operations are implemented using a very simple state machine. FAXAPP can be in only one of the following states at any given time:

State	Description
STATE_IDLE	No send or receive operation is in progress.
STATE_DIALING	Fax driver is dialing and attempting to connect with a remote fax machine.
STATE_SENDING_FAX	Connection with remote fax machine complete. Fax data is being sent.
STATE_RECEIVE_SETUP	An incoming fax call has been detected. The fax driver is now being set up to receive incoming fax data from a remote fax machine.
STATE_RECEIVING_FAX	Fax driver is receiving incoming fax data.

Table 4-4: FAXAPP States

The FAXAPP state machine proceeds from state-to-state based on MM_MCINOTIFY messages received from the MCI_DIAL and MCI_RECEIVE commands (which are called with the MCI_NOTIFY flag specified). We'll get into more detail about these state changes in later sections, which deal with how to send and receive fax files.

FAXAPP uses the same window procedure (***WndProc()*** in **fax.c**) to handle messages sent by both Microsoft Windows and the Mwave Fax device driver. The single message handling procedure implemented in FAXAPP was done purely for demonstration purposes. A dual procedure approach (one procedure handling Microsoft Windows messages and the other handling event message from the Mwave Fax device driver) could just as easily been used. For an example of a dual-procedure approach, see the TAM sample application provided on the companion diskette.

Received Fax Document Filenames

In order for FAXAPP to receive files in the background without requiring the user to specify a destination filename, a simple file naming scheme is used to automatically store received Fax Document Files.

Each time a fax call is received, the corresponding fax image data (either a single page or multi-page fax) is stored in a file with a file name format of FAX??.TIF, where ?? is a sequential decimal value which is incremented after the completion of every fax call, and is reset to zero whenever FAXAPP is started. Thus, FAXAPP overwrites an existing FAX0.TIF, FAX1.TIF, etc. whenever it receives new incoming fax calls after being restarted.

FAXAPP Source File Descriptions

The following source files comprise the FAXAPP example application:

File	Description
makefile	Microsoft C 7.0 / Windows 3.1 or IBM C Set/2 compatible make file
fax.c	Contains FAXAPP initialization code and procedure to handle all window messages
faxdlgs.c	Contains functions to display and process dialog boxes
faxops.c	Contains all functions which interface to the FAX API
view.c	Contains functions to enable viewing of fax image data
fax.def	Linker definition file
fax.h	FAXAPP specific include file
mciftdd.h	Mwave Fax/TAM device driver include file
fax.rc	FAXAPP resource file

Table 4-5

Opening and Initializing the Mwave Fax Driver - Windows

The first thing FAXAPP must do before sending and/or receiving fax files is to open and initialize the Mwave Fax device driver. The steps required are:

Step 1. Register the MM_MCIEVENT message.
Step 2. Open the Mwave Fax device driver.
Step 3. Set up the fax driver event handler.
Step 4. Set up the call filter.

This sequence of steps is performed automatically whenever FAXAPP is started (see the ***WinMain()*** function in **fax.c**). Let's take a closer look at each step in the Mwave Fax device driver open and initialization process.

Step 1. Register the MM_MCIEVENT message

The Mwave Fax device driver communicates events to the application through the use of the "MM_MCIEVENT" message (see "Developing an Mwave Telephony Application" on page 1-23 for complete details on the "MM_MCIEVENT" message). Because this is not a standard MCI message under Microsoft Windows, it must be registered.

In addition, the "MM_MCIEVENT" message must be registered prior to setting the event handler window procedure (Step 3 in the open/initialize process) which handles messages sent to our application from the Mwave Fax device driver. This is to insure that our application does not miss any "MM_MCIEVENT" messages which can be sent by the driver.

The following call registers the "MM_MCIEVENT" message and assigns the numeric value returned to the global variable `uMCIMessage` .

[fax.c: WinMain()]

```
/*------------------------------------------------------------------*/
/*  Register the MM_MCIEVENT message                                */
/*------------------------------------------------------------------*/
uMCIMessage = RegisterWindowMessage( "MM_MCIEVENT" );
```

The Mwave Fax device driver issues a "MM_MCIEVENT" message to Microsoft Windows whenever the driver needs to inform the application that some telephony event has occurred. Microsoft Windows then translates the "MM_MCIEVENT" message request and send the corresponding numeric value returned from the RegisterWindowMessage() function to our application's event handling procedure.

Now that the "MM_MCIEVENT" message has been registered, we can safely open the Mwave Fax device driver.

Step 2. Open the Mwave Fax device driver

The Mwave Fax device driver is identified by the device type "Mwavefax" (case is not sensitive). This device type is used with the MCI_OPEN command to open the driver.

[faxops.c: InitFax()]

```
/*------------------------------------------------------------------*/
/*  Open the device by specifying only the "Mwavefax" device type   */
/*------------------------------------------------------------------*/
ResourceMessageBox( hWnd, IDS_MSG_INIT_DRIVER, 0, szAppName, MB_OK);
SetCursor(hcWaitCursor);
mciOpenParms.dwCallback = (DWORD)hWnd;
mciOpenParms.lpstrDeviceType = "Mwavefax";
dwReturn = mciSendCommand( NULL,                    // device ID
                        MCI_OPEN,                   // command
                        MCI_WAIT | MCI_OPEN_TYPE,   // flags
                        (DWORD)lpmciOpenParms );    // parameter block
if( dwReturn )
{
    ResourceMessageBox(hWnd, IDS_ERR_INIT_DRIVER, (UINT)dwReturn, NULL, MB_OK);
    return( FALSE );
}
wMwaveFaxID = lpmciOpenParms->wDeviceID;
```

NOTE: For Windows, the handle of the window procedure responsible for processing MM_MCINOTIFY messages **MUST** be specified by assigning it to `mciOpenParms.` dwCallback prior to calling the MCI_OPEN command, regardless of whether the MCI_WAIT or MCI_NOTIFY flag is specified in the MCI_OPEN call. Failure to do so when using versions earlier than 2.1 will result in erratic behavior of the Mwave Fax device driver.

If the MCI_OPEN command completes successfully, the device ID of the Mwave Fax device driver (returned in `lpmciOpenParms->wDeviceID`) is assigned to the variable `wMaveFaxID`. This variable is specified in the remaining ***mciSendCommand*** calls to identify the Mwave Fax device driver.

As illustrated in the code example above, you should always check the return value from the MCI_OPEN command for an error. There are a number of conditions (insufficient memory or MIPS available on the Mwave board) which can cause the Mwave Fax device driver to fail opening, and these cases should be handled properly by the application.

Step 3. Set up the fax driver event handler

After opening the driver, the next step is to set the window procedure our application uses to handle incoming "MM_MCIEVENT" event messages sent by the Mwave Fax device driver. This should be done immediately after opening the driver to minimize the chance of missing any driver event messages. The MCI_SET command with the MCI_FAX_SET_EVENT_HANDLER item is used to set the event handler procedure.

Recall that FAXAPP uses the same window procedure (***WndProc()*** in **fax.c**) to process messages sent by both Microsoft Windows and the Mwave Fax device driver. This window procedure is assigned to our main application window, identified as `hWnd`. Thus, we specify `hWnd` as the window to receive "MM_MCIEVENT" messages.

[faxops.c: InitFax()]

```
/*--------------------------------------------------------------*/
/*  Set up the FAX driver event handler to our main application    */
/*  window procedure, since this is where we will process event    */
/*  messages sent from the FAX driver.                             */
/*--------------------------------------------------------------*/
mciSetParms.dwItem = MCI_FAX_SET_EVENT_HANDLER;
mciSetParms.dwSetData = (DWORD)hWnd;
SetCursor(hcWaitCursor);
dwReturn = mciSendCommand( wMwaveFaxID,
                        MCI_SET,
                        MCI_WAIT | MCI_SET_ITEM,
                        (DWORD)lpmciSetParms );
if( dwReturn )
{
    ResourceMessageBox(hWnd, IDS_ERR_SET_EVENT_HANDLER, (UINT)dwReturn, NULL,
       MB_OK);
    return( FALSE );
}
```

Step 4. Set up the call filter

The last step required in the Mwave Fax device driver initialization process involves setting up the call filter. Setting the call filter to TRUE informs the Mwave Fax device driver that it is to receive fax calls.

Setting the call filter also provides a mechanism to insure that no other application is expecting to receive a fax call. Attempting to enable the call filter when another application has already enabled the filter results in an error return.

[faxops.c: InitFax()]

```
        /*--------------------------------------------------------------*/
        /*  Set the call filter                                         */
        /*--------------------------------------------------------------*/
        mciSetParms.dwItem = MCI_FAX_SET_CALL_FILTER;
        mciSetParms.dwSetData = TRUE;
        SetCursor(hcWaitCursor);
        dwReturn = mciSendCommand( wMwaveFaxID,
                                   MCI_SET,
                                   MCI_WAIT | MCI_SET_ITEM,
                                   (DWORD)lpmciSetParms );
        if( dwReturn )
        {
            ResourceMessageBox(hWnd, IDS_ERR_SET_FILTER, (UINT)dwReturn, NULL, MB_OK);
            return( FALSE );
        }
```

Opening and Initializing the Mwave Fax Driver - OS/2

The first thing FAXAPP must do before sending and/or receiving fax files is to open and initialize the Mwave FAX device driver. The steps required are:

1. Open the Mwave FAX device driver.
2. Set up the fax driver event handler.
3. Set up the call filter.

This sequence of steps is performed automatically whenever FAXAPP is started (see the **main** function in **fax.c**). Let's take a closer look at each step in the Mwave FAX device driver open and initialization process.

Step 1. Open the Mwave FAX device driver

The Mwave FAX device driver is identified by the device type "Mwavefax" (case is not sensitive). This device type is used with the MCI_OPEN command to open the driver.

faxops.c: InitFAX

```
int InitFax(HWND hWnd)
{
char messagestring[255];

   MessageBox (hWnd, "Initializing the fax driver",
               szAppName, MB_OK|MB_ICONEXCLAMATION);

   WinSetPointer (HWND_DESKTOP,
                  WinQuerySysPointer(HWND_DESKTOP,
                  SPTR_WAIT, FALSE));

        mciOpenParms.lpstrDeviceType = (LPSTR) "Mwavefax";
        mciOpenParms.dwCallback = (DWORD) hWnd;

        dwReturn = mciSendCommand(wDeviceID,
                   MCI_OPEN,
                   MCI_WAIT | MCI_OPEN_TYPE,
                   (DWORD) lpmciOpenParms, ++mciCall);

        if (dwReturn)
        {       /* Error, unable to open device  */
          if (!(mciGetErrorString(dwReturn,
              (int *)messagestring, sizeof(messagestring))))
          {
            MessageBox(hWnd, messagestring, NULL,
                       MB_OK|MB_ERROR);
          }
          else
          {
            sprintf(messagestring,
                    "Unable to open device or
                    GetErrorString.RC= %d",
                    (LOWORD(dwReturn)));

            MessageBox(hWnd, messagestring, NULL,
                       MB_OK|MB_ERROR);
          }

          return FALSE;
        }
        /* Device opened successfully, get the device ID */
        wDeviceID = lpmciOpenParms->wDeviceID;
```

If the MCI_OPEN command completes successfully, the device ID of the Mwave FAX device driver is assigned to the variable **wMwaveFaxID**. This variable is specified in the remaining **mciSendCommand** calls to identify the Mwave FAX device driver.

As illustrated in the code example above, you should always check the return value from the MCI_OPEN command for an error. There are a number of conditions (such as insufficient memory or MIPS available in the Mwave DSP) that can cause the Mwave FAX device driver to fail opening. These cases should be handled properly by the application.

Step 2. Set up the fax driver event handler

After opening the driver, the next step is to set the window procedure our application uses to handle incoming "MM_MCIEVENT" event messages sent by the Mwave FAX device driver. This should be done immediately after opening the driver to minimize the chance of missing any driver event messages. The MCI_SET command with the MCI_FAX_SET_EVENT_HANDLER item is used to set the event handler procedure.

Recall that FAXAPP uses the same window procedure (***WndProc()*** in **fax.c**) to process messages sent by both MMPM and the Mwave FAX device driver. This window procedure is assigned to our main application window, identified as `hWnd`. Thus, we specify `hWnd` as the window to receive MM_MCIEVENT messages.

faxops.c: InitFax

```
 mciSetParms.dwCallback = (DWORD) hWnd;
mciSetParms.dwItem = MCI_FAX_SET_EVENT_HANDLER;
mciSetParms.dwSetData = (DWORD) hWnd;

WinSetPointer (HWND_DESKTOP,                WinQuerySysPointer(HWND_DESKTOP, SPTR_WAIT,
FALSE));

dwReturn = mciSendCommand(wDeviceID, MCI_SET,
                          MCI_WAIT | MCI_SET_ITEM,
                          (DWORD) lpmciSetParms, ++mciCall);
if (dwReturn)
   {
       if (!(mciGetErrorString(dwReturn,
             (int *)messagestring, sizeof(messagestring))))
          {
            MessageBox (hWnd, messagestring,
                        NULL, MB_OK|MB_ERROR);
          }
          else
          {
           MessageBox(hWnd,"Unable to set the event handler",
                      NULL, MB_OK);
          }
      return FALSE;•
    }
```

Step 3. Set up the call filter

The last step required in the Mwave FAX device driver initialization process involves setting up the call filter. Setting the call filter to TRUE informs the Mwave FAX device driver that it is to receive fax calls.

Setting the call filter also provides a mechanism to insure that no other application is expecting to receive a fax call. Attempting to enable the call filter when another application has already enabled the filter results in an error return.

```
mciSetParms.dwItem = MCI_FAX_SET_CALL_FILTER;
mciSetParms.dwSetData = TRUE;

WinSetPointer (HWND_DESKTOP, WinQuerySysPointer(HWND_DESKTOP,
                       SPTR_WAIT, FALSE));

dwReturn = mciSendCommand (wDeviceID, MCI_SET,
                           MCI_WAIT | MCI_SET_ITEM,
                           (DWORD) lpmciSetParms, ++mciCall);

if (dwReturn)
  {
     if (!(mciGetErrorString (dwReturn,
          (int *)messagestring, sizeof(messagestring))))
       {
          MessageBox(hWnd, messagestring,
                     NULL, MB_OK|MB_ERROR);
       }
     else
       {
          MessageBox (hWnd,
                  "Another telephony application is in use",
                   NULL, MB_OK);
       }
          return FALSE;
  }
        return TRUE;
```

This concludes the steps required to properly open and initialize the Mwave FAX device driver. We can now proceed with other driver operations, such as sending and receiving FAX Document Files.

Sending a Fax - Windows

Sending a fax using the FAX API and the Mwave Fax device driver requires the following steps:

Step 1. Inform the Fax driver of the names of the Fax Document File(s) to be sent.

Step 2. Take the phone off-hook and dial the phone number of the destination fax machine.

Step 3. Respond to a change (either completion, status change, or error) to the send operation.

The send command is initiated by selecting the Send command from the FAXAPP Options menu. The message procedure for the send command is:

[fax.c: WndProc()]

```
case IDM_SEND:
    /*--------------------------------------------------*/
    /*  Send a fax.                                     */
    /*--------------------------------------------------*/
    strcpy(FileName, "*.tif");
    lpFileName = (GetFileName(hInst, hWnd, "Send Fax", FileName,
        FileNamesz, TIF_FILTERSTRING));
    if (lpFileName!=NULL)
    {
        if (tif = IsTif(lpFileName))
             SendFax(hWnd, lpFileName);
        else if (bmp = IsBmp(lpFileName))
            if ( ConvertBMP2TIF(hWnd, lpFileName)
                 SendFax(hWnd, lpFileName);
            else if (!(tif || bmp))
                ResourceMessageBox(hWnd, IDS_ERR_FILE_FORMAT, 0, NULL, MB_OK);
    }
    break;
```

Please note the following in the message procedure above:

a. FAXAPP is designed to allow only one file (single or multiple pages) to be sent at a time. This is a limitation of FAXAPP, and not of the FAX API and Mwave Fax device driver, both of which provide support for sending multiple files at the same time.

b. FAXAPP allows the user to send either TIF files (TIFF Class F format) or BMP files (DIB format). Since the Mwave FAX driver supports only TIF file sending, FAXAPP converts BMP files to TIF format via the ***ConvertBMP2TIF()*** function. See "Converting Fax Document Files to/from DIB format" on page 1-65 for more information.

The ***SendFax()*** function initiates the three step procedure required to send a fax. Let's examine how each of these steps is implemented.

Step 1. Inform the Fax driver of the names of the Fax Document File(s) to be sent.

The MCI_SEND command is used to specify the name(s) of the Fax Document File(s) to be sent. The filenames are specified by providing to MCI_SEND a pointer to an array of pointers to strings containing the name of each file to be sent. For example, assume `lpSendPtr` is the array of pointers to the 'n' number of filename strings. It is initialized as follows:

```
LPSTR lpSendPtr[n+1];

lpSendPtr[0]   = address of string containing file #1 filename
lpSendPtr[1]   = address of string containing file #2 filename
     :
lpSendPtr[n-1] = address of string containing file #n filename
lpSendPtr[n]   = (LPSTR)NULL;
```

Note that the filename list is terminated by a NULL filename pointer.

In FAXAPP, we declare a two-dimensional array `SendBuff` to store up to two filenames (although only one is used), and then assign the address of the `SendBuff` strings to the `lpSendPtr` array. The filename to send (the address of which is passed as the argument `srcFileName` to the ***SendFax()*** function), is copied into the first element of the `SendBuff` array (pointed to by `lpSendPtr[0]`). Finally, the address of the `lpSendPtr` array is sent to the MCI_SEND command.

[faxops.c: SendFax()]

```
LPSTR   lpSendPtr[10];
char    SendBuff[2][128];

/*-----------------------------------------------------------------*/
/*  Send the fax file                                              */
/*-----------------------------------------------------------------*/
lpSendPtr[0] = SendBuff[0];
lpSendPtr[1] = SendBuff[1];
lstrcpy( lpSendPtr[0], srcFileName );   // send file name
lpSendPtr[1] = '\0';
mciSendParms.lpstrFilename = (LPSTR)lpSendPtr;

SetCursor(hcWaitCursor);
dwReturn = mciSendCommand( wMwaveFaxID,
                MCI_SEND,
                MCI_WAIT | MCI_SEND_FILE,
                (DWORD)lpmciSendParms );
if( dwReturn )
    ResourceMessageBox( hWnd, IDS_ERR_SEND_FILE, (UINT)dwReturn, NULL, MB_OK );
else ...
```

The MCI_SEND command causes the Mwave Fax device driver to be configured for a send. Once the MCI_SEND command completes, the driver has prepared the Fax Document Files for transmission to a remote fax machine.

Step 2. Take the phone off-hook and dial the phone number of the destination fax machine.

The next step is to dial and connect to the remote fax machine. This is done via the MCI_DIAL command.

[faxops.c: SendFax()]

```
/*-------------------------------------------------------------*/
/*  Prompt user for the phone number                           */
/*-------------------------------------------------------------*/
lpfnGetNbr = (DLGPROC)MakeProcInstance((FARPROC)GetNbr_DlgProc, hInst);
if( DialogBox(hInst, "phonenumdlg", hWnd, lpfnGetNbr) )
{
    /*---------------------------------------------------------*/
    /*  Dial the phone number                                  */
    /*---------------------------------------------------------*/
    mciDialParms.lpstrDialString = (LPSTR)PhoneNumber;
    mciDialParms.dwCallback = (DWORD)hWnd;
    SetCursor(hcWaitCursor);
    uAppState = STATE_DIALING;
    mciSendCommand( wMwaveFaxID,
         MCI_DIAL,
         MCI_NOTIFY | MCI_DIAL_STRING | MCI_DIAL_MONITOR | MCI_DIAL_VERIFY,
         (DWORD)lpmciDialParms );
}
```

In FAXAPP, the user is prompted to enter the phone number of the destination fax machine. The phone number is then supplied to the MCI_DIAL command.

Note that MCI_DIAL is called using the MCI_NOTIFY flag instead of the MCI_WAIT flag. This was done for two reasons. The first is that the dial and connect operation could be a lengthy one, and we do not want to tie up the Microsoft Windows system waiting for this operation to complete. Secondly, it enables our message procedure to track the machine state via the MM_MCINOTIFY message. Note that the FAXAPP state is set to `STATE_DIALING` prior to calling MCI_DIAL. A transition from the dialing state to the send fax data state is handled by the message procedure.

Hint: In Microsoft Windows, it is best to use MCI_WAIT with MCI commands when debugging your code. Using MCI_WAIT allows the application to get more descriptive error messages.

Also note that the MCI_DIAL_MONITOR flag is specified in the MCI_DIAL command. This allows the user to monitor the connection negotiation and call progress via speakers or headphones attached to the Mwave board's audio output connector.

After dialing and connecting to the destination fax machine, the Mwave Fax device driver begins sending the fax data to the destination fax machine. At the same time, MCI sends a MM_MCINOTIFY message (since MCI_DIAL was called with the MCI_NOTIFY flag specified) to indicate either successful completion or failure of the MCI_DIAL command. Our message procedure (see below) responds to a successful connection by changing the machine state to `STATE_SENDING_FAX` . If a dial or connection failure occurred, FAXAPP displays an error message, hangs up the line, and resets the machine state to `STATE_IDLE` .

[fax.c: WndProc()]

```
case MM_MCINOTIFY:
    switch(wParam)
    {
        case MCI_NOTIFY_ABORTED:
        case MCI_NOTIFY_FAILURE:
            if( uAppState ==  STATE_DIALING )
                idResource = IDS_ERR_DIALING;
            else if( uAppState == STATE_RECEIVE_SETUP )
                idResource = IDS_ERR_RECEIVE;
            else
                idResource = IDS_ERR_UNKNOWN_STATE;
            ResourceMessageBox(hWnd, idResource, 0, "MM_MCINOTIFY", MB_OK);
            SetOnHook();
            uAppState = STATE_IDLE;
            break;

        case MCI_NOTIFY_SUCCESSFUL:
            if( uAppState ==  STATE_DIALING )
            {
                uAppState = STATE_SENDING_FAX;
                ResourceMessageBox(hWnd, IDS_MSG_SENDING_FILE, 0, szAppName,
       MB_OK);
            }
    else if( uAppState == STATE_RECEIVE_SETUP )
                uAppState = STATE_RECEIVING_FAX;
            break;

         default:
             break;
    }
    break;
```

Step 3. Respond to a change (either completion, status change, or error) to the send operation.

After completing a successful connection, the Mwave Fax device driver sends the fax data to the destination fax machine (in the background) without requiring any support from the application. Upon completion, either successful or due to an error, the driver sends a MM_MCIEVENT event message of type PHONE_EVENT_CALL_TERMINATED to our application.

This message is processed by our event handling procedure as follows:

[fax.c: WndProc()]

```
if( Message == uMCIMessage )
{
    LPMCI_EVENT_PARMS lpMciEventParms = (LPMCI_EVENT_PARMS)lParam;

    /*------------------------------------------------------*/
    /*  A MM_MCIEVENT message was issued by the Mwave FAX   */
    /*  driver.                                             */
    /*------------------------------------------------------*/
    switch(wParam)
    {
        case PHONE_EVENT_CALL_TERMINATED:
            /*----------------------------------------------*/
            /*  Call was terminated.  Check termination     */
            /*  code (in dwEventData) for cause.            */
            /*----------------------------------------------*/
            switch( lpMciEventParms->dwEventData )
            {
                case TERMINATION_NORMAL:
                    if( uAppState ==  STATE_SENDING_FAX )
                        idResource = IDS_MSG_SEND_OK;
                    else if( uAppState == STATE_RECEIVING_FAX )
                        idResource = IDS_MSG_RECEIVE_OK;
                    else
                        idResource = IDS_ERR_UNKNOWN_STATE;
                    break;

                case TERMINATION_UNEXPECTED:
                case TERMINATION_ERROR_XMIT:
                case TERMINATION_ERROR_RECV:
                    if( uAppState ==  STATE_SENDING_FAX )
                        idResource = IDS_ERR_SEND_FAIL;
                    else if( uAppState == STATE_RECEIVING_FAX )
                        idResource = IDS_ERR_RECEIVE_FAIL;
                    else
                        idResource = IDS_ERR_UNKNOWN_STATE;
                    break;

                default:
                    idResource = 0;
                    break;
            }
            if( idResource )
                ResourceMessageBox( hWnd, idResource, 0, "MM_MCIEVENT", MB_OK );
            /*----------------------------------------------*/
            /*  Hang up the phone                           */
            /*----------------------------------------------*/
            SetOnHook();
            uAppState = STATE_IDLE;
            break;
```

Upon receipt of the PHONE_EVENT_CALL_TERMINATED case of the MM_MCIEVENT message, FAXAPP displays either a success or failure message (using the context of the current machine state), hangs up the phone, and resets the machine state back to `STATE_IDLE` .

You might want to respond to other types of MM_MCIEVENT messages (for example, PHONE_EVENT_FAX_PAGE_STATUS) to provide real-time status information to the user during the send operation.

Sending a Fax - OS/2

Sending a fax using the FAX API and the Mwave FAX device driver requires the following steps:

Step 1. Inform the FAX driver of the names of the FAX Document File(s) to be sent.
Step 2. Take the phone off-hook and dial the phone number of the destination fax machine.
Step 3. Respond to a change (either completion, status change, or error) to the send operation.

The send command is initiated by selecting the Send command from the FAXAPP Options menu. The message procedure for the send command is:

fax.c: WndProc

```
case IDM_SEND:
 /*---------------------------------*/
 /*  Send a fax.                    */
 /*---------------------------------*/
 strcpy(FileName, "*.tif");
 lpFileName = (GetFileName("Send Fax",
               (LPSTR) FileName));
 if (strlen((char *)lpFileName) == 0)
 break;
 if (IS_TIF(lpFileName))
    {
    SendFax(hWnd, lpFileName, "");
    break;
    }
 if (IS_BMP(lpFileName))
    if (ConvertBMP2TIF(hWnd, wDeviceID,
        lpFileName))
    {
    SendFax(hWnd, lpFileName, "");
    break;
    }
    MessageBox(hWnd, "Unsupported File Format",
               NULL, MB_OK);
    break;
```

Please note the following in the message procedure above:

- FAXAPP is designed to allow only one file (single or multiple pages) to be sent at a time. This is a limitation of FAXAPP, and not of the FAX API and Mwave FAX device driver, both of which provide support for sending multiple files at the same time.

- FAXAPP allows the user to send either TIF files (TIFF Class F format) or BMP files (DIB format). Since the Mwave FAX driver supports only TIF file sending, FAXAPP converts BMP files to TIF format via the **ConvertBMP2TIF** function. See Section Converting Fax Document Files to/from DIB format on page 1-65 for more information.

The **SendFax** function initiates the three step procedure required to send a fax. Let's examine how each of these steps is implemented.

Step 1. Inform the FAX driver of the names of the FAX Document File(s) to be sent.

The MCI_SEND command is used to specify the name(s) of the FAX Document File(s) to be sent. The filenames are specified by providing to MCI_SEND a pointer to an array of pointers to strings containing the name of each file to be sent. For example, assume *lpSendPtr* is the array of pointers to the 'n' number of filename strings. It is initialized as follows:

```
LPSTR lpSendPtr[n+1];

lpSendPtr[0]   = address of string containing file #1
lpSendPtr[1]   = address of string containing file #2
     :
lpSendPtr[n-1] = address of string containing file #n
lpSendPtr[n]   = (LPSTR)NULL;
```

Note that the filename list is terminated by a NULL filename pointer.

In FAXAPP, we declare a two-dimensional array SendBuff to store up to two filenames (although only one is used), and then assign the address of the SendBuff strings to the lpSendPtr array. The filename to send (the address of which is passed as the argument *srcFileName* to the **SendFax** function), is copied into the first element of the SendBuff array (pointed to by lpSendPtr[0]). Finally, the address of the lpSendPtr array is sent to the MCI_SEND command.

faxops.c: SendFax

```
void SendFax(HWND hWnd, LPSTR srcFileName, char phonenbr[25])
{
  char * lpSendPtr[10];
  char SendBuff[2][256];


/* 1.  Send the fax file */
    mciSendParms.dwCallback = hWnd;
    lpSendPtr[0] = SendBuff[0];
    lpSendPtr[1] = SendBuff[1];
    strcpy((CHAR *)lpSendPtr[0],(CHAR *)srcFileName );
    lpSendPtr[1] = '\0';
    mciSendParms.lpstrFilename = (char *)lpSendPtr;

    WinSetPointer(HWND_DESKTOP,
                  WinQuerySysPointer(HWND_DESKTOP,
                  SPTR_WAIT,FALSE));

    dwReturn = mciSendCommand(wDeviceID, MCI_SEND,
                              MCI_WAIT | MCI_SEND_FILE,
                              (DWORD) lpmciSendParms,
                              mciCall++ );
    if (dwReturn)
    {
        /* Error, unable to send file  */
        MessageBox(hWnd, "Unable to send file", NULL, MB_OK);
    }
    else . . .
```

The MCI_SEND command causes the Mwave Fax device driver to be configured for a send. Once the MCI_SEND command completes, the driver has prepared the Fax Document Files for transmission to a remote fax machine.

Step 2. Take the phone off-hook and dial the phone number of the destination fax machine.

The next step is to dial and connect to the remote fax machine. This is done via the MCI_DIAL command.

faxops.c: SendFax

```
if (WinDlgBox(HWND_DESKTOP, hWnd,
              GetNbr_DlgProc, 0, PHONENUMDLG, NULL))
        {
          mciDialParms.lpstrDialString = PhoneNumber;
          mciDialParms.dwCallback = hWnd;

          WinSetPointer(HWND_DESKTOP,
                        WinQuerySysPointer(HWND_DESKTOP,
                        SPTR_WAIT, FALSE));

          ulRC = mciSendCommand(wDeviceID, MCI_DIAL,
                          MCI_WAIT |
                          MCI_'DIAL_STRING |
                          MCI_DIAL_VERIFY |
                          MCI_DIAL_MONITOR_HANDSHAKING_ONLY,
                          (DWORD) lpmciDialParms,
                          mciCall++);
        }
```

In FAXAPP, the user is prompted to enter the phone number of the destination fax machine. The phone number is then supplied to the MCI_DIAL command.

Also note that the MCI_DIAL_MONITOR_HANDSHAKING_ONLY flag is specified in the MCI_DIAL command. This allows the user to monitor the connection negotiation via speakers or headphones attached to the Mwave board's audio output connector.

After dialing and connecting to the destination fax machine, the Mwave FAX device driver begins sending the fax data to the destination fax machine. If a dial or connection failure occurred, FAXAPP displays an error message, and hangs up the line.

faxevnts.c: ProcessEvent

```
#include "mciftdd.h"
#include <stdio.h>
#include <string.h>
#include "fax.h"

extern int mciCall;

void ProcessEvent(HWND hWnd, WORD wParam,
                  LPMCI_EVENT_PARMS lParam)
{
   char   buf[256];

   switch(wParam)
   {
     case PHONE_EVENT_CALL_FAX:
         /* incoming call--  receive fax and notify user */
         ++NumFax;
         ReceiveFax(hWnd);
         break;

     case PHONE_EVENT_CALL_TERMINATED:
         /* notify user, enter into log */

         switch (lParam->dwEventData)
             {
             case TERMINATION_NORMAL:
               sprintf(buf, "Call completed normally");
               break;
             case TERMINATION_UNEXPECTED:
               sprintf(buf, "Call terminated unexpectedly");
               break;
             case TERMINATION_ERROR_XMIT:
               sprintf(buf,"Callterminated:transmit error");
               break;
             case TERMINATION_ERROR_RECV:
               sprintf(buf,"Callterminated:receive error");
             case TERMINATION_REQUESTED:
               sprintf(buf,"Callterminated:type requested");
               break;
             default:
               break;
             } /* switch */
                            /* hang up the phone    */

       MessageBox(hWnd, buf, "Fax Informations", MB_OK);
                    SetOnHook(hWnd);
       break;

     case PHONE_EVENT_CALLER_ID:
     case PHONE_EVENT_FAX_HEADER:
     case PHONE_EVENT_FAX_PAGE_COMPLETE:
     case PHONE_EVENT_FAX_PAGE_STATUS:
     case PHONE_EVENT_FAX_CONNECT:
     case PHONE_EVENT_LINE:
     case PHONE_EVENT_HANDSET:
     case PHONE_EVENT_CALL_PROGRESS:
       break;

     default:
         break;
     }
 }
```

Step 3. Respond to a change (either completion, status change, or error) to the send operation.

After completing a successful connection, the Mwave FAX device driver sends the fax data to the destination fax machine without requiring any support from the application. Upon completion, either successful or due to an error, the driver sends a MM_MCIEVENT event message of type PHONE_EVENT_CALL_TERMINATED to our application. This message is processed by our event handling procedure as follows:

faxevnts.c: ProcessEvent

```
#include "mciftdd.h"
#include <stdio.h>
#include <string.h>
#include "fax.h"•

extern int mciCall;

void ProcessEvent(HWND hWnd, WORD wParam, LPMCI_EVENT_PARMS lParam)
{
   char   buf[256];

   switch(wParam)
   {
     case PHONE_EVENT_CALL_FAX:
         /* incoming call, receive fax and notify user */
         ++NumFax;
         ReceiveFax(hWnd);
         break;

     case PHONE_EVENT_CALL_TERMINATED:
         /* notify user, enter into log  */
         switch (lParam->dwEventData)
             {
             case TERMINATION_NORMAL:
               sprintf(buf, "Call completed normally");
               break;
             case TERMINATION_UNEXPECTED:
               sprintf(buf, "Call terminated unexpectedly");
               break;
             case TERMINATION_ERROR_XMIT:
               sprintfbuf,"Call terminated: transmit error");
               break;
             case TERMINATION_ERROR_RECV:
               sprintf(buf,"Call terminated: receive error");
             case TERMINATION_REQUESTED:
               sprintfbuf,"Call terminated: type requested");
               break;
             default:
               break;
             } /* switch  */

         /*  Hang up the phone  */
         MessageBox(hWnd, buf, "Fax Informations", MB_OK);
         SetOnHook(hWnd);
         break;

     case PHONE_EVENT_CALLER_ID:
     case PHONE_EVENT_FAX_HEADER:
     case PHONE_EVENT_FAX_PAGE_COMPLETE:
     case PHONE_EVENT_FAX_PAGE_STATUS:
     case PHONE_EVENT_FAX_CONNECT:
     case PHONE_EVENT_LINE:
     case PHONE_EVENT_HANDSET:
     case PHONE_EVENT_CALL_PROGRESS:
       break;

     default:
         break;
     }
 }
```

Upon receipt of PHONE_EVENT_CALL_TERMINATED , FAXAPP displays either a success or failure message and hangs up the phone.

You might want to respond to other types of MM_MCIEVENT messages (for example, PHONE_EVENT_FAX_PAGE_STATUS) to provide real-time status information to the user during the send operation.

This concludes the steps required to send fax data.

Receiving a Fax - Windows

Receiving a fax using the FAX API and the Mwave Fax device driver requires the following steps:

Step 1. Respond to the MM_MCIEVENT message PHONE_EVENT_CALL_FAX

Step 2. Initiate the receive operation in the Mwave Fax device driver.

Step 3. Respond to a change (either completion, status change, or error) to the receive operation.

Let's look at how FAXAPP implements each of these steps.

Step 1. Respond to the MM_MCIEVENT message PHONE_EVENT_CALL_FAX

The Mwave Fax device driver notifies FAXAPP that an incoming fax call has been detected by sending the PHONE_EVENT_CALL_FAX type of MM_MCIEVENT message. This message is processed by our event handling procedure as follows:

[fax.c: WndProc()]

```
if( Message == uMCIMessage )
{
    LPMCI_EVENT_PARMS lpMciEventParms = (LPMCI_EVENT_PARMS)lParam;

    /*------------------------------------------------------*/
    /*  A MM_MCIEVENT message was issued by the Mwave FAX   */
    /*  driver.                                             */
    /*------------------------------------------------------*/
    switch(wParam)
    {
        case PHONE_EVENT_CALL_FAX:
            /*----------------------------------------------*/
            /*  Incoming call. Receive fax and notify user. */
            /*----------------------------------------------*/
            ReceiveFax(hWnd);
            NumFax++;   // counter for received fax's filenames
            break;
```

On receipt of this message, the application must initiate the receive operation of the Mwave Fax device driver as soon as possible. FAXAPP calls the ***ReceiveFax()*** function which performs this operation.

Step 2. Initiate the receive operation in the Mwave Fax device driver.

The Mwave Fax device driver begins to receive incoming fax data after the application calls the MCI_RECEIVE command. FAXAPP uses file names of the form "FAX??.TIF" to store received fax data. See"" on page 1-43 for more information on this file naming convention.

[faxops.c: ReceiveFax()]

```
void ReceiveFax( HWND hWnd )
{
    char    buffer[32];

    wsprintf( (LPSTR)buffer, "Fax%d.tif", NumFax);
    mciReceiveParms.lpstrFilename = (LPSTR)buffer;
    mciReceiveParms.dwCallback = (DWORD)hWnd;
    SetCursor(hcWaitCursor);
    uAppState = STATE_RECEIVE_SETUP;
    mciSendCommand( wMwaveFaxID,
             MCI_RECEIVE,
             MCI_NOTIFY | MCI_RECEIVE_FILE,
             (DWORD)lpmciReceiveParms );
    MessageBox(hWnd, (LPSTR)buffer, "RECEIVING FAX FILE", MB_OK);
}
```

Note that MCI_RECEIVE is called using the MCI_NOTIFY flag instead of the MCI_WAIT flag. This was done for the sole purpose of allowing our message procedure to track the machine state via the MM_MCINOTIFY message. Note that the FAXAPP state is set to `STATE_RECEIVE_SETUP` prior to calling MCI_RECEIVE. A transition from the receive setup state to the receive fax data state is handled by the message procedure.

After initiating the receive operation, the Mwave Fax device driver begins to receive the fax data from the remote fax machine. At the same time, MCI sends a MM_MCINOTIFY message (since MCI_RECEIVE was called with the MCI_NOTIFY flag specified) to indicate either successful completion or failure of the MCI_RECEIVE command. Our message procedure (see below) responds to a successful receive setup by changing the machine state to `STATE_RECEIVING_FAX` . If a failure occurred during receive setup, FAXAPP displays an error, hangs up the line, and resets the machine state to `STATE_IDLE` .

[fax.c: WndProc()]

```
case MM_MCINOTIFY:
    switch(wParam)
    {
        case MCI_NOTIFY_ABORTED:
        case MCI_NOTIFY_FAILURE:
            if( uAppState == STATE_DIALING )
                idResource = IDS_ERR_DIALING;
            else if( uAppState ==  STATE_RECEIVE_SETUP )
                idResource = IDS_ERR_RECEIVE;
            else
                idResource = IDS_ERR_UNKNOWN_STATE;
            ResourceMessageBox(hWnd, idResource, 0, "MM_MCINOTIFY", MB_OK);
            SetOnHook();
            uAppState = STATE_IDLE;
            break;

        case MCI_NOTIFY_SUCCESSFUL:
            if( uAppState == STATE_DIALING )
            {
                uAppState = STATE_SENDING_FAX;
                ResourceMessageBox(hWnd, IDS_MSG_SENDING_FILE, 0, szAppName,
       MB_OK);
            } else if( uAppState ==  STATE_RECEIVE_SETUP )
                uAppState = STATE_RECEIVING_FAX;
            break;

         default:
             break;
    }
    break;
```

Step 3. Respond to a change (either completion, status change, or error) to the receive operation.

After initiating the receive operation, the Mwave Fax device driver receives incoming fax data from the remote fax machine (in the background) without requiring any support from the application.

Upon receive completion, either successful or due to an error, the driver sends a MM_MCIEVENT event message of type PHONE_EVENT_CALL_TERMINATED to our application. This message is processed by our event handling procedure as follows:

[fax.c: WndProc()]

```
if( Message == uMCIMessage )
{
    LPMCI_EVENT_PARMS lpMciEventParms = (LPMCI_EVENT_PARMS)lParam;

    /*------------------------------------------------------*/
    /*  A MM_MCIEVENT message was issued by the Mwave FAX   */
    /*  driver.                                             */
    /*------------------------------------------------------*/
    switch(wParam)
    {
        case PHONE_EVENT_CALL_TERMINATED:
            /*----------------------------------------------*/
            /*  Call was terminated.  Check termination     */
            /*  code (in dwEventData) for cause.            */
            /*----------------------------------------------*/
            switch( lpMciEventParms->dwEventData )
            {
                case TERMINATION_NORMAL:
                    if( uAppState == STATE_SENDING_FAX )
                        idResource = IDS_MSG_SEND_OK;
                    else if( uAppState ==  STATE_RECEIVING_FAX )
                        idResource = IDS_MSG_RECEIVE_OK;
                    else
                        idResource = IDS_ERR_UNKNOWN_STATE;
                    break;

                case TERMINATION_UNEXPECTED:
                case TERMINATION_ERROR_XMIT:
                case TERMINATION_ERROR_RECV:
                    if( uAppState == STATE_SENDING_FAX )
                        idResource = IDS_ERR_SEND_FAIL;
                    else if( uAppState ==  STATE_RECEIVING_FAX )
                        idResource = IDS_ERR_RECEIVE_FAIL;
                    else
                        idResource = IDS_ERR_UNKNOWN_STATE;
                    break;

                default:
                    idResource = 0;
                    break;
            }
            if( idResource )
                ResourceMessageBox( hWnd, idResource, 0, "MM_MCIEVENT", MB_OK );
            /*----------------------------------------------*/
            /*  Hang up the phone                           */
            /*----------------------------------------------*/
            SetOnHook();
            uAppState = STATE_IDLE;
            break;
```

Upon receipt of the PHONE_EVENT_CALL_TERMINATED case of the MM_MCIEVENT message, FAXAPP displays either a success or failure message (using the context of the current machine state), hangs up the phone, and resets the machine state back to `STATE_IDLE` .

Receiving a FAX - OS/2

Receiving a fax using the FAX API and the Mwave FAX device driver requires the following steps:

Step 1. Respond to the MM_MCIEVENT message PHONE_EVENT_CALL_FAX
Step 2. Initiate the receive operation in the Mwave FAX device driver.
Step 3. Respond to a change (either completion, status change, or error) to the receive operation.

Let's look at how FAXAPP implements each of these steps.

Step 1. Respond to the MM_MCIEVENT message PHONE_EVENT_CALL_FAX

The Mwave FAX device driver notifies FAXAPP that an incoming fax call has been detected by sending the PHONE_EVENT_CALL_FAX type of MM_MCIEVENT message. This message is processed by our event handling procedure as follows:

faxevnts.c: ProcessEvent

```
void ProcessEvent(HWND hWnd, WORD wParam,
                  LPMCI_EVENT_PARMS lParam)
{
   char   buf[256];

   switch(wParam)
   {
     case PHONE_EVENT_CALL_FAX:
         /* incoming call - receive fax and notify user */
         ++NumFax;
         ReceiveFax(hWnd);
         break;
```

On receipt of this message, the application must initiate the receive operation of the Mwave FAX device driver as soon as possible. FAXAPP calls the **ReceiveFax** function which performs this operation.

Step 2. Initiate the receive operation in the Mwave FAX device driver.

The Mwave FAX device driver begins to receive incoming fax data after the application calls the MCI_RECEIVE command. FAXAPP uses file names of the form "FAX??.TIF" to store received fax data..

faxops.c: ReceiveFax

```
void ReceiveFax(HWND hWnd)
{
  char buff[256];
  LPSTR lpBuff = (LPSTR) buff;

  sprintf(buff,"c:\\Fax%d.tif",NumFax);
  mciReceiveParms.lpstrFilename = (char *) lpBuff;
  mciReceiveParms.dwCallback = (DWORD)hWnd;

  WinSetPointer(HWND_DESKTOP,
                WinQuerySysPointer(HWND_DESKTOP,
                SPTR_WAIT, FALSE));

   ulRC = mciSendCommand (wDeviceID, MCI_RECEIVE,
                          MCI_WAIT | MCI_RECEIVE_FILE,
                          (DWORD) lpmciReceiveParms,
                          mciCall++);

   MessageBox(hWnd, buff, "RECEIVING FAX FILE", MB_OK);
} /* End ReceiveFax */
```

After initiating the receive operation, the Mwave FAX device driver begins to receive the fax data from the remote fax machine. If a failure occurred during receive setup, FAXAPP displays an error and hangs up the line.

Step 3. Respond to a change (either completion, status change, or error) to the receive operation.

After initiating the receive operation, the Mwave FAX device driver receives incoming fax data from the remote fax machine (in the background) without requiring any support from the application.

Upon receive completion, either successful or due to an error, the driver sends a MM_MCIEVENT event message of type PHONE_EVENT_CALL_TERMINATED to our application. This message is processed by our event handling procedure as follows:

faxevnts.c:

```
void ProcessEvent(HWND hWnd, WORD wParam, LPMCI_EVENT_PARMS lParam)
{
   char   buf[256];

   switch(wParam)
   {
     case PHONE_EVENT_CALL_FAX:
         /* incoming call time to receive fax and notify user */
         ++NumFax;
         ReceiveFax(hWnd);
         break;

     case PHONE_EVENT_CALL_TERMINATED:
         /* notify user, enter into log                      */
         switch (lParam->dwEventData)
             {
             case TERMINATION_NORMAL:
               sprintf(buf, "Call completed normally");
               break;
             case TERMINATION_UNEXPECTED:
               sprintf(buf, "Call terminated unexpectedly");
               break;
             case TERMINATION_ERROR_XMIT:
               sprintf(buf, "Call terminated: transmit error");
               break;
             case TERMINATION_ERROR_RECV:
               sprintf(buf,"Call terminated: receive error");
             case TERMINATION_REQUESTED:
               sprintf(buf,"Call terminated: type requested");
               break;
             default:
               break;
             } /* switch                                     */
         /*  Hang up the phone                               */
         MessageBox(hWnd, buf, "Fax Informations", MB_OK);
         SetOnHook(hWnd);
         break;

     case PHONE_EVENT_CALLER_ID:
     case PHONE_EVENT_FAX_HEADER:
     case PHONE_EVENT_FAX_PAGE_COMPLETE:
     case PHONE_EVENT_FAX_PAGE_STATUS:
     case PHONE_EVENT_FAX_CONNECT:
     case PHONE_EVENT_LINE:
     case PHONE_EVENT_HANDSET:
     case PHONE_EVENT_CALL_PROGRESS:
       break;

     default:
         break;
     }
 }
```

Upon receipt of the PHONE_EVENT_CALL_TERMINATED case of the MM_MCIEVENT message, FAXAPP displays either a success or failure message and hangs up the phone,

This concludes the steps required to receive fax data.

Converting Fax Document Files to/from DIB format

As mentioned previously, the FAX API specifies several different Fax Document File formats which might be supported by a compliant fax driver. In the case of the Mwave Fax device driver, the supported Fax Document File format is TIFF Class F. However, in the Microsoft Windows environment, it is much more convenient to view, print, and edit image data in Device Independent Bitmap (DIB) format. For this reason, the FAX API supports the command MCI_CONVERT which enables conversion from Fax Document File format to DIB format and visa versa.

FAXAPP uses the MCI_CONVERT command to enable sending of DIB files and viewing of TIFF Class F files by first converting a given file into the correct format. To illustrate the use of MCI_CONVERT, we'll use the example of sending a BMP file (a type of DIB file) to a destination fax machine. See "Sending a Fax" on page 65 for an overview of the fax send operation. Recall that the Mwave Fax device driver can only send files in TIFF Class F format, so we must first convert the BMP file into TIFF Class F format.

FAXAPP uses the ***ConvertBMP2TIF()*** function to convert a BMP format source file to a TIFF Class F format destination file using the MCI_CONVERT command as follows:

[faxops.c: ConvertBMP2TIF()] - Windows

```
        int ConvertBMP2TIF(HWND hWnd, LPSTR SrcFileName)
        {
            DLGPROC lpfnGetDest;
            char    buffer[32];

            wFileConvertType = BMP_TO_TIF;
```

```
/*-----------------------------------------------------------------*/
/*  Prompt user for destination file name                          */
/*-----------------------------------------------------------------*/
lpfnGetDest = (DLGPROC)MakeProcInstance((FARPROC)GetDest_DlgProc, hInst);
if (!DialogBox(hInst, "DestinationFile", hWnd, lpfnGetDest))
{
    FreeProcInstance(lpfnGetDest);
    return( FALSE );
}
lstrcpy( (LPSTR)buffer, (LPCSTR)DestFile );
mciConvertParms.lpstrDestFilename = (LPSTR)buffer;
mciConvertParms.dwDestFormat = MCI_FAX_CONVERT_FMT_DEVFAX;
mciConvertParms.lpstrSrcFilename = SrcFileName;

SetCursor(hcWaitCursor);
dwReturn = mciSendCommand(  wMwaveFaxID,
                MCI_CONVERT,
                MCI_WAIT | MCI_CONVERT_SOURCE_FILE |
                MCI_CONVERT_CREATE |
                MCI_CONVERT_DESTINATION_FILE |
                MCI_CONVERT_DESTINATION_FORMAT,
                (DWORD) lpmciConvertParms);
  :
```

faxops.c: ConvertBMP2TIF - OS/2

```
int ConvertBMP2TIF(HWND hWnd, WORD wDeviceID,
                   LPSTR SrcFileName)
 {
  static HANDLE hBuff;
  LPSTR  lpBuff;
  char buf[255];
  char messagestring[255];

    /* 1.  Get Destination File Name */
    hBuff = (HANDLE) malloc(32);
    lpBuff = (LPSTR) hBuff;

    if (!WinDlgBox(HWND_DESKTOP, hWnd,
        GetDest_DlgProc, 0, DestinationFile, NULL))
    {
      return FALSE;
    }
    strcpy((CHAR *)lpBuff, DestFile);
    mciConvertParms.dwCallback = (DWORD)hWnd;
    mciConvertParms.lpstrDestFilename = (char *) lpBuff;
    mciConvertParms.dwDestFormat =
       MCI_FAX_CONVERT_FMT_DEVFAX;
    mciConvertParms.lpstrSrcFilename = (char *) SrcFileName;

    WinSetPointer(HWND_DESKTOP,
                  WinQuerySysPointer(HWND_DESKTOP,
                  SPTR_WAIT, FALSE));

    ulRC = mciSendCommand(wDeviceID, MCI_CONVERT,
                   MCI_WAIT | MCI_CONVERT_SOURCE_FILE |
                   MCI_CONVERT_CREATE |
                   MCI_CONVERT_DESTINATION_FILE |
                   MCI_CONVERT_DESTINATION_FORMAT,
                   (DWORD) lpmciConvertParms, mciCall++);

    free((void *) hBuff);

    if (ulRC)
    {
      /* Error, unable to convert file  */
      mciGetErrorString(ulRC, (int *)messagestring,
                        sizeof(messagestring));

      sprintf(buf,
              "ERROR: %d:  Unable to convert image file. %s",
               (LOWORD(ulRC)),        messagestring );
      MessageBox(hWnd, buf, NULL, MB_OK);
      return FALSE;
    }
    strcpy((CHAR *)lpFileName,(CHAR *)DestFile);
    MessageBox(hWnd, "Bitmap file converted to TIFF",
               szAppName, MB_OK);
    return TRUE;
 }  /* end ConvertBMP2TIF */
```

Note that MCI_CONVERT is called with the MCI_CONVERT_CREATE flag. This flag specifies that the destination file should be created if it doesn't exist. If the destination file does exist, its contents are destroyed and overwritten with the converted data (the same effect as using the MCI_CONVERT_OVERWRITE flag).

Alternatively, converted data can be inserted into (or appended onto) an existing destination file. Set the `dwDestFrom` field of the MCI_CONVERT_PARMS structure to the document page number (starting at page zero) where you want the converted data to be written into the destination file. You'll also need to include the MCI_CONVERT_DESTINATION_FROM flag and remove the MCI_CONVERT_CREATE flag in the MCI_CONVERT call.

Closing the Mwave Fax Device Driver

Before exiting your application, you should always close the Mwave Fax device driver. Closing the driver frees memory, processor, and connection resources on the Mwave board, making them available for use by other Mwave applications.

When closing the FAX driver, always assign the window procedure handle to mciGenericParms.dwCallback prior to issuing the MCI_CLOSE. Failure to do so will result in erratic behavior. In FAXAPP, CloseFax looks like this:

[faxops.c: CloseFax()] - Windows

```
        MCI_GENERIC_PARMS   mciGenericParms;

        mciGenericParms.dwCallBack=(DWORD)hWnd;
        mciSendCommand( wMwaveFaxID,     // device ID
              MCI_CLOSE,   // MCI command
              MCI_WAIT,    // flags
                    (DWORD)(LPVOID) &mciGenericParms);
```

faxops.c: CloseFax - OS/2

```
void CloseFax(HWND hWnd )
{
    MCI_GENERIC_PARMS   mciGenericParms;

    mciGenericParms.dwCallback = (DWORD)hWnd;
    mciSendCommand( wDeviceID,
                    MCI_CLOSE,
                    MCI_WAIT,
                    (DWORD)&mciGenericParms, mciCall++);
}
```

Summary

FAXAPP provides a simple example of using the basic send and receive capabilities of the FAX API and the Mwave Fax device driver. Using the programming techniques outlined in this chapter, you should be able to add additional capabilities, such as providing real-time send and receive status information, into your own Mwave fax application. Be sure to check the FAX API Reference for a complete description of the available capabilities.

Chapter 5 - Telephone Answering Machine (TAM) Services

This chapter describes the telephony services available to application developers for the purpose of developing Mwave compatible TAM based applications.

Mwave TAM Architecture

TAM functionality is provided with two separate MCI device drivers: Phone Line and Message. The Phone Line driver is used for all operations involving the telephone line. This includes making a call, answering a call and remotely (i.e. via a telephone call) reviewing or recording a message. The Message driver is used only for those operations which do not involve use of the phone line (i.e. locally reviewing or recording messages). This multiple driver approach allows simultaneous telephone answering and message review. TAM can answer an incoming phone call at the same time a user is reviewing (i.e. listening to) his messages.

TAM Programming Environment

For many MCI devices, including TAM, the MCI controls are similar to those of a tape recorder (for example; record, play, stop, pause, and seek). The MCI command message API specification for TAM applications begins with this conventional design, and adds enhancements such as voice compression and speakerphone operation.

The Phone Line driver can record and play through the telephone line only. (for local message record and review, the Message driver is used). The Phone Line driver can additionally be used to connect the phone line to the handset (as with a standard telephone) or the audio port (speaker/microphone).

TAM resembles a media recorder and player which can be connected to various telephony related voice channels. Each of the telephony related voice channels has an audio input and output driver. All channels "connected" to the media are used for play or record operations. If multiple channels are connected to the media, they are also connected to each other, even if no play or record operation is in progress. The audio channels defined for use with the TAM device drivers are as follows:

• MCI_TAM_AUDIO (speaker & microphone)	**MSG**
• MCI_TAM_HANDSET	**MSG**
• MCI_TAM_PHONELINE	**PL**
• MCI_TAM_AUDIO_PHONELINE (speakerphone)	**PL**
• MCI_TAM_HANDSET_PHONELINE (standard phone operation)	**PL**
• MCI_TAM_SPEAKER_PHONELINE (answering machine w/ call screening)	**PL**

Various telephony operations are achieved by configuring or "connecting" the TAM drivers via calls to MCI_SET. For the Message driver, this includes the following:

- Connect MCI_TAM_AUDIO to record a new announcement from the microphone or review messages on the speaker.
- Connect MCI_TAM_HANDSET to record a new announcement or review messages through the local handset or desk telephone.

For the Phone Line driver, this includes the following:

- Connect MCI_TAM_PHONELINE to play an announcement, record a new message, or review messages from a remote telephone. This connection is required for all operations involving an outside phoneline.
- Connect MCI_TAM_AUDIO_PHONELINE for speakerphone operation. Note that with this operation, the media recorder/player is not available. No record or playback can be done. Setting speakerphone operation disables call discrimination based on calling tones. Also DTMF key detection is also disabled.
- Connect MCI_TAM_HANDSET_PHONELINE to allow the normal use of the desk phone.
- Connect MCI_TAM_SPEAKER_PHONELINE to allow the user to screen calls. Note that the microphone will not be connected.

As suggested above, both drivers can use the audio port (speaker/microphone) and telephone handset, but they cannot share them. **When either of these devices are in use by one of the drivers it is unavailable to the other**. Also, the handset must be available when opening the Phone Line driver and the audio port must be free when opening the Message driver. **Programmers must track the device connection status of the two drivers to avoid device conflict errors**.

Once again, the actual operation of the TAM device is similar to a physical answering machine. The programming model incurs some complexity when implemented using a message driven architecture, but its similarities to a tape recorder remain. See the code example at the end of this chapter for more details.

TAM File Formats

Sound files are notorious for their size. One of the more significant problems with the accumulation of large amounts of audio data is data storage. Although OS/2 MMPM and Microsoft Windows wave files support several different data formats, most MCI devices support only uncompressed PCM.

To efficiently deal with data storage, the TAM API specification allows a device driver to support a device dependent format (custom format tag), which allows the device driver to store data files in the most optimum format available. This removes the burden of data compression from the application writer, and at the same time, allows for increased functionality on the part of the device driver.

Support of the custom format tag is optional under this specification, and the application can still choose to use the standard PCM wave file format. For those device drivers which do not support the standard PCM format for play and record operations, a conversion command is available to convert standard wave files to the custom format used by the device. In general, if a device supports a custom format tag, it is to the advantage of the application (in terms of file storage) to use the custom tag in place of the standard file tag. The custom format requires about 4K bytes per second of audio. (**Note**: Standard PCM wave format is supported by version 3.1 and above of the TAM drivers).

The programming example described at the end of this chapter uses the custom format tag and does not need to perform any file format conversions.

Command Message Summary

The following table lists the MCI commands, most of which are used by the sample application. For more information on the actual command messages, see Chapter 7 of this document.

MCI Command	Description
MCI_CLOSE	Close the device driver
MCI_CONVERT	Convert between device dependent and device independent files
MCI_DIAL	Dial the phone
MCI_GETDEVCAPS	Get capabilities of the device
MCI_INFO	Get device string identifier
MCI_LOAD	Load a *voice* file for playing
MCI_OPEN	Open the device driver
MCI_PAUSE	Pause the *voice* stream play or record
MCI_PLAY	Play a *voice* file
MCI_RECORD	Record a *voice* file
MCI_RESUME	Resume a paused *voice* stream
MCI_SAVE	Save a recorded *voice* stream
MCI_SEEK	Change current position of the media
MCI_SET	Configure the device
MCI_STATUS	Query device configuration
MCI_STOP	Stop a *voice* stream

Table 5-1: TAM Driver MCI Command Messages

Programmers familiar with MCI will note the new command messages; **MCI_CONVERT** and **MCI_DIAL**. For most TAM related applications, it is not necessary to make use of the **MCI_CONVERT** message. (MCI_CONVERT is supported in TAM drivers version 3.1 and above.)

Event Message Summary

The following table is a summary of the MCI event messages which may be received from the TAM device driver. Most of these event messages are used in the sample application. These messages are described in more detail in Chapter 7 of this document.

Event Message	Description
PHONE_EVENT_ADVANCED_RING	Advanced format ring notification
PHONE_EVENT_CALL_PROGRESS	Call progress state has changed
PHONE_EVENT_CALL_TAM	Received an incoming voice telephone call
PHONE_EVENT_CALL_TERMINATED	Call has been terminated (supplies termination code)
PHONE_EVENT_CALLER_ID	Caller ID string detected (supplies completion status)
PHONE_EVENT_DISTINCTIVE_RING	Distinctive ring detected
PHONE_EVENT_HANDSET	Change in handset status (supplies handset status)
PHONE_EVENT_HANDSET_KEY	Keypad press from handset (supplies character)
PHONE_EVENT_LINE	Change in phone line hook status (supplies status)
PHONE_EVENT_LINE_KEY	Keypad press from phone line (supplies character)
PHONE_EVENT_RING	Telephone ring status change (supplies ring on/off)

Table 5-2: TAM Driver Event Messages

Event messages are received by the event message handler which is declared to the TAM device driver using **MCI_SET**. For more information on the event handler, and how it relates to the TAM device driver, refer to Chapter 3 on Telephony Services.

Developing an Mwave TAM Application

This section describes how to develop an application, which calls the TAM API to access the Mwave TAM device drivers, providing telephone answering capabilities.

Handset/Speakerphone Interactions

The following describes the interactions and application source required for changing from speakerphone to handset and back again.

Scenario:
Assume the application is connected to speakerphone, a call has come in and the application has gone off hook.

User Lifts Handset:
-App receives PHONE_EVENT_HANDSET with dwHandsetStatus = 1 (off hook)
-App issues set connect to MCI_TAM_HANDSET_PHONELINE
-App issues an onhook

Note: The application should issue on hook because connecting handset creates 2 extensions on the phone line. When the application issues the “onhook” the line will only have one extension. (Disconnecting TAIO from the line allows the handset to act as a normal phone) . For example, if the application had not done the onhook after connecting to handset then when the user put the phone down the call would still be offhook (because TAIO is still connected).

User Requests Speakerphone:
-App issues off hook (turn on second extension before connecting to that extension)
-App issues set phoneline/audio

User wants to do conversation record:
The app must be in either normal phone or speakerphone mode.
-App issues off hook (if in normal phone mode the app needs to reconnect the computer (TAIO) to the phoneline by issuing and off hook) This will allow the remote side to be recorded.
-App issues Mciphone record (only voi supported for conversation record)

Sample Application Definition

The first step in developing a solid application, is the definition of its intended functionality. For the purpose of demonstration, the application defined here is modeled after a simple telephone answering machine. Defined functionality includes: playing announcements, recording messages, and reviewing messages from the control panel or a remote telephone. Below is a diagram of how an incoming telephone call is handled.

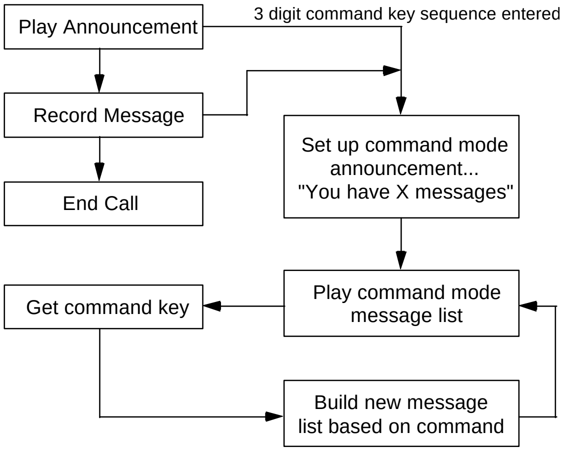
Play Announcement
3 digit command key sequence entered
Record Message
End Call
Set up command mode announcement... "You have X messages"
Get command key
Play command mode message list
Build new message list based on command

Figure 5-1: Answering Machine Model

Once placed in command mode, the touch-tone keypad of the remote telephone can be used for an unlimited number of functions. Using keypad entries and voice menus, the remote telephone may be used for anything from reviewing messages to requesting FAX documents. Keypad commands can consist of one or multiple key entries. A list of the actual keypad commands used in the sample application can be found later in this chapter.

The following sections assumes you are familiar with the operation of the TAM sample application included on the companion diskette. See "Using the TAM Sample Application" on page 1-85 for complete details.

Sample Code Design

The primary goal of this application example is to illustrate the operation of an event (message) driven TAM device, which is able to execute as a background task.

The TAM device has been defined as a finite state machine, which is mainly driven by the **MM_MCINOTIFY** message issued on a MCI command completion. To implement the state concept, a single variable is defined:

```
        short wTamState;           \\ State of TAM device
```

This variable can have any of the following states:

- TS_IDLE
- TS_PLAY_ANNOUNCEMENT
- TS_RECORD_MESSAGE
- TS_COMMAND_MODE
- TS_PLAY_MESSAGE
- TS_REMOTE_PLAY
- TS_ARCHIVE_PLAY

These states correspond roughly to the diagram shown in the previous section. The 'idle' state, not shown in the diagram, is the state where the application is waiting to answer a call. The 'COMMAND_MODE' state is the state of the TAM device when in message review mode. The state of the TAM device is used to determine the next operation after receipt of successful completion messages in the *NOTIFY* section of the event handler.

Some of the additional variables used in the program are listed below. Most of these values are user definable, and are read from the application INI file when the program is first executed.

Global Variables	(* Stored in INI file)
wActiveMessages	Number of active messages on disk
* wCommandCode	digit command code
wCurrentMessage	Index of current message being played
* wMonitor	Incoming call monitoring (0-Off 1-On)
wMsgDate	Current message data in MSDOS format
* dwMsgIndex	Index of next message to be saved
wMsgTime	Current message time in MSDOS format
wNewMessages	New messages since last MSG review flag
wPlaySpeed	Message playback speed (0-Slow 1-Norm 2-Fast)
* wRingCount1	Ring count with messages
* wRingCount2	Ring count with out messages
wTamState	State of TAM system (See TAM States above)
* wVolume	Speaker volume level
wExclusive	Set to '1' when app can not answer the telephone
wBFE	Error code used by error_box()

Global Strings	(* Stored in INI file)
CurrentTimeStamp	String containing time stamp of current file
MsgFileSpec	Full path & filename of current message file
* MsgPath	Path to message file storage on system

Table 5-3: Selected Global Variables and Strings

Because this example is targeted to show TAM operation and not necessarily efficient file management, the file storage system for the voice messages is defined using a simple prefix and suffix system. Under this system, all active messages contain a single letter prefix ('M'), a seven-digit suffix, and the extension '.TAM'. For example, the first message stored in this system is 'M0000000.TAM'.

Recording and Playing the Announcement - The announcement file is stored as 'ANNOUNCE.TAM' in the message directory. When recording a new announcement, a temporary filename is used until the application user chooses to accept the new announcement.

Recording a New Message - New messages are recorded to a filename consisting of the letter 'M', followed by a seven-digit suffix which is the current value of **dwMsgIndex** (master message index). The extension '.TAM' is added to the filename, and the master message index is incremented. This master message index is stored in the INI file, and is never reset. This allows for the creation of 10 million unique message filenames.

Playing Active Messages - To play the active messages, all files are searched and those matching the 'M???????.TAM' pattern are sorted alphabetically (also chronologically) and stored in a list. The list can then be 'walked' forwards or backwards.

Erasing a Message - To erase a message, the message file is simply deleted.

The TAM device driver is obviously not dependent on any single file management system, and an application programmer might want to use a more sophisticated system in the implementation of a more complex TAM application.

TAM State Machine Operation

Most programmers familiar with MCI agree that programming to the MCI interface is not a difficult task. The main 'trick' involved in programming a TAM application is writing the application in an event driven fashion, so that when the system is idle, it consumes minimal processor time.

As mentioned above, the application is implemented as a state machine. The application proceeds from state to state based on messages received by the event handler routine. Although most state changes occur as a result of the **MM_MCINOTIFY** message, the **MM_MCIEVENT** message is also of interest. Most of the application logic is executed based on event messages.

MCI Event Message Handling

MCI event messages are sent as a direct result of an external telephony event detected by the device driver. All event messages are for notification purposes only, and the application is not required to perform any action to 'handle' these events. The event messages are very useful however for writing event driven applications. The messages that are handled in the TAM application example are listed below:

- **PHONE_EVENT_CALL_TAM -** This message initiates the TAM state machine execution. The application starts by playing the announcement. Before the call has come in, the application has already connected to the phone line (via MCI_SET_CONNECT) and loaded the announcement (via MCI_LOAD). (Both these operations are performed by the **ExitExclusive()** function.) At this point, the application executes an MCI_PLAY call to begin playing the announcement. The MCI_NOTIFY flag is supplied with the PLAY call so that the event handler is notified when the play is complete, and it is time to start incoming message recording. The TAM state (**dwTamState**) is set to **TS_PLAY_ANNOUNCEMENT**.

- **PHONE_EVENT_CALL_TERMINATED -** Here a call has been terminated because the caller has hung up the telephone. If a message record was in progress, the message is saved. The **ExitExclusive()** function is called to stop any current operation, connect the driver to the phone line, load the announcement file, and set the TAM state (**dwTamState**) to **TS_IDLE**. At this point, the application is ready to answer another call.

- **PHONE_EVENT_CALL_PROGRESS -** This message is used to detect a 'hang-up' condition that does not produce a 'CALL_TERMINATED' message. If dialtone is detected, the **CallTerminated()** is called to perform the same actions produced by a 'CALL_TERMINATED' message.

- **PHONE_EVENT_HANDSET -** This message indicates that the handset on the local telephone has been either picked-up, or replaced. Picking up the handset causes different effects at different times. When the system is IDLE, picking up the handset auto-switches from speakerphone to normal phone mode. When reviewing messages to the speaker, picking up the handset disables the speaker and continues play to the handset. If the handset is picked up while the system is recording a message, the record operation is aborted and the system is placed into normal phone mode (connecting the phone line to the handset).

  **** This functionality is not implemented in the sample application!****

- **PHONE_EVENT_LINE_KEY -** This message sends the ASCII character of the telephone key which has been pressed on the incoming telephone line. The actions taken on receipt of this key vary according to the current state of the TAM device.

  If the TAM state machine is in command mode, this message is the potential gateway into remote review mode. It tracks the keys that have been pressed, and if the correct 3 digit sequence has been entered, the current operation is stopped, the TAM state is set to **TS_REMOTE_PLAY**, and the remote review announcement is played ("You have messages...").

  If the TAM state machine is already in remote review mode when this message is received, the key is interpreted as a new command. Key commands can be used to skip, erase, save, repeat, and control the playback of messages stored on the system.

  A second state variable (**wRemoteState**) is used to track the state of the remote message review.

MCI Notification Message Handling

The MCI notification message system (**MM_MCINOTIFY**) is the standard method for MCI to notify an application that a driver action has been completed. For the purposes of our application, we need to be notified when a message play or record command has been completed. In most cases, the successful completion of a play or record operation requires the execution of another event, and sometimes advances the state of the TAM state machine.

The types of notification messages possible, as well as how they are treated, is as follows:

- **MCI_NOTIFY_ABORTED -** There are cases when a play or record operation will be aborted. The most common being when the user picks-up the telephone handset in order to talk 'live' to the caller. Since aborting an operation is not a normal part of the TAM state machine, this command does not examine or alter the state of the TAM device. *** *This functionality is not implemented in the Mwave TAM driver! ****

- **MCI_NOTIFY_SUCCESSFUL -** The successful completion of an event is the only automatic method to drive the state change in the TAM state machine. The application should use the NOTIFY flag only for API calls requiring significant execution time. These include playing and recording voice files. Below is a state change table based on the completion of a play or record operation.

Current State	Next State
TS_PLAY_ANNOUNCEMENT	TS_RECORD_MESSAGE
TS_RECORD_MESSAGE	TS_IDLE
TS_PLAY_MESSAGE	TS_COMMAND_MODE
TS_ARCHIVE_PLAY	TS_COMMAND_MODE
TS_COMMAND_MODE	TS_COMMAND_MODE
TS_REMOTE_PLAY	TS_REMOTE_PLAY

Table 5-4

For example, when the machine is in the PLAY ANNOUNCEMENT state, the completion message indicates that it is time to start recording an incoming message. The message handler starts the record operation. Because detecting the command mode code digit entry is not handled by the event handler, this routine assumes that the RECORD MESSAGE state always follows completion of the PLAY ANNOUNCEMENT state. Similarly, the completion of the RECORD MESSAGE state is always followed by the IDLE state and call termination.

- **MCI_NOTIFY_SUPERSEDED -** This message should not occur under normal operating conditions since all play and record operations using the NOTIFY flag are invoked as a result of the notification that the previous play or record command has been completed. In the event that a record or play operation is aborted due to user interruption, the **MCI_NOTIFY_ABORT** message will be received. This message is treated the same as the successful notification message.

- **MCI_NOTIFY_FAILURE -** This message is treated the same as the successful notification message. For debug purposes, it generates an error message, but the example program supplied with the companion diskette does not attempt to correct for errors.

Remote Message Review

As mentioned above, the remote message playback feature of the application uses a separate state system than the main program logic. When in remote message review mode, the TAM state (**wTamState**) is set to **TS_REMOTE_PLAY**, and the remote state variable (**wRemoteState**) determines the state of the remote playback operation. Possible remote state values are as follows:

- RS_WAIT
- RS_WAITING
- RS_PLAYMENU
- RS_PLAYEND

These states determine the current action, or the next action to be taken when the current play is complete.

When the command key sequence for remote play is first entered, and messages are available, **wRemoteState** is set to RS_WAIT, and a greeting file is played (*"You have messages..."*). When the MCI_NOTIFY message is received indicating the end of the greeting, **wRemoteState** is changed to

RS_WAITING, and the system does nothing until a key is pressed. If no key is pressed in a set amount of time, the application disconnects the telephone and resets the system.

If there are no messages available when the command sequence is entered, **wRemoteState** is set to RS_PLAYEND, and an exit message is played (*"You have no messages..."*). When the MCI_NOTIFY message is received indicating the end of the exit greeting, the telephone is disconnected, and the system is reset.

For normal message reviewing, **wRemoteState** is set to RS_PLAYMENU before a the message is played indicating that when the play has completed that the system should then play the verbal menu greeting (*"Press 1 for next, 2 for erase..."*). Before this menu is played, **wRemoteState** is set to RS_WAIT, indicating that the system should wait for a key press after playing the verbal menu.

The physical transition table for these states is as follows:

Current State	**Next State**
RS_WAIT	RS_WAITING *(wait for a key)*
RS_WAITING	*undefined(continue waiting)*
RS_PLAYMENU	RS_WAIT *(play verbal menu)*
RS_PLAYEND	*Telephone disconnect & system reset*

Table 5-5

Sample Application Source Code

To better illustrate the concepts of the TAM state machine introduced in this section, the source code to a sample application using the state machine is provided in the "\tam" subdirectory on the companion diskette.

This section documents the structure of the sample application source code, and explains some of the more interesting routines.

Source Code Organization

The TAM sample application is more complex than the average sample application. The purpose of this is to fully demonstrate all the available functionality of the TAM MCI device drivers. The following is a short synopsis of the files included on the companion diskette:

Source file	**Description**
MAKEFILE	Application makefile (for use with MS NMAKE.EXE)
MCIFTDD.H	Mwave MCI FAXTAM include file
TAM.H	TAM include - Global variable references
TAM.DEF	Windows definition file for use with the linker
TAM.RC	TAM resource source file. Contains system menus & dialog box definitions
TAMDEFS.C	Counterpart of TAM.H, containing global variable definitions
TAM.C	Program entry point. Contains initialization logic, and user interface code.
TAMFST.C	Main TAM state logic. Contains event handler & the majority of code which actually commands the MCI TAM driver.
DIALOG.C	Contains dialog procs for controlling all the dialog boxes used in the TAM application.

Table 5-6

In addition to the above, the diskette contains files with a .TAM extension. These are audio files used during the operation of the sample.

Initializing the TAM driver environment

A startup example is supplied in the chapter on Telephony Services. The function shown below has been taken from the startup example, and modified to perform a simple TAM environment initialization. The steps to initialize the driver include:

1. Open the driver and register the event message handler.
2. Verifying minimum TAM capabilities (CAN_PLAY, CAN_RECORD, and CAN_SAVE). Exit if any are not supported.
3. Check for optional speakerphone capability.
4. Check the supported file formats, and use the custom format tag when available.
5. Set the TAM call filter to have TAM auto answer incoming voice calls and route them to our application. Exit if another application is screening voice calls.

These initialization steps have been simplified in the sample code below, because we know the capabilities of the Mwave driver. This code would have to be adjusted for a more complex TAM application.

Initializing the TAM Driver - Windows

```
// InitDriverEnv
//
// This routine is called to initialize the TAM driver environment
//
static InitDriverEnv()
{
    // Open the MCI TPL Driver
    mciOpenParms.dwCallback = hEventHandler; // Always required on OPEN and CLOSE
    mciOpenParms.lpstrDeviceType = "Mwavetpl";
    if( dwBFE = mciSendCommand(0,MCI_OPEN,MCI_WAIT | MCI_OPEN_TYPE,
                                       (DWORD)(LPVOID)&mciOpenParms) )
    {
        error_box();
        return(0);
    }

    // Get the device ID & register the Event Handler for TPL so incoming phone
    //  calls are sent to the application's event handler
    wTplDeviceID = mciOpenParms.wDeviceID;
    mciSetParms.dwCallback = hEventHandler;
    mciSetParms.dwItem     = MCI_TAM_SET_EVENT_HANDLER;
    mciSetParms.dwSetData  = hEventHandler;
    mciSendCommand( wTplDeviceID,MCI_SET,MCI_WAIT | MCI_SET_ITEM,
                                       (DWORD)(LPVOID) &mciSetParms);

    // Open the MCI TPS Driver
    mciOpenParms.dwCallback = hEventHandler;
    mciOpenParms.lpstrDeviceType = "Mwavetps";
    if( dwBFE = mciSendCommand(0,MCI_OPEN,MCI_WAIT | MCI_OPEN_TYPE,
                                       (DWORD)(LPVOID)&mciOpenParms) )
    {
        error_box();
        //although not in sample, should close TPL here
        return(0);
    }

    // Get the device ID & register the Event Handler for TPS
    wTpsDeviceID = mciOpenParms.wDeviceID;
    mciSetParms.dwCallback = hEventHandler;
    mciSetParms.dwItem     = MCI_TAM_SET_EVENT_HANDLER;
    mciSetParms.dwSetData  = hEventHandler;
    mciSendCommand( wTpsDeviceID,MCI_SET,MCI_WAIT | MCI_SET_ITEM,
                                           (DWORD)(LPVOID) &mciSetParms);

    // Set to auto answer incoming phone calls
    mciSetParms.dwItem    = MCI_TAM_SET_CALL_FILTER;
    mciSetParms.dwSetData = 1;
    mciSendCommand( wTplDeviceID,MCI_SET,MCI_WAIT | MCI_SET_ITEM,
                                            (DWORD)(LPVOID) &mciSetParms);

    return(wTplDeviceID);
}
```

NOTE: With Windows, the handle of the window procedure responsible for processing MM_MCINOTIFY messages**MUST** be specified by assigning it to `mciOpenParms.dwCallback` prior to calling the MCI_OPEN command, regardless of whether the MCI_WAIT or MCI_NOTIFY flag is specified in the MCI_OPEN call. Failure to do so when using versions earlier than 2.1 will result in erratic behavior of the device driver.

Initializing the TAM Driver - OS/2

tam.c: InitDriverEnv

```
// InitDriverEnv
//
// This routine is called to initialize the TAM driver
//
static int InitDriverEnv(void)
{•
    // Open the MCI TPL Driver
    mciOpenParms.dwCallback = hEventHandler;
    mciOpenParms.lpstrDeviceType = (INT *) "Mwavetpl";

 dwBFE = mciSendCommand(0,
                        MCI_OPEN,
                        MCI_WAIT | MCI_OPEN_TYPE,
                        (DWORD)&mciOpenParms,
                        mci_cmd_ctr++);

    if( dwBFE )
    {
        error_box();
        return(0);
    }

    // Get the device ID & register Event Handler for TPL

    wTplDeviceID = mciOpenParms.wDeviceID;
    mciSetParms.dwCallback = hEventHandler;
    mciSetParms.dwItem     = MCI_TAM_SET_EVENT_HANDLER;
    mciSetParms.dwSetData  = hEventHandler;

    mciSendCommand( wTplDeviceID,
                    MCI_SET,
                    MCI_WAIT | MCI_SET_ITEM,
                    (DWORD)&mciSetParms, mci_cmd_ctr++);

    // Open the MCI TPS Driver

    mciOpenParms.dwCallback = hEventHandler;
    mciOpenParms.lpstrDeviceType = (INT *)"Mwavetps";

    dwBFE = mciSendCommand(0,
                           MCI_OPEN,
                           MCI_WAIT | MCI_OPEN_TYPE,
                           (DWORD)&mciOpenParms,
                           mci_cmd_ctr++);

    if( dwBFE )
    {
        error_box();
        return(0);
    }
    // Get the device ID & register Event Handler for TPS

    wTpsDeviceID = mciOpenParms.wDeviceID;
    mciSetParms.dwCallback = hEventHandler;
    mciSetParms.dwItem     = MCI_TAM_SET_EVENT_HANDLER;
    mciSetParms.dwSetData  = hEventHandler;

    mciSendCommand( wTpsDeviceID,
                    MCI_SET,
                    MCI_WAIT | MCI_SET_ITEM,
                    (DWORD)&mciSetParms,
                    mci_cmd_ctr++);

    // Set to receive TAM phone calls

    mciSetParms.dwItem    = MCI_TAM_SET_CALL_FILTER;
    mciSetParms.dwSetData = 1;

    mciSendCommand( wTplDeviceID,
                    MCI_SET,
                    MCI_WAIT | MCI_SET_ITEM,
                    (DWORD)&mciSetParms,
                    mci_cmd_ctr++);

    return(wTplDeviceID);
}
```

Implementation of the Event Handler

The event handler routine is the core of the TAM state machine. All state changes are a result of a *NOTIFY* or *EVENT* message sent to this routine. Note that although this routine has been isolated into its own procedure, the event handler code could be easily merged into the main window procedure, eliminating the need to create a separate window. It is also possible to implement a design where *NOTIFY* and *EVENT* messages are posted to different message procedures in the same application.

This module is the heart of the TAM application, and is the key for understanding the various operations of the TAM state machine.

Handling Events - Windows

```
//
/EventHandler
//
//This function is called whenever a message is sent from the MCI TAM
//driver. These messages drive new states of the TAM state machine.
//
long FAR PASCAL EventHandler(hWnd, message, wParam, lParam)
HWND hWnd;
unsigned message;
WPARAM wParam;
LPARAM lParam;
{
static UINT  uMCIMessage = 0xffff; // Initialize to invalid value
static short wKeys[3];        // Last 3 keys entered
static short wQuiet;         // Count for QUIET messages
static short wKeysPressed;      // Count for 3 key command
static short wCmdKey;        // Flag for 5-x play ctrl
unsigned short wEvent;
unsigned long   dwEventData;

switch (message)
{
case WM_CREATE:
       // Register the message we wish to look for
       uMCIMessage = RegisterWindowMessage("MM_MCIEVENT");
       break;

case MM_MCINOTIFY:
       // *** Received a NOTIFY message indicating earlier call
       //     w/ MCI_NOTIFY has completed
       switch( wParam )
            {
             case MCI_NOTIFY_FAILURE:
```

```
            case MCI_NOTIFY_SUCCESSFUL:
            case MCI_NOTIFY_SUPERSEDED:
            case MCI_NOTIFY_ABORTED:
                switch( wTamState )
                  {
                   case TS_COMMAND_MODE:
                   case TS_PLAY_MESSAGE:
                       PlayComplete();
                       break;

                   case TS_REMOTE_PLAY:
                       ContinueRemote();
                       wQuiet = 0;
                       break;

                   case TS_PLAY_ANNOUNCEMENT:
                       RecordMessage();
                       wQuiet = 0;
                       break;

                   case TS_RECORD_MESSAGE:
                       SaveMessage();
                       break;

                   case TS_ARCHIVE_PLAY:
                       PlayComplete();
                       SendMessage(hMainWnd,WM_COMMAND,IDM_ARCHIVE,0l);
                       break;

                   default:
                       break;
                  }
                break;
           }
       break;

default:
       if( message == uMCIMessage )
           {
            // *** Received an EVENT message ***

            // Isolate the message parameters
            MCI_EVENT_PARMS far *mep = (MCI_EVENT_PARMS far *)lParam;
            wEvent   = LOWORD( mep->dwDataParam1 ); // or wParam
            dwEventData = mep->dwEventData;

            switch( wEvent )
               {
                case PHONE_EVENT_CALL_TAM:
                   wKeysPressed = 0;
                   AnswerCall();
                   break;

                case PHONE_EVENT_CALL_TERMINATED:
                   CallTerminated();
                   break;

                case PHONE_EVENT_CALL_PROGRESS:
                   if( wTamState == TS_RECORD_MESSAGE ||
                          (wTamState==TS_REMOTE_PLAY &&
                          wRemoteState==RS_WAITING))
                   switch( dwEventData )
                      {
                       case DIALTONE:
                       case SLOWBUSY:
                       case FASTBUSY:
                          CallTerminated();
                          break;
                      }
                   break;

                case PHONE_EVENT_LINE_KEY:
                   if( wTamState == TS_REMOTE_PLAY )
                      {
                       if( wCmdKey == 5 ) // Check for play ctrl sequence
                         {
                          wCmdKey = -1;
                          switch( dwEventData )
                             {
                              case 1:    // (51) Seek back 5 seconds
                                 SeekMessage(TB_BACK);
                                 break;
                              case 2:    // (52) Pause (or resume)
                                 if(!(wPause^=1))
```

```
                                    mciSendComma nd( wOurDeviceID,
                                           MCI_RESUME, MCI_WAIT, 0);
                                 else
                                    mciSendCommand( wOurDeviceID,
                                           MCI_PAUSE, MCI_WAIT, 0);
                                 break;
                             case 3:    // (53) Seek ahead 5 seconds
                                 SeekMessage(TB_FORWARD);
                                 break;
                            }
                        }
                      else        // Standard Remote Play command
                        {
                         switch( dwEventData )
                            {
                             case 1:    // (1) Play first / next
                                 RemoteNext();
                                 break;
                             case 2:    // (2) Remove current message
                                 RemoteRemove();
                                 break;
                             case 3:    // (3) Repeat current message
                                 RemoteRepeat();
                                 break;
                             case 4:    // (4) Archive current message
                                 RemoteArchive();
                         break;
                             case 5:    // Initiate 2 key (5x) sequence
                                 wCmdKey = (short)dwEventData;
                                 break;
                            }
                        }
                  }

                   else         // Check for 3 digit command code
                     {
                      wKeys[2] = wKeys[1];
                      wKeys[1] = wKeys[0];
                      wKeys[0] = (short)dwEventData;
                      if( ++wKeys Pressed > 2 )
                        {
                         if((wKeys[2]*100+wKeys[1]*10+wKeys[0])==wCommandCode)
                            {
                             BeginRemote(); // Initiate remote playback
                             wCmdKey = -1;  // Reset command key status
                            }
                        }
                     }
                   break;

                default:
                   break;
               }
           }
        else
            return (DefWindowProc(hWnd, message, wParam, lParam));
      }
return (NULL);
}
```

Handling Events - OS/2

```
MRESULT EXPENTRY MyWindowProc ( HWND hwnd,
                                ULONG msg,
                                MPARAM mp1,
                                MPARAM mp2 )
{
  HDC hdc;
  static int          InitEnv    = 0;
  static short        wKeys[3];               // Last 3 keys entered
  static short        wQuiet;                 // Count for QUIET messages
  static short        wKeysPressed;           // Count for 3 key command
  static short        wCmdKey;                // Flag for 5-x play ctrl
  unsigned short      wEvent;
  unsigned long       dwEventData;
  static int          FlashState = 0;

  switch( msg )
  {
    case MM_MCINOTIFY:
```

```
    switch( SHORT1FROMMP(mp1) )
    {
     case MCI_NOTIFY_FAILURE:
     case MCI_NOTIFY_SUCCESSFUL:
     case MCI_NOTIFY_SUPERSEDED:
     case MCI_NOTIFY_ABORTED:
      switch( wTamState )
      {
       case TS_COMMAND_MODE:
       case TS_PLAY_MESSAGE:
        PlayComplete();
        break;

       case TS_REMOTE_PLAY:
        ContinueRemote();
        wQuiet = 0;
        break;

       case TS_PLAY_ANNOUNCEMENT:
        RecordMessage();
        wQuiet = 0;
        break;

       case TS_RECORD_MESSAGE:
        SaveMessage();
        break;

       case TS_ARCHIVE_PLAY:
        PlayComplete();
        break;

       default:
        break;
      }
      break;•
    }
   break;
  case MM_MCIEVENT:
      mep = (LPMCI_EVENT_PARMS)mp2;
      wEvent = LOWORD( mep->dwDataParam1 );  // or wParam
      dwEventData = mep->dwEventData;
      if (dwEventData >= '0')
       dwEventData -= '0';
      else if (dwEventData == '#')
       dwEventData = 35;
      else if (dwEventData == '*')
       dwEventData == 42;

      switch(wEvent)
      {
       case PHONE_EVENT_CALL_TAM:
        wKeysPressed = 0;
        AnswerCall();
        break;

       case PHONE_EVENT_CALL_TERMINATED:
        CallTerminated();
        break;

       case PHONE_EVENT_CALL_PROGRESS:
        if( wTamState == TS_RECORD_MESSAGE ||
          (wTamState==TS_REMOTE_PLAY && wRemoteState==RS_WAITING) )
          switch( dwEventData )
          {
           case DIALTONE:
           case SLOWBUSY:
           case FASTBUSY:
            CallTerminated();
            break;
          }
        break;

       case PHONE_EVENT_LINE_KEY:
        if( wTamState == TS_REMOTE_PLAY )
        {
          if( wCmdKey == 5 )  // Check for play ctrl sequence
          {
```

```
                        wCmdKey = -1;
                        switch( dwEventData )
                        {
                          case 1:
                            SeekMessage(TB_BACK);
                            break;
                          case 2:
                            if(!(wPause^=1))
                                mciSendCommand( wTpsDeviceID, MCI_RESUME, MCI_WAIT,
                                               (DWORD)&mciGenericParms, mci_cmd_ctr++);
                            else
                                mciSendCommand( wTpsDeviceID, MCI_PAUSE, MCI_WAIT,
                                                (DWORD)&mciGenericParms, mci_cmd_ctr++);
                            break;
                          case 3:
                            SeekMessage(TB_FORWARD);
                            break;
                        }
                    }
                    else                // Standard Remote Play command
                    {
                        switch( dwEventData)
                        {
                          case 1:
                            RemoteNext();
                            break;
                          case 2:
                            RemoteRemove();
                            break;
                          case 3:
                            RemoteRepeat();
                            break;
                          case 4:
                            RemoteArchive();
                            break;
                          case 5:       // Initiate play ctrl sequence
                            wCmdKey = (short)dwEventData;
                            break;
                        }
                    }
                }
                else                    // Check for 3 digit command code
                {
                    wKeys[2] = wKeys[1];
                    wKeys[1] = wKeys[0];
                    wKeys[0] = (short)dwEventData;
                    if( ++wKeysPressed > 2 )
                    {
                        if((wKeys[2]*100+wKeys[1]*10+wKeys[0])==wCommandCode)
                        {
                            BeginRemote();
                            wCmdKey = -1;
                        }
                    }
                }
                break;

              default:
                break;
            }
```

Using the TAM Sample Application

The sample application included on the companion diskette is designed primarily to illustrate some of the concepts behind the creation of an event driven application using the Mwave TAM API. Although the sample is also a functional telephone answering machine, it doesn't contain the error recovery or feature set required of a robust application.

The TAM example applet requires the following hardware in addition to the base
Mwave hardware:

- A telephone handset attached to the Mwave adapter telephone port (if you want to try the handset functions)

- An analog phone line attached to the public switch network (if you want to initiate and receive real calls)

- A microphone attached to the Mwave adapter microphone input (if you want to use the microphone functions)

- One or two speakers attached to the Mwave adapter speaker ports (if you want to use the speaker functions)

The TAM applet is designed to operate both as a standard telephone and a telephone answering system. When using either the telephone to take a call, or the answering system to review messages, the application allows the user to select either a desk telephone handset or an external microphone/speaker as an input/output device. This allows for private reviewing of messages through the handset, and adds speakerphone capability to a standard telephone through the microphone and speaker devices.

To implement this dual functionality, the application operates in two distinct modes, a 'Telephone' mode and a 'Message Review' mode. The operating mode is set by the user, through the 'Mode' pulldown menu.

In 'Telephone' mode, the applet answers incoming calls, plays an announcement, and records messages. When system output is set to 'Handset', the desk telephone is connected directly to the telephone line and on-hook off-hook is . When the output is set to 'Speaker', the telephone is taken off hook and the system microphone and speaker are enabled.

In 'Message Review' mode, the application can play recorded messages to either the speaker or telephone handset. All message play controls are located under the 'Play Control' menu. When message reviewing is taking place, the system does not answer incoming calls.

The handset volume is not adjustable with the applet, but the speaker volume (used for both speakerphone and message review operations) can be set using the 'Volume' menu.

System Setup

The TAM applet is pre-loaded with a default announcement greeting and some other default settings, but there are some initialization steps to perform if you wish to tailor it to your requirements. Below are specific instructions for the various initialization procedures.

Recording an Announcement

To record a new announcement, select the 'Message Review' mode option under the 'Mode' menu, and then select 'Record Announcement...' under the 'Configure' menu. A three part dialog box is displayed.

The top portion of the dialog box is used for recording. The 'Record from...' box in the upper left hand corner of the dialog box determines the input recording device. The default device is set to 'Microphone' so if you wish to record your announcement from the telephone handset, first select the 'Telephone Handset' button in the 'Record from...' box.

After the input device has been chosen, click on the 'Begin Recording' button and start speaking (once you hear a beep) into the input device you have chosen. When done, click on the 'End Recording' button. To listen to the announcement you just recorded, select an output device in the 'Play to...' box, and click on the 'Play Announcement' button. The announcement plays to the output device selected. If you wish to use the newly recorded announcement, click on the 'OK' button. Otherwise, to keep the old
announcement, click on the 'Cancel' button.

After closing the dialog box, place the application back into telephone mode by selecting the 'Telephone' mode option in the 'Mode' pulldown menu (to enable the system to take messages)

Setting the Ring Count

The ring count determines the number of rings before which the system answers the telephone to record an incoming message. The TAM application has two ring counts, one for when messages are available, and one for when no messages are available. A common ' toll saver' feature is to set the device to answer on the first ring when new messages are available, but not to answer before the fourth ring when there are no new messages. This allows the user to hang-up when calling remotely before the system answers when there are no messages available.

To set the ring count, select the 'Set Ring Count...' item in the 'Configure' menu. A dialog box is displayed, prompting for the two types of ring counts. After entering a new ring count for when messages are available and one for when no messages available, press the 'OK' button to use the new counts, or press the 'Cancel' button to abort any changes.

Setting the Command Code

The TAM application allows the user to retrieve messages from a remote telephone, by calling the device, and when prompted to record a message, entering instead a 3-digit command code on the touch-tone keypad. The command code is configurable, and can be changed any time. To set a new command code, select the 'Set Command Code...' item on the 'Configure' menu. A dialog box prompting for a new command code is displayed. After entering a new command code, press the 'OK' button to accept the change, or press the 'Cancel' button to abort any changes and keep the original code.

Using the Speakerphone

In addition to providing the answering machine function, the TAM application turns a speaker/microphone connected to the Mwave Adapter into a speakerphone.

Initiating a Speakerphone Call

Before initiating a call for use with the speakerphone, verify that the application is in telephone mode by selecting the 'Telephone' option under the 'Mode' pulldown menu, and that the system is connected to the handset by selecting the 'Handset' item under the 'Mode' pulldown menu.

Note: These two options should always be set when you are not reviewing messages. Otherwise, the system will not take messages

The speakerphone call is initiated by picking up the handset and dialing the number r using your standard desk phone. After the number has been dialed, the speakerphone is initiated by selecting the 'Speaker' option under the 'Mode' pulldown menu. After this has been done, you can hang-up the telephone handset.

Speakerphone Volume Control

While the speakerphone call is in progress, the output volume of the speaker can be adjusted by selecting a volume level from the 'Volume' pulldown menu.

Terminating a Speakerphone Call

To terminate a speakerphone call, leave the desk phone handset on hook, and place the system back into handset connect mode by selecting the 'Handset' item under the 'Mode' pulldown menu. This also places the phone 'on-hook'.

Reviewing Messages Locally

The number of active messages on the system is shown in the application window whenever the application is open on the desktop. If new messages arrive while the application is in icon form(i.e. minimized), new messages can be detected by the flashing of the icon text (either on the desktop or in the Minimized Window Viewer). If the TAM application window is open, its title bar flashes if new messages have arrived since the last message review.

Playing Recorded Messages

To review messages, select the 'Message Review' mode from the 'Mode' pulldown menu. Incoming calls are not answered while you are in this mode.

Next, select the output device to which to play your messages. You can play messages to either the telephone handset or the external speaker. To use the handset, select the 'Handset' option on the 'Mode' pulldown menu. To use an external speaker, select the 'Speaker' option on the 'Mode' pulldown menu. When using the speaker, you can adjust the speaker volume be selecting a new volume level from the 'Speaker Volume' pulldown menu.

When in message review mode, some of the entries on the 'Play Control' pulldown menu become visible. To start reviewing messages, select the 'First' item on the 'Play Control' pulldown menu. This prompts the system to play the first active message. If the 'First' item is grayed on the menu, you have no active messages.

After the first message has played, you have a choice of either replaying the message, keeping the message and playing the next message (if any), or erasing the message and playing the next message (if any). To replay the message, select the 'Repeat' option on the 'Play Control' pulldown menu. To keep this message
and play the next message, select the 'Next' option. To erase this message and play the next message (if any), select the 'Erase' option.

After playing one of multiple messages, you can go back and play the previous message by selecting the 'Previous' option.

Note: When you have finished reviewing messages, re-enable call receiving by selecting the 'Telephone' mode option under the 'Mode' pulldown menu.

Message Positioning Controls

While a message is playing, you can step back 5 seconds in the message, step forward 5 seconds in the message, or pause the message playback. When paused, the playback will remain stopped until unpaused (pause selected a second time). These options are performed by selecting the 'Back 5 seconds', 'Ahead 5 seconds', or 'Pause' menu entries under the 'Play Control' pulldown menu.

Message Speed Controls

As well as being able to skip around in a message using 'Step Back' and 'Skip Forward', the TAM application allows you to set the play speed of the message, so that you can play back messages at an accelerated or decelerated rate. To change the message play speed, select a new speed (either 'Play Slow', 'Play Normal', or 'Play Fast') from the 'Play Control' pulldown menu. The speed setting you choose remains in effect until you exit the message review mode at which point it reverts to 'Play Normal'.

Reviewing Messages Remotely

Message review is also possible using a touch-tone telephone, when calling from a remote location. To gain access to the message review mode of the application, first call the system and let the answering machine answer the call. While the announcement is playing, enter the 3 digit command code you selected during the application setup process. The system now tells you how many active messages you have. At this point, you can begin entering touch-tone keypad commands to review messages as described in the previous section. Below is a list of the available keypad command options:

Message Play Commands:

Press '1' to play first/next active message
Press '2' to erase current message and play next active message
Press '3' to replay or restart the current message

Chapter 6 - FAX API Reference

This chapter provides a complete reference of the Mwave FAX Application Program Interface (API).

MCI Telephone Event Handler

Communication of real-time status information from the FAX driver to the application is performed through an application event handler. The handler should be able to service messages posted by the FAX driver through the MCI device, which contain real-time status information about the device. The message, MM_MCIEVENT, is not a standard MCI message under *Microsoft Windows*, thus a Microsoft Windows application must call the **RegisterWindowMessage** function with the string "MM_MCIEVENT", to obtain the numeric value of the notification message.

MM_MCIEVENT

In addition to the message itself, *wParam* and *lParam* are used to pass information to the application.

WPARAM *wParam*
Contains a device specific event message ***wEvent***.

LPMCI_EVENT_PARMS *lParam*
Specifies a far pointer to the following MCI_EVENT_PARMS structure:

```
typedef struct {
   DWORD dwDataParam1;
   DWORD dwEventData;
} MCI_EVENT_PARMS;
```

The data parameters are defined as follows:

DWORD *dwDataParam1*
The low-order word specifies the device specific event message ***wEvent*** (same as *wParam*). The high-order word specifies the device ID of the device initiating the message.

DWORD *dwEventData*
Contains a data parameter, which is dependent on the message type. The actual parameters passed are listed in Table 0 below, and detailed in the event message descriptions.

MM_MCIEVENT - OS/2

In addition to the message itself, *wParam* and *lParam* are used to pass information to the application.

DWORD *MsgParam1*
Contains a device-specific event message and device ID.

WORD *wEvent*
The low-order word of *MsgParam1* specifies the device- specific event message (same as *usEventCode* or *wParam*)

WORD *wDeviceID*
The high-order word of *MsgParam1* specifies the device ID of the device initiating the message.

LPMCI_EVENT_PARMS *MsgParam2*

LPMCI_EVENT_PARMS *EventData*
Specifies a pointer to the following structure:

```
typedef struct {
   DWORD dwDataParam1;
   DWORD dwEventData;
} MCI_EVENT_PARMS;
```

Note: The low-order word of **dwDataParam1** contains the event code (same as *wEvent*). The high-order word is not defined.

FAX Event Message Descriptions

This section describes the Event Messages generated by the FAX API. The following table provides a summary of all the Event Messages (wEvent), along with a short description of the data parameter associated with each:

Event Message (wEvent)	**Data Parm. (dwEventData)**
PHONE_EVENT_CALL_FAX	*undefined*
PHONE_EVENT_CALL_PROGRESS	New call state
PHONE_EVENT_CALL_TERMINATED	Call termination status
PHONE_EVENT_CALLER_ID	Caller ID status
PHONE_EVENT_DISTINCTIVE_RING	Ring Identifier
PHONE_EVENT_FAX_CONNECT	DCS frame information
PHONE_EVENT_FAX_HEADER	Pointer to fax header
PHONE_EVENT_FAX_PAGE_COMPLETE	Document completion status
PHONE_EVENT_FAX_PAGE_STATUS	Page completion status
PHONE_EVENT_FAX_POLL	*undefined*
PHONE_EVENT_HANDSET	Handset Status
PHONE_EVENT_HANDSET_KEY	Keypress character
PHONE_EVENT_LINE	Telephone line status
PHONE_EVENT_LINE_KEY	Keypress character
PHONE_EVENT_ADVANCED_RING	undefined, use lParam
PHONE_EVENT_RING	Telephone ring status

Table 6-1: FAX Driver Event Messages

For all messages posted to the event handler routine, the message value is **MM_MCIEVENT**. The value of *wEvent* and *dwEventData* vary according to the specific message posted. Below is a more detailed description of the event messages and their parameters.

Arguments *wEvent:* **PHONE_EVENT_CALL_FAX**
dwEventData: *undefined*

Description This message is posted when a call has been answered by the device, and has been determined to have originated from a fax device. At this time, the application that is not doing fax polling should immediately make a call to MCI_RECEIVE to receive any incoming fax data. At this point, the application should expect any of four additional messages to be posted by the device:

- PHONE_EVENT_CALLER_ID (If Discriminator running)
- PHONE_EVENT_FAX_CONNECT
- PHONE_EVENT_FAX_HEADER
- PHONE_EVENT_FAX_POLL

These additional messages are documented below.
Note: If this message is posted then you are guaranteed to get a PHONE_EVENT_CALL_TERMINATED. At which time, you must do a MCI_FAX_SET_HOOK (ONHOOK).

Arguments *wEvent:* **PHONE_EVENT_CALL_PROGRESS**
dwEventData: dwCallProgress

Description This message is posted when there has been a change in the current call state (or status). The new state of the call is supplied in dwCallProgress, and can be any of the following:

- DIALTONE
- ANSWERTONE
- SLOWBUSY
- FASTBUSY
- RINGTONE
- UNIDENTIFIEDTONE
- QUIET
- BUSY

Arguments *wEvent:* **PHONE_EVENT_CALL_TERMINATED**
dwEventData: **dwTermination**

Description This message is posted when a call has been terminated either by the caller, by the owning application, or because of an error condition. The reason for call termination is given in **dwTermination**, which may be any of the following values:

- TERMINATION_ERROR_RECV
- TERMINATION_ERROR_XMIT
- TERMINATION_NORMAL
- TERMINATION_REQUESTED
- TERMINATION_UNEXPECTED
- TERMINATION_DISK_FULL

Note: At this time the application MUST perform a MCI_FAX_SET_HOOK (ONHOOK).

Arguments *wEvent:* **PHONE_EVENT_CALLER_ID**
dwEventData: **dwCallerId**

Description This message is posted when a caller ID string has been decoded off a ringing line. It is posted only if a caller ID signal is present. **dwCallerID** indicates the completion status.

- MCI_VALID_CALLER_ID_RECEIVED
- MCI_CALLER_ID_FRAME_ERROR

The application must issue an MCI_INFO message to retrieve the id (for MCI_VALID_CALLER_ID_RECEIVED) or the error code (for MCI_CALLER_ID_FRAME_ERROR).

PHONE_EVENT_CALLER_ID is only supported if Discriminator is loaded.

Arguments *wEvent:* **PHONE_EVENT_DISTINCTIVE_RING** **PL**
dwEventData: **dwRingIdentifier**

Description This message is posted when a distinctive ring has been decoded off a ringing line. It is posted only if distinctive ring support is installed. **dwRingIdentifier** indicates which distinctive ring has been decoded. The ring identifier is a number between 1 and 20. This support is added with Ver 3.2. For FAX it is available only when running the discriminator.

Arguments *wEvent:* **PHONE_EVENT_FAX_CONNECT**
dwEventData: **dwConnect**

Description This message is posted after a fax call has been answered by the device, and has finished the negotiation period and established the Digital Command Signal (DCS) connection parameters. The **dwConnect** specifies a far pointer to MCI_FAX_CONNECT_PARMS data structure containing these connection parameters:

```
typedef struct {
          DWORD          dwSignalRate;
          DWORD          dwCompression;
          DWORD          dwErrorCorrection;
          DWORD          dwResolution;
          DWORD          dwWidth;
          DWORD          dwMinScanLineTime;
}  MCI_FAX_CONNECT_PARMS;
```

The signal rate is passed in **dwSignalRate**, and can be any of the following:

- MCI_FAX_MODEM_V27TER_2400
- MCI_FAX_MODEM_V27TER_4800
- MCI_FAX_MODEM_V29_7200
- MCI_FAX_MODEM_V29_9600
- MCI_FAX_MODEM_V17_7200
- MCI_FAX_MODEM_V17_9600
- MCI_FAX_MODEM_V17_12000
- MCI_FAX_MODEM_V17_14400
- MCI_FAX_MODEM_ANY

The following compression types are passed in **dwCompression.** This message is especially useful if the compression type is BFT (binary file transfer), because in this case, the file resulting from an MCI_RECEIVE is an unencoded binary file.

- MCI_FAX_COMPRESSION_1D
- MCI_FAX_COMPRESSION_2D
- MCI_FAX_COMPRESSION_BFT

The error correction is passed in **dwErrorCorrection**, and can be either TRUE or FALSE.

The resolution is passed in **dwResolution**, and can be any of the following:

- MCI_FAX_RESOLUTION_NORMAL
- MCI_FAX_RESOLUTION_FINE

The document width in pels is passed in **dwWidth**.

The device specific minimum milliseconds to scan a line is passed in **dwMinScanLineTime**.

Arguments *wEvent:* **PHONE_EVENT_FAX_HEADER**
dwEventData: **lpstrFaxHeader**

Description This message is posted when a fax header string has been decoded off a fax call. It is posted only if a header string is present. An application can use this string to identify the fax sender/receiver. A pointer to the null terminated ASCII string is pointed to by **lpstrFaxHeader**.

Arguments *wEvent:* **PHONE_EVENT_FAX_PAGE_COMPLETE**
dwEventData: **dwCompletionStatus**

Description This message is posted when the device has completed either sending or receiving a fax document page. In the event that the device is in the middle of a MCI_SEND, the completion status (measured in percent) is supplied in **dwCompletionStatus**.

Arguments *wEvent:* **PHONE_EVENT_FAX_PAGE_STATUS**
dwEventData: **dwPageStatus**

Description This message is posted several times per page during either sending or receiving a fax document. The completion status (measured in percent) is supplied in **dwPageStatus** for MCI_SEND, and is not supplied for MCI_RECEIVE except for 0% when incoming page is known.

Note: It is expected behavior to only get a the 0% and 100% on the first page of the outgoing fax. The reason is due to the low priority of timer messages in Windows. Subsequent pages should give a % every second.

Arguments *wEvent:* **PHONE_EVENT_FAX_POLL**
dwEventData: *undefined*

Description This message is posted after a call has been answered by the device, and has been determined to have originated from a fax device and a poll command is received. At this time, the application should immediately make a call to MCI_SEND to send the requested fax data.

Arguments *wEvent:* **PHONE_EVENT_HANDSET**
dwEventData: **dwHandsetStatus**

Description This message is posted when the status of the telephone handset changes, due to the user either picking up or replacing the telephone handset. The value of **dwHandsetStatus** is as follows:

dwHandsetStatus = 0	Handset is on-hook
dwHandsetStatus = 1	Handset is off-hook (in use)

The Discriminator must be running to enable receipt of this message.

Arguments *wEvent:* **PHONE_EVENT_HANDSET_KEY**
dwEventData: **dwKeypress**

Description This message is posted when a key has been pressed on the handset device. The index of the pressed key (0 to 11, 10 for '*' and 11 for '#') is supplied in **dwKeypress**.

PHONE_EVENT_HANDSET_KEY Is not supported in current FAX driver.

Arguments *wEvent:* **PHONE_EVENT_LINE**
dwEventData: **dwLineStatus**

Description This message is posted when the status of the telephone line changes, due to another application in the system making use of the telephone line. When an application takes the telephone line off hook, or is called to service an incoming call, it remains in possession of the telephone line for the duration of the call. Applications which require use of the telephone line and find it busy, can simply wait for this message to signal that the telephone line can be used. The value of **dwLineStatus** is as follows:

dwLineStatus = 0	Telephone line is free
dwLineStatus = 1	Telephone line is in use

The Discriminator must be running to enable receipt of this message.

.

Arguments *wEvent:* **PHONE_EVENT_LINE_KEY**
dwEventData: **dwKeypress**

Description This message is posted when a key has been pressed on the incoming telephone line. An ASCII character representing the pressed key ('0' - '9', 'a' - 'd', '#' or '*'), is supplied in **dwKeypress**.

PHONE_EVENT_LINE_KEY is not supported in current FAX driver.

Arguments *wEvent:* **PHONE_EVENT_RING**
dwEventData: **dwRingStatus**

Description This message is posted when a ring signal change is detected by the device. This message can be used by the application to count ring cycles, or determine ring length. The value of **dwRingStatus** is as follows:

dwRingStatus = 0	Telephone ring signal end (not ringing)
dwRingStatus = 1	Telephone ring signal start (ringing)

Arguments *wEvent:* **PHONE_EVENT_ADVANCED_RING**
dwEventData: not used, actual ring count is in lParam

Description If the application has requested 'advanced format ring notifications' by setting advanced ring notify to TRUE, PHONE_EVENT_ADVANCED_RING is sent to the application instead of PHONE_EVENT_RING. In this case, lParam is not a pointer to a structure. Instead, the low word of lParam contains the ring count, and the high word of lParam contains the device ID.

LOWORD(lParam) = 0	Telephone ring signal end (not ringing)
LOWORD(lParam) = 'n'	Telephone ring count (where 'n' is the ring number)

The Discriminator must be running to enable receipt of this message.

FAX Driver API Messages and Flags

This section describes the MCI compliant FAX API messages and flags. The following table provides a summary of the MCI command messages used in the FAX API, and a short description of each:

MCI Message	Description
MCI_CLOSE	Close the device driver
MCI_CONVERT	Convert from/to device dependent file to/from device independent file.
MCI_DIAL	Dial the telephone
MCI_GETDEVCAPS	Get the capabilities of the device
MCI_INFO	Get device string identifier
MCI_OPEN	Open the device driver
MCI_RECEIVE	Receive a *fax* file
MCI_SEND	Send a *fax* file
MCI_SET	Configure the device
MCI_STATUS	Query device configuration
MCI_STOP	Stop sending or receiving a FAX

Table 6-2: FAX Driver API Messages

MCI_CLOSE

This command message closes the FAX driver.

Parameters DWORD *lParam1*

The following flags apply to the FAX device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application. The event handler window must be specified in the **dwCallback** field regardless of whether MCI_NOTIFY or MCI_WAIT is selected.

LPMCI_GENERIC_PARMS *lParam2*

Specifies a far pointer to the following **MCI_GENERIC_PARMS** data structure:

```
typedef struct {
      DWORD        dwCallback;
} MCI_GENERIC_PARMS;
```

Note: Be sure to assign the handle of the window procedure responsible for processing MM_MCINOTIFY messages to dwCallback prior to calling MCI_CLOSE regardless of whether MCI_WAIT or MCI_NOTIFY is specified. Failure to do so results in erratic behavior of the Fax device driver when using versions earlier than Ver 2.1 of the Fax device driver

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_CONVERT

This command message is used to convert data files between a MCI device dependent format, and a standard device independent format. The call is used to convert to and from FAX multi-page documents.

Parameters DWORD *lParam1*

The following flags apply to the FAX device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_CONVERT_CREATE

Indicates that the destination file is a new file which should be created. This overwrites any existing file.

MCI_CONVERT_DESTINATION_FILE

Indicates the **lpstrDestFilename** field of the data structure identified by *lParam2* contains a pointer to a buffer containing the destination file name.

MCI_CONVERT_DESTINATION_FORMAT

Indicates the **dwDestFormat** field of the data structure identified by *lParam2* contains the desired format of the destination file. These include:

- MCI_CONVERT_FMT_DIB_BMP *(from source of type DEVFAX)*
- MCI_CONVERT_FMT_DIB_RLE *(from source of type DEVFAX..not supported in current FAX driver)*
- MCI_FAX_CONVERT_FMT_DEVFAX *(from DIB_BMP or DIB_RLE. DIB_RLE conversion not supported in current FAX driver)*

MCI_CONVERT_DESTINATION_FROM

Specifies that a media starting position is included in the **dwDestFrom** field of the data structure identified by *lParam2*. This specifies the document page (starting at zero) at which the converted data is written to the destination file.

MCI_CONVERT_INFO

Indicates that no conversion operation is to take place, but rather, the **dwLength** field of the data structure identified by *lParam2* should be set to the length of the media of the supplied source device dependent filename. For a device dependent FAX file, the value returned is the document page count. If a device dependent file is not specified, this call returns an error.

MCI_CONVERT_OVERWRITE

Indicates that newly converted information should overwrite any existing data. If this flag is not specified, the new data is inserted into the file.

MCI_CONVERT_SOURCE_FILE

Indicates the **lpstrSrcFilename** field of the data structure identified by *lParam2* contains a pointer to a buffer containing the file name.

MCI_CONVERT_SOURCE_FROM

Specifies that a media starting position is included in the **dwSrcFrom** field of the data structure identified by *lParam2*. This specifies the document page (starting at zero) at which the data to be converted is read from the source file.

LPMCI_CONVERT_PARMS *lParam2*

Specifies a far pointer to the following **MCI_CONVERT_PARMS** data structure:

```
typedef struct {
      DWORD       dwCallback;
      LPCSTR      lpstrDestFilename;
      DWORD       dwDestFormat;
      DWORD       dwDestFrom;
      DWORD       dwLength;
      LPCSTR      lpstrSrcFilename;
      DWORD       dwSrcFrom;
} MCI_CONVERT_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_DIAL

This command message takes the phone off-hook, and dials the supplied number. If the telephone is owned by another application at the time of this call, the command will fail.

Parameters DWORD *lParam1*

The following flags apply to the telephone device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_DIAL_DIALMODE

Specifies that the **dwDialMode** field of the data structure identified by *lParam2* contains a constant specifying the phone dialing mode. Two modes are defined:

- MCI_DIAL_MODE_PULSE
- MCI_DIAL_MODE_TONE

MCI_DIAL_FLASH

Indicates that the telephone should be flashed before dialing the supplied number (if any).

MCI_DIAL_MONITOR

Specifies that the audio speaker device should be enabled during the calling process.

MCI_DIAL_MONITOR_HANDSHAKING_ONLY

Specifies that the audio speaker device should be enabled only during the negotiation period of the calling process.

MCI_DIAL_STRING

Specifies that the **lpstrDialString** field of the data structure identified by *lParam2* contains a pointer to a null terminated dialing string. Numeric characters '0' to '9' correspond to phone digits. The '*' and '#' characters, the alpha characters 'a' to 'd' and the '-' are also supported ('-' is ignored). The 'w' character in the string specifies that the device should wait for a second dial tone before proceeding, and a ',' character indicates a pause in the dialing sequence. The time-out limit for the wait command (default 30 seconds) and the delay time for the pause command (default 2 seconds) are configurable using MCI_SET. The ‘@’ character in the string specifies wait for quiet. The ‘p’ character in the string specifies switch to pulse dialing. The ‘t’ character in the string specifies switch to tone dialing. The ‘!’ character in the string specifies *flash* the line

The maximum size string that can be dialed is specified by MAX_DIAL_STRING.

MCI_DIAL_VERIFY

Specifies that the call is to be verified. The phone is verified to be off-hook, and that a dial tone is present before dialing. The correct line type format is also verified.

LPMCI_DIAL_PARMS *lParam2*

Specifies a far pointer to the following **MCI_DIAL_PARMS** data structure:

```
typedef struct {
      DWORD       dwCallback;
      DWORD       dwDialMode;
      LPCSTR      lpstrDialString;
} MCI_DIAL_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_GETDEVCAPS

This command is used to obtain static information about a device.

Parameters DWORD *lParam1*

The following flags apply to the FAX device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_GETDEVCAPS_ITEM

Specifies that the **dwItem** field of the data structure identified by *lParam2* contains a constant specifying which device capability to obtain. The following constants are defined:

MCI_GETDEVCAPS_CAN_EJECT

The **dwReturn** field is set to FALSE.

MCI_GETDEVCAPS_CAN_PLAY

The **dwReturn** field is set to FALSE.

MCI_GETDEVCAPS_CAN_RECORD

The **dwReturn** field is set to FALSE.

MCI_GETDEVCAPS_CAN_SAVE

The **dwReturn** field is set to FALSE.

MCI_GETDEVCAPS_COMPOUND_DEVICE

The **dwReturn** field is set to FALSE.

MCI_GETDEVCAPS_DEVICE_TYPE

The **dwReturn** field is set to MCI_DEVTYPE_OTHER.

MCI_GETDEVCAPS_HAS_AUDIO

The **dwReturn** field is set to FALSE.

MCI_GETDEVCAPS_HAS_VIDEO

The **dwReturn** field is set to FALSE.

MCI_GETDEVCAPS_USES_FILES

The **dwReturn** field is set to TRUE.

MCI_FAX_GETDEVCAPS_COMPRESSION_TYPES

The **dwReturn** field is set to the logical ORing of the following supported compression types:

- MCI_FAX_COMPRESSION_MH

- MCI_FAX_COMPRESSION_MR
- MCI_FAX_COMPRESSION_MMR
- MCI_FAX_COMPRESSION_NONE
- MCI_FAX_COMPRESSION_BFT

Only MCI_FAX_COMPRESSION_MH supported in current FAX driver.

MCI_FAX_GETDEVCAPS_CAN_RECEIVE
The **dwReturn** field is set to TRUE if the device supports receiving FAX file data from the telephone line. Otherwise, it is set to FALSE.

MCI_FAX_GETDEVCAPS_CAN_SEND
The **dwReturn** field is set to TRUE if the device supports sending FAX file data to the telephone line. Otherwise, it is set to FALSE.

MCI_FAX_GETDEVCAPS_HAS_HANDSET
The **dwReturn** field is set to TRUE if the device supports call monitoring through an external handset; otherwise, it returns FALSE.

MCI_FAX_GETDEVCAPS_HAS_HANDSET not supported in current FAX driver.

MCI_FAX_GETDEVCAPS_MODEM_TYPES

The **dwReturn** field is set to the logical ORing of the following supported modem types:

- MCI_FAX_MODEM_V27TER_2400
- MCI_FAX_MODEM_V27TER_4800
- MCI_FAX_MODEM_V29_7200
- MCI_FAX_MODEM_V29_9600
- MCI_FAX_MODEM_V17_7200
- MCI_FAX_MODEM_V17_9600
- MCI_FAX_MODEM_V17_12000
- MCI_FAX_MODEM_V17_14400

MCI_FAX_GETDEVCAPS_POLLING

The **dwReturn** field is set to TRUE if FAX polling is supported, and FALSE if not.

MCI_FAX_GETDEVCAPS_RESOLUTION

The **dwReturn** field is set to the resolution of the device.

- MCI_FAX_RESOLUTION_FINE 200x200 PIXELS/INCH
- MCI_FAX_RESOLUTION_NORMAL 100x200 PIXELS/INCH

MCI_FAX_GETDEVCAPS_SUPPORTS_ECM

The **dwReturn** field is set to TRUE if FAX ECM is supported, and FALSE if not.

MCI_FAX_GETDEVCAPS_FILE_FORMATS

File formats supported for fax send/receive. The **dwReturn** field is set to logical ORing of the following file formats:

- TIFF_CLASS_F
- DCX
- RIFF
- TIFF_6.0

MCI_FAX_GETDEVCAPS_WIDTH

The **dwReturn** field is set to the width in pels of the device.

MCI_FAX_GETDEVCAPS_WIDTH not supported in current FAX driver.

LPMCI_GETDEVCAPS_PARMS *lParam2*

Specifies a far pointer to the following **MCI_GETDEVCAPS_PARMS** data structure:

```
typedef struct {
      DWORD       dwCallback;
      DWORD       dwReturn;
      DWORD       dwItem;
} MCI_GETDEVCAPS_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_INFO

This command message obtains string information from the device.

Parameters DWORD *lParam1*

The following flags apply to the FAX device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_INFO_PRODUCT

Obtains a description of the hardware associated with a device. The description identifies both the driver and the hardware used. The string is copied to the buffer pointer to by the **lpstrReturn** field of the structure identified by *lParam2*. The size of this buffer is specified by the **dwRetSize** field of the same structure, and if the buffer is of insufficient size to contain the string, the string is truncated to fit the buffer. The string contains a version number (i.e, “Ver 3.0”). Driver enhancements will be denoted in this document with the “Ver x.y”. that corresponds with the first release that the feature shows up in. The version number will always increase in future releases, so a program can parse the string, looking for “Ver “, convert the characters that follow “Ver “ to a number, and do a numeric greater-than-or-equal compare to determine if the function is available in the release the application is running with.

Note: Unless otherwise noted, all functions are available as of Ver 2.2

MCI_INFO_CALLER_ID

Obtains a caller ID string. (See PHONE_EVENT_CALLER_ID). The caller ID data is copied into the buffer pointed to by the **lpstrReturn** field of the structure identified by *lParam2*. The size of this buffer is specified by the **dwRetSize** field of the same structure (maximum size = MCI_MAX_CALLER_ID_SIZE). If the buffer is of insufficient size to contain the data, the data is truncated to fit the buffer, the return code is set to MCIERR_INVALID_BUFFER, and the dwRetSize is set to the size needed to retrieve the entire caller ID buffer.

Note: The caller ID data is in the format defined by Bellcore’s technical reference bulletin TR-TSY-000031 and TR-NWT-001188. Also note that a checksum is included at the end of the caller ID data.

MCI_INFO_CALLER_ID is only supported when the discriminator is loaded.

MCI_INFO_CALLER_ID_ERROR

Obtains the caller ID error code. (See PHONE_EVENT_CALLER_ID). The code is copied into the buffer pointed to by the **lpstrReturn** field of the structure identified by *lParam2*. The size of this buffer is specified by the **dwRetSize** field of the same structure. The error code is either MCI_CHECKSUM_ERROR or MCI_FRAME_ERROR.

MCI_INFO_CALLER_ID_ERROR is only supported when the discriminator is loaded.

MCI_INFO_CALLER_PARSED_CALLER_ID

Obtains a caller an already-parsed Caller ID string. (See PHONE_EVENT_CALLER_ID). The information is copied into the structure pointed to by the **lpstrReturn** (Windows) or **dwReturn** (OS/2) field of the structure identified by *lParam2*. The structure is:

```
typedef struct
{
   char szDateTime[DATE_TIME_LEN+1];
   char szNumber[MCI_MAX_CALLER_ID_SIZE]; /* callers
number */
   char szName[MCI_MAX_CALLER_ID_SIZE];    /* callers name
(may
be null) */
} CIDINFO;
```

This function is implemented in "Ver 3.0" of the TAM driver.

MCI_INFO_CALLER_PARSED_CALLER_ID is only supported when the discriminator is loaded.

LPMCI_INFO_PARMS *lParam2*

Specifies a far pointer to the following **MCI_INFO_PARMS** data structure:

```
typedef struct {
      DWORD       dwCallback;
      LPSTR       lpstrReturn;
      DWORD       dwRetSize;
} MCI_INFO_PARMS;
```

```
typedef struct {
   DWORD       dwCallback;
   LPSTR       lpstrReturn;
   DWORD       dwRetSize;
} MCI_INFO_PARMS;              /*OS/2*/
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_OPEN

This command message initializes the telephony driver and hardware.

Parameters DWORD *lParam1*

The following flags apply to the FAX device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application. In older versions of the driver, the event handler window must be specified in the **dwCallback** field regardless of whether MCI_NOTIFY or MCI_WAIT is selected.

MCI_OPEN_ALIAS

Specifies that an alias is included in the **lpstrAlias** field of the data structure identified by *lParam2*. This command is handled by MCI.

MCI_OPEN_SHAREABLE

Specifies that the device should be opened as shareable.

MCI_OPEN_SHAREABLE is not supported in current FAX driver.

MCI_OPEN_TYPE

Specifies that a device type name or constant is included in the **lpstrDeviceType** field of the data structure identified by *lParam2*. To open the fax driver, specify “Mwavefax” in the **lpstrDeviceType.** This command is handled by MCI.

MCI_OPEN_TYPE_ID

Specifies that the low-order word of the lpstrDeviceType field of the associated data structure contains a standard MCI device type ID and the high-order word optionally contains the ordinal index for the device. This command is handled by MCI.

LPMCI_OPEN_PARMS *lParam2*

Specifies a far pointer to the following **MCI_OPEN_PARMS** data structure:

```
typedef struct {
      DWORD        dwCallback;
      WORD         wDeviceID;
      WORD         wReserved0;
      LPCSTR       lpstrDeviceType;
      LPCSTR       lpstrElementName;
      LPCSTR       lpstrAlias;
} MCI_OPEN_PARMS;
```

Note: With Microsoft Windows, be sure to assign the handle of the window procedure responsible for processing MM_MCINOTIFY messages to dwCallback prior to calling MCI_OPEN regardless of whether MCI_WAIT or MCI_NOTIFY is specified. Failure to do so results in erratic behavior when using versions earlier than Ver 2.1 of the Fax device driver.

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

Remarks

- Case is ignored in the device name, but there must not be any leading or trailing blanks.

- Note that the device type is the **pszDeviceType** field of the **MCI_OPEN_PARMS** data structure, but it does not have a corresponding flag because it is required and does not have a command-string parameter.

  For the Mwave Fax and TAM drivers, the device types are:

  Mwavetpl
  Mwavetps
  Mwavefax

- OS/2 only: If automatic type selection is desired (through the extensions or EA section or INI), the file name (including the extension) must be passed in the **pszElementName** parameter, the **pszDeviceType** is left null, and the **MCI_OPEN_ELEMENT** flag is set.

MCI_RECEIVE

This command message receives a file. In the case of FAX, this file is an OEM dependent FAX file consisting of one or more image pages. The number of pages actually received is available in MCI_STATUS.

Parameters DWORD *lParam1*

The following flags apply to the FAX device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_RECEIVE_FILE

Indicates the **lpstrFilename** field of the data structure identified by *lParam2* contains a pointer to a buffer containing the file name where the received fax data is to be stored.

MCI_ALREADY_DIALED

Indicates the document is to be received immediately because the application has already connected to the partner fax machine. In FAX vernacular, this is often referred to as Manual Receive.

LPMCI_RECEIVE_PARMS *lParam2*

Specifies a far pointer to the following **MCI_RECEIVE_PARMS** data structure:

```
typedef struct {
      DWORD       dwCallback;
      LPCSTR      lpstrFilename;
} MCI_RECEIVE_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_SEND

This command message sets up a document or documents for sending, which then takes place during a following MCI_DIAL command message. In the case of FAX, this file is an OEM dependent FAX file or files consisting of one or more image pages. The number of pages sent can be obtained via MCI_STATUS.

Parameters DWORD *lParam1*

The following flags apply to the FAX device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_SEND_FILE

Indicates the **lpstrFilename** field of the data structure identified by *lParam2* contains an array of pointers to pointers to strings identifying the file name of each FAX file to send. The **lpstrFilename** array is terminated with a NULL string pointer to indicate the end of the file name list.

MCI_FAX_SEND_SINGLE_FILE

Indicates the **lpstrFilename** field of the data structure identified by *lParam2* contains a string identifying the file name of the FAX file to send.

MCI_ALREADY_DIALED

Indicates the document is to be sent immediately because the application has already connected to the partner fax machine. In FAX vernacular, this is often referred to as Manual Send.

MCI_SEND_HEADING

Indicates the **lpstrHeading** field of the data structure identified by *lParam2* contains a string identifying the full path and file name of the heading file. The heading file must be in Tiff Class F format, single strip. Each heading should be a tiff page.

Note: The heading file: should contain a heading for every page to be sent, must have the same fill order and resolution as the page being sent with it, and must be less than 24K.

This function is implemented in “Ver 3.0” of the FAX driver

LPMCI_SEND_PARMS *lParam2*

Specifies a far pointer to the following **MCI_SEND_PARMS** data structure:

```
typedef struct {
      DWORD       dwCallback;
      LPCSTR       lpstrFilename[];
      LPSTR       lpstrHeading;
} MCI_SEND_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_SET

This command is used to set the FAX device configuration. This configuration determines the environment used to send Fax Document Files. The item to set is specified by **dwItem** field of the MCI_FAX_SET_PARMS structure, pointed to by *lParam2,* and set data information is passed in **dwSetData**.

Parameters DWORD *lParam1*

The following flags apply to the FAX device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_SET_ITEM

Specifies that the **dwItem** field of the data structure identified by *lParam2* contains a constant specifying which item to set. The following constants are defined:

MCI_FAX_SET_ADVANCED_RING_NOTIFY

The **dwSetData** field is set to the indicate the type of message that is sent to the application when the phone rings. When set to FALSE (the default), a PHONE_EVENT_RING is sent to the application. When the flag is set to TRUE, a PHONE_EVENT_ADVANCED_RING is sent to the application. With advanced format ring events, *lParam* does not contain a pointer to **dwSetData**. Instead, *lParam* contains the device ID and the actual ring count (not a pointer to it). A ring count of zero signifies the end of a ring.

MCI_FAX_SET_ADVANCED_RING_NOTIFY is not supported in current driver.

MCI_FAX_SET_API_STYLE

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains the API style of the FAX device. The possible values are:

- MCI_FAXTAM_STYLE_MMPM
- MCI_FAXTAM_STYLE_WINDOWS

The default style under Windows is WINDOWS. The default style under OS/2 is MMPM. The API style affects return codes for MCI_STATUS, MCI_GETDEVCAPS, and MCI_INFO. The style also affects return codes and return information for MM_MCINOTIFY. See Microsoft Windows Multimedia Programmer's Reference and IBM's Programming Reference for Multimedia Presentation Manager Toolkit/2 for details of the MCI interface as specified for Windows and OS/2.

MCI_FAX_SET_AUDIO_VOLUME

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the volume level of the speaker device. The volume level is specified from 0x0 (silence) to 0xFFFF (maximum volume) and is interpreted logarithmically. This means the perceived volume increase is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

MCI_FAX_SET_CALL_FILTER

Specifies that the **dwSetData** field of the data structure identified by *lParam2* is set to TRUE if the device is to receive fax calls; otherwise it is set to FALSE. If another application has this filer enabled, attempting to enable the filter causes an error return.

MCI_FAX_SET_COMPRESSION_TYPES

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains the allowable FAX compression type(s) for files to be received. The following type flags are defined:

- MCI_FAX_COMPRESSION_MH
- MCI_FAX_COMPRESSION_MR
- MCI_FAX_COMPRESSION_MMR
- MCI_FAX_COMPRESSION_NONE
- MCI_FAX_COMPRESSION_BFT
- MCI_FAX_COMPRESSION_ANY

Only MH compression type supported in current FAX driver

MCI_FAX_SET_DIAL_FLASH_TIME

The **dwSetData** field is set to the desired flash time (in milliseconds) of the telephone flash option in the MCI_DIAL command. The default value is 500 (one half second).

MCI_FAX_SET_DIAL_PAUSE_TIME

The **dwSetData** field is set to the desired pause time (in milliseconds) that an embedded ',' character produces in the dial string. The default value is 2000 (2 seconds).

MCI_FAX_SET_DIAL_WAIT_TIME

The **dwSetData** field is set to the desired time-out limit (in milliseconds) that an embedded 'w' character in the dial string allows, waiting for a second dial tone. The default value is 30000 (30 seconds).

MCI_FAX_SET_ECM_LEVEL

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains the current Error Correction Mode (ECM) quality level. The higher quality telephone lines require less rigorous ECM checking. The following line quality levels are defined:

- MCI_FAX_ECM_POOR_LINE
- MCI_FAX_ECM_AVERAGE_LINE
- MCI_FAX_ECM_QUALITY_LINE
- MCI_FAX_ECM_NONE

MCI_FAX_SET_ECM_LEVEL not supported in current FAX driver

.

MCI_FAX_SET_EVENT_HANDLER

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains the handle of the application event handler. The MCI driver posts **MM_MCIEVENT** messages when an event occurs which changes the status of the driver. Setting this value to zero disables event posting. See the event handler section of the document for more details.

MCI_FAX_SET_HOOK

The **dwSetData** field is set to the desired hook status of the telephone line. It is set to TRUE to take the handset off-hook, and FALSE to place the handset on-hook. If another application owns the phone line, this call will fail. When an application sets **dwSetData** to FALSE, it relinquishes ownership of the line.

MCI_FAX_SET_MODEM_TYPES

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains the **desired** maximum and minimum FAX modem types ORed together for calls received and transmitted. If the actual negotiation speed is lower than the selected minimum modem type, the call is terminated. The following modem type flags are defined (in fall back order; from highest speed to lowest):

- MCI_FAX_MODEM_ANY
- MCI_FAX_MODEM_V17_14400

- MCI_FAX_MODEM_V17_12000
- MCI_FAX_MODEM_V17_9600
- MCI_FAX_MODEM_V17_7200
- MCI_FAX_MODEM_V29_9600
- MCI_FAX_MODEM_V29_7200
- MCI_FAX_MODEM_V27TER_4800
- MCI_FAX_MODEM_V27TER_2400

MCI_FAX_MODEM_ANY not supported in current FAX driver

MCI_FAX_SET_PASS_CALL

Specifies that the **dwSetData** field of the data structure identified by lParam2 contains a constant specifying the device to which the phone line should be passed. The line can only be passed from the fax driver when the MODE is OPEN (see MCI_STATUS_MODE). If the mode is not open, the application must do a MCI_STOP to reset the fax out of send or receive mode.
The possible values of dwSetData are:

- MCI_FAXTAM_PASS_VOICE
- MCI_FAXTAM_PASS_MODEM

MCI_FAX_SET_PASS_CALL is only supported when the discriminator is loaded.

MCI_FAXTAM_PASS_MODEM not supported in current driver.

MCI_FAX_SET_POLLING

Specifies that the **dwSetData** field of the data structure identified by *lParam2* is set to TRUE if the device is to be set to receive a FAX poll for this application. Both calling and called applications must issue this command followed by an MCI_RECEIVE to set up for polling.

MCI_FAX_SET_RESOLUTION

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains the resolution mode of the FAX device. This setting is used to tell the calling party the fax device’s capabilities (DIS info) for negotiating the receive. The possible values are:

- MCI_FAX_RESOLUTION_NORMAL
- MCI_FAX_RESOLUTION_FINE

MCI_FAX_SET_RING_COUNT

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the ring count at which the device should answer the telephone. The default ring count for FAX is 1.

If the discriminator is loaded, it will answer the telephone on the shortest ring count request of all registered applications (Windows), but never on less than two rings (OS/2). Caller ID can arrive between rings 1 and 2.

The maximum ring count that can be set is specified by MAX_RING_COUNT.

Note: If application is providing homologation support see MCI_STATUS for more information on the min and max ring count allowable.

MCI_FAX_SET_STATION_ID

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a pointer to a null terminated character string which gives the station identifier that is sent by the device during negotiation.

LPMCI_FAX_SET_PARMS *lParam2*

Specifies a far pointer to the following **MCI_FAX_SET_PARMS** data structure:

```
typedef struct {
      DWORD       dwCallback;
      DWORD       dwSetData;
      DWORD       dwItem;
} MCI_FAX_SET_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_STATUS

This command is used to obtain information about the FAX device configuration. Information is returned in the **dwReturn** field of the MCI_STATUS_PARMS structure, pointed to by *lParam2*.

Parameters DWORD *lParam1*

The following flags apply to the FAX device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_STATUS_ITEM

Specifies that the **dwItem** field of the data structure identified by *lParam2* contains a constant specifying which status item to obtain. The following constants are defined:

MCI_STATUS_LENGTH

The **dwReturn** field is set to the number of pages of the last Fax sent or received.

MCI_STATUS_MODE

The **dwReturn** field is set to the current mode of the device. The following modes are defined:

- MCI_MODE_NOT_READY
- MCI_MODE_OPEN
- MCI_MODE_RECEIVE
- MCI_MODE_SEND

MCI_STATUS_POSITION

The **dwReturn** field is set to the current number of pages received or sent.

MCI_STATUS_READY

The **dwReturn** field is set to TRUE if the device is ready; otherwise, it is set to FALSE. If another telephony application has ownership of the telephone line, this status command returns FALSE.

MCI_STATUS_TIME_FORMAT

The **dwReturn** field is set to the time format of the play/record media. This always returns MCI_FAX_FORMAT_PAGES.

MCI_FAX_STATUS_AUDIO_VOLUME

The **dwReturn** field of the data structure identified by *lParam2* returns a constant specifying the volume level of the speaker device. The volume level is specified from 0x0 (silence) to 0xFFFF (maximum volume) and is interpreted logarithmically. This means the perceived volume increase is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

MCI_FAX_STATUS_CALL_FILTER

The **dwReturn** field of the data structure identified by *lParam2* is set to TRUE if the device is currently set to receive fax calls; otherwise it is set to FALSE.

MCI_FAX_STATUS_COMPRESSION_TYPES

Specifies that the **dwReturn** field of the data structure identified by *lParam2* is set to the allowable FAX compression type(s) for calls received and sent. The following type flags are defined:

- MCI_FAX_COMPRESSION_MH
- MCI_FAX_COMPRESSION_MR
- MCI_FAX_COMPRESSION_MMR
- MCI_FAX_COMPRESSION_NONE
- MCI_FAX_COMPRESSION_BFT
- MCI_FAX_COMPRESSION_ANY

Only MH compression type supported in current FAX driver.

MCI_FAX_STATUS_DIAL_FLASH_TIME

The **dwReturn** field is set to the current flash time (in milliseconds) of the telephone flash option in the MCI_DIAL command.

MCI_FAX_STATUS_DIAL_PAUSE_TIME

The **dwReturn** field is set to the current pause time (in milliseconds) that an embedded ',' character produces in the dial string.

MCI_FAX_STATUS_DIAL_WAIT_TIME

The **dwReturn** field is set to the current time-out limit (in milliseconds) that an embedded 'w' character in the dial string allows, waiting for a second dial tone.

MCI_FAX_STATUS_ECM_LEVEL

Specifies that the **dwReturn** field of the data structure identified by *lParam2* is set to the current ECM quality level. The higher quality telephone lines require less rigorous ECM checking. The following line quality levels are defined:

- MCI_FAX_ECM_POOR_LINE
- MCI_FAX_ECM_AVERAGE_LINE
- MCI_FAX_ECM_QUALITY_LINE

- MCI_FAX_ECM_NONE

MCI_FAX_STATUS_ECM_LEVEL not supported in current FAX driver.

MCI_FAX_STATUS_HANDSET

The **dwReturn** field is set to the current status of the telephone handset. It is set to TRUE if the handset is off-hook; otherwise, it is set to FALSE.

MCI_FAX_STATUS_HANDSET not supported in current FAX driver.

MCI_FAX_STATUS_HANDSET_VOLUME

The **dwReturn** field of the data structure identified by lParam2 returns a constant specifying the volume level of the speaker device. The volume level is specified from 0x0 (silence) to 0xFFFF (maximum volume). The perceived volume increase is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

MCI_FAX_STATUS_HANDSET_VOLUME not supported in current FAX driver.

MCI_FAX_STATUS_HOOK

The **dwReturn** field is set to the current hook status of the telephone line. It is set to TRUE if the phone is off-hook; otherwise, it is set to FALSE.

MCI_FAX_STATUS_LINE

The **dwReturn** field is set to the current phone line status. The following status modes are defined:

- MCI_FAX_LINE_ONHOOK
- MCI_FAX_LINE_DIALTONE
- MCI_FAX_LINE_BUSY
- MCI_FAX_LINE_RINGTONE
- MCI_FAX_LINE_FAX_CARRIER
- MCI_FAX_LINE_UNKNOWN

MCI_FAX_STATUS_MAX_MODEM_SPEED

Specifies that the **dwReturn** field of the data structure identified by *lParam2* is set to the highest speed FAX modem type desired for calls received and sent. The following modem type flags are defined (in order from highest speed to lowest):

- MCI_FAX_MODEM_ANY
- MCI_FAX_MODEM_V17_14400
- MCI_FAX_MODEM_V17_12000
- MCI_FAX_MODEM_V17_9600
- MCI_FAX_MODEM_V17_7200

- MCI_FAX_MODEM_V29_9600
- MCI_FAX_MODEM_V29_7200
- MCI_FAX_MODEM_V27TER_4800
- MCI_FAX_MODEM_V27TER_2400

MCI_FAX_MODEM_ANY not supported in current FAX driver

MCI_FAX_STATUS_MIN_MODEM_SPEED

Specifies that the **dwReturn** field of the data structure identified by *lParam2* is set to the lowest speed FAX modem type desired for calls received and sent. The following modem type flags are defined (in order from lowest speed to highest):

- MCI_FAX_MODEM_ANY
- MCI_FAX_MODEM_V17_14400
- MCI_FAX_MODEM_V17_12000
- MCI_FAX_MODEM_V17_9600
- MCI_FAX_MODEM_V17_7200
- MCI_FAX_MODEM_V29_9600
- MCI_FAX_MODEM_V29_7200
- MCI_FAX_MODEM_V27TER_4800
- MCI_FAX_MODEM_V27TER_2400

MCI_FAX_MODEM_ANY not supported in current FAX driver

MCI_FAX_STATUS_POLLING

Specifies that the **dwReturn** field of the data structure identified by *lParam2* is set to TRUE if the device is set for fax polling for this application. Otherwise this value is FALSE.

MCI_FAX_STATUS_RESOLUTION

Specifies that the **dwReturn** field of the data structure identified by *lParam2* is set to the resolution mode of the FAX device. The possible return values are:

- MCI_FAX_RESOLUTION_NORMAL
- MCI_FAX_RESOLUTION_FINE

MCI_FAX_STATUS_RING_COUNT

The **dwReturn** field is set to a constant specifying the ring count at which the device answers the telephone. The driver answers on the shortest ring count request of all active applications, so this value might not match the value specified in MCI_SET.

MCI_FAX_STATUS_STATION_ID

The **dwReturn** field of the data structure identified by *lParam2* contains a pointer to a null terminated character string containing the station identifier.

MCI_FAX_STATUS_WORLDTRADE_SUPPORT

The **dwReturn** field is set to a binary encoded set of values indicating restrictions that are in effect for the current country. Some of the bit settings require the application to make a subsequent MCI_STATUS call to determine a maximum value. This support is added with driver version 3.4. The defined bits include:

- PULSE_DIAL_NOT_ALLOWED is set TRUE if pulse dialing is not supported.
- DTMF_DIAL_NOT_ALLOWED is set TRUE if DTMF dialing is not supported.
- BUSYTONE_DETECT_NOT_VALID is set TRUE if busy tone detection is not available in the country.
- BUSYTONE_DETECT_REQUIRED is set TRUE if busy tone detection is required in country.
- DIALTONE_DETECT_NOT_VALID is set TRUE if dial tone detection is not available in the country.
- DIALTONE_DETECT_REQUIRED is set TRUE if dial tone detection is required in country.

MCI_FAX_STATUS_COUNTRY_CODE

The **dwReturn** field is set to the current country code. This can be used by applications that must change the looks of the user interface for different countries like a French keypad in France. This support is added with driver version 3.4. The following table shows the codes assigned to each country:

COUNTRY	CODE
USA/Canada	1
Belgium	2
Hong Kong	3
Singapore	4
New Zealand	5
Japan	6
Portugal	7
Ireland	8
Generic	9
Spain	10
Greece	11
Israel	12
Taiwan	13
Australia	14
Austria	15
Mexico	16
South Africa	17
Chile	18
Switzerland	19
Germany	20
Brazil	21
Russia	22
Yugoslavia	23
Hungary	24
Czechrepublic	25
Luxembourg	26
Norway	27
Denmark	28
France	29
Netherlands	30
U. K.	31
Sweden	32
Italy	33
Finland	34
Thailand	35
Korea	36
Malaysia	37
PRC	38
Slovakia	39

TABLE 6-3: Country Codes

MCI_FAX_STATUS_AUTO_ANSWER_MIN_RINGS

The **dwReturn** field contains the minimum number of rings that can be set in MCI_FAX_SET_RING_COUNT. This support is added with driver version 3.4.

MCI_FAX_STATUS_AUTO_ANSWER_MAX_RINGS

The **dwReturn** field contains the maximum number of rings can be set in MCI_FAX_SET_RING_COUNT. If the value is '7FFF'x then there is no limit in that country. This support is added with driver version 3.4.

MCI_FAX_STATUS_MAX_CALL_RETRIES

The **dwReturn** field contains the maximum number of unsuccessful retries allowed. If the value is '7FFF'x there is no max in that country. This support is added with driver version 3.4.

MCI_FAX_STATUS_MIN_CALL_RETRY_TIME

The **dwReturn** field contains the minimum time allowed between retries. This support is added with driver version 3.4.

LPMCI_STATUS_PARMS *lParam2*

Specifies a far pointer to the following **MCI_STATUS_PARMS** data structure:

```
typedef struct {
      DWORD        dwCallback;
      DWORD        dwReturn;
      DWORD        dwItem;
} MCI_STATUS_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_STOP

This command is used to cancel a fax send or a fax receive.

Parameters DWORD *lParam1*

The following flags apply to the FAX device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lParam2*

Specifies a far pointer to the following **MCI_GENERIC_PARMS** data structure:

```
typedef struct {
      DWORD       dwCallback;
} MCI_GENERIC_PARMS;
```

Note: It is necessary to wait for PHONE_EVENT_CALL_TERMINATED before hanging up the phone.

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

Chapter 7 - TAM API Reference

This chapter is a complete reference to the Mwave TAM Application Program Interface (API).

TAM functionality is provided by two separate, but related, drivers: TAM Phone Line and TAM Message.

- The Phone Line driver is used for all operations involving the phone line. This includes playing a recorded message to the phone line, recording a message from the phone line, initiating calls, answering calls and speakerphone operation.
- The Message driver includes supports all TAM operations that do not involve the phone line. This includes local (i.e. phone line not used) recording and playing of messages.

This chapter is divided into two parts. The first part describes the event messages issued by the two drivers and the second part describes the API messages and flags for the two drivers. For the most part, the event messages and API messages/flags are identical for the two drivers. Where the description is specific to a particular driver, the description is marked as follows:

MSG	Applicable to Message driver only
PL	Applicable to Phone Line driver only

Descriptions containing neither mark are applicable to both drivers.

MCI Telephone Event Handler

Communication of real-time status information from the TAM driver to the application is performed through an application event handler. The handler should be able to service messages posted by the TAM driver through the MCI device, which contain real-time status information about the device.

MM_MCIEVENT, is not a standard MCI message under *Microsoft Windows*, thus a Microsoft Windows application must call the **RegisterWindowMessage** function with the string "MM_MCIEVENT", to obtain the numeric value of the notification message.

MM_MCIEVENT - Windows

In addition to the message itself, *wParam* and *lParam* are used to pass information to the application.

WPARAM *wParam*
Contains a device specific event message ***wEvent***.

LPMCI_EVENT_PARMS *lParam*
Specifies a far pointer to the following MCI_EVENT_PARMS structure:

```
typedef struct {
    DWORD  dwDataParam1;
    DWORD  dwEventData;
} MCI_EVENT_PARMS;
```

The data parameters are defined as follows:

DWORD *dwDataParam1*
The low-order word specifies the device specific event message ***wEvent*** (same as *wParam*). The high-order word specifies the device ID of the device initiating the message.

DWORD *dwEventData*
Contains a data parameter, which is dependent on the message type. The actual parameters passed are listed in Table **Error! Bookmark not defined.** below, and detailed in the event message descriptions.

MM_MCIEVENT - OS/2

In addition to the message itself, *wParam* and *lParam* are used to pass information to the application.

DWORD *MsgParam1*
Contains a device-specific event message and device ID.

WORD *wEvent*
The low-order word of *MsgParam1* specifies the device-specific event
code
(same as *usEventCode* or *wParam*)

WORD *wDeviceID*
The high-order word of *MsgParam1* specifies the device ID of the device initiating the message.

LPMCI_EVENT_PARMS *MsgParam2*

```
typedef struct {
  DWORD   dwDataParam1;
  DWORD   dwEventData;
} MCI_EVENT_PARMS;
```

Note: The low-order word of **dwDataParam1** contains the event code (same as *wEvent*). The high-order word is not defined.

TAM Event Message Descriptions

This section describes the Event Messages generated by the TAM API. The following table lists the Event Messages (wEvent), a short description of the data parameters, and the associated drivers.

Event Message (wEvent)	Data Parameter (dwEventData)	Driver(s)
PHONE_EVENT_ADVANCED_RING	undefined, use lParam	PL
PHONE_EVENT_CALL_PROGRESS	New call state	PL
PHONE_EVENT_CALL_TAM	undefined	PL
PHONE_EVENT_CALL_TERMINATED	Call termination status	PL
PHONE_EVENT_CALLER_ID	Caller ID Status	PL
PHONE_EVENT_DISTINCTIVE_RING	Ring Identifier	PL
PHONE_EVENT_HANDSET	Handset Status	PL, MSG
PHONE_EVENT_HANDSET_KEY	Keypress character	MSG
PHONE_EVENT_LINE	Telephone line status	PL
PHONE_EVENT_LINE_KEY	Keypress character	PL
PHONE_EVENT_RING	Telephone ring status	PL

Table 7-1: TAM Driver Event Messages

For all messages posted to the event handler routine, the message value is **MM_MCIEVENT**. The value of *wEvent* and *dwEventData* vary according to the specific message posted. Below is a more detailed description of the event messages and their parameters.

Arguments wEvent: **PHONE_EVENT_ADVANCED_RING** **PL**
dwEventData: not used, actual ring count is in lParam

Description If the application has requested ‘advanced format ring notifications’ by setting advanced ring notify to TRUE, PHONE_EVENT_ADVANCED_RING is sent to the application instead of PHONE_EVENT_RING. In this case, lParam is not a pointer to a structure. Instead, the low word of lParam contains the ring count, and the high word of lParam contains the device ID.

LOWORD(lParam) = 0 Telephone ring signal end (not ringing)
LOWORD(lParam) = ‘n’ Telephone ring count (where ‘n’ is the ring number)

Arguments *wEvent:* **PHONE_EVENT_CALL_PROGRESS** **PL**
dwEventData: **dwCallProgress**

Description This message is posted when there has been a change in the current call state (or status). The new state of the call is supplied in dwCallProgress, and can be any of the following:

- CALL_PROGRESS_ANSWER_TONE (supported in version 3.0 and above of the TAM driver)

- CALL_PROGRESS_BUSY (in current driver, returned for both Fast Busy and Slow Busy)
- CALL_PROGRESS_DIAL_TONE
- CALL_PROGRESS_FAST_BUSY (unsupported in current driver)
- CALL_PROGRESS_QUIET
- CALL_PROGRESS_REMOTE_RINGING (supported in version 3.0 and above of the TAM driver)
- CALL_PROGRESS_SLOW_BUSY (unsupported in current driver)
- CALL_PROGRESS_UNIDENTIFIED_TONE

Arguments	***wEvent:***	**PHONE_EVENT_CALL_TAM**	**PL**
	dwEventData:	**undefined**	

Description This message is posted when a call has been answered by the device, and has been determined to have originated from a voice source. At this time, the application can play a greeting and begin voice mail operations.

Arguments	*wEvent:*	**PHONE_EVENT_CALL_TERMINATED**	**PL**
	dwEventData:	**dwTermination**	

Description This message is posted when a call has been terminated either by the caller, by the owning application, or because of an error condition. The reason for call termination is given in **dwTermination**, which can be any of the following values:

- TERMINATION_ERROR_RECV
- TERMINATION_ERROR_XMIT
- TERMINATION_NORMAL
- TERMINATION_REQUESTED (returned when the Discriminator is handing call off to a different driver)
- TERMINATION_UNEXPECTED (returned if the PC goes into power saving mode in the middle of a call)

Arguments	*wEvent:*	**PHONE_EVENT_CALLER_ID**	**PL**
	dwEventData:	**dwCompStatus**	

Description This message is posted when a caller ID string has been decoded off a ringing line. It is posted only if a caller ID signal is present. **dwCompStatus** indicates the completion status.

- MCI_VALID_CALLER_ID_RECEIVED
- MCI_CALLER_ID_FRAME_ERROR

The application must issue an MCI_INFO message to retrieve the id (for MCI_VALID_CALLER_ID_RECEIVED) or the error code (for MCI_CALLER_ID_FRAME_ERROR).

Arguments *wEvent:* **PHONE_EVENT_DISTINCTIVE_RING** **PL**
dwEventData: **dwRingIdentifier**

Description This message is posted when a distinctive ring has been decoded off a ringing line. It is posted only if distinctive ring support is installed. **dwRingIdentifier** indicates which distinctive ring has been decoded. The ring identifier is a number between 1 and 20. This support is added with Ver 3.2.

Arguments *wEvent:* **PHONE_EVENT_HANDSET**
dwEventData: **dwHandsetStatus**

Description This message is posted when the status of the telephone handset changes, due to the user either picking up or replacing the telephone handset. This message can be monitored to play an automatic greeting when the handset is removed from the cradle. The value of **dwHandsetStatus** is as follows:

dwHandsetStatus = 0	Handset is on-hook
dwHandsetStatus = 1	Handset is off-hook (in use)

Arguments *wEvent:* **PHONE_EVENT_HANDSET_KEY**
dwEventData: **dwKeypress**

Description This message is posted when a key has been pressed on the handset device. An ASCII character representing the pressed key ('0' - '9', 'a' - 'd', '#', '*', '!') is supplied in **dwKeypress**.

The current PL driver reports only '!'. The '!' (flash) is reported only if the application has set the min and/or max flash time.

Arguments *wEvent:* **PHONE_EVENT_LINE** **PL**
dwEventData: **dwLineStatus**

Description This message is posted when the status of the telephone line changes, due to another application in the system making use of the telephone line. When an application takes the telephone line off hook, or is called to service an incoming call, it remains in possession of the telephone line for the duration of the call. Applications which require use of the telephone line and find it busy, can simply wait for this message to signal that the telephone line may be used. The value of **dwLineStatus** is as follows:

dwLineStatus = 0	Telephone line is free
dwLineStatus = 1	Telephone line is in use

Arguments *wEvent:* **PHONE_EVENT_LINE_KEY** **PL**
dwEventData: **dwKeypress**

Description This message is posted when a key has been pressed on the incoming telephone line. An ASCII character representing the pressed key ('0' - '9', 'a' - 'd', '#' or '*'), is supplied in **dwKeypress**.

Arguments *wEvent:* **PHONE_EVENT_RING** **PL**
dwEventData: **dwRingStatus**

Description This message is posted when a ring signal change is detected by the device. This message can be used by the application to count ring cycles, or determine ring length. The value of **dwRingStatus** is as follows:

dwRingStatus = 0	Telephone ring signal end (not ringing)
dwRingStatus = 1	Telephone ring signal start (ringing)

TAM Driver API Messages and Flags

This section describes the MCI compliant TAM API messages and flags. The following table lists MCI command messages used in the TAM API, a short description of the message and the associated drivers.

MCI Message	**Description**	**Drivers**
MCI_CLOSE	Close the device driver	PL, MSG
MCI_CONVERT	Convert from/to device dependent file to/from device independent file.	MSG
MCI_DIAL	Dial the telephone	PL
MCI_GETDEVCAPS	Get the capabilities of the device	PL, MSG
MCI_INFO	Get device string identifier	PL, MSG
MCI_LOAD	Load a *voice or wave* file for playing	PL, MSG
MCI_OPEN	Open the device driver	PL, MSG
MCI_PAUSE	Pause the *voice or wave* stream play or record	PL, MSG
MCI_PLAY	Play a *voice or wave* file	PL, MSG
MCI_RECORD	Record a *voice or wave file*	PL, MSG
MCI_RESUME	Resume a paused *voice or wave* stream	PL, MSG
MCI_SAVE	Save a recorded *voice or wave* file	PL, MSG
MCI_SEEK	Change the current position of the media	PL, MSG
MCI_SET	Configure the device	PL, MSG
MCI_STATUS	Query device configuration	PL, MSG
MCI_STOP	Stop a *voice or wave* stream	PL, MSG

Table 7-2: TAM Driver API Messages

MCI_CLOSE

This command message closes the TAM driver.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY
Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT
Specifies that the operation should finish before MCI returns control to the application. The event handler window must be specified in the **dwCallback** field regardless of whether MCI_NOTIFY or MCI_WAIT is selected.

LPMCI_GENERIC_PARMS *lParam2*

Specifies a far pointer to the following **MCI_GENERIC_PARMS** data structure:

```
typedef struct {
   DWORD         dwCallback;
} MCI_GENERIC_PARMS;
```

Note: Be sure to assign the handle of the window procedure responsible for processing MM_MCINOTIFY messages to dwCallback prior to calling MCI_CLOSE regardless of whether MCI_WAIT or MCI_NOTIFY is specified. Failure to do so results in erratic behavior when using versions earlier than Ver 2.1 of the TAM device driver.

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_CONVERT MSG

This command message is used to convert data files between an MCI device dependent format, and a standard device independent format. The call is used to convert to and from device dependent format (PCM Wave in the case of Microsoft Windows) and TAM compressed voice files.

MCI_CONVERT is intended to be run ‘off-line’ as it consumes a fair amount of MIPS, and conversion time is the same as the duration of the file being converted.

MCI_CONVERT is supported in TAM drivers version 3.1 and above. Support may not be installed on a system even if the driver version is 3.1 or above. MCI_CONVERT will return a non-zero return code if it is not supported. An AP can issue MCI_GETDEVCAPS to determine if wave file support is installed.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*. MCI_NOTIFY should be specified unless MCI_CONVERT_INFO is specified.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_CONVERT_CREATE

Indicates that the destination file is a new file which should be created. This will overwrite any existing file.

MCI_CONVERT_DESTINATION_FILE

Indicates the **lpstrDestFilename** field of the data structure identified by *lParam2* contains a pointer to a buffer containing the destination file name.

MCI_CONVERT_DESTINATION_FORMAT

Indicates the **dwDestFormat** field of the data structure identified by *lParam2* contains the desired format of the destination file. These include:

- MCI_CONVERT_FMT_WAVE_PCM *(from source of type DEVTAM)*
- MCI_TAM_CONVERT_FMT_DEVTAM *(from WAVE_PCM)*

MCI_CONVERT_DESTINATION_FROM

Specifies that a media starting position is included in the **dwDestFrom** field of the data structure identified by *lParam2*. This specifies the starting point at which the converted data is written to an existing destination file. This option is not supported with the MCI_CONVERT_CREATE option. For TAM, the index is in units of milliseconds.

MCI_CONVERT_INFO

Indicates that no conversion operation is to take place, but rather, the **dwLength** field of the data structure identified by *lParam2* should be set to the length of the media of the supplied source device dependent filename. For a device dependent TAM file, the value is returned in milliseconds. If a device dependent file is not specified, this call returns an error.

MCI_CONVERT_LENGTH

Indicates that the **dwLength** field if the structure identified by *lParam2* contains a value specifying the length of the media to be converted. If this value is not supplied, the entire media is converted from the starting index. For TAM, the length is expressed in units of milliseconds.

MCI_CONVERT_OVERWRITE

Indicates that newly converted information should overwrite any existing data. If this flag is not specified, the new data is inserted into the file.

MCI_CONVERT_SOURCE_FILE

Indicates the **lpstrSrcFilename** field of the data structure identified by *lParam2* contains a pointer to the source file name.

MCI_CONVERT_SOURCE_FROM

Specifies that a media starting position is included in the **dwSrcFrom** field of the data structure identified by *lParam2*. This specifies the starting point at which the data to be converted is read from the source file. For TAM, the index is in units of milliseconds.

LPMCI_CONVERT_PARMS *lParam2*

Specifies a far pointer to the following **MCI_CONVERT_PARMS** data structure:

```
typedef struct {
    DWORD         dwCallback;
    LPCSTR        lpstrDestFilename;
    DWORD         dwDestFormat;
    DWORD         dwDestFrom;
    DWORD         dwLength;
    LPCSTR        lpstrSrcFilename;
    DWORD         dwSrcFrom;
} MCI_CONVERT_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_DIAL PL

This command message takes the phone off-hook, and dials the supplied number. If the telephone is owned by another application at the time of this call, the command will fail.

Parameters DWORD *lParam1*

The following flags apply to the telephone device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_DIAL_DIALMODE

Specifies that the **dwDialMode** field of the data structure identified by *lParam2* contains a constant specifying the phone dialing mode. Two modes are defined:

- MCI_DIAL_MODE_PULSE
- MCI_DIAL_MODE_TONE

MCI_DIAL_FLASH

Indicates that the telephone should be flashed before dialing the supplied number (if any).

MCI_DIAL_MONITOR

Specifies that the audio speaker device should be enabled during the calling process.

MCI_DIAL_STRING

Specifies that the **lpstrDialString** field of the data structure identified by *lParam2* contains a pointer to a null terminated dialing string. Numeric characters '0' to '9' correspond to phone digits. The '*' and '#' characters, the alpha characters 'a' to 'd' and the '-' are also supported ('-' is ignored).

The 'w' character in the string specifies that the device should wait for a second dial tone before proceeding, and a ',' character indicates a pause in the dialing sequence. The time-out limit for the wait command (default 30 seconds) and the delay time for the pause command (default 2 seconds) are configurable using MCI_SET. The ‘@’ character in the string specifies wait for quiet. The ‘p’ character in the string specifies switch to pulse dialing. The ‘t’ character in the string specifies switch to tone dialing. The ‘!’ character in the string specifies *flash* the line. Note that the setting of flash time has no effect on the duration of *flash* that is specified with a ‘!’ in the dial string. That setting only has effect on the *flash* that occurs as a result of the MCI_DIAL_FLASH flag.

The maximum size string that can be dialed is specified by MAX_DIAL_STRING.

MCI_DIAL_VERIFY

Specifies that the call is to be verified. The phone is verified to be off-hook, and that a dial tone is present before dialing.

LPMCI_DIAL_PARMS *lParam2*

Specifies a far pointer to the following **MCI_DIAL_PARMS** data structure:

```
typedef struct {
   DWORD        dwCallback;
   DWORD        dwDialMode;
   LPCSTR lpstrDialString;
} MCI_DIAL_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_GETDEVCAPS

This command is used to obtain static information about a device.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY
Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT
Specifies that the operation should finish before MCI returns control to the application.

MCI_GETDEVCAPS_ITEM
Specifies that the **dwItem** field of the data structure identified by *lParam2* contains a constant specifying which device capability to obtain. The following constants are defined:

MCI_GETDEVCAPS_CAN_EJECT
The **dwReturn** field is set to FALSE.

MCI_GETDEVCAPS_CAN_PLAY
The **dwReturn** field is set to TRUE if the device supports playing voice files to the speaker, handset, or telephone line. Otherwise, it is set to FALSE.

MCI_GETDEVCAPS_CAN_RECORD
The **dwReturn** field is set to TRUE if the device supports voice recording from the microphone, handset, or telephone line. Otherwise, it is set to FALSE.

MCI_GETDEVCAPS_CAN_SAVE
The **dwReturn** field is set to TRUE if the device supports saving voice data recorded from the microphone, handset, or telephone line. Otherwise, it is set to FALSE.

MCI_GETDEVCAPS_COMPOUND_DEVICE
The **dwReturn** field is set to FALSE prior to Ver 3.1. For Ver 3.1 and beyond, it is set TRUE.

MCI_GETDEVCAPS_DEVICE_TYPE
The **dwReturn** field is set to MCI_DEVTYPE_OTHER.

MCI_GETDEVCAPS_HAS_AUDIO
The **dwReturn** field is set to TRUE if the device supports play and record through an external audio device (speaker and microphone). Otherwise, it is set to FALSE.

MCI_GETDEVCAPS_HAS_VIDEO
The **dwReturn** field is set to FALSE.

MCI_GETDEVCAPS_USES_FILES

The **dwReturn** field is set to TRUE if the device supports voice recording or playing. Otherwise, it is set to FALSE.

MCI_TAM_GETDEVCAPS_SUPPORTS_CUSTOM_TAG

The **dwReturn** field is set to TRUE if the TAM operations support custom audio file formats. These formats are intended to save disk space over the conventional PCM wave file format.

MCI_TAM_GETDEVCAPS_SUPPORTS_PCM_TAG

The **dwReturn** field is set to non-zero if the TAM operations support the use of standard PCM wave files in its play and record operations: Otherwise it is set FALSE. See MCI_SET for MCI_TAM_SET_LOW_LEVEL_WAVE_IO for related information.

LPMCI_GETDEVCAPS_PARMS *lParam2*

Specifies a far pointer to the following **MCI_GETDEVCAPS_PARMS** data structure:

```
typedef struct {
   DWORD        dwCallback;
   DWORD        dwReturn;
   DWORD        dwItem;
} MCI_GETDEVCAPS_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_INFO

This command message obtains string information from the device.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_INFO_PRODUCT

Obtains a description of the hardware associated with a device. The description identifies both the driver and the hardware used. The string is copied to the buffer pointer to by the **lpstrReturn** field of the structure identified by *lParam2*. The size of this buffer is specified by the **dwRetSize** field of the same structure, and if the buffer is of insufficient size to contain the string, the string is truncated to fit the buffer. The string contains a version number (i.e., “Ver 3.0”). Driver enhancements will be denoted in this document with the “Ver x.y”. that corresponds with the first release that the feature shows up in. The version number will always increase in future releases, so a program can parse the string, looking for “Ver “, convert the characters that follow “Ver “ to a number, and do a numeric greater-than-or-equal compare to determine if the function is available in the release the application is running with.

Note: Unless otherwise noted, all functions are available as of Ver 2.2

MCI_INFO_CALLER_ID

Obtains a caller ID string. (See PHONE_EVENT_CALLER_ID). The string is copied into the buffer pointed to by the **lpstrReturn** (Windows) or **dwReturn** (OS/2) field of the structure identified by *lParam2*. The size of this buffer is specified by the **dwRetSize** field of the same structure (maximum size = MCI_MAX_CALLER_SIZE). If the buffer is of insufficient size to contain the string, the string is truncated to fit the buffer.

The caller ID data is in the format defined by Bellcore's technical reference bulletin TR-TSY-000031 and TR-NWT-001188. Also note that a checksum is included at the end of the Caller ID data.

MCI_INFO_CALLER_ID_ERROR

Obtains the caller ID error code. (See PHONE_EVENT_CALLER_ID). The code is copied into the buffer pointed to by the **lpstrReturn** (Windows) or **dwReturn** (OS/2)field of the structure identified by *lParam2*. The size of this buffer is specified by the **dwRetSize** field of the same structure. The error code is either MCI_FRAME_ERROR or MCI_CHECKSUM_ERROR.

MCI_INFO_CALLER_PARSED_CALLER_ID

Obtains an already-parsed Caller ID string. (See PHONE_EVENT_CALLER_ID). The information is copied into the structure pointed to by the **lpstrReturn** (Windows) or **dwReturn** (OS/2) field of the structure identified by *lParam2*. The structure is:

```
typedef struct
{
   char szDateTime[DATE_TIME_LEN+1];
   char szNumber[MCI_MAX_CALLER_ID_SIZE]; /* callers number */
   char szName[MCI_MAX_CALLER_ID_SIZE];    /* callers name (may
                                              be null) */
} CIDINFO;
```

If the call doesn't not have a caller ID, szName will be 'out of area caller' if one phone system doesn't support delivering caller ID to another phone system, or 'private caller' if the caller blocked the sending of caller ID.

This function is implemented in "Ver 3.0" of the TAM driver.

LPMCI_INFO_PARMS *lParam2*

Specifies a far pointer to the following **MCI_INFO_PARMS** data structure:

```
typedef struct {
    DWORD         dwCallback;
    LPSTR         lpstrReturn;
    DWORD         dwRetSize;
} MCI_INFO_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_LOAD

This command message loads a file, and the data used as the current media. The current position is set to the start of the media.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY
Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT
Specifies that the operation should finish before MCI returns control to the application.

MCI_LOAD_FILE
Indicates the **lpstrFilename** field of the data structure identified by *lParam2* contains a pointer to a buffer containing the file name.

To open a new file, you can either:

- Specify the MCI_LOAD_FILE and a null pointer for the file name.
- If running with API style set to MCI_FAXTAM_STYLE_MMPM, omit the MCI_LOAD_FILE flag.

In driver version 3.1, the ability to play and record wave files over the telephone is added. If the file extension is 'wav' the file is assumed to be a wave file. If a new file is loaded, the value set by MCI_SET MCI_TAM_SET_FORMATTAG is used to determine that the file is a wave file. The default setting is TAM_WAVE_FORMAT_CUSTOM. Wave file support is a separately installable option that may not be installed on a particular machine. If it is not installed, the application will get a non-zero return code on MCI_LOAD, MCI_PLAY or MCI_RECORD. The TAM application can issue MCI_GETDEVCAPS for MCI_TAM_GETDEVCAPS_SUPPORTS_PCM_TAG to determine if support is installed. Recording to wave files is not recommended as it takes much more disk space than recording to the custom formatted files.

Use **LOADFILENAME** instead of **lpstrFilename** (Windows) or **pszElementName** (OS/2). This label makes it easier to port applications between Windows and OS/2.

MCI_OPEN_ELEMENT
This flag is defined in mciftdd.h to be identical **MCI_LOAD_FILE**.

LPMCI_LOAD_PARMS *lParam2*

Specifies a far pointer to the following **MCI_LOAD_PARMS** data structure:

```
typedef struct {
   DWORD        dwCallback;
   LPCSTR lpstrFilename;
} MCI_LOAD_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_OPEN

This command message initializes the telephony driver and hardware.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY
Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT
Specifies that the operation should finish before MCI returns control to the application. In older versions of the driver, the event handler window must be specified in the **dwCallback** field regardless of whether MCI_NOTIFY or MCI_WAIT is selected.

MCI_OPEN_ALIAS
Specifies that an alias is included in the **lpstrAlias** field of the data structure identified by *lParam2*. This command is handled by MCI.

MCI_OPEN_ELEMENT
Specifies that a filename is included in the **lpstrElementName** field of the data structure identified by *lParam2*. The file is loaded as part of MCI_OPEN processing. This function is new to driver version 3.1.

MCI_OPEN_SHAREABLE
Specifies that the device should be opened as shareable.

MCI_OPEN_SHAREABLE is not supported in current TAM drivers.

MCI_OPEN_TYPE
Specifies that a device type name or constant is included in the **lpstrDeviceType** field of the data structure identified by *lParam2*. This command is handled by MCI. To open the telephone message driver, specify "Mwavetps" in the **lpstrDeviceType**. To open the telephone line driver, specify "Mwavetpl".

MCI_OPEN_TYPE_ID *(Not supported in OS/2; defined as zero)*
Specifies that the low-order word of the **lpstrDeviceType** field of the associated data structure contains a standard MCI device type ID and the high-order word optionally contains the ordinal index for the device. This command is handled by MCI.

LPMCI_OPEN_PARMS *lParam2*

Specifies a far pointer to the following **MCI_OPEN_PARMS** data structure:

```
typedef struct {
   DWORD         dwCallback;
   WORD          wDeviceID;
   WORD          wReserved0;
   LPCSTR        lpstrDeviceType;
   LPCSTR        lpstrElementName;
   LPCSTR        lpstrAlias;
} MCI_OPEN_PARMS;
```

Note: With Microsoft Windows, be sure to assign the handle of the window procedure responsible for processing MM_MCINOTIFY messages to dwCallback prior to calling MCI_OPEN regardless of whether MCI_WAIT or MCI_NOTIFY is specified. Failure to do so results in erratic behavior when using versions earlier than Ver 2.1 of the TAM device driver.

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

Remarks
Case is ignored in the device name, but there must not be any leading or trailing blanks.

Note that the device type is the **pszDeviceType** field of the **MCI_OPEN_PARMS** data structure, but it does not have a corresponding flag because it is required and does not have a command-string parameter.

For the Mwave Fax and TAM drivers, the device types are:

Mwavetpl
Mwavetps
Mwavefax

MCI_PAUSE

This command message pauses the current **MCI_PLAY** or **MCI_RECORD** operation.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY
Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT
Specifies that the operation should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lParam2*

Specifies a far pointer to the following **MCI_GENERIC_PARMS** data structure:

```
typedef struct {
    DWORD         dwCallback;
} MCI_GENERIC_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_PLAY

This command message plays the current media on the connected device(s).

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

If an MCI_NOTIFY_ABORTED is posted with the notification, the call discriminator determined that the call was not a voice call. The application can now wait for the next incoming call.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_FROM

Specifies that a media starting position is included in the **dwFrom** field of the data structure identified by *lParam2*. The units assigned to the position values are milliseconds (MCI_FORMAT_MILLISECONDS). If MCI_FROM is not specified, the current position in the media is used.

MCI_TO

Specifies that a media ending position is included in the **dwTo** field of the data structure identified by *lParam2*. The units assigned to the position values are milliseconds (MCI_FORMAT_MILLISECONDS). If MCI_TO is not specified, the device plays to the end of the media.

LPMCI_PLAY_PARMS *lParam2*

Specifies a far pointer to the following **MCI_PLAY_PARMS** data structure:

```
typedef struct {
   DWORD         dwCallback;
   DWORD         dwFrom;
   DWORD         dwTo;
} MCI_PLAY_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_RECORD

This command message records the connected device(s) to the current media.

The ability to record a conversation is added in version 3.1 of the TAM driver. This occurs if MCI_RECORD is issued when the handset is connected to the phoneline, and the handset is up, or if speakerphone is in use when MCI_RECORD is issued. If the conversation is being recorded, the remote party will hear periodic beeps to indicate that the conversation is being recorded. If the user wishes to change connections (i.e., from handset to microphone during conversation recording) it is necessary for the application to issue MCI_STOP before issuing MCI_SET MCI_TAM_SET_CONNECT. After the MCI_SET is complete, the application should issue MCI_RECORD without specifying MCI_FROM to continue recording from the position where the initial recording stopped

With Ver 3.1, the ability to record PCM files is supported. However, PCM files take up more disk space than the default TAM sub-band-coded files. Also, when recording PCM files over the phone line, neither silence nor dialtones are automatically removed from the recorded file. With sub-band-coded files, recording over the phone is automatically terminated when the call is complete. With PCM files, the application must issue MCI_STOP to terminate the record.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

If MCI_NOTIFY_ABORTED is posted with the notification, the call discriminator determined that the call was not a voice call. The application should not save the recorded file. If MCI_NOTIFY_FAILURE is reported, it probably indicates that nothing but silence was recorded. There is no reason to save the recorded file.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_RECORD_INSERT

Indicates that newly recorded information should be inserted or pasted into the existing media data.

MCI_FROM

Specifies that a media starting position is included in the **dwFrom** field of the data structure identified by *lParam2*. The units assigned to the position values are milliseconds (MCI_FORMAT_MILLISECONDS). If MCI_FROM is not specified, the current position in the media is used.

MCI_RECORD_OVERWRITE

Specifies that newly recorded data should overwrite existing data.

MCI_TO

Specifies that a media ending position is included in the **dwTo** field of the data structure identified by *lParam2*. The units assigned to the position values are mS (MCI_FORMAT_MILLISECONDS). If MCI_TO is not specified, the device records to the end of the media (a substantial amount of time in TAM).

MCI_TAM_BEEP

Specifies that a 500 Hz tone of 0.5 second duration should be played before recording begins.

MCI_TAM_TO_MESSAGE_END

Specifies that the device should record until it detects the end of the message, and then truncates prolonged silence or dial tone periods from the newly recorded media. The MCI_TAM_TO_MESSAGE_END flag should always be set when not using the MCI_TO option.

LPMCI_RECORD_PARMS *lParam2*

Specifies a far pointer to the following **MCI_RECORD_PARMS** data structure:

```
typedef struct {
   DWORD        dwCallback;
   DWORD        dwFrom;
   DWORD        dwTo;
} MCI_RECORD_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_RESUME

This command message resumes the current **MCI_PLAY** or **MCI_RECORD** operation, after a **MCI_PAUSE** operation has been issued.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lParam2*

Specifies a far pointer to the following **MCI_GENERIC_PARMS** data structure:

```
typedef struct {
   DWORD         dwCallback;
} MCI_GENERIC_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_SAVE

This command message saves the current media to a file, retaining its current format via the format tag.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY
Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT
Specifies that the operation should finish before MCI returns control to the application.

MCI_SAVE_FILE
Indicates the **lpstrFilename** field of the data structure identified by *lParam2* contains a pointer to a buffer containing the file name where the current media data is saved.

LPMCI_SAVE_PARMS *lParam2*

Specifies a far pointer to the following **MCI_SAVE_PARMS** data structure:

```
typedef struct {
   DWORD        dwCallback;
   LPCSTR lpstrFilename;
} MCI_SAVE_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_SEEK

This MCI command message changes the current position of the media. Audio output is disabled during the seek. After the seek completes, the device stops.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY
Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT
Specifies that the operation should finish before MCI returns control to the application.

MCI_SEEK_TO_END
Specifies that the device should seek to the end of the current media.

MCI_SEEK_TO_START
Specifies that the device should seek to the start of the current media.

MCI_TO
Specifies a position is included in the **dwTo** field of the structure identified by *lParam2*, to which the device should seek using the current media. Seek distance is specified in units of mS (MCI_FORMAT_MILLISECONDS).

LPMCI_SEEK_PARMS *lParam2*

Specifies a far pointer to the following **MCI_SEEK_PARMS** data structure:

```
typedef struct {
   DWORD        dwCallback;
   DWORD        dwTo;
} MCI_SEEK_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_SET

This command is used to set TAM device information. The item to set is specified by **dwItem** field of the MCI_TAM_SET_PARMS structure, pointed to by *lParam2*, and set data information is passed in **dwSetData**.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_SET_ITEM

Specifies that the **dwItem** field of the data structure identified by *lParam2* contains a constant specifying which item to set. The following constants are defined:

MCI_TAM_SET_ADVANCED_RING_NOTIFY

The **dwSetData** field is set to the indicate the type of message that is sent to the application when the phone rings. When set to FALSE (the default), a PHONE_EVENT_RING is sent to the application. When the flag is set to TRUE, a PHONE_EVENT_ADVANCED_RING is sent to the application. With advanced format ring events, *lParam* does not contain a pointer to **dwSetData**. Instead, *lParam* contains the device ID and the actual ring count (not a pointer to it). A ring count of zero indicates the end of a ring.

MCI_TAM_SET_AP_DISCRIMINATED **PL**

This function, new to driver version 3.1, gives the TAM application the ability to influence the call-discrimination outcome. For example, a TAM application that has a caller ID database can indicate the incoming call is for FAX, MODEM, VOICE, or don’t answer. The duration of the setting is for the current call only.

The application should preferably issue this call before the current call is answered.

The **dwSetData** field is set to indicate MCI_FAXTAM_PASS_FAX, MCI_FAXTAM_PASS_MODEM, MCI_FAXTAM_PASS_VOICE, or MCI_FAXTAM_DONT_ANSWER. If this call is issued before the discriminator discriminates, the AP’s preference will take precedence over any other discrimination criteria, and the discriminator will not discriminate based on calling tones or information in the discriminator’s database.

MCI_TAM_SET_API_STYLE

Specifies that the **dwSetData** field contains the API style of the TAM device. The possible values are:

- MCI_FAXTAM_STYLE_MMPM
- MCI_FAXTAM_STYLE_WINDOWS

The default style under OS/2 is MMPM. The default style under Windows is WINDOWS. The API style affects return codes for MCI_STATUS, MCI_GETDEVCAPS and MCI_INFO. The style also affects return codes and return information for MM_MCINOTIFY. See Microsoft Windows *Multimedia Programmer's Reference* and IBM's *Programming Reference for Multimedia Presentation Manager Toolkit/2* for details of the MCI interface as specified for Windows and OS/2.

MCI_TAM_SET_AUDIO_MUTE

The **dwSetData** field is set to the desired mute status of the system microphone. When set to TRUE, the microphone (audio input of TAM_AUDIO) is disconnected from the telephone line, and any record operation in progress. When set to FALSE, the device operates normally.

MCI_TAM_SET_AUDIO_VOLUME

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the volume level of the speaker device. The volume level is specified from 0x0 (silence) to 0xFFFF (maximum volume) and is interpreted logarithmically. This means the perceived volume increase is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

MCI_TAM_SET_AVGBYTESPERSEC

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the desired average bytes per second rate of any new RECORD operation.

With Ver 3.1, PCM files are supported. Average bytes per second used with sub-band-coded files is treated as follows:

For most messages with normal speech, the data rate observed will be close to this average. If this value is greater than zero, it overrides with a finer granularity the current MCI_TAM_SET_QUALITY level.

The minimum non-zero value that can be input is 1000. The actual level being used can be found by calling MCI_STATUS. If **dwSetData** is set to zero, the value of MCI_TAM_SET_QUALITY is used. This request does not effect message playback.

With PCM files, average bytes per second must be set to be consistent with bits per second and bits per sample. The formula for calculating bytes per second is:

average bytes per second = bits per second * (bits per sample/8)

The default for average bytes per second is 11025. The default for bits per second is 11025. The default for bits per sample is 8. A list of valid combinations includes:

Bytes/second	Bits/sample	Samples/second
11025	8	11025
22050	16	11025
22050	8	22050
44100	16	22050
44100	8	44100
88200	16	44100

TABLE 7-3

NOTE: TAM_WAVE_FORMAT_CUSTOM (sub band coded) files are equivalent to 15 bits/sample at 11025 samples/second. On average, sub band coded files take less than 4000 bytes/second of disk space.

MCI_TAM_SET_BITSPERSAMPLE

Sets the desired bits per sample (either 8 or 16) used for playing, recording, and saving to the **dwSetData** field of the data structure identified by *lParam2*. This command is used in conjunction with MCI_TAMSET_SAMPLESPERSEC for PCM format wave files only. Using 16 bits per sample sounds noticeably better than 8 bits per sample, but uses twice the disk space.

Note: PCM format files are only supported by version 3.1 or above, of the TAM driver. See MCI_TAM_SET_AVGBYTESPERSEC for more information.

MCI_TAM_SET_CALLER_ID **PL**

Specifies that the **dwSetData** field of the data structure identified by *lParam2* is set to FALSE to disable caller ID processing; otherwise, it is set to TRUE. Caller ID processing uses Mwave DSP resources. Disabling caller ID permits more concurrency. The default is TRUE on systems that have Mwave call discrimination installed.

MCI_TAM_SET_CALL_FILTER **PL**

Specifies that the **dwSetData** field of the data structure identified by *lParam2* is set to TRUE if the device is to receive voice calls; otherwise it is set to FALSE. If another application has this filter enabled, attempting to enable the filter causes an error return.

MCI_TAM_SET_CONNECT

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the new target play or record device, and any inter-device connections. The device flags for the devices to connect are OR'ed together, and the result is placed in **dwSetData**. The TAM device flags are defined as follows:

- MCI_TAM_AUDIO (speaker & microphone) **MSG**
- MCI_TAM_HANDSET **MSG**
- MCI_TAM_PHONELINE **PL**
- MCI_TAM_AUDIO_PHONELINE **PL**

(speakerphone)

- MCI_TAM_HANDSET_PHONELINE **PL**
  (standard phone operation)
- MCI_TAM_SPEAKER_PHONELINE **PL**
  (answering machine w/ call screening)

Note: The device MCI_TAM_PHONELINE is required for all operations involving an outside phone line.
Note: Setting speakerphone operation disables call discrimination based on calling tones. DTMF key detection is also disabled.

MCI_TAM_SET_DIAL_FLASH_TIME **PL**
The **dwSetData** field is set to the desired flash time (in milliseconds) of the telephone flash option in the MCI_DIAL command. The default value is 500 (one half second).

MCI TAM_SET_DIAL_PAUSE_TIME
The **dwSetData** field is set to the desired pause time (in milliseconds) that an embedded ',' character produces in the dial string. The default value is 2000 (2 seconds).

MCI_TAM_SET_DIAL_WAIT_TIME **PL**
The **dwSetData** field is set to the desired time-out limit (in milliseconds) that an embedded 'w' character in the dial string allows, waiting for a second dial tone. The default value is 30000 (30 seconds).

MCI_TAM_SET_EVENT_HANDLER
Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains the handle of the application event handler. The MCI driver posts **MM_MCIEVENT** messages when an event occurs which changes the status of the driver. Setting this value to zero disables event posting. See the event handler section of the document for more details.

MCI_TAM_SET_FORMATTAG
Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the compression/format of the media to be played or recorded. The following formats are allowed:

- WAVE_FORMAT_PCM (Supported by version 3.1 and above of the TAM drivers)
- TAM_WAVE_FORMAT_CUSTOM

MCI_TAM_SET_HANDSET_MUTE **MSG**
The **dwSetData** field is set to the desired mute status of the telephone handset. When set to TRUE, the audio input of the handset is disconnected from the telephone line, and any record operation in progress. When set to FALSE, the device operates normally.

MCI_TAM_SET_HANDSET_VOLUME **MSG**
Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the volume level of the speaker device. The volume level is specified from 0x0 (silence) to 0xFFFF (maximum volume) and is interpreted logarithmically. This means the

perceived volume increase is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

MCI_TAM_SET_HOOK **PL**

The **dwSetData** field is set to the desired hook status of the telephone line. It is set to TRUE to take the handset off-hook, and FALSE to place the handset on-hook. If another application owns the phone line, and the value is set to TRUE, this call will fail. When an application sets **dwSetData** to FALSE, it relinquishes ownership of the line.

MCI_TAM_SET_LOW_LEVEL_WAVE_IO

The **dwSetData** field is set to inform TAM driver that the application intends to use the low level wave audio API to play or record from the phone. This function is available with Ver 3.2 of the TAM drivers. Valid values include:

- MCI_TAM_WAVE_IN_START
- MCI_TAM_WAVE_IN_STOP
- MCI_TAM_WAVE_OUT_START
- MCI_TAM_WAVE_OUT_STOP

In general, a TAM application that uses wave files will not use this interface. However specialized applications, such as voice recognition applications, cannot wait until an entire file has been recorded and saved. Those applications will want to analyze the PCM data as it arrives. To examine buffers as they are received, the application must use the audio driver directly.

Under Windows, the set of calls that the application should make are MCI_GETDEVCAPS for MCI_TAM_GETDEVCAPS_SUPPORTS_PCM_TAG. This returns the device ID of the wave driver that can play to the telephone or handset. The application uses this device ID on the low level audio calls (e.g., waveOutSetVolume). Before opening the wave driver, the application should call MCI_SET to inform the TAM driver that it is about to open the wave driver to start input or output. Likewise, after closing the wave driver, MCI_SET is issued to inform the TAM driver that the low level audio is done.

Under MMPM (OS/2), there are no low level audio API’s. However, if the application wants to inspect PCM buffers it must use the audio driver directly, and do I/O using memory playlists. To accomplish this, the application issues MCI_SET to inform the TAM driver that the wave driver is going to be used (as above). It then issues MCI_OPEN for the wave audio device. After that, it issues ‘connection <alias> query type wave stream alias conndev wait’. The connection command is followed by ‘connector conndev enable type phone line wait’. If using the MSG driver, use ‘phone set’ instead of ‘phone line’. After the application is done using the wave device, MCI_SET is issued to inform the TAM driver that the wave driver is no longer in use.

The application is responsible for setting the speaker volume if it uses the low level audio API.

Most applications will not need this interface. To play or record wave files, it is much simpler to issue ‘load’, and ‘play’ or ‘record’ directly to the TAM driver.

MCI_TAM_SET_MAX_FLASH_TIME **PL**

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant maximum number of milliseconds used for detecting when a handset *flash* has been pressed. A *flash* essentially is an off-hook followed by an on-hook. The period of time between the two events determines if the telco detects one flash hook or two separate events (on-hook and off-hook). Different telcos may use different values. This call allows the application to adjust to the different telcos. The default is zero, meaning that *flash* will not be reported to the application. When *flash* is detected, it is reported in a PHONE_EVENT_HANDSET_KEY, with the key value set to ‘!’. If the max flash time is set less than the min flash time, it is treated as an error.

MCI_TAM_SET_MICROPHONE_GAIN

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the gain of the microphone in dB. Valid values are from 0 to 100 decimal. The default is 50 dB.

MCI_TAM_SET_MIN_FLASH_TIME **PL**

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant indicating the smallest number of milliseconds used for detecting when a handset *flash* has been pressed. A *flash* essentially is an off-hook followed by an on-hook. The period of time between the two events determines if the telco detects one flash hook or two separate events (on-hook and off-hook). Different telcos may use different values. This call allows the application to adjust to the different telcos. The default is zero. When *flash* is detected, it is reported in a PHONE_EVENT_HANDSET_KEY, with the key value set to ‘!’. If the min flash time is set greater than the max flash time, it is treated as an error.

MCI_TAM_SET_PASS_CALL **PL**

The **dwSetData** field is set to a constant indicating the desired type of application that the current phone call will be passed to. The possible values are:

- MCI_FAXTAM_PASS_FAX to pass to a fax application
- MCI_FAXTAM_PASS_MODEM to pass to a modem application

If the specified application is not currently accepting incoming calls, the application retains ownership of the call, and should remember to hang up the phone.

This call can work only if the Mwave call discriminator is active.

MCI_TAM_SET_QUALITY

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the quality level of the phone recording and playback. Quality range is (0-7), where "0" is lowest quality, and "7" is highest quality.

MCI_TAM_SET_QUIET_DURATION **PL**

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the continuous phone line quiet time in seconds before the application will get the first MM_MCIEVENT message specifying CALL_PROGRESS_QUIET. The minimum non-zero value is 4. Zero indicates that the application doesn't want the CALL_PROGRESS_QUIET interrupt returned. The default is 10. After the first CALL_PROGRESS_QUIET, the application will continue receiving this message every second until the call is terminated.

MCI_TAM_SET_QUIET_DURATION is not supported in current TAM drivers. In the interim, the first MM_MCIEVENT message specifying CALL_PROGRESS_QUIET is returned after 4.5 seconds of continuous phone line quiet time.

MCI_TAM_SET_RING_COUNT **PL**

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the ring count at which the device should answer the telephone. The driver will answer on the shortest ring count request of all active applications. Setting this value to 0 requests that the telephone not be answered. The default ring count for TAM is 3.

Prior to driver version 3.1 if the discriminator is loaded, it will answer the telephone on the shortest ring count request of all registered applications. In version 3.1 and above, the discriminator uses the phoneline application's ring count if there is a phoneline application active.

The maximum ring count that can be set is specified by MAX_RING_COUNT.

MCI_TAM_SET_SAMPLESPERSEC

Sets the samples per second used for playing, recording, and saving to the **dwSetData** field of the data structure identified by *lParam2*. This is used for PCM format only.

MCI_TAM_SET_SAMPLESPERSEC is supported in version 3.1 and above of the TAM drivers. See MCI_TAM_SET_AVGBYTESPERSEC for more information.

MCI_TAM_SET_SPEED **MSG**

Specifies that the **dwSetData** field of the data structure identified by *lParam2* contains a constant specifying the speed to play to the current media device. The speed index passed in **dwSetData** is the play speed factor (1/32 to 2) multiplied by 32. Examples include (but are not limited to) the following:

- 16 - 1/2 x normal speed

- 24 - 3/4 x normal speed
- 32 - Normal (recorded) speed
- 40 - 1.25 x normal speed
- 48 - 1.5 x normal speed
- 56 - 1.75 x normal speed
- 63 - 2x normal speed

LPMCI_TAM_SET_PARMS *lParam2*

Specifies a far pointer to the following **MCI_TAM_SET_PARMS** data structure:

```
typedef struct {
   DWORD        dwCallback;
   DWORD        dwSetData;
   DWORD        dwItem;
} MCI_TAM_SET_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_STATUS

This command is used to obtain information about the TAM device. Information is returned in the **dwReturn** field of the MCI_STATUS_PARMS structure, pointed to by *lParam2*.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY

Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_STATUS_ITEM

Specifies that the **dwItem** field of the data structure identified by *lParam2* contains a constant specifying which status item to obtain. The following constants are defined:

MCI_STATUS_CALLER_ID **PL**

The **dwReturn** field is set to the status of the caller ID processing. The following status caller ID are defined:

- MCI_CALLER_ID_ACTIVE (Mwave support is installed and loaded)
- MCI_CALLER_ID_NOT_SUPPORTED (Mwave support not installed)
- MCI_CALLER_ID_DISABLED (by application issuing MCI_SET_CALLER_ID FALSE or because Mwave is processing a fax or modem call)

MCI_STATUS_LENGTH

The **dwReturn** field is set to the length of the current play/record media in milliseconds.

MCI_STATUS_MODE

The **dwReturn** field is set to the current mode of the device. The following modes are defined:

- MCI_MODE_NOT_READY
- MCI_MODE_PAUSE
- MCI_MODE_PLAY
- MCI_MODE_STOP
- MCI_MODE_OPEN
- MCI_MODE_RECORD
- MCI_MODE_SEEK

MCI_STATUS_POSITION

The **dwReturn** field is set to the current position of the play/record media in milliseconds.

MCI_STATUS_READY

The **dwReturn** field is set to TRUE if the device is ready to receive a call; otherwise, it is set to FALSE. If another telephony application has ownership of the telephone line, this status command returns FALSE.

MCI_STATUS_TIME_FORMAT

The **dwReturn** field is set to the time format of the play/record media. This always returns MCI_FORMAT_MILLISECONDS.

MCI_TAM_STATUS_AUDIO_MUTE

The **dwReturn** field of the data structure identified by *lParam2* returns a constant specifying the mute status of the system microphone. When set to TRUE, the microphone (audio input of TAM_AUDIO) is disconnected from the telephone line, and any record operation in progress. When set to FALSE, the device operates normally.

MCI_TAM_STATUS_AUDIO_VOLUME

The **dwReturn** field of the data structure identified by *lParam2* returns a constant specifying the volume level of the speaker device. The volume level is specified from 0x0 (silence) to 0xFFFF (maximum volume) and is interpreted logarithmically. This means the perceived volume increase is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

MCI_TAM_STATUS_AVGBYTESPERSEC

The **dwReturn** field is set to the actual average bytes per second of the current media record or play operation. This value is valid only when a desired rate has been set using MCI_TAM_SET_AVGBYTESPERSEC. See MCI_SET for details.

MCI_TAM_STATUS_BITSPERSAMPLE

The **dwReturn** field is set to the number of bits per sample (8 or 16) used for playing, recording, and saving, when using the PCM wave format.

MCI_TAM_STATUS_BITSPERSAMPLE is supported in version 3.1 and above of TAM drivers.

MCI_TAM_STATUS_CONNECT

The **dwReturn** field is set to the current device connections. Connected device flags are OR'ed together, and the result returned to the application. Connected devices are also the target of any play or record operations, thus for some operations, applications "connect" only a single device. The device flags are defined as follows:

- MCI_TAM_AUDIO
- MCI_TAM_HANDSET
- MCI_TAM_PHONELINE

MCI_TAM_STATUS_CALL_FILTER **PL**

The **dwReturn** field of the data structure identified by *lParam2* is set to TRUE if the device is currently set to receive voice calls; otherwise it is set to FALSE.

MCI_TAM_STATUS_DIAL_FLASH_TIME **PL**

The **dwReturn** field is set to the current flash time (in milliseconds) of the telephone flash option in the MCI_DIAL command.

MCI_TAM_STATUS_DIAL_FLASH_TIME is supported in version 2.2 and above of the TAM drivers.

MCI_TAM_STATUS_DIAL_PAUSE_TIME **PL**

The **dwReturn** field is set to the current pause time (in milliseconds) that an embedded ',' character produces in the dial string.

MCI_TAM_STATUS_DIAL_WAIT_TIME **PL**

The **dwReturn** field is set to the current time-out limit (in milliseconds) that an embedded 'w' character in the dial string allows, waiting for a second dial tone.

MCI_TAM_STATUS_FORMATTAG

The **dwReturn** field is set to the format tag of the current device being recorded or played. The following formats are allowed:

- WAVE_FORMAT_PCM (Supported in version 3.1 and above of the TAM drivers)
- TAM_WAVE_FORMAT_CUSTOM

MCI_TAM_STATUS_HANDSET

The **dwReturn** field is set to the current status of the telephone handset. It is set to TRUE if the handset is off-hook; otherwise, it is set to FALSE.

MCI_TAM_STATUS_HANDSET_MUTE **MSG**

The **dwReturn** field of the data structure identified by *lParam2* returns a constant specifying the mute status of the telephone handset. When set to TRUE, the audio input of the handset is disconnected from the telephone line, and any record operation in progress. When set to FALSE, the device operates normally.

MCI_TAM_STATUS_HANDSET_VOLUME **MSG**

The **dwReturn** field of the data structure identified by *lParam2* returns a constant specifying the volume level of the speaker device. The volume level is specified from 0x0 (silence) to 0xFFFF (maximum volume) and is interpreted logarithmically. This means the perceived volume increase is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

MCI_TAM_STATUS_HOOK **PL**

The **dwReturn** field is set to the current hook status of the telephone line. It is set to TRUE if the phone is off-hook; otherwise, it is set to FALSE.

MCI_TAM_STATUS_LINE **PL**

The **dwReturn** field is set to the current phone line status. The following status modes are defined:

- MCI_PHONE_LINE_ONHOOK
- MCI_PHONE_LINE_DIALTONE
- MCI_PHONE_LINE_BUSY
- MCI_PHONE_LINE_QUIET
- MCI_PHONE_LINE_RINGTONE
- MCI_PHONE_LINE_VOICE
- MCI_PHONE_LINE_FAX
- MCI_PHONE_LINE_MODEM
- MCI_PHONE_LINE_UNKNOWN

MCI_TAM_STATUS_MAX_AUDIO_VOLUME

Some countries limit the maximum audio volume that the application can set. To determine if the application is running in such a country, the application should issue MCI_STATUS for MCI_STATUS_WORLD_TRADE_SUPPORT. If the information returned from that call indicates LIMIT_MAX_VOLUME, the **dwReturn** field of this call is set to the maximum audio volume permitted in the country. This support is added with driver version 3.3.

MCI_TAM_STATUS_MAX_FLASH_TIME **PL**

The **dwReturn** field is set to the current maximum time between on-hook and off-hook that will be reported as a handset *flash* on a PHONE_EVENT_HANDSET_KEY event.

MCI_TAM_STATUS_MAX_GREETING_LEN

Some countries limit the maximum duration a greeting may be. To determine if the application is running in such a country, the application should issue MCI_STATUS for MCI_STATUS_WORLDTRADE_SUPPORT. If the information returned from that call indicates LIMIT_GREETING_LENGTH, the **dwReturn** field of this call is set to the maximum greeting length, in seconds, permitted in the country. It is up to the application to make sure the greeting doesn't exceed this length. This support is added with driver version 3.3.

MCI_TAM_STATUS_MAX_GREETING_LEN_NO_REC

Some countries limit the maximum duration a greeting may be when no message is going to be recorded. To determine if the application is running in such a country, the application should issue MCI_STATUS for MCI_STATUS_WORLDTRADE_SUPPORT. If the information returned from that call indicates LIMIT_GREETING_LENGTH_NO_RECORD, the **dwReturn** field of this call is set to the maximum permitted greeting length, in seconds, when no message is going to be recorded. It is up to the application to make sure the greeting doesn't exceed this length. This support is added with driver version 3.3.

MCI_TAM_STATUS_MAX_MIC_GAIN

The **dwReturn** field is set to the maximum permitted microphone gain, in decibels. This supported is added with driver version 3.3.

MCI_TAM_STATUS_MAX_MSG_RETRIEVE_LEN

Some countries limit the maximum time between user inputs. This would require that the application gets input from the user at least every ‘n’ seconds. To determine if the application is running in such a country, the application should issue MCI_STATUS for MCI_STATUS_WORLDTRADE_SUPPORT. If the information returned from that call indicates LIMIT_MAX_MSG_RETRIEVE_LENGTH, the **dwReturn** field of this call is set to the maximum length, in seconds, that messages can be played without prompting the user for input (e.g., DTMF keys). It is up to the application to make sure greetings don’t exceed this length. This support is added with driver version 3.3.

MCI_TAM_STATUS_MAX_RECORD_LEN

Some countries limit the maximum duration of a message recorded from the phoneline. To determine if the application is running in such a country, the application should issue MCI_STATUS for MCI_STATUS_WORLDTRADE_SUPPORT. If the information returned from that call indicates LIMIT_MESSAGE_RECORD_LENGTH, the **dwReturn** field of this call is set to the maximum length, in seconds, of a message recorded from the phoneline, permitted in the country. It is up to the application to make sure the greeting doesn’t exceed this length. The simplest way to accomplish this is to specify MCI_TO on the MCI_RECORD, and use the value returned from this call as the MCI_TO value. This support is added with driver version 3.3.

MCI_TAM_STATUS_MICROPHONE_GAIN

The **dwReturn** field is set to the current microphone gain, in decibels.

MCI_TAM_STATUS_MIN_FLASH_TIME **PL**

The **dwReturn** field is set to the current minimum time between on-hook and off-hook that will be reported as a handset *flash* on a PHONE_EVENT_HANDSET_KEY event.

MCI_TAM_STATUS_QUALITY

The **dwReturn** field is set to the current telephone device play/record quality level. Expected range of quality is from 0 (lowest quality) to 7 (highest quality).

MCI_TAM_STATUS_QUIET_DURATION **PL**

The **dwReturn** field is set to the current value for the continuous phone line quiet time in seconds before an application will get the first **MM_MCIEVENT** message specifying CALL_PROGRESS_QUIET. Zero indicates the application will not get the CALL_PROGRESS_QUIET interrupt returned. After receiving the first CALL_PROGRESS_QUIET, the application continues receiving this message every second until the call terminates.

MCI_TAM_STATUS_QUIET_DURATION is not supported in current driver.

MCI_TAM_STATUS_RING_COUNT **PL**

The **dwReturn** field is set to a constant specifying the ring count at which the device answers the telephone. The driver answers on the shortest ring count request of all active applications, so this value might not match the value specified in MCI_SET.

MCI_TAM_STATUS_SAMPLESPERSEC

The **dwReturn** field is set to the number of samples per second used for playing, recording, and saving, when using the PCM wave format. MCI_TAM_STATUS_SAMPLESPERSEC is supported in version 3.1 and above of the TAM drivers.

MCI_TAM_STATUS_SPEED **MSG**

The **dwReturn** field is set to the device speed factor of the current device. See MCI_TAM_SET_SPEED for details.

MCI_TAM_STATUS_WORLDTRADE_SUPPORT

The **dwReturn** field is set to a binary encoded set of values indicating restrictions that are in effect for the current country. Some of the bit settings require the application to make a subsequent MCI_STATUS call to determine a maximum value. This support is added with driver version 3.3. The defined bits include:

- GAIN_CHANGE_NOT_ALLOWED is set TRUE if the application is not permitted to change the microphone gain.
- GAIN_CHANGE_NOT_ALLOWED_OFFHOOK is set TRUE if the application is not permitted to change the microphone gain when the phone is off hook.
- LIMIT_MAX_VOLUME is set TRUE if the maximum speaker volume is limited. See MCI_STATUS for MCI_TAM_STATUS_MAX_AUDIO_VOLUME for related information.
- LIMIT_GREETING_LENGTH is set TRUE if the greeting length is limited. See MCI_STATUS for MCI_TAM_STATUS_MAX_GREETING_LEN for related information.
- LIMIT_GREETING_LENGTH_NO_RECORD is set TRUE if the greeting length is limited when no message will be recorded. See MCI_STATUS for MCI_TAM_STATUS_MAX_GREETING_LEN_NO_REC for related information.
- DISALLOW_GREETING_WITH_NO_RECORD is set TRUE if informational greetings are not permitted.
- LIMIT_MESSAGE_RECORD_LENGTH is set TRUE if the length of messages recorded from the phone line is limited. See MCI_STATUS for MCI_TAM_STATUS_MAX_RECORD_LEN for related information.
- REMOTE_GREETING_RECORD_REVIEW is set TRUE if the country requires the application to play back a remotely recorded greeting before the new greeting goes into effect.
- LIMIT_MAX_MSG_RETRIEVE_LENGTH is set TRUE if the country limits the maximum time between user input. This would require that the application get input from the user every ‘n’

seconds. See MCI_STATUS for MCI_TAM_STATUS_MAX_MSG_RETRIEVE_LEN for related information.

- NEVER_ANSWER_SILENT is set TRUE if the phone can never be answered with silence.
- TAM_NOT_ALLOWED_IN_COUNTRY is set TRUE if the telephone answering machine functions are not permitted in the country. If this is the case the **PL** application will get a bad return code on MCI_OPEN. However the **MSG** application can query this information.
- SPK_PHONE_NOT_ALLOWED_IN_COUNTRY is set TRUE if connecting to speaker phone is not permitted.
- AUTODISCRIM_TAM_NOT_ALLOWED is set TRUE if the automatic call discrimination of voice calls is not permitted.
- AUTODISCRIM_FAX_NOT_ALLOWED is set TRUE if the automatic call discrimination of FAX calls is not permitted.
- AUTODISCRIM_MODEM_NOT_ALLOWED is set TRUE if the automatic call discrimination of MODEM calls is not permitted.

This support is added with driver version 3.4.

- PULSE_DIAL_NOT_ALLOWED is set TRUE if pulse dialing is not supported.
- DTMF_DIAL_NOT_ALLOWED is set TRUE if DTMF dialing is not supported.
- BUSYTONE_DETECT_NOT_VALID is set TRUE if busy tone detection is not available in the country.
- BUSYTONE_DETECT_REQUIRED is set TRUE if busy tone detection is required in country.
- DIALTONE_DETECT_NOT_VALID is set TRUE if dial tone detection is not available in the country.
- DIALTONE_DETECT_REQUIRED is set TRUE if dial tone detection is required in country.
- OFFHOOK_NOT_ALLOWED_HANDSET_UP if the application is not permitted to have the phone electronically off hook (SET HOOK TRUE) when the handset is up.

MCI_TAM_STATUS_COUNTRY_CODE

The **dwReturn** field is set to the current country code. This can be used by applications that must change the looks of the user interface for different countries like a French keypad in France. This support is added with driver version 3.4. The following table shows the codes assigned to each country:

COUNTRY	CODE
USA/Canada	1
Belgium	2
Hong Kong	3
Singapore	4
New Zealand	5
Japan	6
Portugal	7
Ireland	8
Generic	9

COUNTRY	CODE
Australia	14
Austria	15
Mexico	16
South Africa	17
Chile	18
Switzerland	19
Germany	20
Brazil	21
Russia	22

COUNTRY	CODE
Norway	27
Denmark	28
France	29
Netherlands	30
U. K.	31
Sweden	32
Italy	33
Finland	34
Thailand	35

Spain	10	Yugoslavia	23	Korea	36
Greece	11	Hungary	24	Malaysia	37
Israel	12	Czechrepublic	25	PRC	38
Taiwan	13	Luxembourg	26	Slovakia	39

TABLE 7-4: Country Codes

MCI_TAM_STATUS_AUTO_ANSWER_MIN_RINGS

The **dwReturn** field contains the minimum number of rings that can be set in MCI_TAM_SET_RING_COUNT. If the value is 'FFFF'x then there is no min in that country. This support is added with driver version 3.4.

MCI_TAM_STATUS_AUTO_ANSWER_MAX_RINGS

The **dwReturn** field contains the maximum number of rings can be set in MCI_TAM_SET_RING_COUNT. If the value is 'FFFF'x then there is no limit in that country. This support is added with driver version 3.4.

MCI_TAM_STATUS_MAX_CALL_RETRIES

The **dwReturn** field contains the maximum number of unsuccessful retries allowed. If the value is 'FFFF'x there is no max in that country. This support is added with driver version 3.4.

MCI_TAM_STATUS_MIN_CALL_RETRY_TIME

The **dwReturn** field contains the minimum time allowed between retries. If the value is 'FFFF'x then there is no min in that country. This support is added with driver version 3.4.

LPMCI_STATUS_PARMS *lParam2*

Specifies a far pointer to the following **MCI_STATUS_PARMS** data structure:

```
typedef struct {
   DWORD        dwCallback;
   DWORD        dwReturn;
   DWORD        dwItem;
} MCI_STATUS_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_STOP

This command message stops an **MCI_PLAY** or **MCI_RECORD** command in operation.

Parameters DWORD *lParam1*

The following flags apply to the TAM device:

MCI_NOTIFY
Specifies that MCI should post the **MM_MCINOTIFY** message when this command completes. The window to receive this message is specified in the **dwCallback** field of the data structure identified by *lParam2*.

MCI_WAIT
Specifies that the operation should finish before MCI returns control to the application.

MCI_STOP_REMOVE_DTMF
Specifies that if the MCI_STOP is stopping a record operation, and the record has recorded a DTMF key, all information recorded after the first DTMF key was pressed will be removed from the recorded message.

LPMCI_GENERIC_PARMS *lParam2*

Specifies a far pointer to the following **MCI_GENERIC_PARMS** data structure:

```
typedef struct {
   DWORD         dwCallback;
} MCI_GENERIC_PARMS;
```

Return Value Returns zero if successful. Otherwise, it returns an MCI error code.

Chapter 8 - Error Codes

This chapter contains descriptions of device specific error codes supported by the Mwave FAX and TAM drivers. Other error codes that may be returned by the drivers are defined by the MMPM and Windows MCI support.

The Mwave FAX and TAM driver errors are returned by the **mciSendCommand** and **mciSendString** function when a failure occurs. The error code constants are defined in the **MCIFTDD.H** include file. The application can issue **mciGetErrorString** to retrieve a textual description of the given error code.

The error codes are comprised of seven functional groups; each group is represented by a range of error codes and an error code prefix. The following table lists the error code ranges, the associated error code prefix, and the issuing drivers.

Error Codes (Windows)	**Error Codes (OS/2)**	**Prefix**	**Issuing Driver**
1-511	1-5255	MCIERR_	Errors defined by Windows or MMPM
512-545	5256-5289	MCIERR_FT_	Errors common to FAX and TAM
613-652	5357-5396	MCIERR_FAX_	Errors specific to FAX
813-897	5557-5641	MCIERR_TAM_	Errors specific to TAM
913-932	5657-5676	MCIERR_DIS_	Errors specific to the Call Discriminator
1025-1213	5769-6334	MCIERR_FAX_TIF_	Errors specific to the FAX driver's BMP/TIFF conversions.
1313	6569	MCIERR_MEIO_	Errors specific to MEIO

Table 8-1

When using wave files in the TAM drivers, errors less than MCIERR_FT_DSP_NO_RESOURCES (which is defined as MCIERR_CUSTOM_DRIVER_BASE) come directly from the wave device driver. These are the codes below 512 (for Windows) and 5256 (for OS/2).

Error codes for the FAX and TAM specific errors are listed below in numeric order. Where appropriate, possible causes and solutions to the error are provided.

FAX/TAM Driver Error Codes

The error codes described in this section are common to both the Mwave FAX and TAM drivers.

Win	OS/2	
512	**5256**	**MCIERR_FT_DSP_NO_RESOURCES** Insufficient resources in the DSP card.
513	**5257**	**MCIERR_FT_DSP_FILE_NOT_FOUND** DSP file or module in the DSP file not found.
514	**5258**	**MCIERR_FT_DSP_LABEL_NOT_FOUND** Label of resource in the DSP card not found.
515	**5259**	**MCIERR_FT_DSP_INVALID_HANDLE** Invalid handle for DSP resource.
516	**5260**	**MCIERR_FT_DSP_CALL_FAILED** Call to the Mwave Manager failed.
517	**5261**	**MCIERR_FT_UNRECOGNIZED_COMMAND** Invalid or unknown command requested.
518	**5262**	**MCIERR_FT_CMD_COMPLETE_NOT_RTN** Command complete status for FAX or TAM command not received.
519	**5263**	**MCIERR_FT_UNRECOGNIZED_MODE** Invalid or unknown FAX or TAM mode.
520	**5264**	**MCIERR_FT_POSTMESSAGE** Error in executing function,**PostMessage**.
521	**5265**	**MCIERR_FT_MAKEPROCINSTANCE** Error in executing function,**MakeProcInstance**.
522	**5266**	**MCIERR_FT_SETWINDOWSHOOKEX** Error in executing function,**tWindowsHookEx**.
523	**5267**	**MCIERR_FT_GLOBALALLOC** Error in executing function,**GlobalAlloc**.
524	**5268**	**MCIERR_FT_GLOBALLOCK** Error in executing function,**GlobalLock**.
525	**5269**	**MCIERR_FT_GLOBALPAGELOCK** Error in executing function,**GlobalPageLock**.
526	**5270**	**MCIERR_FT_GLOBALUNLOCK** Error in executing function,**GlobalUnlock**.

527 5271 MCIERR_FT_GLOBALPAGEUNLOCK
Error in executing function,**GlobalPageUnlock**.

528 5272 MCIERR_FT_GLOBALFREE
Error in executing function,**GlobalFree**.

529 5273 MCIERR_FT_DSP_HARDWARE_IN_USE
Requested hardware already allocated.

530 5274 MCIERR_FT_DSP_HARDWARE_UNAVAILABLE
Requested hardware unavailable for allocation.

531 5275 MCIERR_FT_MEIO_MIC_S1_TO_CDADC_S1
Requested connection could not complete. Check to assure audio is off.

532 5276 MCIERR_FT_MEIO_MIC_L1_TO_VOICEADC_1
Requested connection could not complete.

533 5277 MCIERR_FT_MEIO_HANDIN_1_TO_VOICEADC_1
Requested connection could not complete.

534 5278 MCIERR_FT_MEIO_HANDIN_1_TO_TELEOUT_1
Requested connection could not complete.

535 5279 MCIERR_FT_CDDAC_S1_TO_LINEOUT_1
Requested connection could not complete.

536 5280 MCIERR_FT_CDDAC_S1_TO_INTSPKROUT_L1
Requested connection could not complete.

537 5281 MCIERR_FT_MEIO_TELEDAC_1_TO_TELEOUT_1
Requested connection could not complete.

538 5282 MCIERR_FT_MEIO_VOICEDAC_1_TO_HANDOUT_1
Requested connection could not complete.

539 5283 MCIERR_FT_INSUFFICIENT_MIPS
Insufficient DSP MIPs available to satisfy the requested operation.

540 5284 MCIERR_FT_INVALID_ABS_SEG_START
Invalid Mwave absolute segment start address (0). This indicates that a DSP task was not loaded when I/O was requested.

541 5285 MCIERR_FT_INI_LABEL_NOT_FOUND
Unable to find label in an ini file.

542 5286 MCIERR_FT_CALLER_ID_NOT_VALID
Caller ID is no longer available.

543 5050 MCIERR_FT_INVALID BUFFER
The buffer length specified on MCI_INFO is not large enough to hold all of the information. See MCI_INFO for further explanation.

544 5288 MCIERR_FT_POWERED_DOWN
System is in "power saving" mode.

545 5289 MCIERR_FT_CANT_CALL_NOW
The system is not permitted to call this phone number at the current time. Some countries restrict automated calling machines from calling the same number too often.

546 5290 MCIERR_FT_UNKNOWN_CALLER_ID_FORMAT
The received caller ID is not in a format that the driver knows how to parse.

547 5291 MCIERR_FT_PREEMPTED_BY_HIGHER_PRTY
The driver is temporarily unavailable because higher priority work is using the DSP.

548 5292 MCIERR_FT_FUNCTION_NOT_ALLOWED_IN_COUNTRY
The requested function is not permitted based on the laws of the particular country.

549 5293 MCIERR_FT_LINE_NOT_IN_USE
The phone is not in use by any application.

550 5294 MCIERR_FT_WRONG_PHONE_COUPLER
FAX or TAM Error, the selected country does not match the external telephone coupler.

FAX Driver Error Codes

The error codes in this section are specific to the Mwave FAX driver.

Win OS/2

613 5357 MCIERR_FAX_GLOBALALLOC
Error in executing function,**GlobalAlloc**.

614 5358 MCIERR_FAX_GLOBALLOCK
Error in executing function,**GlobalLock**.

615 5359 MCIERR_FAX_GLOBALUNLOCK
Error in executing function,**GlobalUnlock**.

616 5360 MCIERR_FAX_GLOBALFREE
Error in executing function,**GlobalFree**.

617 5361 MCIERR_FAX_LCLOSE
Error in executing function, **_lclose**.

618	**5362**	**MCIERR_FAX_LLSEEK** Error in executing function,**_llseek**.
619	**5363**	**MCIERR_FAX_LREAD** Error in executing function,**_lread**.
620	**5364**	**MCIERR_FAX_LSTRCPY** Error in executing function,**lstrcpy**.
621	**5365**	**MCIERR_FAX_OPENFILE** Error in executing function,**OpenFile**.
622	**5366**	**MCIERR_FAX_POSTMESSAGE** Error in executing function,**PostMessage**.
623	**5367**	**MCIERR_FAX_NO_ELEMENT_ALLOWED** MCI_OPEN was called with a device element specified. No device element is allowed for simple devices.
624	**5368**	**MCIERR_FAX_FLAGS_NOT_COMPATIBLE** Flags cannot be set together.
625	**5369**	**MCIERR_FAX_UNRECOGNIZED_KEYWORD** Invalid or unknown keyword used in request.
626	**5370**	**MCIERR_FAX_CANNOT_SET_HOOK** Phone hook cannot be set.
627	**5371**	**MCIERR_FAX_UNRECOGNIZED_COMMAND** Invalid or unknown command requested.
628	**5372**	**MCIERR_FAX_UNRECOGNIZED_FLAG** Invalid or unknown flag used in request.
629	**5373**	**MCIERR_FAX_INVALID_DIAL_DIGIT** Invalid dial digit found in dial string.
630	**5374**	**MCIERR_FAX_NULL_DIAL_STRING** Empty dial string.
631	**5375**	**MCIERR_FAX_FILENAME_REQUIRED** Filename is required for the execution of command.
632	**5376**	**MCIERR_FAX_UNSUPPORTED_FUNCTION** Requested function is not supported.
633	**5377**	**MCIERR_FAX_MISSING_FLAG** Required flag not set.
634	**5378**	**MCIERR_FAX_GLOBALREALLOC** Error in executing function,**GlobalReAlloc**.
635	**5379**	**MCIERR_FAX_LSTRCAT** Error in executing function,**lstrcat**.

636 5380 MCIERR_FAX_WRONG_COMMAND
Command complete status received is not for outstanding command.

637 5381 MCIERR_FAX_COMMAND_REJECT
Undefined command was received.

638 5382 MCIERR_FAX_NO_FREE_STATUS_BLOCK
Insufficient buffers for status blocks.

639 5383 MCIERR_FAX_UNRECOGNIZED_STATUS
Invalid or unknown FAX status received.

640 5384 MCIERR_FAX_GLOBALPAGEUNLOCK
Error in executing **GlobalPageUnlock**

641 5385 Unused

642 5386 MCIERR_FAX_LWRITE
Error in executing function, **_lwrite**.

643 5387 MCIERR_FAX_UNRECOGNIZED_STREAM_ID
Invalid or unknown stream identifier.

644 5388 MCIERR_FAX_INVALID_CONFIG
Configuration requested is invalid.

645 5389 MCIERR_FAX_FILTER_NOT_SET
MCI_RECEIVE issued when SET_CALL_FILTER is FALSE.

646 5390 MCIERR_FAX_MULTIPLE_OPEN
MCI_OPEN issued when another application had device open.

647 5391 MCIERR_FAX_TASK_NOT_FOUND
Driver tried to find an address in a non-existent task.

648 5392 MCIERR_FAX_INVALID_HANDLE
dwCallback specified an invalid HWND.

649 5393 MCIERR_FAX_CONFLICT_FLAGS
Specified flags conflict with one another.

650 5394 MCIERR_FAX_INVALID_STATE
Device is in incorrect state for option specified.

651 5395 MCIERR_FAX_INVALID_PARM
Invalid parameter was used in the request.

652 5396 MCIERR_FAX_HEADINGNOTSET
MCI_FAX_SET_HEADING was not performed before request to use heading (MCI_SEND_HEADING).

TAM Driver Error Codes

The error codes in this section are specific to the Mwave TAM driver.

Win	OS/2	
813	**5557**	**MCIERR_TAM_GLOBALALLOC** Error in executing function,**GlobalAlloc**.
814	**5558**	**MCIERR_TAM_GLOBALLOCK** Error in executing function,**GlobalLock**.
815	**5559**	**MCIERR_TAM_GLOBALUNLOCK** Error in executing function,**GlobalUnlock**.
816	**5560**	**MCIERR_TAM_GLOBALFREE** Error in executing function,**GlobalFree**.
817	**5561**	**MCIERR_TAM_LCLOSE** Error in executing function,**_lclose**.
818	**5562**	**MCIERR_TAM_LLSEEK** Error in executing function,**_llseek**.
819	**5563**	**MCIERR_TAM_LREAD** Error in executing function,**_lread**.
820	**5564**	**MCIERR_TAM_LSTRCPY** Error in executing function,**lstrcpy**.
821	**5565**	**MCIERR_TAM_OPENFILE** Error in executing function,**OpenFile**.
822	**5566**	**MCIERR_TAM_POSTMESSAGE** Error in executing function,**PostMessage**.
823	**5567**	**MCIERR_TAM_NO_ELEMENT_ALLOWED** MCI_OPEN was called with a device element specified. No device element is allowed for simple devices. This return code is not returned after driver Version 2.2.
824	**5568**	**MCIERR_TAM_FLAGS_NOT_COMPATIBLE** Flags cannot be set together.
825	**5569**	**MCIERR_TAM_UNRECOGNIZED_KEYWORD*:*** Invalid or unknown keyword used in request.
826	**5570**	**MCIERR_TAM_CANNOT_SET_HOOK** Phone hook cannot be set.
827	**5571**	**MCIERR_TAM_UNRECOGNIZED_COMMAND** Invalid or unknown command requested.

828 5572 MCIERR_TAM_UNRECOGNIZED_FLAG
Invalid or unknown flag used in request.

829 5573 MCIERR_TAM_INVALID_DIAL_DIGIT
Invalid dial digit found in dial string.

830 5574 MCIERR_TAM_NULL_DIAL_STRING
Empty dial string.

831 5575 MCIERR_TAM_FILENAME_REQUIRED
Filename is required for the execution of command.

832 5576 MCIERR_TAM_UNSUPPORTED_FUNCTION
Requested function is not supported.

833 5577 MCIERR_TAM_MISSING_FLAG
Required flag not set.

834 5578 MCIERR_TAM_GLOBALREALLOC
Error in executing function,**GlobalReAlloc**.

835 5579 MCIERR_TAM_LSTRCAT
Error in executing function,**lstrcat**.

836 5580 MCIERR_TAM_WRONG_COMMAND
Command complete status received is not for outstanding command.

837 5581 MCIERR_TAM_COMMAND_REJECT
Undefined command was receivedor the command is disallowed in the particular country. This is also received if the application indicates dial and wait for dial tone, but no dial tone is heard.

838 5582 MCIERR_TAM_NO_FREE_STATUS_BLOCK
Insufficient buffers for status blocks.

839 5583 MCIERR_TAM_UNRECOGNIZED_STATUS
Invalid or unknown TAM status received.

840 5584 MCIERR_TAM_GLOBALPAGEUNLOCK
Error in executing GlobalPageUnlock.

841 5585 MCIERR_TAM_GLOBALPAGELOCK
Error in executing function,**GlobalPageLock**.

842 5586 MCIERR_TAM_LWRITE
Error in executing function,**_lwrite**.

843 5587 MCIERR_TAM_UNRECOGNIZED_STREAM_ID
Invalid or unknown stream identifier.

844 5588 MCIERR_TAM_UNHOOKWINDOWSHOOKEX
Error in executing **UnhookWindowsHookEx**.

845 5589 MCIERR_TAM_INVALID_MEDIA_HANDLE
Invalid handle for media.

846 5590 MCIERR_TAM_INVALID_MEDIA_LENGTH
Media length less than 1 quality word.

847 5591 MCIERR_TAM_INVALID_MEDIA_HEADER
Invalid quality word in media header.

848 5592 MCIERR_TAM_INVALID_MEDIA_FRAME
Invalid length for SSTM frame.

849 5593 MCIERR_TAM_INVALID_MEDIA_FORMAT
Invalid format for SSTM frame.

850 5594 MCIERR_TAM_INVALID_MEDIA_DATA
Invalid data for SSTM frame.

851 5595 MCIERR_TAM_DWFROM_OUTOFRANGE
The dwFrom parameter greater than**dwTo** position or greater than the length of the media.

852 5596 MCIERR_TAM_DWTO_OUTOFRANGE
The **dwTo** parameter greater than length of the media.

853 5597 MCIERR_TAM_ACCESS_ZERO_LENGTH
The interval between **dwFrom** and **dwTo** is 0 or too small (i.e. it is within the same frame) to be executed.

854 5598 MCIERR_TAM_EMPTY_MEDIA
Media (file) is empty.

855 5599 MCIERR_TAM_NO_STREAM_EXIST
Attempt to operate a non-existing stream.

856 5600 MCIERR_TAM_ANOTHER_STREAM_RUNNING
Attempt to operate a stream while another stream is running.

857 5601 MCIERR_TAM_MODULE_NOT_LOADED
Requested module not loaded.

858 5602 MCIERR_TAM_DEVICE_NOT_USED
An unused device is selected in keyword of MCI command.

859 5603 MCIERR_TAM_DATA_OUTOFRANGE
The value of keyword of MCI command is out of range.

860 5604 MCIERR_TAM_WRONG_CONNECT
Invalid connect to devices.

861 5605 MCIERR_TAM_INVALID_FILE_HANDLE
Invalid file handle.

862 5606 MCIERR_TAM_INVALID_POSITION
Request to move to an invalid position in the media.

863 5607 MCIERR_TAM_INVALID_CONFIG
Configuration requested is invalid.

864 5608 MCIERR_TAM_INVALID_STATUS_BLOCK
Empty status block or its handle is null.

865 5609 MCIERR_TAM_UNSUITABLE_CONDITION
Operating conditions for a command are wrong or not ready.

866 5610 MCIERR_TAM_UNSUITABLE_OBJECT
Device exists but should not be operated for current command.

867 5611 MCIERR_TAM_DATA_INCORRECT
Parameter is within range, but is incorrect.

868 5612 MCIERR_TAM_MULTIPLE_OPEN
Open attempted for already open device.

869 5613 MCIERR_TAM_INVALID_MODE
Invalid mode.

870 5614 MCIERR_TAM_TPL_MEIO_ALREADY_OPENED
TAM TPL error. MEIO already opened.

871 5615 MCIERR_TAM_TPS_MEIO_ALREADY_OPENED
TAM TPS error. MEIO already opened.

872 5616 MCIERR_TAM_SWITCH_TO_TPS_AUDIO
TAM error switching to TPS audio.

873 5617 MCIERR_TAM_SWITCH_TO_TPS_AUDIO_REC
TAM error switching to TPS audio record.

874 5618 MCIERR_TAM_SWITCH_TO_TPS_HANDSET
TAM error switching to TPS handset

875 5619 MCIERR_TAM_SWITCH_TO_TPL_PHONELINE
TAM error switching to TPL phoneline.

876 5620 MCIERR_TAM_SWITCH_TO_TPL_SPEAKERPHONE
TAM error switching to TPL speakerphone

877 5621 MCIERR_TAM_SWITCH_TO_TPL_NORMALPHONE
TAM error switching to TPL normal phone.

878 5622 MCIERR_TAM_SWITCH_TO_TPL_CALL_SCREEN
TAM error switching to TPL call screening

879 5623 MCIERR_TAM_NO_DIAL_TONE
No dial tone received.

880 **5624** **MCIERR_TAM_UNSUPPORTED_FLAG**
Flag combination is not allowed.

881 **5625** **MCIERR_TAM_INVALID_HANDLE**
The **dwCallback** was specified with an invalid **HWND**.

882 **5626** **MCIERR_FT_malloc**
Error allocating storage.

883 **5627** **MCIERR_FT_dspLockMem**
Error locking memory down for DMA transfer.

884 **5628** **MCIERR_FT_dspUnlockMem**
Error unlocking memory after a DMA transfer.

885 **5629** **MCIERR_TAM_OPEN_WAVE_DRIVER**
The TAM driver was unable to open the supporting wave driver.

886 **5630** **MCIERR_TAM_LOAD_WAVE_DRIVER**
The TAM driver was unable to load a wave file.

887 **5631** **MCIERR_TAM_PLAY_WAVE_DRIVER**
The TAM driver was unable to play a wave file.

888 **5632** **MCIERR_TAM_RECORD_WAVE_DRIVER**
The TAM driver was unable to record a wave file.

889 **5633** **MCIERR_TAM_CLOSE_WAVE_DRIVER**
The TAM driver was unable to close the supporting wave driver.

890 **5634** **MCIERR_TAM_PAUSE_WAVE_DRIVER**
The TAM driver was unable to pause a wave file.

891 **5635** **MCIERR_TAM_SAVE_WAVE_DRIVER**
The TAM driver was unable to save a wave file.

892 **5636** **MCIERR_TAM_RESUME_WAVE_DRIVER**
The TAM driver was unable to resume playing or recording a wave file.

893 **5637** **MCIERR_TAM_STOP_WAVE_DRIVER**
The TAM driver was unable to stop a wave file.

894 **5638** **MCIERR_TAM_SEEK_WAVE_DRIVER**
The TAM driver was unable to seek in a wave file.

895 **5639** **MCIERR_TAM_STATUS_WAVE_DRIVER**
The TAM driver was unable to determine the status of a wave file.

896 **5640** **MCIERR_TAM_SET_WAVE_DRIVER**
The TAM driver was unable to set an item for a wave file.

897 **5641** **MCIERR_TAM_CONVERT_WAVE_DRIVER**
The TAM driver was unable to convert a wave file.

908 5652 MCIERR_TAM_GAIN_CHANGE_NOT_ALLOWED
The application tried to change the microphone sensitivity in a country where that is not permitted at this time.

Discriminator Error Codes

The error codes in this section are specific to the Call Discriminator.

Win	OS/2	
913	**5657**	**MCIERR_DIS_TYPE_ALREADY_REGISTERED** Discriminator type already registered.
914	**5658**	**MCIERR_DIS_TYPE_NOT_REGISTERED** Discriminator type not registered.
915	**5659**	**MCIERR_DIS_NOT_LOADED** Discriminator not loaded.
916	**5660**	**MCIERR_DIS_INVALID_TYPE** Discriminator invalid type.
917	**5661**	**MCIERR_DIS_LOAD_FAIL** Discriminator load failed.
918	**5662**	**MCIERR_DIS_APPLICATION_NOT_REGISTERED** Application of the specifiedtype is not registered for autoanswer.
919	**5663**	**DISCR_ALREADY_REGISTERED** An application of the specified type is already registered for autoanswer.
920	**5664**	**DISCR_TYPE_INVALID** The discriminator parameter usType is not valid.
921	**5665**	**DISCR_FAX_HAS_LINE** The line is already 'owned' by the fax application.
922	**5666**	**DISCR_TAM_HAS_LINE** The line is already 'owned' by the TAM application.
923	**5667**	**DISCR_MODEM_HAS_LINE** The line is already 'owned' by the modem application.
924	**5668**	**DISCR_REQUESTOR_NOT_REGISTERED** Not currently used.
925	**5669**	**DISCR_DISCRIM_ID_INVALID** The LineID is greater than the maximum number ofphone lines supported.
926	**5670**	**DISCR_PASS_PARM_INVALID** Parameter to set pass call is invalid.

927 5671 DISCR_OPERATING_SYSTEM_ERROR
Some operating system function failed.

928 5672 DISCR_HWND_INVALID
The specified window handle is not valid.

929 5673 DISCR_INVALID_STATE
Discriminator invalid state for requested action.

930 5674 MCIERR_DIS_LOADLIBRARY_ERROR
Unable to load the discriminator library (DLL).

931 5675 MCIERR_DIS_GETPROCADDR_ERROR
Unable to get the discriminator procedure address.

932 5676 DISCR_AT_FAX_HAS_LINE
The line is already ‘owned’ by a fax modem that uses the AT command set.

TIFF Error Codes

The error codes in this section are specific to I/O problems with TIFF files. The Mwave FAX driver uses the TIFF file format to store fax documents.

> **Note**: The following descriptions use the abbreviation **MH** to refer to Modified-Huffman.

Win **OS/2**

1025 **5769** **MCIERR_FAX_TIF_MHTIF_CANNOTCREATETIFF**
Cannot create a TIFF file for storing the MH images, because an invalid TIFF file name was supplied. Use a TIFF file name that conforms to DOS convention.

1026 **5770** **MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE**
Cannot allocate sufficient global memory to store byte aligned MH data. Increase available RAM.

1027 **5771** **MCIERR_FAX_TIF_MHTIF_GLOCKHWRITE**
Cannot lock memory for storing byte aligned MH data because either the memory block or handle is invalid, or the memory block is 0 byte. Check the file handle.

1028 **5772** **MCIERR_FAX_TIF_MHTIF_GALLOCHTGT**
Cannot allocate global memory to store MH filenames. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE

1029 **5773** **MCIERR_FAX_TIF_MHTIF_GLOCKHTGT**
Cannot lock global memory for storing MH filenames. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1030 **5774** **MCIERR_FAX_TIF_MHTIF_GALLOCHMEM**
Cannot allocate global memory to read MH data. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1031 **5775** **MCIERR_FAX_TIF_MHTIF_GLOCKHMEM**
Cannot lock global memory designated for reading MH data. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1032 **5776** **MCIERR_FAX_TIF_MHTIF_WRITETIF**
Unused.

1033 **5777** **MCIERR_FAX_TIF_MHTIF_IMAGEMH**
Cannot open a MH file because the file is either invalid or does not exist in the current directory. Check the MH filename for validity.

1063 **5778** **MCIERR_FAX_TIF_TIFMH_GALLOCHR**
Cannot allocate global memory for reading image data. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1064 5779 MCIERR_FAX_TIF_TIFMH_GLOCKHR
Cannot lock memory for reading image data. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1065 5780 MCIERR_FAX_TIF_TIFMH_GALLOCHW
Cannot allocate global memory for swapping every two bytes of Modified Huffman data. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1066 5781 MCIERR_FAX_TIF_TIFMH_GLOCKHW
Cannot lock memory for swapping every two bytes of MH data. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1067 5782 MCIERR_FAX_TIF_TIFMH_GALLOCHMHLIST
Cannot allocate global memory for storing MH filenames to be returned to calling function (FAX driver). See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1068 5783 MCIERR_FAX_TIF_TIFMH_GLOCKHMHLIST
Cannot lock memory for storing MH filenames. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1069 5784 MCIERR_FAX_TIF_TIFMH_GALLOCTIFTOMHBUF
Cannot allocate global memory for storing MH filenames. See **MCIERR_FAX_MHTIF_GALLOCHWRITE**.

1070 5785 MCIERR_FAX_TIF_TIFMH_GLOCKTIFTOMHBUF
Cannot lock memory for storing MH filenames. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1071 5786 MCIERR_FAX_TIF_TIFMH_GALLOCHPART
Cannot allocate memory for storing MH filenames for sorting. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1072 5787 MCIERR_FAX_TIF_TIFMH_GLOCKHPART
Cannot lock memory for storing MH filenames. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1073 5788 MCIERR_FAX_TIF_TIFMH_GALLOCHTEMP
Cannot allocate memory for an intermediate buffer during sorting of MH filenames. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1074 5789 MCIERR_FAX_TIF_TIFMH_GLOCKHTEMP
Cannot lock memory for sorting of MH filenames. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1075 5790 MCIERR_FAX_TIF_TIFMH_NOTINTELFORMAT
The TIFF file format is either invalid or not Intel. Use Intel format TIFF file.

1076 5791 MCIERR_FAX_TIF_TIFMH_CANNOTCREATEMH
Cannot create the MH file because the specified filename is invalid. Use proper MH filename.

1077 5792 MCIERR_FAX_TIF_TIFMH_CANNOTOPENTIFF
Cannot open the TIFF file because the specified filename is either invalid or non-existent. Ensure TIFF file is present in the current directory and specify its name correctly.

1103 5793 MCIERR_FAX_TIF_TIFBMP_CANNOTOPENTIFF
Cannot open the TIFF file because the TIFF file does not exist or is invalid. Ensure TIFF file is present in current directory or valid.

1104 5794 MCIERR_FAX_TIF_TIFBMP_NOSUCHPAGEINTIFF
Cannot find the specified page number in TIFF file because it does not exit in the TIFF file. Specify a valid page number.

1105 5795 MCIERR_FAX_TIF_TIFBMP_CANNOTCREATETEMPMH
Cannot create **temp.mh**, which is the intermediate image file extracted from the TIFF file for converting to BMP format because there is insufficient disk space. Free up disk space.

1106 5796 MCIERR_FAX_TIF_TIFBMP_GALLOCHTIFF
Cannot allocate global memory for reading image data from TIFF file. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1107 5797 MCIERR_FAX_TIF_TIFBMP_GLOCKHTIFF
Cannot lock memory for reading image data from TIFF file. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1108 5798 MCIERR_FAX_TIF_TIFBMP_LOADCCITTIMAGE
Cannot decode MH data into BMP format. Invalid MH image file, device context, bitmap handle. Check image filename, device context, bitmap handle.

1109 5799 MCIERR_FAX_TIF_TIFBMP_SAVEBITMAP
Cannot save the bitmap into a file because there is insufficient memory for buffers to store bitmap. Ensure sufficient RAM and hard disk space is available.

1110 5800 MCIERR_FAX_TIF_TIFBMP_NOTTIFFFILE
Source file is not a TIFF file. Ensure source file is a valid TIFF file.

1133 5801 MCIERR_FAX_TIF_INSERT_LOADBITMAPFROMFILE
Cannot load bitmap into memory from the BMP file, because either the BMP file is invalid or there is insufficient memory to load the bitmap. Ensure sufficient RAM space and valid BMP file.

1134 5802
MCIERR_FAX_TIF_INSERT_SAVEBITMAPINCCITTFORMAT
Cannot save the memory bitmap into a MH file. The encoding of the bitmap into MH format has failed; this indicates a device context problem. Check device context and ensure the bitmap is valid.

1135 5803 MCIERR_FAX_TIF_INSERT_OPENTEMPMH
Cannot open the **temp.mh** file created by **SaveBitmapInCcittFormat** function. Too many files are open. Check the **FILE** parameter in **config.sys** to ensure that it is sufficiently large and close all unnecessary files.

1136 5804 MCIERR_FAX_TIF_INSERT_OPENTIFF
Cannot open the TIFF file because the specified TIFF file is either invalid or non-existent. Check that TIFF file exists in current directory and is a valid one.

1137 5805 MCIERR_FAX_TIF_INSERT_GALLOCHTIFF
Cannot allocate global memory for storing MH data. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1138 5806 MCIERR_FAX_TIF_INSERT_GLOCKHTIFF
Cannot lock memory for storing MH data. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1163 5807 MCIERR_FAX_TIF_REPLACE_LOADBITMAPFROMFILE
Cannot load bitmap into memory from the BMP file. Invalid BMP file; insufficient memory to load the bitmap. Ensure sufficient RAM space and valid BMP file.

1164 5808
MCIERR_FAX_TIF_REPLACE_SAVEBITMAPINCCITTFORMAT
Cannot save the memory bitmap into a MH file. Encoding of bitmap into MH format failed; device context problem. Check device context; ensure valid bitmap.

1165 5809 MCIERR_FAX_TIF_REPLACE_CANNOTOPENTEMPMH
Cannot open the **temp.mh** file created by **SaveBitmapInCcittFormat** function because too many files are open. Verify that the **FILES** parameter in **config.sys** is sufficiently large and close all unnecessary files.

1166 5810 MCIERR_FAX_TIF_REPLACE_OPENTIFF
Cannot open the specified TIFF file. Check that TIFF file exists in current directory and is valid.

1167 5811 MCIERR_FAX_TIF_REPLACE_OPENTEMPTIFF
Cannot create **temp.tif** for duplicating current TIFF file because there is a DOS or Windows problem. Check system configuration.

1168 5812 MCIERR_FAX_TIF_REPLACE_GALLOCHTIFF
Cannot allocate global memory for storing MH data. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1169 5813 MCIERR_FAX_TIF_REPLACE_GLOCKHTIFF
Cannot lock memory for storing MH data. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1170 5814 MCIERR_FAX_TIF_REPLACE_CANNOTCREATETIFF
Cannot create the new TIFF file because there is a DOS or Windows problem. Check system configuration.

1193 5815 MCIERR_FAX_TIF_BMPTIF_LOADBITMAPFROMFILE
Cannot load bitmap into memory from the BMP file. If there is insufficient memory to load the bitmap, increase the available RAM space. Verify that the BMP file is valid.

1194 5816 MCIERR_FAX_TIF_BMPTIF_SAVEBITMAPINCCITTFORMAT
Cannot save the memory bitmap into a MH file. The encoding of bitmap into MH format has failed, indicating a device context problem. Check device context and ensure the bitmap is valid.

1195 5817 MCIERR_FAX_TIF_BMPTIF_CANNOTOPENTEMPMH
Cannot open the **temp.mh** file created by **SaveBitmapInCcittFormat** function because too many files are open. Check the **FILES** parameter in **config.sys** to ensure that it is sufficiently large and close all unnecessary files.

1196 5818 MCIERR_FAX_TIF_BMPTIF_CANNOTCREATETIFF
Cannot create the TIFF file because the specified TIFF filename is invalid. Check that the TIFF filename supplied is valid.

1197 5819 MCIERR_FAX_TIF_BMPTIF_GALLOCHTIFF
Cannot allocate global memory for storing MH data. See **MCIERR_FAX_MHTIF_GALLOCHWRITE**.

1198 5820 MCIERR_FAX_TIF_BMPTIF_GLOCKHTIFF
Cannot lock memory for storing MH data. See MCIERR_FAX_TIF_MHTIF_GALLOCHWRITE.

1199 5821 MCIERR_FAX_TIF_BMPTIF_NOTBMPFILE
The specified source file is not a valid **.bmp** file. Ensure source file is a valid BMP file.

1213 5957 MCIERR_FAX_TIF_NUMBER_CANNOTOPENTIFF
Unable to open the specified TIFF file. The TIFF file is either invalid or non-existent. Check for TIFF file validity or that it exists in the current directory.

MEIO error codes

One MEIO-specific error code exists:

1313 6057 MCIERR_MEIO_DSPMEIOCONNECT
Mwave MEIO disconnect error.

APPENDIX A - String Interfaces

The following two sections describe the MCI string interface for FAX and TAM. The string interface allows you to use English-language commands to communicate with MCI devices. An overview of the string interface is provided in the *Microsoft Windows Multimedia Programmer's Reference* and the *IBM Multimedia Presentation Manager Toolkit/2 Programming Reference.*

A1 - String Interface FAX

This section describes the string interface for Mwave FAX under OS/2 and Windows 3.1.

MCI_CLOSE

MCI_CLOSE contains no extensions specific to the Mwave FAX API. It is the standard MCI_CLOSE call. See the *Microsoft Windows Multimedia Programmer's Reference* or the *IBM Multimedia Presentation Manager Toolkit/2 Programming Reference* for the exact syntax.

Example: close fax wait

MCI_CONVERT

MCI_CONVERT does contain Mwave specific values. These values pertain to bitmap and tiff file formats.

String Interface	Command Flag Equivalent
convert	MCI_CONVERT
notify	MCI_NOTIFY
wait	MCI_WAIT
info	MCI_CONVERT_INFO
overwrite	MCI_CONVERT_OVERWRITE
create	MCI_CONVERT_CREATE
destination file	MCI_CONVERT_DESTINATION_FILE
destination format	MCI_CONVERT_DESTINATION_FORMAT
dib bmp	MCI_CONVERT_FMT_DIB_BMP
dib rle	MCI_CONVERT_FMT_DIB_RLE
devfax	MCI_FAX_CONVERT_FMT_DEVFAX
destination from	MCI_CONVERT_DESTINATION_FROM
source file	MCI_CONVERT_SOURCE_FILE
source from	MCI_CONVERT_SOURCE_FROM

Example:

```
convert fax create wait destination file c:\viewfax.bmp
  destination format dib bmp destination from 0
  source file c:\rcvdfax.tif
```

MCI_DIAL

MCI_DIAL is not part of the base MCI calls. It is completely defined by Mwave.

String Interface	Command Flag Equivalent
dial	MCI_DIAL
notify	MCI_NOTIFY
wait	MCI_WAIT
flash	MCI_DIAL_FLASH
monitor	MCI_DIAL_MONITOR
monitor handshake	MCI_DIAL_MONITOR_HANDSHAKING_ONLY
verify	MCI_DIAL_VERIFY
dial mode	MCI_DIAL_DIALMODE
pulse	MCI_DIAL_MODE_PULSE
tone	MCI_DIAL_MODE_TONE

Examples: dial fax 919-254-7410 wait
dial fax 9,1-900-555-1212 monitor handshake notify

MCI_GETDEVCAPS

MCI_GETDEVCAPS has some Mwave specific extensions. MCI_GETDEVCAPS returns information as a null terminated string. Windows returns all information as an ASCII representation of an integer. So, if the MCI API defines the output as TRUE, FALSE, windows will return '0' or '1'.

For OS/2 MMPM, MCI_GETDEVCAPS has some Mwave specific extensions. MCI_GETDEVCAPS returns a value that depends upon the particular capability that was queried. Under MMPM, the high order word of the return code indicates the type of data that is returned. In most cases MCI_TRUE_FALSE_RETURN type is returned. This means the string that is returned contains "TRUE" or "FALSE". A number of calls don't return true or false. They are:

- 'device type' which returns a type of MCI_DEVICENAME_RETURN. The returned string in this case is "Other".

- 'compression types' which returns a type of MCI_USER_RETURN_COMPRESS. The returned string is "MH","MR", "MMR", "NONE" or "BFT", "ANY", "1D", "2D"

- 'modem types' which returns a type of MCI_USER_RETURN_MODEMS. The returned string is "V27TER 2400", "V27TER 4800" "V29 7200", "V29 9600", "V17 7200", "V17 9600", "V17 12000", "V17 14400", "V27TER(2400,4800), V29(7200, 9600)", or "V27TER(2400,4800), V29(7200, 9600), or V17(7200, 9600, 12000, 14400)"

- 'resolution' which returns a type of MCI_USER_RETURN_RESOLUTION. The returned string is "Fine(200x200)", or "Normal(100x200)"

- 'file formats' which returns a type of MCI_USER_RETURN_FILE_FORMATS. The returned string is "Tiff Class F", "DCX", "RIFF", or "TIFF 6",

String Interface	Command Flag Equivalent
capability	MCI_GETDEVCAPS
notify	MCI_NOTIFY
wait	MCI_WAIT
can eject	MCI_GETDEVCAPS_CAN_EJECT

can play	MCI_GETDEVCAPS_CAN_PLAY
can record	MCI_GETDEVCAPS_CAN_RECORD
can save	MCI_GETDEVCAPS_CAN_SAVE
compound device	MCI_GETDEVCAPS_COMPOUND_DEVICE
device type	MCI_GETDEVCAPS_DEVICE_TYPE
has audio	MCI_GETDEVCAPS_HAS_AUDIO
has video	MCI_GETDEVCAPS_HAS_VIDEO
uses files	MCI_GETDEVCAPS_USES_FILES
modem types	MCI_FAX_GETDEVCAPS_MODEM_TYPES
compression types	MCI_FAX_GETDEVCAPS_COMPRESSION_TYPES
can receive	MCI_FAX_GETDEVCAPS_CAN_RECEIVE
can send	MCI_FAX_GETDEVCAPS_CAN_SEND
has handset	MCI_FAX_GETDEVCAPS_HAS_HANDSET
supports ecm	MCI_FAX_GETDEVCAPS_SUPPORTS_ECM
polling	MCI_FAX_GETDEVCAPS_POLLING
file formats	MCI_FAX_GETDEVCAPS_FILE_FORMATS
resolution	MCI_FAX_GETDEVCAPS_RESOLUTION
width	MCI_FAX_GETDEVCAPS_WIDTH

Example: capability fax modem types wait

MCI_INFO

MCI_INFO is extended by Mwave to include Caller ID support. MCI info returns a string. In the case of caller ID, this string may not successfully be converted to ASCII since it contains non-ASCII characters.

String Interface	Command Flag Equivalent
info	MCI_INFO
notify	MCI_NOTIFY
wait	MCI_WAIT
product	MCI_INFO_PRODUCT
caller id error	MCI_INFO_CALLER_ID_ERROR
caller id	MCI_INFO_CALLER_ID
parsed caller id	MCI_INFO_PARSED_CALLER_ID

Example: info fax caller id wait

MCI_OPEN

MCI_OPEN contains no extensions specific to the Mwave FAX API. It is the standard MCI_OPEN call. See the *Microsoft Windows Multimedia Programmer's Reference* or the *IBM Multimedia Presentation Manager Toolkit/2 Programming Reference* for the exact syntax.

Example: open mwavefax alias fax notify

MCI_RECEIVE

MCI_RECEIVE is unique to Mwave FAX support. The already dialed parameter is used for manually receiving a fax.

String Interface	Command Flag Equivalent
receive	MCI_RECEIVE
notify	MCI_NOTIFY
wait	MCI_WAIT
already dialed	MCI_ALREADY_DIALED

Examples:

```
receive fax c:\newfax01.tif notify
receive fax c:\newfax01.tif already dialed wait
```

MCI_SEND

MCI_SEND is unique to Mwave FAX support. The already dialed parameter is used for manually sending a fax. If this parameter is not specified, the document is not sent until 'dial' is issued.

String Interface	Command Flag Equivalent
send	MCI_SEND
notify	MCI_NOTIFY
wait	MCI_WAIT
already dialed	MCI_ALREADY_DIALED
send heading	MCI_SEND_HEADING

Examples:

```
send fax c:\outfax01.tif notify
send fax c:\outfax01.tif already dialed wait
```

MCI_SET

MCI_SET contains many extensions specific to the Mwave API. The Windows version does not permit symbolic keywords for the information that is being set. Further, in the windows version, it is necessary to use the keyword 'value' before specifying the information you are setting. In the table below, the third column shows the valid values that can be set.

String Interface	Command Flag Equivalent	Valid Values
set	MCI_SET	
notify	MCI_NOTIFY	
wait	MCI_WAIT	
audio volume	MCI_FAX_SET_AUDIO_VOLUME	integer
call filter	MCI_FAX_SET_CALL_FILTER	0, 1
API style	MCI_FAX_SET_API_STYLE	1 = MMPM 2 = windows
dial flash time	MCI_FAX_SET_DIAL_FLASH_TIME	integer
dial pause time	MCI_FAX_SET_DIAL_PAUSE_TIME	integer
dial wait time	MCI_FAX_SET_DIAL_WAIT_TIME	integer
event handler	MCI_FAX_SET_EVENT_HANDLER	integer
hook	MCI_FAX_SET_HOOK	0, 1
pass call	MCI_FAX_SET_PASS_CALL	16 = voice 8 = modem
advanced ring	MCI_FAX_SET_ADVANCED_RING_NOTIFY	0, 1

compression types	MCI_FAX_SET_COMPRESSION_TYPES	1 = MH 2 = MR 4 = MMR 8 = NONE 16 = BFT 32 = ANY 64 = 1D 128 = 2D
ecm level	MCI_FAX_SET_ECM_LEVEL	
polling	MCI_FAX_SET_POLLING	0, 1
resolution	MCI_FAX_SET_RESOLUTION	1 = normal 2 = fine
station id	MCI_FAX_SET_STATION_ID	ASCII
ring count	MCI_FAX_SET_RING_COUNT	integer
modem types	MCI_FAX_SET_MODEM_TYPES	01 = V27TER_2400 02 = V27TER_4800 04 = V29_7200 08 = V29_9600 16 = V17_7200 32 = V17_9600 64 = V17_12000 128 = V17_14400

Examples: set fax station id value 919-543-3113 wait
set fax hook value 1 notify
set fax event handler value 48937930 wait
set fax pass call value 16 notify

MCI_SET in the MMPM version permits symbolic keywords for the information that is being set. Further, in the MMPM version, it is not necessary to use the keyword 'value' before specifying the information you are setting. In the table below, the third column shows the valid values that can be set.

String Interface	Command Flag Equivalent	Valid Values
set	MCI_SET	
notify	MCI_NOTIFY	
wait	MCI_WAIT	
audio volume	MCI_FAX_SET_AUDIO_VOLUME	integer
call filter	MCI_FAX_SET_CALL_FILTER	FALSE, TRUE
API style	MCI_FAX_SET_API_STYLE	mmpm, windows
dial flash time	MCI_FAX_SET_DIAL_FLASH_TIME	integer
dial pause time	MCI_FAX_SET_DIAL_PAUSE_TIME	integer
dial wait time	MCI_FAX_SET_DIAL_WAIT_TIME	integer
event handler	MCI_FAX_SET_EVENT_HANDLER	integer
hook	MCI_FAX_SET_HOOK	FALSE, TRUE
pass call	MCI_FAX_SET_PASS_CALL	voice, modem
advanced ring	MCI_FAX_SET_ADVANCED_RING_NOTIFY	FALSE, TRUE
compression types	MCI_FAX_SET_COMPRESSION_TYPES	MH, MR, MMR, NONE, BFT,ANY,

		1D, 2D
ecm level	MCI_FAX_SET_ECM_LEVEL	not supported
polling	MCI_FAX_SET_POLLING	FALSE, TRUE
resolution	MCI_FAX_SET_RESOLUTION	normal, fine
station id	MCI_FAX_SET_STATION_ID	ASCII
ring count	MCI_FAX_SET_RING_COUNT	integer
modem types	MCI_FAX_SET_MODEM_TYPES	V27TER 2400, V27TER 4800, V29 7200, V29 9600, V17 7200, V17 9600, V17 12000, V17 14400

Examples: set fax station id 919-543-3113 wait
set fax hook true notify
set fax event handler 48937930 wait
set fax pass call voice notify

MCI_STATUS

MCI_STATUS has many Mwave FAX specific extensions. In addition, it returns information. Under windows, it is up to the application to know how to interpret the information. The third column in the table below indicates what the returned values mean.

String Interface	Command Flag Equivalent	Returned Values
status	MCI_STATUS	
notify	MCI_NOTIFY	
wait	MCI_WAIT	
time format	MCI_STATUS_TIME_FORMAT	0 = milliseconds
length	MCI_STATUS_LENGTH	integer
mode	MCI_STATUS_MODE	1 = receive 2 = send 524 = not ready 530 = open
position	MCI_STATUS_POSITION	integer
ready	MCI_STATUS_READY	0, 1
audio volume	MCI_FAX_STATUS_AUDIO_VOLUME	integer
call filter	MCI_FAX_STATUS_CALL_FILTER	0, 1
dial flash time	MCI_FAX_STATUS_DIAL_FLASH_TIME	integer
dial pause time	MCI_FAX_STATUS_DIAL_PAUSE_TIME	integer
dial wait time	MCI_FAX_STATUS_DIAL_WAIT_TIME	integer
handset	MCI_FAX_STATUS_HANDSET	0 = down 1 = up
hook	MCI_FAX_STATUS_HOOK	0 = on hook 1 = off hook
line	MCI_FAX_STATUS_LINE	1 = on hook 2 = dial tone 3 = busy 4 = ring tone 6 = unknown

ring count	MCI_FAX_STATUS_RING_COUNT	integer
polling	MCI_FAX_STATUS_POLLING	0 = no polling 1 = polling
resolution	MCI_FAX_STATUS_RESOLUTION	1 = normal 2 = fine
station id	MCI_FAX_STATUS_STATION_ID	ASCII
compression types	MCI_FAX_STATUS_COMPRESSION_TYPES	1 = MH 2 = MR 4 = MMR 8 = NONE 16=BFT 32 = ANY 64 = 1D 128 = 2D
max modem types	MCI_FAX_STATUS_MAX_MODEM_SPEED	01 = V27TER_2400 02 = V27TER_4800 04 = V29_7200 08 = V29_9600 16 = V17_7200 32 = V17_9600 64 = V17_12000 128 = V17_14400
min modem types	MCI_FAX_STATUS_MIN_MODEM_SPEED	01 = V27TER_2400 02 = V27TER_4800 04 = V29_7200 08 = V29_9600 16 = V17_7200 32 = V17_9600
worldtrade support	MCI_FAX_STATUS_WORLDTRADE_SUPPORT	integer
country code	MCI_FAX_STATUS_COUNTRY_CODE	integer
min rings allowed	MCI_FAX_STATUS_AUTO_ANSWER_MIN_RINGS	integer
max rings allowed	MCI_FAX_STATUS_AUTO_ANSWER_MAX_RINGS	integer
max call retries	MCI_FAX_STATUS_MAX_CALL_RETRIES	integer
min call retry time	MCI_FAX_STATUS_MIN_CALL_RETRY_TIME	integer (in seconds)

Examples: status fax hook wait
status fax station id notify
status fax ring count wait

In MMPM, MCI_STATUS has many Mwave FAX specific extensions. In addition, it returns information. Under MMPM the high order word of the return code indicates the type of returned information.

String Interface	Command Flag Equivalent	Returned Values
status	MCI_STATUS	
notify	MCI_NOTIFY	
wait	MCI_WAIT	
time format	MCI_STATUS_TIME_FORMAT	milliseconds
length	MCI_STATUS_LENGTH	integer

mode	MCI_STATUS_MODE	not ready sending receiving open
position	MCI_STATUS_POSITION	integer
ready	MCI_STATUS_READY	FALSE, TRUE
call filter	MCI_FAX_STATUS_CALL_FILTER	FALSE, TRUE
dial flash time	MCI_FAX_STATUS_DIAL_FLASH_TIME	integer
dial pause time	MCI_FAX_STATUS_DIAL_PAUSE_TIME	integer
dial wait time	MCI_FAX_STATUS_DIAL_WAIT_TIME	integer
handset	MCI_FAX_STATUS_HANDSET	down up
hook	MCI_FAX_STATUS_HOOK	on hook off hook
line	MCI_FAX_STATUS_LINE	on hook dial tone busy ring tone unknown
ring count	MCI_FAX_STATUS_RING_COUNT	integer
polling	MCI_FAX_STATUS_POLLING	FALSE (no polling) TRUE (polling)
resolution	MCI_FAX_STATUS_RESOLUTION	normal , fine
station id	MCI_FAX_STATUS_STATION_ID	ASCII
compression types	MCI_FAX_STATUS_COMPRESSION_TYPES	MH, MR, MMR, NONE, ANY, 1D, 2D, BFT
max modem types	MCI_FAX_STATUS_MAX_MODEM_SPEED	V27TER 2400 , V27TER 4800, V29 7200 , V29 9600, V17 7200, V17 9600, V17 12000, V17 14400
	V27TER(2400,4800), V29(7200, 9600), V27TER(2400,4800), V29(7200,9600), V17(7200,9600,12000,14400)	
min modem types	MCI_FAX_STATUS_MIN_MODEM_SPEED	V27TER 2400 V27TER 4800 V29 7200 V29 9600 V17 7200 V1 9600 V17 12000 V17 14400
	V27TER(2400,4800), V29(7200, 9600), V27TER(2400,4800), V29(7200,9600), V17(7200,9600,12000,14400)	

Examples: status fax hook wait
status fax station id notify
status fax ring count wait

MCI_STOP

MCI_STOP has no Mwave FAX extensions.

Example: stop fax wait

A2 - String Interface TAM

This section describes the string interface for Mwave TAM under OS/2 2.1 and Windows 3.1

MCI_CLOSE
MCI_CLOSE contains no extensions specific to the Mwave TAM API. It is the standard MCI_CLOSE call. See the *Microsoft Windows Multimedia Programmer's Reference* or the *IBM Multimedia Presentation Manager Toolkit/2 Programming Reference* for the exact syntax.

Example: close tam wait

MCI_CONVERT
MCI_CONVERT does contain Mwave specific values. Note that it is not supported in releases before driver version 3.1.

String Interface	Command Flag Equivalent
convert	MCI_CONVERT
notify	MCI_NOTIFY
wait	MCI_WAIT
info	MCI_CONVERT_INFO
overwrite	MCI_CONVERT_OVERWRITE
create	MCI_CONVERT_CREATE
destination file	MCI_CONVERT_DESTINATION_FILE
destination format	MCI_CONVERT_DESTINATION_FORMAT
wave pcm	MCI_CONVERT_FMT_WAVE_PCM
devtam	MCI_TAM_CONVERT_FMT_DEVTAM
destination from	MCI_CONVERT_DESTINATION_FROM
length	MCI_CONVERT_LENGTH
source file	MCI_CONVERT_SOURCE_FILE
source from	MCI_CONVERT_SOURCE_FROM

Example: convert tps create wait destination file c:\newwave.wav
  destination format wave pcm destination from 0
  source file c:\recorded.voi

MCI_DIAL
MCI_DIAL is not part of the base MCI calls. It is completely defined by Mwave.

String Interface	Command Flag Equivalent
dial	MCI_DIAL
notify	MCI_NOTIFY
wait	MCI_WAIT
flash	MCI_DIAL_FLASH
monitor	MCI_DIAL_MONITOR
verify	MCI_DIAL_VERIFY
dial mode	MCI_DIAL_DIALMODE
pulse	MCI_DIAL_MODE_PULSE
tone	MCI_DIAL_MODE_TONE

Examples: dial tpl 919-254-7410 wait
dial tpl verify 9,1-900-555-1212 dial mode pulse notify

MCI_GETDEVCAPS

MCI_GETDEVCAPS has some Mwave specific extensions. MCI_GETDEVCAPS returns information as a null terminated string. Windows returns all information as an ASCII representation of an integer. So, if the MCI API defines the output as TRUE, FALSE, windows will return '0' or '1'.

For MMPM, MCI_GETDEVCAPS returns a value that depends upon the particular capability that was queried. The high order word of the return code indicates the type of data that is returned. In most cases MCI_TRUE_FALSE_RETURN type is returned. This means the string that is returned contains "TRUE" or "FALSE". The only exception is - 'device type' which returns a type of MCI_DEVICENAME_RETURN. The returned string in this case is "Other".

String Interface	Command Flag Equivalent
capability	MCI_GETDEVCAPS
notify	MCI_NOTIFY
wait	MCI_WAIT
can eject	MCI_GETDEVCAPS_CAN_EJECT
can play	MCI_GETDEVCAPS_CAN_PLAY
can record	MCI_GETDEVCAPS_CAN_RECORD
can save	MCI_GETDEVCAPS_CAN_SAVE
compound device	MCI_GETDEVCAPS_COMPOUND_DEVICE
device type	MCI_GETDEVCAPS_DEVICE_TYPE
has audio	MCI_GETDEVCAPS_HAS_AUDIO
has video	MCI_GETDEVCAPS_HAS_VIDEO
uses files	MCI_GETDEVCAPS_USES_FILES
supports custom tag	MCI_TAM_GETDEVCAPS_SUPPORTS_CUSTOM_TAG
supports pcm tag	MCI_TAM_GETDEVCAPS_SUPPORTS_PCM_TAG

Example: capability tps can save wait

MCI_INFO

MCI_INFO is extended by Mwave to include Caller ID support. MCI info returns a string. In the case of caller ID, this string may not successfully be converted to ASCII since it contains non-ASCII characters.

In the case of caller ID for MMPM, an integer is returned since the caller ID contains non-ASCII characters. The application must type cast the integer to an address and then use the contents of the address to retrieve the caller ID information.

String Interface	Command Flag Equivalent
info	MCI_INFO
notify	MCI_NOTIFY
wait	MCI_WAIT
product	MCI_INFO_PRODUCT
caller id error	MCI_INFO_CALLER_ID_ERROR
caller id	MCI_INFO_CALLER_ID
parsed caller id	MCI_INFO_PARSED_CALLER_ID

Example: info tps product wait

MCI_LOAD

MCI_LOAD contains no extensions specific to the Mwave TAM API. It is the standard MCI_LOAD call. See the *Microsoft Windows Multimedia Programmer's Reference* or the *IBM Multimedia Presentation Manager Toolkit/2 Programming Reference* for the exact syntax.

Example:

```
load tpl c:\greeting.voi wait
load tpl new wait           /*this opens a new, empty file (OS/2)*/
load tpl "" wait            /*this opens a new empty file (windows)*/
```

MCI_OPEN

MCI_OPEN contains no extensions specific to the Mwave TAM API. It is the standard MCI_OPEN call. See the Windows Multimedia Developer's Manual or the MMPM/2 Programmer's Reference Manual for the exact syntax.

Example:

```
open mwavetps alias tps notify
open mwavetpl alias tpl wait
```

MCI_PAUSE

MCI_PAUSE contains no extensions specific to the Mwave TAM API. It is the standard MCI_PAUSE call. See the *Microsoft Windows Multimedia Programmer's Reference* or the *IBM Multimedia Presentation Manager Toolkit/2 Programming Reference* for the exact syntax.

Example:

```
pause tps wait
```

MCI_PLAY

MCI_PLAY contains no extensions specific to the Mwave TAM API. It is the standard MCI_PLAY call. See the *Microsoft Windows Multimedia Programmer's Reference* or the *IBM Multimedia Presentation Manager Toolkit/2 Programming Reference* for the exact syntax.

Example:

```
play tpl notify
play tps from 10 to 50 wait
```

MCI_RECORD

MCI_RECORD has an Mwave-specific extension to the base MCI record call to allow beeping before recording begins.

String Interface	Command Flag Equivalent
record	MCI_RECORD
notify	MCI_NOTIFY
wait	MCI_WAIT
insert	MCI_RECORD_INSERT
overwrite	MCI_RECORD_OVERWRITE
to message end	MCI_TAM_TO_MESSAGE_END
beep	MCI_TAM_BEEP
from	MCI_FROM
to	MCI_TO

Examples:

```
record tps notify
record tps from 20 to 50 insert notify
record tpl beep notify
```

MCI_RESUME
MCI_RESUME contains no extensions specific to the Mwave TAM API. It is the standard MCI_RESUME call. See the *Microsoft Windows Multimedia Programmer's Reference* or the *IBM Multimedia Presentation Manager Toolkit/2 Programming Reference* for the exact syntax.

Example: resume tps notify

MCI_SAVE
MCI_SAVE contains no extensions specific to the Mwave TAM API. It is the standard MCI_SAVE call. See the Windows Multimedia Developer's Manual or the MMPM/2 Programmer's Reference Manual for the exact syntax.

Example: save tps c:\newname.voi wait

MCI_SEEK
MCI_SEEK contains no extensions specific to the Mwave TAM API. It is the standard MCI_SEEK call. See the *Microsoft Windows Multimedia Programmer's Reference* or the *IBM Multimedia Presentation Manager Toolkit/2 Programming Reference* for the exact syntax.

Example: seek tpl to 100 wait

MCI_SET
MCI_SET contains many extensions specific to the Mwave API. The Windows version does not permit symbolic keywords for the information that is being set. Further, in the windows version, it is necessary to use the keyword 'value' before specifying the information you are setting. In the table below, the third column shows the valid values that can be set.

String Interface	Command Flag Equivalent	Valid Values
set	MCI_SET	
notify	MCI_NOTIFY	
wait	MCI_WAIT	
audio mute	MCI_TAM_SET_AUDIO_MUTE	0, 1
audio volume	MCI_TAM_SET_AUDIO_VOLUME	integer
avgbytespersec	MCI_TAM_SET_AVGBYTESPERSEC	integer
bitspersample	MCI_TAM_SET_BITSPERSAMPLE	integer
call filter	MCI_TAM_SET_CALL_FILTER	0, 1
API style	MCI_TAM_SET_API_STYLE	1 = MMPM 2 = windows
ap discriminated	MCI_TAM_SET_AP_DISCRIMINATED	4 = FAX 8 = modem 16 = VOICE 32 = Don’t Answer
connect	MCI_TAM_SET_CONNECT	1 = audio 2 = handset 4 = phoneline 5 = audio&phoneline 6 = handset&phoneline 8 = speaker 12 =speaker&phoneline

dial flash time	MCI_TAM_SET_DIAL_FLASH_TIME	integer
dial pause time	MCI_TAM_SET_DIAL_PAUSE_TIME	integer
dial wait time	MCI_TAM_SET_DIAL_WAIT_TIME	integer
event handler	MCI_TAM_SET_EVENT_HANDLER	integer
formattag	MCI_TAM_SET_FORMATTAG	1 = wave 2 = custom
handset mute	MCI_TAM_SET_HANDSET_MUTE	0, 1
handset volume	MCI_TAM_SET_HANDSET_VOLUME	integer
hook	MCI_TAM_SET_HOOK	0, 1
caller id	MCI_TAM_SET_CALLER_ID	0, 1
quality	MCI_TAM_SET_QUALITY	integer
quiet	MCI_TAM_SET_QUIET_DURATION	integer
ring count	MCI_TAM_SET_RING_COUNT	integer
samplespersec	MCI_TAM_SET_SAMPLESPERSEC	integer
speed	MCI_TAM_SET_SPEED	integer
pass call	MCI_TAM_SET_PASS_CALL	4 = FAX 8 = modem
advanced ring	MCI_TAM_SET_ADVANCED_RING_NOTIFY	0, 1
microphone gain	MCI_TAM_SET_MICROPHONE_GAIN	integer
dial min flash time	MCI_TAM_SET_DIAL_MIN_FLASH_TIME	integer
dial max flash time	MCI_TAM_SET_DIAL_MAX_FLASH_TIME	integer
low level wave io	MCI_TAM_SET_LOW_LEVEL_WAVE_IO	1 = wave in start 2 = wave in stop 4 = wave out start 8 = wave out stop

Examples: set tps microphone gain value 75 wait
set tpl hook value 1 notify
set tpl event handler value 48937930 wait
set tpl pass call value 4 notify

MCI_SET in the MMPM version permits symbolic keywords for the information that is being set. Further, in the MMPM version, it is not necessary to use the keyword 'value' before specifying the information you are setting. In the table below, the third column shows the valid values that can be set.

String Interface	Command Flag Equivalent	Valid Values
set	MCI_SET	
notify	MCI_NOTIFY	
wait	MCI_WAIT	
audio mute	MCI_TAM_SET_AUDIO_MUTE	FALSE, TRUE
audio volume	MCI_TAM_SET_AUDIO_VOLUME	integer
call filter	MCI_TAM_SET_CALL_FILTER	FALSE, TRUE
avgbytespersec	MCI_TAM_SET_AVGBYTESPERSEC	integer
bitspersample	MCI_TAM_SET_BITSPERSAMPLE	integer
API style	MCI_TAM_SET_API_STYLE	mmpm, windows
ap discriminated	MCI_TAM_SET_AP_DISCRIMINATED	fax, modem, voice, dont answer
connect	MCI_TAM_SET_CONNECT	audio handset

		phoneline audio&phoneline handset&phoneline speaker speaker&phoneline
dial flash time	MCI_TAM_SET_DIAL_FLASH_TIME	integer
dial pause time	MCI_TAM_SET_DIAL_PAUSE_TIME	integer
dial wait time	MCI_TAM_SET_DIAL_WAIT_TIME	integer
event handler	MCI_TAM_SET_EVENT_HANDLER	integer
formattag	MCI_TAM_SET_FORMATTAG	pcm, custom
handset mute	MCI_TAM_SET_HANDSET_MUTE	FALSE, TRUE
handset volume	MCI_TAM_SET_HANDSET_VOLUME	integer
hook	MCI_TAM_SET_HOOK	FALSE, TRUE
caller id	MCI_TAM_SET_CALLER_ID	FALSE, TRUE
quality	MCI_TAM_SET_QUALITY	integer
quiet	MCI_TAM_SET_QUIET_DURATION	integer
ring count	MCI_TAM_SET_RING_COUNT	integer
samplespersec	MCI_TAM_SET_SAMPLESPERSEC	integer
speed	MCI_TAM_SET_SPEED	integer
pass call	MCI_TAM_SET_PASS_CALL	fax, modem
advanced ring	MCI_TAM_SET_ADVANCED_RING_NOTIFY	FALSE, TRUE
microphone gain	MCI_TAM_SET_MICROPHONE_GAIN	integer
dial min flash time	MCI_TAM_SET_DIAL_MIN_FLASH_TIME	integer
dial max flash time	MCI_TAM_SET_DIAL_MAX_FLASH_TIME	integer
low level wave io	MCI_TAM_SET_LOW_LEVEL_WAVE_IO	wave in start, wave in stop, wave out start, wave out stop

Examples: set tps microphone gain 75 wait
set tpl hook true notify
set tpl event handler 48937930 wait
set tpl pass call fax notify

MCI_STATUS

MCI_STATUS has many Mwave TAM specific extensions. In addition, it returns information. Under windows, it is up to the application to know how to interpret the information. The third column in the table below indicates what the returned values mean.

String Interface	Command Flag Equivalent	Returned Values
status	MCI_STATUS	
notify	MCI_NOTIFY	
wait	MCI_WAIT	
time format	MCI_STATUS_TIME_FORMAT	0 = milliseconds
length	MCI_STATUS_LENGTH	integer
mode	MCI_STATUS_MODE	524 = not ready 525 = stop 526 = play 527 = record 528 = seek 529 = pause 530 = open
position	MCI_STATUS_POSITION	integer

ready	MCI_STATUS_READY	0, 1
audio mute	MCI_TAM_STATUS_AUDIO_MUTE	0, 1
audio volume	MCI_TAM_STATUS_AUDIO_VOLUME	integer
avgbytespersec	MCI_TAM_STATUS_AVGBYTESPERSEC	integer
bitspersample	MCI_TAM_STATUS_BITSPERSAMPLE	integer
connect	MCI_TAM_STATUS_CONNECT	1 = audio 2 = handset 4 = phoneline 5 = audio&phoneline 6 = handset&phoneline 8 = speaker 12 = speaker&phoneline
call filter	MCI_TAM_STATUS_CALL_FILTER	0, 1
dial flash time	MCI_TAM_STATUS_DIAL_FLASH_TIME	integer
dial pause time	MCI_TAM_STATUS_DIAL_PAUSE_TIME	integer
dial wait time	MCI_TAM_STATUS_DIAL_WAIT_TIME	integer
formattag	MCI_TAM_STATUS_FORMATTAG	1 = wave; 2= custom
handset mute	MCI_TAM_STATUS_HANDSET_MUTE	0, 1
handset volume	MCI_TAM_STATUS_HANDSET_VOLUME	integer
handset	MCI_TAM_STATUS_HANDSET	0 = down 1 = up
hook	MCI_TAM_STATUS_HOOK	0 = on hook 1 = off hook
line	MCI_TAM_STATUS_LINE	1 = on hook 2 = dial tone 3 = busy 4 = ring tone 5 = quiet 6 = unknown 7 = voice 8 = modem 9 = fax
quality	MCI_TAM_STATUS_QUALITY	integer
quiet	MCI_TAM_STATUS_QUIET_DURATION	integer
ring count	MCI_TAM_STATUS_RING_COUNT	integer
samplespersec	MCI_TAM_STATUS_SAMPLESPERSEC	integer
speed	MCI_TAM_STATUS_SPEED	integer
caller id	MCI_TAM_STATUS_CALLER_ID	1 = active 2 = disabled 3 = unsupported
microphone gain	MCI_TAM_STATUS_MICROPHONE_GAIN	integer
dial min flash time	MCI_TAM_STATUS_DIAL_MIN_FLASH_TIME	integer
dial max flash time	MCI_TAM_STATUS_DIAL_MAX_FLASH_TIME	integer
worldtrade support	MCI_TAM_STATUS_WORLDTRADE_SUPPORT	integer
max mic gain	MCI_TAM_STATUS_MAX_MIC_GAIN	integer
max audio volume	MCI_TAM_STATUS_MAX_AUDIO_VOLUME	integer
max greeting len no rec	MCI_TAM_STATUS_MAX_GREETING_LEN_NO_REC	integer
max greeting len	MCI_TAM_STATUS_MAX_GREETING_LEN	integer
max record len	MCI_TAM_STATUS_MAX_RECORD_LEN	integer
max msg retrieve len	MCI_TAM_STATUS_MAX_MSG_RETRIEVE_LEN	integer
country code	MCI_TAM_STATUS_COUNTRY_CODE	integer
min rings allowed	MCI_TAM_STATUS_AUTO_ANSWER_MIN_RINGS	integer
max rings allowed	MCI_TAM_STATUS_AUTO_ANSWER_MAX_RINGS	integer
max call retries	MCI_TAM_STATUS_MAX_CALL_RETRIES	integer

min call retry time	MCI_TAM_STATUS_MIN_CALL_RETRY_TIME	integer (in seconds)

Examples:
status tpl hook wait
status fax station id notify
status tpl handset volume wait

In MMPM, MCI_STATUS has many Mwave TAM specific extensions. In addition, it returns information. Under MMPM the high order word of the return code indicates the type of returned information.

String Interface	Command Flag Equivalent	Returned Values
status	MCI_STATUS	
notify	MCI_NOTIFY	
wait	MCI_WAIT	
time format	MCI_STATUS_TIME_FORMAT	milliseconds
	(type is MCI_TIME_FORMAT_RETURN)	
length	MCI_STATUS_LENGTH	integer
mode	MCI_STATUS_MODE	not ready
		stop
		play
		record
		seek
		pause
		open
	(type is MCI_MODE_RETURN)	
position	MCI_STATUS_POSITION	integer
ready	MCI_STATUS_READY	FALSE, TRUE
audio mute	MCI_TAM_STATUS_AUDIO_MUTE	FALSE, TRUE
audio volume	MCI_TAM_STATUS_AUDIO_VOLUME	integer
avgbytespersec	MCI_TAM_STATUS_AVGBYTESPERSEC	integer
bitspersample	MCI_TAM_STATUS_BITSPERSAMPLE	integer
connect	MCI_TAM_STATUS_CONNECT	audio
		handset
		phoneline
		audio&phoneline
		handset&phoneline
		speaker
		speaker&phoneline
	(type is MCI_CONNECTOR_TYPE_RETURN)	
call filter	MCI_TAM_STATUS_CALL_FILTER	FALSE, TRUE
dial flash time	MCI_TAM_STATUS_DIAL_FLASH_TIME	integer
dial pause time	MCI_TAM_STATUS_DIAL_PAUSE_TIME	integer
dial wait time	MCI_TAM_STATUS_DIAL_WAIT_TIME	integer
formattag	MCI_TAM_STATUS_FORMATTAG	custom format
		wave format
	(type is MCI_FORMAT_TAG_RETURN)	
handset mute	MCI_TAM_STATUS_HANDSET_MUTE	FALSE, TRUE
handset volume	MCI_TAM_STATUS_HANDSET_VOLUME	integer
handset	MCI_TAM_STATUS_HANDSET	down
		up
hook	MCI_TAM_STATUS_HOOK	TRUE
		FALSE

line	MCI_TAM_STATUS_LINE	on hook dial tone busy ringing quiet unknown voice modem fax (type is MCI_USER_RETURN_LINE)
quality	MCI_TAM_STATUS_QUALITY	integer
quiet	MCI_TAM_STATUS_QUIET_DURATION	integer
ring count	MCI_TAM_STATUS_RING_COUNT	integer
samplespersec	MCI_TAM_STATUS_SAMPLESPERSEC	integer
speed	MCI_TAM_STATUS_SPEED	integer
caller id	MCI_TAM_STATUS_CALLER_ID	active disabled unsupported (type is MCI_USER_RETURN_CALLER_ID)
microphone gain	MCI_TAM_STATUS_MICROPHONE_GAIN	integer
dial min flash time	MCI_TAM_STATUS_DIAL_MIN_FLASH_TIME	integer
dial max flash time	MCI_TAM_STATUS_DIAL_MAX_FLASH_TIME	integer
worldtrade support	MCI_TAM_STATUS_WORLDTRADE_SUPPORT	integer
max mic gain	MCI_TAM_STATUS_MAX_MIC_GAIN	integer
max audio volume	MCI_TAM_STATUS_MAX_AUDIO_VOLUME	integer
max greeting len no rec	MCI_TAM_STATUS_MAX_GREETING_LEN_NO_REC	integer
max greeting len	MCI_TAM_STATUS_MAX_GREETING_LEN	integer
max record len	MCI_TAM_STATUS_MAX_RECORD_LEN	integer
max msg retrieve len	MCI_TAM_STATUS_MAX_MSG_RETRIEVE_LEN	integer
country code	MCI_TAM_STATUS_COUNTRY_CODE	string

The following information is returned:

MCI_RETURN_TYPE_STRING For Country Code

COUNTRY	STRING	COUNTRY	STRING
USA or Canada	WT_COUNTRY_USA_CANADA,0,	Germany	WT_COUNTRY_GERMANY,0,
Belgium	WT_COUNTRY_BELGIUM,0,	Brazil	WT_COUNTRY_BRAZIL,0,
Hong Kong	WT_COUNTRY_HONG_KONG,0,	Russia	WT_COUNTRY_RUSSIA,0,
Singapore	WT_COUNTRY_SINGAPORE,0,	Yugoslavia	WT_COUNTRY_YUGOSLAVIA,0,
New Zealand	WT_COUNTRY_NEW_ZEALAND,0,	Hungary	WT_COUNTRY_HUNGARY,0,
Japan	WT_COUNTRY_JAPAN,0,	Czech Republic	WT_COUNTRY_CZECHREPUBLIC,0,
Portugal	WT_COUNTRY_PORTUGAL,0,	Luxembourg	WT_COUNTRY_LUXEMBORG,0,
Ireland	WT_COUNTRY_IRELAND,0,	Norway	WT_COUNTRY_NORWAY,0,
Generic	WT_COUNTRY_GENERIC,0,	Denmark	WT_COUNTRY_DENMARK,0,
Spain	WT_COUNTRY_SPAIN,0,	France	WT_COUNTRY_FRANCE,0,
Greece	WT_COUNTRY_GREECE,0,	Netherlands	WT_COUNTRY_NETHERLANDS,0,
Israel	WT_COUNTRY_ISRAEL,0,	United Kingdom	WT_COUNTRY_U_K,0,
Taiwan	WT_COUNTRY_TAIWAN,0,	Sweden	WT_COUNTRY_SWEDEN,0,
Australia	WT_COUNTRY_AUSTRALIA,0	Italy	WT_COUNTRY_ITALY,0,
Austria	WT_COUNTRY_AUSTRIA,0,	Finland	WT_COUNTRY_FINLAND,0,
Mexico	WT_COUNTRY_MEXICO,0,	Thailand	WT_COUNTRY_THAILAND,0,
South Africa	WT_COUNTRY_SOUTH_AFRICA,0,	Korea	WT_COUNTRY_KOREA,0
Chile	WT_COUNTRY_CHILE,0,	Malaysia	WT_COUNTRY_MALAYSIA,0,
Switzerland	WT_COUNTRY_SWITZERLAND,0,	China	WT_COUNTRY_PRC,0,

min rings allowed	MCI_TAM_STATUS_AUTO_ANSWER_MIN_RINGS	integer
max rings allowed	MCI_TAM_STATUS_AUTO_ANSWER_MAX_RINGS	integer
max call retries	MCI_TAM_STATUS_MAX_CALL_RETRIES	integer
min call retry time	MCI_TAM_STATUS_MIN_CALL_RETRY_TIME	integer (in seconds)

Examples: status tpl hook wait
status tps speed notify
status tpl handset volume wait

MCI_STOP
MCI_STOP has one Mwave TAM extension: the ability to stop a recording and remove DTMF keys.

String Interface	Command Flag Equivalent
stop	MCI_STOP
notify	MCI_NOTIFY
wait	MCI_WAIT
remove dtmf	MCI_STOP_REMOVE_DTMF

Example: stop tps wait
stop tpl remove dtmf wait

APPENDIX B - Programmer's Notes

B1 - Fax Notes

Warning: The fax and TAM drivers are different. Do not assume they work the same way.

Do not hang up the phone in any situation other than PHONE_EVENT_CALL_TERMINATED.

NOTE: If the user wants to abort the call then the app should issue a MCI_STOP. The app will then receive a PHONE_EVENT_CALL_TERMINATED.

B2 - TAM Notes

Warning: The fax and TAM drivers are different. Do not assume they work the same way.

The app should hang up the phone for the following reasons:

- 5 uninterrupted CALL_PROGRESS_QUIET events in a row
- 5 CALL_PROGRESS_DIAL_TONE events in a row
- A CALL_PROGRESS_QUIET event during play

Warning: Not all phone systems put out a dial tone when a person has hung up. The application should count the number of uninterrupted quiets when not in a record or play state then hang up. The TAM driver will not give you a phone event call terminated.
An unidentified tone event should reset the quiet and dial tone counter.

The application can load a file before a call in order to save processing time during the call.

Call progress event unidentified tone can be received during a transition state i.e. between ring tone and answer tone.

Beware of connection conflicts (handset , audio) between the TPS and TPL drivers
When the TPL driver is opened it will come up connected to normal phone.
When the TPS driver is opened it will come up connected to audio.

If the dial fails then the application needs to hang up the phone (DO NOT DO THIS WITH THE FAX!!).

A mci-stop should be made with the notify flag set so that it will be queued up behind the outstanding play or record.

The application should be aware that the discriminator can cause the record to be aborted. In such a case the app will receive a PHONE_EVENT_CALL_TERMINATED as well.

Set audio volume vs. handset volume

Application MUST perform WinCreateMsgQueue(HAB, 100) - OS/2
SetMessageQueue(100) - Windows
This will ensure that the application does not lose event messages. The windows default is only 8!!!

If a record or play fails the application still needs to do a MCI_STOP

Set speed takes nearest match (rounding down) if no exact match.

Set speakerphone mode disables discrimination on calling tones.

The application can not load a file while a play or record is in progress.

If MCI_OPEN fails and the error code > 512 then just print out the number because GetErrorString will not provide the correct information.
MciGetErrorString requires a device id along with the error code. When MCI_OPEN fails no device id is returned therefore calling MciGetErrorString with the error code > 512 (fax/tam error codes) will return an incorrect error string. The error string that will be returned will be for a device driver currently in the system and your error code may be something like (cdaudio error).

The microphone should not be right next to the monitor, it will cause problems.

If in speakerphone mode answer tone will not work properly.

On hook and Off hook is electrical.

TAM set audio volume cant be done during dial.

The application can only get handset key events in TPS if TPS is connected to handset. Currently TPL only gets flash "!".

Status position during Play From is not updated until play gets going. To get around this the application should seek to 0 before play to get current position in right place.

Application can not set volume during dial

There is no volume change for the phoneline record or play.

Mci set quality only with new file loaded.

Rule: If app takes phone off hook it is responsible for putting it back on hook, else the driver does it.

B3 - Integrated Application Notes

If the app chooses to use the same event handler for both fax and tam, care should be taken as to which device an event is from.
i.e. PHONE_EVENT_CALL_TERMINATED will be sent to the tam app when a call has been discriminated as a fax call. The app should issue an on hook (making sure to use the TAM DEVICE ID).

The app writer should be aware that mci_set and mci_status use the same constants.
i.e. MCI_TAM_SET_EVENT_HANDLER is different than MCI_FAX_SET_EVENT_HANDLER

APPENDIX C - Mwave Play and Record Mixer Definition File

The following is example source code for the Mwave play and record mixer definition file. A copy of this code can be found on the companion diskette.

```
//-------------------------------
//Mwave Play and Record mixer definition file
//Some hints in creating:
// Dont use the pound sign it is a special character
// Dont use any of the key words
// Always put something as the last thing in the file so that we dont reach
// eof before or during the read of the last record
// Last modification 3 13 95
//-------------------------------
#DESTNUM
2,              // Number of Destinations (waveply, waverecord)
#SOURCENUM
6,              // Real Number of Sources IN THIS ORDER ( MIDI, WAVE, SB, CD, LINE, MIC)
#TOTALCONTROLS
76,             // total number of controls
//-------------------------------
// source controls map
// The first index is the dest no
// The second is the "relative" source no
// The value is the no of controls at this source as connected to this dest.
//-------------------------------
#CONTROLSSRC
4,    // d0, rs 0 (MIDI), 4 c (VOL, BAL, MUTE, PM)
4,    // d0, rs 1 (WAVE), 4 c (VOL, BAL, MUTE, PM)
4,    // d0, rs 2 (SB), 3 c (VOL, BAL, MUTE, PM)
5,    // d0, rs 3 (CD), 5 c (VOL, BAL, MUTE, PM, SWITCH SELECT)
5,    // d0, rs 4 (LINE), 5 c (VOL, BAL, MUTE, PM, SWITCH SELECT)
5,    // d0, rs 5 (MIC), 5 c (VOL, BAL, MUTE, PM, SWITCH SELECT)
4,    // d1, rs 0 (MIDI), 4 c (VOL, BAL, MUTE, PM)
4,    // d1, rs 1 (WAVE), 4 c (VOL, BAL, MUTE, PM)
4,    // d1, rs 2 (SB), 4 c (VOL, BAL, MUTE, PM)
5,    // d1, rs 3 (CD), 5 c (VOL, BAL, MUTE, PM, SWITCH SELECT)
5,    // d1, rs 4 (LINE), 5 c (VOL, BAL, MUTE, PM, SWITCH SELECT)
5,    // d1, rs 5 (MIC), 5 c (VOL, BAL, MUTE, PM, SWITCH SELECT)
//------------------------------
// auControlMap- these come in triplets
// The index is the control number
// The first UINT is the destination.
// The second INT is the "relative" source number
// When "relative" source num is -1 (INT_MAX), the control is "at the dest."
// The third entry is the number of channels for this control
//-------------------------------
#CONTROLMAP
0,   //Control 0 (volume), dest 0 (waveply), at dest, 2 channels
-1,
2,
0,   //Control 1 (balance), dest 0 (waveply), at dest, 2 channels
-1,
2,
0,   //Control 2 (mute), dest 0 (waveply), at dest, 1 channel
-1,
1,
0,   //Control 3 (peakmeter), dest 0 (waveply), at dest, 2 channels
```

```
-1,
2,
0,   //Control 4 (qsound on), dest 0 (waveply), at dest, 2 channels
-1,
1,
0,   //Control 5 (reverb on), dest 0 (waveply), at dest, 1 channels
-1,
1,
0,   //Control 6 (reverb depth), dest 0 (waveply), at dest, 2 channels
-1,
2,
0,   //Control 7  (chorus depth), dest 0 (waveply), at dest, 2 channels
-1,
2,
0,   //Control 8 (treble on), dest 0 (waveply), at dest, 1 channels
-1,
1,
0,   //Control 9 (treble slider), dest 0 (waveply), at dest, 2 channels
-1,
2,
0,   //Control 10(bass slider), dest 0 (waveply), at dest, 2 channels  A
-1,
2,
1,   //Control 11 (volume), dest 1 (waverec), at dest, 2 channels      B
-1,
2,
1,   //Control 12 (balance), dest 1 (waverec), at dest, 2 channels     C
-1,
2,
1,   //Control 13 (mute), dest 1 (waverec), at dest, 1 channel         D
-1,
1,
1,   //Control 14 (peakmeter), dest 1 (waverec), at dest, 2 channels E
-1,
2,
1,   //Control 15 (qsound on), dest 1 (waverec), at dest, 1 channels F
-1,
1,
1,   //Control 16(reverb on), dest 0 (waveply), at dest, 1 channels   10
-1,
1,
1,   //Control 17(reverb depth), dest 1 (waveply), at dest, 2 channels             11
-1,
2,
1,   //Control 18 (chorus depth), dest 1 (waveply), at dest, 2 channels 12
-1,
2,
1,   //Control 19 (treble on), dest 1 (waverec), at dest, 1 channels   13
-1,
1,
1,   //Control 20 (treble depth), dest 1 (waverec), at dest, 2 channels             14
-1,
2,
1,   //Control 21 (bass depth), dest 1 (waverec), at dest, 2 channels 15
-1,
2,
0,   //Control 22 (volume), dest 0 (waveply), source 0 (midiout), 2 channels  16
0,
2,
0,   //Control 23 (balance), dest 0 (waveply), source 0 (midiout), 2 channels  17
0,
2,
0,   //Control 24 (mute), dest 0 (waveply), source 0 (midi), 1 channels            18
0,
1,
0,  //Control 25 (pm), dest 0 (waveply), source 0 (midi), 2 channels                   19
0,
2,
0,  //Control 26 (volume), dest 0 (waveply), source 1 (wave), 2 channels     1A
1,
2,
0,  //Control 27 (balance), dest 0 (waveply), source 1 (wave), 2 channels    1B
1,
2,
0,  //Control 28 (mute), dest 0 (waveply), source 1 (wave), 1 channels       1C
1,
1,
0,  //Control 29 (pm), dest 0 (waveply), source 1 (wave), 2 channels         1D
1,
2,
0,  //Control 30 (volume), dest 0 (waveply), source 2 (SB), 2 channels       1E
2,
2,
0,  //Control 31 (balance), dest 0 (waveply), source 2 (SB), 2 channels      1F
2,
2,
0,  //Control 32 (mute), dest 0 (waveply), source 2 (SB), 1 channels         20
2,
1,
0,  //Control 33 (pm), dest 0 (waveply), source 2 (SB), 2 channels           21
2,
2,
0,  //Control 34 (volume), dest 0 (waveply), source 3 (CD), 2 channels       22
3,
2,
0,  //Control 35 (balance), dest 0 (waveply), source 3 (CD), 2 channels      23
3,
2,
0,  //Control 36 (mute), dest 0 (waveply), source 3 (CD), 1 channels         24
3,
1,
0,  //Control 37 (pm), dest 0 (waveply), source 3 (CD), 2 channels           25
3,
2,
0,  //Control 38 (on or off), dest 0 (waveply), source 3 (CD), 1 channels    26
3,
1,
0,  //Control 39 (volume), dest 0 (waveply), source 4 (LINE), 2 channels     27
4,
2,
0,  //Control 40 (balance), dest 0 (waveply), source 4 (LINE), 2 channels    28
4,
2,
0,  //Control 41 (mute), dest 0 (waveply), source 4 (LINE), 1 channels       29
4,
1,
0,  //Control 42 (pm), dest 0 (waveply), source 4 (LINE), 2 channels         2A
4,
2,
0,  //Control 43 (on or off), dest 0 (waveply), source 4 (LINE), 1 channels  2B
4,
1,
0,  //Control 44 (volume), dest 0 (waveply), source 5 (MIC), 2 channels      2C
5,
2,
0,  //Control 45 (balance), dest 0 (waveply), source 5 (MIC), 2 channels     2D
5,
2,
0,   //Control 46 (mute), dest 0 (waveply), source 5 (MIC), 1 channels         2E
5,
1,
0,   //Control 47 (pm), dest 0 (waveply), source 5 (MIC), 2 channels                      2F
5,
2,
0,   //Control 48 (on or off), dest 0 (waveply), source 5 (MIC), 1 channels     30
5,
1,
1,   //Control 49 (volume), dest 1 (waverec), source 0 (midiout), 2 channels    31
0,
2,
1,   //Control 50 (balance), dest 1 (waverec), source 0 (midiout), 2 channels   32
0,
2,
1,   //Control 51 (mute), dest 1 (waverec), source 0 (midi), 1 channels         33
0,
1,
1,   //Control 52 (pm), dest 1 (waverec), source 0 (midi), 2 channels                     34
0,
2,
1,   //Control 53 (volume), dest 1 (waverec), source 1 (wave), 2 channels       35
1,
2,
1,   //Control 54 (balance), dest 1 (waverec), source 1 (wave), 2 channels
1,
2,
1,   //Control 55 (mute), dest 1 (waverec), source 1 (wave), 1 channels
1,
1,
1,   //Control 56 (pm), dest 1 (waverec), source 1 (wave), 2 channels
1,
2,
1,   //Control 57 (volume), dest 1 (waverec), source 2 (SB), 2 channels
2,
2,
1,   //Control 58 (balance), dest 1 (waverec), source 2 (SB), 2 channels
2,
2,
1,   //Control 59 (mute), dest 1 (waverec), source 2 (SB), 1 channels
2,
1,
1,   //Control 60 (pm), dest 1 (waverec), source 2 (SB), 2 channels
2,
2,
1,   //Control 61 (volume), dest 1 (waverec), source 3 (CD), 2 channels
3,
2,
1,   //Control 62 (balance), dest 1 (waverec), source 3 (CD), 2 channels
3,
2,
1,   //Control 63 (mute), dest 1 (waverec), source 3 (CD), 1 channels
3,
1,
1,   //Control 64 (pm), dest 1 (waverec), source 3 (CD), 2 channels
3,
2,
1,   //Control 65 (on or off), dest 1 (waverec), source 3 (CD), 1 channels
3,
1,
1,   //Control 66 (volume), dest 1 (waverec), source 4 (LINE), 2 channels
4,
2,
1,   //Control 67 (balance), dest 1 (waverec), source 4 (LINE), 2 channels
```

```
4,
2,
1,    //Control 68 (mute), dest 1 (waverec), source 4 (LINE), 1 channels
4,
1,
1,    //Control 69 (pm), dest 1 (waverec), source 4 (LINE), 2 channels
4,
2,
1,    //Control 70 (on or off), dest 1 (waverec), source 4 (LINE), 1 channels
4,
1,
1,    //Control 71 (volume), dest 1 (waverec), source 5 (MIC), 2 channels
5,
2,
1,    //Control 72 (balance), dest 1 (waverec), source 5 (MIC), 2 channels
5,
2,
1,    //Control 73 (mute), dest 1 (waverec), source 5 (MIC), 1 channels
5,
1,
1,    //Control 74 (pm), dest 1 (waverec), source 5 (MIC), 2 channels
5,
2,
1,    //Control 75 (on or off), dest 1 (waverec), source 5 (MIC), 1 channels
5,
1,
//-------------------------------
// Source map, maps a relative source to the actual source per destination.
// these come in pairs
// The first index is the destination no
// the 2nd the relative source no (0,1,2 etc)
// then each one that doesnt exist at that dest gets a UINT_MAX
// and the third the actual source number
//-------------------------------
#SOURCEMAP
0, // for dest 0, relsource 0, actsource 0
1, // for dest 0, relsource 1, actsource 1
2, // for dest 0, relsource 2, actsource 2
3, // for dest 0, relsource 3, actsource 3
4, // for dest 0, relsource 4, actsource 4
5, // for dest 0, relsource 5, actsource 5
0, // for dest 1, relsource 0, actsource 0
1, // for dest 1, relsource 1, actsource 1
2, // for dest 1, relsource 2, actsource 2
3, // for dest 1, relsource 3, actsource 3
4, // for dest 1, relsource 4, actsource 4
5, // for dest 1, relsource 5, actsource 5
//-------------------------------
// Source Definitions
//-------------------------------
//Source0 - MidiOut
//-------------------------------
#SOURCEDEF
0,            // dwDestination
0,            // dwSource SOURCE_MIDIOUT
0,            // dwLineID  SOURCE_MIDIOUT
h80000000,    // MIXERLINE_LINEF_SOURCE fdwLine
0,            // dwUser
h00001004,    // MIXERLINE_COMPONENTTYPE_SRC_SYNTHESIZER dwComponentType
2,            // cChannels
0,            // cConnections
0,            // cControls
Midi,         // short name
Midi Play Out,// long name
```

```
MIDI,         // szLineTypeName (my keyword for type of line)
MIDIOUT,      // szLineLongName (my keyword)
3,            // MIXERLINE_TARGETTYPE_MIDIOUT  target dwType
0,            // target dwDeviceID
1,            // target wMid MM_MICROSOFT
h7FFF,        // PID_SYNTH target wPid
h0100,        // DRV_VERSION_SYNTH target vDriverVersion
Mwave MIDI Synthesizer,     // target szPname
//-------------------------------
// Source 1 - WaveOut
//-------------------------------


#SOURCEDEF
0,                          // dwDestination
1,                          // dwSource SOURCE_WAVEOUT
1,                          // dwLineID  SOURCE_WAVEOUT
h80000000,                  // MIXERLINE_LINEF_SOURCE fdwLine
0,                          // dwUser
h00001008,                  // MIXERLINE_COMPONENTTYPE_SRC_WAVEOUT dwComponentType
2,                          // cChannels
0,                          // cConnections
0,                          // cControls
Wave,                       // short name
Wave Player Output,         // long name
WAVE,                       // szLineTypeName (my keyword for type of line)
WAVEOUT,                    // szLineLongName (my keyword)
1,                          // MIXERLINE_TARGETTYPE_WAVEOUT  target dwType
0,                          // target dwDeviceID
1,                          // target wMid MM_IBM
15,                         // PID_WAVEOUT target wPid
h0200,                      // DRV_VERSION_WAVEOUT target vDriverVersion
Mwave Wave Audio Driver,// target szPname
//------------------------------
// Source 2 - SB Out (games)
//-------------------------------
#SOURCEDEF
0,                          // dwDestination (not use for sources)
2,                          // dwSource SOURCE_SB
2,                          // dwLineID  SOURCE_SB
h80000000,                  // MIXERLINE_LINEF_SOURCE fdwLine
0,                          // dwUser
h00001000,                  // MIXERLINE_COMPONENTTYPE_SRC_UNDEFINED dwComponentType
2,                          // cChannels
0,                          // cConnections
0,                          // cControls
SndBlstr,                   // short name
SoundBlaster,               // long name
GAMES,                      // szLineTypeName (my keyword for type of line)
SOUNDBLASTER,               // szLineLongName (my keyword)
0,                          // MIXERLINE_TARGETTYPE_UNDEFINED  target dwType
0,                          // target dwDeviceID
0,                          // target wMid
0,                          // target wPid
0,                          // target vDriverVersion
Undefined,                  // target szPname
//------------------------------
// Source 3 - SB CD
//-------------------------------
#SOURCEDEF
0,                          // dwDestination (not used for sources)
3,                          // dwSource CD
3,                          // dwLineID CD
h80000000,                  // MIXERLINE_LINEF_SOURCE fdwLine
```

```
0,                             // dwUser
h00001005,                     // MIXERLINE_COMPONENTTYPE_SRC_COMPACTDISC dwComponentType
2,                             // cChannels
0,                             // cConnections
0,                             // cControls
CD,                            // short name
CD,                            // long name
CD,                            // szLineTypeName (my keyword for type of line)
CD,                            // szLineLongName (my keyword)
0,                             // MIXERLINE_TARGETTYPE_UNDEFINED  target dwType
0,                             // target dwDeviceID
0,                             // target wMid
0,                             // target wPid
0,                             // target vDriverVersion
Undefined,                     // target szPname
//------------------------------
// Source 4 - SB LINE
//-------------------------------
#SOURCEDEF
0,                             // dwDestination (not used for sources)
4,                             // dwSource LINE
4,                             // dwLineID LINE
h80000000,                     // MIXERLINE_LINEF_SOURCE fdwLine
0,                             // dwUser
h00001002,                     // MIXERLINE_COMPONENTTYPE_SRC_LINE dwComponentType
2,                             // cChannels
0,                             // cConnections
0,                             // cControls
LINE,                          // short name
LINE,                          // long name
LINE,                          // szLineTypeName (my keyword for type of line)
LINE,                          // szLineLongName (my keyword)
0,                             // MIXERLINE_TARGETTYPE_UNDEFINED  target dwType
0,                             // target dwDeviceID
0,                             // target wMid
0,                             // target wPid
0,                             // target vDriverVersion
Undefined,                     // target szPname
//------------------------------
// Source 5 - MIC
//-------------------------------
#SOURCEDEF
0,                             // dwDestination (not used for sources)
5,                             // dwSource MIC
5,                             // dwLineID MIC
h80000000,                     // MIXERLINE_LINEF_SOURCE fdwLine
0,                             // dwUser
h00001003,                     // MIXERLINE_COMPONENTTYPE_SRC_MICROPHONE dwComponentType
2,                             // cChannels
0,                             // cConnections
0,                             // cControls
MIC,                           // short name
MIC,                           // long name
MIC,                           // szLineTypeName (my keyword for type of line)
MIC,                           // szLineLongName (my keyword)
0,                             // MIXERLINE_TARGETTYPE_UNDEFINED  target dwType
0,                             // target dwDeviceID
0,                             // target wMid
0,                             // target wPid
0,                             // target vDriverVersion
Undefined,                     // target szPname
//-------------------------------
// mxlDestinations
//-------------------------------
```

```
// Destination 0 - Waveout


//-------------------------------
#DESTDEF
0,                       // dwDestination DEST_WAVEOUT
0,                       // dwSource
hFFFF0000,               // dwLineID
0,                       // fdwLine
0,                       // dwUser
h00000004,               // MIXERLINE_COMPONENTTYPE_DST_SPEAKERS dwComponentType
2,                       // cChannels
6,                       // cConnections
11,                      // cControls
Master,                  // short name
Master Speaker Out,      // long name
WAVE,                    // szLineTypeName (my keyword for type of line)
WAVEOUT,                 // szLineLongName (my keyword)
1,                       // MIXERLINE_TARGETTYPE_WAVEOUT  target dwType
0,                       // target dwDeviceID
1,                       // target wMid MM_IBM
15,                      // PID_WAVEOUT target wPid
h0200,                   // DRV_VERSION_WAVEOUT target vDriverVersion
Mwave Wave Audio Driver,// target szPname
//-------------------------------
// Destination 1 - Wavein
//-------------------------------
#DESTDEF
1,                       // dwDestination DEST_WAVEIN
0,                       // dwSource
hFFFF0001,               // dwLineID
0,                       // fdwLine
0,                       // dwUser
h00000007,               // MIXERLINE_COMPONENTTYPE_DST_WAVEIN dwComponentType
2,                       // cChannels
6,                       // cConnections
11,                      // cControls
Master,                  // short name
Master Record In,        // long name
WAVEIN,                  // szLineTypeName (my keyword for type of line)
WAVEIN,                  // szLineLongName (my keyword)
2,                       // MIXERLINE_TARGETTYPE_WAVEIN  target dwType
0,                       // target dwDeviceID
1,                       // target wMid MM_IBM
14,                      // PID_WAVEIN target wPid
h0200,                   // DRV_VERSION_WAVEIN target vDriverVersion
Mwave Wave Audio Driver,// target szPname
//-------------------------------
// end of initialization of destinations
//----------------------------------------------
// mxc (controls)
//----------------------------------------------
//
//  The following numbers are the destination numbers:
//    DEST_WAVEOUT       0
//    DEST_WAVEIN        1
//
//  The following numbers are the relative source numbers for dest 0
//    SOURCE_MIDIOUT     0
//    SOURCE_WAVEOUT     1
//    SOURCE_SB          2
//    SOURCE_CD          3
//    SOURCE_LINE        4
//    SOURCE_MIC         5
```

```
//
//  The following numbers are the relative source numbers for dest 1
//    SOURCE_MIDIOUT    0
//    SOURCE_WAVEOUT  1
//    SOURCE_CD             3
//    SOURCE_LINE          4
//    SOURCE_MIC            5
//
//-----------------------------------------------
//Control0     - MASTER volume at DAC
//-----------------------------------------------
#CONTROLDEF
0,                        // VOL_OUTMIDI dwControlID
h50030001,                // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                        // fdwControl
0,                        // cMultipleItems
VOLUME,                   // szShortName
MASTER,                   // szName
VOLUME,                   // szControlTypeName (my keyword for type of control)
MASTER,                   // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                        // Bounds.dwMinimum
hFFFF,                    // Bounds.dwMaximum
16,                       // Metrics.cSteps
SPKRLVOL,                 // ini file entry
SPKRRVOL,                 // ini file entry
//-----------------------------------------------
//Control1     - MASTER balance at DAC
//-----------------------------------------------
#CONTROLDEF
1,                        // BAL_LINE dwControlID
h40020001,                // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                        // fdwControl
0,                        // cMultipleItems
BALANCE,                  // szShortName
MASTER,                   // szName
BALANCE,                  // szControlTypeName (my keyword for type of control)
MASTER,                   // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                   // Bounds.dwMinimum
32767,                    // Bounds.dwMaximum
16,                       // Metrics.cSteps
BALMAST,                  // INI file entry
BALMAST,                  // INI file entry (only 1 is used)
//-----------------------------------------------
//Control2     - MUTE of Master DAC
//-----------------------------------------------
#CONTROLDEF
2,                        // MUTE_OUTLINE dwControlID
h20010002,                // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                        // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                        // cMultipleItems
MUTE,                     // szShortName
MASTER,                   // szName
MUTE,                     // szControlTypeName (my keyword for type of control)
MASTER,                   // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                        // Bounds.dwMinimum
1,                        // Bounds.dwMaximum
1,                        // Metrics.cSteps
MUTEMAST,                 // ini file entry
//-----------------------------------------------
//Control3     - MASTER Peak meter at DAC
//-----------------------------------------------
```

```
#CONTROLDEF
3,                      // VU_LINEOUT dwControlID
h10020001,              // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
PEAKMETER,              // szShortName
MASTER,                 // szName
PEAKMETER,              // szControlTypeName (my keyword for type of control)
MASTER,                 // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                 // Bounds.dwMinimum
32767,                  // Bounds.dwMaximum
0,                      // Metrics.cSteps
NONE,                   // ini file entry
//----------------------------------------------
//Control4    - Q Sound enable at DAC
//----------------------------------------------
#CONTROLDEF
4,                      // QSND_ENABLE dwControlID
h20010005,              // MIXERCONTROL_CONTROLTYPE_STEREOENH dwControlType
1,                      // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                      // cMultipleItems
ENABLE,                 // szShortName
QSOUND,                 // szName
QSOUND,                 // szControlTypeName (my keyword for type of control)
MASTER,                 // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
1,                      // Bounds.dwMaximum
0,                      // Metrics.cSteps
QSOUND,                 // INI file entry
//----------------------------------------------
//Control5   - REVERB and CHORUS ON at DAC
//----------------------------------------------
#CONTROLDEF
5,                      // REV_ENABLE dwControlID
h20010000,              // MIXERCONTROL_CONTROLTYPE_BOOLEAN dwControlType
1,                      // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                      // cMultipleItems
ENABLE,                 // szShortName
EFFECTS ON,             // szName
REVEN,                  // szControlTypeName (my keyword for type of control)
MASTER,                 // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
1,                      // Bounds.dwMaximum
1,                      // Metrics.cSteps
REVEN,                  // INI file entry
//----------------------------------------------
//Control6   - REVERB depth at DAC
//----------------------------------------------
#CONTROLDEF
6,                      // REV_ENABLE dwControlID
h50030000,              // MIXERCONTROL_CONTROLTYPE_FADER dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
REVERB,                 // szShortName
REVERB DEPTH,                   // szName
REVERB,                 // szControlTypeName (my keyword for type of control)
MASTER,                 // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
65535,                  // Bounds.dwMaximum
32,                     // Metrics.cSteps
```

```
REVMAST,                // INI file entry
REVMAST,                // INI file entry
//---------------------------------------------
//Control7   - CHORUS DEPTH at DAC
//---------------------------------------------
#CONTROLDEF
7,                      // REV_ENABLE dwControlID
h50030000,              // MIXERCONTROL_CONTROLTYPE_FADER dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
CHORUS,                 // szShortName
CHORUS DEPTH,                   // szName
CHORUS,                 // szControlTypeName (my keyword for type of control)
MASTER,                 // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
65535,                  // Bounds.dwMaximum
32,                     // Metrics.cSteps
CHOMAST,
CHOMAST,
//---------------------------------------------
//Control8     - MASTER BASS and TREBLE enable at DAC
//---------------------------------------------
#CONTROLDEF
8,                      // BASS_LINEOUT dwControlID
h20010000,              // MIXERCONTROL_CONTROLTYPE_BOOLEAN dwControlType
1,                      // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                      // cMultipleItems
ENABLE,                 // szShortName
TREBLE ENABLE,          // szName
BASSEN,                 // szControlTypeName (my keyword for type of control)
MASTER,                 // szControlLineName (my keyword for line thiscontrol applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
1,                      // Bounds.dwMaximum
1,                      // Metrics.cSteps
BASSEN,                 // INI file entry
//---------------------------------------------
//Control9     - MASTER TREBLE slider at DAC
//---------------------------------------------
#CONTROLDEF
9,                      // BASS_LINEOUT dwControlID
h50030003,              // MIXERCONTROL_CONTROLTYPE_TREBLE dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
TREBLE,                 // szShortName
TREBLE CONTROL,         // szName
TREBLE,                 // szControlTypeName (my keyword for type of control)
MASTER,                 // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
65535,                  // Bounds.dwMaximum
32,                     // Metrics.cSteps
TREMAST,                // INI file entry
TREMAST,                // INI file entry (only 1 is used)
//---------------------------------------------
//Control10    - MASTER Bass slider at dac
//---------------------------------------------
#CONTROLDEF
10,                     // BASS_LINEOUT dwControlID
h50030002,              // MIXERCONTROL_CONTROLTYPE_BASS dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
BASS,                   // szShortName
```

```
BASS SLIDER,            // szName
BASS,                   // szControlTypeName (my keyword for type of control)
MASTER,                 // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
65535,                  // Bounds.dwMaximum
32,                     // Metrics.cSteps
BASMAST,                // INI file entry
BASMAST,                // INI file entry (only 1 is used)
//----------------------------------------------
//Control11   - MASTER volume at wavein
//----------------------------------------------
#CONTROLDEF
11,                     // VOL_WAVEIN dwControlID
h50030001,              // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
VOLUME,                 // szShortName
MASTERIN,               // szName
VOLUME,                 // szControlTypeName (my keyword for type of control)
MASTERIN,               // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
hFFFF,                  // Bounds.dwMaximum
16,                     // Metrics.cSteps
RECLVOL,                // ini file entry
RECRVOL,                // ini file entry
//----------------------------------------------
//Control12    - MASTER balance at wavein
//----------------------------------------------
#CONTROLDEF
12,                     // BAL_LINE dwControlID
h40020001,              // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
BALANCE,                // szShortName
MASTERIN,               // szName
BALANCE,                // szControlTypeName (my keyword for type of control)
MASTERIN,               // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                 // Bounds.dwMinimum
32767,                  // Bounds.dwMaximum
16,                     // Metrics.cSteps
BALMASTIN,              // INI file entry
BALMASTIN,              // INI file entry (only 1 is used)
//----------------------------------------------
//Control13    - MUTE of Master wavein
//----------------------------------------------
#CONTROLDEF
13,                     // MUTE_OUTLINE dwControlID
h20010002,              // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                      // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                      // cMultipleItems
MUTE,                   // szShortName
MASTERIN,               // szName
MUTE,                   // szControlTypeName (my keyword for type of control)
MASTERIN,               // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
1,                      // Bounds.dwMaximum
0,                      // Metrics.cSteps
MUTEMASTIN,             // ini file entry
//----------------------------------------------
//Control14    - Peak meter at wavin dest
```

```
//-----------------------------------------------
#CONTROLDEF
14,                     // VU_LINEOUT dwControlID
h10020001,              // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
PEAKMETER,              // szShortName
MASTERIN,               // szName
PEAKMETER,              // szControlTypeName (my keyword for type of control)
MASTERIN,               // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                 // Bounds.dwMinimum
32767,                  // Bounds.dwMaximum
0,                      // Metrics.cSteps
NONE,                   // ini file entry
//-----------------------------------------------
//Control15   - Q Sound enable at wi
//-----------------------------------------------
#CONTROLDEF
15,                     // QSND_ENABLE dwControlID
h20010005,              // MIXERCONTROL_CONTROLTYPE_STEREOENH dwControlType
1,                      // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                      // cMultipleItems
ENABLE,                 // szShortName
QSOUND,                 // szName
QSOUND,                 // szControlTypeName (my keyword for type of control)
MASTERIN,               // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
1,                      // Bounds.dwMaximum
1,                      // Metrics.cSteps
QSOUNDIN,               // INI file entry
//-----------------------------------------------
//Control16  - REVERB and CHORUS ON at wi
//-----------------------------------------------
#CONTROLDEF
16,                     // REV_ENABLE dwControlID
h20010000,              // MIXERCONTROL_CONTROLTYPE_BOOLEAN dwControlType
1,                      // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                      // cMultipleItems
ENABLE,                 // szShortName
REVERB ON,              // szName
REVEN,                  // szControlTypeName (my keyword for type of control)
MASTERIN,               // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
1,                      // Bounds.dwMaximum
1,                      // Metrics.cSteps
REVENIN,                // INI file entry
//-----------------------------------------------
//Control17  - REVERB depth at wi
//-----------------------------------------------
#CONTROLDEF
17,                     // REV_ENABLE dwControlID
h50030000,              // MIXERCONTROL_CONTROLTYPE_FADER dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
REVERB,                 // szShortName
REVERB DEPTH,                   // szName
REVERB,                 // szControlTypeName (my keyword for type of control)
MASTERIN,               // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
65535,                  // Bounds.dwMaximum
```

```
32,                         // Metrics.cSteps


REVMASTI,                   // INI file entry
REVMASTI,                   // INI file entry
//-----------------------------------------------
//Control18  - CHORUS DEPTH at wi
//-----------------------------------------------
#CONTROLDEF
18,                         // REV_ENABLE dwControlID
h50030000,                  // MIXERCONTROL_CONTROLTYPE_FADER dwControlType
0,                          // fdwControl
0,                          // cMultipleItems
CHORUS,                     // szShortName
CHORUS DEPTH,                         // szName
CHORUS,                     // szControlTypeName (my keyword for type of control)
MASTERIN,                   // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                          // Bounds.dwMinimum
65535,                      // Bounds.dwMaximum
32,                         // Metrics.cSteps
CHOMASTI,
CHOMASTI,
//-----------------------------------------------
//Control19    - MASTER BASS TREBLE on at wi
//-----------------------------------------------
#CONTROLDEF
19,                         // BASS_LINEOUT dwControlID
h20010000,                  // MIXERCONTROL_CONTROLTYPE_BOOLEAN dwControlType
1,                          // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                          // cMultipleItems
ENABLE,                     // szShortName
TONE CONTROL,               // szName
BASSEN,                     // szControlTypeName (my keyword for type of control)
MASTERIN,                   // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                          // Bounds.dwMinimum
1,                          // Bounds.dwMaximum
1,                          // Metrics.cSteps
BASSENIN,                   // INI file entry
//-----------------------------------------------
//Control20    - MASTER TREBLE slider at wi
//-----------------------------------------------
#CONTROLDEF
20,                         // BASS_LINEOUT dwControlID
h50030003,                  // MIXERCONTROL_CONTROLTYPE_TREBLE dwControlType
0,                          // fdwControl
0,                          // cMultipleItems
TREBLE,                     // szShortName
TREBLE CONTROL,             // szName
TREBLE,                     // szControlTypeName (my keyword for type of control)
MASTERIN,                   // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                          // Bounds.dwMinimum
h7FFF,                      // Bounds.dwMaximum
32,                         // Metrics.cSteps
TREMASTI,                   // INI file entry
TREMASTI,                   // INI file entry (only 1 is used)
//-----------------------------------------------
//Control21    - MASTER Bass slider at wi
//-----------------------------------------------
#CONTROLDEF
21,                         // BASS_LINEOUT dwControlID
h50030002,                  // MIXERCONTROL_CONTROLTYPE_BASS dwControlType
```

```
0,                              // fdwControl
0,                              // cMultipleItems
BASS,                           // szShortName
BASS SLIDER,                    // szName
BASS,                           // szControlTypeName (my keyword for type of control)
MASTERIN,                       // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                              // Bounds.dwMinimum
65535,                          // Bounds.dwMaximum
32,                             // Metrics.cSteps
BASMASTI,                       // INI file entry
BASMASTI,                       // INI file entry (only 1 is used)
//----------------------------------------------
//Control22    - Volume between MIDI and DAC
//----------------------------------------------
#CONTROLDEF
22,                             // VOL_OUTMIDI dwControlID
h50030001,                      // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                              // fdwControl
0,                              // cMultipleItems
VOLUME,                         // szShortName
MIDI,                           // szName
VOLUME,                         // szControlTypeName (my keyword for type of control)
MIDI,                           // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                              // Bounds.dwMinimum
hFFFF,                          // Bounds.dwMaximum
64,                             // Metrics.cSteps
MIDILVOL,                       // ini file entry
MIDIRVOL,                       // ini file entry
//----------------------------------------------
//Control23    - Balance between midi and DAC
//----------------------------------------------
#CONTROLDEF
23,                             // BAL_LINE dwControlID
h40020001,                      // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                              // fdwControl
0,                              // cMultipleItems
BALANCE,                        // szShortName
MIDI,                           // szName
BALANCE,                        // szControlTypeName (my keyword for type of control)
MIDI,                           // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                         // Bounds.dwMinimum
32767,                          // Bounds.dwMaximum
16,                             // Metrics.cSteps
BALMIDI,                        // INI file entry
BALMIDI,                        // INI file entry (only 1 is used)
//----------------------------------------------
//Control24    - MUTE of Midiout to DAC
//----------------------------------------------
#CONTROLDEF
24,                             // MUTE_OUTMIDI dwControlID
h20010002,                      // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                              // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                              // cMultipleItems
MUTE,                           // szShortName
MIDI,                           // szName
MUTE,                           // szControlTypeName (my keyword for type of control)
MIDI,                           // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                              // Bounds.dwMinimum
1,                              // Bounds.dwMaximum
0,                              // Metrics.cSteps
```

```
MUTEMIDI,               // ini file entry
//----------------------------------------------
//Control25    - Peak meter at MIDIout to DAC
//----------------------------------------------
#CONTROLDEF
25,                     // VU_MIDIOUT dwControlID
h10020001,              // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
PEAKMETER,              // szShortName
MIDI,                   // szName
PEAKMETER,              // szControlTypeName (my keyword for type of control)
MIDI,                   // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                 // Bounds.dwMinimum
32767,                  // Bounds.dwMaximum
0,                      // Metrics.cSteps
NONE,                   // ini file entry
//----------------------------------------------
//Control26    - Volume between WAVE and DAC
//----------------------------------------------
#CONTROLDEF
26,                     // VOL_OUTWAVE dwControlID
h50030001,              // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
VOLUME,                 // szShortName
WAVE,                   // szName
VOLUME,                 // szControlTypeName (my keyword for type of control)
WAVE,                   // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
hFFFF,                  // Bounds.dwMaximum
64,                     // Metrics.cSteps
WAVELVOL,               // ini file entry
WAVERVOL,               // ini file entry
//----------------------------------------------


//Control27    - Balance between wave and DAC
//----------------------------------------------
#CONTROLDEF
27,                     // BAL_WAVE dwControlID
h40020001,              // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
BALANCE,                // szShortName
WAVE,                   // szName
BALANCE,                // szControlTypeName (my keyword for type of control)
WAVE,                   // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                 // Bounds.dwMinimum
32767,                  // Bounds.dwMaximum
16,                     // Metrics.cSteps
BALWAVE,                // INI file entry
BALWAVE,                // INI file entry (only 1 is used)
//----------------------------------------------
//Control28    - MUTE of WAVE to DAC
//----------------------------------------------
#CONTROLDEF
28,                     // MUTE_OUTWAVE dwControlID
h20010002,              // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                      // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                      // cMultipleItems
```

```
MUTE,                        // szShortName
WAVE,                        // szName
MUTE,                        // szControlTypeName (my keyword for type of control)
WAVE,                        // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                           // Bounds.dwMinimum
1,                           // Bounds.dwMaximum
0,                           // Metrics.cSteps
MUTEWAVE,                    // ini file entry
//----------------------------------------------
//Control29    - Peak meter at WAVEout to DAC
//----------------------------------------------
#CONTROLDEF
29,                          // VU_WAVEOUT dwControlID
h10020001,                   // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                           // fdwControl
0,                           // cMultipleItems
PEAKMETER,                   // szShortName
WAVE,                        // szName
PEAKMETER,                   // szControlTypeName (my keyword for type of control)
WAVE,                        // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                      // Bounds.dwMinimum
32767,                       // Bounds.dwMaximum
0,                           // Metrics.cSteps
NONE,                        // ini file entry
//----------------------------------------------
//Control30    - Volume between SB and DAC
//----------------------------------------------
#CONTROLDEF
30,                          // VOL_OUTSB dwControlID
h50030001,                   // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                           // fdwControl
0,                           // cMultipleItems
VOLUME,                      // szShortName
GAMES,                       // szName
VOLUME,                      // szControlTypeName (my keyword for type of control)
GAMES,                       // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                           // Bounds.dwMinimum
hFFFF,                       // Bounds.dwMaximum
64,                          // Metrics.cSteps
SBLVOL,                      // ini file entry
SBRVOL,                      // ini file entry
//----------------------------------------------
//Control31    - Balance between SB and DAC
//----------------------------------------------
#CONTROLDEF
31,                          // BAL_SB dwControlID
h40020001,                   // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                           // fdwControl
0,                           // cMultipleItems
BALANCE,                     // szShortName
GAMES,                       // szName
BALANCE,                     // szControlTypeName (my keyword for type of control)
GAMES,                       // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                      // Bounds.dwMinimum
32767,                       // Bounds.dwMaximum
16,                          // Metrics.cSteps
BALSB,                       // INI file entry
BALSB,                       // INI file entry (only 1 is used)
//----------------------------------------------
//Control32    - MUTE of SB to DAC
```

```
//-----------------------------------------------
#CONTROLDEF
32,                           // MUTE_OUTSB dwControlID
h20010002,                    // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                            // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                            // cMultipleItems
MUTE,                         // szShortName
GAMES,                        // szName
MUTE,                         // szControlTypeName (my keyword for type of control)
GAMES,                        // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                            // Bounds.dwMinimum
1,                            // Bounds.dwMaximum
0,                            // Metrics.cSteps
MUTESB,                       // ini file entry
//-----------------------------------------------
//Control33    - Peak meter at SBout to DAC
//-----------------------------------------------
#CONTROLDEF
33,                           // VU_SBOUT dwControlID
h10020001,                    // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                            // fdwControl
0,                            // cMultipleItems
PEAKMETER,                    // szShortName
GAMES,                        // szName
PEAKMETER,                    // szControlTypeName (my keyword for type of control)
GAMES,                        // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                       // Bounds.dwMinimum
32767,                        // Bounds.dwMaximum
0,                            // Metrics.cSteps
NONE,                         // ini file entry
//-----------------------------------------------
//Control34    - Volume between CD and DAC
//-----------------------------------------------
#CONTROLDEF
34,                           // VOL_OUTCD dwControlID
h50030001,                    // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                            // fdwControl
0,                            // cMultipleItems
VOLUME,                       // szShortName
CD,                           // szName
VOLUME,                       // szControlTypeName (my keyword for type of control)
AUX,                          // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                            // Bounds.dwMinimum
hFFFF,                        // Bounds.dwMaximum
64,                           // Metrics.cSteps
AUXCDVL,                      // ini file entry
AUXCDVR,                      // ini file entry
//-----------------------------------------------
//Control35    - Balance between CD and DAC
//-----------------------------------------------
#CONTROLDEF
35,                           // BAL_CDOUT dwControlID
h40020001,                    // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                            // fdwControl
0,                            // cMultipleItems
BALANCE,                      // szShortName
CD,                           // szName
BALANCE,                      // szControlTypeName (my keyword for type of control)
AUX,                          // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                       // Bounds.dwMinimum
```

```
32767,                   // Bounds.dwMaximum
16,                      // Metrics.cSteps
AUXCDB,                  // INI file entry
AUXCDB,                  // INI file entry (only 1 is used)
//-----------------------------------------------
//Control36    - MUTE of CD to DAC
//-----------------------------------------------
#CONTROLDEF
36,                      // MUTE_OUTCD dwControlID
h20010002,               // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                       // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                       // cMultipleItems
MUTE,                    // szShortName
CD,                      // szName
MUTE,                    // szControlTypeName (my keyword for type of control)
AUX,                     // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                       // Bounds.dwMinimum
1,                       // Bounds.dwMaximum
0,                       // Metrics.cSteps
AUXCDM,                  // ini file entry
//-----------------------------------------------
//Control37    - Peak meter at CDout to DAC
//-----------------------------------------------
#CONTROLDEF
37,                      // VU_CDOUT dwControlID
h10020001,               // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                       // fdwControl
0,                       // cMultipleItems
PEAKMETER,               // szShortName
CD,                      // szName
PEAKMETER,               // szControlTypeName (my keyword for type of control)
AUX,                     // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                  // Bounds.dwMinimum
32767,                   // Bounds.dwMaximum
0,                       // Metrics.cSteps
NONE,                    // ini file entry
//-----------------------------------------------
//Control38   - Switch on or off of CD
//-----------------------------------------------
#CONTROLDEF
38,                      // CDOUT_ENABLE dwControlID
h20010000,               // MIXERCONTROL_CONTROLTYPE_BOOLEAN dwControlType
1,                       // fdwControl uniform
0,                       // cMultipleItems
ENABLE,                  // szShortName
CD,                      // szName
ENABLE,                  // szControlTypeName (my keyword for type of control)
AUX,                     // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                       // Bounds.dwMinimum
1,                       // Bounds.dwMaximum
0,                       // Metrics.cSteps
AUXCDE,                  // INI file entry
//-----------------------------------------------
//Control39    - Volume between LINE and DAC
//-----------------------------------------------
#CONTROLDEF
39,                      // VOL_OUTLINE dwControlID
h50030001,               // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                       // fdwControl
0,                       // cMultipleItems
VOLUME,                  // szShortName
```

```
LINE,                       // szName
VOLUME,                     // szControlTypeName (my keyword for type of control)
AUX,                        // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                          // Bounds.dwMinimum
hFFFF,                      // Bounds.dwMaximum
64,                         // Metrics.cSteps
AUXLVL,                     // ini file entry
AUXLVR,                     // ini file entry
//-----------------------------------------------
//Control40    - Balance between LINE and DAC
//-----------------------------------------------
#CONTROLDEF
40,                         // BAL_CDOUT dwControlID
h40020001,                  // MIXERCONTROL_CONTROLTYPE_PAN dwContrdType
0,                          // fdwControl
0,                          // cMultipleItems
BALANCE,                    // szShortName
LINE,                       // szName
BALANCE,                    // szControlTypeName (my keyword for type of control)
AUX,                        // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                     // Bounds.dwMinimum
32767,                      // Bounds.dwMaximum
16,                         // Metrics.cSteps
AUXLB,                      // INI file entry
AUXLB,                      // INI file entry (only 1 is used)
//-----------------------------------------------
//Control41    - MUTE of LINE to DAC
//-----------------------------------------------
#CONTROLDEF
41,                         // MUTE_OUTLINE dwControlID
h20010002,                  // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                          // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                          // cMultipleItems
MUTE,                       // szShortName
LINE,                       // szName
MUTE,                       // szControlTypeName (my keyword for type of control)
AUX,                        // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                          // Bounds.dwMinimum
1,                          // Bounds.dwMaximum
0,                          // Metrics.cSteps
AUXLM,                                // ini file entry
//-----------------------------------------------
//Control42    - Peak meter at LINEout to DAC
//-----------------------------------------------
#CONTROLDEF
42,                         // VU_LINEOUT dwControlID
h10020001,                  // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                          // fdwControl
0,                          // cMultipleItems
PEAKMETER,                  // szShortName
LINE,                       // szName
PEAKMETER,                  // szControlTypeName (my keyword for type of control)
AUX,                        // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                     // Bounds.dwMinimum
32767,                      // Bounds.dwMaximum
0,                          // Metrics.cSteps
NONE,                       // ini file entry
//-----------------------------------------------
//Control43   - Switch on or off of LINE to DAC
//-----------------------------------------------
```

```
#CONTROLDEF
43,                     // CDOUT_ENABLE dwControlID
h20010000,              // MIXERCONTROL_CONTROLTYPE_BOOLEAN dwControlType
1,                      // fdwControl uniform
0,                      // cMultipleItems
ENABLE,                 // szShortName
LINE,                   // szName
ENABLE,                 // szControlTypeName (my keyword for type of control)
AUX,                    // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
1,                      // Bounds.dwMaximum
0,                      // Metrics.cSteps
AUXLE,                  // INI file entry
//-----------------------------------------------
//Control44    - Volume between MIC and DAC
//-----------------------------------------------
#CONTROLDEF
44,                     // VOL_OUTMIC dwControlID
h50030001,              // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
VOLUME,                 // szShortName
MIC,                    // szName
VOLUME,                 // szControlTypeName (my keyword for type of control)
AUX,                    // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
hFFFF,                  // Bounds.dwMaximum
64,                     // Metrics.cSteps
AUXMVL,                 // ini file entry
AUXMVR,                 // ini file entry
//-----------------------------------------------
//Control45    - Balance between MIC and DAC
//-----------------------------------------------
#CONTROLDEF
45,                     // BAL_CDOUT dwControlID
h40020001,              // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
BALANCE,                // szShortName
MIC,                    // szName
BALANCE,                // szControlTypeName (my keyword for type of control)
AUX,                    // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                 // Bounds.dwMinimum
32767,                  // Bounds.dwMaximum
16,                     // Metrics.cSteps
AUXMB,                  // INI file entry
AUXMB,                  // INI file entry
//-----------------------------------------------
//Control46    - MUTE of MIC to DAC
//-----------------------------------------------
#CONTROLDEF
46,                     // MUTE_OUTMIC dwControlID
h20010002,              // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                      // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                      // cMultipleItems
MUTE,                   // szShortName
MIC,                    // szName
MUTE,                   // szControlTypeName (my keyword for type of control)
AUX,                    // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
```

```
1,                          // Bounds.dwMaximum
0,                          // Metrics.cSteps
AUXMM,                      // ini file entry
//----------------------------------------------
//Control47    - Peak meter at MICout to DAC
//----------------------------------------------
#CONTROLDEF
47,                         // VU_MICOUT dwControlID
h10020001,                  // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                          // fdwControl
0,                          // cMultipleItems
PEAKMETER,                  // szShortName
MIC,                        // szName
PEAKMETER,                  // szControlTypeName (my keyword for type of control)
AUX,                        // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                     // Bounds.dwMinimum
32767,                      // Bounds.dwMaximum
0,                          // Metrics.cSteps
NONE,                       // ini file entry
//----------------------------------------------
//Control48   - Switch on or off of MIC to DAC
//----------------------------------------------
#CONTROLDEF
48,                         // CDOUT_ENABLE dwControlID
h20010000,                  // MIXERCONTROL_CONTROLTYPE_BOOLEAN dwControlType
1,                          // fdwControl uniform
0,                          // cMultipleItems
ENABLE,                     // szShortName
MIC,                        // szName
ENABLE,                     // szControlTypeName (my keyword for type of control)
AUX,                        // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                          // Bounds.dwMinimum
1,                          // Bounds.dwMaximum
0,                          // Metrics.cSteps
AUXME,                                 // INI file entry
//----------------------------------------------
//  NOW STARTS THE SAME STUFF FOR RECORD IN!!!
//
//Control49    - Volume between MIDI and WAVEIN
//----------------------------------------------
#CONTROLDEF
49,                         // VOL_INMIDI dwControlID
h50030001,                  // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                          // fdwControl
0,                          // cMultipleItems
VOLUME,                     // szShortName
MIDIOUTIN,                  // szName
VOLUME,                     // szControlTypeName (my keyword for typeof control)
MIDIOUTIN,                  // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                          // Bounds.dwMinimum
hFFFF,                      // Bounds.dwMaximum
64,                         // Metrics.cSteps
MIDILVOLIN,                 // ini file entry
MIDIRVOLIN,                 // ini file entry
//----------------------------------------------
//Control50    - Balance between midi and WAVEIN
//----------------------------------------------
#CONTROLDEF
50,                         // BAL_LINE dwControlID
h40020001,                  // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                          // fdwControl
```

```
0,                          // cMultipleItems
BALANCE,                    // szShortName
MIDIOUTIN,                  // szName
BALANCE,                    // szControlTypeName (my keyword for type of control)
MIDIOUTIN,                  // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                     // Bounds.dwMinimum
32767,                      // Bounds.dwMaximum
16,                         // Metrics.cSteps
BALMIDIIN,                  // INI file entry
BALMIDIIN,                  // INI file entry (only 1 is used)
//-----------------------------------------------
//Control51    - MUTE of Midiout to WAVEIN
//-----------------------------------------------
#CONTROLDEF
51,                         // MUTE_INMIDI dwControlID
h20010002,                  // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                          // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                          // cMultipleItems
MUTE,                       // szShortName
MIDIOUTIN,                  // szName
MUTE,                       // szControlTypeName (my keyword for type of control)
MIDIOUTIN,                  // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                          // Bounds.dwMinimum
1,                          // Bounds.dwMaximum
1,                          // Metrics.cSteps
MUTEMIDIIN,                 // ini file entry
//-----------------------------------------------
//Control52    - Peak meter at MIDIout to WAVEIN
//-----------------------------------------------
#CONTROLDEF
52,                         // VU_MIDIOUTIN dwControlID
h10020001,                  // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                          // fdwControl
0,                          // cMultipleItems
PEAKMETER,                  // szShortName
MIDIOUTIN,                  // szName
PEAKMETER,                  // szControlTypeName (my keyword for type of control)
MIDIOUTIN,                  // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                     // Bounds.dwMinimum
32767,                      // Bounds.dwMaximum
0,                          // Metrics.cSteps
NONE,                       // ini file entry
//-----------------------------------------------
//Control53    - Volume between WAVE and WAVEIN
//-----------------------------------------------
#CONTROLDEF
53,                         // VOL_INWAVE dwControlID
h50030001,                  // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                          // fdwControl
0,                          // cMultipleItems
VOLUME,                     // szShortName
WAVEOUTIN,                  // szName
VOLUME,                     // szControlTypeName (my keyword for type of control)
WAVEOUTIN,                  // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                          // Bounds.dwMinimum
hFFFF,                      // Bounds.dwMaximum
64,                         // Metrics.cSteps
WAVELVOLIN,                 // ini file entry
WAVERVOLIN,                 // ini file entry
//-----------------------------------------------
```

```
//Control54    - Balance between wave and WAVEIN
//-----------------------------------------------
#CONTROLDEF
54,                       // BAL_WAVE dwControlID
h40020001,                // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                        // fdwControl
0,                        // cMultipleItems
BALANCE,                  // szShortName
WAVEOUTIN,                // szName
BALANCE,                  // szControlTypeName (my keyword for type of control)
WAVEOUTIN,                // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                   // Bounds.dwMinimum
32767,                    // Bounds.dwMaximum
16,                       // Metrics.cSteps
BALWAVEIN,                // INI file entry
BALWAVEIN,                // INI file entry (only 1 is used)
//-----------------------------------------------
//Control55    - MUTE of WAVE to WAVEIN
//-----------------------------------------------
#CONTROLDEF
55,                       // MUTE_INWAVE dwControlID
h20010002,                // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                        // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                        // cMultipleItems
MUTE,                     // szShortName
WAVEOUTIN,                // szName
MUTE,                     // szControlTypeName (my keyword for type of control)
WAVEOUTIN,                // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                        // Bounds.dwMinimum
1,                        // Bounds.dwMaximum
0,                        // Metrics.cSteps
MUTEWAVEIN,               // ini file entry
//-----------------------------------------------
//Control56    - Peak meter at WAVEout to WAVEIN
//-----------------------------------------------
#CONTROLDEF
56,                       // VU_WAVEIN dwControlID
h10020001,                // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                        // fdwControl
0,                        // cMultipleItems
PEAKMETER,                // szShortName
WAVEOUTIN,                // szName
PEAKMETER,                // szControlTypeName (my keyword for type of control)
WAVEOUTIN,                // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                   // Bounds.dwMinimum
32767,                    // Bounds.dwMaximum
0,                        // Metrics.cSteps
NONE,                     // ini file entry
//-----------------------------------------------
//Control57    - Volume between SB and wavein
//-----------------------------------------------
#CONTROLDEF
57,                       // VOL_OUTSB dwControlID
h50030001,                // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                        // fdwControl
0,                        // cMultipleItems
VOLUME,                   // szShortName
SB,                       // szName
VOLUME,                   // szControlTypeName (my keyword for type of control)
GAMESOUTIN,               // szControlLineName (my keyword for line this control applies to)
#DBLWORD
```

```
0,                        // Bounds.dwMinimum
hFFFF,                    // Bounds.dwMaximum
64,                       // Metrics.cSteps
SBLVOLIN,                 // ini file entry
SBRVOLIN,                 // ini file entry
//----------------------------------------------
//Control58    - Balance between SB and wavein
//----------------------------------------------
#CONTROLDEF
58,                       // BAL_SB dwControlID
h40020001,                // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                        // fdwControl
0,                        // cMultipleItems
BALANCE,                  // szShortName
SB,                       // szName
BALANCE,                  // szControlTypeName (my keyword for type of control)
GAMESOUTIN,               // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                   // Bounds.dwMinimum
32767,                    // Bounds.dwMaximum
16,                       // Metrics.cSteps
BALSBIN,                  // INI file entry
BALSBIN,                  // INI file entry (only 1 is used)
//----------------------------------------------
//Control59    - MUTE of SB to wavein
//----------------------------------------------
#CONTROLDEF
59,                       // MUTE_OUTSB dwControlID
h20010002,                // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                        // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                        // cMultipleItems
MUTE,                     // szShortName
SB,                       // szName
MUTE,                     // szControlTypeName (my keyword for type of control)
GAMESOUTIN,               // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                        // Bounds.dwMinimum
1,                        // Bounds.dwMaximum
1,                        // Metrics.cSteps
MUTESBIN,                 // ini file entry
//----------------------------------------------
//Control60    - Peak meter at SBoutin to wavein
//----------------------------------------------
#CONTROLDEF
60,                       // VU_SBOUT dwControlID
h10020001,                // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                        // fdwControl
0,                        // cMultipleItems
PEAKMETER,                // szShortName
SB,                       // szName
PEAKMETER,                // szControlTypeName (my keyword for type of control)
GAMESOUTIN,               // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                   // Bounds.dwMinimum
32767,                    // Bounds.dwMaximum


0,                        // Metrics.cSteps
NONE,                     // ini file entry
//----------------------------------------------
//Control61    - Volume between CD and WAVEIN
//             This is same as waveout but just let the
//             user think that they are different
//----------------------------------------------
```

```
#CONTROLDEF
61,                          // VOL_INCD dwControlID
h50030001,                   // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                           // fdwControl
0,                           // cMultipleItems
VOLUME,                      // szShortName
CD,                          // szName
VOLUME,                      // szControlTypeName (my keyword for type of control)
WAVEIN,                      // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                           // Bounds.dwMinimum
hFFFF,                       // Bounds.dwMaximum
64,                          // Metrics.cSteps
RECCDVL,                     // ini file entry
RECCDVR,                     // ini file entry
//-----------------------------------------------
//Control62    - Balance between CD and WAVEIN
//-----------------------------------------------
#CONTROLDEF
62,                          // BAL_CDOUT dwControlID
h40020001,                   // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                           // fdwControl
0,                           // cMultipleItems
BALANCE,                     // szShortName
CD,                          // szName
BALANCE,                     // szControlTypeName (my keyword for type of control)
WAVEIN,                      // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                      // Bounds.dwMinimum
32767,                       // Bounds.dwMaximum
16,                          // Metrics.cSteps
RECCDB,                      // INI file entry
RECCDB,                      // INI file entry (only 1 is used)
//-----------------------------------------------
//Control63    - MUTE of CD to WAVEIN
//-----------------------------------------------
#CONTROLDEF
63,                          // MUTE_OUTCD dwControlID
h20010002,                   // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                           // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                           // cMultipleItems
MUTE,                        // szShortName
CD,                          // szName
MUTE,                        // szControlTypeName (my keyword for type of control)
WAVEIN,                      // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                           // Bounds.dwMinimum


1,                           // Bounds.dwMaximum
1,                           // Metrics.cSteps
RECCDM,                      // ini file entry
//-----------------------------------------------
//Control64    - Peak meter at CDout to WAVEIN
//-----------------------------------------------
#CONTROLDEF
64,                          // VU_CDOUT dwControlID
h10020001,                   // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                           // fdwControl
0,                           // cMultipleItems
PEAKMETER,                   // szShortName
CD,                          // szName
PEAKMETER,                   // szControlTypeName (my keyword for type of control)
WAVEIN,                      // szControlLineName (my keyword for line this control applies to)
```

```
#LNGWORD
-32768,                 // Bounds.dwMinimum
32767,                  // Bounds.dwMaximum
0,                      // Metrics.cSteps
NONE,                   // ini file entry
//-----------------------------------------------
//Control65   - Switch on or off of CD
//-----------------------------------------------
#CONTROLDEF
65,                     // CDOUT_ENABLE dwControlID
h20010000,              // MIXERCONTROL_CONTROLTYPE_BOOLEAN dwControlType
1,                      // fdwControl uniform
0,                      // cMultipleItems
ENABLE,                 // szShortName
CD,                     // szName
ENABLE,                 // szControlTypeName (my keyword for type of control)
WAVEIN,                 // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
1,                      // Bounds.dwMaximum
1,                      // Metrics.cSteps
RECCDE,                 // INI file entry
//-----------------------------------------------
//Control66    - Volume between LINE and WAVEIN
//-----------------------------------------------
#CONTROLDEF
66,                     // VOL_OUTLINE dwControlID
h50030001,              // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
VOLUME,                 // szShortName
LINE,                   // szName
VOLUME,                 // szControlTypeName (my keyword for type of control)
WAVEIN,                 // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
hFFFF,                  // Bounds.dwMaximum
64,                     // Metrics.cSteps
RECLVL,                 // ini file entry
RECLVR,                 // ini file entry
//-----------------------------------------------
//Control67    - Balance between LINE and WAVEIN
//-----------------------------------------------
#CONTROLDEF
67,                     // BAL_CDOUT dwControlID
h40020001,              // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                      // fdwControl
0,                      // cMultipleItems
BALANCE,                // szShortName
LINE,                   // szName
BALANCE,                // szControlTypeName (my keyword for type of control)
WAVEIN,                 // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                 // Bounds.dwMinimum
32767,                  // Bounds.dwMaximum
16,                     // Metrics.cSteps
RECLB,                  // INI file entry
RECLB,                  // INI file entry (only 1 is used)
//-----------------------------------------------
//Control68    - MUTE of LINE to WAVEIN
//-----------------------------------------------
#CONTROLDEF
68,                     // MUTE_OUTLINE dwControlID
h20010002,              // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
```

```
1,                            // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                            // cMultipleItems
MUTE,                         // szShortName
LINE,                         // szName
MUTE,                         // szControlTypeName (my keyword for type of control)
WAVEIN,                       // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                            // Bounds.dwMinimum
1,                            // Bounds.dwMaximum
1,                            // Metrics.cSteps
RECLM,                        // ini file entry
//-----------------------------------------------
//Control69    - Peak meter at LINEout to WAVEIN
//-----------------------------------------------
#CONTROLDEF
69,                           // VU_LINEOUT dwControlID
h10020001,                    // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                            // fdwControl
0,                            // cMultipleItems
PEAKMETER,                    // szShortName
LINE,                         // szName
PEAKMETER,                    // szControlTypeName (my keyword for type of control)
WAVEIN,                       // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                       // Bounds.dwMinimum
32767,                        // Bounds.dwMaximum
0,                            // Metrics.cSteps
NONE,                         // ini file entry
//-----------------------------------------------


//Control70   - Switch on or off of LINE to WAVEIN
//-----------------------------------------------
#CONTROLDEF
70,                           // CDOUT_ENABLE dwControlID
h20010000,                    // MIXERCONTROL_CONTROLTYPE_BOOLEAN dwControlType
1,                            // fdwControl uniform
0,                            // cMultipleItems
ENABLE,                       // szShortName
LINE,                         // szName
ENABLE,                       // szControlTypeName (my keyword for type of control)
WAVEIN,                       // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                            // Bounds.dwMinimum
1,                            // Bounds.dwMaximum
1,                            // Metrics.cSteps
RECLE,                        // INI file entry
//-----------------------------------------------
//Control71    - Volume between MIC and WAVEIN
//-----------------------------------------------
#CONTROLDEF
71,                           // VOL_OUTMIC dwControlID
h50030001,                    // MIXERCONTROL_CONTROLTYPE_VOLUME dwControlType
0,                            // fdwControl
0,                            // cMultipleItems
VOLUME,                       // szShortName
MIC,                          // szName
VOLUME,                       // szControlTypeName (my keyword for type of control)
WAVEIN,                       // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                            // Bounds.dwMinimum
hFFFF,                        // Bounds.dwMaximum
64,                           // Metrics.cSteps
RECMVL,                       // ini file entry
```

```
RECMVR,                    // ini file entry
//----------------------------------------------
//Control72    - Balance between MIC and WAVEIN
//----------------------------------------------
#CONTROLDEF
72,                        // BAL_CDOUT dwControlID
h40020001,                 // MIXERCONTROL_CONTROLTYPE_PAN dwControlType
0,                         // fdwControl
0,                         // cMultipleItems
BALANCE,                   // szShortName
MIC,                       // szName
BALANCE,                   // szControlTypeName (my keyword for type of control)
WAVEIN,                    // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                    // Bounds.dwMinimum
32767,                     // Bounds.dwMaximum
16,                        // Metrics.cSteps
RECMB,                     // INI file entry
RECMB,                     // INI file entry (only 1 is used)
//----------------------------------------------
//Control73    - MUTE of MIC to WAVEIN


//----------------------------------------------
#CONTROLDEF
73,                        // MUTE_OUTMIC dwControlID
h20010002,                 // MIXERCONTROL_CONTROLTYPE_MUTE dwControlType
1,                         // fdwControl MIXERCONTROL_CONTROLF_UNIFORM
0,                         // cMultipleItems
MUTE,                      // szShortName
MIC,                       // szName
MUTE,                      // szControlTypeName (my keyword for type of control)
WAVEIN,                    // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                         // Bounds.dwMinimum
1,                         // Bounds.dwMaximum
1,                         // Metrics.cSteps
RECMM,                     // ini file entry
//----------------------------------------------
//Control74    - Peak meter at MICout to WAVEIN
//----------------------------------------------
#CONTROLDEF
74,                        // VU_MICOUT dwControlID
h10020001,                 // MIXERCONTROL_CONTROLTYPE_PEAKMETER dwControlType
0,                         // fdwControl
0,                         // cMultipleItems
PEAKMETER,                 // szShortName
MIC,                       // szName
PEAKMETER,                 // szControlTypeName (my keyword for type of control)
WAVEIN,                    // szControlLineName (my keyword for line this control applies to)
#LNGWORD
-32768,                    // Bounds.dwMinimum
32767,                     // Bounds.dwMaximum
0,                         // Metrics.cSteps
NONE,                      // ini file entry
//----------------------------------------------
//Control75   - Switch on or off of MIC to WAVEIN
//----------------------------------------------
#CONTROLDEF
75,                        // CDOUT_ENABLE dwControlID
h20010000,                 // MIXERCONTROL_CONTROLTYPE_BOOLEAN dwControlType
1,                         // fdwControl uniform
0,                         // cMultipleItems
ENABLE,                    // szShortName
```

```
MIC,                    // szName
ENABLE,                 // szControlTypeName (my keyword for type of control)
WAVEIN,                 // szControlLineName (my keyword for line this control applies to)
#DBLWORD
0,                      // Bounds.dwMinimum
1,                      // Bounds.dwMaximum
1,                      // Metrics.cSteps
RECME,                  // INI file entry
#ENDFILE                // indicates end of text file (put something here!)
```