Dialogic®

Dialogic® Multimedia API

Programming Guide and Library Reference

August 2011

05-2454-009

Publication Date: August 2011

Document Number: 05-2454-009

Contents

Revision History 7

About This Publication 15

Purpose 15
Applicability 15
Intended Audience 15
How to Use This Publication 16
Related Information 16

1 Product Description 17

1.1 Overview of Video Technology 17
1.2 IP Media Server 21
1.3 Multimedia Capabilities 21
1.4 File Formats 22
1.5 SIP Call Control Using Global Call 23
1.6 SDP Functionality 24
1.7 Features 24

2 Feature Support by Platform 28

2.1 High-Level Feature Support by Platform 28
2.2 Dialogic® Multimedia API Function Support by Platform 29
2.3 Dialogic® Multimedia API Audio Formats by Platform 31

3 Event Handling 33

4 Error Handling 34

5 Application Development Guidelines 35

5.1 Developing Multimedia Applications 35
5.2 Building Blocks for Multimedia Applications 38
5.3 Call Flow of Video Mail 38
5.4 Requesting I-Frame Using SIP INFO 42
5.5 Making Connections Using Virtual CT Bus and New Packet Bus 44
5.6 Enabling Digit Detection Using DX 48
5.7 **Enabling and Using Transcoding** **49**
5.8 Play a Still Image 58
5.9 Capture a Still Image 59
5.10 3GP File Format Direct Playback 59

6 Building Applications 63

6.1 Compiling and Linking 63

7 Function Summary by Category 65

7.1 Dialogic® Multimedia API Header File 65
7.2 Device Management Functions 65
7.3 Configuration Functions 66
7.4 Input/Output Functions 66

7.5 Media Streaming Functions 67
7.6 DVR Control Functions 67
7.7 Event Information Functions 68
7.8 Error Processing Functions 68

8 Function Information 69

8.1 Function Syntax Conventions 69
mm_Capture( ) – capture a still image from video stream 70
mm_Close( ) – close a previously opened multimedia device 73
mm_DisableEvents( ) – disable optional notification events 75
mm_EnableEvents( ) – enable optional notification events 77
mm_ErrorInfo( ) – retrieve current error information for a multimedia function 79
mm_GetDuration( ) – get the duration of current presentation 81
mm_GetElapsedTime( ) – get current presentation time 83
mm_GetMetaEvent( ) – get current SRL event information 85
mm_GetParm( ) – get the current configuration parameters 87
mm_Open( ) – open a multimedia device 89
mm_Pause( ) – pause the current presentation 92
mm_Play( ) – play a media object 95
mm_Record( ) – record a synchronized media object 101
mm_Reset( ) – reset an open multimedia device 107
mm_ResultInfo( ) – retrieve current event information for multimedia events 110
mm_Resume( ) – resume current presentation 112
mm_Seek( ) – place current presentation at the specified position 114
mm_SetParm( ) – set the configuration parameters 116
mm_Stop( ) – stop the device operations 118
mm_StreamClose( ) – close a media stream 122
mm_StreamGetStat( ) – get media stream statistics 127
mm_StreamOpen( ) – open a media stream 128
mm_StreamRead( ) – read from a media stream 130
mm_StreamReset( ) – reset a media stream 132
mm_StreamSetWaterMark( ) – set high or low watermark 133
mm_StreamWrite( ) – write to media stream 135

9 Events 137

9.1 Overview of Dialogic® Multimedia API Events 137
9.2 Dialogic® Multimedia API Event Types 138
9.3 Dialogic® Multimedia API Event Types by Function Type 139
9.4 Dialogic® Multimedia API Events 139

10 Data Structure Types 145

10.1 Overview of Dialogic® Multimedia API Data Structures 145
10.2 Data Structures for Function I/O 146
10.3 Data Structures for Analyzing Event Information 147
10.4 Play/Record Data Structure Levels 149
10.5 Other Data Structure Levels (_DETAILS) 149
10.6 Data Structures By Function 150

11 Data Structures 153

MM_AUDIO_CODEC – audio codec specification 156
MM_AUDIO_CODEC_OPTION_LIST – list of codec options 158
MM_CAPTURE_CMPLT_DETAILS – event data for capture still image 162
MM_ERROR_RESULT – error event information 163
MM_EVENTS – information for optional notification event functions 164
MM_GET_PARM – information for get parameter function 165
MM_GET_PARM_RESULT – retrieved parameter event information 166
MM_GETDURATION_CMPLT_DETAILS – request reply message details 167
MM_GETDURATION_CMPLT – get event data payload 168
MM_GETDURATION_INFO – get duration information 169
MM_GETELAPSEDTIME_CMPLT – event data payload count 170
MM_GETELAPSEDTIME_CMPLT_DETAILS – request reply message details 171
MM_GETELAPSEDTIME_INFO – get stream type details 172
MM_IMAGE_FORMAT – specifies data contents of image format 173
MM_INFO – error or result information 174
MM_MEDIA_ACCESS_MEMORY – access the media buffer 175
MM_MEDIA_ACCESS_STREAM – access media stream 176
MM_MEDIA_AUDIO – audio media item specification 177
MM_MEDIA_IMAGE – still image for play or capture 179
MM_MEDIA_ITEM – media item information 181
MM_MEDIA_ITEM_LIST – media item list information 182
MM_MEDIA_TERM – media termination information 183
MM_MEDIA_VIDEO – video media item specification 184
MM_METAEVENT – event descriptor for a metaevent 186
MM_PAUSE_INFO – pause request details 188
MM_PAUSE_PLAY_CMPLT – event data count 189
MM_PAUSE_PLAY_CMPLT_DETAILS – stream type details 190
MM_PLAY_RECORD_CMPLT – play/record completion event information 191
MM_PLAY_RECORD_CMPLT_DETAILS – play/record completion details 192
MM_PLAY_RECORD_INFO – information for play and record functions 195
MM_PLAY_RECORD_LIST – list of items to play or record 196
MM_RECORD_CMPLT_DATA_BLOCK – specifies data block 198
MM_RECORD_CMPLT_INFO_DATA_BLOCKS – contains data block info 200
MM_RESUME_INFO – resume play infomation 201
MM_RESUME_PLAY_CMPLT – resume/play completion event information 202
MM_RESUME_PLAY_CMPLT_DETAILS – pause request reply message details 203
MM_RET_CODE – error return code information 204
MM_RUNTIME_CONTROL – set runtime control 206
MM_SEEK_CMPLT – count of event data payload 208
MM_SEEK_CMPLT_DETAILS – seek request reply details 209
MM_SET_PARM – information for set parameter function 210
MM_STOP – information for stop device operations function 211
MM_SEEK_INFO – seek operation position 213
MM_STOP_ACK – stop ACK event information 215
MM_STOP_ACK_DETAILS – stop ACK detail information 216
MM_STOP_DETAILS – detailed stop request information 217

MM_STREAM_OPEN_INFO – retrieve information on open stream 218
MM_STREAM_STAT – retrieve stream statistics 219
MM_STREAM_WATERMARK_INFO – retrieve stream watermark info 221
MM_VIDEO_CODEC – characteristics of video coder 222
MM_VIDEO_RECORD_STARTED – I-Frame detection information 227
MM_YUV – content of a YUV bitmap 228

12 Error Codes 229

12.1 Overview of Dialogic® Multimedia API Errors 229
12.2 Dialogic® Multimedia API Function Error Codes 231
12.3 Multimedia API Event Information Error Return Codes 232
12.4 Media Streaming Event Information Error Return Codes 234
12.5 Terminating and Non-Terminating Play/Record Errors 235

About This Publication

The following topics provide information about this publication:

- Purpose
- Applicability
- Intended Audience
- How to Use This Publication
- Related Information

Purpose

This document provides information about video technology and multimedia library features, as well as guidelines for those choosing to develop applications using the Dialogic® Multimedia API.

This document also contains reference information for the functions, parameters, data structures, values, events, and error codes in the Dialogic® Multimedia API. The API provides the ability to record and play back digitized multimedia (audio and video) to support video services in application programs.

Applicability

This document is published for the Dialogic® Host Media Processing (HMP) Software Release.

This document may also be applicable to other software releases (including service updates) on Linux or Windows® operating systems. Check the Release Guide for your software release to determine whether this document is supported.

Intended Audience

This document is intended for software developers who will access the Dialogic® Multimedia API Library. This may included any of the following:

- Distributors
- System Integrators
- Toolkit Developers
- Independent Software Vendors (ISVs)
- Value Added Resellers (VARs)
- Original Equipment Manufacturers (OEMs)

How to Use This Publication

Refer to this document after you have installed the hardware and the system software which includes the Dialogic® Multimedia API. This publication assumes that you are familiar with the Linux or Windows® operating system and the C programming language.

The information in this publication is organized as follows:

- Chapter 1, "Product Description" – provides an overview of video technology and the multimedia features of the Dialogic® Host Media Processing (HMP) software.
- Chapter 2, "Feature Support by Platform" provides a list of high-level feature support by Dialogic® platforms (software releases).
- Chapter 3, "Event Handling" – provides information about Multimedia API events.
- Chapter 4, "Error Handling" – provides information about Multimedia API errors.
- Chapter 5, "Application Development Guidelines" – provides some guidelines for those choosing to develop multimedia applications.
- Chapter 6, "Building Applications" – provides general information about building applications using the Multimedia API library.
- Chapter 7, "Function Summary by Category" introduces the categories of functions and provides a brief description of each function.
- Chapter 8, "Function Information" provides an alphabetical reference to the functions in the library.
- Chapter 9, "Events" describes the events that are generated by the Multimedia API functions.
- Chapter 10, "Data Structure Types" describes the types of data structures supported by the Multimedia API, including the basic categories, specific types, naming conventions, purpose, hierarchy, and association with specific functions.
- Chapter 11, "Data Structures" provides an alphabetical reference to the Multimedia API data structures, along with their fields and valid values.
- Chapter 12, "Error Codes" describes the errors that can be returned by the Multimedia API.

Related Information

See the following additional information:

- *http://www.dialogic.com/manuals/* (for Dialogic® product documentation)
- *http://www.dialogic.com/support/* (for Dialogic technical support)
- *http://www.dialogic.com/* (for Dialogic® product information)

Revision History

This revision history summarizes the changes made in each published version of this document.

Document No.	Publication Date	Description of Revisions
05-2454-009	August 2011	Removed references to Dialogic® Multimedia Kit Software for PCIe (MMK) and Dialogic® Multimedia Platform for AdvancedTCA (MMP) as these products are no longer supported. Product Description: Added H.264 to the first sentence of the first paragraph in Video Standards. and added H.264 to last sentence of first paragraph in Intraframe (I-Frame). Feature Support by Platform: Updated all of the support tables in this chapter. Application Development Guidelines: Added a note about image file size to the end of Section 5.8, “Play a Still Image”. (IPY00081664) and updated Section 5.10, “3GP File Format Direct Playback” due to feature enhancements. mm_Play( ): Added an item to the Cautions header. (IPY00081664) MM_PLAY_RECORD_CMPLT_DETAILS: In the Status field, changed the description of EMM_STATUS_RCRD_V_MISSING_MPEG4_VISUALCONFIG_ERROR. MM_RET_CODE: Remove the list of EMMRC_xxx error codes from the unRetCode field. (IPY00081664) MM_VIDEO_CODEC: Replaced the reference page with new information because of new features. Error Codes: Added new error codes to the Multimedia API Event Information list.
05-2454-008	October 2009	Product Description: Added 3GP File Format Direct Playback to the Features section. Feature Support by Platform: Updated all of the support tables in this chapter. Application Development Guidelines: Added a new section about 3GP File Format Direct Playback. mm_DisableEvents( ): Added a reference to the new table of Optional Notification Events. mm_EnableEvents( ): Added a reference to the new table of Optional Notification Events. mm_StreamClose( ): Updated example code. mm_StreamRead( ): Added MMEV_RECORD_AUDIO_EOD and MMEV_RECORD_VIDEO_EOD to the Events section. Events: Added two events, MMEV_RECORD_AUDIO_EOD and MMEV_RECORD_VIDEO_EOD, added a table of optional notification events in the Dialogic® Multimedia API Event Types section, and changed text due to the addition of the new events. MM_EVENTS: Added a reference to the new table of Optional Notification Events. MM_MEDIA_AUDIO: Added the EMM_AUD_FILEFORMAT_3GP value to the eFileFormat field. MM_MEDIA_VIDEO: Added the EMM_FILE_FORMAT_3GP value to the eFileFormat field. Error Codes: Added new error codes for the 3GP feature.

Document No.	Publication Date	Description of Revisions
05-2454-007	November 2008	About This Publication chapter: Updated the Applicability section. Product Description: Updated H.264 support bullets for audio and video. Multimedia Capabilities section: Updated the links to the Multimedia File Conversion Tools utilities. Application Development Guidelines: Updated the Play a Still Image and Capture a Still Image sections. Feature Support by Platform chapter: Added the Dialogic® Multimedia Kit for PCIe to the Dialogic® Multimedia API Function Support Platform table. Function Information chapter: For supported functions, added the Dialogic® Multimedia Kit for PCIe to the Platform line. mm_Capture( ): Added three notes on the function page and updated code example. mm_Close( ): Updated the pOpenInfo parameter definition and Cautions section. mm_GetDuration( ): Added function for DVR Control support. mm_GetElapsedTime( ): Added function for DVR Control support. mm_Open( ): Updated the pOpenInfo parameter definition. mm_Pause( ): Added function for DVR Control support. mm_Play( ): Updated code examples. Added a fourth bullet to the Cautions section. mm_Record( ): Updates the entire function reference information page. mm_Reset( ): Updated the entire function reference information page. mm_Resume( ): Added function for DVR Control support. mm_Seek( ): Added function for DVR Control support. mm_StreamOpen( ): Added that the pUserInfo parameter is reserved for future use. Added a caution statement. mm_StreamRead( ): Updated the entire function reference information page. mm_StreamReset( ): Updated the entire function reference information page. mm_StreamWrite( ): Updated the entire function reference information page. Events: Updated event definitions and added events for DVR Control functions.

Document No.	Publication Date	Description of Revisions
05-2454-007 (continued)	November 2008	Data Structure Types: Added MM_MEDIA_IMAGE to hierarchy. MM_AUDIO_CODEC: Updated unCoding values. MM_CAPTURE_CMPLT_DETAILS: Updated structure example and description. MM_GETDURATION_CMPLT: Updated description and added the details field. MM_GETDURATION_CMPLT_DETAILS: Updated the entire data structure reference page. MM_GETDURATION_INFO: Updated the entire data structure reference page. MM_GETELAPSEDTIME_CMPLT: Updated description and added the unCount and details field. MM_GETELAPSEDTIME_CMPLT_DETAILS: Updated description and added the unStreamType field. MM_GETELAPSEDTIME_INFO: Updated structure example and definition. MM_IMAGE_FORMAT: Updated the structure example and description. Added a cross-reference to MM_YUV. MM_MEDIA_ACCESS_MEMORY: Updated the pBuffer and unBufferSize field descriptions. MM_MEDIA_ACCESS_STREAM: Added the nStreamHandle field. MM_MEDIA_AUDIO: Updated all field descriptions. MM_MEDIA_IMAGE: Updated the structure example, description, and field descriptions. Added valid values for the eFormat field. MM_MEDIA_ITEM: Updated field descriptions. MM_MEDIA_ITEM_LIST: Updated descriptions of the item, next, and prev fields. MM_MEDIA_TERM: Indicated that this data structure is for future use. MM_MEDIA_VIDEO: Updated descriptions of the unMode, szFileName, stream, memory, and unAccessMode fields. MM_METAEVENT: Updated descriptions of the evtdatap, evtlen, evtdev, evttype, and evtUserInfo fields. MM_PAUSE_INFO: Updated the structure example, description, and added the unStreamType field. MM_PAUSE_PLAY_CMPLT: Updated the structure description and the unCount field description. Added details field. MM_PAUSE_PLAY_CMPLT_DETAILS: Updated the structure description and added the unStreamType field. MM_PLAY_RECORD_CMPLT_DETAILS: Updated the description note and the Complete and Reason fields. MM_PLAY_RECORD_LIST: Updated the ItemType field. MM_RECORD_CMPLT_INFO_DATA_BLOCKS: Updated the structure example. MM_RESUME_INFO: Updated the structure example and the unStreamType and unAttribute fields. MM_RESUME_PLAY_CMPLT: Updated the description and the unCount field. Added the details field. MM_RESUME_PLAY_CMPLT_DETAILS: Updated the description and the unStreamType field. MM_SEEK_CMPLT: Updated the description and the unCount field. Added the details field. MM_SEEK_CMPLT_DETAILS: Updated the description and the unStreamType field. MM_SEEK_INFO: Updated the description and field descriptions. MM_STREAM_OPEN_INFO: Updated structure example and description. MM_STREAM_STAT: Updated structure example and description.

Document No.	Publication Date	Description of Revisions
05-2454-007 (continued)	November 2008	MM_STREAM_WATERMARK_INFO: Updated structure example, description, and unValue field. MM_YUV: Added a valid value to the eFormat field. MM_MODE_AUD_FILE_TYPE_VOX: Removed; deprecated.
05-2454-006	June 2008	Added information from the former Programming Guide to achieve a combined Programming Guide and API Library Reference. Added a "Platform" row to each API reference page function table to indicate Platform support. Application Development Guidelines: Added the following sections: Enabling and Using Transcoding, Play a Still Image, Capture a Still Image. Error Codes: Updated error codes to support the Capture Still Image feature. Function Summary by Category: Added DVR Control functions category to support the DVR control feature. mm_Capture( ): Added this new function to support the Capture Still Image feature. mm_GetDuration( ): Added this new function to support the DVR control feature. mm_GetElapsedTime( ): Added this new function to support the DVR control feature. mm_Pause( ): Added this new function to support the DVR control feature. mm_Play( ): New code example with RTC condition support. mm_Record( ): New code example with RTC condition support. mm_Resume( ): Added this new function to support the DVR control feature. mm_Seek( ): Added this new function to support the DVR control feature. MM_CAPTURE_CMPLT_DETAILS: Added this new function to support the Capture Still Image feature. MM_GETDURATION_CMPLT: Added this new data structure to support the DVR control feature. MM_GETDURATION_CMPLT_DETAILS: Added this new data structure to support the DVR control feature. MM_GETDURATION_INFO: Added this new data structure to support the DVR control feature. MM_GETELAPSEDTIME_CMPLT: Added this new data structure to support the DVR control feature. MM_GETELAPSEDTIME_CMPLT_DETAILS: Added this new data structure to support the DVR control feature. MM_GETELAPSEDTIME_INFO: Added this new data structure to support the DVR control feature. MM_MEDIA_AUDIO: Updated data structure fields for WAVE support. MM_IMAGE_FORMAT: Added this new data structure to support the Capture Still Image feature. Continued on the next page.

Document No.	Publication Date	Description of Revisions
05-2454-006 (continued)	June 2008	MM_MEDIA_IMAGE: Added this new data structure to support the Play Still Image feature. MM_MEDIA_ITEM: Updated field definitions to support the Play/Capture Still Image feature. MM_MEDIA_ITEM_LIST: Updated field definitions to support the Play/Capture Still Image feature. MM_PAUSE_INFO: Added this new data structure to support the DVR control feature. MM_PAUSE_PLAY_CMPLT: Added this new data structure to support the DVR control feature. MM_PAUSE_PLAY_CMPLT_DETAILS: Added this new data structure to support the DVR control feature. MM_RESUME_INFO: Added this new data structure to support the DVR control feature. MM_RESUME_PLAY_CMPLT: Added this new data structure to support the DVR control feature. MM_RESUME_PLAY_CMPLT_DETAILS: Added this new data structure to support the DVR control feature. MM_RET_CODE: Added MM_CAPTURE_ACK. MM_SEEK_CMPLT: Added this new data structure to support the DVR control feature. MM_SEEK_CMPLT_DETAILS: Added this new data structure to support the DVR control feature. MM_SEEK_INFO: Added this new data structure to support the DVR control feature. MM_VIDEO_CODEC: Updated the values for the eFramesPerSec field. MM_YUV: Added this new data structure to support the Play/Capture Still Image feature.

Document No.	Publication Date	Description of Revisions
05-2454-005	March 2008	mm_Capture( ): Removed; not yet supported. MM_CAPTURE_CMPLT_DETAILS: Removed; not yet supported. MM_IMAGE_FORMAT: Removed; not yet supported. MM_MEDIA_IMAGE: Removed; not yet supported. MM_YUV: Removed; not yet supported. Feature Support by Platform: Added this new chapter to indicate feature support by platform. Dialogic® Multimedia API Events: Removed events related to Capture; not yet supported. MM_AUDIO_CODEC: Added the MM_DATA_FORMAT_ALAW type and MM_DATA_FORMAT_MULAW type to the “Uncoding” field of this data structure. MM_AUDIO_CODEC_OPTION_LIST: Updated the example code by replacing the depreciated MM_MODE_AUD_FILE_TYPE_VOX with a 0. mm_Play( ): Updated the example code by replacing the deprecated MM_MODE_AUD_FILE_TYPE_VOX with a 0. Removed references to Play Still Image; not yet supported. MM_MEDIA_ACCESS_MEMORY: Corrected data structure description. MM_MEDIA_AUDIO: Removed note referencing WAVE in the unOffset field; not yet supported. Removed the following value from the eFileFormat field, EMM_AUD_FILEFORMAT_WAVE; not yet supported. MM_MEDIA_VIDEO: Updated the description of the unOffset field and updated the values in the eFileFormat field. MM_PLAY_RECORD_CMPLT: Updated this data structure to support the User Recorded Data Update feature. MM_PLAY_RECORD_CMPLT_DETAILS: Updated the “Complete” and “Status” fields of this data structure, and added macros for simplifying data update retrieval to support the User Recorded Data Update feature.Removed the EMM_TR_DURATION_COMPLETE value. mm_Record( ): Updated the example code by replacing the deprecated MM_MODE_AUD_FILE_TYPE_VOX with a 0. MM_RECORD_CMPLT_DATA_BLOCK: Added this new data structure to support the User Recorded Data Update feature. MM_RECORD_CMPLT_INFO_DATA_BLOCKS: Added this new data structure to support the User Recorded Data Update feature. MM_RUNTIME_CONTROL: Added “Valid only as a return code” to the EMM_TERM_NORTC value of the “Reason” field. (IPY00042141) Also added the EMM_TERM_TIME_FROM_AUDIO_RECORD_STARTED value to the Reason field. Continued on next page.

Document No.	Publication Date	Description of Revisions
05-2454-005 (continued)	March 2008	mm_StreamRead( ): Add EMM_STREAM_EMPTY to the list of "Returns" for this function. mm_StreamWrite( ): Changed EMM_STREAM_EMPTY in the list of "Returns" to EMM_STREAM_FULL for this function. MM_VIDEO_CODEC: Added suggested values in bits per second (bps) to the "Bitrate" field. Media Streaming Event Information Error Return Codes: Added "Valid only as a return code" to the EMM_TERM_NORTC. (IPY00042141) Media Streaming Event Information Error Return Codes: Added EMM_TERM_TIME_FROM_AUDIO_RECORD_STARTED. Terminating and Non-Terminating Play/Record Errors: Added EMM_STATUS_PLAY_BAD_PACKET, EMM_STATUS_PLAY_CODING_ERROR, and EMM_STATUS_RCRD_CODING_ERROR.
05-2454-004	February 2008	Error Codes: Updated error codes to support the Capture Still Image feature. mm_Capture( ): Added this new function to support the Capture Still Image feature. mm_StreamRead( ): Corrected the description for the pDataSize parameter. (IPY00041590) Events: Correct the descriptions for MMEV_PLAY_VIDEO_HIGHWATER and MMEV_PLAY_VIDEO_LOWWATER. (IPY00041454) MM_IMAGE_FORMAT: Added this new data structure to support the Capture Still Image feature. MM_MEDIA_AUDIO: Updated field definitions to support the WAVE file format. MM_MEDIA_IMAGE: Added this new data structure to support the Play Still Image feature. MM_MEDIA_ITEM: Updated field definitions to support the Play/Capture Still Image feature. MM_MEDIA_ITEM_LIST: Updated field definitions to support the Play/Capture Still Image feature. MM_MEDIA_VIDEO: Updated field definitions to support the WAVE file format. mm_Play( ): Updated the function definitions to support the Play Still Image feature. MM_PLAY_RECORD_CMPLT_DETAILS: Updated field definitions to support the Play Still Image feature. MM_PLAY_RECORD_INFO: Updated field definitions to support the WAVE file format. MM_RET_CODE: Updated field definitions to support the Play/Capture Still Image feature. MM_STOP: Updated field definitions to support the Play/Capture Still Image feature. MM_STOP_ACK_DETAILS: Updated field definitions to support the Play/Capture Still Image feature. MM_STREAM_OPEN_INFO: Corrected the descriptions for the BufferMode field. (IPY00041320) MM_YUV: Added this new data structure to support the Play/Capture Still Image feature.

Document No.	Publication Date	Description of Revisions
05-2454-003	October 2007	Dialogic® Multimedia API Header File: Added the *videodefs.h* file. **mm_Play( )** and **mm_Record( )**: Updated code examples that include compilation fixes and missing INIT inline functions. Data Structures: Updated the unVersion field description for all data structure reference pages. Overview of Dialogic® Multimedia API Data Structures: Added a note about using the corresponding INIT_ inline function, required by the application for initialization, for each data structure. MM_AUDIO_CODEC: Updated the structure definition and added new fields and new values for enumerations. MM_AUDIO_CODEC_OPTION_LIST: Added this new data structure. MM_MEDIA_VIDEO: Updated structure definition, and description of eFileFormat and unOffset. MM_PLAY_RECORD_CMPLT_DETAILS: Added the new error code, EMM_STATUS_RCRD_V_MISSING_MPEG4_VISUALCONFIG_ERROR. MM_RUNTIME_CONTROL: Updated the Reason field description. MM_SET_PARM: Added the EMM_TONE_DETECTION parameter. Terminating and Non-Terminating Play/Record Errors: Added the new error code, EMM_STATUS_RCRD_V_MISSING_MPEG4_VISUALCONFIG_ERROR.
05-2454-002	August 2007	Made global changes to reflect Dialogic brand. Function Summary by Category chapter: Added the Media Streaming category. Dialogic® Multimedia API Events section: Added the following new events: MMEV_PLAY_AUDIO_LOWWATER, MMEV_PLAY_AUDIO_HIGHWATER, MMEV_PLAY_VIDEO_LOWWATER, MMEV_PLAY_VIDEO_HIGHWATER. MMEV_RECORD_AUDIO_LOWWATER, MMEV_RECORD_AUDIO_EOD, MMEV_RECORD_VIDEO_LOWWATER, MMEV_RECORD_VIDEO_EOD. Function Information chapter: Added the following new functions: **mm_StreamClose( ), mm_StreamGetStat( ), mm_StreamOpen( ), mm_StreamRead( ), mm_StreamReset( ), mm_StreamSetWaterMark( ), mm_StreamWrite( ), mm_ResultInfo( )** Data Structures chapter: Added the following new data structures: MM_STREAM_STAT, MM_RUNTIME_CONTROL, MM_STREAM_OPEN_INFO MM_STREAM_WATERMARK_INFO, MM_MEDIA_ACCESS_MEMORY. MM_MEDIA_ACCESS_STREAM. MM_AUDIO_CODEC: Added new data format values to the unCoding field for AMR-NB and G.723.1. Media Streaming Event Information Error Return Codes section: Added new error codes.
05-2454-001	August 2005	Initial version of document.

Product Description 1

This chapter provides an overview of video technology and Dialogic® multimedia features. The following topics are included:

- Overview of Video Technology 17
- IP Media Server 21
- Multimedia Capabilities 21
- File Formats 22
- SIP Call Control Using Global Call 23
- SDP Functionality 24
- Features 24

1.1 Overview of Video Technology

This section covers the following concepts in video technology:

- Video Codec
- Audio Codec
- Video Standards
- Intraframe (I-Frame)
- IP Media Server

1.1.1 Video Codec

A video codec is a device or software module that encodes and compresses raw video data to a digital format or decodes and decompresses from a digital video data format to a raw video data format. A video codec may also transcode between two digital formats. For multimedia, video transcoding is provided, if necessary, between the digital video format of the Real-time Transport Protocol (RTP) stream and the digital video format of the multimedia file that is being played from or recorded to.

1.1.2 Audio Codec

An audio codec is a device or software module that encodes and compresses analog audio data to a digital format or decodes and decompresses from a digital audio data format to analog. An audio codec may also transcode between two digital formats. For multimedia, audio transcoding is provided, if necessary, between the digital audio format of the RTP stream and the digital audio format of the multimedia file that is being played from or recorded to.

1.1.3 Video Standards

The most important video standards of today are H.261, H.263, MPEG-1, MPEG-2 and MPEG-4, and H.264. These and other standards are discussed in the following sections:

- H. 261 (P*64)
- H.263
- MPEG-1
- MPEG-2
- MPEG-4
- H.264

Compared to video codecs for CD-ROM or TV broadcast, codecs designed for the Internet require greater scalability, lower computational complexity, greater resiliency to network losses, and lower encode/decode latency for video conferencing. In addition, the codecs must be tightly linked to network delivery software to achieve high frame rates and picture quality.

Not all video standards are well suited for Internet video. Standards such as MPEG-4 part 10 and H.264 (which are the same codec) have emerged which bridge the gap between high efficiency error-resistant codecs for telephony applications (H.26x) and codecs designed for entertainment content streaming (MPEG-x). It is poised for more effort to be focused on algorithms specifically designed for real-time video applications in the Internet. Research is currently underway looking at new scalable and flexible codecs, video QoS, etc.

H. 261 (P*64)

H.261, also known as P*64 codec (where P is an integer number meant to represent multiples of 64 Kbps), is an old standard and was targeted at teleconferencing applications. H.261 is used for carrying video over ISDN – in particular for face-to-face videophone applications and video conferencing. The actual encoding algorithm is similar to but incompatible with that of MPEG codecs. However, H.261 needs substantially less CPU power for real-time encoding than MPEG. The algorithm includes a mechanism that optimizes bandwidth usage by trading picture quality against motion, so that a quickly-changing picture will have a lower quality than a relatively static picture. Used in this way, H.261 is a constant-bit-rate encoding codec rather than a constant-quality, variable-bit-rate encoding codec.

H.263

H.263 is the ITU-T standard designed for low bit rate communication. However, this standard can probably be used for a wide range of applications – not just low bit rate applications. H.263 is expected to replace H.261 in most applications.

The coding algorithm of H.263 is similar to that used by H.261, but it has been enhanced to improve performance and error recovery. H.263 uses half-pixel precision, whereas H.261 uses full-pixel precision and a loop filter. Some parts of the hierarchical structure of the data stream are now optional, so the codec can be configured for a lower data rate or better error recovery.

There are now four optional negotiable options included to improve performance:

- Unrestricted motion vectors
- Syntax-based arithmetic coding
- Advance prediction
- Forward and backward frame prediction similar to MPEG called P-B frames.

H.263 supports the following resolutions:

- Quarter Common Intermediate Format (QCIF) and Common Intermediate Format (CIF) – the resolutions supported by H.261
- Sub-QCIF (SQCIF) – half the resolution of QCIF
- 4CIF – 4 times the resolution of CIF
- 16CIF – 16 times the resolution of CIF

The support of 4CIF and 16CIF means the codec can compete with other, higher bit rate video coding standards such as the MPEG standards.

MPEG-1

MPEG-1, 2, and 4 are currently accepted standards for the bandwidth efficient transmission of video and audio. The MPEG-1 codec targets a bandwidth of 1 to 1.5 Mbps offering VHS quality video at CIF (352x288) resolution and 30 frames per second. MPEG-1 requires expensive hardware for real-time encoding. Although decoding can be done in software, most implementations consume a large fraction of a high-end processor. MPEG-1 does not offer resolution scalability and the video quality is susceptible to packet losses due to the dependencies present in the P (predicted) and B (bi-directionally predicted) frames (for more information about P and B frames, refer to Section 1.1.4, “Intraframe (I-Frame)”, on page 21). The B-frames also introduce latency in the encode process, since encoding frame N needs access to frame N+k, making it less suitable for video conferencing.

MPEG-2

MPEG-2 extends MPEG-1 by including support for higher resolution video and increased audio capabilities. The targeted bit rate for MPEG-2 is 4 to15 Mbps, providing broadcast quality full-screen video. The MPEG-2 draft standard does provide for scalability. Three types of scalability have been defined:

- Signal-to-Noise Ratio (SNR)
- Spatial and temporal
- One extension that can be used to implement scalability: data partitioning.

Compared with MPEG-1, MPEG-2 requires even more expensive hardware to encode and decode. MPEG-2 is also prone to poor video quality in the presence of losses, for the same reasons as MPEG-1. Both MPEG-1 and MPEG-2 are well suited to the purposes for which they were developed. For example, MPEG-1 works very well for playback from CD-ROM, and MPEG-2 performs well for high-quality archiving applications and for TV broadcast applications. In the case of satellite broadcasts, MPEG-2 allows more than five digital channels to be encoded using

the same bandwidth as used by a single analog channel today, without sacrificing video quality. The large encoding costs are really not a factor. However, for existing computer and Internet infrastructures, MPEG-based solutions can be expensive and can require too much bandwidth because they were not designed with the Internet in mind.

MPEG-4

MPEG-4, the most recent encoding standard from MPEG, was finalized in October 1998 and ratified as a standard in 1999. MPEG-4 arose from a need to have a scalable standard supporting a wide bandwidth range from streaming video at less than 64 Kbps, suitable for Internet applications, to approximately 4 Mbps for higher-bandwidth video needs. MPEG-4 also arose from a desire, as digital encoding matures, to advance beyond simple conversion and compression to object recognition and encoding, as well as provide synchronized text and metadata tracks, to create a digital file that carries a meaning greater than the sum of its individual parts.

MPEG-4 supports both progressive and interlaced video encoding. One notable concept of MPEG-4 is that it is object-based, coding multiple video object planes into images of arbitrary shape. Successive video object planes (VOPs) belonging to the same object in the same scene are encoded as video objects. MPEG-4 supports both natural (“analog”) and synthetic (“computer-generated”) data coding. Some VRML (Virtual Reality Modeling Language) technology is also incorporated to encode dimensionality.

MPEG-4 compression provides temporal scalability utilizing object recognition, providing higher compression for background objects, such as trees and scenery, and lower compression for foreground objects, such as an actor or speaker – much as the human eye filters information by focusing on the most significant object in view, such as the other party in a conversation. Object encoding provides potential for object or visual recognition indexing, based on discrete objects within a frame rather than requiring a separate text-based or storyboard indexing database. In addition, MPEG-4 provides a synchronized text tract for courseware development and a synchronized metadata track for indexing and access at the frame level.

H.264

The H.264 project was to develop a high-performance video coding standard by adopting a “back to basics” approach where simple and straightforward design using well-known building blocks is used. The H.264 standard has a number of advantages that distinguish it from the other existing standards mentioned above, while at the same time sharing common features with other existing standards. The following are some of the key advantages of H.264:

- Compared to H.263v2 (H.263+) or MPEG-4 Simple Profile, H.264 permits a reduction in bit rate by up to 50% for a similar degree of encoder optimization at most bit rates.
- Higher quality video: H.264 offers consistently good video quality at high and low bit rates.
- Error resilience: H.264 provides the tools necessary to deal with packet loss in packet networks and bit errors in error-prone wireless networks.
- Network friendliness: Through the Network Adaptation Layer, H.264 bit streams can be easily transported over different networks.

In general, H.263 and MPEG-4 have become the de facto standards for video delivery over low bandwidths. But broadband standards such as MPEG-1 and MPEG-2, which are useful for many

types of broadcast and CD-ROM applications, are unsuitable for the Internet. In fact, with the adoption of broadband, there may be significant growth in MPEG-1 and MPEG-2 applications. Although MPEG-2 has had scalability enhancements, these likely will not be exploitable until the availability of reasonably priced hardware encoders and decoders that support scalable MPEG-2.

1.1.4 Intraframe (I-Frame)

To create motion in a video, individual frames of pictures are grouped together and played back. An Intraframe (I-frame) is a single frame of digital content that the compressor examines, independent of the frames that precede and follow it, and stores all of the data needed to display that frame. Several video codec standards, such as MPEG-4, H.263, and H.264, use I-frames.

In a compressed video, I-frames are usually used along with P-frames and B-frames, which are sometimes referred to as Interframes or delta frames. P-frames (short for predictive frames) follow I-frames and contain only data that has changed from the I-frame that comes before it. B-frames (short for bi-directional predictive frames) contain only the data that have changed from the preceding frame or differ from the data in the next frame.

So only the I-frame contains all the data needed to display the frame whereas the P- and B-frames store the changes that occur between the I-frames. For this reason, video decoders typically require at least a single initial I-frame before they can begin to accurately display video content. The quality of the video increases with a higher amount of I-frames, but I-frames take up more storage space and transmission bandwidth because they contain higher amounts of information bits.

1.2 IP Media Server

The Dialogic® Host Media Processing (HMP) Software provides the services of an IP Multimedia Server (IPMS). An IPMS provides various audio and video media functions for wireline and wireless network environments. Some applications that use the Dialogic® HMP Software include video mail, video conferencing, and video streaming. The user application uses the Dialogic® Multimedia API to access and manage these multimedia functions.

1.3 Multimedia Capabilities

This section lists the multimedia capabilities provided by the Dialogic® Multimedia API Library and other related Dialogic® API libraries:

- Dialogic® Multimedia API Library - records and plays the multimedia data using a multimedia device.
- Dialogic® Device Management API Library - used to manage connections between devices for communication.
- Dialogic® IP Media Library API Library - provides the IP multimedia session control.
- Dialogic® Global Call API Library - provides IP call control for multimedia using SIP and Session Description Protocol (SDP) and must be used in third party call control (3PCC) mode.

Refer for Chapter 7, "Function Summary by Category" and beyond for more information about the Dialogic® Multimedia API Library. Information about the other APIs can be found in the corresponding API Library Reference documentation provided with this release.

These capabilities support applications providing video services, such as video mail, video color ring, video caller ID, and video location-based services.

The following multimedia capabilities are provided:

- Multimedia record and playback with basic playback control and synchronized audio and video.
  - Record and playback of audio and video, video only, or audio only
  - Record to and playback from a file or set of files
  - Transmit notification tone at start of recording
- Record from RTP stream to multimedia file. Play from multimedia file into RTP stream while maintaining synchronization.
- Supports audio codecs for RTP. Refer to the Release Guide for a current list of supported audio codecs.
- Supports video picture formats at 30, 15, 10, or 6 frames per second. Refer to the Release Guide for a current list of supported video picture formats.
- Supports video codec for RTP. Refer to the Release Guide for a current list of supported video codecs.
- Multimedia File Conversion Tools: These utilities (e.g., mmconvert and hmp3gp) provide off-line conversion of multimedia files. For information on obtaining and using the multimedia file conversion tools, see following web sites. Check these web sites periodically for updates to the conversion tools and their capabilities, as well as to the Multimedia File Conversion Tools User Guide:
  - http://www.dialogic.com/products/media-server-software/default.htm
  - http://www.dialogic.com/manuals
- Play to and record from SIP devices, depending on the capability of the device (audio or audio/video). Play video only if no audio is required. Play audio only for non-video devices.
- Supports existing Dialogic® IP Media Library API audio alarms for the voice portion of the multimedia stream.

  Note: Quality of Service (QoS) alarms and events are not supported for video streams.
- Play Dialogic® Voice API audio files in a multimedia session. You can play Dialogic® Voice API audio files in a multimedia session as long as tight synchronization with video is not required (as when playing with a video menu or status display). In this case, the "ipm" device in a multimedia session will listen to the "dxxx" device to which the Dialogic® Voice API is playing an audio file. This overrides any audio stream (but not video) from the "mm" device in the multimedia session.

1.4 File Formats

The Dialogic® Multimedia API supports the following file formats:

- Dialogic® Voice API Audio Play and Record File Formats

  - G.711 μ-law and A-law (48 kbps and 64 kbps)
  - OKI ADPCM (24 kbps and 32 kbps)
  - G.726 (16 kbps and 32 kbps)
  - Linear PCM (88 kbps)
  - Linear PCM (128 kbps).
- Dialogic® Multimedia API Audio Play and Record File Format
  - Linear PCM (128 kbps): 16-bit, 8 kHz, mono, LSB-MSB ("little-endian").
- Dialogic® Multimedia API Video Play and Record File Formats
  - Dialogic proprietary format

  Note: Except for the 128 kbps linear PCM file format, the Dialogic® Multimedia API does not directly support the Dialogic® Voice API audio files. However, they are indirectly supported and you can play these Dialogic® Voice API audio files in a multimedia session as long as tight synchronization with video is not required (as when playing with a video menu or status display). In this case, the "ipm" device in a multimedia session will listen to the "dxxx" device to which the Dialogic® Voice API is playing an audio file. This overrides any audio stream (but not video) from the "mm" device in the multimedia session.

1.5 SIP Call Control Using Global Call

This section discusses the use of Dialogic® Global Call APIs for SIP call control in multimedia applications.

Note: More information about the Dialogic® Global Call API can be found in the *Dialogic® Global Call API Library Reference* and *Dialogic® Global Call API Programming Guide*. The Dialogic® Global Call API provides a "C" interface that allows you to develop both TDM and IP call control applications using a variety of TDM protocols such as ISDN PRI and T1 CAS, and IP protocols like SIP and H.323. It provides an abstraction to the application from the lower level details of each individual protocol, thus enabling faster deployment of multimedia applications.

The Global Call call control library operates in two modes:

- First Party Call Control (1PCC) mode, which is the default
- Third Party Call Control (3PCC) mode

First Party Call Control (1PCC) mode is where the Global Call call control library manages IP signaling (SIP or H.323) as well as the RTP media control via the Dialogic® IP Media Library API. The Dialogic® IP Media Library API is a complementary API that is used to manipulate RTP media streams (start, stop, or modify them). The call control library synchronizes the call control and media control state machines, thus abstracting the application from the details of creating and parsing Session Description Protocol (SDP) bodies that are embedded in SIP messages and invoking the Dialogic® IP Media Library API functions to start, stop, or modify RTP streams.

Note: More information about the Dialogic® IP Media Library API can be found in the *Dialogic® IP Media Library API Library Reference* and *Dialogic® IP Media Library API Programming Guide*.

In the Third Party Call Control (3PCC) mode, the Dialogic® Global Call API call control library provides maximum flexibility to the application, allowing it to manipulate both SDP bodies and

invoke Dialogic® IP Media Library API functions as needed. The Dialogic® Global Call API library in this mode is acting as a pure SIP application. The application is responsible for monitoring the various SIP transactions and invoking appropriate IPML functions to start, stop, or modify RTP streaming and create or parse SDP bodies.

Apart from the capabilities discussed above, Dialogic® Global Call API allows the application to manipulate any standard SIP header (read or write), while also providing the flexibility to add custom headers to SIP messages.

Note: To play and record multimedia with the Multimedia API, you must configure and use the Dialogic® Global Call API library for 3PCC mode.

1.6 SDP Functionality

This section discusses Session Description Protocol (SDP) functionality.

SDP, which is defined in IETF RFC 2327, provides a standard way to specify the media information for an IP call. It is used by SIP, MGCP, and H.248 (or Megaco) protocols. The media information could be anything from RTP audio and video details such as UDP port numbers, IP addresses, audio/video coder information, or bandwidth. A sample SDP body is shown below:

```
v=0
o=mhandley 2890844526 2890842807 IN IP4 126.16.64.4
s=SDP Seminar
i=A Seminar on the session description protocol
u=http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps
e=mjh@isi.edu (Mark Handley)
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
a=recvonly
m=audio 49170 RTP/AVP 0
m=video 51372 RTP/AVP 31
m=application 32416 udp wb
a=orient:portrait
```

In this Dialogic® HMP Software release, the application is responsible for creating and parsing SDP bodies that are embedded in SIP messages. For convenience, a sample class library is provided “as is” for developers who choose to create and parse SDP bodies. It is a simple C++ interface provided as source and can be modified to suit application needs.

1.7 Features

This section covers the following features included in this release and implemented using the Dialogic® Multimedia API library. The library is used to play and record digitized multimedia in support of applications providing video services, such as video mail, video color ring, video caller ID, and video location-based services.

- Digital Video Recorder (DVR) Controls
- Native RTP Hairpinning

- Native RTP Play and Record
- Multimedia User I/O
- Multimedia Buffer I/O
- Multimedia Runtime Control
- Play/Capture Still Image
- Video Transcoding
- 3GP File Format Direct Playback

Note: Not all features are supported on all platforms and releases. Refer to the Release Guide for the applicable platform to determine feature support. See Chapter 2, "Feature Support by Platform" for more information.

1.7.1 Digital Video Recorder (DVR) Controls

This feature enables the application to pause, resume, and seek during audio and video playback.

1.7.2 Native RTP Hairpinning

Native RTP hairpinning enables applications to form RTP media stream connections between IP media streams, allowing the RTP media stream received from one IP media session to be retransmitted to the outgoing RTP media stream of another IP media session. This is done without processing or transcoding the RTP payload. The RTP packets that are hairpinned may be hairpinned as received, with packets out of order or even missing as long as the RTP stream is retransmitted so that the receiving terminating endpoint can perform necessary packet loss recovery type functions.

Native RTP hairpinning is supported for both audio and video RTP streams.

In addition to IPM to IPM connections, native hairpinning connections may also be formed between 3G-324M and IPM devices. Connections between devices are made using the Dialogic® Device Management API library.

Use cases for native RTP hairpinning include switching type applications, such as an IPPBX, or streaming data to an external speech server. Benefits of using native RTP hairpinning include increasing achievable system densities, reducing latencies, and improving voice quality by eliminating an additional decode/encode operation.

Note: RTCP data is not hairpinned, and SRTP data cannot be hairpinned.

1.7.3 Native RTP Play and Record

The RTP data in both incoming and outgoing directions is not processed or transcoded by Dialogic® HMP Software. With this feature, the RTP data is stored directly by and retrieved directly from Dialogic® HMP Software without application data handling.

The application must negotiate the proper coder formats when establishing the IP media sessions and match these formats when receiving and sourcing data. Additional media stream data such as

RTP timestamps and sequence numbers are made available to the application by the native RTP feature. The RTP packets may be stored exactly as received, with packets out of order or even missing, as long as the RTP stream is retransmitted so that the receiving terminating endpoint can perform necessary packet loss recovery (PLR) activity.

1.7.4 Multimedia User I/O

This feature enables applications to directly play and record RTP data via user I/O buffers. New functions and data structures have been added to support native RTP play and record, multimedia user I/O, multimedia buffer I/O.

1.7.5 Multimedia Buffer I/O

This feature enables applications to directly play and record RTP data via memory.

1.7.6 Multimedia Runtime Control

This feature allows multimedia play and record functions to be terminated on certain conditions such as digits received. These conditions are specified in the MM_RUNTIME_CONTROL data structure.

1.7.7 Play/Capture Still Image

This feature provides a method to play a still image over a video stream and to capture a still image from a video stream.

1.7.8 Video Transcoding

Video transcoding enables applications to record incoming video in a different format than what is being received from the network and to play back outgoing video in a different format than that of the locally stored file. Transcoding involves decoding and decompressing the original data to a raw intermediate format (YUV format).

Video transrating adjusts the number of video frames per second (and bit rate of the video) between two endpoints to suit the requirements of the device at each endpoint.

Image resizing converts video from one image size to another (for example, from CIF to QCIF) between two endpoints to suit the requirements of the device at each endpoint.

Note: In this document, the term “video transcoding” encompasses video transcoding, video transrating, and image resizing.

1.7.9 3GP File Format Direct Playback

With this feature, Dialogic® products support the playback of contiguous multimedia files in an ISO/IEC standardized structure, known as 3GPP or 3GP file format. This feature also enables

direct playback of 3GP files via the Dialogic® Multimedia API library. For more information, refer to Section 5.10, "3GP File Format Direct Playback", on page 59.

Feature Support by Platform 2

This Multimedia API Library Reference is used across many Dialogic® platforms. There may be information in this manual that is not yet implemented in a Dialogic® platform software release. Be sure to refer to the Release Guide for the applicable platform to determine feature support.

- High-Level Feature Support by Platform
- Dialogic® Multimedia API Function Support by Platform
- Dialogic® Multimedia API Audio Formats by Platform

2.1 High-Level Feature Support by Platform

This section lists the high-level features documented in this current version and lists the Dialogic® platform (software release) that currently supports each feature.

Note: Features and functionality are being introduced in phases. This table will be updated as a feature documented in this manual becomes available on a particular platform (software release) or service update (SU). Refer to the Release Guide for the applicable platform to determine feature support.

Table 1. High-Level Feature Support by Platform

Feature	Dialogic® HMP Software3.0WIN	Dialogic® HMP Software 3.1LIN	Dialogic® HMP Software 4.1LIN
Multimedia User I/O	NS	S	S
Multimedia Buffer I/O	NS	S	S
Multimedia Play and Record from/to File	S	S	S
Multimedia Runtime Control (RTC)	NS	S	S
Native Audio Play and Record	S	S	S
Native Video Play and Record	S	S	S
Audio Transcoding	S	S	S
Video Transcoding	NS	NS	S
Play Still Image	NS	NS	S
Capture Still Image	NS	NS	S

Legend:
S = supported; NS = not supported
Dialogic® HMP Software 3.0WIN = Dialogic® Host Media Processing Software Release 3.0WIN
Dialogic® HMP Software 3.1LIN = Dialogic® Host Media Processing Software Release 3.1LIN
Dialogic® HMP Software 4.1LIN = Dialogic® Host Media Processing Software Release 4.1LIN
File Format: P-Proprietary, V-Vox, W-Wave, O-3GP Offline Conversion, G-3GP Runtime Support (play only)

Table 1. High-Level Feature Support by Platform (Continued)

Feature	Dialogic® HMP Software3.0WIN	Dialogic® HMP Software 3.1LIN	Dialogic® HMP Software 4.1LIN
DVR Control	NS	NS	S
File Format	P,V, O	P,V,W,O	P,V,W, O,G

Legend:
S = supported; NS = not supported
Dialogic® HMP Software 3.0WIN = Dialogic® Host Media Processing Software Release 3.0WIN
Dialogic® HMP Software 3.1LIN = Dialogic® Host Media Processing Software Release 3.1LIN
Dialogic® HMP Software 4.1LIN = Dialogic® Host Media Processing Software Release 4.1LIN
File Format: P-Proprietary, V-Vox, W-Wave, O-3GP Offline Conversion, G-3GP Runtime Support (play only)

2.2 Dialogic® Multimedia API Function Support by Platform

The following table provides an alphabetical listing of Dialogic® Multimedia API functions. The table indicates which platforms are supported for each of the functions.

Although a function may be supported on all platforms, there may be some differences on its use. For details, see the function reference descriptions in Chapter 8, "Function Information".

Table 2. Dialogic® Multimedia API Function Support by Platform

Function Name	Dialogic® HMP Software 3.0WIN	Dialogic® HMP Software 3.1LIN	Dialogic® HMP Software 4.1LIN
Device Management Functions			
mm_Close( )	S	S	S
mm_Open( )	S	S	S
mm_Reset( )	S	S	S
Configuration Functions			
mm_GetParm( )	S	S	S
mm_SetParm( )	S	S	S
Input/Output Functions			
mm_Capture( )	NS	NS	S
mm_Play( )	S	S	S
mm_Record( )	S	S	S
mm_Stop( )	S	S	S

Legend:
S = supported, NS = not supported
Dialogic® HMP Software 3.0WIN = Dialogic® Host Media Processing Software Release 3.0WIN
Dialogic® HMP Software 3.1LIN = Dialogic® Host Media Processing Software Release 3.1LIN
Dialogic® HMP Software 4.1LIN = Dialogic® Host Media Processing Software Release 4.1LIN

Table 2. Dialogic® Multimedia API Function Support by Platform

Function Name	Dialogic® HMP Software 3.0WIN	Dialogic® HMP Software 3.1LIN	Dialogic® HMP Software 4.1LIN
Media Streaming Functions			
mm_StreamClose( )	NS	S	S
mm_StreamGetStat( )	NS	S	S
mm_StreamOpen( )	NS	S	S
mm_StreamRead( )	NS	S	S
mm_StreamReset( )	NS	S	S
mm_StreamSetWaterMark( )	NS	S	S
mm_StreamWrite( )	NS	S	S
DVR Control Functions			
mm_GetDuration( )	NS	NS	S
mm_GetElapsedTime( )	NS	NS	S
mm_Pause( )	NS	NS	S
mm_Resume( )	NS	NS	S
mm_Seek( )	NS	NS	S
Event Information Functions			
mm_DisableEvents( )	S	S	S
mm_EnableEvents( )	S	S	S
mm_GetMetaEvent( )	S	S	S
mm_ResultInfo( )	S	S	S
Error Processing Function			
mm_ErrorInfo( )	S	S	S

Legend:
S = supported, NS = not supported
Dialogic® HMP Software 3.0WIN = Dialogic® Host Media Processing Software Release 3.0WIN
Dialogic® HMP Software 3.1LIN = Dialogic® Host Media Processing Software Release 3.1LIN
Dialogic® HMP Software 4.1LIN = Dialogic® Host Media Processing Software Release 4.1LIN

2.3 Dialogic® Multimedia API Audio Formats by Platform

The following table indicates the formats in which audio data can be recorded and played back by the Dialogic® software. For more information, refer to the MM_AUDIO_CODEC reference page in this document.

Table 3. Audio Format Support by Platform

Value	Description	Dialogic® HMP Software 3.0WIN	Dialogic® HMP Software 3.1LIN	Dialogic® HMP Software 4.1LIN
MM_DATA_FORMAT_PCM	Linear PCM, 16-bit "little endian"	S*	S*	S
MM_DATA_FORMAT_ALAW	A-Law PCM	NS	S	S
MM_DATA_FORMAT_MULAW	µ-Law PCM	NS	S	S
MM_DATA_FORMAT_AMR_NB_4_75K	GSM AMR-NB, 4.75 kbps	NS	S**	S
MM_DATA_FORMAT_AMR_NB_5_15K	GSM AMR-NB, 5.15 kbps	NS	S**	S
MM_DATA_FORMAT_AMR_NB_5_90K	GSM AMR-NB, 5.9 kbps	NS	S**	S
MM_DATA_FORMAT_AMR_NB_6_70K	GSM AMR-NB, 6.7 kbps	NS	S**	S
MM_DATA_FORMAT_AMR_NB_7_40K	GSM AMR-NB, 7.4 kbps	NS	S**	S
MM_DATA_FORMAT_AMR_NB_7_95K	GSM AMR-NB, 7.95 kbps	NS	S**	S
MM_DATA_FORMAT_AMR_NB_10_20K	GSM AMR-NB, 10.2 kbps	NS	S**	S
MM_DATA_FORMAT_AMR_NB_12_20K	GSM AMR-NB, 12.2 kbps	NS	S**	S
MM_DATA_FORMAT_G723_1_5_30K	G.723.1, 5.3 kbps	NS	S**	S
MM_DATA_FORMAT_G723_1_6_30K	G.723.1, 6.3 kbps	NS	S**	S
Legend: S = Audio data can be recorded and played in this data format by a non-native record/play and a native record/play. S* = Audio data can be recorded and played in this format by a non-native record/play only. S** = Audio data can be recorded and played in this format by a native record/play only. S*** = Audio data can be natively recorded/played in this format though a 3G-324 interface only. NS = Not Supported - Audio data cannot be recorded or played in this format. Dialogic® HMP Software 3.0WIN = Dialogic® Host Media Processing Software Release 3.0WIN Dialogic® HMP Software 3.1LIN = Dialogic® Host Media Processing Software Release 3.1LIN Dialogic® HMP Software 4.1LIN = Dialogic® Host Media Processing Software Release 4.1LIN				

Table 3. Audio Format Support by Platform (Continued)

Value	Description	Dialogic® HMP Software 3.0WIN	Dialogic® HMP Software 3.1LIN	Dialogic® HMP Software 4.1LIN
MM_DATA_FORMAT_G726	G.726	NS	S**	S
MM_DATA_FORMAT_G729A	G.729A	NS	S**	S
Legend: S = Audio data can be recorded and played in this data format by a non-native record/play and a native record/play. S* = Audio data can be recorded and played in this format by a non-native record/play only. S** = Audio data can be recorded and played in this format by a native record/play only. S*** = Audio data can be natively recorded/played in this format though a 3G-324 interface only. NS = Not Supported - Audio data cannot be recorded or played in this format. Dialogic® HMP Software 3.0WIN = Dialogic® Host Media Processing Software Release 3.0WIN Dialogic® HMP Software 3.1LIN = Dialogic® Host Media Processing Software Release 3.1LIN Dialogic® HMP Software 4.1LIN = Dialogic® Host Media Processing Software Release 4.1LIN				

Note: A native record/play is configured by disabling audio transcoding. Audio transcoding is disabled if the DMFL_TRANSCODE_NATIVE flag is set for audio port connections when the **dev_PortConnect( )** function is called. See the *Dialogic® Device Management API Library Reference* for more information about this function.

Event Handling 3

This chapter provides information about event handling

An event indicates that a specific activity has occurred. Events provide feedback on the progress and completion of functions and indicate the occurrence of other activities. Multimedia API library events are defined in the *mmevts.h* header file.

Refer to Chapter 9, "Events" for a list of events that may be returned by the multimedia software.

Following is an example of how Multimedia API events and errors are handled. The function used in the example is **mm_Play( )**.

A typical application scenario when doing **mm_Play( )** is as follows:

1. The application initiates the multimedia play operation by calling the **mm_Play( )** function.
2. A return value of the call (EMM_SUCCESS or EMM_ERROR) indicates the success or failure of the call. Failure usually indicates that the parameter list is incorrect. No Dialogic® Standard Runtime Library (SRL) events are generated for this.

   If EMM_ERROR is returned, use the **mm_ErrorInfo( )** function to obtain the reason for the failure. This function should be called immediately after the failed **mm_Play( )** call and in the same execution thread. The **mm_ErrorInfo( )** function returns a structure with an error code (one of the EMM_ prefixed values defined in the *mmerrs.h* header file) and character strings with text messages describing the error.
3. Upon initiation of the play operation, either the MMEV_PLAY_ACK or MMEV_PLAY_ACK_FAIL event is posted to the SRL event queue with event data pointing to the MM_PLAY_ACK structure. This structure holds the return code of the operation. Return codes are defined in the *mmerrs.h* header file and have a EMMRC_ prefix. Success is indicated with the EMMRC_OK return code.

   If the play initiation fails, no further processing takes place and no more SRL events are generated for this operation.
4. When the play operation finishes (because the end of data or stop operation is issued), either the MMEV_PLAY or MMEV_PLAY_FAIL event is posted to the SRL queue with event data pointing to the MM_PLAY_CMPLT structure. This structure holds the details of the operation.

It is recommended that you use the **mm_GetMetaEvent( )** function to obtain detailed information about SRL events related to Multimedia API calls. This function returns a structure with all details of the event.

At any point of operation execution, a general error event (MMEV_ERROR) could be posted to the SRL queue. This error indicates that some unexpected system error event occurred. It could indicate a problems such as resources depletion, memory corruption, or internal communication failures.

Error Handling 4

This chapter describes error handling for the multimedia software.

All Dialogic® Multimedia API library functions return a value that indicates the success or failure of the function call. Success is indicated by a return value of EMM_SUCCESS for success or EMM_ERROR for failure. Failure usually indicates that the parameter list is incorrect. No Dialogic® Standard Runtime Library (SRL) events are generated for this.

If a function fails, use the **mm_ErrorInfo( )** function to obtain the reason for the failure. This function should be called immediately after the failed function call and in the same execution thread. It returns a structure with the error code (one of the EMM_ prefixed values defined in the *mmerrs.h* header file) and character strings with test messages describing the error.

At any point of operation execution, a general error event (MMEV_ERROR) could be posted to the SRL queue. This error indicates that some unexpected system error event occurred. It could be indication of a problem such as resource depletion, memory corruption, or internal communication failures.

Refer to Chapter 12, “Error Codes” for a list of error codes.

Application Development Guidelines 5

This chapter provides some guidelines for those choosing to develop multimedia applications. The following topics are included:

- Developing Multimedia Applications 35
- Building Blocks for Multimedia Applications 38
- Call Flow of Video Mail 38
- Requesting I-Frame Using SIP INFO 42
- Making Connections Using Virtual CT Bus and New Packet Bus 44
- Enabling Digit Detection Using DX 48
- Enabling and Using Transcoding 49
- Play a Still Image. 58
- Capture a Still Image 59
- 3GP File Format Direct Playback 59

5.1 Developing Multimedia Applications

This section describes the following multimedia applications:

- Video Mail
- Video Color Ring
- Video Caller ID
- Video Location Based Services

5.1.1 Video Mail

In the context of Dialogic® Host Media Processing (HMP) Software, video mail is an extension of voice mail services. The IP Multimedia Server (IPMS), in conjunction with other network elements, can provide the following capabilities to phone users, both mobile and wired, including soft-phone users:

- a party of an a/v (audio/video) call can leave an a/v message (or greeting)
- a party of an a/v call can be greeted by an a/v message
- a party of an a/v call can retrieve an a/v message
- a party of an a/v call can retrieve an audio only message while being presented a status menu, a still image, or no video

- a party of an a/v call can navigate through a menu, both visual and audio, by pressing phone keys
- a party of an audio only call can retrieve the audio portion of an a/v message

The IPMS, in conjunction with other system elements (primarily the application), can provide the following capabilities to PC web browser users:

- leave an a/v message or greeting
- retrieve an a/v message

The IPMS, along with tools and utilities provided with Dialogic® HMP Software, allows the output (such as a/v files) of popular content creation tools to be used by the IPMS as a source of the a/v content that is streamed to a caller. One use of this a/v content would be to support the creation and use of video menus.

5.1.2 Video Color Ring

Video color ring is a telecom service where instead of standard ring-back tones, a caller receives a specially prepared audio or audio/visual greeting while the called phone is ringing. The greeting is specific to the called number and may also be specific to the calling number. For example, special greetings may be provided to family or friends and a general greeting may be provided to all other callers.

Unlike video mail, where a full duplex, normal media session is established between the IPMS and the calling party, for video color ring, a half duplex, early media session (pre-answer) needs to be established between the IPMS and the calling party.

For this release, Dialogic® HMP Software will not directly support early media between the IPMS and the caller and therefore depends on other network elements, such as a gateway, to provide this capability. Phones that support video color ring are also required.

In a 3G or IP environment, where the IPMS resides behind a gateway, video color ring can be supported without the IPMS directly supporting early media. Early media support can be established to the caller by another subsystem in the network, such as by the gateway. Again, this assumes that the gateway supports early media. A full duplex call between the IPMS and the gateway could be established and the IPMS could provide the IP address and RTP port of the caller, allowing the IPMS to stream the video color ring to the caller. As an alternate approach, call signaling to the IPMS is not even required. The IPMS could be instructed to just stream the appropriate media to a specified IP address and RTP port(s).

Assuming that early media is handled by other network elements and phones supporting video color ring are available, the IPMS can provide the following capabilities in support of the video color ring service:

- set up an a/v ring back greeting or greetings from a phone or a PC web browser
- set up an a/v session and stream an a/v ring back greeting to a specified IP address and port.

5.1.3 Video Caller ID

Video caller identification is a telecom service where instead of the caller's phone number appearing on the called phone, a pre-recorded audio/visual message is sent and presented according to the capability of the called phone.

The capabilities needed to be supported by the IPMS are similar to those required to support video mail and video color ring services (refer to Section 5.1.1, "Video Mail", on page 35 and Section 5.1.2, "Video Color Ring", on page 36).

Similar to the video color ring service, where a half duplex, early media session (pre-answer) is established between the calling party and the IPMS, for video caller ID, a half duplex, early media session (pre-answer) is also established but in this case it is between the IPMS and the called party.

For this Dialogic® HMP Software release, Dialogic® HMP Software will not directly support early media between the IPMS and the called party and therefore depends on other network elements, such as a gateway, to provide this capability. Phones that support video caller ID will also be required. Most phones that support video caller ID today support a different form of video caller ID. These phones allow photos to be stored locally on the phone and the stored picture is associated with a caller ID number and are displayed when a incoming call with one of these numbers arrives at the phone.

In a 3G or IP environment, where the IPMS resides behind a gateway, video caller ID can be supported without the IPMS directly supporting early media. Early media support can be established to the called party by another subsystem in the network, such as by the gateway. Again, this assumes that the gateway supports early media. A full duplex call between the IPMS and the gateway could be established and the IPMS could provide the IP address and RTP port of the called party, allowing the IPMS to stream the video caller ID to the called party. As an alternate approach, call signaling to the IPMS is not even required. The IPMS could be instructed to just stream the appropriate media to a specified IP address and RTP port(s).

Assuming that early media is handled by other network elements and phones supporting this type of video caller ID are available, the IPMS can provide the following capabilities in support of the video caller ID service:

- set up an a/v caller ID greeting (or greetings) from a phone or a PC web browser.
- set up an a/v session and stream an a/v caller ID greeting to a specified IP address and port.

5.1.4 Video Location Based Services

Video location based service is a telecom service where subscribers can access local information such as local weather, restaurant guides and reviews, movie listings, and sports highlight clips. Video content is stored on a central server where it can be accessed by video subscribers.

Like video mail, a full duplex, normal media session is established between the IPMS and the calling party.

The IPMS, in conjunction with other network elements, can provide the following capabilities to phone users, both mobile and wired:

- a party of an a/v call can be greeted by an a/v message
- a party of an a/v call can navigate through a menu by pressing phone keys
- a party of an a/v call can retrieve a/v content

The IPMS, along with tools and utilities provided with Dialogic® HMP Software, will allow the output (such as a/v files) of popular content creation tools to be used by the IPMS as a source of the a/v content that is streamed to a caller.

5.2 Building Blocks for Multimedia Applications

The Dialogic® HMP Software release provides API libraries for developing a variety of applications. The Apes or building blocks for those choosing to develop audio/video applications include the following:

- Dialogic® Global Call API Library for SIP call control (gc_)
- Dialogic® IP Media Library API Library for RTP control (ipm_)
- Dialogic® Multimedia API Library for playing and recording audio/video files (mm_)
- Dialogic® Voice API Library for DTMF detection/generation (dx_)
- Dialogic® Device Management API Library for setting up virtual connections between devices (dev_)
- Session Description Protocol API Library to create and parse SDP bodies

Using the above sets of APIs, the applications that have been discussed in previous sections can be easily developed. The Dialogic® HMP Software release includes a sample demo (multimedia demo) that demonstrates a video mail and a video portal application using the above APIs and their associated libraries.

5.3 Call Flow of Video Mail

This section provides illustrations of a typical call flow and the steps to take if one chooses to develop a video mail application using the Dialogic® Global Call API.

Figure 1 shows a typical scenario of the video mail application.

Figure 1. Typical Call Flow of Video Mail

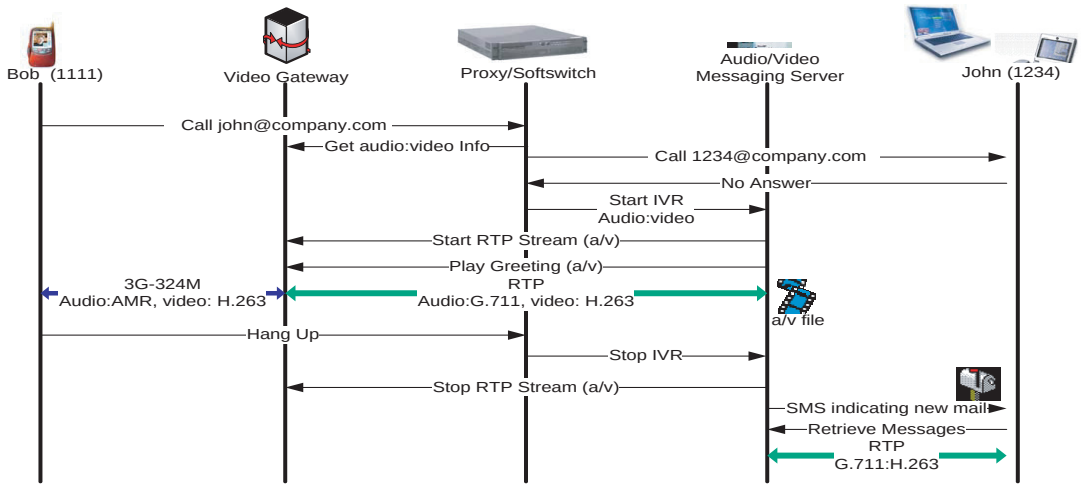


Figure 2 and Figure 3 show the steps to take to develop a video mail application using the Dialogic® Global Call API. Figure 2 shows a scenario in which the SIP message has both audio and video information in the SDP.

Figure 2. Incoming SIP Call with Audio and Video SDP

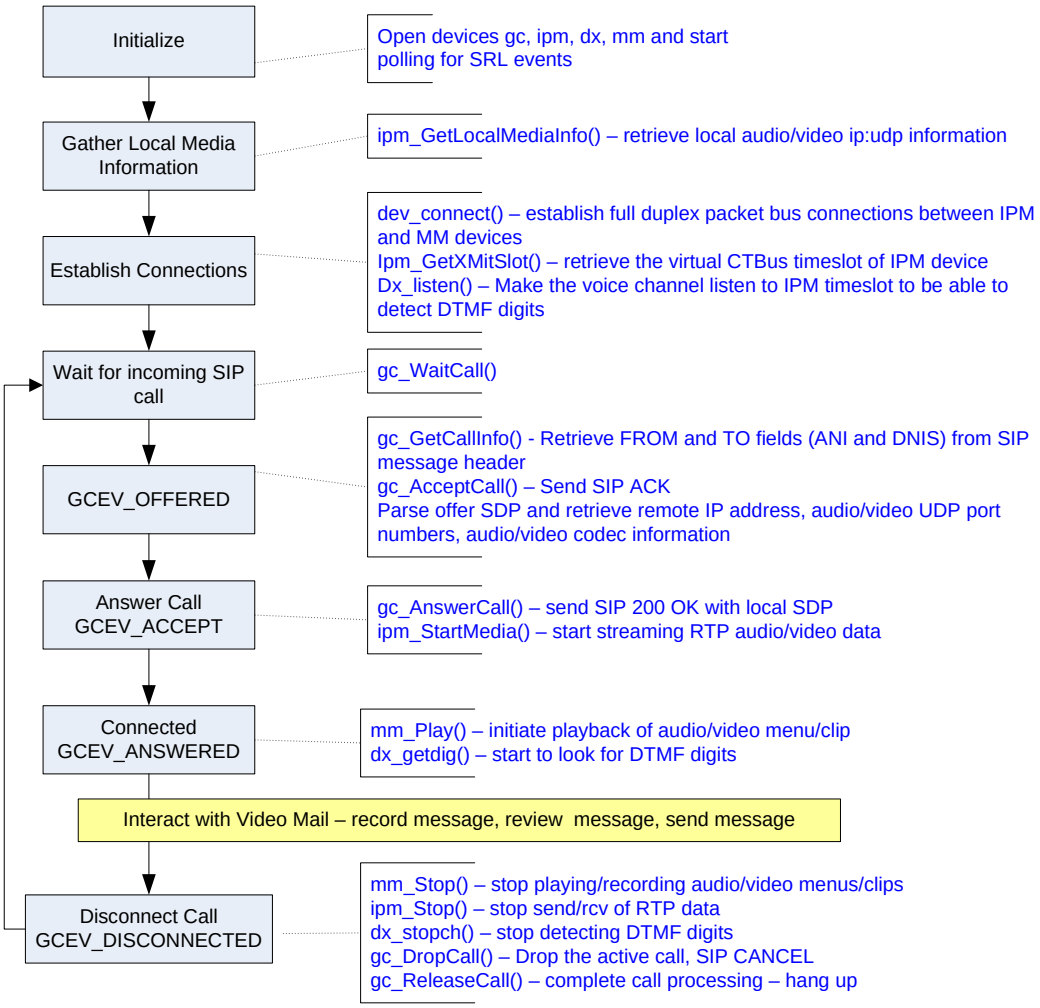
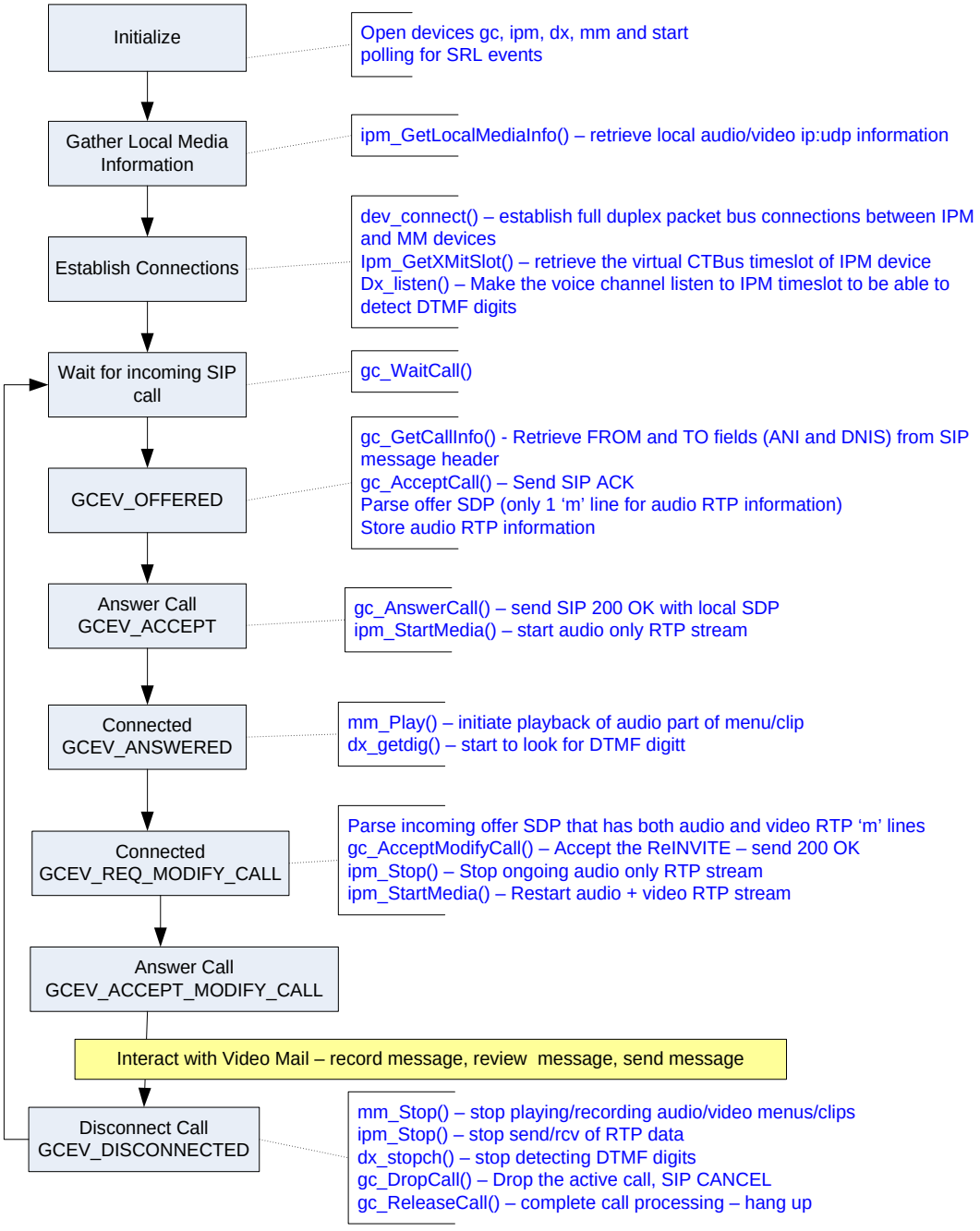


Figure 3 shows a scenario in which the SIP message has only audio first and then with a subsequent re-INVITE the audio plus video information is provided in the SDP.

Figure 3. Incoming SIP Call with Only Audio SDP (video part is sent via re-INVITE)



5.4 Requesting I-Frame Using SIP INFO

As previously discussed in Section 1.1.4, "Intraframe (I-Frame)", on page 21, Intraframe, or I-frame, is a frame that contains complete information about the scene or picture of video. The other types of frames in the video stream that contain only partial information are called P or B-frames. When a SIP User Agent (UA) detects that its video picture quality is being compromised, it may request the source to send an I-frame so that it can refresh the screen and improve picture quality. This mechanism has been defined by the IETF working group mmusic: Multiparty Multimedia Session Control (draft-levin-mmusic-xml-schema-media-control-03). Although the draft never got ratified and has expired, it has been through a few versions and has already been used in the industry: some SIP phone vendors and SIP servers currently implement it.

Using Global Call SIP in 3PCC mode, an application is able to attach the XML schema as the message body to a SIP INFO method and sent that to a SIP endpoint to request an I-frame. Shown below is the XML schema and a code snippet from the multimedia demo that is part of the Dialogic® HMP Software:

```
INFO sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bK7
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
Contact: <sip:alice@pc33.atlanta.com>
Content-Type: application/media_control+xml
Message Body
<?xml version="1.0" encoding="utf-8" ?>
<media_control>
          <vc_primitive>
                <to_encoder>
          <picture_fast_update>
          </picture_fast_update>
                </to_encoder>
          </vc_primitive>
</media_control>
```

```
//*****************************************************************************
//         NAME : bool CMMStream::SendIFrameRequest()
// DESCRIPTION : Sends the IFrame Request
//        INPUT : None
//        OUTPUT : None
//       RETURNS : Bool - True if the function succeeded, False otherwise
//      CAUTIONS : None
//*****************************************************************************
bool CMMStream::SendIFrameRequest()
{
       // Local Variable Declaration
       GC_PARM_BLKP   gcParmBlk_mime = 0;
       GC_PARM_BLKP   gcParmBlk_mime1 = 0;
       GC_PARM_BLKP   gcParmBlk_info = 0;
       bool        bOk            = true;
       char        *pBodyType     = "Content-Type:application/media_control+xml"; // specify the
body type
```

```
        mmReport(INFO_MSG, s_eType, "SendIFrameRequest()");
        if (gc_util_insert_parm_ref(&gcParmBlk_mime,
                IPSET_MIME,
                IPPARM_MIME_PART_TYPE,
                (unsigned char)(strlen(pBodyType) + 1),
                pBodyType) < 0)
        {
                mmReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_util_insert_parm_ref()
failed on %s for IPPARM_MIME_PART_TYPE ", m_devName);
                bOk = false;
        }

        // insert the body size
        if (gc_util_insert_parm_val(&gcParmBlk_mime,
                IPSET_MIME,
                IPPARM_MIME_PART_BODY_SIZE,
                sizeof(unsigned long),
                strlen(c_iFrameRequest)) < 0)
        {
                mmReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_util_insert_parm_val()
failed on %s for IPPARM_MIME_PART_BODY_SIZE ", m_devName);
                bOk = false;
        }

        // insert the body
        if (gc_util_insert_parm_val(&gcParmBlk_mime,
                IPSET_MIME,
                IPPARM_MIME_PART_BODY,
                sizeof(unsigned long),
                (unsigned long)(c_iFrameRequest)) < 0)
        {
                mmReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_util_insert_parm_val()
failed on %s for IPPARM_MIME_PART_BODY ", m_devName);
                bOk = false;
        }

        // insert the list of parmBlks into the top level parmBlk
        if (gc_util_insert_parm_val(&gcParmBlk_mime1,
                IPSET_MIME,
                IPPARM_MIME_PART,
                sizeof(unsigned long),
                (unsigned long)gcParmBlk_mime) < 0)
        {
                mmReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_util_insert_parm_val()
failed on %s for IPPARM_MIME_PART", m_devName);
                bOk = false;
        }



        // now set it on the device
        if (gc_SetUserInfo(GCTGT_GCLIB_CRN,
                m_gcCurrentCrn,
                gcParmBlk_mime1,
                GC_SINGLECALL) < 0) // for this call only
        {
                mmReport(ERROR_GCALL, s_eType, "gc_SetUserInfo() failed on %s for MIME body in
INFO");
                bOk = false;
         }
         // insert the message type
         if (gc_util_insert_parm_val(&gcParmBlk_info,
                 IPSET_MSG_SIP,
                 IPPARM_MSGTYPE,
                 sizeof(int),
                 IP_MSGTYPE_SIP_INFO) < 0)
```

```
        {
               mmReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_util_insert_parm_val()
failed on %s for SIP INFO", m_devName);
               bOk = false;
        }

       if (gc_Extension(GCTGT_GCLIB_CRN, m_gcCurrentCrn, IPEXTID_SENDMSG, gcParmBlk_info, NULL,
EV_ASYNC) < 0)
        {
               mmReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_Extension failed");
               bOk = false;
        }
        gc_util_delete_parm_blk(gcParmBlk_info);
        gc_util_delete_parm_blk(gcParmBlk_mime);

        return bOk;
}
```

The above code snippet can be found in the *mmstream.cpp* file of the multimedia demo application.

5.5 Making Connections Using Virtual CT Bus and New Packet Bus

This section shows several connection scenarios.

Figure 4 shows a connection diagram for a multimedia play and/or record session. Connections are made between the MM (multimedia) Device and the IPM Device using the **dev_Connect( )** API.

Figure 4. Connection Scenario 1

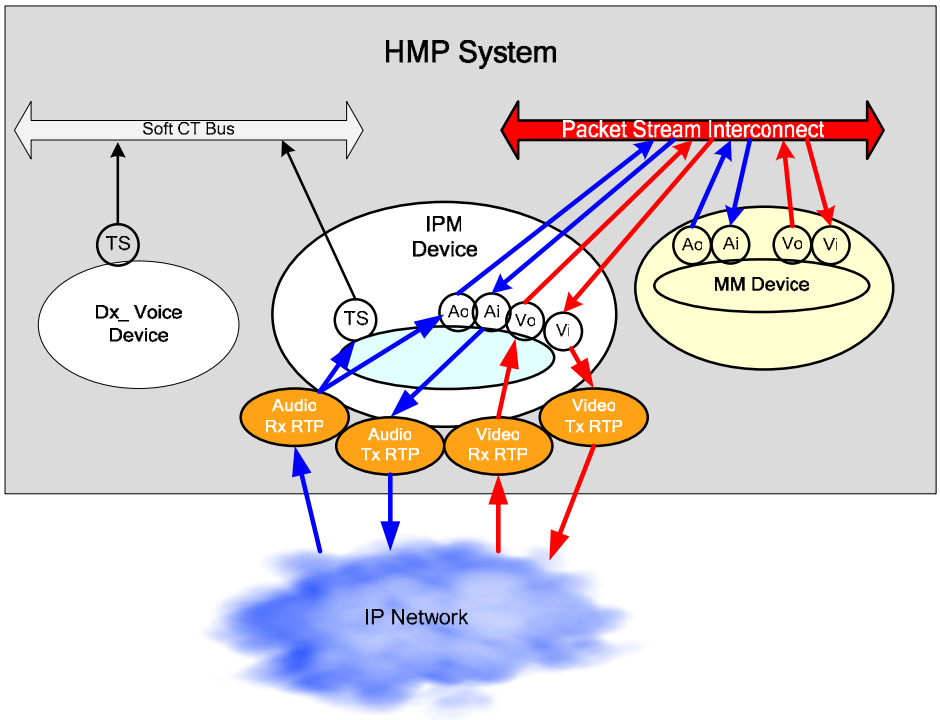


Figure 5 shows a connection diagram for a multimedia play and/or record session in which a dx_ Voice Device is used simultaneously to listen for inband digits received from the audio RTP stream received from the IP Network via the IPM Device. Connections are made between the MM (multimedia) Device and the IPM Device using the **dev_Connect( )** API followed by a **dx_Listen( )** to the IPM Device transmit timeslot.

Figure 5. Connection Scenario 2

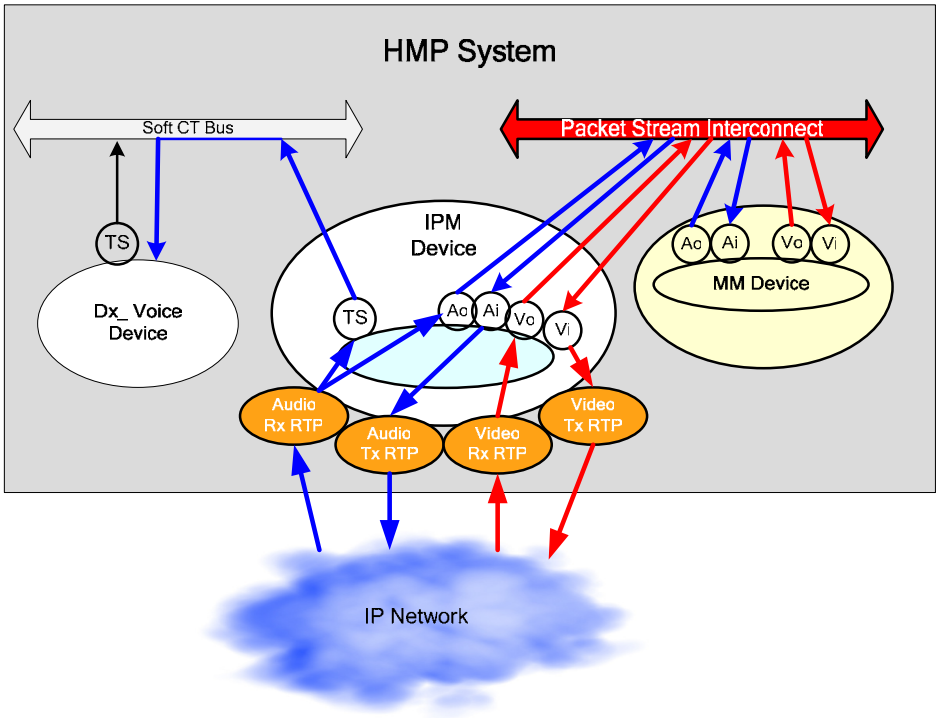
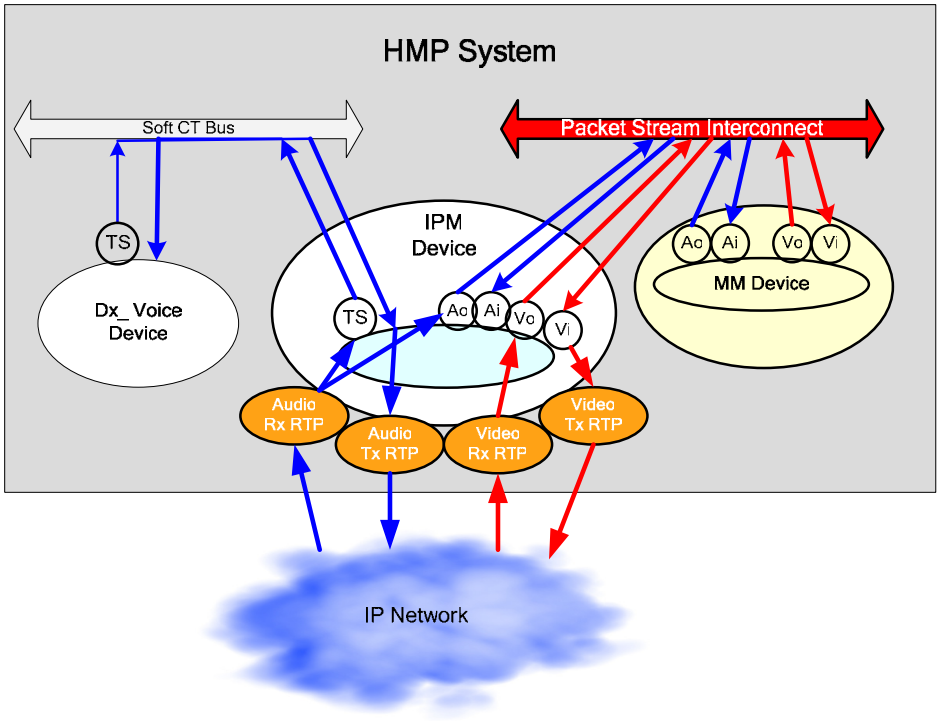


Figure 6 shows a connection diagram for a multimedia play session where the MM (multimedia) Device plays video, the dx_ Voice Device plays audio, and the dx_ Voice Device is also used to listen for inband digits received from the audio RTP stream received from the IP Network via the IPM Device. Connections are first made between the MM Device and the IPM Device using the **dev_Connect( )** API followed by a **dx_Listen( )** to the IPM Device transmit timeslot and an **ipm_Listen( )** to the dx_ Voice Device transmit timeslot. To satisfy this use case, the order of connections *must* be **dev_Connect( )** followed by the **ipm_Listen( )**.

Note: The audio data sent from the audio Tx RTP port of the IPM Device to the IP Network depends on the last active connection established. If an **ipm_UnListen( )** is now called, the source of the audio data will become the MM Device again.

Figure 6. Connection Scenario 3



Note: If calls are recorded using a dx device, you can still play these recordings to a multimedia (audio/video) session. When you do this, the video is not tightly synchronized with the audio since the audio was recorded separately, but you may wish to play a video menu or status display while playing back the audio message. You can do this by making a dev_connect between the IPM device and the MM Device and follow this with an **ipm_Listen( )** to a dx device. Then play the video file using the MM Device and the audio file using the dx_ Voice Device. If you then do an **ipm_UnListen( )**, the audio connection to the IPM device (towards IP) will reconnect to the MM Device as the audio source.

5.6 Enabling Digit Detection Using DX

DTMF digits can be detected using the Dialogic® Voice API **dx_getdig( )**. However, to specify the type of digits (inband or RFC 2833), the application should use **ipm_SetParm( )** to specify the type of digits. This information can be retrieved in the SDP body that is provided in the incoming SIP INVITE from the SIP endpoint. The following code snippet shows how this is done in the multimedia demo (the sample program provided in the Dialogic® HMP Software release).

Note: For more information about the multimedia demo, refer to the *Dialogic® Multimedia Demo Guide*.

```
//*************************************************************************
//                NAME : bool CMMStream::SetDtmfMode(dtmfMode_e eMode)
// DESCRIPTION : Sets the DTMF XFer Mode based on the input
//               INPUT : eMode - Object of type dtmfMode containing the input
//          OUTPUT : None
//         RETURNS : Bool - True if function succeeds, False otherwise
//        CAUTIONS : None
//*************************************************************************
bool CMMStream::SetDtmfMode(dtmfMode_e eMode)
{
        // Local Variable Declaration
        IPM_PARM_INFO           parmInfo;
            eIPM_DTMFXFERMODE       value;
        int                   PLType  = 0;
        switch (eMode)
        {

            case dtmfMode_rfc2833:
            {
                 // set the dtmf mode to RFC2833
                            value                                   = DTMFXFERMODE_RFC2833;
                 parmInfo.eParm                 = PARMCH_DTMFXFERMODE;
                            parmInfo.pvParmValue    = &value;

                            mmReport(INFO_MSG, s_eType, "[%s] Setting Parameter
PARMCH_DTMFXFERMODE to DTMFXFERMODE_RFC2833", m_ipmDevName);

                 if (ipm_SetParm(m_ipmH, &parmInfo, EV_SYNC) < 0)
                            {
                   mmReport(ERROR_IPM, s_eType, "ipm_SetParm() on %s", m_ipmDevName);
                            }

                 // set the TX Payload Type

                 PLType                                 = m_nRfc2833PayloadType;
                 parmInfo.eParm                 = PARMCH_RFC2833EVT_TX_PLT;
                 parmInfo.pvParmValue    = &PLType;

                mmReport(INFO_MSG, s_eType, "[%s] Setting Parameter PARMCH_RFC2833EVT_TX_PLT to
%d", m_ipmDevName, m_nRfc2833PayloadType);

                 if (ipm_SetParm(m_ipmH, &parmInfo, EV_SYNC) < 0)
                 {
                   mmReport(ERROR_IPM, s_eType, "ipm_SetParm() on %s", m_ipmDevName);
                 }

                   // set the RX Payload Type
                   PLType                                 = m_nRfc2833PayloadType;
                   parmInfo.eParm                 = PARMCH_RFC2833EVT_RX_PLT;
                   parmInfo.pvParmValue = &PLType;

                  mmReport(INFO_MSG, s_eType, "[%s] Setting Parameter PARMCH_RFC2833EVT_RX_PLT
to %d", m_ipmDevName, m_nRfc2833PayloadType);
```

```
                    if (ipm_SetParm(m_ipmH, &parmInfo, EV_SYNC) < 0)
                    {
                       mmReport(ERROR_IPM, s_eType, "ipm_SetParm() on %s", m_ipmDevName);
                    }
                }
                break;

                case dtmfMode_inband:
                {
                        eIPM_DTMFXFERMODE value = DTMFXFERMODE_INBAND;
                        parmInfo.eParm                  = PARMCH_DTMFXFERMODE;
                        parmInfo.pvParmValue    = &value;

                        mmReport(INFO_MSG, s_eType, "[%s] Setting Parameter PARMCH_DTMFXFERMODE
to DTMFXFERMODE_INBAND", m_ipmDevName);

                        if (ipm_SetParm(m_ipmH, &parmInfo, EV_SYNC) < 0)
                        {
                             mmReport(ERROR_IPM, s_eType, "ipm_SetParm() on %s", m_ipmDevName);

                        }
                }
                break;
        }
        return true;
}
```

5.7 Enabling and Using Transcoding

This section provides high-level guidelines on enabling and using transcoding in your application. You will use API functions from the Dialogic® Multimedia API library, Dialogic® IP Media Library API library, and Dialogic® Device Management API library.

1. Specify coder options through **mm_Play( )**, **mm_Record( )**, and **ipm_StartMedia( )**.
2. Set the DMFL_TRANSCODE_ON flag in the DM_PORT_CONNECT_INFO structure to enable transcoding. Transcoding is invoked when connections are formed between devices, such as between MM and IPM devices.
3. If digit terminated runtime control (RTC) on a transcoded connection is desired, set the EMM_TONE_DETECTION parameter in the MM_SET_PARM structure using **mm_SetParm( )**. This parameter enables/disables detection of DTMF tones in the incoming audio stream on the given device.

Setting Video Coder Parameters

This section describes how to set video coder parameters in **mm_Play( )**, **mm_Record( )**, and **ipm_StartMedia( )**.

Table 4, “Video Coder Parameters for Decode Operation”, on page 50 applies to **ipm_StartMedia(**MEDIATYPE_VIDEO_LOCAL_CODER_INFO**)** to decode incoming video stream or to **mm_Play( )** to decode a video file being played.

Table 4. Video Coder Parameters for Decode Operation

Field	If transcoding is disabled	If transcoding is enabled
Coding	Field ignored. Use default value initialized by inline function.	Used to decode incoming steam or file being played.
Profile		For MPEG-4, specifies both profile and level.
Level		For MPEG-4, field ignored. Use default value initialized by inline function.
Width		Sets image width of decoded output. If image width of file being played or incoming video stream is different from the application setting, image will be resized in decoded output.
Height		Sets image height of decoded output. If image height of file being played or incoming video stream is different from the application setting, image will be resized in decoded output.
Bitrate		Field ignored. Use default value initialized by inline function.
Frames per second		Field ignored. Use default value initialized by inline function.
Sampling rate		Field ignored. Use default value initialized by inline function.
VisualConfiguration		Applies to MEPG-4 only. Used to decode incoming stream or file being played. Application settings are used only if the incoming stream or the file being played does not contain VisualConfiguration data. If the application has not specified this data either, an error will be reported.

Table 5, “Video Coder Parameters for Encode Operation”, on page 51 applies to **ipm_StartMedia**(MEDIATYPE_VIDEO_REMOTE_CODER_INFO) to encode outgoing video stream or to **mm_Record( )** to record encoded video file.

Table 5. Video Coder Parameters for Encode Operation

Field	If transcoding is disabled	If transcoding is enabled
Coding	Application settings are embedded in the outgoing stream or the file being recorded.	Application settings determine the encoded output.
Profile		
Level		
Width		
Height		
Bitrate		
Frames per second		
Sampling rate		
VisualConfiguration	Applies to MPEG-4 only. If available, VisualConfiguration data from the incoming stream is embedded in the outgoing stream or recorded file. If not available and the application has not specified this data either, an error will be reported.	Applies to MPEG-4 only. Application setting not used. VisualConfiguration data string is automatically created and embedded in the outgoing stream or recorded file.

The following example code illustrates transcoding.

```
#include <ipmlib.h>
#include <mmlib.h>
#include <devmgmt.h>
#include <port_connect.h>

#define PLAY  0x0004
#define RECORD  0x0008

/*****************************************************************************************/
int getportindex(DM_PORT_INFO_LIST portlist, DM_PORT_MEDIA_TYPE porttype)
{
    for (int i = 0; i < portlist.unCount; i++)
    {
        if (portlist.port_info[i].port_media_type == porttype)
            return i;
    }
    return -1;
}

/*****************************************************************************************/
void add_audio_item(PMM_MEDIA_ITEM_LIST AudioItemList, int *AudioItemCountp, MM_AUDIO_CODEC
AudioCodec, char *AudioFileName)
{
    int index = *AudioItemCountp;

    INIT_MM_MEDIA_ITEM_LIST (&AudioItemList[index]);
    INIT_MM_MEDIA_AUDIO(&(AudioItemList[index].item.audio));

    if (index >= 1)
    {
        AudioItemList[index-1].ItemChain = EMM_ITEM_CONT;
    }
```

```
    AudioItemList[index].ItemChain = EMM_ITEM_EOT;
    AudioItemList[index].item.audio.codec = AudioCodec;
    AudioItemList[index].item.audio.unMode = 0;
    AudioItemList[index].item.audio.unOffset = 0;
    AudioItemList[index].item.audio.szFileName = AudioFileName;
    AudioItemList[index].item.audio.eFileFormat = EMM_AUD_FILEFORMAT_VOX;
    AudioItemList[index].item.audio.unAccessMode = MM_MEDIA_ACCESS_MODE_FILE;

    (*AudioItemCountp)++;
}

/*****************************************************************************************/
void add_video_item(PMM_MEDIA_ITEM_LIST VideoItemList, int *VideoItemCountp, MM_VIDEO_CODEC
VideoCodec, char *VideoFileName)
{
    int index = *VideoItemCountp;

    INIT_MM_MEDIA_ITEM_LIST (&(VideoItemList[index]));
    INIT_MM_MEDIA_VIDEO(&(VideoItemList[index].item.video));

    if (index >= 1)
    {
        VideoItemList[index-1].ItemChain = EMM_ITEM_CONT;
    }

    VideoItemList[index].ItemChain = EMM_ITEM_EOT;
    VideoItemList[index].item.video.codec = VideoCodec;
    VideoItemList[index].item.video.unMode = 0;
    VideoItemList[index].item.video.szFileName = VideoFileName;
    VideoItemList[index].item.video.eFileFormat = EMM_FILE_FORMAT_PROPRIETARY;
    VideoItemList[index].item.video.unAccessMode = MM_MEDIA_ACCESS_MODE_FILE;

    (*VideoItemCountp)++;
}

/*****************************************************************************************/
void add_playrecordlist_item(PMM_PLAY_RECORD_LIST PlayRecordList, int *PlayRecordListItemCount,
CPMM_MEDIA_ITEM_LIST PlayRecordListItem, eMM_MEDIA_TYPE MediaType)
{
    int index = *PlayRecordListItemCount;
    INIT_MM_PLAY_RECORD_LIST(&(PlayRecordList[index]));

    if (index >= 1)
    {
        PlayRecordList[index-1].ItemChain = EMM_ITEM_CONT;
    }

    PlayRecordList[index].ItemChain = EMM_ITEM_EOT;
    PlayRecordList[index].ItemType= MediaType;
    PlayRecordList[index].list= PlayRecordListItem;

    (*PlayRecordListItemCount)++;
}

void mpeg4_transcode_sample()
{
    DM_PORT_INFO_LIST mmportinfo_tx;
    DM_PORT_INFO_LIST mmportinfo_rx;
    DM_PORT_INFO_LIST ipmportinfo_tx;
    DM_PORT_INFO_LIST ipmportinfo_rx;

    DM_PORT_CONNECT_INFO_LIST mmConnList;
    DM_PORT_CONNECT_INFO_LIST ipmConnList;

    int mmdevh, ipmdevh;
```

```
int mm_arx, mm_vrx, ipm_arx, ipm_vrx, mm_atx, mm_vtx, ipm_atx, ipm_vtx;
int count=0;

/****************************** open devices ***********************************/
ipmdevh = ipm_Open("ipmB1C1", 0, EV_ASYNC);
mmdevh = mm_Open("mmB1C1", 0, NULL);

/*.
.
.

  wait for MMEV_OPEN & IPMEV_OPEN
*/

/****************************** get port information ****************************/
dev_GetTransmitPortInfo(ipmdevh, &ipmportinfo_tx);
dev_GetReceivePortInfo(ipmdevh, &ipmportinfo_rx);
dev_GetTransmitPortInfo(mmdevh, &mmportinfo_tx);
dev_GetReceivePortInfo(mmdevh, &mmportinfo_rx);

/*.
.
.

  wait for DMEV_GET_TX_PORT_INFO & DMEV_GET_RX_PORT_INFO on both ipm & mm devices
*/

mm_atx = getportindex(mmportinfo_tx,DM_PORT_MEDIA_TYPE_AUDIO);
mm_vtx = getportindex(mmportinfo_tx,DM_PORT_MEDIA_TYPE_VIDEO);

mm_arx = getportindex(mmportinfo_rx,DM_PORT_MEDIA_TYPE_AUDIO);
mm_vrx = getportindex(mmportinfo_rx,DM_PORT_MEDIA_TYPE_VIDEO);

ipm_atx = getportindex(ipmportinfo_tx,DM_PORT_MEDIA_TYPE_AUDIO);
ipm_vtx = getportindex(ipmportinfo_tx,DM_PORT_MEDIA_TYPE_VIDEO);

ipm_arx = getportindex(ipmportinfo_rx,DM_PORT_MEDIA_TYPE_AUDIO);
ipm_vrx = getportindex(ipmportinfo_rx,DM_PORT_MEDIA_TYPE_VIDEO);

if ((mm_atx == -1)  || (mm_vtx == -1)  || (mm_arx == -1)  || (mm_vrx == -1) ||
    (ipm_atx == -1) || (ipm_vtx == -1) || (ipm_arx == -1) || (ipm_vrx == -1))
{
    printf("ERROR: a port index is -1\n");
}


/****************************** connect MM ports to IPM ***************************/
INIT_DM_PORT_CONNECT_INFO_LIST(&mmConnList);
count=0;

/* set up MM audio tx to IPM port connections with transcoding enabled */
mmConnList.port_connect_info[count].unFlags = DMFL_TRANSCODE_ON;
mmConnList.port_connect_info[count].port_info_tx = mmportinfo_tx.port_info[mm_atx];
mmConnList.port_connect_info[count].port_info_rx = ipmportinfo_rx.port_info[ipm_arx];
count++;

/* set up MM video tx to IPM port connections with transcoding enabled */
mmConnList.port_connect_info[count].unFlags = DMFL_TRANSCODE_ON;
mmConnList.port_connect_info[count].port_info_tx = mmportinfo_tx.port_info[mm_vtx];
mmConnList.port_connect_info[count].port_info_rx = ipmportinfo_rx.port_info[ipm_vrx];
count++;

mmConnList.unCount = count;

/* make the MM tx connections */
dev_PortConnect(mmdevh, &mmConnList, NULL);
```

```
    /*.
    .
    .

      wait for DMEV_PORT_CONNECT events
    */

    /****************************** connect IPM ports to MM ***************************/
    INIT_DM_PORT_CONNECT_INFO_LIST(&ipmConnList);
    count = 0;

    /* set up IPM audio tx to MM port connections with transcoding enabled */
    ipmConnList.port_connect_info[count].unFlags = DMFL_TRANSCODE_ON;
    ipmConnList.port_connect_info[count].port_info_tx = ipmportinfo_tx.port_info[ipm_atx];
    ipmConnList.port_connect_info[count].port_info_rx = mmportinfo_rx.port_info[mm_arx];
    count++;

    /* set up IPM video tx to MM port connections with transcoding enabled */
    ipmConnList.port_connect_info[count].unFlags = DMFL_TRANSCODE_ON;
    ipmConnList.port_connect_info[count].port_info_tx = ipmportinfo_tx.port_info[ipm_vtx];
    ipmConnList.port_connect_info[count].port_info_rx = mmportinfo_rx.port_info[mm_vrx];
    count++;

    ipmConnList.unCount = count;


    /* make the MM tx connections */
    dev_PortConnect(ipmdevh, &ipmConnList, NULL);

    /*.
    .
    .

      wait for DMEV_PORT_CONNECT events
    */

    /****************************** start audio/video RTP streaming ***********************/
    int rc;

    IPM_MEDIA_INFO MediaInfo;
    unsigned int rartpPort, rartcpPort, rvrtpPort, rvrtcpPort;
    char rartpAddress[IP_ADDR_SIZE], rartcpAddress[IP_ADDR_SIZE], rvrtpAddress[IP_ADDR_SIZE],
rvrtcpAddress[IP_ADDR_SIZE];

    /*.
    .
    .

      get remote side's port ids for:
      audio rtp (rartpPort),
      audio rtcp (rartcpPort),
      video rtp (rvrtpPort),
      video rtcp (rvrtcpPort)

        get remote side's IP address for:
        audio rtp (rartpAddress),
        audio rtcp (rartcpAddress),
        video rtp (rvrtpAddress),
        video rtcp (rvrtcpAddress)

    */

    count = 0;

    MediaInfo.MediaData[count].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTP_INFO;
    MediaInfo.MediaData[count].mediaInfo.PortInfo.unPortId = rartpPort;
    strcpy(MediaInfo.MediaData[count].mediaInfo.PortInfo.cIPAddress, rartpAddress);
```

```
count++;

MediaInfo.MediaData[count].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTCP_INFO;
MediaInfo.MediaData[count].mediaInfo.PortInfo.unPortId = rartcpPort;
strcpy(MediaInfo.MediaData[count].mediaInfo.PortInfo.cIPAddress, rartcpAddress);
count++;

MediaInfo.MediaData[count].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.eFrameSize = CODER_FRAMESIZE_20;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.unCoderPayloadType = 0;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.unRedPayloadType = 0;
count++;

MediaInfo.MediaData[count].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.eFrameSize = CODER_FRAMESIZE_20;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.unCoderPayloadType = 0;
MediaInfo.MediaData[count].mediaInfo.CoderInfo.unRedPayloadType = 0;
count++;

MediaInfo.MediaData[count].eMediaType = MEDIATYPE_VIDEO_REMOTE_RTP_INFO;
MediaInfo.MediaData[count].mediaInfo.PortInfo.unPortId = rvrtpPort;
strcpy(MediaInfo.MediaData[count].mediaInfo.PortInfo.cIPAddress, rvrtpAddress);
count++;

MediaInfo.MediaData[count].eMediaType = MEDIATYPE_VIDEO_REMOTE_RTCP_INFO;
MediaInfo.MediaData[count].mediaInfo.PortInfo.unPortId = rvrtcpPort;
strcpy(MediaInfo.MediaData[count].mediaInfo.PortInfo.cIPAddress, rvrtcpAddress);
count++;

/* For MEDIATYPE_VIDEO_LOCAL_CODER_INFO:
        1. When video transcoding is enabled, no need to specify VisualConfiguration info
           (HMP automatically generates it and inserts into every outgoing I-Frame

        2. if coder settings do not match received image  (in this case, the mm_Play()
           settings), the IPM device will resize the video image going out on the network
*/
IPM_VIDEO_CODER_INFO_EX ExtraCoderInfo1;
INIT_IPM_VIDEO_CODER_INFO_EX(&ExtraCoderInfo1);
ExtraCoderInfo1.eProfile = VIDEO_PROFILE_LEVEL_SP0_MPEG4; // simple profile, level 0
ExtraCoderInfo1.eImageWidth = VIDEO_IMAGE_WIDTH_128; // Sub-QCIF
ExtraCoderInfo1.eImageHeight = VIDEO_IMAGE_HEIGHT_96; // Sub-QCIF
ExtraCoderInfo1.unBitRate = 64000; // 64 kbps
ExtraCoderInfo1.eFramesPerSec = VIDEO_FRAMESPERSEC_6; // 6 fps

MediaInfo.MediaData[count].eMediaType = MEDIATYPE_VIDEO_REMOTE_CODER_INFO;
INIT_IPM_VIDEO_CODER_INFO(&MediaInfo.MediaData[count].mediaInfo.VideoCoderInfo);
MediaInfo.MediaData[count].mediaInfo.VideoCoderInfo.eCoderType = CODER_TYPE_MP4V_ES;
MediaInfo.MediaData[count].mediaInfo.VideoCoderInfo.unCoderPayloadType  = 0;
MediaInfo.MediaData[count].mediaInfo.VideoCoderInfo.pExtraCoderInfo  = &ExtraCoderInfo1;
count++;


/* For MEDIATYPE_VIDEO_LOCAL_CODER_INFO:
        1. the following fields do not neet to be specified for MPEG4
                unBitRate (not applicable)
                eFramesPerSec (not applicable)
                eLevel (profile & level in Profile field for MPEG4)

        2. VisualConfiguration info from the app (MEDIATYPE_VIDEO_LOCAL_CODER_INFO) is
           used by the IPM device for decoding only if this information is not already
           present in the incoming video stream
```

```
        3. if coder settings do not match received image  (incoming video stream
           from network), the IPM device will resize the video image

*/

/*
   VisualConfiguration string below (encoded as per the ISO standard ISO/IEC 14496-2)
   indicates the following:

     Simple Profile Level 0, Image Height/Width = QCIF
*/

unsigned char DCI[ ] = {    0x00, 0x00, 0x01, 0xb0, 0x08, 0x00, 0x00, 0x01,
                            0xb5, 0x09, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
                            0x01, 0x20, 0x00, 0x84, 0x5d, 0x4c, 0x28, 0x2c,
                            0x20, 0x90, 0xa2, 0x8f
                       };
IPM_VIDEO_CODER_INFO_EX ExtraCoderInfo2;
INIT_IPM_VIDEO_CODER_INFO_EX(&ExtraCoderInfo2);
ExtraCoderInfo2.eProfile = VIDEO_PROFILE_LEVEL_SP1_MPEG4;  // simple profile, level 1
ExtraCoderInfo2.eImageWidth = VIDEO_IMAGE_WIDTH_176; // QCIF
ExtraCoderInfo2.eImageHeight = VIDEO_IMAGE_HEIGHT_144; // QCIF
ExtraCoderInfo2.unVisualConfigSize   = sizeof(DCI);
ExtraCoderInfo2.szVisualConfiguration = DCI;

MediaInfo.MediaData[count].eMediaType = MEDIATYPE_VIDEO_LOCAL_CODER_INFO;
INIT_IPM_VIDEO_CODER_INFO(&MediaInfo.MediaData[count].mediaInfo.VideoCoderInfo);
MediaInfo.MediaData[count].mediaInfo.VideoCoderInfo.eCoderType = CODER_TYPE_MP4V_ES;
MediaInfo.MediaData[count].mediaInfo.VideoCoderInfo.unCoderPayloadType  = 0;
MediaInfo.MediaData[count].mediaInfo.VideoCoderInfo.pExtraCoderInfo = &ExtraCoderInfo2;
count++;

MediaInfo.unCount = count;

rc = ipm_StartMedia(ipmdevh, &MediaInfo, DATA_IP_TDM_BIDIRECTIONAL, EV_ASYNC);

/*.
    .
    .

     wait for IPMEV_STARTMEDIA event
*/

/****************************** start playing/recording multimedia***********************/
MM_PLAY_RECORD_INFO PlayRecordInfo;
MM_PLAY_RECORD_LIST PlayRecordList[2];
MM_MEDIA_ITEM_LIST AudioItemList[1];
MM_MEDIA_ITEM_LIST VideoItemList[1];

MM_AUDIO_CODEC AudioCodec;
MM_VIDEO_CODEC VideoRecordCodec;
MM_VIDEO_CODEC VideoPlayCodec;

//Set PlayRecordOption to choose between mm_Play() and mm_Record()
int PlayRecordOption = PLAY;  //set to RECORD if recording

INIT_MM_AUDIO_CODEC (&AudioCodec);
INIT_MM_VIDEO_CODEC (&VideoRecordCodec);
INIT_MM_VIDEO_CODEC (&VideoPlayCodec);

/*
   VisualConfiguration string below (encoded as per the ISO standard ISO/IEC 14496-2)
   indicates the following:

     Simple Profile Level 2, Image Height/Width = CIF
*/
```

```
    unsigned char PlayDCI[ ] = {    0x00, 0x00, 0x01, 0xb0, 0x02, 0x00, 0x00, 0x01,
                                    0xb5, 0x09, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
                                    0x01, 0x20, 0x00, 0x84, 0x5d, 0x4c, 0x28, 0x58,
                                    0x21, 0x20, 0xa2, 0x8f
                               };

    /*  1. The following fields are not applicable for mm_Play():
              BitRate
              FramesPerSec

        2. When video transcoding is enabled, no need to specify VisualConfiguration info
           for mm_Record(), HMP automatically generates it and inserts into every I-Frame in
           recorded file

        3. The following fields are not applicable for MPEG4 (mm_Play nor mm_Record):
              Level (profile & level in Profile field for MPEG4)

        4. App settings for VisualConfiguration in mm_Play will be used only if the play file
           does not already contain this information

        5. The following fields don't need to be explicitly set once the INIT macros are called:
              SamplingRate (the INIT macros already set it to the one valid value)
    */

    /* video play settings (image will be resized by MM device if file being played is not QCIF)
*/
    VideoPlayCodec.Coding = VIDEO_CODING_MP4V_ES;
    VideoPlayCodec.Profile = VIDEO_PROFILE_LEVEL_SP2_MPEG4;
    VideoPlayCodec.ImageWidth = VIDEO_IMAGE_WIDTH_176;   //QCIF
    VideoPlayCodec.ImageHeight = VIDEO_IMAGE_HEIGHT_144; //QCIF
    VideoPlayCodec.VisualConfigSize = sizeof(PlayDCI);
    VideoPlayCodec.VisualConfiguration = PlayDCI;  // will only be used for decoding of MPEG4
           // file if file being played does not contain visual config info

    /* video record settings (image will be resized by MM device since
       received image (in this case, settings in
       ipm_StartMedia(MEDIATYPE_VIDEO_LOCAL_CODER_INFO)) is not CIF)
     */
    VideoRecordCodec.Coding = VIDEO_CODING_MP4V_ES;
    VideoRecordCodec.Profile = VIDEO_PROFILE_LEVEL_SP2_MPEG4;
    VideoRecordCodec.ImageWidth = VIDEO_IMAGE_WIDTH_352;   //CIF
    VideoRecordCodec.ImageHeight = VIDEO_IMAGE_HEIGHT_288; //CIF
    VideoRecordCodec.BitRate = (eVIDEO_BITRATE)64000;
    VideoRecordCodec.FramesPerSec = VIDEO_FRAMESPERSEC_10;

    char VideoFileName[] = "clip1a.vid";
    char AudioFileName[] = "clip1a.pcm";

    int AudioItemCount = 0;
    int VideoItemCount = 0;
    int PlayRecordListItemCount = 0;

    add_audio_item(AudioItemList, &AudioItemCount, AudioCodec, AudioFileName);
    add_playrecordlist_item(PlayRecordList, &PlayRecordListItemCount, AudioItemList,
        EMM_MEDIA_TYPE_AUDIO);

    if (PlayRecordOption == PLAY)
    {
        add_video_item(VideoItemList, &VideoItemCount, VideoPlayCodec, VideoFileName);
    }
    else
    {
        add_video_item(VideoItemList, &VideoItemCount, VideoRecordCodec, VideoFileName);
    }

    add_playrecordlist_item(PlayRecordList, &PlayRecordListItemCount, VideoItemList,
        EMM_MEDIA_TYPE_VIDEO);
```

```
    PlayRecordInfo.list = PlayRecordList;

    if (PlayRecordOption == PLAY)
    {
        rc = mm_Play(mmdevh, &PlayRecordInfo, NULL, NULL);
    }
    else
    {
        rc = mm_Record(mmdevh, &PlayRecordInfo, NULL, NULL);
    }


    /*.
    .
    .

      wait for MMEV_PLAY or MMEV_RECORD event
    */
}
/*****************************************************************************************/
int main(int argc, char *argv[])
{
    mpeg4_transcode_sample();
    return 0;
}
```

5.8 Play a Still Image

This features provides the capability of playing a still image over a video stream.

A still image play can only be performed after a connection is set up using the **dev_PortConnect( )** function between an outbound endpoint device, such as an IP media device, and a multimedia device with transcoding ON. Refer to Section 5.7, "Enabling and Using Transcoding", on page 49 and the **dev_PortConnect( )** function in the Dialogic® Device Management API Library Reference for details about turning transcoding ON when connecting an IP media device to another device.

To play a still image, the application sets ItemType in the MM_PLAY_RECORD_LIST structure to EMM_MEDIA_TYPE_IMAGE and specifies details of the image such as image format (YUV or JPEG), format details (such as height or width) and type of I/O access through the MM_MEDIA_IMAGE structure.

Notes: ***1.*** For still image play, the unNumberOfBytes field in the event data for MMEV_PLAY, MM_PLAY_RECORD_CMPLT_DETAILS, will be the total byte count of the image data. For a JPEG image, the total count will not include the header data size.

2. The input YUV 4:2:0 or JPEG image file size should not exceed 152,064 bytes. This limit is based upon the maximum file size required to support CIF (352x288) frames in YUV 4:2:0 format. A compressed JPEG input file may contain a frame with a resolution larger than CIF (352x288) as long as the file size is less than 152,064 bytes

5.9 Capture a Still Image

This features provides the capability of capturing a still image from a video stream that is currently playing.

A still image capture can only be performed after a connection is set up using the **dev_PortConnect( )** function between an outbound endpoint device, such as an IP media device, and a multimedia device with transcoding ON. Refer to Section 5.7, "Enabling and Using Transcoding", on page 49 and the **dev_PortConnect( )** function in the Dialogic® Device Management API Library Reference for details about turning transcoding ON when connecting an IP media device to another device.

Before a still image can be captured, a video play must first be initiated and then paused. To capture a still image, the application pauses a currently playing video stream using the **mm_Pause( )** function and then calls the **mm_Capture( )** function. Specifics of the image to be captured such as image format (YUV), format details (height or width) and type of I/O access are to be provided in the MM_MEDIA_IMAGE structure of the **mm_Capture( )** function.

The application will receive an MMEV_CAPTURE event upon successful completion of the capture. The associated event data is an MM_CAPTURE_CMPLT_DETAILS structure.

Note: For still image capture the unWidth and unHeight fields in the event data for MMEV_CAPTURE, MM_CAPTURE_CMPLT_DETAILS, will be the width and height of the captured image.

Refer to **mm_Capture( )** for a code example of capturing a still image.

5.10 3GP File Format Direct Playback

This feature enables Dialogic® products to support the playback of contiguous multimedia files in an ISO/IEC standardized structure, known as 3GPP or 3GP file format. A 3GP file (suffix ".3gp") is a multimedia (audio and video) storage container format defined by the 3GPP standards body for transmitting and transferring media content between 3G cell phones and 3G network entities over the Internet. It is structured to facilitate the streaming of multimedia data to 3G and IP network entities and is commonly used by 3G compatible phones in 3G mobile networks.

The 3GP file format stores video streams and audio streams as encoded data tracks. Files describe image sizes, bandwidth, and other characteristics of the media, so that content is correctly sized for display screens when streamed to the intended network endpoint.

This feature also enables direct playback of 3GP files via the Dialogic® Multimedia API library. The file must be compliant with the 3GPP standard and meet the guidelines described in Section 5.10.2, "Guidelines", on page 61. Third party content creation tools can be used to generate 3GP files in addition to the hmp3gp utility, which converts Dialogic® HMP proprietary format files to 3GP files and vice versa. The feature supports 3GP files containing any of the following audio/video media data tracks defined by the 3GP file format:

- MPEG-4 video
- H.263 Video

- H.264 Video
- AMR-NB audio
- AMR-WB audio

Applications are capable of directly streaming multimedia stored in 3GP files to 3G and IP endpoints. Media can be streamed natively or transcoded during the playback operation. When performing a playback operation, compliance with the guidelines in Section 5.10.2, "Guidelines", on page 61 is recommended.

Hint tracks are used to facilitate the packetization of the media data for streaming. They describe the protocol characteristics used during the streaming process. Dialogic® HMP supports a 3GP play operation regardless of whether or not hint tracks are present in the file.

Authoring tools may be used to create hint tracks that can be added to existing 3GP files. See the Section 5.10.4, "Dialogic® Multimedia API Library Support", on page 61 for additional information.

5.10.1 Feature Highlights

The following table summarizes the features supported for 3GP file direct playback.

Table 6. 3GP File Format Direct Playback Feature Table

Feature	Support
Format Native Streaming	Yes
Transcode StreamingVideo	Yes (any mix of MPEG-4, H.263, H.264)
Transcode StreamingAudio	Yes (Either AMR-NB or AMR-WB to any supported audio codec)
IP Endpoint	Yes
3G Endpoint	Yes
Record	DMF[1]
3GP Profiles	Basic, Streaming Server, Progressive Download[2]
Hint Tracks	Hinted or non-hinted supported
MPEG-4 Codec	Simple profile, levels 0,1,2,3. RFC3016 packetization
H.263 Codec	Profile 0, levels 10,20,30. RFC2190 or RFC2429/4629 packetization[3]
H.264 Codec	Baseline profile, levels 1-1.3, RFC3984 packetization[4]
AMR-NB/AMR-WB Codec	1fpp, RFC3267/4867 packetization.
Random Access (seek)	No
Multi-track files	Partial[5]

1.Record operations are possible via the Dialogic® Multimedia API. Media data is recorded in Dialogic's proprietary Media Format (DMF). Media stored in DMF format can be converted to 3GP format using the hmp3gp conversion tool.
2.Progressive download profile 3GP files can be played only if they are stored locally.
3.For hinted files with H.263, follow the packetization used in the hint track. For non-hinted files, will default to RFC2429/4629 packetization.
4.For hinted files with H.264, follow the packetization used in the hint track. For non-hinted files, will default to Mode 1 (non-interleaved) packetization.
5.While track selection is not currently supported, a multi-track 3GP file can be played. The last audio/video media track will be automatically selected for playing.

5.10.2 Guidelines

The following guidelines provide additional information, limitations, and constraints when performing direct 3GP file operations. Applications utilizing this feature must comply with these guidelines:

- Only a single file can be played at a time; a multiple file list is not supported.
- All 3GP files playback operations using the Dialogic® Multimedia API library must reference files that are locally stored (this includes NFS mounts). HTTP, FTP or other remote access protocols are not supported for direct 3GP file playback.
- When streaming to 3G endpoints (3G-324M channel) the audio, video, and control data must not exceed 64 kbps bit rate. Third-party authoring and content creation tools can be used to alter files for compliance. See the Section 5.10.3, "3GP File Compatibility", on page 61 for more details.
- Files can be played natively or played using video and/or audio transcoding.

5.10.3 3GP File Compatibility

Often an application will require playback of multiple 3GP files during a call and/or media session in succession (such as announcements, prompts, and advertisements). Such cases require that all files be compatible with the system and have the same media characteristics (coding format, bit rate, frame rate, and display size) to ensure expected playback results. In some cases, network constraints require that the files conform to specific transmission guidelines.

Creating compatible 3GP files for direct playback using the Dialogic® Multimedia API library may require specific offline content authoring and editing tools due to the profiles/levels/bit rates/frame rates/picture sizes, etc., supported by HMP. While there are many such tools that can be used, Dialogic has specifically validated playback operations with the following:

- Helix Mobile Producer (Standard) version 11.1
  Content creation utility software for 3GPP mobile content.

5.10.4 Dialogic® Multimedia API Library Support

For 3GP file playback, the Dialogic® Multimedia API library provides the EMM_FILE_FORMAT_3GP value to indicate that the file is in 3GP format. Refer to the MM_MEDIA_VIDEO and MM_MEDIA_AUDIO data structures for file formats.

For event information error return codes used to indicate playback failure, refer to Section 12.3, "Multimedia API Event Information Error Return Codes", on page 232.

The following sample scenario illustrates how to record, convert, and then play a 3GP file.

1. The application calls the **mm_Record( )** function to create a Dialogic® Media Format for HMP (DMF) audio and/or video file(s).
2. The application converts the DMF files recorded to 3GP file format using the hmp3gp utility tool. Refer to the *Dialogic® Multimedia File Conversion Tools User Guide* for information about using this utility.
3. The application calls the **mm_Play( )** function to play the converted 3GP file.

The following sample scenario describes how to record and play a DMF audio/video file, and then convert it to a 3GP file.

1. The application calls the **mm_Record( )** function to initiate a recording of a multimedia stream and create DMF audio and/or video file(s).
2. The application calls the **mm_Play( )** function to play the DMF audio and/or video file(s).
3. The application converts the DMF files (audio/video) to 3GP file format using the hmp3gp utility tool. Refer to the *Dialogic® Multimedia File Conversion Tools User Guide* for information about using this utility.

Building Applications 6

This chapter provides general information on building applications using the Dialogic® Multimedia API library. The following information is provided:

- Compiling and Linking 63

6.1 Compiling and Linking

The following topics discuss compiling and linking requirements:

- Include Files
- Required Libraries
- Variables for Compiling and Linking

Note: When compiling an application, you must list Dialogic® libraries before all other libraries such as operating system libraries.

6.1.1 Include Files

The following Dialogic® Multimedia API header file is required when building your C or C++ application: *mmlib.h*

Other Dialogic® header files may also be required for your application, depending on the functionality. A commonly included header file is the *srllib.h* file with prototypes and definitions for the Dialogic® Standard Runtime Library (SRL).

6.1.2 Required Libraries

The following Dialogic® Multimedia API library is required when building your application: *libmml.so*

Other Dialogic® libraries may also be required for your application, depending on the functionality. A commonly used library is *libsrl.so* with implementation of the Dialogic® Standard Runtime Library (SRL).

6.1.3 Variables for Compiling and Linking

When building your applications you may use the following variables to reference the directories that contain header files and libraries:

INTEL_DIALOGIC_INC
: Variable that points to the directory where header files are stored.

INTEL_DIALOGIC_LIB
Variable that points to the directory where library files are stored.

These variables are automatically set at login and should be used in compiling and linking commands.

Function Summary by Category 7

This chapter contains an overview of the API functions and the categories into which they are logically grouped. Major topics include the following:

- Dialogic® Multimedia API Header File 65
- Device Management Functions 65
- Configuration Functions 66
- Input/Output Functions 66
- Media Streaming Functions 67
- DVR Control Functions 67
- Event Information Functions 68
- Error Processing Functions 68

7.1 Dialogic® Multimedia API Header File

The Dialogic® Multimedia API functions, parameters, data structures, values, events, and error codes are defined in the *mmlib.h*, *mmevts.h*, *mmparms.h*, and *mmerrs.h* header files. Video enumerations are defined in the *videodefs.h* header file. The Dialogic® Multimedia API functions use an "mm_" prefix.

7.2 Device Management Functions

Device Management functions open and close channel devices.

Before you can call any other library function on a device, that device must be opened using a Device Management open function. The open function specifies the name of a device and returns a unique device handle, which contains a numerical reference to the device. This device handle is how the device is identified once the device has been opened. The handle is specified as an input parameter for any function that operates on that device. The close function is used to close a device and release its handle. Device Management functions do not cause a device to be busy and will work on a device whether the device is busy or idle.

The following naming convention is used for channel-level multimedia device names:

mmBnCy
: where "n" is the board device number assigned to the virtual multimedia board and "y" is the number of a multimedia channel device associated with that board. Examples of multimedia channel device names are mmB1C1 and mmB1C2.

See the *Dialogic® Standard Runtime Library API Programming Guide* for more information onabout device names.

Note: This category of Device Management function is common among the APIs of the Dialogic® Software, but is a completely different subject from the Dialogic® Device Management API library functions, which provide run-time control and management of configurable system devices.

mm_Close( )
Close a previously opened multimedia device

mm_Open( )
Open a multimedia device

mm_Reset( )
Reset an open multimedia device

7.3 Configuration Functions

Configuration functions allow you to alter, examine, and control the configuration of an open device. They operate on an idle device, cause the device to be busy, and return the device to an idle state once the configuration operation is complete.

Configuration functions can only be executed asynchronously. They return immediately to indicate successful initiation or an error. If successfully initiated, they perform their operations and stop, reporting either completion or failure through an appropriately named event. See the *Dialogic® Standard Runtime Library API Programming Guide* for more information on asynchronous operation.

mm_GetParm( )
Get the current configuration parameters

mm_SetParm( )
Set the configuration parameters

7.4 Input/Output Functions

The Input/Output functions control the transfer of data. Except for the **mm_Stop( )** function, the Input/Output functions transfer data to and from an open, idle channel; they cause a channel to be busy while data transfer is taking place and return the channel to an idle state when data transfer is complete.

Input/Output functions can only be executed asynchronously. They return immediately to indicate successful initiation or an error. If successfully initiated, they perform their operations, reporting any intermediate events, and then stop, reporting either completion or failure through an appropriately named event.

See the *Dialogic® Standard Runtime Library API Programming Guide* for more information on asynchronous operation.

mm_Capture( )
Capture a still image from video stream

mm_Play( )
Play a media object

mm_Record( )
Record a media object

mm_Stop( )
Stop the device operations

7.5 Media Streaming Functions

Media Streaming functions allow the application to create, maintain, and delete a circular stream buffer within the library. These functions also provide notification when high and low water marks are reached.

mm_StreamClose( )
Close a media stream

mm_StreamGetStat( )
Get media stream statistics

mm_StreamOpen( )
Open a media stream

mm_StreamRead( )
Read from a media stream

mm_StreamReset( )
Reset a media stream

mm_StreamSetWaterMark( )
Set a media stream watermark

mm_StreamWrite( )
Write to a media stream

7.6 DVR Control Functions

Digital Video Recorder (DVR) Control functions allow the application to pause, resume, and seek during audio and video playback.

mm_GetDuration( )
Get duration of current presentation

mm_GetElapsedTime( )
Get current presentation time

mm_Pause( )
Pause current presentation

mm_Resume( )
Resume current presentation

mm_Seek( )
Place current presentation at specified position

7.7 Event Information Functions

Event Information functions primarily provide information on Dialogic® Multimedia API events.

mm_DisableEvents( )
Disable optional notification events

mm_EnableEvents( )
Enable optional notification events

mm_GetMetaEvent( )
Get current SRL event information

mm_ResultInfo( )
Retrieve error information for certain failure events.

7.8 Error Processing Functions

Error Processing functions provide Dialogic® Multimedia API error information.

mm_ErrorInfo( )
Retrieve current error information for a multimedia function

Function Information 8

This chapter is arranged in alphabetical order by function name and contains detailed information about each function in the Dialogic® Multimedia API.

8.1 Function Syntax Conventions

The Dialogic® Multimedia API functions use the following format:

```
int mm_FunctionName (nDeviceHandle, Parameter1, Parameter2, ..., ParameterN)
```

where:

int
specifies integer as the return data type of the function.

mm_FunctionName
represents the function name. All Dialogic® Multimedia API functions use the "mm_" prefix.

nDeviceHandle
represents the device handle, which contains a numerical reference to a device. The device handle is obtained when the device is opened and must be specified as an input parameter for any function that operates on that device.

Parameter1, Parameter2, ..., ParameterN
represent input or output parameters

mm_Capture( )

Name: int mm_Capture(nDeviceHandle, pImage, pRuntimeControl, pUserInfo)

Inputs:

int nDeviceHandle	• valid SRL handle representing a multimedia device
CPMM_MEDIA_IMAGE pImage	• pointer to image structure
CPMM_RUNTIME_CONTROL pRuntimeControl	• set to NULL
void* pUserInfo	• pointer to user-defined buffer

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Input/Output

Mode: Asynchronous

Platform: Dialogic® HMP 4.1LIN software

Description

The **mm_Capture( )** function provides the capability to capture a still image from a video stream, and is called after a connection is set up using the **dev_PortConnect( )** function between an outbound endpoint device and a multimedia device.

Parameter	Description
nDeviceHandle	specifies a valid multimedia device handle obtained from a previous open
pImage	points to the MM_MEDIA_IMAGE structure
pRuntimeControl	set to NULL; reserved for future use
pUserInfo	points to user-defined buffer. See the evtUserInfo field in the MM_METAEVENT structure.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_CAPTURE
Termination event reported upon successful completion of the operation.

Event Data: MM_CAPTURE_CMPLT_DETAILS structure

MMEV_CAPTURE_FAIL

Operation failure termination event reported upon encountering an error during the operation.

Event Data: MM_CAPTURE_CMPLT_DETAILS structure

MMEV_ERROR

Unsolicited event reported upon encountering an unexpected failure.

Event Data: MM_ERROR_RESULT structure

Cautions

- Before a still image can be captured, a video play should be initiated and then paused. To capture a still image, the application pauses a currently playing video stream using the **mm_Pause( )** function and then calls the **mm_Capture( )** function. Specifics of the image to be captured, such as image format (YUV), format details (height and width), and type of I/O access are specified in an MM_MEDIA_IMAGE structure and then passed to the **mm_Capture( )** function.
- Currently, only the file I/O mode is supported. For this mode, a file name must be specified.
- Currently, only the YUV 4:2:0 format is supported for image capture. The width and height of the image captured is determined by the video stream that is being played and is not configurable through the **mm_Capture( )** function. The width and height specified in the MM_MEDIA_IMAGE pointer argument, pImage, is not used for capture. The actual width and height of the image captured is returned in the capture completion event data, MM_CAPTURE_CMPLT_DETAILS.

Errors

If a Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, “Error Codes” for the function error codes and event information error return codes.

Example

```
int main(int argc, char* argv[])
{

    .
    .
    .
    /*
     * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to mm_Open( )
     * and mm_Play() has been called on this device.
     */

    // pause the playing stream



    // wait for and process the pause events...

       const char imageFileName[] = "./overlay2.yuv";

    MM_MEDIA_IMAGE image;

    INIT_MM_MEDIA_IMAGE(&image);
```

```
    image.unAccessMode = MM_MEDIA_ACCESS_MODE_FILE;
    image.szFileName = imageFileName;
    image.eFormat = eMTK_IMAGE_FORMAT_YUV;
    image.format.yuv.eFormat = eMTK_YUV_IMAGE_FORMAT_420;
    image.format.yuv.unWidth = 352;
    image.format.yuv.unHeight = 288;

    cout << "Calling mm_Capture" <<endl;
    /* Initiate Capture */
    if (mm_Capture(nDeviceHandle, &image, NULL, NULL) == EMM_ERROR)
    {
        //process error
        exit(1);
    }

        /* wait for and process capture events */

.
.
.

/* issue resume */
/* wait for and process resume events */
    exit(0);
}
```

See Also

- None.

mm_Close( )

Name: int mm_Close(nDeviceHandle, pCloseInfo)

Inputs: int nDeviceHandle
- valid SRL handle representing a multimedia device

CPMM_CLOSE_INFO pCloseInfo
- set to NULL

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Device Management

Mode: Synchronous

Platform: Dialogic® HMP software

Description

This function closes a multimedia device handle that was previously opened using **mm_Open( )**.

Note: This function does not affect any of the parameters that have been set for the device.

The **mm_Close( )** function stops any media operations on the device, but does not report these terminations with corresponding events (e.g., does not generate MMEV_PLAY or MMEV_PLAY_FAIL), because this function releases the handle on which the device events are reported. This function discards any outstanding events on the device handle and disables the generation of any new events on the handle.

Parameter	Description
nDeviceHandle	specifies a valid multimedia device handle obtained from a previous open
pCloseInfo	set to NULL; reserved for future use

Cautions

- The only process affected by this function is the process that called the function. Once a device is closed, a process can no longer act on that device using that device handle.
- This function discards any outstanding events on that handle.
- This function disables the generation of all events on that handle.
- Do not use the operating system close command to close a multimedia device; unpredictable results will occur.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure

event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

```
#include <mmlib.h>
int main(int argc, char* argv[])
{
   int nDeviceHandle; /* multimedia device handle */
   /* Main Processing
   .
   .
   */

   /*
    * Application is shutting down.
    * Need to close MM device handle.
    * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to mm_Open( ).
    */
   if (mm_Close(nDeviceHandle, NULL) == EMM_ERROR)
   {
      /* process error */
   }
}
```

See Also

- **mm_Open( )**
- **mm_Reset( )**
- **mm_Close( )**

mm_DisableEvents( )

Name: int mm_DisableEvents(nDeviceHandle, pEvents, pUserInfo)

Inputs:

int nDeviceHandle	• valid SRL handle representing a multimedia device
CPMM_EVENTS pEvents	• pointer to events information structure
void* pUserInfo	• pointer to user-defined buffer

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Event Information

Mode: Asynchronous

Platform: Dialogic® HMP software

Description

This function disables optional notification events that are enabled by default or were previously enabled by **mm_EnableEvents( )**. It disables optional notification events only in the process in which it is called. See Table 7, “Optional Notification Events”, on page 139 for the list of the optional notification events that can be enabled or disabled.

Use the INIT_MM_EVENTS( ) function to initialize the MM_EVENTS structure.

Parameter	Description
nDeviceHandle	specifies a valid multimedia device handle obtained from a previous open
pEvents	points to MM_EVENTS structure
pUserInfo	points to user-defined buffer. See MM_METAEVENT evtUserInfo field.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_DISABLEEVENTS
Termination event reported upon successful completion of the function.
Event Data: MM_DISABLE_EVENTS_RESULT structure (MM_RET_CODE)

MMEV_DISABLEEVENTS_FAIL

Operation failure termination event reported upon encountering an error during the operation.

Event Data: MM_DISABLE_EVENTS_RESULT structure (MM_RET_CODE)

MMEV_ERROR

Unsolicited event reported upon encountering an unexpected failure.

Event Data: MM_ERROR_RESULT structure

Cautions

None.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

```
int main(int argc, char* argv[])
{
    /* . . . */
    /*
    * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to mm_Open( ).
    */
    MM_EVENTS events;
    INIT_MM_EVENTS(&events);
    events.unMask = MMR_EVENT_VIDEO_RECORD_STARTED;
    if (mm_DisableEvents(nDeviceHandle, &events, NULL) == EMM_ERROR)
    {
        /* process error */
    }
```

See Also

- **mm_EnableEvents( )**
- **mm_GetMetaEvent( )**

mm_EnableEvents( )

Name: int mm_EnableEvents(nDeviceHandle, pEvents, pUserInfo)

Inputs:

int nDeviceHandle	• valid SRL handle representing a multimedia device
CPMM_EVENTS pEvents	• pointer to events information structure
void* pUserInfo	• pointer to user-defined buffer

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Event Information

Mode: Asynchronous

Platform: Dialogic® HMP software

Description

This function enables optional notification events only in the process in which it is called. See Table 7, "Optional Notification Events", on page 139 for the list of the optional notification events that can be enabled or disabled.

Use the INIT_MM_EVENTS( ) function to initialize the MM_EVENTS structure.

Parameter	Description
nDeviceHandle	specifies a valid multimedia device handle obtained from a previous open
pEvents	points to MM_EVENTS structure
pUserInfo	points to user-defined buffer. See MM_METAEVENT evtUserInfo field.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, "Events".

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_ENABLEEVENTS
Termination event reported upon successful completion of the function.
Event Data: MM_ENABLE_EVENTS_RESULT structure (MM_RET_CODE)

MMEV_ENABLEEVENTS_FAIL

Operation failure termination event reported upon encountering an error during the operation.

Event Data: MM_ENABLE_EVENTS_RESULT structure (MM_RET_CODE)

MMEV_ERROR

Unsolicited event reported upon encountering an unexpected failure.

Event Data: MM_ERROR_RESULT structure

Cautions

None.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

```
int main(int argc, char* argv[])
{
    /* . . . */
    /*
    * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to mm_Open( ).
    */
    MM_EVENTS events;
    INIT_MM_EVENTS(&events);
    events.unMask = MMR_EVENT_VIDEO_RECORD_STARTED;
    if (mm_EnableEvents(nDeviceHandle, &events, NULL) == EMM_ERROR)
    {
        /* process error */
    }
```

See Also

- **mm_DisableEvents( )**
- **mm_GetMetaEvent( )**

mm_ErrorInfo( )

Name: int mm_ErrorInfo(pInfo)

Inputs: PMM_INFO pInfo • pointer to the MM_INFO data structure

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Error Processing

Mode: Synchronous

Platform: Dialogic® HMP software

Description

This function obtains the error information for the last error that occurred in the Dialogic® Multimedia API and provides it in the MM_INFO structure. To retrieve the error information about a failed function, the **mm_ErrorInfo( )** function must be called immediately after the Dialogic® Multimedia API function failed.

Use the INIT_MM_INFO( ) inline function to initialize the MM_INFO structure.

Parameter	Description
pInfo	points to the MM_INFO structure where information about the error is contained.

Cautions

- The **mm_ErrorInfo( )** function can only be called in the same thread in which the routine that had the error was called. The **mm_ErrorInfo( )** function cannot be called to retrieve error information for a function that returned error information in another thread.
- Because the Dialogic® Multimedia API keeps the error information for the last Dialogic® Multimedia API function call that sets it, the error information may be changed by succeeding Dialogic® Multimedia API calls. Therefore it is recommended to check and retrieve error information immediately after a Dialogic® Multimedia API function fails.

Errors

This function returns an EMM_ERROR if an incorrect parameter is specified. The **mm_ErrorInfo( )** function should not be called recursively if it returns EMM_ERROR to indicate failure. An EMM_ERROR generally indicates that **pInfo** is NULL or invalid.

Example

```
#include <mmlib.h>
/*
* This function is called to print MM_INFO to the system console
* Typically it would be called after a call to mm_ErrorInfo
* to print the resulting MM_INFO data structure
*/
void PrintMM_INFO(const MM_INFO* pInfo)
{
    printf("mmValue = %#x\n", pInfo->mmValue);
    printf("mmMsg = %s\n", pInfo->mmMsg);
    printf("additionalInfo = %s\n", pInfo->additionalInfo);
}

/*
* This function can be called anytime an error occurs
* Not shown is an MM function which fails and calls this function
* This procedure prints error information to the console with no other side effects
*/
void PrintErrorInfo(void)
{
    int retCode;
    MM_INFO t_Info;
    INIT_MM_INFO(&t_Info);
    retCode = mm_ErrorInfo(&t_Info);
    if (retCode == EMM_SUCCESS)
    {
        printf("mm_ErrorInfo successfully called\n");
        PrintMM_INFO(&t_Info);
    }
    else
    {
         printf("mm_ErrorInfo call failed\n");
    }
}
int main(int argc, char* argv[])
{
    /* . . . */

    /*
    * Calling an MM function with error processing
    */
    if (mm_DisableEvents(nDeviceHandle, &events, NULL) == EMM_ERROR)
    {
        /* process error */
        PrintErrorInfo();
    }
}
```

See Also

- **mm_GetMetaEvent( )**

mm_GetDuration( )

Name:	int mm_GetDuration(nDeviceHandle, pGetDurationInfo, * pUserInfo);	
Inputs:	int nDeviceHandle	• Multimedia device handle
	CPMM_GETDURATION_INFO pGetDurationInfo	• pointer to details of the request
	void* pUserInfo	• pointer to user-defined data
Returns:	EMM_SUCCESS if successful EMM_ERROR if failure	
Includes:	mmlib.h	
Category:	DVR Control	
Mode:	Asynchronous	
Platform:	Dialogic® HMP software	

Description

The **mm_GetDuration( )** function gets the duration of the currently playing multimedia stream on the specified multimedia device. The returned value will be the length of the entire file in time units.

Parameter	Description
nDeviceHandle	valid SRL handle representing a multimedia device
pGetDurationInfo	specifies a pointer to the MM_GETDURATION_INFO structure that contains the details of the request
pUserInfo	points to the user-defined buffer. See the MM_METAEVENT evtUserInfo field.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_GETDURATION

Termination event reported upon successfully getting the requested duration.

Event Data: MM_GETDURATION_CMPLT structure

MMEV_GETDURATION_FAIL

Termination event reported upon encountering an error while getting the requested duration.

Event Data: MM_GETDURATION_CMPLT structure

Cautions

- **mm_GetDuration( )** will only be valid during the period of time between the MMEV_PLAY_ACK event and the MMEV_PLAY event for the requested stream type. See MM_GETDURATION_INFO for more information.
- **mm_GetDuration( )** is only supported for multimedia that is playing in file I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_FILE in **mm_Play( )**) and not in memory I/O or stream I/O mode.
- **mm_GetDuration( )** is only supported for files in the Dialogic® proprietary file format – when eFileFormat is set to EMM_FILE_FORMAT_PROPRIETARY or EMM_AUD_FILEFORMAT_PROPRIETARY in **mm_Play( )**.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

For an example of using the **mm_GetDuration( )** function, refer to the Example section for **mm_Pause( )**.

See Also

None.

mm_GetElapsedTime( )

Name: int mm_GetElapsedTime(nDeviceHandle, pGetElapsedTimeInfo, * pUserInfo)

Inputs:

int nDeviceHandle	• Multimedia device handle
CPMM_GETELAPSEDTIME_INFO pGetElapsedTimeInfo	• pointer to details of the request
void* pUserInfo	• pointer to user-defined data

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: DVR Control

Mode: Asynchronous

Platform: Dialogic® HMP software

Description

The **mm_GetElapsedTime( )** function gets the elapsed time of the currently playing multimedia stream on the specified multimedia device. The returned value will be the time between the beginning of the specified stream and the current position within that stream.

Parameter	Description
nDeviceHandle	valid SRL handle representing a multimedia device
pGetElapsedTimeInfo	specifies a pointer to the MM_GETELAPSEDTIME_INFO structure that contains the details of the request
pUserInfo	points to user-defined buffer. See the MM_METAEVENT evtUserInfo field.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_GETELAPSEDTIME

Termination event reported upon successfully getting the requested elapsed time.

Event Data: MM_GETELAPSEDTIME_CMPLT structure

MMEV_GETELAPSEDTIME_FAIL

Termination event reported upon encountering an error while getting the requested elapsed time.

Event Data: MM_GETELAPSEDTIME_CMPLT structure

Cautions

- **mm_GetElapsedTime( )** will only be valid during the period of time between the MMEV_PLAY_ACK event and the MMEV_PLAY event for the requested stream type. See MM_GETELAPSEDTIME_INFO for more information.
- **mm_GetElapsedTime( )** is only supported for multimedia that is playing in file I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_FILE in **mm_Play( )**) and not in memory I/O or stream I/O mode.
- **mm_GetElapsedTime( )** is only supported for files in the Dialogic® proprietary file format – when eFileFormat is set to EMM_FILE_FORMAT_PROPRIETARY or EMM_AUD_FILEFORMAT_PROPRIETARY in **mm_Play( )**.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

For an example of using the **mm_GetElapsedTime( )** function, refer to the Example section for **mm_Pause( )**.

See Also

None.

mm_GetMetaEvent( )

Name: int mm_GetMetaEvent(pMetaEvent)

Inputs: MM_METAEVENT pMetaEvent
- pointer to MM_METAEVENT data structure of metaevent data

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Event Information

Mode: Synchronous

Platform: Dialogic® HMP software

Description

This function retrieves event information for the current Dialogic® Standard Runtime Library (SRL) event that stores the Dialogic® Multimedia API and non-Dialogic® Multimedia API event information. The MM_METAEVENT data structure contains explicit information describing the SRL event to be returned to the application program. This data structure provides uniform information retrieval among call control libraries and across operating systems.

You must call the **mm_GetMetaEvent( )** function to retrieve any Dialogic® Multimedia API event information and any other event information if you are not sure of the event type. If the metaevent is a Dialogic® Multimedia API event, the MMME_MM_EVENT bit in the MM_METAEVENT flags field will be set. The MM_METAEVENT fields contain valid Dialogic® Multimedia API-related data only when the MMME_MM_EVENT bit is set. Do **not** use these fields for obtaining multimedia information if the bit is not set.

The current SRL event information is not changed or altered by calling the **mm_GetMetaEvent( )** function to retrieve event information. This function may be used as a convenience function to retrieve the event information for all SRL events. Whether the event is a Dialogic® Multimedia API event or any other SRL event, the SRL event information (for example, evtdatap, evttype) may be retrieved from the MM_METAEVENT data structure instead of using SRL functions to retrieve this information. For information on the Dialogic® SRL API, see the *Dialogic® Standard Runtime Library API Library Reference*.

Use the INIT_MM_METAEVENT( ) function to initialize the MM_METAEVENT structure.

Parameter	Description
pMetaEvent	points to the MM_METAEVENT structure filled by this function

Cautions

- The **mm_GetMetaEvent( )** function **must** be the first function called before processing any Dialogic® Multimedia API event.

- An application should call the **mm_GetMetaEvent( )** function only once for a given event. Calling the function more than once will result in data corruption or an access violation.
- The event must be processed entirely in the same thread or all information about the event must be retrieved before processing the event in another thread.

Errors

This function returns an EMM_ERROR if an incorrect parameter is specified.

If this function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. See Chapter 12, “Error Codes” for the function error codes.

Example

```
MM_METAEVENT metaevent;
INIT_MM_METAEVENT(&metaevent);
if (sr_waitevt(timeout) != -1)
{
    if (mm_GetMetaEvent(&metaevent) == EMM_ERROR)
    {
        /* get and process the error */
    }
    else
    {
        /* Process retrieved metaevent */
    }
}
```

See Also

- **mm_DisableEvents( )**
- **mm_EnableEvents( )**

mm_GetParm( )

Name: int mm_GetParm(nDeviceHandle, pGetParm, pUserInfo)

Inputs:

int nDeviceHandle	• valid SRL handle representing a multimedia device
CPMM_GET_PARM pGetParm	• pointer to get parameters information structure
void* pUserInfo	• pointer to user-defined buffer

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Configuration

Mode: Asynchronous

Platform: Dialogic® HMP software

Description

This function gets the value of the specified configuration parameters.

Use the INIT_MM_GET_PARM( ) function to initialize the MM_GET_PARM structure.

Parameter	Description
nDeviceHandle	specifies a valid multimedia device handle obtained from a previous open
pGetParm	points to MM_GET_PARM information structure
pUserInfo	points to user-defined buffer. See MM_METAEVENT evtUserInfo field.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_GETPARM
Termination event reported upon successful completion of the function.
Event Data: MM_GET_PARM_RESULT structure

MMEV_GETPARM_FAIL
Operation failure termination event reported upon encountering an error during the operation.
Event Data: MM_GET_PARM_RESULT structure

MMEV_ERROR

Unsolicited event reported upon encountering an unexpected failure.

Event Data: MM_ERROR_RESULT structure

Cautions

- If an MMEV_GET_PARM_FAIL event is generated, the MM_GET_PARM_RESULT data structure contains invalid data.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

```
#include <mmlib.h>
int main(int argc, char* argv[])
{
    /* . . . */
    /*
    * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to mm_Open( ).
    */
    MM_GET_PARM parm;
    INIT_GET_PARM(&parm);
    parm.eParm = MM_PARM_REC_IFRAME_TIMEOUT;
    if (mm_GetParm(nDeviceHandle, &parm, NULL) == EMM_ERROR)
    {
        /* process error */
    }
}
```

See Also

- **mm_SetParm( )**

mm_Open( )

Name: int mm_Open(szDevName, pOpenInfo, pUserInfo)

Inputs:

const char* szDevName	• pointer to device name to open
CPMM_OPEN_INFO pOpenInfo	• set to NULL
void* pUserInfo	• pointer to user-defined buffer

Returns: **device handle** if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Device Management

Mode: Asynchronous

Platform: Dialogic® HMP software

Description

The **mm_Open( )** function opens a multimedia device and returns a unique Dialogic® Standard Runtime Library (SRL) device handle to identify the device. All subsequent references to the opened device must be made using the handle until the device is closed. For information on the Dialogic® SRL API, see the *Dialogic® Standard Runtime Library API Library Reference*.

The device handle returned by this function is defined by Dialogic. It is not a standard operating system file descriptor.

Parameter	Description
szDevName	points to a multimedia device name to open. The following naming convention is used for channel-level multimedia device names: • mmBnCy – where "n" is the board device number assigned to the virtual multimedia board and "y" is the number of a multimedia channel device associated with that board. Examples of multimedia channel device names are mmB1C1 and mmB1C2.
pOpenInfo	set to NULL; reserved for future use
pUserInfo	points to user-defined buffer. See MM_METAEVENT evtUserInfo field.

If this function is called with valid arguments, a device handle is returned immediately. Before using this device handle in other function calls, the application must wait for an MMEV_OPEN event indicating the handle is valid.

If this function is called and it generates an MMEV_OPEN_FAIL event, a device handle is returned, but the application must close the handle by calling **mm_Close( )**.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, "Events".

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_OPEN
: Termination event reported upon successful completion of the function.

  Event Data: MM_OPEN_RESULT structure (MM_RET_CODE)

MMEV_OPEN_FAIL
: Operation failure termination event reported upon encountering an error during the operation.

  Event Data: MM_OPEN_RESULT structure (MM_RET_CODE)

  Note: The application program must call **mm_Close( )** to clean up after this failure.

MMEV_ERROR
: Unsolicited event reported upon encountering an unexpected failure.

  Event Data: MM_ERROR_RESULT structure

Cautions

- You must obtain an MMEV_OPEN event before using the device handle.
- You must close the handle by calling **mm_Close( )** if the function generates an MMEV_OPEN_FAIL event.
- Do not use the operating system open function to open multimedia devices; unpredictable results will occur.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

```
#include <mmlib.h>
int main(int argc, char* argv[])
{
   int nDeviceHandle; /* multimedia device handle */
   /* Open Device */
   if ((nDeviceHandle = mm_Open("mmB1C1", NULL, NULL)) == EMM_ERROR) {
      /* process error */
}
.
.
.
```

See Also

- **mm_Close( )**
- **mm_Reset( )**

mm_Pause( )

Name: int mm_Pause(nDeviceHandle, pPauseInfo, pUserInfo)

Inputs:

int nDeviceHandle	• Multimedia device handle
CPMM_PAUSE_INFO pPauseInfo	• pointer to details of the pause request
void* pUserInfo	• pointer to user-defined data

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: DVR Control

Mode: Asynchronous

Platform: Dialogic® HMP software

Description

The **mm_Pause( )** function pauses a multimedia stream that is currently playing.

Parameter	Description
nDeviceHandle	valid SRL handle representing a multimedia device
pPauseInfo	specifies a pointer to the MM_PAUSE_INFO structure that contains the details of the pause request
pUserInfo	points to user-defined buffer. See the MM_METAEVENT evtUserInfo field

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_PAUSE

Termination event reported upon successfully pausing.

Event Data: MM_PAUSE_PLAY_CMPLT structure

MMEV_PAUSE_FAIL

Termination event reported upon encountering an error while attempting to pause.

Event Data: MM_PAUSE_PLAY_CMPLT structure

Cautions

- The **mm_Pause( )** function is only valid during the period of time between the MMEV_PLAY_ACK event and the MMEV_PLAY event for the requested stream type. See the MM_PAUSE_INFO for more information.
- **mm_Pause( )** is only supported for multimedia that is playing in file I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_FILE in **mm_Play( )**) and not in memory I/O or stream I/O mode.
- **mm_Pause( )** is only supported for files in the Dialogic® proprietary file format – when eFileFormat is set to EMM_FILE_FORMAT_PROPRIETARY or EMM_AUD_FILEFORMAT_PROPRIETARY in **mm_Play( )**.
- In an audiovideo playback, if the playback is synchronous (both audio and video play is initiated by a single **mm_Play( )** function call), **BOTH** the audio and video streams will be resumed, regardless of the unStreamType setting in MM_PAUSE_INFO. This allows synchronization to be maintained.

Example

```
int main()
{

#include "mmlib.h"
.................

            // Open device

            sprintf(channame,"mmB1C1");
            if ((ddd = mm_Open(channame, 0, NULL)) < 0)
            {
                printf("mm_Open() failed for %s\n", channame);
            }
...................

            // Do a play

..................

            // Pause
           MM_PAUSE_INFO pauseinfo;
           INIT_MM_PAUSE_INFO(&pauseinfo);
           pauseinfo.unStreamType = EMM_PAUSE_VIDEO_AUDIO;
           pauseinfo.unAttribute = EMM_PAUSE_AUDIO_SILENCE;


           ret = mm_Pause(ddd, &pauseinfo, NULL);
           if (ret == EMM_ERROR) {
             printf("mm_Pause() failed!\n");
             exit (-1);
           }
           printf("Pausing\n");

           // Resume
            MM_RESUME_INFO resumeinfo;
           INIT_MM_RESUME_INFO(&resumeinfo);
           resumeinfo.unStreamType = EMM_RESUME_VIDEO_AUDIO;
           resumeinfo.unAttribute = EMM_RESUME_VIDEO_NONE;

           ret = mm_Resume(ddd, &resumeinfo, NULL);
           if (ret == EMM_ERROR) {
             printf("mm_Resume() failed!\n");
             exit (-1);
```

```
  }
  printf("Resuming\n");

   // Seek
  MM_SEEK_INFO seekinfo;
  INIT_MM_SEEK_INFO(&seekinfo);
  seekinfo.unStreamType = EMM_SEEK_VIDEO_AUDIO;
  seekinfo.unOrigin = EMM_SEEK_SET;
  seekinfo.nOffset  = atoi(&buffer[1]);

  ret = mm_Seek(ddd, &seekinfo, NULL);
  if (ret == EMM_ERROR) {
    printf("mm_Seek() failed!\n");
    exit (-1);
  }
  printf("Seeking\n");
}

   // Get elapsed time
  MM_GETELAPSEDTIME_INFO etinfo;
  INIT_MM_GETELAPSEDTIME_INFO(&etinfo);
  etinfo.unStreamType = EMM_GETELAPSEDTIME_VIDEO_AUDIO;
  ret = mm_GetElapsedTime(ddd, &etinfo, NULL);
  if (ret == EMM_ERROR) {
    printf("mm_GetElapsedTime() failed!\n");
    exit (-1);
  }
  printf("getting time\n");
}

   // Get duration
  MM_GETDURATION_INFO dinfo;
  INIT_MM_GETDURATION_INFO(&dinfo);
  dinfo.unStreamType = EMM_GETDURATION_VIDEO_AUDIO;

  ret = mm_GetDuration(ddd, &dinfo, NULL);
  if (ret == EMM_ERROR) {
    printf("mm_GetDuration() failed!\n");
    exit (-1);
  }
  printf("getting duration\n");
}
```

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, “Error Codes” for the function error codes and event information error return codes.

See Also

None.

mm_Play( )

Name: int mm_Play(nDeviceHandle, pPlayInfo, pRuntimeControl, pUserInfo)

Inputs:	int nDeviceHandle	• valid SRL handle representing a multimedia device
	CPMM_PLAY_INFO pPlayInfo	• pointer to play information structure
	CPMM_RUNTIME_CONTROL pRuntimeControl	• pointer to runtime control information structure.
	void* pUserInfo	• pointer to user-defined buffer
Returns:	EMM_SUCCESS if successful EMM_ERROR if failure	
Includes:	mmlib.h	
Category:	Input/Output	
Mode:	Asynchronous	
Platform:	Dialogic® HMP software	

Description

The **mm_Play( )** function plays back audio and video data from multimedia files while maintaining their synchronization. It can also play back only the audio portion or video portion. Multimedia files consist of a Dialogic® Multimedia API audio file and video file as specified in the MM_MEDIA_AUDIO and MM_MEDIA_VIDEO structures. When performing multimedia recording or playback, the video data is synchronized with the audio data.

Note: This function currently supports a single set of concurrent audio and video files. It does not support multiple sets of audio and video files.

The **mm_Play( )** function also includes the ability to play a still image over a video stream. To play a still image, use the pPlayInfo parameter to specify the image. Set the ItemType field in the MM_PLAY_RECORD_LIST structure to EMM_MEDIA_TYPE_IMAGE and provide details such as image format (YUV or JPEG), format details (height, width) and type of I/O access in the MM_MEDIA_IMAGE structure.

Use the INIT_MM_PLAY_INFO( ) inline function to initialize the MM_PLAY_INFO structure. Also initialize all structures nested within MM_PLAY_INFO with their corresponding INIT inline functions.

Parameter	Description
nDeviceHandle	specifies a valid multimedia device handle obtained from a previous open.
pPlayInfo	points to MM_PLAY_INFO (MM_PLAY_RECORD_INFO) structure.
pRuntimeControl	points to MM_RUNTIME_CONTROL structure.
pUserInfo	points to user-defined buffer. See MM_METAEVENT evtUserInfo field.

Notes: ***1.*** The MM_MEDIA_AUDIO and MM_MEDIA_VIDEO structures must be initialized using the INIT_MM_MEDIA_AUDIO and INIT_MM_MEDIA_VIDEO inline functions, respectively.

2. This function can transmit a tone or "beep" to indicate the start of playback. See MM_MEDIA_AUDIO and MM_MEDIA_VIDEO for more information.

3. For memory I/O, the pointer to the buffer and size of the buffer, pBuffer and unBufferSize, must be specified. For file I/O, the file name, szFileName, must be specified. For stream I/O, the stream handle, nStreamHandle, must be specified.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, "Events".

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_PLAY_ACK
Initiation event reported upon successful start of the play operation.

Event Data: MM_PLAY_ACK structure (MM_RET_CODE)

MMEV_PLAY_ACK_FAIL
Initiation failure termination event reported upon encountering an error before the play operation begins.

Event Data: MM_PLAY_ACK structure (MM_RET_CODE)

MMEV_PLAY
Termination event reported upon successful completion or successful termination of the play operation.

Event Data: MM_PLAY_CMPLT structure (MM_PLAY_RECORD_CMPLT)

After a play operation has begun, an MMEV_PLAY event is reported to the application program when one of the following conditions occurs:

- the device finishes playing the media specified in the **pPlayInfo** parameter; i.e., it reaches the end of file
- the application program issues a stop operation with **mm_Stop( )**. See the **mm_Stop( )** function for conditions that generate no completion events or multiple completion events
- the application program issues a reset operation with **mm_Reset( )**
- one of the runtime control conditions specified in the **pRuntimeControl** parameter has been met

MMEV_PLAY_FAIL
Operation failure termination event reported upon encountering an error during the play operation.

Event Data: MM_PLAY_CMPLT structure (MM_PLAY_RECORD_CMPLT)

Note: See Section 12.5, "Terminating and Non-Terminating Play/Record Errors", on page 235 for related information.

MMEV_ERROR

Unsolicited event reported upon encountering an unexpected failure of the play operation.

Event Data: MM_ERROR_RESULT structure

Note: For more information on these events, see Chapter 9, "Events". See also Section 12.5, "Terminating and Non-Terminating Play/Record Errors", on page 235 for useful information.

Cautions

- See the MMEV_PLAY event above for conditions under which you can receive more than one MMEV_PLAY event.
- When playing a WAVE file, codec information contained in the WAVE header of the file being played will take precedence over the application's settings for the unCoding, unSampleRate, and unBitsPerSample fields of the MM_AUDIO_CODEC structure.
- Only JPEG and YUV formats are currently supported for playing a still image.
- Only file I/O access mode is supported for still image play. Memory I/O and stream I/O are not supported.
- If there is no place to send packets, the expected behavior is that the **mm_play( )** function will hang. This occurs when an IP media session has not been established and the multimedia device connects to and relies on the IP media device to be a transport layer, or when the IP media session has been established but the direction goes against the **mm_play( )** function. To resolve this, the application should call the **mm_Stop( )** function to terminate the play, or set the runtime control condition EMM_TERM_MAXTIME beforehand.
- The input YUV 4:2:0 or JPEG image file size should not exceed 152,064 bytes. This limit is based upon the maximum file size required to support CIF (352x288) frames in YUV 4:2:0 format. A compressed JPEG input file may contain a frame with a resolution larger than CIF (352x288) as long as the file size is less than 152,064 bytes.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

1: Audio play scenario

```
#include <mmlib.h>

void add_rtc_item(PMM_RUNTIME_CONTROL RTCList, int *RTCItemCountp, eMM_TERMINATION_REASON
reason, int value, eMM_TERMINATION_ACTION action)
{

   int index = *RTCItemCountp;
   if (index >= 1)
      {
       RTCList[index-1].next = &(RTCList[index]);
      }

   INIT_MM_RUNTIME_CONTROL(&(RTCList[index]));
   RTCList[index].Reason = reason;
   RTCList[index].unValue = value;
```

```
   RTCList[index].Action = action;
   RTCList[index].next = NULL;
   *RTCItemCountp)++;
}
int main(int argc, char* argv[])
{
    /* . . . */
    /*
     * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to mm_Open( ).
     */
     MM_PLAY_INFO play_info;
     INIT_MM_PLAY_INFO(&play_info);
     MM_PLAY_RECORD_LIST playlist[2];
     MM_MEDIA_ITEM_LIST mediaitemlist1[1];
     MM_MEDIA_ITEM_LIST mediaitemlist2[1];
     MM_VIDEO_CODEC VideoCodecType1;
     INIT_MM_VIDEO_CODEC(&VideoCodecType1);

     VideoCodecType1.Coding = VIDEO_CODING_H263;
     VideoCodecType1.Profile = VIDEO_PROFILE_0_H263;
     VideoCodecType1.Level = VIDEO_LEVEL_10_H263;
     VideoCodecType1.ImageWidth = VIDEO_IMAGE_WIDTH_176;
     VideoCodecType1.ImageHeight = VIDEO_IMAGE_HEIGHT_144;
     VideoCodecType1.BitRate = (eVIDEO_BITRATE)0;
     VideoCodecType1.FramesPerSec = VIDEO_FRAMESPERSEC_15;
     VideoCodecType1.SamplingRate = VIDEO_SAMPLING_RATE_90000;
     VideoCodecType1.VisualConfigSize = 0;
     VideoCodecType1.VisualConfiguration = NULL;

     MM_AUDIO_CODEC AudioCodecType1;
     INIT_MM_AUDIO_CODEC(&AudioCodecType1);
     AudioCodecType1.unCoding = MM_DATA_FORMAT_PCM;
     AudioCodecType1.unSampleRate = MM_DRT_8KHZ;
     AudioCodecType1.unBitsPerSample = 16;
     const char VideoFileName1[] = "/dir/file1.vid";
     const char AudioFileName1[] = "/dir/file3.aud";
     int cc;
     int xx;
     cc = 0;
     // Build Video Item 1

     INIT_MM_MEDIA_ITEM_LIST(&mediaitemlist1[cc]);
     INIT_MM_MEDIA_VIDEO(&mediaitemlist1[cc].item.video);
     mediaitemlist1[cc].ItemChain = EMM_ITEM_EOT;
     mediaitemlist1[cc].item.video.eFileFormat = EMM_FILE_FORMAT_PROPRIETARY;
     mediaitemlist1[cc].item.video.codec = VideoCodecType1;
     mediaitemlist1[cc].item.video.unMode = 0;
     mediaitemlist1[cc].item.audio.unOffset = 0;
     mediaitemlist1[cc].item.video.szFileName = VideoFileName1;
     mediaitemlist1[cc].item.video.unAccessMode = MM_MEDIA_ACCESS_MODE_FILE;
     cc++;
     xx = 0;
     // Add Video Items to the PlayList
     INIT_MM_PLAY_RECORD_LIST(&playlist[xx]);
     playlist[xx].ItemChain = EMM_ITEM_CONT;
     playlist[xx].ItemType = EMM_MEDIA_TYPE_VIDEO;
     playlist[xx].list = mediaitemlist1;
     xx++;
     cc = 0;
     // Build Audio Item 1

     INIT_MM_MEDIA_ITEM_LIST(&mediaitemlist2[cc]);
     INIT_MM_MEDIA_AUDIO(&mediaitemlist2[cc].item.audio);
     mediaitemlist2[cc].ItemChain = EMM_ITEM_EOT;
     mediaitemlist2[cc].item.audio.eFileFormat = EMM_AUD_FILEFORMAT_VOX;
     mediaitemlist2[cc].item.audio.codec = AudioCodecType1;
     mediaitemlist2[cc].item.audio.unMode = 0;
```

```
    mediaitemlist2[cc].item.audio.unOffset = 0;
    mediaitemlist2[cc].item.audio.szFileName = AudioFileName1;
    mediaitemlist2[cc].item.audio.unAccessMode = MM_MEDIA_ACCESS_MODE_FILE;
    cc++;
    // Add Audio Items to the PlayList
    INIT_MM_PLAY_RECORD_LIST(&playlist[xx]);
    playlist[xx].ItemChain = EMM_ITEM_EOT;
    playlist[xx].list = mediaitemlist2;

    // Form Play Info
    INIT_MM_PLAY_RECORD_INFO(&play_info);
    play_info.list = playlist;
    play_info.eFileFormat=EMM_FILE_FORMAT_UNDEFINED;

    MM_RUNTIME_CONTROL RTCList[3];
    int RTCItemCount = 0;

    //stop playing both audio and video 20 seconds after the MMEV_PLAY_ACK event
    add_rtc_item(RTCList, &RTCItemCount, EMM_TERM_MAXTIME, 20000, EMM_TA_AUDIO_VIDEO_STOP);

    // Initiate Play
    if (mm_Play(nDeviceHandle, &play_info, RTCList, NULL) == EMM_ERROR)
    {
        /* process error */
    }
}
```

2: Simple image play from file scenario

```
int main(int argc, char* argv[])
{

   .
   .
   .
   /*
    * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to mm_Open( ).
    */
    MM_PLAY_INFO play_info;
    MM_PLAY_RECORD_LIST playList[1];
    MM_MEDIA_ITEM_LIST imageMediaItemList[1];
    MM_RUNTIME_CONTROL mmRtc;

    enum {IMAGE_INDEX};

    const char imageFileName[] = "./overlay1.yuv";

    /* Build the mm image item */
   INIT_MM_MEDIA_ITEM_LIST(&(imageMediaItemList[0]));}
   imageMediaItemList[0].ItemChain = EMM_ITEM_EOT;
   INIT_MM_MEDIA_IMAGE(&imageMediaItemList[0].item.image);
   imageMediaItemList[0].item.image.unAccessMode = MM_MEDIA_ACCESS_MODE_FILE;
   imageMediaItemList[0].item.image.szFileName = imageFileName;
   imageMediaItemList[0].item.image.eFormat = eMTK_IMAGE_FORMAT_YUV;
   imageMediaItemList[0].item.image.format.yuv.eFormat = eMTK_YUV_IMAGE_FORMAT_420;
   imageMediaItemList[0].item.image.format.yuv.unWidth = 352;
   imageMediaItemList[0].item.image.format.yuv.unHeight = 288;
   INIT_MM_PLAY_RECORD_LIST(&(playList[IMAGE_INDEX]));
   /* Add Image Item to the PlayList */
   playList[IMAGE_INDEX].ItemChain = EMM_ITEM_EOT;
   playList[IMAGE_INDEX].ItemType = EMM_MEDIA_TYPE_IMAGE;
   playList[IMAGE_INDEX].list = imageMediaItemList;
   /* Form Play Info */
   INIT_MM_PLAY_INFO(&play_info)
    play_info.eFileFormat = EMM_FILE_FORMAT_PROPRIETARY;
    play_info.list = playList;
    /* Form RTC */
```

```
    INIT_MM_RUNTIME_CONTROL(&mmRtc);
    mmRtc.Reason = EMM_TERM_MAXTIME;
    mmRtc.Action = EMM_TA_VIDEO_STOP;
    mmRtc.unValue = 5000; // 5 seconds
    /* Initiate Play */
    if (mm_Play(nDeviceHandle, &play_info, &mmRtc, NULL) == EMM_ERROR)
    {
        /*process error */
        exit(1);
    }
    exit(0);
}
```

See Also

- mm_Record( )
- mm_Reset( )
- mm_Stop( )

mm_Record( )

Name:	int mm_Record(nDeviceHandle, pRecordInfo, pRuntimeControl, pUserInfo)	
Inputs:	int nDeviceHandle	• valid SRL handle representing a multimedia device
	CPMM_RECORD_INFO pRecordInfo	• pointer to record information structure
	CPMM_RUNTIME_CONTROL pRuntimeControl	• pointer to runtime control information structure
	void* pUserInfo	• pointer to user-defined buffer
Returns:	EMM_SUCCESS if successful EMM_ERROR if failure	
Includes:	mmlib.h	
Category:	Input/Output	
Mode:	Asynchronous	
Platform:	Dialogic® HMP software	

Description

The **mm_Record( )** function records synchronized audio and video data to multimedia files. It can also record only the audio portion or video portion.

Multimedia files consist of a Dialogic® Multimedia API audio file and video file as specified in the MM_MEDIA_AUDIO and MM_MEDIA_VIDEO structures. When performing multimedia recording or playback, the video data is synchronized with the audio data.

Note: Incoming RFC2833 DTMFs will not be received by the **mm_Record( )** function. Use a voice device to capture and detect these DTMFs.

This function can transmit a start-of-recording tone or “beep” to notify the party being recorded. See MM_MEDIA_AUDIO and MM_MEDIA_VIDEO for more information.

Use the INIT_MM_RECORD_INFO( ) inline function to initialize the MM_RECORD_INFO structure. Also initialize all structures nested within MM_RECORD_INFO with their corresponding INIT inline functions.

Parameter	Description
nDeviceHandle	specifies a valid multimedia device handle obtained from a previous open.
pRecordInfo	points to MM_RECORD_INFO (MM_PLAY_RECORD_INFO) structure.
pRuntimeControl	points to MM_RUNTIME_CONTROL structure.
pUserInfo	points to user-defined buffer. See MM_METAEVENT evtUserInfo field.

Detection of an I-frame (complete video frame) can be used to trigger recording. The API controls I-frame detection and its responses through the following:

Video Record Beep

On I-frame detection, transmit a beep to the party being recorded (start-of-recording notification tone). The video record beep is enabled by default and is transmitted upon detection of an I-frame or upon time-out waiting for an I-frame. The beep can be disabled by specifying the MM_MODE_VID_NOIFRMBEEPINITIATED bit in the MM_MEDIA_VIDEO unMode field.

I-Frame Time-Out Parameter

The EMM_REC_IFRAME_TIMEOUT parameter controls the time to wait for an I-frame. Video recording, or multimedia (audio and video) recording, starts when an I-frame is detected or when the time-out is reached. The default time-out is 5000 ms (5 seconds). To change this, specify the EMM_REC_IFRAME_TIMEOUT parameter in the MM_SET_PARM eParm field, specify a value in the unParmValue field, and call **mm_SetParm( )** to set it. A setting of 0 (zero) causes an immediate time-out and starts recording immediately.

To get the current setting of the I-frame time-out parameter, specify EMM_REC_IFRAME_TIMEOUT in the MM_GET_PARM eParm field and call **mm_GetParm( )**. When the function generates an MMEV_GETPARM event, the associated event data provides the current setting in the MM_GET_PARM_RESULT unParmValue field.

MMEV_VIDEO_RECORD_STARTED event

This ***optional*** **intermediate (non-terminating) notification event** is enabled by default and indicates the actual start of video recording (I-frame received or time-out waiting for an I-frame). This event can be disabled by **mm_DisableEvents( )** and enabled by **mm_EnableEvents( )** by specifying the MMR_EVENT_VIDEO_RECORD_STARTED bit in the MM_EVENTS unMask field.

The event data associated with the event indicates the status in the MM_VIDEO_RECORD_STARTED unStatus field. The field indicates EMM_VIDEO_RCRD_IFRAME_DETECTED for recording started due to I-frame detection, and EMM_VIDEO_RCRD_IFRAME_TIMEOUT for recording started due to I-frame time-out.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_RECORD_ACK

Initiation event reported upon successful start of the record operation.

Event Data: MM_RECORD_ACK structure (MM_RET_CODE)

MMEV_RECORD_ACK_FAIL
Initiation failure termination event reported upon encountering an error before the record operation begins.

Event Data: MM_RECORD_ACK structure (MM_RET_CODE)

MMEV_VIDEO_RECORD_STARTED
***Optional* intermediate (non-terminating) notification event** (enabled by default), indicating actual start of recording (complete video frame, or I-frame, received, or time-out waiting for an I-frame). This event can be disabled by **mm_DisableEvents( )** and enabled by **mm_EnableEvents( )**.

Event Data: MM_VIDEO_RECORD_STARTED structure

MMEV_VIDEO_RECORD_STARTED_FAIL (**reserved for future use**)
***Optional* intermediate (non-terminating) failure notification event** that is **reserved for future use** (there are no conditions that generate it).

MMEV_RECORD
Termination event reported upon successful completion or successful termination of the record operation.

Event Data: MM_RECORD_CMPLT structure (MM_PLAY_RECORD_CMPLT)

After a record operation has begun, an MMEV_RECORD event is reported to the application program when one of the following conditions occurs:

- the application program issues a stop operation with **mm_Stop( )**. See the **mm_Stop( )** function for conditions that generate no completion events or multiple completion events.
- the application program issues a reset operation with **mm_Reset( )**
- one of the runtime control conditions specified in the **pRuntimeControl** parameter has been met

MMEV_RECORD_FAIL
Operation failure termination event indicating operation errors reported upon completion or termination of the record operation. The errors that can cause this failure event are **intermediate (non-terminating) errors**.

Event Data: MM_RECORD_CMPLT structure (MM_PLAY_RECORD_CMPLT)

MMEV_ERROR
Unsolicited event reported upon encountering an unexpected failure of the record operation.

Event Data: MM_ERROR_RESULT structure

Note: For more information on these events, see Chapter 9, "Events". See also Section 12.5, "Terminating and Non-Terminating Play/Record Errors", on page 235 for useful information.

Cautions

See the MMEV_RECORD event above for conditions under which you can receive more than one MMEV_RECORD event.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure

event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

```
#include <mmlib.h>

void add_rtc_item(PMM_RUNTIME_CONTROL RTCList, int *RTCItemCountp, eMM_TERMINATION_REASON
reason, int value, eMM_TERMINATION_ACTION action)
{

        int index = *RTCItemCountp;

        if (index >= 1)
        {
                RTCList[index-1].next = &(RTCList[index]);
        }

        INIT_MM_RUNTIME_CONTROL(&(RTCList[index]));
        RTCList[index].Reason = reason;
        RTCList[index].unValue = value;
        RTCList[index].Action = action;
        RTCList[index].next = NULL;

        (*RTCItemCountp)++;

}

int main(int argc, char* argv[])
{

    /* . . . */
    /*
     * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to mm_Open( ).
     */
    MM_RECORD_INFO record_info;
    INIT_MM_RECORD_INFO(&record_info);
    MM_PLAY_RECORD_LIST recordlist[2];
    MM_MEDIA_ITEM_LIST mediaitemlist1[1];
    MM_MEDIA_ITEM_LIST mediaitemlist2[1];
    MM_VIDEO_CODEC VideoCodecType1;
    INIT_MM_VIDEO_CODEC(&VideoCodecType1);


    VideoCodecType1.Coding       = VIDEO_CODING_H263;
    VideoCodecType1.Profile      = VIDEO_PROFILE_0_H263;
    VideoCodecType1.Level        = VIDEO_LEVEL_10_H263;
    VideoCodecType1.ImageWidth   = VIDEO_IMAGE_WIDTH_176;
    VideoCodecType1.ImageHeight  = VIDEO_IMAGE_HEIGHT_144;
    VideoCodecType1.BitRate      = (eVIDEO_BITRATE)0;
    VideoCodecType1.FramesPerSec = VIDEO_FRAMESPERSEC_15;
    VideoCodecType1.SamplingRate = VIDEO_SAMPLING_RATE_90000;
    VideoCodecType1.VisualConfigSize     = 0;
    VideoCodecType1.VisualConfiguration  = NULL;

    MM_AUDIO_CODEC AudioCodecType1;
    INIT_MM_AUDIO_CODEC(&AudioCodecType1);
    AudioCodecType1.unCoding = MM_DATA_FORMAT_PCM;
    AudioCodecType1.unSampleRate = MM_DRT_8KHZ;
    AudioCodecType1.unBitsPerSample = 16;
    const char VideoFileName1[] = "/dir/file1.vid";
    const char AudioFileName1[] = "/dir/file3.aud";
    int cc;
    int xx;
    cc = 0;
    // Build Video Item 1
```

```
    INIT_MM_MEDIA_ITEM_LIST(&mediaitemlist1[cc]);

    INIT_MM_MEDIA_VIDEO(&mediaitemlist1[cc].item.video);
    mediaitemlist1[cc].ItemChain = EMM_ITEM_EOT;
    mediaitemlist2[cc].item.video.eFileFormat = EMM_FILE_FORMAT_PROPRIETARY;
    mediaitemlist1[cc].item.video.codec = VideoCodecType1;
    mediaitemlist1[cc].item.video.unMode = 0;
    mediaitemlist2[cc].item.audio.unOffset = 0;
    mediaitemlist1[cc].item.video.szFileName = VideoFileName1;

    mediaitemlist1[cc].item.video.unAccessMode=MM_MEDIA_ACCESS_MODE_FILE;
    cc++;
    xx = 0;
    // Add Video Items to the RecordList
    INIT_MM_PLAY_RECORD_LIST(&recordlist[xx]);
    recordlist[xx].ItemChain = EMM_ITEM_CONT;
    recordlist[xx].ItemType = EMM_MEDIA_TYPE_VIDEO;
    recordlist[xx].list = mediaitemlist1;
    xx++;
    cc = 0;
    // Build Audio Item 1

    INIT_MM_MEDIA_ITEM_LIST(&mediaitemlist2[cc]);

    INIT_MM_MEDIA_AUDIO(&mediaitemlist2[cc].item.audio);
    mediaitemlist2[cc].ItemChain = EMM_ITEM_EOT;
    mediaitemlist2[cc].item.audio.eFileFormat = EMM_AUD_FILEFORMAT_VOX;
    mediaitemlist2[cc].item.audio.codec = AudioCodecType1;
    mediaitemlist2[cc].item.audio.unMode = 0;
    mediaitemlist2[cc].item.audio.unOffset = 0;
    mediaitemlist2[cc].item.audio.szFileName = AudioFileName1;
    mediaitemlist2[cc].item.audio.unAccessMode-MM_MEDIA_ACCESS_MODE_FILE;
    cc++;
    // Add Audio Items to the PlayList
    INIT_MM_PLAY_RECORD_LIST(&recordlist[xx]);
    recordlist[xx].ItemChain = EMM_ITEM_EOT;
    recordlist[xx].ItemType = EMM_MEDIA_TYPE_AUDIO;
    recordlist[xx].list = mediaitemlist2;
    xx++;
    // Form Record Info
    INIT_MM_PLAY_RECORD_INFO(&record_info);
    record_info.list = recordlist;
    record_info.eFileFormat=EMM_FILE_FORMAT_UNDEFINED;

    MM_RUNTIME_CONTROL RTCList[3];
    int RTCItemCount = 0;

    //stop both audio and video recording 20 seconds after audio recording has started
    add_rtc_item(RTCList, &RTCItemCount, EMM_TERM_TIME_FROM_AUDIO_RECORD_STARTED, 20000,
                 EMM_TA_AUDIO_VIDEO_STOP);

    //as a backup termination condition: stop both audio and video recording 30 seconds after
      the MMEV_RECORD_ACK event
    add_rtc_item(RTCList, &RTCItemCount, EMM_TERM_MAXTIME, 30000, EMM_TA_AUDIO_VIDEO_STOP);

    // Initiate Record
    if (mm_Record(nDeviceHandle, &record_info, RTCList, NULL) == EMM_ERROR)
    {
        /* process error */
    }
}
```

See Also

- **mm_Play( )**
- **mm_Reset( )**
- **mm_Stop( )**

mm_Reset( )

Name: int mm_Reset(nDeviceHandle, pReset, pUserInfo)

Inputs:

int nDeviceHandle	• valid SRL handle representing a multimedia device
CPMM_RESET pReset	• set to NULL
void* pUserInfo	• pointer to user-defined buffer

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Input/Output

Mode: Asynchronous

Platform: Dialogic® HMP software

Description

This function terminates all active media on the device and resets the device state to idle. All previously set parameters are removed and the defaults are applied. The state of the device is equivalent to the state after the device was first opened, except that **mm_Reset( )** does not reset any device connections made through the Dialogic® Device Management API.

If this function stops an operation that is in progress (i.e., busy), it generates a termination event corresponding to the operation (e.g., MMEV_PLAY or MMEV_PLAY_FAIL) in addition to any reset-specific events (MMEV_RESET_ACK, MMEV_RESET_ACK_FAIL, MMEV_RESET, MMEV_RESET_FAIL). See Asynchronous Mode Events below.

Parameter	Description
nDeviceHandle	specifies a valid multimedia device handle obtained from a previous open
pReset	set to NULL; reserved for future use
pUserInfo	points to user-defined buffer. See MM_METAEVENT evtUserInfo field.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_RESET_ACK
Initiation event reported upon successful start of the reset operation.
Event Data: MM_RESET_ACK structure (MM_RET_CODE)

MMEV_RESET_ACK_FAIL
Initiation failure termination event reported upon encountering an error before the reset operation begins.
Event Data: MM_RESET_ACK structure (MM_RET_CODE)

MMEV_RESET
Termination event reported upon successful completion or successful termination of the reset operation.
Event Data: MM_RESET_RESULT structure (MM_RET_CODE)

MMEV_RESET_FAIL
Operation failure termination event reported upon encountering an error during the reset operation.
Event Data: MM_RESET_RESULT structure (MM_RET_CODE)

MMEV_PLAY
Termination event reported upon successful completion or successful termination of the play operation.
Event Data: MM_PLAY_CMPLT structure (MM_PLAY_RECORD_CMPLT)

MMEV_PLAY_FAIL
Operation failure termination event reported upon encountering an error during the play operation.
Event Data: MM_PLAY_CMPLT structure (MM_PLAY_RECORD_CMPLT)
Note: See Section 12.5, "Terminating and Non-Terminating Play/Record Errors", on page 235 for related information.

MMEV_RECORD
Termination event reported upon successful completion or successful termination of the record operation.
Event Data: MM_RECORD_CMPLT structure (MM_PLAY_RECORD_CMPLT)

MMEV_RECORD_FAIL
Operation failure termination event indicating operation errors reported upon completion or termination of the record operation. The record operation errors that can cause this failure event are **intermediate (non-terminating) errors**.
Event Data: MM_RECORD_CMPLT structure (MM_PLAY_RECORD_CMPLT)

MMEV_ERROR
Unsolicited event reported upon encountering an unexpected failure.
Event Data: MM_ERROR_RESULT structure

Cautions

- If user information is passed in the **mm_Reset( ) pUserInfo** parameter, it only gets passed to the metaevent data for the reset-specific events and not the media operation events, which can have their own user information associated with them.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

```
#include <mmlib.h>
int main(int argc, char* argv[])
{
   int nDeviceHandle; /* multimedia device handle */
   /*
   .
   .
   Main Processing
   .
   .
   */

   /*
    * Application is shutting down.
    * Need to close MM device handle.
    * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to mm_Open( ).
    */
   if (mm_Reset(nDeviceHandle, NULL, NULL) == EMM_ERROR)
   {
      /* process error */
   }
}
```

See Also

- **mm_Open( )**
- **mm_Close( )**

mm_ResultInfo( )

Name: int mm_ResultInfo(pMetaEvent, pInfo)

Inputs:

MM_METAEVENT pMetaEvent	• pointer to the MM_METAEVENT structure containing the metaevent data used as input
PMM_INFO pInfo	• pointer to the MM_INFO data structure where error information about the event is output

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Event Information

Mode: Synchronous

Platform: Dialogic® HMP software

Description

This function retrieves error information for certain Dialogic® Multimedia API events. This function uses a metaevent from the **mm_GetMetaEvent( )** function as input and outputs the error information in an MM_INFO structure.

The mmValue field of the MM_INFO structure provides an error code obtained internally by the Standard Runtime Library (SRL) **ATDV_LASTERR( )** function for the event to which the **pMetaEvent** parameter points, and the mmMsg field provides a descriptive error message obtained internally by the SRL **ATDV_ERRMSGP( )** function. The additionalInfo field is not set. For information on the SRL API, see the *Standard Runtime Library API Library Reference*.

To retrieve the correct information, the application must call **mm_GetMetaEvent( )** and **mm_ResultInfo( )** immediately after the failure event arrives and before the next Dialogic® Multimedia API event is requested, otherwise, the metaevent data may be outdated or invalid. For a list of the error codes and messages returned in the MM_INFO structure, see Section 12.2, "Dialogic® Multimedia API Function Error Codes", on page 231.

Parameter	Description
pMetaEvent	points to the MM_METAEVENT structure containing the metaevent data used as input which was acquired by **mm_GetMetaEvent( )** for the event
pInfo	points to the MM_INFO structure where error information about the event is output

Cautions

- The lifetime of the strings pointed to by the MM_INFO data structure is from the time the **mm_ResultInfo( )** function returns to the time the next event is requested.

Errors

If the **mm_ResultInfo( )** function fails, error analysis should **not** be done by calling the **mm_ErrorInfo( )** function. A failure return generally indicates invalid parameters or that **pInfo** is NULL.

Example

```
MM_METAEVENT metaevent;
INIT_MM_METAEVENT(&metaevent);
if (sr_waitevt(timeout) != -1)
{
    if (mm_GetMetaEvent(&metaevent) == EMM_ERROR)
    {
         /* get and process the error */
    }

    switch(metaevent.evttype)
    {
       /* Process Error Events */
        case MMEV_PLAY_FAIL:
        case MMEV_RECORD_FAIL:

          /* Get error associated with failure events */
           mm_ResultInfo(&mmMetaEvent, &mmInfo);
           printf("ERROR: 0x%X, ERROR MESSAGE: %s\n", mmInfo.mmValue, mmInfo.mmMsg);

         break;
     }
}
```

See Also

- None.

mm_Resume( )

Name: int mm_Resume(nDeviceHandle,pResumeInfo,* pUserInfo)

Inputs:

int nDeviceHandle	• Multimedia device handle
CPMM_RESUME_INFO pResumeInfo	• pointer to details of the resume request
void* pUserInfo	• pointer to user-defined data

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: DVR Control

Mode: Asynchronous

Platform: Dialogic® HMP software

Description

The **mm_Resume( )** function resumes playing a multimedia stream that was previously paused.

Parameter	Description
nDeviceHandle	valid SRL handle representing a multimedia device
pResumeInfo	specifies a pointer to the MM_RESUME_INFO structure that contains the details of the resume request
pUserInfo	points to user-defined buffer. See MM_METAEVENT evtUserInfo field.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_RESUME
Termination event reported upon successfully resuming play.
Event Data: MM_RESUME_PLAY_CMPLT structure

MMEV_RESUME_FAIL
Termination event reported upon encountering an error while attempting to resume play.
Event Data: MM_RESUME_PLAY_CMPLT structure

Cautions

- This function should only be called on multimedia streams that are currently paused. Otherwise, an error will occur.
- **mm_Resume( )** is only valid during the period of time between the MMEV_PLAY_ACK event and the MMEV_PLAY event for the requested stream type. See MM_RESUME_INFO for more information.
- **mm_Resume( )** is only supported for multimedia that is playing in file I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_FILE in **mm_Play( )**) and not in memory I/O or stream I/O mode.
- **mm_Resume( )** is only supported for files in the Dialogic® proprietary file format – when eFileFormat is set to EMM_FILE_FORMAT_PROPRIETARY or EMM_AUD_FILEFORMAT_PROPRIETARY in **mm_Play( )**.
- In an audiovideo playback, if the playback is synchronous (both audio and video play is initiated by a single **mm_Play( )** function call), **BOTH** the audio and video streams will be resumed, regardless of the unStreamType setting in MM_RESUME_INFO. This allows synchronization to be maintained.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

For an example of using the **mm_Resume( )** function, refer to the Example section for **mm_Pause( )**.

See Also

None.

mm_Seek( )

Name: int mm_Seek(nDeviceHandle, pSeekInfo, * pUserInfo)

Inputs:

int nDeviceHandle	• Multimedia device handle
CPMM_SEEK_INFO pSeekInfo	• pointer to details of the seek request
void* pUserInfo	• pointer to user-defined data

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: DVR Control

Mode: Asynchronous

Platform: Dialogic® HMP software

Description

The **mm_Seek( )** function sets the play position in a multimedia stream that is currently playing. For a video stream, the play position will be moved to the I-frame nearest (before or after) the indicated location.

Parameter	Description
nDeviceHandle	valid SRL handle representing a multimedia device
pSeekInfo	pointer to the MM_SEEK_INFO structure that contains the details of the request
pUserInfo	points to user-defined buffer. See MM_METAEVENT evtUserInfo field.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_SEEK

Termination event reported upon successfully setting the requested seek position.

Event Data: MM_SEEK_CMPLT structure

MMEV_SEEK_FAIL

Termination event reported upon encountering an error while attempting to set the requested seek position.

Event Data: MM_SEEK_CMPLT structure

Cautions

- If the video being played has significant time gaps between I-frames, the play position may be moved to a location that is significantly different from the requested position since the nearest I-frame could potentially be far from the requested position.
- Ensure that the unOffset field in MM_SEEK_INFO is set to a value that will not cause the seek position to go beyond the bounds of the file. Otherwise, an error will occur. For example, for a four second file if the application requests a seek position five seconds from the beginning of the file (unOrigin= EMM_SEEK_SET and nOffset=5000), an error will occur since it would be one second beyond the end of the file. Similarly, if we are currently two seconds from the beginning of the file and the application requests a seek position minus seven seconds from the current location (unOrigin= EMM_SEEK_CUR and nOffset=-7000), an error will occur since that would place the seek position five seconds before the start of the file.
- If **mm_Seek( )** is called on a stream that is currently paused, the multimedia will begin playing from the requested seek position only upon resuming the play.
- **mm_Seek( )** will only be valid during the period of time between the MMEV_PLAY_ACK event and the MMEV_PLAY event for the requested stream type. See MM_SEEK_INFO for more information.
- **mm_Seek( )** is only supported for multimedia that is playing in file I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_FILE in **mm_Play( )**) and not in memory I/O or stream I/O mode.
- **mm_Seek( )** is only supported for files in the Dialogic® proprietary file format – when eFileFormat is set to EMM_FILE_FORMAT_PROPRIETARY or EMM_AUD_FILEFORMAT_PROPRIETARY in **mm_Play( )**.
- In an audiovideo playback, if the playback is synchronous (both audio and video play is initiated by a single **mm_Play( )** function call), **BOTH** the audio and video stream play positions will be moved, regardless of the unStreamType setting in MM_SEEK_INFO. This allows synchronization to be maintained.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

For an example of using the **mm_Seek( )** function, refer to the Example section for **mm_Pause( )**.

See Also

None.

mm_SetParm( )

Name: int mm_SetParm(nDeviceHandle, pSetParm, pUserInfo)

Inputs:

int nDeviceHandle	• valid SRL handle representing a multimedia device
CPMM_SET_PARM pSetParm	• pointer to set parameters information structure
void* pUserInfo	• pointer to user-defined buffer

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Configuration

Mode: Asynchronous

Platform: Dialogic® HMP software

Description

This function sets the value of the specified configuration parameters. Parameters that are set remain in effect even after an **mm_Close( )**. However, an **mm_Reset( )** will reset all parameters to their default values.

Use the INIT_MM_SET_PARM( ) function to initialize the MM_SET_PARM structure.

Parameter	Description
nDeviceHandle	specifies a valid multimedia device handle obtained from a previous open
pSetParm	points to MM_SET_PARM structure
pUserInfo	points to user-defined buffer. See MM_METAEVENT evtUserInfo field.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_SETPARM

Termination event reported upon successful completion of the function.

Event Data: MM_SET_PARM_RESULT structure (MM_RET_CODE)

MMEV_SETPARM_FAIL

Operation failure termination event reported upon encountering an error during the operation.

Event Data: MM_SET_PARM_RESULT structure (MM_RET_CODE)

MMEV_ERROR

Unsolicited event reported upon encountering an unexpected failure.

Event Data: MM_ERROR_RESULT structure

Cautions

None.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

```
#include <mmlib.h>
int main(int argc, char* argv[])
{
    /* . . . */
    /*
    * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to mm_Open( ).
    */
    MM_SET_PARM parm;
    INIT_MM_SET_PARM(&parm);
    parm.eParm = MM_PARM_REC_IFRAME_TIMEOUT;
    parm.unParmValue = 2222;
    if (mm_SetParm(nDeviceHandle, &parm, NULL) == EMM_ERROR)
    {
        /* process error */}
    }
}
```

See Also

- **mm_GetParm( )**
- **mm_Reset( )**

mm_Stop( )

Name: int mm_Stop(nDeviceHandle, pStop, pUserInfo)

Inputs:

int nDeviceHandle	• valid SRL handle representing a multimedia device
CPMM_STOP pStop	• pointer to stop information structure
void* pUserInfo	• pointer to user-defined buffer

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Input/Output

Mode: Asynchronous

Platform: Dialogic® HMP software

Description

This function forces termination of specified multimedia I/O functions or operations on a channel. For example, this function can stop selected **mm_Play( )** or **mm_Record( )** operations (items) such as a video play, video record, audio play, and audio record. The operation or item type is specified in the MM_STOP structure. To stop all types of multimedia I/O operations on the channel, you must specify each item.

Use the INIT_MM_STOP( ) function to initialize the MM_STOP structure.

Parameter	Description
nDeviceHandle	specifies a valid multimedia device handle obtained from a previous open
pStop	points to MM_STOP structure
pUserInfo	points to user-defined buffer. See MM_METAEVENT evtUserInfo field.

If a specified operation (item) is not currently active on the channel, the function completes successfully for that operation but has no effect. If all possible operations are successfully stopped on a channel that is busy, it forces the channel to become idle. If the channel is already idle, it completes successfully but has no effect.

Regardless of whether the function stops an operation that is idle or in progress (i.e., busy), it generates an MM_STOP_ACK event to indicate successful initiation. If there are no busy operations, *this is the only event generated*. However, if a busy operation is stopped, it generates a corresponding event (e.g., MMEV_PLAY or MMEV_PLAY_FAIL) in addition to any stop-specific events (MMEV_STOP_ACK or MMEV_STOP_ACK_FAIL). See Asynchronous Mode Events below.

If the application program stops a selected media type (e.g., ItemType EMM_STOP_VIDEO_PLAY), it will receive an MMEV_PLAY event upon completion of the stop and the rest of the media types (if any; e.g., audio play) will continue to play until termination

or completion of the play, at which time, it will generate another MMEV_PLAY event. Therefore, *it is possible to receive multiple MMEV_PLAY events* resulting from one **mm_Play( )** function.

Similarly, if the program stops a selected recording media type (e.g., ItemType EMM_STOP_VIDEO_RECORD), it will receive an MMEV_RECORD event upon completion of the stop, and the rest of the media types (if any; e.g., audio record) will continue to record until termination or completion, at which time, it will generate another MMEV_RECORD event. Therefore, *it is possible to receive multiple MMEV_RECORD events* resulting from one **mm_Record( )** function.

If an error is encountered for any of the selected stop items, none of the operations will be stopped (no matter where in the list the error occurred). To identify which item caused the error, use the MM_STOP_ACK and MM_STOP_ACK_DETAILS event information structures associated with the MMEV_STOP_ACK_FAIL event. The MM_STOP_ACK_DETAILS structure gives the item type and the error return code for each item specified in MM_STOP. An EMMRC_OK error return code indicates that no error occurred for that particular operation.

Asynchronous Mode Events

Use the Dialogic® Standard Runtime Library (SRL) functions to process the events. Use the Dialogic® Multimedia API **mm_GetMetaEvent( )** Event Information function to retrieve the event information. Event data is indicated for the following events where applicable and can be obtained through the MM_METAEVENT structure evtdatap field after calling the **mm_GetMetaEvent( )** function. The event data is valid only until the next **mm_GetMetaEvent( )** is called. For more information on these events, see Chapter 9, “Events”.

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_STOP_ACK
: Initiation event reported upon successful start of the stop operation.

  Event Data: MM_STOP_ACK structure

MMEV_STOP_ACK_FAIL
: Initiation failure termination event reported upon encountering an error before the stop operation begins.

  Event Data: MM_STOP_ACK structure

MMEV_PLAY
: Termination event reported upon successful completion or successful termination of the play operation.

  Event Data: MM_PLAY_CMPLT structure (MM_PLAY_RECORD_CMPLT)

MMEV_PLAY_FAIL
: Operation failure termination event reported upon encountering an error during the play operation.

  Event Data: MM_PLAY_CMPLT structure (MM_PLAY_RECORD_CMPLT)

  Note: See Section 12.5, “Terminating and Non-Terminating Play/Record Errors”, on page 235 for related information.

MMEV_RECORD
: Termination event reported upon successful completion or successful termination of the record operation.

  Event Data: MM_RECORD_CMPLT structure (MM_PLAY_RECORD_CMPLT)

MMEV_RECORD_FAIL
: Operation failure termination event indicating operation errors reported upon completion or termination of the record operation. The record operation errors that can cause this failure event are **intermediate (non-terminating) errors**.

  Event Data: MM_RECORD_CMPLT structure (MM_PLAY_RECORD_CMPLT)

MMEV_ERROR
: Unsolicited event reported upon encountering an unexpected failure.

  Event Data: MM_ERROR_RESULT structure

Cautions

- If there are no busy operations, the MM_STOP_ACK event (indicating successful initiation) is the only event generated.
- If user information is passed in the **mm_Stop( ) pUserInfo** parameter, it only gets passed to the metaevent data for the stop-specific events. It does not get passed to the media operation events, which can have their own user information associated with them.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

```
#include <mmlib.h>
  int main(int argc, char* argv[])
{
    /* . . . */
    /*
    * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to mm_Open( ).
    */
    MM_STOP stop_info[4];
    INIT_MM_STOP(&stop_info[0]);
    stop_info[0].ItemChain = EMM_ITEM_CONT;
    stop_info[0].ItemType = EMM_STOP_VIDEO_PLAY;
    INIT_MM_STOP(&stop_info[1]);
    stop_info[1].ItemChain = EMM_ITEM_CONT;
    stop_info[1].ItemType = EMM_STOP_VIDEO_RECORD;
    INIT_MM_STOP(&stop_info[2]);
    stop_info[2].ItemChain = EMM_ITEM_CONT;
    stop_info[2].ItemType = EMM_STOP_AUDIO_PLAY;
    INIT_MM_STOP(&stop_info[3]);
    stop_info[3].ItemChain = EMM_ITEM_EOT;
    stop_info[3].ItemType = EMM_STOP_AUDIO_RECORD;
    if (mm_Stop(nDeviceHandle, stop_info, NULL) == EMM_ERROR)
    {
        /* process error */
    }
}
```

See Also

- **mm_Play( )**
- **mm_Record( )**
- **mm_Reset( )**

mm_StreamClose( )

Name: int mm_StreamClose (unStreamHandle)

Inputs: int nStreamHandle • stream device handle

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Media Streaming

Mode: Synchronous

Platform: Dialogic® HMP Software

Description

The **mm_StreamClose( )** function closes a previously opened media stream.

Parameter	Description
nStreamHandle	the stream device handle returned by **mm_StreamOpen( )**

Cautions

None.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

```
#include <stdio.h>
#include <mmlib.h>

#define DATASIZE 100

int recStreamHandle;
int playStreamHandle;
int nDeviceHandle; /* Multimedia device handle */

int main(void)
{
          MM_STREAM_OPEN_INFO streamOpenInfo;
          MM_STREAM_WATERMARK_INFO streamWaterMarkInfo;
          MM_RUNTIME_CONTROL runtimeControl;
          MM_MEDIA_ITEM_LIST mediaitemlist[1];
          MM_PLAY_INFO playInfo;
          MM_RECORD_INFO recordInfo;
```

```
/* Main Processing
.
    .
    .
    */

    //Open stream for video play
    INIT_MM_STREAM_OPEN_INFO(&streamOpenInfo);
    streamOpenInfo.unBufferSize = 1024;
    streamOpenInfo.BufferMode = EMM_SM_WRITE;

    playStreamHandle = mm_StreamOpen (&streamOpenInfo, NULL);
    if(playStreamHandle < 0)
    {
              /* process error */
    }

    //Open stream for video record
    streamOpenInfo.BufferMode = EMM_SM_READ;
    recStreamHandle = mm_StreamOpen (&streamOpenInfo, NULL);
    if(recStreamHandle < 0)
    {
            /* process error */
    }


   //Reset stream stats
   if (mm_StreamReset(playStreamHandle) == EMM_ERROR)
   {
            /* process error */
   }
   if (mm_StreamReset(recStreamHandle) == EMM_ERROR)
   {
            /* process error */
   }

   MM_STREAM_STAT recStreamStat;
   MM_STREAM_STAT playStreamStat;

   INIT_MM_STREAM_STAT(&recStreamStat);
   INIT_MM_STREAM_STAT(&playStreamStat);
   if (mm_StreamGetStat(playStreamHandle, &playStreamStat) == EMM_ERROR)
   {
            /* process error */
   }
   if (mm_StreamGetStat(recStreamHandle, &recStreamStat) == EMM_ERROR)
   {
            /* process error */
   }

   //Set low watermark level for video play stream to at least 20%
   if (playStreamStat.unLowWaterMark < (0.2 * playStreamStat.unBufferSize))
   {

            INIT_MM_STREAM_WATERMARK_INFO(&streamWaterMarkInfo);
            streamWaterMarkInfo.Level = EMM_WM_LOW;
           streamWaterMarkInfo.unValue = (unsigned int) (0.2 *
                    playStreamStat.unBufferSize);

          if (mm_StreamSetWaterMark (playStreamHandle, &streamWaterMarkInfo) ==
                    EMM_ERROR)
          {
                /* process error */
          }

}
//Set high watermark level for video record stream no higher than 80%
```

```
        if (recStreamStat.unHighWaterMark > (0.8 * recStreamStat.unBufferSize))
        {
            INIT_MM_STREAM_WATERMARK_INFO(&streamWaterMarkInfo);
            streamWaterMarkInfo.Level = EMM_WM_HIGH;
            streamWaterMarkInfo.unValue = (unsigned int) (0.8 * recStreamStat.unBufferSize);
            if (mm_StreamSetWaterMark (recStreamHandle, &streamWaterMarkInfo) == EMM_ERROR)
            {
                    /* process error */
            }
      }


       // Open multimedia device handle
       nDeviceHandle = mm_Open("mmB1C1", NULL, NULL);

       INIT_MM_MEDIA_VIDEO(&mediaitemlist[0].item.video);
       mediaitemlist[0].item.video.unAccessMode = MM_MEDIA_ACCESS_MODE_STREAM;
       mediaitemlist[0].item.video.stream.nStreamHandle = playStreamHandle;

      /* Setting up the rest of playInfo
      .
      .
      .
      */

      //Play video stream
      if (mm_Play(nDeviceHandle, &playInfo, NULL, NULL) == EMM_ERROR)
      {
               /* process error */
      }

       INIT_MM_MEDIA_VIDEO(&mediaitemlist[0].item.video);
       mediaitemlist[0].item.video.unAccessMode = MM_MEDIA_ACCESS_MODE_STREAM;
       mediaitemlist[0].item.video.stream.nStreamHandle = recStreamHandle;

       /* Setting up the rest of recordInfo
       .
       .
       .
       */

       INIT_MM_RUNTIME_CONTROL(&runtimeControl);
       runtimeControl.Reason = EMM_TERM_MAXTIME;
       runtimeControl.unValue = 100;
       runtimeControl.Action = EMM_TA_VIDEO_STOP;
       runtimeControl.next = NULL;

       //Record video stream
       if (mm_Record(nDeviceHandle, &recordInfo, &runtimeControl, NULL) == EMM_ERROR)
       {
               /* process error */
       }

       /* Application shutdown
       .
       .
       .
      */
      //Close video play & record streams
      mm_StreamClose (playStreamHandle);
      mm_StreamClose (recStreamHandle);

     //Close multimedia device
     mm_Close (nDeviceHandle, NULL);
 }

int SRL_EventHandler(...)
```

```
{
   int rc;
   int eventType;
   unsigned char *pData;
   unsigned int dataSize;
   unsigned char recordData[DATASIZE];
   unsigned int EndFlag = MM_FLAG_STREAM_CONT;
   /* Retrieve SRL event information
   .
   .
   .
   */

   switch(eventType)
   {
      /* Process other events
      .
      .
      .
      */

      //Take action (write more data to stream) if video play stream hits low watermark
      case MMEV_PLAY_VIDEO_LOWWATER:
         rc = EMM_SUCCESS;
         while ((rc != EMM_STREAM_FULL) && (rc != EMM_ERROR))
         {
            /* Get dataSize bytes from storage
            .
            .
            .
            */
            /* Set EndFlag to MM_FLAG_STREAM_EOD if last block of data
            .
            .
            .
            */
            rc = mm_StreamWrite(playStreamHandle, pData, dataSize, EndFlag);
            if (rc == EMM_ERROR)
            {
               /* process error */
            }
         } /* end while */
         break;

      case MMEV_RECORD_VIDEO_HIGHWATER:
         //Take action (read data out of stream) if video record stream
         //hits high watermark
         rc = EMM_SUCCESS;
         while ((EndFlag != MM_FLAG_STREAM_EOD) && (rc != EMM_ERROR)){
            dataSize = DATASIZE;
            rc = mm_StreamRead(recStreamHandle, recordData, &dataSize, &EndFlag);
            if (rc == EMM_ERROR){
               /* process error */
            }
            else if ( rc != EMM_STREAM_EMPTY ) {
               /* Save dataSize bytes to storage
               .
               .
               .
               */
            }
            /* If rc == EMM_STREAM_EMPTY then the application has two options:
             *
             * Option 1: Ignore EMM_STREAM_EMPTY status and continue polling the
             * stream for data until then continue polling the stream for the data
             * until EndFlag is set to MM_FLAG_STREAM_EOD.
             *
```

```
                * Option 2: Exit this loop logic and wait until MMEV_RECORD_VIDEO_HIGHWATER
                * or MMEV_RECORD_VIDEO_EOD event is received. Once one of these events is
                * the application should start reading the stream again.
                *
                * Using the combination of MMEV_RECORD_VIDEO_HIGHWATER and MMEV_RECORD_VIDEO_EOD
                * events allows the application to be written to be completely event driven
                * and eliminate the need for stream polling.
                */
            } /* end while */
            break;

        case MMEV_RECORD_VIDEO_EOD:
            //Take action (read data out of stream) if video record stream indicates
            //end of data (EOD)
            /* When EOD event is received it is guaranteed that the mm_StreamRead()
             * will not return EMM_STREAM_EMPTY status until EndFlag is set to
             * MM_FLAG_STREAM_EOD.
             */

            rc = EMM_SUCCESS;
            while ((EndFlag != MM_FLAG_STREAM_EOD) && (rc != EMM_ERROR)) {
               dataSize = DATASIZE;
               rc = mm_StreamRead(recStreamHandle, recordData, &dataSize, &EndFlag);
               if (rc == EMM_ERROR) {
                  /* process error */
               }
               /* Save dataSize bytes to storage
               .
               .
               .
               */
            } /* end while */
            break;

        } /* end switch */

} /* end SRL_EventHandler */
```

See Also

- **mm_StreamOpen( )**
- **mm_StreamSetWaterMark( )**

mm_StreamGetStat( )

Name:	int mm_StreamGetStat (unStreamHandle, pStreamStat)	
Inputs:	int nStreamHandle	• stream device handle
	PMM_STREAM_STAT pStreamStat	• internal stream buffer statistics
Returns:	EMM_SUCCESS if successful EMM_ERROR if failure	
Includes:	mmlib.h	
Category:	Media Streaming	
Mode:	Synchronous	
Platform:	Dialogic® HMP software	

Description

The **mm_StreamGetStat( )** function allows to retrieve internal stream statistics.

Use the INIT_MM_STREAM_STAT( ) inline function to initialize the MM_STREAM_STAT structure.

Parameter	Description
nStreamHandle	the stream device handle returned by **mm_StreamOpen( )**
pStreamStat	points to the MM_STREAM_STAT structure, which displays the internal stream buffer statistics

Cautions

None.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, “Error Codes” for the function error codes and event information error return codes.

Example

Refer to the Example section for **mm_StreamClose( )**.

See Also

None.

mm_StreamOpen( )

Name: int mm_StreamOpen (pStreamOpenInfo, pUserInfo)

Inputs: CPMM_STREAM_OPEN_INFO pStreamOpenInfo — pointer to stream open parameter structure

void *pUserInfo — set to NULL

Returns: Stream device handle if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Media Streaming

Mode: Synchronous; streaming flow control is performed asynchronously

Platform: Dialogic® HMP software

Description

The **mm_StreamOpen( )** function creates a media stream to use with the **mm_Play( )** and **mm_Record( )** functions.

Use the INIT_MM_STREAM_OPEN_INFO( ) function to initialize the MM_STREAM_OPEN_INFO structure.

Parameter	Description
pStreamOpenInfo	points to MM_STREAM_OPEN_INFO structure
pUserInfo	set to NULL; reserved for future use

Cautions

The stream device handle returned by **mm_StreamOpen( )** is not an SRL handle and cannot be used in sr_setparm. Thus, the SR_USERCONTEXT (if any) associated with events such as MMEV_RECORD_VIDEO_HIGHWATER will be the one associated with the device that is playing or recording, rather than the stream itself.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

Refer to the Example section for **mm_StreamClose( )**.

See Also

- **mm_StreamClose( )**
- **mm_StreamSetWaterMark( )**

mm_StreamRead( )

Name: int mm_StreamRead (unStreamHandle, pData, pDataSize, pEndFlag)

Inputs:

int nStreamHandle	• stream device handle
unsigned char *pData	• pointer to a block of data
unsigned int *pDataSize	• pointer to size of data block
unsigned int *pEndFlag	• pointer to end flag

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure
EMM_STREAM_EMPTY if stream is empty

Includes: mmlib.h

Category: Media Streaming

Mode: Synchronous only

Platform: Dialogic® HMP software

Description

The **mm_StreamRead( )** function reads data from the selected media stream. This function is used to stream data for a multimedia record operation.

Parameter	Description
nStreamHandle	the stream device handle returned by **mm_StreamOpen( )**
pData	points to a block of data allocated by the application. Data that is read from the stream will be placed into this block.
pDataSize	this is an input/output parameter. The application sets this parameter to indicate the number of bytes to read from the media stream. When the function returns, this parameter will be modified to indicate the actual number of bytes read from the media stream.
pEndFlag	points to end flag. When the function returns, *pEndFlag will be set to either MM_FLAG_STREAM_EOD for last block of data or MM_FLAG_STREAM_CONT otherwise.

Events

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_RECORD_AUDIO_EOD
Optional notification event (disabled by default). Indicates that the last data byte was inserted into the video streaming buffer. The data can be read, using **mm_StreamRead( )**, until a returned *pEndFlag value is set to MM_FLAG_STREAM_EOD.

MMEV_RECORD_AUDIO_LOWWATER
Audio streaming buffer reached its low watermark. Data can be read until the return code is EMM_STREAM_EMPTY or returned *pEndFlag value is set to MM_FLAG_STREAM_EOD .

MMEV_RECORD_AUDIO_HIGHWATER
: Audio streaming buffer reached its high watermark; data must be read as soon as possible from the specified stream to avoid an overrun condition. Data can be read until the return code is EMM_STREAM_EMPTY or returned *pEndFlag value is set to MM_FLAG_STREAM_EOD.

MMEV_RECORD_VIDEO_EOD
: Optional notification event (disabled by default). Indicates that the last data byte was inserted into the video streaming buffer. The data can be read, using **mm_StreamRead( )**, until a returned *pEndFlag value is set to MM_FLAG_STREAM_EOD.

MMEV_RECORD_VIDEO_LOWWATER
: Video streaming buffer reached its low watermark. Data can be read until the return code is EMM_STREAM_EMPTY or returned *pEndFlag value is set to MM_FLAG_STREAM_EOD.

MMEV_RECORD_VIDEO_HIGHWATER
: Video streaming buffer reached its high watermark; data must be read as soon as possible from the specified stream to avoid an overrun condition. Data can be read until the return code is EMM_STREAM_EMPTY or returned *pEndFlag value is set to MM_FLAG_STREAM_EOD.

Cautions

- The specified media stream has to be opened for read access mode (EMM_SM_READ).
- The events for this function can arrive in any order.
- To flush the stream buffer after **mm_Record( )** completes, check the pEndFlag parameter or the return value of the function.
- The application must ensure that the amount of data requested, pDataSize, does not exceed the size of the data block pointed to by pData.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, “Error Codes” for the function error codes and event information error return codes.

Example

Refer to the Example section for **mm_StreamClose( )**.

See Also

None.

mm_StreamReset( )

Name: int mm_StreamReset (unStreamHandle)

Inputs: int nStreamHandle • stream device handle

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Media Streaming

Mode: Synchronous

Platform: Dialogic® HMP software

Description

The **mm_StreamReset( )** function resets a previously opened media stream. As a result of calling this function, the read/write pointers are set to the beginning of the buffer. Also, the buffer content and the collected metrics are reset to zero.

Note: The **mm_StreamReset( )** function does not reset the stream watermark.

Parameter	Description
nStreamHandle	the stream device handle returned by **mm_StreamOpen( )**

Cautions

None.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

Refer to the Example section for **mm_StreamClose( )**.

See Also

None.

mm_StreamSetWaterMark( )

Name: int mm_StreamSetWaterMark (unStreamHandle, pStreamStat)

Inputs:

int nStreamHandle	• stream device handle
CPMM_STREAM_WATERMARK_INFO pStreamWaterMarkInfo	• parameter structure for setting a specific watermark level

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure

Includes: mmlib.h

Category: Media Streaming

Mode: Synchronous

Platform: Dialogic® HMP software

Description

The **mm_StreamSetWaterMark( )** function allows the application to set a high or low watermark for media streaming flow control purposes.

Use the INIT_MM_STREAM_WATERMARK_INFO( ) function to initialize the MM_STREAM_WATERMARK_INFO structure.

Parameter	Description
nStreamHandle	the stream device handle returned by **mm_StreamOpen( )**
pStreamWaterMarkInfo	points to MM_STREAM_WATERMARK_INFO, which sets specific watermark levels

Notes

The **mm_StreamSetWaterMark( )** function is an optional function for media streaming. The default high and low watermarks are respectively 90 percent and 10 percent of the stream buffer size.

Cautions

None.

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

Refer to the Example section for **mm_StreamClose( )**.

See Also

None.

mm_StreamWrite( )

Name: int mm_StreamWrite (unStreamHandle, pData, unDataSize, unEndFlag)

Inputs:

int nStreamHandle	• stream device handle
unsigned char *pData	• pointer to a block of data
unsigned int unDataSize	• size of data block in bytes
unsigned int unEndFlag	• end of a block of data

Returns: EMM_SUCCESS if successful
EMM_ERROR if failure
EMM_STREAM_FULL if the end of the streaming buffer has been reached

Includes: mmlib.h

Category: Media Streaming

Mode: Synchronous

Platform: Dialogic® HMP software

Description

The **mm_StreamWrite( )** function allows the application to write data in the selected media stream. This function can be used to stream data for a multimedia play operation.

Parameter	Description
nStreamHandle	the stream device handle returned by **mm_StreamOpen( )**
pData	points to a block of data
unDataSize	specifies the size. in bytes, of data block pointed to by pData
unEndFlag	flag used to indicate end of data. Set to either: MM_FLAG_STREAM_EOD for last block of data MM_FLAG_STREAM_CONT otherwise

Synchronous Mode Events

If the function returns EMM_SUCCESS, it can generate any of the following events:

MMEV_PLAY_AUDIO_LOWWATER
: Audio streaming buffer reached its low watermark. Data must be written immediately to the specified stream to avoid an underrun condition. Data can be written until the return code is EMM_STREAM_FULL.

MMEV_PLAY_AUDIO_HIGHWATER
: Audio streaming buffer reached its high watermark. Data can be written until the return code is EMM_STREAM_FULL.

MMEV_PLAY_VIDEO_LOWWATER
: Video streaming buffer reached its low watermark. Data must be written immediately to the specified stream to avoid an underrun condition. Data can be written until the return code is EMM_STREAM_FULL.

MMEV_PLAY_VIDEO_HIGHWATER
: Video streaming buffer reached its high watermark. Data can be written until the return code is EMM_STREAM_FULL.

Cautions

The specified media stream has to be opened for write access mode (EMM_SM_WRITE).

Errors

If a Dialogic® Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. If the function generates a failure event, use the **mm_GetMetaEvent( )** function to obtain the error information. See Chapter 12, "Error Codes" for the function error codes and event information error return codes.

Example

Refer to the Example section for **mm_StreamClose( )**.

See Also

None.

Events 9

This chapter provides information about the events that are generated by the Dialogic® Multimedia API functions.

- Overview of Dialogic® Multimedia API Events 137
- Dialogic® Multimedia API Event Types 138
- Dialogic® Multimedia API Event Types by Function Type 139
- Dialogic® Multimedia API Events 139

9.1 Overview of Dialogic® Multimedia API Events

An event indicates that a specific activity has occurred on a channel. The Dialogic® Multimedia API reports channel activity to the application program in the form of events, which allows the program to identify and respond to a specific occurrence on a channel. Events provide feedback on the progress and completion of functions and indicate the occurrence of other channel activities. Events are sometimes referred to according to the type of event, such as initiation event, intermediate event, notification event, termination or completion event, unsolicited event, success event, and failure event. These types are not always mutually exclusive. The most common type of event is one that reports on the result of function operations. Typically, each function generates different events, and the functions documented in Chapter 8, "Function Information" describe the events applicable to them.

To collect an event code, use **sr_waitevt( )**, **sr_enbhdlr( )** or other Standard Runtime Library (SRL) function, depending upon the programming model in use. For detailed information on event handling and management, see the *Dialogic® Standard Runtime Library API Library Reference* and the *Dialogic® Standard Runtime Library API Programming Guide*.

The Dialogic® Multimedia API Event Information functions are listed in Section 7.7, "Event Information Functions", on page 68. The **mm_GetMetaEvent( )** function maps the current SRL event into an MM_METAEVENT data structure, which contains explicit data describing the event. This data mechanism helps to provide uniform information retrieval among libraries.

For Dialogic® Multimedia API events, see the **mm_GetMetaEvent( )** function and the MM_METAEVENT structure for the specific multimedia information provided. This mechanism can also be used for non-Multimedia API events, for which the MM_METAEVENT structure provides the device descriptor, the event type, a pointer to variable length event data, and the length of the event data. No additional SRL calls are required to access event data, because all the data associated with any type of event are accessible via the MM_METAEVENT structure.

The **mm_EnableEvents( )** and **mm_DisableEvents( )** functions allow you to enable and disable optional notification events, such as the MMEV_VIDEO_RECORD_STARTED event, which indicates the actual start of recording. See Table 7, "Optional Notification Events", on page 139 for the list of optional notification events that can be enabled or disabled.

9.2 Dialogic® Multimedia API Event Types

The Dialogic® Multimedia API can generate the following types of events:

Initiation Events

Format: **MMEV_xxxx_ACK** (where "xxxx" is the name of the function or operation)

These intermediate events are generated upon successful start of a function. For example, MMEV_PLAY_ACK indicates the successful start of the **mm_Play( )** function. The event data provides information on the reason for the event.

Initiation Failure Termination Events

Format: **MMEV_xxxx_ACK_FAIL** (where "xxxx" is the name of the function or operation)

These termination events are generated upon encountering an error before the main operation of a function begins. For example, MMEV_PLAY_ACK_FAIL indicates that a failure occurred before starting the **mm_Play( )** playback operation. The event data provides error information on the reason for the failure.

Completion or Successful Termination Events

Format: **MMEV_xxxx** (where "xxxx" is the name of the function or operation)

These termination events are generated upon the successful completion or successful termination of a function operation. For example, MMEV_PLAY indicates successful completion of the playback operation being performed by **mm_Play( )**; e.g., the end of file was reached. The deliberate termination of an I/O function operation by **mm_Stop( )** or **mm_Reset( )** can also produce this type of event; for example, MMEV_PLAY can indicate the successful termination of an **mm_Play( )** playback operation by **mm_Stop( )**. The event data provides result information on the reason for the completion or termination event.

Operation Failure Termination Events

Format: **MMEV_xxxx_FAIL** (where "xxxx" is the name of the function or operation)

These termination events are generated upon encountering an error during the main operation of a function. (Or, in the in the case of MMEV_RECORD_FAIL, upon completion or termination of the operation.) For example, MMEV_PLAY_FAIL indicates failure of the playback operation being performed by **mm_Play( )**. The deliberate termination of an I/O function operation by **mm_Stop( )** or **mm_Reset( )** can also produce this type of event; for example, MMEV_PLAY_FAIL can indicate termination of an **mm_Play( )** playback operation by **mm_Stop( )**. The event data provides result information on the reason for the completion or termination event.

Optional Notification Events

These events are enabled ~~by default and are enabled~~ and disabled by the **mm_EnableEvents( )** and **mm_DisableEvents( )** functions. These events are generated upon encountering the condition specified by the event; for example, the MMEV_VIDEO_RECORD_STARTED optional **intermediate notification event** reports the the actual start of recording (detection of an I-frame or a time-out waiting for an I-frame). The MMEV_VIDEO_RECORD_STARTED_FAIL failure event that is associated with this operation is **reserved for future use** (there are no conditions that generate it). See Table 7, "Optional Notification Events", on page 139 for the list of the optional events and their default enable/disable status.

Unsolicited Events

These events are not requested by the application. They are triggered by, and provide information about, internal or external events. For example, MMEV_ERROR indicates an unexpected failure. The event data provides error result information.

Note: For more information on failure events, see Chapter 12, "Error Codes".

Table 7. Optional Notification Events

Event	Default	Event mask name inside MM_EVENTS
MMEV_VIDEO_RECORD_STARTED	Enabled	MMR_EVENT_VIDEO_RECORD_STARTED
MMEV_RECORD_VIDEO_EOD	Disabled	MMR_EVENT_RECORD_VIDEO_EOD
MMEV_RECORD_AUDIO_EOD	Disabled	MMR_EVENT_RECORD_AUDIO_EOD

9.3 Dialogic® Multimedia API Event Types by Function Type

To identify the specific events that a particular function can generate, see the "Asynchronous Mode Events" section of that function under Chapter 8, "Function Information".

The following summarizes the type of events that different types of functions can generate:

- All asynchronous functions are subject to unsolicited events (e.g., MMEV_ERROR).
- All asynchronous functions can generate a successful termination/completion event or a failure termination event corresponding to the function operation (e.g., MMEV_PLAY and MMEV_PLAY_FAIL).

  Note: One exception is the **mm_Stop( )** function. Successful termination or completion is reported by MMEV_PLAY or MMEV_RECORD events, and failure is reported by MMEV_PLAY_FAIL or MMEV_RECORD_FAIL.
- All functions that initiate or terminate I/O, including **mm_Play( )**, **mm_Record( )**, **mm_Stop( )**, and **mm_Reset( )**, can generate a successful initiation event (which is an intermediate event) or a corresponding initiation failure termination event (e.g., MMEV_PLAY_ACK and MMEV_PLAY_ACK_FAIL).
- Only specific functions or operations generate optional notification events (e.g., MMEV_VIDEO_RECORD_STARTED applies only to **mm_Record( )** operation).

9.4 Dialogic® Multimedia API Events

The Dialogic® Multimedia API can generate the following events (listed in alphabetical order):

MMEV_CAPTURE

Termination event reported upon successful completion of a capture still image operation.

Event Data: MM_CAPTURE_CMPLT_DETAILS structure

MMEV_CAPTURE_FAIL
Operation failure termination event reported upon encountering an error during the operation.
Event Data: MM_CAPTURE_CMPLT_DETAILS structure

MMEV_DISABLEEVENTS
Termination event reported upon successful completion of the function.
Event Data: MM_DISABLE_EVENTS_RESULT structure (MM_RET_CODE)

MMEV_DISABLEEVENTS_FAIL
Operation failure termination event reported upon encountering an error during the operation.
Event Data: MM_DISABLE_EVENTS_RESULT structure (MM_RET_CODE)

MMEV_ENABLEEVENTS
Termination event reported upon successful completion of the function.
Event Data: MM_ENABLE_EVENTS_RESULT structure (MM_RET_CODE)

MMEV_ENABLEEVENTS_FAIL
Operation failure termination event reported upon encountering an error during the operation.
Event Data: MM_ENABLE_EVENTS_RESULT structure (MM_RET_CODE)

MMEV_ERROR
Unsolicited failure event reported upon encountering an unexpected failure.
Event Data: MM_ERROR_RESULT structure

MMEV_GETDURATION
Termination event reported upon successfully getting the requested duration.
Event Data: MM_GETDURATION_CMPLT structure

MMEV_GETDURATION_FAIL
Termination event reported upon encountering an error while getting the requested duration.
Event Data: MM_GETDURATION_CMPLT structure

MMEV_GETELAPSEDTIME
Termination event reported upon successfully getting the requested elapsed time.
Event Data: MM_GETELAPSEDTIME_CMPLT structure

MMEV_GETELAPSEDTIME_FAIL
Termination event reported upon encountering an error while getting the requested elapsed time.
Event Data: MM_GETELAPSEDTIME_CMPLT structure

MMEV_GETPARM
Termination event reported upon successful completion of the function.
Event Data: MM_GET_PARM_RESULT structure

MMEV_GETPARM_FAIL
Operation failure termination event reported upon encountering an error during the operation.
Event Data: MM_GET_PARM_RESULT structure

MMEV_OPEN
Termination event reported upon successful completion of the function.
Event Data: MM_OPEN_RESULT structure (MM_RET_CODE)

MMEV_OPEN_FAIL
Operation failure termination event reported upon encountering an error during the operation.

Event Data: MM_OPEN_RESULT structure (MM_RET_CODE)

Note: The application program must call **mm_Close( )** to clean up after this failure.

MMEV_PAUSE
Termination event reported upon successfully pausing.

Event Data: MM_PAUSE_PLAY_CMPLT structure

MMEV_PAUSE_FAIL
Termination event reported upon encountering an error while attempting to pause.

Event Data: MM_PAUSE_PLAY_CMPLTstructure

MMEV_PLAY
Termination event reported upon successful completion or successful termination of the operation.

Event Data: MM_PLAY_CMPLT structure (MM_PLAY_RECORD_CMPLT)

MMEV_PLAY_ACK
Initiation event reported upon successful start of the function.

Event Data: MM_PLAY_ACK structure (MM_RET_CODE)

MMEV_PLAY_ACK_FAIL
Initiation failure termination event reported upon encountering an error before the operation begins.

Event Data: MM_PLAY_ACK structure (MM_RET_CODE)

MMEV_PLAY_AUDIO_HIGHWATER
Audio streaming buffer reached its high watermark. Data can be written until the return code is EMM_STREAM_FULL.

MMEV_PLAY_AUDIO_LOWWATER
Audio streaming buffer reached its low watermark. Data must be written immediately to the specified stream to avoid an underrun condition. Data can be written until the return code is EMM_STREAM_FULL.

MMEV_PLAY_FAIL
Operation failure termination event reported upon encountering an error during the operation.

Event Data: MM_PLAY_CMPLT structure (MM_PLAY_RECORD_CMPLT)

Note: See Section 12.5, “Terminating and Non-Terminating Play/Record Errors”, on page 235 for related information.

MMEV_PLAY_VIDEO_HIGHWATER
Video streaming buffer reached its high watermark. Data can be written until the return code is EMM_STREAM_FULL.

MMEV_PLAY_VIDEO_LOWWATER
Video streaming buffer reached its low watermark. Data must be written immediately to the specified stream to avoid an underrun condition. Data can be written until the return code is EMM_STREAM_FULL.

MMEV_RECORD
Termination event reported upon successful completion or successful termination of the operation.

Event Data: MM_RECORD_CMPLT structure (MM_PLAY_RECORD_CMPLT)

MMEV_RECORD_ACK
Initiation event reported upon successful start of the function.

Event Data: MM_RECORD_ACK structure (MM_RET_CODE)

MMEV_RECORD_ACK_FAIL
Initiation failure termination event reported upon encountering an error before the operation begins.

Event Data: MM_RECORD_ACK structure (MM_RET_CODE)

MMEV_RECORD_AUDIO_EOD
Optional notification event (disabled by default). Indicates that the last data byte was inserted into the video streaming buffer. The data can be read, using mm_StreamRead( ), until a returned *pEndFlag value is set to MM_FLAG_STREAM_EOD.

MMEV_RECORD_AUDIO_HIGHWATER
Audio streaming buffer reached its high watermark; data must be read as soon as possible from the specified stream to avoid an overrun condition. Data can be read until the return code is EMM_STREAM_EMPTY or a returned *pEndFlag value is set to MM_FLAG_STREAM_EOD.

MMEV_RECORD_AUDIO_LOWWATER
Audio streaming buffer reached its low watermark. Data can be read until the return code is EMM_STREAM_EMPTY.

MMEV_RECORD_FAIL
Operation failure termination event indicating operation errors reported upon completion or termination of the operation. The operation errors that can cause this failure event are **intermediate (non-terminating) errors**.

Event Data: MM_RECORD_CMPLT structure (MM_PLAY_RECORD_CMPLT)

Note: See Section 12.5, "Terminating and Non-Terminating Play/Record Errors", on page 235 for related information.

MMEV_RECORD_VIDEO_EOD
Optional notification event (disabled by default). Indicates that the last data byte was inserted into the video streaming buffer. The data can be read, using mm_StreamRead( ), until a returned *pEndFlag value is set to MM_FLAG_STREAM_EOD.

MMEV_RECORD_VIDEO_HIGHWATER
Video streaming buffer reached its high watermark; data must be read as soon as possible from the specified stream to avoid an overrun condition. Data can be read until the return code is EMM_STREAM_EMPTY or a returned *pEndFlag value is set to MM_FLAG_STREAM_EOD.

MMEV_RECORD_VIDEO_LOWWATER
Video streaming buffer reached its low watermark. Data can be read until the return code is EMM_STREAM_EMPTY.

MMEV_RESET
Termination event reported upon successful completion or successful termination of the operation.
Event Data: MM_RESET_RESULT structure (MM_RET_CODE)

MMEV_RESET_ACK
Initiation event reported upon successful start of the function.
Event Data: MM_RESET_ACK structure (MM_RET_CODE)

MMEV_RESET_ACK_FAIL
Initiation failure termination event reported upon encountering an error before the operation begins.
Event Data: MM_RESET_ACK structure (MM_RET_CODE)

MMEV_RESET_FAIL
Operation failure termination event reported upon encountering an error during the operation.
Event Data: MM_RESET_RESULT structure (MM_RET_CODE)

MMEV_RESUME
Termination event reported upon successfully resuming play.
Event Data: MM_RESUME_PLAY_CMPLT structure

MMEV_RESUME_FAIL
Termination event reported upon encountering an error while attempting to resume play.
Event Data: MM_RESUME_PLAY_CMPLT structure

MMEV_SEEK
Termination event reported upon successfully setting the requested seek position.
Event Data: MM_SEEK_CMPLT structure

MMEV_SEEK_FAIL
Termination event reported upon encountering an error while attempting to set the requested seek position.
Event Data: MM_SEEK_CMPLT structure

MMEV_SETPARM
Termination event reported upon successful completion of the function.
Event Data: MM_SET_PARM_RESULT structure (MM_RET_CODE)

MMEV_SETPARM_FAIL
Operation failure termination event reported upon encountering an error during the operation.
Event Data: MM_SET_PARM_RESULT structure (MM_RET_CODE)

MMEV_STOP_ACK
Initiation event reported upon successful start of the function.
Event Data: MM_STOP_ACK structure

MMEV_STOP_ACK_FAIL
Initiation failure termination event reported upon encountering an error before the operation begins.
Event Data: MM_STOP_ACK structure

MMEV_VIDEO_RECORD_STARTED

Optional **intermediate (non-terminating) notification event** (enabled by default), indicating actual start of recording (complete video frame, or I-frame, received, or time-out waiting for an I-frame). This event can be disabled by **mm_DisableEvents( )** and enabled by **mm_EnableEvents( )**.

Event Data: MM_VIDEO_RECORD_STARTED structure

MMEV_VIDEO_RECORD_STARTED_FAIL (**reserved for future use**)

Optional **intermediate (non-terminating) failure notification event** that is **reserved for future use** (there are no conditions that generate it).

Data Structure Types 10

This chapter describes the *types* of data structures supported by the Dialogic® Multimedia API, including the basic categories, naming convention for specific types, their purpose, hierarchy or nesting levels, and association with specific functions.

- Overview of Dialogic® Multimedia API Data Structures 145
- Data Structures for Function I/O 146
- Data Structures for Analyzing Event Information. 147
- Play/Record Data Structure Levels 149
- Other Data Structure Levels (_DETAILS) 149
- Data Structures By Function 150

Note: For an alphabetical list of data structures followed by a comprehensive reference organized alphabetically and providing detailed information on the structures, see Chapter 11, "Data Structures".

10.1 Overview of Dialogic® Multimedia API Data Structures

The Dialogic® Multimedia API data structures are defined in the *mmlib.h* header file. These data structures are used to control the operation of functions and to obtain information about events.

Note: For each data structure, use the corresponding INIT_ inline function required by the application for initialization. For example, use the INIT_MM_MEDIA_ACCESS_MEMORY inline function to initialize the MM_MEDIA_ACCESS_MEMORY data structure.

Dialogic® Multimedia API Data Structure Categories

There are two basic categories relating to how data structures are used in the :

Data Structures for Function I/O

This category of data structure is used directly by the function as either **input** or **output**, depending upon the function. For example, the **pStop** parameter in the **mm_Stop( )** function points to an MM_STOP data structure that is used for input to the function.

A data structure may also be *nested* within the top level function I/O data structure. For example, the MM_STOP input data structure contains a details field that is a data structure of type MM_STOP_DETAILS, which is thus a 2nd level in the hierarchy. Additional *nesting levels* in data structure hierarchy are possible, as in the **mm_Play( )** and **mm_Record( )** functions, which reach up to 6 levels (see Section 10.4, "Play/Record Data Structure Levels", on page 149). See also Section 10.5, "Other Data Structure Levels (_DETAILS)", on page 149.

Data Structures for Analyzing Event Information

This category of data structure is used to analyze event data provided by the **mm_GetMetaEvent( )** function in the MM_METAEVENT data structure evtdatap field. For example, **mm_GetMetaEvent( )** function gets the event data for the MMEV_ENABLE_EVENTS or MMEV_ENABLE_EVENTS_FAIL function completion events through an MM_ENABLE_EVENTS_RESULT (MM_RET_CODE) data structure, which provides result information on the function termination.

A data structure may also be *nested* within the top level structure of event data. For example, the MM_PLAY_RECORD_CMPLT structure for event data from a play or record termination event contains a details field that is a data structure of type MM_PLAY_RECORD_CMPLT_DETAILS, which is thus a second level in the hierarchy. Additional *nesting levels* in this type of information hierarchy are possible.

10.2 Data Structures for Function I/O

The following information is a guide to understanding the naming convention for, and purpose of, data structures that are used for *function input or output*. This information is organized by the type of data structure, as identified in the name, usually through a suffix. In the following list, a name is given for a data structure type, where "xxxx" represents the name of the function, operation, or event which applies to the data structure.

MM_xxxx_**INFO** (function input/output information)

This type of data structure is used *primarily* for function **input**, but in some cases may be used for function output. For example, the MM_INFO data structure is used for **output** by the **mm_ErrorInfo( )** function. The following data structures are included in this type:

- MM_INFO – The **pInfo** parameter in the **mm_ErrorInfo( )** function points to an MM_INFO data structure that is used for **output** by the functions.
- MM_PLAY_INFO – Used as function input for **mm_Play( )**, it is of type MM_PLAY_RECORD_INFO.
- MM_PLAY_RECORD_INFO – This data structure is used for input by the **mm_Play( )** and **mm_Record( )** functions, and it contains data structures nested in a multi-level hierarchy within it. For a description of these *nesting levels*, see Section 10.4, "Play/Record Data Structure Levels", on page 149. It is used as a typedef for the MM_RECORD_INFO and MM_PLAY_INFO structures.
- MM_RECORD_INFO – Used as function input for **mm_Record( )**, it is of type MM_PLAY_RECORD_INFO.

MM_**xxxx** (function input/output)

This type of data structure is used *primarily* for function **input**, but in some cases may be used for function output. For example, the MM_METAEVENT data structure is used for **output** by the **mm_GetMetaEvent( )** function. The following data structures are included in this type:

- MM_EVENTS – The **pEvents** parameter in the **mm_EnableEvents( )** function and the **mm_DisableEvents( )** function points to an MM_EVENTS data structure that is used for **input** by the functions.
- MM_METAEVENT – The **pMetaEvent** parameter in the **mm_GetMetaEvent( )** function points to an MM_METAEVENT data structure that is used for **output** by **mm_GetMetaEvent( )**.
- MM_GET_PARM – Used as function input for **mm_GetParm( )**.
- MM_SET_PARM – Used as function input for **mm_SetParm( )**.

- MM_STOP – Used as function input for **mm_Stop( )**. It contains a second-level MM_STOP_DETAILS structure that is **reserved for future use**.

10.3 Data Structures for Analyzing Event Information

Note: See Chapter 9, "Events" for additional information on events and event data.

The following information is a guide to understanding the naming convention for, and purpose of, data structures that are used for *analyzing event information*. This information is organized by the type of data structure, as identified in the name, usually through a suffix. In the following list, a name is given for a data structure type, where "xxxx" represents the name of the function, operation, or event which applies to the data structure.

MM_xxxx_**ACK** (function initiation event data)

This type of data structure is used for analyzing the results of a *function initiation event*. For example, the MM_PLAY_ACK data structure is used to analyze the result of an **mm_Play( )** function initiation as reported by the MMEV_PLAY_ACK or MMEV_PLAY_ACK_FAIL event. The following data structures are included in this type:

- MM_PLAY_ACK (MM_RET_CODE)*
- MM_RECORD_ACK (MM_RET_CODE)*
- MM_RESET_ACK (MM_RET_CODE)*
- MM_STOP_ACK
- MM_STOP_ACK_DETAILS

Note: The MM_STOP_ACK_DETAILS data structure is a second-level event data structure under the MM_STOP_ACK structure.

*These structures are of type MM_RET_CODE.

MM_xxxx_**CMPLT** (media function termination event data)

This type of data structure is used for analyzing the results of a *media function termination or completion event*. For example, the MM_PLAY_CMPLT data structure is used to analyze the result of an **mm_Play( )** function as reported by the MMEV_PLAY or MMEV_PLAY_FAIL event. The following data structures are included in this type:

- MM_PLAY_CMPLT (MM_PLAY_RECORD_CMPLT)*
- MM_PLAY_RECORD_CMPLT
- MM_RECORD_CMPLT (MM_PLAY_RECORD_CMPLT)*
- MM_PLAY_RECORD_CMPLT_DETAILS

Note: The MM_PLAY_RECORD_CMPLT_DETAILS data structure is a second-level event data structure under the MM_PLAY_RECORD_CMPLT structure.

*These structures are of type MM_PLAY_RECORD_CMPLT.

MM_xxxx_**DETAILS** (additional event data details)

This type of data structure is used for analyzing *additional details of event information*. For example, the MM_PLAY_RECORD_CMPLT_DETAILS data structure is used to analyze the result of an **mm_Play( )** or **mm_Record( )** function as reported by the MMEV_PLAY, MMEV_PLAY_FAIL, MMEV_RECORD, or MMEV_RECORD_FAIL event. The following data structures are included in this type (these data structures encompass a sub-type because

they provide details for a higher-level structure; see also Section 10.5, "Other Data Structure Levels (_DETAILS)", on page 149):

- MM_STOP_ACK_DETAILS

  Note: The MM_STOP_ACK_DETAILS data structure is a second-level event data structure under the MM_STOP_ACK structure.

- MM_PLAY_RECORD_CMPLT_DETAILS

  Note: The MM_PLAY_RECORD_CMPLT_DETAILS data structure is a second-level event data structure under the MM_PLAY_RECORD_CMPLT structure.

MM_xxxx_**RESULT** (results of a termination or completion event)

This type of data structure is used for analyzing the *results of a termination or completion event*, primarily non-media function operation events. For example, the MM_ENABLE_EVENTS_RESULT data structure is used to analyze the result of an **mm_EnableEvents( )** function as reported by the MMEV_ENABLEEVENTS or MMEV_ENABLEEVENTS_FAIL event. The following data structures are included in this type:

- MM_DISABLE_EVENTS_RESULT (MM_RET_CODE)*
- MM_ENABLE_EVENTS_RESULT (MM_RET_CODE)*
- MM_ERROR_RESULT

  Note: The MM_ERROR_RESULT data structure is used to analyze the causes of an MMEV_ERROR unsolicited failure event.

- MM_GET_PARM_RESULT
- MM_OPEN_RESULT (MM_RET_CODE)*
- MM_RESET_RESULT (MM_RET_CODE)*
- MM_SET_PARM_RESULT (MM_RET_CODE)*

*These structures are of type MM_RET_CODE

MM_**RET_CODE** (error information event data structure)

The MM_RET_CODE event information data structure describes error return code information related to an event. (See Section 12.3, "Multimedia API Event Information Error Return Codes", on page 232 for a list of the error return codes and related information.) This structure is used as a typedef for the following event information data structures.

ACK event information structures:

1. MM_PLAY_ACK
2. MM_RECORD_ACK
3. MM_RESET_ACK

RESULT event information structures:

4. MM_DISABLE_EVENTS_RESULT
5. MM_ENABLE_EVENTS_RESULT
6. MM_OPEN_RESULT
7. MM_RESET_RESULT
8. MM_SET_PARM_RESULT

Note: Error codes are also returned in the MM_STOP_ACK_DETAILS unRetCode field, which is used to provide details on the MM_STOP_ACK event data associated with the MMEV_STOP_ACK and MMEV_STOP_ACK_FAIL events.

10.4 Play/Record Data Structure Levels

The following hierarchy (starting at the function parameter level) illustrates the *nesting levels* for the function input data structures used by the **mm_Play( )** and **mm_Record( )** functions. The hierarchy is shown here by indenting the nested data structures and indicating the nesting level with a level number.

```
mm_Play( )( nDeviceHandle, pPlayInfo, pRuntimeControl, pUserInfo )
mm_Record( )( nDeviceHandle, pRecordInfo, pRuntimeControl, pUserInfo )

The pPlayInfo and pRecordInfo parameters point to the first level data structure.

1. MM_PLAY_RECORD_INFO (MM_PLAY_INFO, MM_RECORD_INFO)*

      2. MM_PLAY_RECORD_LIST list

            3. MM_MEDIA_ITEM_LIST list

                  4. MM_MEDIA_ITEM item

                     5. MM_MEDIA_VIDEO video

                           6. MM_VIDEO_CODEC codec

                     5. MM_MEDIA_AUDIO audio

                           6. MM_AUDIO_CODEC codec

                     5. MM_MEDIA_IMAGE image

                           6. MM_IMAGE_FORMAT image format

                     5. MM_MEDIA_TERM term  [reserved for future use]
```

The MM_PLAY_INFO structure pointed to by **pPlayInfo, and the MM_RECORD_INFO structure pointed to by **pRecordInfo**, are both of typedef MM_PLAY_RECORD_INFO.*

10.5 Other Data Structure Levels (_DETAILS)

The following data structures also contain a hierarchy, or *nesting level*s, but they are limited to a more simple, ***two-level hierarchy***:

- The MM_PLAY_RECORD_CMPLT_DETAILS data structure is a second-level event information data structure under the MM_PLAY_RECORD_CMPLT (MM_PLAY_CMPLT, MM_RECORD_CMPLT) structure. These event information data structures are used for analyzing the results of a *media function termination or completion event* (MMEV_PLAY, MMEV_PLAY_FAIL, MMEV_RECORD, and MMEV_RECORD_FAIL). Both MM_PLAY_CMPLT and MM_RECORD_CMPLT are of typedef MM_PLAY_RECORD_CMPLT.
- The MM_STOP_ACK_DETAILS data structure is a second-level event information data structure under the MM_STOP_ACK structure. These event information data structures are used for analyzing the results of the MMEV_STOP_ACK or MMEV_STOP_ACK_FAIL function initiation events produced by the **mm_Stop( )** function.

- The MM_STOP_DETAILS data structure (**reserved for future use**) is a second-level function input data structure under the MM_STOP structure. These data structures provide input for the **mm_Stop( )** function.

10.6 Data Structures By Function

The following list identifies the data structures associated with specific functions, either as input or through event information. Some of the data structures are associated with more than one function.

Note: For an alphabetical list of data structures followed by a comprehensive reference organized alphabetically and providing detailed information on the structures, see Chapter 11, "Data Structures".

The parenthetical codes in the following list represent the type of structure: **E** for *event information*, **FI** for *function input*, **FO** for f*unction output*, and **#** (a number) for nesting level when the structure occurs in a hierarchy. For example, FI2 indicates a second-level function input structure.

The parenthetical codes in the following list represent the type of structure:

- **FI** for *function input*,
- **FO** for f*unction output*,
- **E** for *event information*,
- **L#** for *nesting level number* (when the structure occurs in a hierarchy)
  For example, FI L2 indicates a function input second-level structure.

All Asynchronous Functions

- MM_ERROR_RESULT (E)

mm_DisableEvents( )

- MM_EVENTS (FI)
- MM_DISABLE_EVENTS_RESULT (MM_RET_CODE) (E)

mm_EnableEvents( )

- MM_EVENTS (FI)
- MM_ENABLE_EVENTS_RESULT (MM_RET_CODE) (E)

mm_ErrorInfo( )

- MM_INFO (FO)

mm_GetMetaEvent( )

- MM_METAEVENT (FO)

mm_GetParm( )

- MM_GET_PARM (FI)

- MM_GET_PARM_RESULT (E)

mm_Open( )

- MM_OPEN_RESULT (MM_RET_CODE) (E)

mm_Play( )

- MM_PLAY_INFO (MM_PLAY_RECORD_INFO) (FI L1)
- MM_PLAY_RECORD_LIST (FI L2)
- MM_MEDIA_ITEM_LIST (FI L3)
- MM_MEDIA_ITEM (FI L4)
- MM_MEDIA_VIDEO (FI L5)
- MM_VIDEO_CODEC (FI L6)
- MM_MEDIA_AUDIO (FI L5)
- MM_AUDIO_CODEC (FI L6)
- MM_AUDIO_CODEC_OPTION_LIST
- MM_MEDIA_TERM (FI L5) (**reserved for future use**)
- MM_PLAY_ACK (MM_RET_CODE) (E)
- MM_PLAY_CMPLT (MM_PLAY_RECORD_CMPLT) (E L1)
- MM_PLAY_RECORD_CMPLT_DETAILS (E L2)

mm_Record( )

- MM_RECORD_INFO (MM_PLAY_RECORD_INFO) (FI L1)
- MM_PLAY_RECORD_LIST (FI L2)
- MM_MEDIA_ITEM_LIST (FI L3)
- MM_MEDIA_ITEM (FI L4)
- MM_MEDIA_VIDEO (FI L5)
- MM_VIDEO_CODEC (FI L6)
- MM_MEDIA_AUDIO (FI L5)
- MM_AUDIO_CODEC (FI L6)
- MM_AUDIO_CODEC_OPTION_LIST
- MM_MEDIA_TERM (FI L5) (**reserved for future use**)
- MM_RECORD_ACK (MM_RET_CODE) (E)
- MM_RECORD_CMPLT (MM_PLAY_RECORD_CMPLT) (E L1)
- MM_PLAY_RECORD_CMPLT_DETAILS (E L2)
- MM_VIDEO_RECORD_STARTED (E)

mm_Reset( )

- MM_RESET_ACK (MM_RET_CODE) (E)
- MM_RESET_RESULT (MM_RET_CODE) (E)

- MM_PLAY_RECORD_CMPLT) (E L1)
- MM_PLAY_RECORD_CMPLT_DETAILS (E L2)

mm_SetParm( )

- MM_SET_PARM (FI)
- MM_SET_PARM_RESULT (MM_RET_CODE) (E)

mm_Stop( )

- MM_STOP (FI)
- MM_STOP_DETAILS (FI L2) (**reserved for future use**)
- MM_STOP_ACK (E L1)
- MM_STOP_ACK_DETAILS (E L2)
- MM_PLAY_RECORD_CMPLT) (E L1)
- MM_PLAY_RECORD_CMPLT_DETAILS (E L2)

Data Structures 11

This chapter provides a list of data structures followed by a comprehensive data structure reference organized alphabetically and providing detailed information on structures.

Note: See Chapter 10, "Data Structure Types" for a description of the *types* of data structures supported by the Multimedia API, including the basic categories, naming convention for specific types, their purpose, hierarchy or nesting levels, and association with specific functions.

- MM_AUDIO_CODEC 156
- MM_AUDIO_CODEC_OPTION_LIST 158
- MM_CAPTURE_CMPLT_DETAILS 162
- MM_ERROR_RESULT 163
- MM_EVENTS 164
- MM_GET_PARM 165
- MM_GET_PARM_RESULT 166
- MM_GETDURATION_CMPLT 168
- MM_GETDURATION_CMPLT_DETAILS 167
- MM_GETDURATION_INFO 169
- MM_GETELAPSEDTIME_CMPLT 170
- MM_GETELAPSEDTIME_CMPLT_DETAILS 171
- MM_GETELAPSEDTIME_INFO 172
- MM_IMAGE_FORMAT 173
- MM_INFO 174
- MM_MEDIA_ACCESS_MEMORY 175
- MM_MEDIA_ACCESS_STREAM 176
- MM_MEDIA_AUDIO 177
- MM_MEDIA_ITEM 181
- MM_MEDIA_ITEM_LIST 182
- MM_MEDIA_TERM 183
- MM_MEDIA_VIDEO 184
- MM_METAEVENT 186
- MM_PAUSE_INFO 188
- MM_PAUSE_PLAY_CMPLT 189
- MM_PAUSE_PLAY_CMPLT_DETAILS 190

- MM_PLAY_RECORD_CMPLT 191
- MM_PLAY_RECORD_CMPLT_DETAILS 192
- MM_PLAY_RECORD_INFO 195
- MM_PLAY_RECORD_LIST 196
- MM_RECORD_CMPLT_DATA_BLOCK 198
- MM_RECORD_CMPLT_INFO_DATA_BLOCKS 200
- MM_RESUME_INFO 201
- MM_RESUME_PLAY_CMPLT 202
- MM_RESUME_PLAY_CMPLT_DETAILS 203
- MM_RET_CODE 204
- MM_RUNTIME_CONTROL 206
- MM_SEEK_CMPLT 208
- MM_SEEK_CMPLT_DETAILS 209
- MM_SEEK_INFO 213
- MM_SET_PARM 210
- MM_STOP 211
- MM_STOP_ACK 215
- MM_STOP_ACK_DETAILS 216
- MM_STOP_DETAILS 217
- MM_STREAM_OPEN_INFO 218
- MM_STREAM_STAT 219
- MM_STREAM_WATERMARK_INFO 221
- MM_VIDEO_CODEC 222
- MM_VIDEO_RECORD_STARTED 227
- MM_YUV 228

Note: The following data structures are not listed under their own name but under the name of their typedef structure. The following list shows the type followed by a list of structure names that are of that type:

MM_PLAY_RECORD_INFO
MM_PLAY_INFO
MM_RECORD_INFO

MM_PLAY_RECORD_CMPLT
MM_PLAY_CMPLT
MM_RECORD_CMPLT

MM_RET_CODE
- MM_PLAY_ACK
- MM_RECORD_ACK
- MM_RESET_ACK

MM_RET_CODE
- MM_DISABLE_EVENTS_RESULT
- MM_ENABLE_EVENTS_RESULT
- MM_OPEN_RESULT
- MM_RESET_RESULT
- MM_SET_PARM_RESULT

MM_AUDIO_CODEC

```
typedef struct tagMM_AUDIO_CODEC
{
    unsigned int     unVersion;
    unsigned int     unCoding;
    unsigned int     unSampleRate;
    unsigned int     unBitsPerSample;
    CPMM_AUDIO_CODEC_OPTION_LIST options;
} MM_AUDIO_CODEC, *PMM_AUDIO_CODEC;
typedef const MM_AUDIO_CODEC* CPMM_AUDIO_CODEC
```

Description

The MM_AUDIO_CODEC structure specifies the characteristics of the audio coder. This structure is a nested *function input* structure for the **mm_Play( )** or **mm_Record( )** function. It is nested directly under the MM_MEDIA_AUDIO structure, and it is a sixth-level structure under the MM_PLAY_RECORD_INFO (MM_PLAY_INFO, MM_RECORD_INFO) structure. (For a description of these *nesting levels*, see Section 10.4, "Play/Record Data Structure Levels", on page 149.)

The INIT_MM_AUDIO_CODEC inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion

Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unCoding

Specifies the type of audio coding. A comprehensive list of values is provided below.

Note: Support for coders varies by Dialogic® Platform software release. Refer to Section 2.3, "Dialogic® Multimedia API Audio Formats by Platform", on page 31 for more information.

- MM_DATA_FORMAT_PCM (or Voice API: DATA_FORMAT_PCM) – Linear PCM, mono, LSB-MSB ("little endian")
- MM_DATA_FORMAT_ALAW – Alaw PCM
- MM_DATA_FORMAT_MULAW – µ-Law PCM
- MM_DATA_FORMAT_AMR_NB_4_75K – GSM AMR-NB, 4.75 kbps
- MM_DATA_FORMAT_AMR_NB_5_15K – GSM AMR-NB, 5.15 kbps
- MM_DATA_FORMAT_AMR_NB_5_90K – GSM AMR-NB, 5.9 kbps
- MM_DATA_FORMAT_AMR_NB_6_70K – GSM AMR-NB, 6.7 kbps
- MM_DATA_FORMAT_AMR_NB_7_40K – GSM AMR-NB, 7.4 kbps
- MM_DATA_FORMAT_AMR_NB_7_95K – GSM AMR-NB, 7.95 kbps
- MM_DATA_FORMAT_AMR_NB_10_20K – GSM AMR-NB, 10.2 kbps
- MM_DATA_FORMAT_AMR_NB_12_20K – GSM AMR-NB, 12.2 kbps
- MM_DATA_FORMAT_G723_1_5_30K – G.723.1, 5.3 kbps
- MM_DATA_FORMAT_G723_1_6_30K – G.723.1, 6.3 kbps
- MM_DATA_FORMAT_G726 – G726 Coder

- MM_DATA_FORMAT_G729A – G729a Coder

Note: Field validation is performed on the unCoding field. Setting an unsupported value results in an MMEV_PLAY_ACK_FAIL event in **mm_Play( )** and **mm_Record( )**.

unSampleRate

Specifies the audio sampling rate in samples per second. Defined value is:

- MM_DRT_8KHZ (or Voice API: DRT_8KHZ) – 8000 samples per second

unBitsPerSample

Specifies the number of bits per audio sample. Defined values are:

- 16
- 8
- 4

options

Points to the list of audio coder options. These definitions are found in the MM_AUDIO_CODEC_OPTION_LIST structure. Set to NULL if no options need to be specified.

Note: The Multimedia API supports the Linear PCM (128 kbps) audio file format. This audio file format can also be used in the Voice API by setting the DX_XPB structure fields to the following values:

- wFileFormat = FILE_FORMAT_VOX
- wDataFormat = DATA_FORMAT_PCM
- nSamplesPerSecond = DRT_8KHZ
- wBitsPerSample = 16

MM_AUDIO_CODEC_OPTION_LIST

```
typedef struct tag MM_AUDIO_CODEC_OPTION_LIST
{
        unsigned int              unVersion
        eMM_ITEM                  ItemChain;
        eMM_AUDIO_CODEC_OPTION    eParm;
        int                       nValue;
        struct tag MM_AUDIO_CODEC_OPTION_LIST* next;
        struct tagMM_AUDIO_CODEC_OPTION_LIST* prev;       /* optional */
}MM_AUDIO_CODEC_OPTION_LIST, *PMM_AUDIO_CODEC_OPTION_LIST;
typedef const MM_AUDIO_CODEC_OPTION_LIST* CPMM_AUDIO_CODEC_OPTION_LIST;
```

Description

The MM_AUDIO_CODEC_OPTION_LIST structure is pointed to from the MM_AUDIO_CODEC structure. This structure provides additional coder options for the specified coder. If no options need to be specified, set the value to NULL; otherwise use the address of a valid MM_AUDIO_CODEC_OPTION_LIST list.

The INIT_MM_AUDIO_CODEC_OPTION_LIST inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion

Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

ItemChain

Specifies the next list element for iteration. Defined valid values include:

- EMM_ITEM_CONT – Indicates that the next list element immediately follows the current list
- EMM_ITEM_LINK – Indicates that the next and previous list elements link to the current list via pointers
- EMM_ITEM_EOT – Indicates that the current list element is the last in the list

eParm

Identifies the option that needs to be set. Defined valid values include:

- EMM_AUDIO_CODEC_OPTION_UNDEFINED – No option will be set.
- EMM_AUDIO_CODEC_OPTION_INTERLEAVE_LENGTH – Used to set interleave length for EVRC coders.
- EMM_AUDIO_CODEC_OPTION_CDMA_RATE_REDUC – Used to set rate reduction for EVRC and QCELP coders.
- EMM_AUDIO_CODEC_OPTION_PAYLOAD_FORMAT – Used to set payload format for AMR coders.

nValue

Value for the option specified in eParm. This set of valid values depends on the coder and the option specified in eParm for that coder.

For AMR:

where eParm= EMM_AUDIO_CODEC_OPTION_PAYLOAD_FORMAT, possible values are:

- EMM_AUDIO_PAYLOAD_FORMAT_OCTET_ALIGNED – octet aligned frame format
- EMM_AUDIO_PAYLOAD_FORMAT_BANDWIDTH_EFFICIENT – –bandwidth efficient frame format

For EVCR:

- where eParm= EMM_AUDIO_CODEC_OPTION_INTERLEAVE_LENGTH, possible values are in the range 0 to 7. Default value is 0.
- where eParm= EMM_AUDIO_CODEC_OPTION_CDMA_RATE_REDUC, possible values are 0 and 4. Default value is 0.

For QCELP 8 kbps:

- where eParm = EMM_AUDIO_CODEC_OPTION_CDMA_RATE_REDUC, possible values are 0 and 4. Default value is 0.

For QCELP 13 kbps:

- where eParm = EMM_AUDIO_CODEC_OPTION_CDMA_RATE_REDUC, possible values are 0, 1, 2, 3 and 4. Default value is 0.

next

Points to the next MM_AUDIO_CODEC_OPTION_LIST item. Needed only when ItemChain specifies a linked list element. NULL value indicates last item.

prev

Points to the previous MM_AUDIO_CODEC_OPTION_LIST item (optional). Needed only when ItemChain specifies a linked list element. NULL value indicates first item.

Example

The following code illustrates the use of multiple audio options passed in as a linked list.

```
/*******************************************************************************************/
void destroy_audio_options(PMM_AUDIO_CODEC_OPTION_LIST optionList)
{

    PMM_AUDIO_CODEC_OPTION_LIST curOption = optionList;
    PMM_AUDIO_CODEC_OPTION_LIST nextOption = 0;

    while (curOption)
    {
        nextOption = curOption->next;
        delete(curOption);
        curOption = nextOption;
    }
}
/*******************************************************************************************/
void add_audio_option(PMM_AUDIO_CODEC_OPTION_LIST &firstOption, PMM_AUDIO_CODEC_OPTION_LIST
&curOption,
                      eMM_AUDIO_CODEC_OPTION option, int optionValue)
{
    PMM_AUDIO_CODEC_OPTION_LIST prevOption = curOption;
    PMM_AUDIO_CODEC_OPTION_LIST newOption = new(MM_AUDIO_CODEC_OPTION_LIST);
    INIT_MM_AUDIO_CODEC_OPTION_LIST(newOption);
```

```
    INIT_MM_AUDIO_CODEC_OPTION_LIST(newOption);
    newOption->eParm = option;
    newOption->nValue = optionValue;
    if (!curOption)
    {
        firstOption = curOption = newOption;
    }
    else
    {
        curOption->ItemChain = EMM_ITEM_LINK;
        curOption = curOption->next = newOption;
    }

}
/*****************************************************************************/
void add_audio_item(PMM_MEDIA_ITEM_LIST AudioItemList, int *AudioItemCountp, MM_AUDIO_CODEC
AudioCodec, char *AudioFileName)
{
    int index = *AudioItemCountp;

    INIT_MM_MEDIA_ITEM_LIST (&AudioItemList[index]);
    INIT_MM_MEDIA_AUDIO(&(AudioItemList[index].item.audio));


    if (index >= 1)
    {
        AudioItemList[index-1].ItemChain = EMM_ITEM_CONT;
    }
        AudioItemList[index].ItemChain = EMM_ITEM_EOT;
    AudioItemList[index].item.audio.codec = AudioCodec;
    AudioItemList[index].item.audio.unMode = 0;
    AudioItemList[index].item.audio.unOffset = 0;
    AudioItemList[index].item.audio.szFileName = AudioFileName;
    AudioItemList[index].item.audio.eFileFormat = EMM_AUD_FILEFORMAT_VOX;
    AudioItemList[index].item.audio.unAccessMode = MM_MEDIA_ACCESS_MODE_FILE;

    (*AudioItemCountp)++;
}
/*****************************************************************************/
void add_playrecordlist_item(PMM_PLAY_RECORD_LIST PlayRecordList, int *PlayRecordListItemCount,
CPMM_MEDIA_ITEM_LIST PlayRecordListItem, eMM_MEDIA_TYPE MediaType)
{
    int index = *PlayRecordListItemCount;
    INIT_MM_PLAY_RECORD_LIST(&(PlayRecordList[index]));

    if (index >= 1)
    {
        PlayRecordList[index-1].ItemChain = EMM_ITEM_CONT;
    }

    PlayRecordList[index].ItemChain = EMM_ITEM_EOT;
    PlayRecordList[index].ItemType  = MediaType;
    PlayRecordList[index].list      = PlayRecordListItem;

    (*PlayRecordListItemCount)++;
}
/*****************************************************************************/
int main(int argc, char *argv[])
{
    MM_PLAY_RECORD_INFO PlayRecordInfo;
    MM_PLAY_RECORD_LIST PlayRecordList[2];
    MM_MEDIA_ITEM_LIST AudioItemList[1];

    MM_AUDIO_CODEC AudioCodec;

    INIT_MM_AUDIO_CODEC (&AudioCodec);
```

```
    AudioCodec.unCoding = MM_DATA_FORMAT_EVRC;

    PMM_AUDIO_CODEC_OPTION_LIST curOption, firstOption;
    curOption = firstOption = (PMM_AUDIO_CODEC_OPTION_LIST)AudioCodec.options;

    add_audio_option(firstOption, curOption, EMM_AUDIO_CODEC_OPTION_INTERLEAVE_LENGTH, 2);
    add_audio_option(firstOption, curOption, EMM_AUDIO_CODEC_OPTION_CDMA_RATE_REDUC, 4);

    AudioCodec.options = firstOption;

    char AudioFileName[] = "clip1a.pcm";

    int AudioItemCount = 0;
    int PlayRecordListItemCount = 0;

    add_audio_item(AudioItemList, &AudioItemCount, AudioCodec, AudioFileName);
    add_playrecordlist_item(PlayRecordList, &PlayRecordListItemCount, AudioItemList,
EMM_MEDIA_TYPE_AUDIO);
    PlayRecordInfo.list = PlayRecordList;

    rc = mm_Record(mmdevh, &PlayRecordInfo, NULL, NULL);
    /*.
    .
    .

      wait for MMEV_RECORD event
    */

    destroy_audio_options((PMM_AUDIO_CODEC_OPTION_LIST)AudioCodec.options);
    return 0;
}
```

MM_CAPTURE_CMPLT_DETAILS

```
typedef struct tagMM_CAPTURE_CMPLT_DETAILS
{
         unsigned int unVersion;
         unsigned int unRetCode;
         unsigned int unNumberOfBytes;
         eMTK_IMAGE_FORMAT eFormat;
         eMTK_YUV_IMAGE_FORMAT eYuvFormat;
         unsigned int  unWidth;
         unsigned int  unHeight;

} MM_CAPTURE_CMPLT_DETAILS, *PMM_CAPTURE_CMPLT_DETAILS;
typedef const MM_CAPTURE_CMPLT_DETAILS* CPMM_CAPTURE_CMPLT_DETAILS;typedef struct
```

Description

The MM_CAPTURE_CMPLT_DETAILS event information data structure is used for analyzing the results of the **mm_Capture( )** function termination or completion as reported by an MMEV_CAPTURE or MMEV_CAPTURE_FAIL event.

Field Descriptions

The fields of the data structure are described as follows:

unVersion
: Specifies the version of the data structure.

unRetCode
: Provides an error return code. EMMRC_OK indicates success. All other values indicate an error or failure condition. Refer to Section 12.3, “Multimedia API Event Information Error Return Codes”, on page 232 for related information.

unNumberOfBytes
: Specifies the total number of bytes captured.

eFormat
: Specifies the image format. Valid value is eMTK_IMAGE_FORMAT_YUV.

eYuvFormat
: Specifies the YUV specific format if the image format is YUV. Valid value is eMTK_YUV_IMAGE_FORMAT_420.

unWidth
: Specifies the width of the image captured in pixels per line.

unHeight
: Specifies the height of the image captured in number of lines.

MM_ERROR_RESULT

```
typedef struct tag MM_ERROR_RESULT
{
    unsigned int    unVersion;
    unsigned int    unErrorCode;
    unsigned int    unErrorMsg;
    unsigned int    unData[4];
} MM_ERROR_RESULT, *PMM_ERROR_RESULT;
typedef const MM_ERROR_RESULT* CPMM_ERROR_RESULT;
```

Description

The MM_ERROR_RESULT *event information* data structure is used for analyzing the MMEV_ERROR unsolicited failure event.

If the MMEV_ERROR unsolicited failure event occurs, use the **mm_GetMetaEvent( )** function to retrieve the reason for the error. The **mm_GetMetaEvent( )** function outputs the MM_ERROR_RESULT event data associated with the metaevent in the MM_METAEVENT data structure.

The INIT_MM_ERROR_RESULT inline function is provided to initialize the structure.

For a list of the error codes and messages returned in the MM_INFO structure, see Section 12.2, "Dialogic® Multimedia API Function Error Codes", on page 231.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unErrorCode
: Specifies an error code. See Section 12.2, "Dialogic® Multimedia API Function Error Codes", on page 231 for the possible values.

unErrorMsg
: Specifies an error message associated with the code. See Section 12.2, "Dialogic® Multimedia API Function Error Codes", on page 231 for the values.

unData
: Specifies an internal block of firmware data for debugging information related to the error; application program can record it in an error log along with the other fields.

MM_EVENTS

```
typedef struct tagMM_EVENTS
{
    unsigned int    unVersion;
    unsigned int    unMask;
} MM_EVENTS, *PMM_EVENTS;
typedef const MM_EVENTS* CPMM_EVENTS;
```

Description

The MM_EVENTS structure specifies the details of an enable or disable events request. This structure is used as *function input* for the **mm_EnableEvents( )** and **mm_DisableEvents( )** functions.

The INIT_MM_EVENTS inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unMask
: Specifies event mask to enable or disable. The mask is formed by ORing the masks shown in Table 7, “Optional Notification Events”, on page 139.

  Note: See **mm_SetParm( )** and the MM_SET_PARM structure for information on the related EMM_REC_IFRAME_TIMEOUT parameter. See also the **mm_Record( )** function.

MM_GET_PARM

```
typedef struct tagMM_GET_PARM
{
    unsigned int    unVersion;
    eMM_PARM        eParm;
} MM_GET_PARM, *PMM_GET_PARM;
typedef const MM_GET_PARM* CPMM_GET_PARM;
```

Description

The MM_GET_PARM structure specifies the details of a get parameter request. This structure is used as function input for the **mm_GetParm( )** function.

The INIT_MM_GET_PARM inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

eParm
: Specifies the parameter to get. Defined values include the following channel-level parameter:
  - EMM_REC_IFRAME_TIMEOUT – Specifies the time to wait for an I-frame. Video recording, or multimedia (audio and video) recording, starts when an I-frame is detected or when the time-out is reached.
  - EMM_TONE_DETECTION – Indicates whether tone detection is enabled or disabled. This parameter is only supported in **mm_GetParm( )** if the active license contains video transcoding resources; otherwise it is invalid.

MM_GET_PARM_RESULT

```
typedef struct tagMM_GET_PARM_RESULT
{
    unsigned int    unVersion;
    eMM_PARM        eParm;
    unsigned int    unParmValue;
} MM_GET_PARM_RESULT, *PMM_GET_PARM_RESULT;
typedef const MM_GET_PARM_RESULT* CPMM_GET_PARM_RESULT;
```

Description

The MM_GET_PARM_RESULT *event information* data structure is used for analyzing the results of the **mm_GetParm( )** function **termination or completion**, as reported by the MMEV_GET_PARM or MMEV_GET_PARM_FAIL event.

The INIT_MM_GET_PARM_RESULT inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

eParm
: Specifies parameter requested. Defined values include the following channel-level parameter:
  - EMM_REC_IFRAME_TIMEOUT – Specifies the time to wait for an I-frame. Video recording, or multimedia (audio and video) recording, starts when an I-frame is detected or when the time-out is reached.
  - EMM_TONE_DETECTION – Indicates whether tone detection is enabled or disabled.

unParmValue
: Parameter value. See MM_SET_PARM, on page 210 for the values.

MM_GETDURATION_CMPLT_DETAILS

```
typedef struct tagMM_GETDURATION_CMPLT_DETAILS
{
unsigned int unVersion;
unsigned int unStreamType;
unsigned int unRetCode;
unsigned int unDuration;
} MM_GETDURATION_CMPLT_DETAILS, *PMM_GETDURATION_CMPLT_DETAILS;
typedef const MM_GETDURATION_CMPLT_DETAILS* CPMM_GETDURATION_CMPLT_DETAILS;
```

Description

The MM_GETDURATION_CMPLT_DETAILS structure is a second-level event information data structure under the MM_GETDURATION_CMPLT structure. (See Section 10.5, "Other Data Structure Levels (_DETAILS)", on page 149.)

The INIT_MM_GETDURATION_CMPLT_DETAILS inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

un StreamType
: Indicates which stream type (audio, video or both) the information in this structure applies to. Value can be ONE of the following:
  - EMM_GETDURATION_VIDEO_AUDIO – information applies to BOTH video and audio.
  - EMM_GETDURATION_VIDEO_ONLY – information applies to video ONLY.
  - EMM_GETDURATION_AUDIO_ONLY – information applies to audio ONLY.

unRetCode
: Contains the firmware return code. A non-zero value indicates an error has occurred.

unDuration
: Specifies duration in milliseconds.

MM_GETDURATION_CMPLT

```
typedef struct tagMM_GETDURATION_CMPLT
{
unsigned int unVersion;
unsigned int unCount;
MM_GETDURATION_CMPLT_DETAILS details[MAX_GETDURATION_CMPLT];
} MM_GETDURATION_CMPLT, *PMM_GETDURATION_CMPLT;
```

Description

The MM_GETDURATION_CMPLT event information data structure is used for analyzing the results of the **mm_GetDuration( )** function termination or completion as reported by an MMEV_GETDURATION or MMEV_GETDURATION_FAIL event.

The INIT_MM_GETDURATION_CMPLT inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unCount
: Indicates the number of MM_GETDURATION_CMPLT_DETAILS elements that follow.

details
: Provides the details of completion. See the MM_GETDURATION_CMPLT_DETAILS structure definition for details.

MM_GETDURATION_INFO

```
typedef struct tagMM_GETDURATION_INFO
 {
unsigned int unVersion;
unsigned int unStreamType;
} MM_GETDURATION_INFO, *PMM_GETDURATION_INFO;
typedef const MM_GETDURATION_INFO* CPMM_GETDURATION_INFO;
```

Description

The MM_GETDURATION_INFO data structure is a function input structure for the **mm_GetDuration( )** function.

The INIT_MM_GETDURATION_INFO inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion

Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unStreamType

Indicates which stream type (audio, video or both) the information is being requested for. Value can be ONE of the following:

- EMM_GETDURATION_VIDEO_AUDIO – Get duration of BOTH video and audio. In this case, two values will be returned - the duration of the video and the duration of the audio. If either stream is in the idle state (the play operation has not been initiated or has ended), an error will be returned for that stream type but a valid duration will be retuned for the non-idle stream.
- EMM_GETDURATION_VIDEO_ONLY – Get duration of video ONLY.
- EMM_GETDURATION_AUDIO_ONLY – Get duration of audio ONLY.

Note: The stream type for which duration is being requested must not be idle (the play operation must have been initiated and it must not have ended). This means that the **mm_GetDuration( )** function will only be valid during the period of time between the MMEV_PLAY_ACK event and the MMEV_PLAY event for the requested stream type. The **mm_GetDuration( )** function is valid for paused streams.

MM_GETELAPSEDTIME_CMPLT

```
typedef struct tagMM_GETELAPSEDTIME_CMPLT
{
unsigned int unVersion;
unsigned int unCount;
MM_GETELAPSEDTIME_CMPLT_DETAILS details[MAX_GETELAPSEDTIME_CMPLT];
} MM_GETELAPSEDTIME_CMPLT, *PMM_GETELAPSEDTIME_CMPLT;
```

Description

The MM_GETELAPSEDTIME_CMPLT event information data structure is used for analyzing the results of the **mm_GetElapsedTime( )** function termination or completion as reported by an MMEV_GETELAPSEDTIME or MMEV_GETELAPSEDTIME_FAIL event.

The INIT_MM_GETELAPSEDTIME_CMPLT inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unCount
: Indicates the number of MM_GETELAPSEDTIME_CMPLT_DETAILS elements that follow.

details
: Provides the details of completion. See the MM_GETELAPSEDTIME_CMPLT_DETAILS structure definition for details.

MM_GETELAPSEDTIME_CMPLT_DETAILS

```
typedef struct tagMM_GETELAPSEDTIME_CMPLT_DETAILS
{
unsigned int unVersion;
unsigned int unStreamType;
unsigned int unRetCode;
unsigned int unElapsedTime;
} MM_GETELAPSEDTIME_CMPLT_DETAILS, *PMM_GETELAPSEDTIME_CMPLT_DETAILS;
typedef const MM_GETELAPSEDTIME_CMPLT_DETAILS* CPMM_GETELAPSEDTIME_CMPLT_DETAILS;
```

Description

The MM_GETELAPSEDTIME_CMPLT_DETAILS structure is a second-level event information data structure under the MM_GETELAPSEDTIME_CMPLT structure. (See Section 10.5, "Other Data Structure Levels (_DETAILS)", on page 149.)

The INIT_MM_GETELAPSEDTIME_CMPLT_DETAILS inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

un StreamType
: Indicates which stream type (audio, video or both) the information in this structure applies to. Value can be ONE of the following:
  - EMM_GETELAPSEDTIME_VIDEO_AUDIO – information applies to BOTH video and audio.
  - EMM_GETELAPSEDTIME_VIDEO_ONLY – information applies to video ONLY.
  - EMM_GETELAPSEDTIME_AUDIO_ONLY – information applies to audio ONLY.

unRetCode
: Contains the firmware return code. A non-zero value indicates an error has occurred and what the firmware error code is.

unElapsedTime
: Indicates elapsed time in milli-seconds.

MM_GETELAPSEDTIME_INFO

```
typedef struct tagMM_GETELAPSEDTIME_INFO
{
    unsigned int unVersion;
    unsigned int unStreamType;
} MM_GETELAPSEDTIME_INFO, *PMM_GETELAPSEDTIME_INFO;
typedef const MM_GETELAPSEDTIME_INFO* CPMM_GETELAPSEDTIME_INFO;
```

Description

The MM_GETELAPSEDTIME_INFO data structure is a function input structure for the **mm_GetElapsedTime( )** function.

The INIT_MM_GETELAPSEDTIME_INFO inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

un StreamType
: Indicates the stream type (audio, video or both) for which the information is being requested. Value can be ONE of the following:

- EMM_GETELAPSEDTIME_VIDEO_AUDIO – Get elapsed time of video and audio. In this case two values will be returned - the elapsed time of the video and the elapsed time of the audio. If either stream is in the idle state (i.e. the play operation has not been initiated or has ended), an error will be returned for that stream type but a valid elapsed time value will be returned for the non-idle stream.
- EMM_GETELAPSEDTIME_VIDEO_ONLY – Get elapsed time of video ONLY.
- EMM_GETELAPSEDTIME_AUDIO_ONLY – Get elapsed time of audio ONLY.

Note: The stream type for which elapsed time is being requested must not be idle (the play operation must have been initiated and it must not have ended). This means that the **mm_GetElapsedTime( )** function will only be valid during the period of time between the MMEV_PLAY_ACK event and the MMEV_PLAY event for the requested stream type. The **mm_GetElapsedTime( )** is valid for paused streams.

MM_IMAGE_FORMAT

```
typedef union tagMM_IMAGE_FORMAT
{
      MM_YUV   yuv;

} MM_IMAGE_FORMAT, *PMM_IMAGE_FORMAT;
```

Description

The MM_IMAGE_FORMAT union is used to provide additional characteristics of the specified image format. This union is a nested function input structure for the **mm_Play( )** function or **mm_Capture( )** function. It is nested directly under the MM_MEDIA_IMAGE structure, and it is a sixth-level structure under the MM_PLAY_INFO structure (For a description of these *nesting levels*, see Section 10.4, "Play/Record Data Structure Levels", on page 149.)

The INIT_MM_IMAGE_FORMAT inline function is provided to initialize the structure.

Field Descriptions

The fields of the data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

yuv
: Specifies the characteristics of a YUV image. Refer to the MM_YUV data structure for details.

MM_INFO

```
typedef struct tagMM_INFO
{
    unsigned int  unVersion;
    int           mmValue;
    const char*   mmMsg;
    const char*   additionalInfo;
} MM_INFO, *PMM_INFO;
typedef const MM_INFO* CPMM_INFO;
```

Description

The MM_INFO structure is used as *function output* for the **mm_ErrorInfo( )** function and contains error or result information.

The INIT_MM_INFO inline function is provided to initialize the structure.

Field Descriptions

The fields of MM_INFO are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

mmValue
: Specifies a Multimedia API error or result value. See Section 12.2, “Dialogic® Multimedia API Function Error Codes”, on page 231.

mmMsg
: Specifies a pointer to a null-terminated string containing a message associated with the Multimedia API error or result value. See Section 12.2, “Dialogic® Multimedia API Function Error Codes”, on page 231.

additionalInfo
: Specifies a pointer to a null-terminated string containing additional information associated with this error or result value. This additional information is optional and may be used as a diagnostic aid. .

MM_MEDIA_ACCESS_MEMORY

```
typedef struct tagMM_MEDIA_ACCESS_MEMORY
{
      unsigned int  unVersion;
      unsigned char *pBuffer;
      unsigned int  unBufferSize;
} MM_MEDIA_ACCESS_MEMORY, *PMM_MEDIA_ACCESS_MEMORY;

typedef const MM_MEDIA_ACCESS_MEMORY *CPMM_MEDIA_ACCESS_MEMORY;
```

Description

This structure is used by the **mm_Play( )** and **mm_Record( )** functions to access the media buffer.

The INIT_MM_MEDIA_ACCESS_MEMORY inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

pBuffer
: Specifies the pointer to the memory location in memory I/O mode. The application is responsible for allocating the memory to which the pBuffer points.

unBufferSize
: Specifies the size in bytes of the memory buffer pointed to by pBuffer. Valid value is >0.

MM_MEDIA_ACCESS_STREAM

```
typedef struct tagMM_MEDIA_ACCESS_STREAM
{
      unsigned int unVersion;
      int nStreamHandle;
} MM_MEDIA_ACCESS_STREAM, *PMM_MEDIA_ACCESS_STREAM;
typedef const MM_MEDIA_ACCESS_STREAM *CPMM_MEDIA_ACCESS_STREAM;
```

Description

This structure is used by the **mm_Play( )** and **mm_Record( )** functions to access the media stream.

The INIT_MM_MEDIA_ACCESS_STREAM inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

nStreamHandle
: Specifies the stream handle in stream I/O mode. The stream handle is the handle returned by **mm_StreamOpen( )**.

MM_MEDIA_AUDIO

```
typedef struct tagMM_MEDIA_AUDIO
{
     unsigned int    unVersion;
     MM_AUDIO_CODEC  codec;
     unsigned int    unMode;
     unsigned int    unOffset;
     union {
          const char*         szFileName;
          MM_MEDIA_ACCESS_STREAM stream;
          MM_MEDIA_ACCESS_MEMORY memory;
     };
     unsigned int    unAccessMode;
     eMM_FILE_FORMAT eFileFormat;
} MM_MEDIA_AUDIO, *PMM_MEDIA_AUDIO;
typedef const MM_MEDIA_AUDIO* CPMM_MEDIA_AUDIO;
```

Description

The MM_MEDIA_AUDIO structure specifies the audio media item. This structure is a nested *function input* structure for the **mm_Play( )** or **mm_Record( )** function. It is nested directly under the MM_MEDIA_ITEM structure, and it is a fifth-level structure under the MM_PLAY_RECORD_INFO (MM_PLAY_INFO, MM_RECORD_INFO) structure. (For a description of these *nesting levels*, see Section 10.4, “Play/Record Data Structure Levels”, on page 149.)

The INIT_MM_MEDIA_AUDIO inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

codec
: Specifies the audio codec. See definition of MM_AUDIO_CODEC structure.

unMode
: Specifies the mask for the mode of operation. The mask is formed from one or more of the following:
  - MM_MODE_AUD_PAUSED – Bitnask to specify pause.
  - MM_MODE_AUD_BEEPINITIATED – Bitmask to **enable** notification tone (or “beep”) at start of **audio play or record**.

    Note: If doing a multimedia playback or record with synchronized audio and video, only a single beep is generated even if the corresponding MM_MEDIA_VIDEO beep is enabled.
  - MM_MODE_AUD_OFFSET_IN_BYTES – If set, specifies unit of the unOffset field is in bytes instead of time.
  - MM_MODE_AUD_AGC_ON – Specifies AGC mode.

unOffset
: Specifies data offset in milliseconds. To specify a byte offset, set the MM_MODE_AUD_OFFSET_IN_BYTES bit in the unMode field. Non-zero offsets are only supported when playing WAVE files. For all other cases, this field must be set to 0.

szFileName
: In file I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_FILE), specifies the file name of the audio media item.

stream
: In stream I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_STREAM), specifies additional information about the audio stream. Refer to the MM_MEDIA_ACCESS_STREAM structure for more information.

memory
: In memory I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_MEMORY), specifies additional information about the memory buffer. Refer to the MM_MEDIA_ACCESS_MEMORY structure for more information.

unAccessMode
: Indicates the I/O mode to be used in accessing the audio item. It also identifies the field in the union to be used. Defines include:
  - MM_MEDIA_ACCESS_MODE_FILE
  - MM_MEDIA_ACCESS_MODE_MEMORY
  - MM_MEDIA_ACCESS_MODE_STREAM

eFileFormat
: Specifies format of the audio file. Valid values include:
  - EMM_AUD_FILEFORMAT_3GP – 3GP format
  - EMM_AUD_FILEFORMAT_PROPRIETARY – HMP proprietary format.
  - EMM_AUD_FILEFORMAT_VOX – linear PCM (VOX) format.
  - EMM_AUD_FILEFORMAT_WAVE – WAVE format.

MM_MEDIA_IMAGE

```
typedef struct tagMM_MEDIA_IMAGE
{
    unsigned int      unVersion;
    eMTK_IMAGE_FORMAT eFormat;
    MM_IMAGE_FORMAT   format;
    unsigned int      unAccessMode;
    union
    {
         const char*            szFileName;
         MM_MEDIA_ACCESS_STREAM stream;
         MM_MEDIA_ACCESS_MEMORY memory;
     };
}MM_MEDIA_IMAGE, *PMM_MEDIA_IMAGE;
typedef const MM_MEDIA_IMAGE* CPMM_MEDIA_IMAGE;
```

Description

The MM_MEDIA_IMAGE structure specifies the image media item to be played or captured. This structure is a nested function input structure for the **mm_Play( )** or **mm_Capture( )** function. It is nested directly under the MM_MEDIA_ITEM structure, and it is a fifth-level structure under the MM_PLAY_INFO structure. (For a description of these *nesting levels*, see Section 10.4, "Play/Record Data Structure Levels", on page 149.)

The INIT_MM_MEDIA_IMAGE inline function is provided to initialize the structure.

Field Descriptions

The fields of the data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

eFormat
: Specifies the image format. Valid values include:
eMTK_IMAGE_FORMAT_YUV
eMTK_IMAGE_FORMAT_JPEG

  Note: Both YUV 4:2:0 and JPEG formats are supported for still image play in this release. Only YUV 4:2:0 format is supported for image capture in this release.

format
: Specifies the characteristics of the image format specified by the eFormat field. Since JPEG headers already contain format specific information, this field is ignored when eFormat is set to eMTK_IMAGE_FORMAT_JPEG. Refer to the definition of MM_IMAGE_FORMAT structure for more information.

unAccessMode

Indicates the I/O mode to be used in accessing the image item. It also identifies the field in the union to be used.

Defines include:

MM_MEDIA_ACCESS_MODE_FILE
MM_MEDIA_ACCESS_MODE_MEMORY
MM_MEDIA_ACCESS_MODE_STREAM

Note: MM_MEDIA_ACCESS_MODE_MEMORY (memory I/O) and MM_MEDIA_ACCESS_MODE_STREAM (stream I/O) are not currently supported for image play or capture.

szFileName

In file I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_FILE), specifies the name of the file in which the image data resides or, in the case of an image capture, the name of the file in which image data will be stored.

stream

Not currently supported.

memory

Not currently supported.

MM_MEDIA_ITEM

```
typedef union tagMM_MEDIA_ITEM
{
  MM_MEDIA_VIDEO    video;
  MM_MEDIA_AUDIO    audio;
  MM_MEDIA_TERM     term;
  MM_MEDIA_IMAGE    image;
} MM_MEDIA_ITEM, *PMM_MEDIA_ITEM;
typedef const MM_MEDIA_ITEM* CPMM_MEDIA_ITEM;
```

Description

The MM_MEDIA_ITEM union specifies the media item. This union is a nested *function input* structure for the **mm_Play( )** or **mm_Record( )** function. It is nested directly under the MM_MEDIA_ITEM_LIST structure, and it is a fourth-level structure under the MM_PLAY_RECORD_INFO (MM_PLAY_INFO, MM_RECORD_INFO) structure. (For a description of these *nesting levels*, see Section 10.4, "Play/Record Data Structure Levels", on page 149.)

Since this structure is a union, only one of the INIT inline functions should be used to initialize the fields in this structure. For example, if the item is a video item (i.e., ItemType in MM_PLAY_RECORD_LIST is MM_MEDIA_TYPE_VIDEO), then only the INIT_MM_MEDIA_VIDEO inline function should be called, not the INIT_MM_MEDIA_AUDIO, INIT_MM_MEDIA_TERM or INIT_MM_MEDIA_IMAGE inline functions.

Field Descriptions

The ItemType field in the MM_PLAY_RECORD_LIST structure identifies which field in the MM_MEDIA_ITEM union is to be used. The elements of the union are described as follows:

video
: Specifies information about a video item. See the definition of the MM_MEDIA_VIDEO structure.

audio
: Specifies information about an audio item. See the definition of the MM_MEDIA_AUDIO structure.

term
: Reserved for future use.

image
: Specifies information about an image item. See the definition of the MM_MEDIA_IMAGE structure.

MM_MEDIA_ITEM_LIST

```
typedef struct tagMM_MEDIA_ITEM_LIST
{
    unsigned int                    unVersion;
    eMM_ITEM                        ItemChain;
    MM_MEDIA_ITEM                   item;
    struct tagMM_MEDIA_ITEM_LIST*   next;
    struct tagMM_MEDIA_ITEM_LIST*   prev;   /* optional */
} MM_MEDIA_ITEM_LIST, *PMM_MEDIA_ITEM_LIST;
typedef const MM_MEDIA_ITEM_LIST* CPMM_MEDIA_ITEM_LIST;
```

Description

The MM_MEDIA_ITEM_LIST structure specifies the media item list. This structure is a nested *function input* structure for the **mm_Play( )** or **mm_Record( )** function. It is nested directly under the MM_PLAY_RECORD_LIST structure, and it is a third-level structure under the MM_PLAY_RECORD_INFO (MM_PLAY_INFO, MM_RECORD_INFO) structure. (For a description of these *nesting levels*, see Section 10.4, "Play/Record Data Structure Levels", on page 149.)

The INIT_MM_MEDIA_ITEM_ LIST inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

ItemChain
: Specifies the next list element for iteration. Defined values include:
  - EMM_ITEM_CONT – Indicates that the next list element immediately follows the current.
  - EMM_ITEM_LINK – Indicates that the next and previous list elements linked to the current via pointers.
  - EMM_ITEM_EOT – Indicates that the current list element is the last in the list.

item
: Specifies additional information about the media item. See the definition of the MM_MEDIA_ITEM structure.

next
: Points to the next MM_MEDIA_ITEM_LIST item. This field is required only when ItemChain is set to EMM_ITEM_LINK. Set to NULL for the last item.

prev
: Points to the previous MM_MEDIA_ITEM_LIST item (optional). This field is applicable only when ItemChain is set to EMM_ITEM_LINK. Set to NULL for the first item.

MM_MEDIA_TERM

```
typedef struct tagMM_MEDIA_TERM
{
    unsigned int   unVersion;
    unsigned int   unRfu;
} MM_MEDIA_TERM, *PMM_MEDIA_TERM;
typedef const MM_MEDIA_TERM* CPMM_MEDIA_TERM;
```

Description

The MM_MEDIA_TERM structure specifies termination details. It is a nested *function input* structure for the **mm_Play( )** or **mm_Record( )** function. It is nested directly under the MM_MEDIA_ITEM union, and it is a fifth-level structure under the MM_PLAY_RECORD_INFO (MM_PLAY_INFO, MM_RECORD_INFO) structure. (For a description of these *nesting levels*, see Section 10.4, "Play/Record Data Structure Levels", on page 149.)

This structure is reserved for future use.

MM_MEDIA_VIDEO

```
typedef struct tagMM_MEDIA_VIDEO
{
     unsigned int    unVersion;
     MM_VIDEO_CODEC  codec;
     unsigned int    unMode;
     union {
          const char*            szFileName;
          MM_MEDIA_ACCESS_STREAM stream;
          MM_MEDIA_ACCESS_MEMORY memory;
    };
     unsigned int    unAccessMode;
     unsigned int    unOffset;
    eMM_FILE_FORMAT eFileFormat;
} MM_MEDIA_VIDEO, *PMM_MEDIA_VIDEO;
```

Description

The MM_MEDIA_VIDEO structure specifies the video media item. This structure is a nested *function input* structure for the **mm_Play( )** or **mm_Record( )** function. It is nested directly under the MM_MEDIA_ITEM structure, and it is a fifth-level structure under the MM_PLAY_RECORD_INFO (MM_PLAY_INFO, MM_RECORD_INFO) structure. (For a description of these *nesting levels*, see Section 10.4, "Play/Record Data Structure Levels", on page 149.)

The INIT_MM_MEDIA_VIDEO inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

codec
: Specifies the video codec. See definition of MM_VIDEO_CODEC structure.

unMode
: Specifies the mask for the mode of operation. The mask is formed from one or more of the following:
  - MM_MODE_VID_PAUSED – Bitnask to specify pause.
  - MM_MODE_VID_BEEPINITIATED – Bitmask to **enable** notification tone (or "beep") at start of **video play**.
  - MM_MODE_VID_NOIFRMBEEPINITIATED – Bitmask to **disable** notification tone at start of **video record**. (The record beep is enabled by default and is transmitted upon detection of an I-frame or upon time-out waiting for an I-frame.)

  Note: If doing a multimedia playback or record with synchronized audio and video, only a single beep is generated even if the corresponding MM_MEDIA_AUDIO beep is enabled.

szFileName
: In file I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_FILE), specifies the file name of the video media item.

stream
: In stream I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_STREAM), specifies additional information about the video stream. Refer to the MM_MEDIA_ACCESS_STREAM structure for more information.

memory
: In memory I/O mode (when unAccessMode is set to MM_MEDIA_ACCESS_MODE_MEMORY), specifies additional information about the memory buffer. Refer to the MM_MEDIA_ACCESS_MEMORY structure for more information.

unAccessMode
: Indicates I/O mode that is to be used in accessing the video item. It also identifies the field in the union to be used. Defines include:
  - MM_MEDIA_ACCESS_MODE_FILE
  - MM_MEDIA_ACCESS_MODE_MEMORY
  - MM_MEDIA_ACCESS_MODE_STREAM

unOffset
: Reserved for future use. Set to 0.

eFileFormat
: Specifies format of the video file. Valid values:
  - EMM_FILE_FORMAT_3GP – 3GP format
  - EMM_FILE_FORMAT_PROPRIETARY – HMP proprietary format

MM_METAEVENT

```
typedef struct tag MM_METAEVENT
{
    unsigned int     unVersion;
    unsigned long    flags;
    void*            evtdatap;
    long             evtlen;
    long             evtdev;
    long             evttype;
    void*            evtUserInfo;
    int              rfu1;
} MM_METAEVENT, *PMM_METAEVENT;
typedef const MM_METAEVENT* CPMM_METAEVENT;
```

Description

The MM_METAEVENT structure is used as *function output* for the **mm_GetMetaEvent( )** function.

The MM_METAEVENT structure contains the event descriptor for a metaevent. The Field Descriptions section below describes each element used in the metaevent data structure, and where applicable, indicates the Dialogic® Standard Runtime Library (SRL) API function that is used to retrieve the information stored in the associated field. This data structure eliminates the need for the application to issue the equivalent Dialogic® SRL functions listed below. For information on the Dialogic® SRL API, see the *Dialogic® Standard Runtime Library API Library Reference*.

Field Descriptions

The fields of MM_METAEVENT are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

flags
: Specifies any flags for the information in the structure. If the metaevent is a Multimedia API event, the MMME_MM_EVENT bit in the flags field will be set. The MM_METAEVENT fields contain valid Multimedia API-related data only when the MMME_MM_EVENT bit is set. Do **not** use these fields for obtaining multimedia information if the bit is not set.

evtdatap
: Specifies a pointer to the data associated with the event. Function equivalent: **sr_getevtdatap( )**

evtlen
: Specifies the length of the data associated with the event. Function equivalent: **sr_getevtlen( )**

evtdev
: Specifies the event device handle. Function equivalent: **sr_getevtdev( )**

evttype
: Specifies the event type. Function equivalent: **sr_getevttype( )**

evtUserInfo

Points to the user information associated with the event. This is the data passed by a **pUserInfo** parameter in an asynchronous Multimedia API function call. If no data was passed, it points to NULL. Function equivalent: **sr_getUserContext( )**

rfu1

Reserved for future use. Must be set to 0.

MM_PAUSE_INFO

```
typedef struct tagMM_PAUSE_INFO
 {
    unsigned int unVersion;
    unsigned int unStreamType;
    unsigned int unAttribute;
} MM_PAUSE_INFO, *PMM_PAUSE_INFO;
typedef const MM_PAUSE_INFO* CPMM_PAUSE_INFO;
```

Description

The MM_PAUSE_INFO data structure is a function input structure for the **mm_Pause( )** function.

The INIT_MM_PAUSE_INFO inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

un StreamType
: Indicates which stream type (audio, video or both) is to be paused. Value can be ONE of the following:
  - EMM_PAUSE_VIDEO_AUDIO – Pause both video and audio. If either stream is in the idle state (the play operation has not been initiated or has ended), an error will be returned for that stream type but the non-idle stream will be paused.
  - EMM_PAUSE_VIDEO_ONLY – Pause video ONLY.
  - EMM_PAUSE_AUDIO_ONLY – Pause audio ONLY.

  Note: The stream type which is to be paused must not be idle (the play operation must have been initiated and it must not have ended). This means that the **mm_Pause( )** function will only be valid during the period of time between the MMEV_PLAY_ACK event and the MMEV_PLAY event for the requested stream type.

  Note: In an audiovideo playback, if the playback is synchronous (both audio and video play are initiated by a single **mm_Play( )** function call), **mm_Pause( )** will cause BOTH the audio and video streams to be paused, regardless of the unStreamType setting in MM_PAUSE_INFO. This allows synchronization to be maintained.

unAttribute
: This field is used for various attributes of the **mm_Pause( )** function. Defined values include:
  - EMM_PAUSE_AUDIO_SILENCE – When audio transmission is paused transmit audio silence (DEFAULT).
  - EMM_PAUSE_AUDIO_CNG – When audio transmission is paused transmit audio comfort noise. This value is not currently supported.

MM_PAUSE_PLAY_CMPLT

```
typedef struct tagMM_PAUSE_PLAY_CMPLT
{
unsigned int unVersion;
unsigned int unCount;
MM_PAUSE_PLAY_CMPLT_DETAILS details[MAX_PAUSE_PLAY_CMPLT];
} MM_PAUSE_PLAY_CMPLT, *PMM_PAUSE_PLAY_CMPLT;
```

Description

The MM_PAUSE_PLAY_CMPLT event information data structure is used for analyzing the results of the **mm_Pause( )** function termination or completion as reported by an MMEV_PAUSE or MMEV_PAUSE_FAIL event.

The INIT_MM_PAUSE_PLAY_CMPLT inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unCount
: Indicates the number of MM_PAUSE_PLAY_CMPLT_DETAILS elements that follow.

details
: Provides the details of completion. See the MM_PAUSE_PLAY_CMPLT_DETAILS structure definition for details.

MM_PAUSE_PLAY_CMPLT_DETAILS

```
typedef struct tagMM_PAUSE_PLAY_CMPLT_DETAILS
{
unsigned int unVersion;
unsigned int unStreamType;
unsigned int unRetCode;
} MM_PAUSE_PLAY_CMPLT_DETAILS, *PMM_PAUSE_PLAY_CMPLT_DETAILS;
typedef const MM_PAUSE_PLAY_CMPLT_DETAILS* CPMM_PAUSE_PLAY_CMPLT_DETAILS;
```

Description

The MM_PAUSE_PLAY_CMPLT_DETAILS structure is a second-level event information data structure under the MM_PAUSE_PLAY_CMPLT structure. (See Section 10.5, "Other Data Structure Levels (_DETAILS)", on page 149.)

The INIT_MM_PAUSE_PLAY_CMPLT_DETAILS inline function is provided to initialize the structure.

Field Descriptions

The fields of the data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

un StreamType
: Indicates which stream type (audio, video or both) the information in this structure applies to. Value can be ONE of the following:
  - EMM_PAUSE_PLAY_VIDEO_AUDIO – information applies to BOTH video and audio.
  - EMM_PAUSE_PLAY_VIDEO_ONLY – information applies to video ONLY.
  - EMM_PAUSE_PLAY_AUDIO_ONLY – information applies to audio ONLY.

unRetCode
: Contains the firmware return code. A non-zero value indicates an error has occurred.

MM_PLAY_RECORD_CMPLT

```
typedef struct tagMM_PLAY_RECORD_CMPLT
{
    unsigned int                    unVersion;
    unsigned int                    unCount;
    MM_PLAY_RECORD_CMPLT_DETAILS    details[MAX_PLAY_RECORD_CMPLT];
} MM_PLAY_RECORD_CMPLT, *PMM_PLAY_RECORD_CMPLT;
typedef const MM_PLAY_RECORD_CMPLT* CPMM_PLAY_RECORD_CMPLT;

typedef MM_PLAY_RECORD_CMPLT MM_PLAY_CMPLT, *PMM_PLAY_CMPLT;
typedef CPMM_PLAY_RECORD_CMPLT CPMM_PLAY_CMPLT;

typedef MM_PLAY_RECORD_CMPLT MM_RECORD_CMPLT, *PMM_RECORD_CMPLT;
typedef CPMM_PLAY_RECORD_CMPLT CPMM_RECORD_CMPLT;
```

Description

The MM_PLAY_RECORD_CMPLT *event information* data structure is used for analyzing the results of the **mm_Play( )** or **mm_Record( )** function **termination or completion** as reported by an MMEV_PLAY, MMEV_PLAY_FAIL, MMEV_RECORD, or MMEV_RECORD_FAIL event.

Note: This structure is used as a typedef for the MM_PLAY_CMPLT and MM_RECORD_CMPLT *event information* structures, which are defined as a convenience.

Extra information may be contained at the end of this structure when the application records to stream or memory. When extra information is present, the application is required to write it to the data that it recorded. This is also required if the record fails and the extra information is present at the end of the structure.

Refer to the MM_PLAY_RECORD_CMPLT_DETAILS data structure for information about the EMM_CMPLT_AUDIO_RECORD_WITH_UPDATE and EMM_CMPLT_VIDEO_RECORD_WITH_UPDATE “Complete” values.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unCount
: Specifies the number of MM_PLAY_RECORD_CMPLT_DETAILS elements that follow.

details
: Specifies the details of completion. See MM_PLAY_RECORD_CMPLT_DETAILS structure definition for details.

Note: See Section 12.5, “Terminating and Non-Terminating Play/Record Errors”, on page 235 for related information.

MM_PLAY_RECORD_CMPLT_DETAILS

```
typedef struct tagMM_PLAY_RECORD_CMPLT_DETAILS
{
    unsigned int                    unVersion;
    eMM_CMPLT_PLAY_RECORD           Complete;
    eMM_CMPLT_PLAY_RECORD_REASON    Reason;
    unsigned int                    unDuration;
    unsigned int                    unNumberOfBytes;
    eMM_CMPLT_PLAY_RECORD_STATUS    Status;
} MM_PLAY_RECORD_CMPLT_DETAILS, *PMM_PLAY_RECORD_CMPLT_DETAILS;
typedef const MM_PLAY_RECORD_CMPLT_DETAILS* CPMM_PLAY_RECORD_CMPLT_DETAILS;
```

Description

The MM_PLAY_RECORD_CMPLT_DETAILS structure specifies the elements of details of MMEV_PLAY, MMEV_PLAY_FAIL, MMEV_RECORD, or MMEV_RECORD_FAIL events. This structure is a second-level *event information* data structure under the MM_PLAY_RECORD_CMPLT structure. (See Section 10.5, "Other Data Structure Levels (_DETAILS)", on page 149.)

The INIT_MM__PLAY_RECORD_CMPLT_DETAILS inline function is provided to initialize the structure.

Note: If the "Complete" field value is EMM_CMPLT_AUDIO_RECORD_WITH_UPDATE or EMM_CMPLT_VIDEO_RECORD_WITH_UPDATE, then the MM_RECORD_CMPLT_INFO_DATA_BLOCKS structure follows the last MM_PLAY_RECORD_CMPLT_DETAILS structure. The number of MM_PLAY_RECORD_CMPLT_DETAILS structures is contained in the unCount field of the MM_PLAY_RECORD_CMPLT structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

Complete
: Specifies the type of terminated or completed operation. Defined values include:
  - EMM_CMPLT_AUDIO_PLAY – Audio playback operation.
  - EMM_CMPLT_AUDIO_RECORD – Audio recording operation.
  - EMM_CMPLT_IMAGE_PLAY – Image playback operation.
  - EMM_CMPLT_VIDEO_PLAY – Video playback operation.
  - EMM_CMPLT_VIDEO_RECORD – Video recording operation.
  - EMM_CMPLT_AUDIO_RECORD_WITH_UPDATE – Audio recording operation plus update information.

- EMM_CMPLT_VIDEO_RECORD_WITH_UPDATE – Video recording operation plus update information.

Note: Refer to the Macros section of this data structure when the "Complete" value is either EMM_CMPLT_AUDIO_RECORD_WITH_UPDATE or EMM_CMPLT_VIDEO_RECORD_WITH_UPDATE.

Reason

Specifies the reason for the termination. Defined values include:

- EMM_TR_DIGMASK – Termination occurred because the EMM_TERM_DIGMASK runtime control (RTC) condition specified in **mm_Play( )** or **mm_Record( )** was met.
- EMM_TR_DIGTYPE – Termination occurred because the EMM_TERM_DIGTYPE RTC condition specified in **mm_Play( )** or **mm_Record( )** was met.
- EMM_TR_EOF – **mm_Play( )** terminated because end of file was reached.
- EMM_TR_ERROR – **mm_Play( )** or **mm_Record( )** terminated because an error occurred. See Status field for details. See also Section 12.5, "Terminating and Non-Terminating Play/Record Errors", on page 235.
- EMM_TR_MAX_DTMF_DIGIT – Termination occurred because the EMM_TERM_MAXDTMF RTC condition specified in **mm_Play( )** or **mm_Record( )** was met.
- EMM_TR_MAXTIME – Termination occurred because the EMM_TERM_MAXTIME RTC condition specified in **mm_Play( )** or **mm_Record( )** was met.
- EMM_TR_TIME__FROM_AUDIO_RECORD_STARTED – Termination occurred because the EMM_TERM_TIME_FROM_AUDIO_RECORD_STARTED RTC condition specified in **mm_Record( )** was met.
- EMM_TR_USERSTOP – **mm_Play( )** or **mm_Record( )** terminated by the **mm_Stop( )** function.

unDuration

Specifies the duration of play or record in milliseconds.

unNumberOfBytes

Specifies the number of bytes played or recorded. For image play, the unNumberOf Bytes will be the total byte count of the image data. For JPEG, this number does not include the header data size.

Status

Specifies any error during play or record. (See also Section 12.5, "Terminating and Non-Terminating Play/Record Errors", on page 235.) Defined values include:

- EMM_STATUS_PLAY_A_FILEREAD_ERROR – Audio file read error.
- EMM_STATUS_PLAY_I_FILEREAD_ERROR – Image file read error.
- EMM_STATUS_PLAY_I_NULLDATA_ERROR – Image bitmap data pointer is null.
- EMM_STATUS_PLAY_V_ERROR_FS_GT_MFS – Video frame size greater than maximum frame size of internal buffer.
- EMM_STATUS_PLAY_V_FILEREAD_ERROR – Video file read error.
- EMM_STATUS_PLAY_BAD_PACKET – File/stream being played contains invalid values.
- EMM_STATUS_PLAY_CODING_ERROR – Data format specified in the "unCoding" field of the MM_AUDIO_CODEC structure is either not supported or does not match the format of the data contained in the played file.
- EMM_STATUS_RCRD_CODING_ERROR – Encoding to the data format specified in the "unCoding" field of the MM_AUDIO_CODEC structure is not supported.

- EMM_STATUS_RCRD_A_DRPD_FRAME_FULL_ERROR – Full audio frame dropped error.
- EMM_STATUS_RCRD_V_DRPD_FRAME_FULL_ERROR – Full video frame dropped error.
- EMM_STATUS_RCRD_V_MISSING_MPEG4_VISUALCONFIG_ERROR – Visual configuration data (either MPEG-4 or H.264) is not present in the incoming video stream and it has not been specified by the application in **mm_Record( )**.
- EMM_STATUS_RCRD_V_PKTS_DROPD_FS_GT_MFS – Video packets dropped frame size greater than maximum frame size of internal buffer.
- EMM_STATUS_SUCCESS – Successful completion or successful termination (no error).

Macros

The following macros simplify the retrieval of record update information, and need only to be called when the “Complete” value is either EMM_CMPLT_AUDIO_RECORD_WITH_UPDATE or EMM_CMPLT_VIDEO_RECORD_WITH_UPDATE.

Note: The following macros only apply to memory and stream records.

MM_RECORD_CMPLT_INIT(recCmplt, tempPtr)

- recCmplt – the event data pointer returned by the **sr_getevtdatap( )** function.
- tempPtr – caller supplied variable of type “unsigned char *” that is used internally by these macros. The caller must not read or modify this variable.

  This macro sets “tempPtr” to point to the first MM_RECORD_CMPLT_INFO_DATA_BLOCKS structure. This macro must be called immediately after the event data is retrieved, using the **sr_getevtdatap( )** function, for stream and memory records.

MM_RECORD_CMPLT_INFO_DATA_BLOCKS_GET(dbInfo, tempPtr)

- dbInfo – caller supplied variable of type PMM_RECORD_CMPLT_INFO_DATA_BLOCKS.
- tempPtr – same variable passed to the MM_RECORD_CMPLT_INIT( ) macro.

  This macro sets "dbInfo" to point to the next MM_RECORD_CMPLT_INFO_DATA_BLOCKS structure. “tempPtr” is set to point to the first MM_RECORD_CMPLT_DATA_BLOCK following this MM_RECORD_CMPLT_INFO_DATA_BLOCKS structure.

MM_RECORD_CMPLT_DATA_BLOCK_GET(dataBlock, tempPtr)

- dataBlock – caller supplied variable of type PMM_RECORD_CMPLT_DATA_BLOCK.
- tempPtr – same variable passed to the MM_RECORD_CMPLT_INIT( ) macro.

  This macro sets "dataBlock" to point to the next MM_RECORD_CMPLT_DATA_BLOCK structure. “tempPtr” is set to the next MM_RECORD_CMPLT_DATA_BLOCK.

MM_PLAY_RECORD_INFO

```
typedef struct tagMM_PLAY_RECORD_INFO
{
    unsigned int             unVersion;
    eMM_FILE_FORMAT          eFileFormat;
    CPMM_PLAY_RECORD_LIST    list;
} MM_PLAY_RECORD_INFO, *PMM_PLAY_RECORD_INFO;
typedef const MM_PLAY_RECORD_INFO* CPMM_PLAY_RECORD_INFO;

typedef MM_PLAY_RECORD_INFO MM_PLAY_INFO, *PMM_PLAY_INFO;
typedef CPMM_PLAY_RECORD_INFO CPMM_PLAY_INFO;
```

Description

This structure is a *function input* structure for the **mm_Play( )** or **mm_Record( )** function. It is a first-level structure that contains a nested hierarchy of structures under it. For a description of these *nesting levels*, see Section 10.4, "Play/Record Data Structure Levels", on page 149.)

The INIT_MM_PLAY_RECORD_INFO inline function is provided to initialize the structure.

Note: This structure is used as a typedef for the MM_PLAY_INFO and MM_RECORD_INFO *function input* structures, which are defined as a convenience for use by the **mm_Play( )** and **mm_Record( )** functions.

Field Descriptions

The fields of this data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

eFileFormat
: This field is ignored. Use the eFileFormat field within the MM_MEDIA_AUDIO and MM_MEDIA_VIDEO structures to specify file format.

list
: Points to the list of items to process. See definition of MM_PLAY_RECORD_LIST structure.

MM_PLAY_RECORD_LIST

```
typedef struct tagMM_PLAY_RECORD_LIST
{
    unsigned int                    unVersion;
    eMM_ITEM                        ItemChain;
    eMM_MEDIA_TYPE                  ItemType;
    CPMM_MEDIA_ITEM_LIST            list;
    unsigned int                    unRFU;
    struct tagMM_PLAY_RECORD_LIST*  next;
    struct tagMM_PLAY_RECORD_LIST*  prev;   /* optional */
} MM_PLAY_RECORD_LIST, *PMM_PLAY_RECORD_LIST;
typedef const MM_PLAY_RECORD_LIST* CPMM_PLAY_RECORD_LIST;
```

Description

The MM_PLAY_RECORD_LIST structure specifies the list of items to play or record. This structure is a nested *function input* structure for the **mm_Play( )** or **mm_Record( )** function. It is a second-level structure nested directly under the MM_PLAY_RECORD_INFO (MM_PLAY_INFO, MM_RECORD_INFO) structure. (For a description of *nesting levels*, see Section 10.4, "Play/Record Data Structure Levels", on page 149.)

The INIT_MM_PLAY_RECORD_LIST inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

ItemChain
: Specifies the next list element for iteration. Defined values include:
  - EMM_ITEM_CONT – Indicates that the next list element immediately follows the current.
  - EMM_ITEM_LINK – Indicates that the next and previous list elements linked to the current via pointers.
  - EMM_ITEM_EOT – Indicates that the current list element is the last in the list.

ItemType
: Specifies the type of items in the list field. Defined values include:
  - EMM_MEDIA_TYPE_VIDEO – Indicates that the list field points to a list of video items
  - EMM_MEDIA_TYPE_AUDIO – Indicates that the list field points to a list of audio items
  - EMM_MEDIA_TYPE_IMAGE – Indicates that the list field points to a list of image items.

list
: Points to the list of media items. See definition of MM_MEDIA_ITEM_LIST structure.

unRFU
: **Reserved for future use**. Must be set to 0.

next
: Points to the next MM_PLAY_RECORD_LIST item. Needed only when ItemChain specifies a linked list element. NULL value indicates last item.

prev
: Points to the previous MM_PLAY_RECORD_LIST item (optional). Needed only when ItemChain specifies a linked list element. NULL value indicates first item.

MM_RECORD_CMPLT_DATA_BLOCK

```
typedef struct tagMM_RECORD_CMPLT_DATA_BLOCK
{

   unsigned int unOffset;
   unsigned int unSize;
   unsigned char data[1];
} MM_RECORD_CMPLT_DATA_BLOCK, *PMM_RECORD_CMPLT_DATA_BLOCK;
```

Description

This structure only applies to memory and stream records. The unNumDataBlocks field in the RECORD_CMPLT_INFO_DATA_BLOCKS structure indicates how many of these structures are present. Call the MM_RECORD_CMPLT_DATA_BLOCK_GET() macro unNumDataBlocks times to retrieve a pointer to the each structure.

Field Descriptions

The fields of this structure are described as follows:

unOffset
: Specifies a zero-based offset (in bytes) in the application recorded data where the following data block is written. Byte 0 is the first byte recorded by the application.

unSize
: Specifies the size of the following data block in bytes (as in the amount of data that must be written to the application recorded data).

data
: Points to the first byte of the data block to be written to the application recorded data. The remaining bytes in the data block follow this structure.

Example

Note: This procedure ensures compatibility with future releases.

```
void handle_event (unsigned int chan)
{
   MM_PLAY_RECORD_CMPLT  *edata;
   MM_RECORD_CMPLT_INFO_DATA_BLOCKS  *dbInfo;
   MM_RECORD_CMPLT_DATA_BLOCK  *dataBlock;
   unsigned char  *tempptr = NULL;
   int  detailsIndex, blockIndex;

   edata = (MM_PLAY_RECORD_CMPLT *)sr_getevtdatap();  // get event data

   MM_RECORD_CMPLT_INIT(edata, tempptr);

   for (detailsIndex=0; detailsIndex<edata->unCount; detailsIndex++)
   {
      switch (edata->details[detailsIndex]).Complete)
      {
         case EMM_CMPLT_VIDEO_RECORD_WITH_UPDATE:
         case EMM_CMPLT_AUDIO_RECORD_WITH_UPDATE:

            // get a pointer to the record update information
```

```
            MM_RECORD_CMPLT_INFO_DATA_BLOCKS_GET(dbInfo, tempptr);

            for(blockIndex = 0; blockIndex < dbInfo->unNumDataBlocks; blockIndex++)
            {
               MM_RECORD_CMPLT_DATA_BLOCK_GET(dataBlock, tempptr);

               // seek to offset in recorded audio or video data
               UserSeek(filep[chan], dataBlock->unOffset, SEEK_SET);

               // update the recorded data
               UserWrite(dataBlock->data, 1, dataBlock->unSize, filep[chan]);
            }
            ...

            break;
         }
      }
   }
```

MM_RECORD_CMPLT_INFO_DATA_BLOCKS

```
typedef struct tagMM_RECORD_CMPLT_INFO_DATA_BLOCKS
{
     unsigned int unNumDataBlocks;  /* number of "data blocks" that follow this structure */
     unsigned int unNumBytes;          /* size of all "data blocks" in bytes */
}
MM_RECORD_CMPLT_INFO_DATA_BLOCKS, *PMM_RECORD_CMPLT_INFO_DATA_BLOCKS;
```

Description

This structure only applies to memory record and stream record. Each MM_RECORD_CMPLT_INFO_DATA_BLOCKS section may contain one or more MM_RECORD_CMPLT_INFO_DATA_BLOCK structures. Call the MM_RECORD_CMPLT_INFO_DATA_BLOCKS_GET( ) macro to retrieve a pointer to this structure when processing details. Refer to the MM_PLAY_RECORD_CMPLT_DETAILS data structure for macro instructions.

Field Descriptions

The field of this structure is described as follows:

unNumDataBlocks
: Specifies the number of MM_RECORD_CMPLT_DATA_BLOCK structures that follow this structure. The application processes that number of MM_RECORD_CMPLT_DATA_BLOCK structures in a loop.

unNumBytes
: Specifies the size of all "data blocks" in bytes.

MM_RESUME_INFO

```
typedef struct tagMM_RESUME_INFO {
   unsigned int unVersion;
   unsigned int unStreamType;
   unsigned int unAttribute;
} MM_RESUME_INFO, *PMM_RESUME_INFO;
typedef const MM_RESUME_INFO* CPMM_RESUME_INFO;
```

Description

The MM_RESUME_INFO data structure is a function input structure for the **mm_Resume( )** function.

The INIT_MM_RESUME_INFO inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion

Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unStreamType

Indicates which stream type (audio, video or both) is to be resumed. Value can be ONE of the following:

- EMM_RESUME_VIDEO_AUDIO – Resume both video and audio. If either stream is in the idle state (the play operation has not been initiated or has ended), an error will be returned for that stream type but the non-idle stream will be resumed.
- EMM_RESUME_VIDEO_ONLY – Resume video ONLY.
- EMM_RESUME_AUDIO_ONLY – Resume audio ONLY.

Note: The stream type which is to be resumed must not be idle (the play operation must have been initiated and it must not have ended). This means that the **mm_Resume( )** function will only be valid during the period of time between the MMEV_PLAY_ACK event and the MMEV_PLAY event for the requested stream type.

Note: In an audiovideo playback, if the playback is synchronous (both audio and video play are initiated by a single **mm_Play( )** function call), **mm_Resume( )** will cause BOTH the audio and video streams to be resumed, regardless of the unStreamType setting in MM_RESUME_INFO. This allows synchronization to be maintained.

unAttribute

Specifies the attributes of the **mm_Resume( )** function. Possible values include:

- EMM_RESUME_VIDEO_NEXT_FRAME – Resume video transmission from next frame following the frame at which it was paused. This is the default option when video play is resumed.
- EMM_RESUME_VIDEO_IFRAME – Retransmits from most recent video keyframe (I-frame).

MM_RESUME_PLAY_CMPLT

```
typedef struct tagMM_RESUME_PLAY_CMPLT
{
unsigned int unVersion;
unsigned int unCount;
MM_RESUME_PLAY_CMPLT_DETAILS details[MAX_RESUME_PLAY_CMPLT];
} MM_RESUME_PLAY_CMPLT, *PMM_RESUME_PLAY_CMPLT;
```

Description

The MM_RESUME_PLAY_CMPLT event information data structure is used for analyzing the results of the **mm_Resume( )** function termination or completion as reported by an MMEV_RESUME or MMEV_RESUME_FAIL event.

The INIT_MM_RESUME_PLAY_CMPLT inline function is provided to initialize the structure.

Field Descriptions

The fields of the examplestructure data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unCount
: Indicates the number of MM_RESUME_PLAY_CMPLT_DETAILS elements that follow.

details
: Provides the details of completion. See the MM_RESUME_PLAY_CMPLT_DETAILS structure definition for details.

MM_RESUME_PLAY_CMPLT_DETAILS

```
typedef struct tagMM_RESUME_PLAY_CMPLT_DETAILS
{
unsigned int unVersion;
unsigned int unStreamType;
unsigned int unRetCode;
} MM_RESUME_PLAY_CMPLT_DETAILS, *PMM_RESUME_PLAY_CMPLT_DETAILS;
typedef const MM_RESUME_PLAY_CMPLT_DETAILS* CPMM_RESUME_PLAY_CMPLT_DETAILS;
```

Description

The MM_RESUME_PLAY_CMPLT_DETAILS structure is a second-level event information data structure under the MM_RESUME_PLAY_CMPLT structure. (See Section 10.5, "Other Data Structure Levels (_DETAILS)", on page 149.)

The INIT_MM_RESUME_PLAY_CMPLT_DETAILS inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

un StreamType
: Indicates which stream type (audio, video or both) the information in this structure applies to. Value can be ONE of the following:
  - EMM_RESUME_PLAY_VIDEO_AUDIO – information applies to BOTH video and audio.
  - EMM_RESUME_PLAY_VIDEO_ONLY – information applies to video ONLY.
  - EMM_RESUME_PLAY_AUDIO_ONLY – information applies to audio ONLY.

unRetCode
: Contains the firmware return code. A non-zero value indicates an error has occurred.

MM_RET_CODE

```
typedef struct tagMM_RET_CODE
{
    unsigned int    unVersion;
    unsigned int    unRetCode;
} MM_RET_CODE, *PMM_RET_CODE;
typedef const MM_RET_CODE* CPMM_RET_CODE;
```

Description

The MM_RET_CODE *event information* data structure is used for analyzing the results of certain functions' **initiation** failure events (reported by their MMEV_xxxx_ACK_FAIL event) and certain functions' **termination or completion** failure events. Collectively, they are *non-media I/O operation failure termination events*. The event data for these metaevents contains an MM_RET_CODE structure with an error return code in the unRetCode field. These *event information error return codes* apply to any metaevent (provided by **mm_GetMetaEvent( )**) for which the event data is of type MM_RET_CODE.

The INIT_MM_RET_CODE inline function is provided to initialize the structure.

Note: The MM_RET_CODE structure is used as a typedef for the following event information data structures.

ACK event information structures (and associated events):

MM_PLAY_ACK
(MMEV_PLAY_ACK, MMEV_PLAY_ACK_FAIL)

MM_RECORD_ACK
(MMEV_RECORD_ACK, MMEV_RECORD_ACK_FAIL)

MM_RESET_ACK
(MMEV_RESET_ACK, MMEV_RESET_ACK_FAIL)

RESULT event information structures (and associated events):

MM_DISABLE_EVENTS_RESULT
(MMEV_DISABLE_EVENTS, MMEV_DISABLE_EVENTS_FAIL)

MM_ENABLE_EVENTS_RESULT
(MMEV_ENABLE_EVENTS, MMEV_ENABLEEVENTS_FAIL)

MM_OPEN_RESULT
(MMEV_OPEN, MMEV_OPEN_FAIL)

MM_RESET_RESULT
(MMEV_RESET, MMEV_RESET_FAIL)

MM_SET_PARM_RESULT
(MMEV_SETPARM, MMEV_SETPARM_FAIL)

Event information error return codes are also returned in the MM_STOP_ACK_DETAILS unRetCode field, which is used to provide details on the MM_STOP_ACK event data associated with the MMEV_STOP_ACK and MMEV_STOP_ACK_FAIL events.

Note: If the function generates a successful initiation event (e.g., MMEV_PLAY_ACK) or a successful completion or termination event (e.g., MM_ENABLE_EVENTS), the unRetCode field returns an EMMRC_OK.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unRetCode
: Provides an error return code. EMMRC_OK indicates success; all other values indicate an error or failure condition. (See Section 12.3, "Multimedia API Event Information Error Return Codes", on page 232 for related information.)

MM_RUNTIME_CONTROL

```
typedef struct tagMM_RUNTIME_CONTROL
{
        unsigned int unVersion;
        eMM_TERMINATION_REASON Reason;
        unsigned int unValue;
        eMM_TERMINATION_ACTION Action;
        struct tagMM_RUNTIME_CONTROL *next;
} MM_RUNTIME_CONTROL, *PMM_RUNTIME_CONTROL;
typedef const MM_RUNTIME_CONTROL* CPMM_RUNTIME_CONTROL;
```

Description

This structure is used by the **mm_Play( )** and **mm_Record( )**functions to set runtime control conditions.

The INIT_MM_RUNTIME_CONTROL inline function is provided to initialize the structure.

Field Descriptions

The fields of the example structure data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

Reason
: Specifies the runtime control condition. Defined values include:

- EMM_TERM_NORTC - No termination condition. Valid only as a return code.
- EMM_TERM_DIGMASK - Digit termination for a bit mask of digits received.
- EMM_TERM_DIGTYPE - Digit termination occurs upon receipt when the digit is provided as an ASCII value (for example, 0-9, a-d, *, #).
- EMM_TERM_MAXDTMF - Maximum number of digits received.
- EMM_TERM_MAXTIME- Maximum execution time for operation.
- EMM_TERM_TIME_FROM_AUDIO_RECORD_STARTED – Termination occurs T milliseconds from when audio record starts. where T is the value specified in the unValue field of this data structure.

Notes: ***1.*** The RTC condition, EMM_TERM_TIME_FROM_AUDIO_RECORD_STARTED, is not applicable to **mm_Play( )** or to video-only **mm_Record( )**. This RTC condition is triggered by the start of audio record and not by any of the MMEV events that may coincide with the trigger. Therefore, it may be used to terminate record, even if the MMEV_VIDEO_RECORD_STARTED is not enabled.

2. The following RTC conditions are not supported when using the linear PCM coder: EMM_TERM_DIGMASK, EMM_TERM_DIGTYPE, and EMM_TERM_MAXDTMF.

unValue
: Sets the value of the termination condition.

Action

Specifies the action to take upon receipt of a termination condition. Defined values include:

- EMM_TA_AUDIO_STOP - Stop current operation for audio track only.
- EMM_TA_VIDEO_STOP - Stop current operation for video track only.
- EMM_TA_AUDIO_VIDEO_STOP - Stop current operation for both audio and video tracks.

MM_SEEK_CMPLT

```
typedef struct tagMM_SEEK_CMPLT
{
unsigned int unVersion;
unsigned int unCount;
MM_SEEK_CMPLT_DETAILS details[MAX_SEEK_CMPLT];
} MM_SEEK_CMPLT, *PMM_SEEK_CMPLT;
```

Description

The MM_SEEK_CMPLT event information data structure is used for analyzing the results of the **mm_Seek( )** function termination or completion as reported by an MMEV_SEEK or MMEV_SEEK_FAIL event.

The INIT_MM_SEEK_CMPLT inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unCount
: Indicates the number of MM_SEEK_CMPLT_DETAILS elements that follow.

details
: Provides the details of completion. See the MM_SEEK_CMPLT_DETAILS structure definition for details.

MM_SEEK_CMPLT_DETAILS

```
typedef struct tagMM_SEEK_CMPLT_DETAILS
{
unsigned int unVersion;
unsigned int unStreamType;
unsigned int unRetCode;
} MM_SEEK_CMPLT_DETAILS, *PMM_SEEK_CMPLT_DETAILS;
typedef const MM_SEEK_CMPLT_DETAILS* CPMM_SEEK_CMPLT_DETAILS;
```

Description

The MM_SEEK_CMPLT_DETAILS structure is a second-level event information data structure under the MM_SEEK_CMPLT structure. (See Section 10.5, "Other Data Structure Levels (_DETAILS)", on page 149.)

The INIT_MM_SEEK_CMPLT_DETAILS inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

un StreamType
: Indicates which stream type (audio, video or both) the information in this structure applies to. Value can be ONE of the following:
  - EMM_SEEK_VIDEO_AUDIO – information applies to BOTH video and audio.
  - EMM_SEEK_VIDEO_ONLY – information applies to video ONLY.
  - EMM_SEEK_AUDIO_ONLY – information applies to audio ONLY.

unRetCode
: Specifies the firmware return code. A non-zero value indicates an error has occurred.

MM_SET_PARM

```
typedef struct tagMM_SET_PARM
{
    unsigned int    unVersion;
    eMM_PARM        eParm;
    unsigned int    unParmValue;
} MM_SET_PARM, *PMM_SET_PARM;
typedef const MM_SET_PARM* CPMM_SET_PARM;
```

Description

The MM_SET_PARM structure specifies the details of a set parameter request. This structure is used as *function input* for the **mm_SetParm( )** function.

The INIT_MM_SET_PARM inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion

Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

eParm

Specifies parameter to set. Defined values include the following channel-level parameter:

- EMM_REC_IFRAME_TIMEOUT – Specifies the time to wait for an I-frame. Video recording, or multimedia (audio and video) recording, starts when an I-frame is detected or when the time-out is reached.

  Note: See **mm_EnableEvents( )** and the MM_VIDEO_RECORD_STARTED structure for information on the related MMEV_VIDEO_RECORD_STARTED ***optional* intermediate (non-terminating) notification event**. See also the **mm_Record( )** function.

- EMM_TONE_DETECTION – Indicates whether tone detection is enabled or disabled. This parameter may not be used and is considered invalid in the **mm_SetParm( )** function if the active license does not contain video transcoding resources.

unParmValue

Specifies the value assigned to the parameter. Valid values are indicated for the following parameters:

- EMM_REC_IFRAME_TIMEOUT – Range: 0 - 0x7FFFFFFF. Units: ms. Default: 5000 (5 seconds). Example: Set to 9000 to specify 9 seconds. Zero (0) causes an immediate time-out and starts recording immediately.
- EMM_TONE_DETECTION – Possible values are:
  - EMM_TONE_DETECTION_RX_DISABLE - Disables detection of DTMF tones in the incoming audio stream on the given device (default).
  - EMM_TONE_DETECTION_RX_ENABLE - Enables detection of DTMF tones in the incoming audio stream on the given device.

MM_STOP

```
typedef struct tagMM_STOP
{
    unsigned int          unVersion;
    eMM_ITEM              ItemChain;
    eMM_STOP              ItemType;
    MM_STOP_DETAILS       details;
    struct tagMM_STOP*    next;
    struct tagMM_STOP*    prev;   /* optional */
} MM_STOP, *PMM_STOP;
typedef const MM_STOP* CPMM_STOP;
```

Description

The MM_STOP structure specifies the details of a stop device operations request. This structure is used as *function input* for the **mm_Stop( )** function.

The INIT_MM_STOP inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

ItemChain
: Specifies the next list element for iteration. Defined values include:
  - EMM_ITEM_CONT – Indicates that the next list element immediately follows the current.
  - EMM_ITEM_LINK – Indicates that the next and previous list elements linked to the current via pointers.
  - EMM_ITEM_EOT – Indicates that the current list element is the last in the list.

ItemType
: Specifies the type of the stop element. Defined values include:
  - EMM_STOP_AUDIO_PLAY – Indicates that the current item specifies stop audio play element.
  - EMM_STOP_AUDIO_RECORD – Indicates that the current item specifies stop audio record element.
  - EMM_STOP_IMAGE_PLAY – Indicates that the current item specifies stop image play element.
  - EMM_STOP_VIDEO_PLAY – Indicates that the current item specifies stop video play element.
  - EMM_STOP_VIDEO_RECORD – Indicates that the current item specifies stop video record element.

details

Specifies the details of the stop element. See definition of MM_STOP_DETAILS structure.

Note: Although this structure is **reserved for future use**, you must pass the structure and set the unRFU field to 0.

next

Points to the next MM_STOP item. Needed only when ItemChain specifies a linked list element. NULL value indicates last item.

prev

Points to the previous MM_STOP item (optional). Needed only when ItemChain specifies a linked list element. NULL value indicates first item.

MM_SEEK_INFO

```
typedef struct tagMM_SEEK_INFO {
   unsigned int unVersion;
   unsigned int unStreamType;
   unsigned int unOrigin;
   int nOffset;
} MM_SEEK_INFO, *PMM_SEEK_INFO;
typedef const MM_SEEK_INFO* CPMM_SEEK_INFO
```

Description

The MM_SEEK_INFO data structure is a function input structure of the **mm_Seek( )** function.

The INIT_MM_SEEK_INFO inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion

Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unStreamType

Indicates which stream type (audio, video or both) to seek. Value can be ONE of the following:

- EMM_SEEK_VIDEO_AUDIO – Sets the play position for both video and audio. If either stream is in the idle state (the play operation has not been initiated or has ended), an error will be returned for that stream type but the play position of the non-idle stream will be set.
- EMM_SEEK_VIDEO_ONLY – Sets play position for video ONLY.
- EMM_SEEK_AUDIO_ONLY – Sets play position for audio ONLY.

Note: The stream type whose play position is to be set must not be idle (the play operation must have been initiated and it must not have ended). This means that the **mm_Seek( )** function will only be valid during the period of time between the MMEV_PLAY_ACK event and the MMEV_PLAY event for the requested stream type.

Note: In an audiovideo playback, if the playback is synchronous (both audio and video play are initiated by a single **mm_Play( )** function call), **mm_Seek( )** will cause BOTH the audio and video streams to be sought, regardless of the unStreamType setting in MM_SEEK_INFO. This allows synchronization to be maintained.

unOrigin

Specifies the starting point for the seek operation. Valid values include:

- EMM_SEEK_SET – Start seek operation from beginning of file.
- EMM_SEEK_CUR – Start seek operation from current position in file.

If unOrigin is set to EMM_SEEK_SET, the value of unOffset must be a positive value and will be interpreted as relative to the start of the file. If it is set to EMM_SEEK_CUR, the value of unOffset will be interpreted as a signed value relative to the current play position. A

negative value in this case, will result in the the play position moving backwards and a positive value will cause it to move forward.

nOffset
Specifies an offset, in milliseconds.

MM_STOP_ACK

```
typedef struct tagMM_STOP_ACK
{
    unsigned int         unVersion;
    unsigned int         unCount;
    MM_STOP_ACK_DETAILS  details[MAX_STOP_ACK];
} MM_STOP_ACK, *PMM_STOP_ACK;
```

Description

The MM_STOP_ACK structure is an *event information* data structure is used for analyzing the results of the **mm_Stop( )** function **initiation** as reported by the MMEV_STOP_ACK or MMEV_STOP_ACK_FAIL event.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unCount
: Specifies the number of MM_STOP_ACK_DETAILS elements that follow. Maximum value is 16.

details
: Specifies the details of completion. See definition of MM_STOP_ACK_DETAILS structure.

MM_STOP_ACK_DETAILS

```
typedef struct tagMM_STOP_ACK_DETAILS
{
    unsigned int         unVersion;
    eMM_STOP             ItemType;
    unsigned int         unRetCode;
} MM_STOP_ACK_DETAILS, *PMM_STOP_ACK_DETAILS;
```

Description

The STOP_ACK_DETAILS structure is a second-level *event information* data structure under the MM_STOP_ACK structure. (See Section 10.5, "Other Data Structure Levels (_DETAILS)", on page 149.)

The INIT_MM_STOP_ACK_DETAILS inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

ItemType
: Specifies the stopped item. Defined values include:
  - EMM_STOP_AUDIO_PLAY – Indicates that the current item specifies stop audio play element.
  - EMM_STOP_AUDIO_RECORD – Indicates that the current item specifies stop audio record element.
  - EMM_STOP_IMAGE_PLAY – Indicates that the current item specifies stop image play element.
  - EMM_STOP_VIDEO_PLAY – Indicates that the current item specifies stop video play element.
  - EMM_STOP_VIDEO_RECORD – Indicates that the current item specifies stop video record element.

unRetCode
: Provides an error return code for the stopped item. EMMRC_OK indicates success; all other values indicate an error or failure condition. (See Section 12.3, "Multimedia API Event Information Error Return Codes", on page 232 for related information.) Defined values applicable to this operation include:
  - EMMRC_A_INVALID_STATE – Invalid state (audio).
  - EMMRC_ALREADYSTOPPED – Device operations are already stopped.
  - EMMRC_I_INVALID_STATE – Invalid state (image).
  - EMMRC_INVALIDARG – Invalid argument.
  - EMMRC_V_INVALID_STATE – Invalid state (video).

MM_STOP_DETAILS

```
typedef struct tagMM_STOP_DETAILS
{
    unsigned int    unVersion;
    unsigned int    unRfu;
} MM_STOP_DETAILS, *PMM_STOP_DETAILS;
typedef const MM_STOP_DETAILS* CPMM_STOP_DETAILS;
```

Description

Note: Although this structure is **reserved for future use**, you must pass the structure and set the unRFU field to 0.

This structure is a second-level *function input* data structure under the MM_STOP structure, which is used as *function input* for the **mm_Stop( )** function. (See Section 10.5, "Other Data Structure Levels (_DETAILS)", on page 149.)

The INIT_MM_STOP_DETAILS inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unRFU
: **Reserved for future use**. Must be set to 0.

MM_STREAM_OPEN_INFO

```
typedef struct tagMM_STREAM_OPEN_INFO
{
        unsigned int unVersion;
        unsigned int unBufferSize;
        eMM_STREAM_MODE BufferMode;
} MM_STREAM_OPEN_INFO, *PMM_STREAM_OPEN_INFO;
typedef const MM_STREAM_OPEN_INFO *CPMM_STREAM_OPEN_INFO
```

Description

The MM_STREAM_OPEN_INFO structure specifies the details of the stream open request. This structure is a function input structure for the **mm_StreamOpen( )** function.

The INIT_MM_STREAM_OPEN_INFO inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unBufferSize
: Specifies the buffer size in bytes. Valid value is >1024.

BufferMode
: Specifies the stream access mode. Defined values include:
  - EMM_SM_READ - Streaming from board; read operation permitted only.
  - EMM_SM_WRITE - Streaming to board; write operation permitted only.

MM_STREAM_STAT

```
typedef struct tagMM_STREAM_STAT
{
    unsigned int unVersion;
    unsigned int unBytesIn;
    unsigned int unBytesOut
    unsigned char *pHeadPointer;
    unsigned char *pTailPointer;
    unsigned int unCurrentState;
    unsigned int unNumberOfBufferUnderruns;
    unsigned int unNumberOfBufferOverruns;
    unsigned int unBufferSize;
    unsigned int unBufferMode;
    unsigned int unSpaceAvailable;
    unsigned int unHighWaterMark;
    unsigned int unLowWaterMark;
} MM_STREAM_STAT, *PMM_STREAM_STAT;
typedef const MM_STREAM_STAT *CPMM_STREAM_STAT
```

Description

This structure is a function output for the **mm_StreamGetStat( )** function.

The INIT_MM_STREAM_STAT inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unBytesIn
: Total number of bytes put into stream. Internal values; read access only.

unBytesOut
: Total number of bytes sent to board. Internal values; read access only.

pHeadPointer
: Pointer to position in stream. Internal values; read access only.

pTailPointer
: Pointer to position in stream buffer. Internal values; read access only.

unCurrentState
: Idle, Streaming, etc. Internal values; read access only.

unNumberOfBufferUnderruns
: Number of buffer underruns. Internal values; read access only.

unNumberOfBufferOverruns
: Number of buffer overruns. Internal values; read access only.

unBufferSize
: Size of buffer in bytes. Internal values; read access only.

unBufferMode
Read or write mode. Internal values; read access only.

unSpaceAvailable
Bytes available in stream buffer. Internal values; read access only.

unHighWaterMark
High water mark for stream buffer. Internal values; read access only.

unLowWaterMark
Low water mark for stream buffer. Internal values; read access only.

MM_STREAM_WATERMARK_INFO

```
typedef struct tagMM_STREAM_WATERMARK_INFO
{
    unsigned int                    unVersion;
    eMM_STREAM_WATERMARK_LEVEL      Level;
     unsigned int                   unValue;
}MM_STREAM_WATERMARK_INFO, *PMM_STREAM_WATERMARK_INFO;
typedefMM_STREAM_WATERMARK_INFO* CPMM_STREAM_WATERMARK_INFO;
```

Description

The MM_STREAM_WATERMARK_INFO structure specifies details for setting watermark levels. This structure is a function input structure for the **mm_StreamSetWaterMark( )** function.

The INIT_MM_STREAM_WATERMARK_INFO inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

Level
: Specifies the watermark level. Defined values include:
  - EMM_WM_HIGH - High watermark level.
  - EMM_WM_LOW - Low watermark level.

unValue
: Specifies the watermark level in bytes from the start of the buffer (stream buffer begins filling from buffer start). Valid values are between 0 and the size of the buffer in bytes.

MM_VIDEO_CODEC

```
typedef struct tagMM_VIDEO_CODEC
{
    unsigned int             unVersion;
    eVIDEO_CODING            Coding;
    eVIDEO_PROFILE           Profile; /* Profile ID for H.263/H.264, Profile and Level ID for
                                         MPEG-4  */
    eVIDEO_LEVEL             Level;   /* Ignored for MPEG-4 */
    eVIDEO_IMAGE_WIDTH       ImageWidth;
    eVIDEO_IMAGE_HEIGHT      ImageHeight;
    eVIDEO_BITRATE           BitRate;
    eVIDEO_FRAMESPERSEC      FramesPerSec;
    eVIDEO_SAMPLING_RATE     SamplingRate;
    unsigned int             VisualConfigSize;
    unsigned char*           VisualConfiguration;
    eH264_ACCEPT_REDUNDAND_SLICES   H264_AcceptRedundantSlices;
    unsigned int                    H264_ProfileIOP;
    eH264_PACKETIZATION_MODE        H264_PacketizationMode;
    unsigned int                    H264_MaxNalUnitSize;
} MM_VIDEO_CODEC, *PMM_VIDEO_CODEC;
typedef const MM_VIDEO_CODEC* CPMM_VIDEO_CODEC;
```

Description

The MM_VIDEO_CODEC structure specifies the characteristics of the video coder. This structure is a nested *function input* structure for the **mm_Play( )** or **mm_Record( )** function. It is nested directly under the MM_MEDIA_VIDEO structure, and it is a sixth-level structure under the MM_PLAY_RECORD_INFO (MM_PLAY_INFO, MM_RECORD_INFO) structure. (For a description of these *nesting levels*, see Section 10.4, “Play/Record Data Structure Levels”, on page 149.)

The INIT_MM_VIDEO_CODEC inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

Coding
: Specifies the type of video coding. Defined values include:
  - VIDEO_CODING_H263 – H263
  - VIDEO_CODING_H263_1998
  - VIDEO_CODING_H264 - H264
  - VIDEO_CODING_MP4V_ES – MPEG-4
  - VIDEO_CODING_DEFAULT = VIDEO_CODING_H263

Profile
: Specifies the video profile. Defined values include:
  - VIDEO_PROFILE_0_H263 – H263 Profile 0
  - VIDEO_PROFILE_1_H263 – H263 Profile 1
  - VIDEO_PROFILE_2_H263 – H263 Profile 2

- VIDEO_PROFILE_3_H263 – H263 Profile 3
- VIDEO_PROFILE_4_H263 – H263 Profile 4
- VIDEO_PROFILE_5_H263 – H263 Profile 5
- VIDEO_PROFILE_6_H263 – H263 Profile 6
- VIDEO_PROFILE_7_H263 – H263 Profile 7
- VIDEO_PROFILE_8_H263 – H263 Profile 8
- VIDEO_PROFILE_9_H263 – H263 Profile 9
- VIDEO_PROFILE_LEVEL_SP0_MPEG-4 – MPEG-4 Simple Profile/Level 0
- VIDEO_PROFILE_LEVEL_SP1_MPEG-4 – MPEG-4 Simple Profile/Level 1
- VIDEO_PROFILE_LEVEL_SP2_MPEG-4 – MPEG-4 Simple Profile/Level 2
- VIDEO_PROFILE_LEVEL_SP3_MPEG-4 – MPEG-4 Simple Profile/Level 3
- VIDEO_PROFILE_DEFAULT = VIDEO_PROFILE_0_H263
- VIDEO_PROFILE_BASELINE_H264 – The only acceptable profile for H.264. The values are defined by the H.264 specification and have been enumerated as such in the eVIDEO_PROFILE, eMM_VIDEO_PROFILE enumerations.

Note: A SIP/H.264 application would typically parse the passed SDP to determine the correct value:
m=video 5108 RTP/AVP 112
a=rtpmap:112 h264/90000
a=fmtp:112 profile-level-id=42E00D; packetization-mode=1

In this case the profile-level-id field specifies the profile of 0x42, which is the baseline profile.

Level

Specifies the video signal level. The only acceptable level for H.264. The values are defined by the H.264 specification and have been enumerated as such in the eVIDEO_LEVEL, eMM_VIDEO_LEVEL enumerations.

Note: Set a video signal level that is appropriate for the frame size and frame rate. No field validation is performed by the software.

Defined values include:

- VIDEO_LEVEL_10_H263 – H263 Level 10
- VIDEO_LEVEL_20_H263 – H263 Level 20
- VIDEO_LEVEL_30_H263 – H263 Level 30
- VIDEO_LEVEL_40_H263 – H263 Level 40
- VIDEO_LEVEL_45_H263 – H263 Level 45
- VIDEO_LEVEL_50_H263 – H263 Level 50
- VIDEO_LEVEL_60_H263 – H263 Level 60
- VIDEO_LEVEL_70_H263 – H263 Level 70
- VIDEO_LEVEL_DEFAULT = VIDEO_LEVEL_10_H263
- VIDEO_LEVEL_1_H264 – -H.264 level 1
- VIDEO_LEVEL_1_B_H264 – h.264 level 1B
- VIDEO_LEVEL_1_1_H264 – H.264 level 1.1
- VIDEO_LEVEL_1_2_H264 – H.264 level 1.2
- VIDEO_LEVEL_1_3_H264 – H.264 level 1.3

Note: A SIP/H.264 application would typically parse the passed SDP to determine the correct value:
m=video 5108 RTP/AVP 112
a=rtpmap:112 h264/90000
a=fmtp:112 profile-level-id=42E00D; packetization-mode=1

In this case the profile-level-id field specifies the profile of 0x0d, decimal 13, so the correct value is VIDEO_LEVEL_1_3_H264.

ImageWidth

Specifies the width of the video image in pixels per line. Defined values include:

- VIDEO_IMAGE_WIDTH_128 – Sub-QCIF
- VIDEO_IMAGE_WIDTH_176 – QCIF
- VIDEO_IMAGE_WIDTH_352 – CIF
- VIDEO_IMAGE_WIDTH_704 – 4CIF
- VIDEO_IMAGE_WIDTH_1408 – 16CIF
- VIDEO_IMAGE_WIDTH_DEFAULT = VIDEO_IMAGE_WIDTH_176

ImageHeight

Specifies the height of the video image in number of lines. Defined values include:

- VIDEO_IMAGE_HEIGHT_96 – Sub-QCIF
- VIDEO_IMAGE_HEIGHT_144 – QCIF
- VIDEO_IMAGE_HEIGHT_288 – CIF
- VIDEO_IMAGE_HEIGHT_576 – 4CIF
- VIDEO_IMAGE_HEIGHT_1152 – 16CIF
- VIDEO_IMAGE_HEIGHT_DEFAULT = VIDEO_IMAGE_HEIGHT_144

BitRate

Specifies the output bit rate of the video signal in bits per second. Valid values depend on whether video transcoding is enabled. (Video transcoding is enabled if the DMFL_TRANSCODE_ON flag is set for video port connections when **dev_PortConnect( )** is called.) Suggested values include:

- VIDEO_BITRATE_DEFAULT – 50kbps
- VIDEO_BITRATE_40K – 40kbps
- VIDEO_BITRATE_64K – 64kbps
- VIDEO_BITRATE_128K – 128kbps
- VIDEO_BITRATE_384K – 384kbps

When video transcoding is disabled, valid values include:

- EMM_VIDEO_BITRATE_DEFAULT – Output bit rate will match the input bit rate. For **mm_Play( )**, this value sets the output bit rate to the file bit rate. For **mm_Record( )**, this value sets the file bit rate to the input bit rate. No transrating is performed.

When video transcoding is enabled, this field is only applicable to **mm_Record( )** and it sets the file bit rate. Valid values include:

- EMM_VIDEO_BITRATE_DEFAULT – Default bit rate will be 50, 000 bits per second.
- Greater than zero – Bit rate will be set to specified value.

Note: When the application specifies custom bit rates by using a value greater than zero, the value may need to be typecast to an eVIDEO_BITRATE data type.

FramesPerSec

Specifies the video frame rate. Defined values include:

- VIDEO_FRAMESPERSEC_6 – frame rate of 6 fps
- VIDEO_FRAMESPERSEC_10 – frame rate of 10 fps
- VIDEO_FRAMESPERSEC_15 – frame rate of 15 fps
- VIDEO_FRAMESPERSEC_30 – frame rate of 30 fps
- VIDEO_FRAMESPERSEC_DEFAULT = frame rate of 15 fps

SamplingRate

Specifies the video sampling rate. Defined values include:

- VIDEO_SAMPLING_RATE_DEFAULT – VIDEO_SAMPLING_RATE_90000
- VIDEO_SAMPLING_RATE_90000 – 90KHz

VisualConfigSize

Specifies the size in bytes of the visual configuration data specified in the VisualConfiguration field. Set to 0 if there is no visual configuration data.

Note: In SIP/H.264, the application would set the VisualConfigSize if that data was passed on the in the SDP. For example:

m=video 49170 RTP/AVP 98
a=rtpmap:98 H264/90000
a=fmtp:98 profile-level-id=42A01E; sprop-parameter-sets=Z0IACpZTBYmI,aMljiA==

In this case, VisualConfigSize should be set to 21.

VisualConfiguration

Points to the hexadecimal sequence of bytes that specifies MPEG-4 visual configuration data. Set to NULL if there is no visual configuration data to be specified.

Note: In SIP/H.264, the application would set VisualConfiguration if that data was passed on the in the SDP. For example:

m=video 49170 RTP/AVP 98
a=rtpmap:98 H264/90000
a=fmtp:98 profile-level-id=42A01E; sprop-parameter-sets=Z0IACpZTBYmI,aMljiA==

In this case, szVisualConfiguration should be set to Z0IACpZTBYmI,aMljiA==.

eH264_ACCEPT_REDUNDAND_SLICES

Reserved for future use. Applications should set this value to the default, which is ACCEPT_REDUNDAND_SLICES_FALSE.

eH264_ProfileIOP

This sets the profile-iop field as defined in RFC3984. An HMP application which uses SIP to pass H.264 over IP would normally get this value from the SDP passed during the SIP negotiation. For example:

m=video 5108 RTP/AVP 112
a=rtpmap:112 h264/90000
a=fmtp:112 profile-level-id=42E00D; packetization-mode=1

In this case, the "profile-level-ld" field specifies the profile (0x42) , profile-iop (0xE0), and level (0x0D). The application should set the value of eH264_ProfileIOP to 0xE0.

eH264_PACKETIZATION_MODE

This sets the encoders output packetization mode. The Packetization modes are defined in RFC3984. Applications can use the new enumeration defined:

```
typedef enum tagH264_PACKETIZATION_MODE
{
```

```
        H264_PACKETIZATION_MODE_SINGLE_NAL   = 0,
        H264_PACKETIZATION_MODE_NON_INTERLEAVED  = 1,
        H264_PACKETIZATION_MODE_INTERLEAVED   = 2,
        H264_PACKETIZATION_MODE_DEFAULT =
        H264_PACKETIZATION_MODE_NON_INTERLEAVED
    eH264_PACKETIZATION_MODE;
```

A SIP/H.264 application would typically parse the passed SDP to determine the correct value:

m=video 5108 RTP/AVP 112
a=rtpmap:112 h264/90000
a=fmtp:112 profile-level-id=42E00D; packetization-mode=1

In this case, the correct value is 1, which has an enumeration value of H264_PACKETIZATION_MODE_NON_INTERLEAVED.

unH264_MaxNalUnitSize
: This sets the maximum NAL unit size generated by the H.264 encoder. The default is 1400 bytes. 1400 bytes allows for the NAL Unit to easily fit within an Ethernet MTU sized RTP packet without fragmentation.

Note: Video enumerations are defined in the header file, *videodefs.h*.

MM_VIDEO_RECORD_STARTED

```
typedef struct tagMM_VIDEO_RECORD_STARTED
{
    unsigned int    unVersion;
    unsigned int    unStatus;
} MM_VIDEO_RECORD_STARTED, *PMM_VIDEO_RECORD_STARTED;
```

Description

The MM_VIDEO_RECORD_STARTED *event information* data structure is used for analyzing the results of an **mm_Record( )** function as reported by the MMEV_VIDEO_RECORD_STARTED ***optional*** **intermediate (non-terminating) notification event** (enabled by default or by the **mm_EnableEvents( )** function).

The INIT_MM_VIDEO_RECORD_STARTED inline function is provided to initialize the structure.

Field Descriptions

The fields of the structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

unStatus
: Specifies the status of I-frame detection. Defined values include:
  - EMM_VIDEO_RCRD_IFRAME_DETECTED – IFrame detected.
  - EMM_VIDEO_RCRD_IFRAME_TIMEOUT – IFrame detection time-out is reached.

  Note: Refer to the MM_SET_PARM data structure for information on the EMM_REC_IFRAME_TIMEOUT parameter.

MM_YUV

```
typedef struct tagMM_YUV
{
     unsigned int          unVersion;
     eMTK_YUV_IMAGE_FORMAT eFormat;
     unsigned int          unWidth;
     unsigned int          unHeight;
}MM_YUV, *PMM_YUV;
```

Description

This data structure specifies the contents of a YUV bitmap.

The INIT_MM_YUV inline function is provided to initialize the structure.

Field Descriptions

The fields of this data structure are described as follows:

unVersion
: Version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. Use the inline function to initialize this field to the current version. Do not modify this field after it has been initialized.

eFormat
: Specifies the YUV data format of the image contents. Valid value is eMTK_YUV_IMAGE_FORMAT_420.

  Note: The YUV 4:2:0 format is supported for still image play and capture in this release.

unWidth
: Specifies the width of each row in pixels per line.

unHeight
: Specifies the height of each column in number of lines.

Error Codes 12

This chapter describes the errors supported by the Dialogic® Multimedia API and covers the following topics:

- Overview of Dialogic® Multimedia API Errors 229
- Dialogic® Multimedia API Function Error Codes 231
- Multimedia API Event Information Error Return Codes 232
- Terminating and Non-Terminating Play/Record Errors 235

12.1 Overview of Dialogic® Multimedia API Errors

The following information describes the basic categories of errors generated in the Dialogic® Multimedia API:

- Function Error Codes for EMM_FAILURE
- Unsolicited Failure Event (MMEV_ERROR) Error Information
- Failure Termination Events (MMEV_xxxx_FAIL) and Error Information

Function Error Codes for EMM_FAILURE

The Dialogic® Multimedia API functions return a value, which in most cases, is EMM_SUCCESS (or 0) for a successful result and EMM_ERROR (or -1) for an error or an unsuccessful result.

If a Multimedia API function returns EMM_ERROR to indicate a failure, use the **mm_ErrorInfo( )** function to retrieve the reason for the error. The **mm_ErrorInfo( )** function outputs in the MM_INFO data structure an error value and error message specific to the Multimedia API function. To retrieve the correct information, the application program must call **mm_ErrorInfo( )** function immediately after the Multimedia API function fails, otherwise, the MM_INFO data may be outdated or invalid.

Note: Multimedia API function errors are thread-specific (they are only in scope for that thread).

For a list of the *function error codes* and messages returned in the MM_INFO structure, see Section 12.2, "Dialogic® Multimedia API Function Error Codes", on page 231.

Unsolicited Failure Event (MMEV_ERROR) Error Information

If the MMEV_ERROR unsolicited failure event occurs, use **mm_GetMetaEvent( )** to retrieve the reason for the error. The **mm_GetMetaEvent( )** function outputs the MM_ERROR_RESULT event data associated with the metaevent in the MM_METAEVENT data structure. To retrieve the correct information, the application must call **mm_GetMetaEvent( )** immediately after the failure event arrives and before the next Multimedia API event is requested, otherwise, the metaevent data

may be outdated or invalid. For a list of the error codes and messages returned in the MM_INFO structure, see Section 12.2, "Dialogic® Multimedia API Function Error Codes", on page 231.

Failure Termination Events (MMEV_xxxx_FAIL) and Error Information

If an error occurs during execution of an asynchronous function, a failure termination event is sent to the application. The Multimedia API failure termination events contain an MMEV_ prefix and a _FAIL suffix (MMEV_xxxx_FAIL); for example, MMEV_RECORD_FAIL.

Note: The application must call the **mm_ResultInfo( )** function to retrieve error information after receiving a _FAIL event.

If a Multimedia API function generates a failure termination event, use the **mm_GetMetaEvent( )** function to retrieve the reason for the error. The **mm_GetMetaEvent( )** function outputs metaevent information in the MM_METAEVENT data structure. If the MMME_MM_EVENT bit is set in the flags field, it indicates that it is a Multimedia API event, and the evtdatap field points to event data that contains failure information specific to the Multimedia API function. To retrieve the correct information, the application must call **mm_GetMetaEvent( )** immediately after the failure event arrives and before the next Multimedia API event occurs, otherwise, the metaevent data may be outdated or invalid.

There are two error information types for failure termination events:

MMEV_PLAY_FAIL and MMEV_RECORD_FAIL Termination Event Error Information

If an **mm_Play( )** or **mm_Record( )** function generates a MMEV_PLAY_FAIL or MMEV_RECORD_FAIL *failure termination event*, see Section 12.5, "Terminating and Non-Terminating Play/Record Errors", on page 235 for information on how to retrieve the error information.

Event Information Error Return Codes (EMMRC_xxxx)

The metaevent information that is associated with a *non-media I/O operation failure termination event* contains an error return code. These *event information error return codes* apply to any metaevent (provided by **mm_GetMetaEvent( )**) for which the event data is of type MM_RET_CODE, which includes the following event information data structures:

ACK event information structures

- MM_PLAY_ACK
- MM_RECORD_ACK
- MM_RESET_ACK

RESULT event information structures

- MM_DISABLE_EVENTS_RESULT
- MM_ENABLE_EVENTS_RESULT
- MM_OPEN_RESULT
- MM_RESET_RESULT

- MM_SET_PARM_RESULT

The MM_RET_CODE event information structure provides in its unRetCode field the error return code related to an event.

Event information error return codes are also returned in the MM_STOP_ACK_DETAILS unRetCode field, which is used to provide details on the MM_STOP_ACK event data associated with the MMEV_STOP_ACK and MMEV_STOP_ACK_FAIL events.

The unRetCode field returns an error return code (see Section 12.3, "Multimedia API Event Information Error Return Codes", on page 232).

Note: If the function generates a successful initiation event (e.g., MMEV_PLAY_ACK) or a successful completion or termination event (e.g., MM_ENABLE_EVENTS), the unRetCode field returns an EMMRC_OK.

12.2 Dialogic® Multimedia API Function Error Codes

The function error codes are defined in *mmerrs.h* header file. The Dialogic® Multimedia API can generate the following function error codes (listed in alphabetical order):

EMM_BADDEV
Invalid device descriptor.

EMM_BADPARM
Invalid parameter in function call.

EMM_BADPROD
Function is not supported on this board.

EMM_BUSY
Device is already busy.

EMM_FWERROR
Firmware error.

EMM_IDLE
Device is idle.

EMM_NOERROR
No errors.

EMM_NOSUPPORT
Data format not supported.

EMM_NOTIMP
Function is not implemented.

EMM_SYSTEM
System error.

EMM_TIMEOUT
Function timed out.

12.3 Multimedia API Event Information Error Return Codes

The event information error return codes are defined in the *mmerrs.h* header file. The API can generate the following event information error return codes (listed in alphabetical order):

EMMRC_A_FILE_OPEN_FAILED
Audio file open failed.

EMMRC_A_INVALID_STATE
Invalid state (audio).

EMMRC_A_NO_HINT_TRACK
Missing hint file in 3GP audio file playback.

EMMRC_A_NO_TRACK
Requested track does not exist in 3GP audio file playback.

EMMRC_A_TOO_MANY_TRACKS
Too many tracks in 3GP audio file playback. Only one track is supported.

EMMRC_A_WAVE_FILE_ERROR
The information inside the Wave file is invalid.

EMMRC_AV_INVALID_STATE
Invalid state (audio/video).

EMMRC_ALREADYSTOPPED
Device operations are already stopped.

EMMRC_EOF
The end of file was detected during the file input.

EMMRC_FAILED
Unspecified failure.

EMMRC_FILE_CORRUPT
Corrupt file in 3GP file playback.

EMMRC_FILEREAD_FAILED
File read failed.

EMMRC_FILEWRITE_FAILED
An error occurred when writing to a file.

EMMRC_I_FILE_OPEN_FAILED
Image file open failed.

EMMRC_I_INVALID_DATA
Image data not valid.

EMMRC_I_INVALID_STATE
Invalid state (image).

EMMRC_INVALID_FILEFORMAT
Invalid file format.

EMMRC_INVALIDARG
Invalid argument.

EMMRC_INVALIDSTATE_ERROR
Invalid state.

EMMRC_LICENSE_IN_USE
This error code is currently not implemented.

EMMRC_MEMALLOC_ERROR
Memory allocation error.

EMMRC_MEMALLOC_POOLNOTFOUND
Memory allocation pool not found.

EMMRC_NOT_IMAGE_FILE
Image file format is invalid.

EMMRC_NOT_VIDEO_FILE
Video file format is invalid.

EMMRC_OK
Successful (no error).

EMMRC_OUT_OF_VIDEO_LICENSE
Out of the video licenses.

EMMRC_RESOURCE_IN_USE
The coder is already in use on this channel. Currently only one coder is supported per channel.

EMMRC_UNKNOWN_ERROR
Unknown error.

EMMRC_UNSUPPORTED_MODE
Unsupported mode.

EMMRC_UNSUPPORTED_RESOURCE
The specified video coder type is not supported.

EMMRC_V_FILE_OPEN_FAILED
Video file open failed.

EMMRC_V_INVALID_STATE
Invalid state (video).

EMMRC_V_NO_HINT_TRACK
Missing hint file in 3GP video file playback.

EMMRC_V_NO_TRACK
Requested track does not exist in 3GP video file playback.

EMMRC_V_TOO_MANY_TRACKS
Too many tracks in 3GP video file playback. Only one track is supported.

EMMRC_XCODE_FILE_IO_ERROR
An error detected during the file I/O operations.

EMMRC_XCODE_FILE_OPEN_ERROR
The file could not be opened.

EMMRC_XCODE_FILE_SIZE_LARGE
The number of bytes in the file is larger than can be handled by the transcoder.

EMMRC_XCODE_GENERIC_ERROR
A generic error returned by the transcoder.

EMMRC_XCODE_IMAGE_DATA_SIZE_MISMATCH
The number of bytes in the file is different from the expected number for this type of format. For example, the number of bytes in the input file for YUV frame of a specified size is less than the number of bytes required to represent that frame.

EMMRC_XCODE_TOO_MANY_OVERLAYS
The number of still image overlays exceeds the maximum supported by the transcoder.

12.4 Media Streaming Event Information Error Return Codes

The event information error return codes are defined in the *mmerrs.h* header file. The API can generate the following event information error return codes associated with media streaming functionality.

eMM_TERMINATION_REASON
EMM_TERM_NORTC No termination condition. Valid only as a return code.
EMM_TERM_DIGMASK Digit termination for a bit mask of digits received.
EMM_TERM_DIGTYPE Digit termination for user-defined tone.
EMM_TERM_MAXDTMF Maximum number of digits received.
EMM_TERM_MAXTIME Maximum execution time for operation.
EMM_TERM_TIME_FROM_AUDIO_RECORD_STARTED Termination occurs T milliseconds from when audio record starts, where T is the value specified in the unValue field of the MM_RUNTIME_CONTROL structure.

Note: The RTC condition, EMM_TERM_TIME_FROM_AUDIO_RECORD_STARTED, is not applicable to **mm_Play( )** or to video-only **mm_Record( )**. This RTC condition is triggered by the start of audio record and not by any of the MMEV events that may coincide with the trigger. Therefore, it may be used to terminate record, even if the MMEV_VIDEO_RECORD_STARTED is not enabled.

eMM_CMPLT_PLAY_RECORD_REASON
EMM_TR_DIGMASK
mm_Play( ) or **mm_Record( )** terminated upon reception of one of the digits in the EMM_TERM_DIGMASK bitmask.

EMM_TR_DIGTYPE
mm_Play( ) or **mm_Record( )** terminated upon reception of the digit specified by EMM_TERM_DIGTYPE.

EMM_TR_MAXTIME
mm_Play( ) or **mm_Record( )** terminated upon reaching the duration specified in EMM_TERM_MAXTIME.

eMM_TERMINATION_ACTION

EMM_TA_AUDIO_STOP	Stop current operation for audio track only.
EMM_TA_VIDEO_STOP	Stop current operation for video track only.
EMM_TA_AUDIO_VIDEO_STOP	Stop current operation for both audio and video tracks.

eMM_STREAM_MODE

EMM_SM_WRITE	Streaming to board; write operation permitted only.
EMM_SM_READ	Streaming from board; read operation permitted only.
EMMRC_A_FILE_OPEN_FAILED	Audio file open failed.
EMMRC_A_INVALID_STATE	Invalid state (audio).

12.5 Terminating and Non-Terminating Play/Record Errors

The **mm_Play( )** function generates an MMEV_PLAY_FAIL termination event upon encountering an error during the playback operation. The **mm_Record( )** function generates an MMEV_RECORD_FAIL termination event upon completion or termination of the record operation. It indicates that an *intermediate (non-terminating) error* occurred during the operation. The following information describes the error information for these failure events, as well as the corresponding non-failure events:

MMEV_PLAY_FAIL and MMEV_PLAY

The metaevent data associated with these termination events is provided in an MM_PLAY_CMPLT (MM_PLAY_RECORD_CMPLT) structure. Within this structure, details on the event termination are provided by the MM_PLAY_RECORD_CMPLT_DETAILS details second-level data structure.

Upon encountering an error, the playback terminates with an **MMEV_PLAY_FAIL**. The Reason field in the details structure indicates an EMM_TR_ERROR, and the Status field in the details structure indicates the type of error:

- EMM_STATUS_PLAY_A_FILEREAD_ERROR – Audio file read error.
- EMM_STATUS_PLAY_V_FILEREAD_ERROR – Video file read error.
- EMM_STATUS_PLAY_I_FILEREAD_ERROR – Image file read error.
- EMM_STATUS_PLAY_I_NULLDATA_ERROR – Image bitmap data pointer is null.
- EMM_STATUS_PLAY_V_ERROR_FS_GT_MFS – Video frame size greater than maximum frame size of internal buffer.
- EMM_STATUS_PLAY_BAD_PACKET – File/stream being played contains invalid values.
- EMM_STATUS_PLAY_CODING_ERROR – Data format specified in the "unCoding" field of the MM_AUDIO_CODEC structure is either not supported or does not match the format of the data contained in the played file.

If the playback terminates with an **MMEV_PLAY**, the Reason field in the details structure indicates a termination reason, such as MM_TR_EOF, meaning it is a successful completion (end of file), or EMM_TR_USERSTOP, which means successful termination by the **mm_Stop( )** function. The Status field in the details structure indicates:

- EMM_STATUS_SUCCESS – Successful completion or successful termination

MMEV_RECORD_FAIL and MMEV_RECORD

The metaevent data associated with these termination events is provided in an MM_RECORD_CMPLT (MM_PLAY_RECORD_CMPLT) structure. Within this structure,

details on the event termination are provided by the MM_PLAY_RECORD_CMPLT_DETAILS details second-level data structure.

If the recording terminates with an **MMEV_RECORD_FAIL**, it indicates that an intermediate (non-terminating) error occurred during the record operation. The Reason field in the details structure indicates an EMM_TR_ERROR, and the Status field in the details structure indicates the type of **non-terminating error**:

- EMM_STATUS_RCRD_V_DRPD_FRAME_FULL_ERROR – Full video frame dropped error.
- EMM_STATUS_RCRD_V_PKTS_DROPD_FS_GT_MFS – Video packets dropped frame size greater than maximum frame size of internal buffer.
- EMM_STATUS_RCRD_A_DRPD_FRAME_FULL_ERROR – Full audio frame dropped error.
- EMM_STATUS_RCRD_V_MISSING_MPEG-4_VISUALCONFIG_ERROR – MPEG-4 visual configuration data is not present in the incoming video stream and it has not been specified by the application in the **mm_Record( )** function.
- EMM_STATUS_RCRD_CODING_ERROR – Encoding to the data format specified in the "unCoding" field of the MM_AUDIO_CODEC structure is not supported.

If the recording terminates with an **MMEV_RECORD**, the Reason field in the details structure indicates a termination reason, such as EMM_TR_USERSTOP, which indicates successful termination by the **mm_Stop( )** function. The Status field in the details structure indicates:

- EMM_STATUS_SUCCESS – Successful completion or successful termination (no error).