Dialogic
Making Innovation Thrive™

Dialogic® IP Media Library API

Programming Guide and Library Reference

October 2009

05-2257-016

Publication Date: October 2009

Document Number: 05-2257-016

Contents

Figures

Tables

Revision History

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

Document No.	Publication Date	Description of Revisions
05-2257-016	October 2009	Feature Support by Platform chapter: Added Dialogic® HMP Software 4.1LIN in the Feature Support by Platform table. Quality of Service (QoS) Alarms and RTCP Reports chapter: Corrected unDebounceOn and unDebounceOff field descriptions in QoS Threshold Attributes. Function Summary by Category chapter: Added Dialogic® HMP Software 4.1LIN, HMP Software 3.1LIN, and HMP Software 3.0WIN in the API Function Support by Platform table. ipm_InitResponseSend( ) function: Specified Dialogic® HMP Software (Linux) in Platform row in function syntax table, as this function is supported on Linux only. ipm_InitSend( ) function: Specified Dialogic® HMP Software (Linux) in Platform row in function syntax table, as this function is supported on Linux only. ipm_StartMedia( ) function: Corrected example code.
05-2257-015	February 2009	Feature Support by Platform chapter: Added RTCP reports (enhanced) feature. Added NAT Traversal feature. Updated "Codec: AMR-NB" to include support for Dialogic® HMP Software 3.1LIN (erroneously marked not supported). Quality of Service (QoS) Alarms and RTCP Reports chapter: Updated chapter title to include RTCP Reports. Added RTCP Reporting section and updated chapter with RTCP reporting alarms and events. Using NAT Traversal in SIP Media Session chapter: New. Function Summary by Category chapter: Added ipm_GetSessionInfoEx( ) to Media Session category. ipm_DisableEvents( ) and ipm_EnableEvents( ) functions: Added EVT_RTCP_JB_HIGH, EVT_RTCP_JB_LOW, EVT_RTCP_SCS event types. Added EVT_ENDPOINTID_NOTIFY event type. ipm_GetSessionInfo( ) function: Added that this function returns parsed data. ipm_GetSessionInfoEx( ) function: New for enhanced RTCP reporting. Events chapter: Added IPMEV_RTCP_NOTIFY_RECEIVED, IPMEV_RTCP_NOTIFY_SENT, and IPMEV_NOTIFY_ENDPOINTID. IPM_ENDPOINTID_INFO structure: New for NAT Traversal feature. IPM_PARM_INFO structure: Added PARMCH_RTCP_ENHANCED_EVENT_FREQ and PARMCH_RTCP_ENHANCED_REPORTING. IPM_QOS_ALARM_DATA, IPM_QOS_SESSION_INFO, and IPM_QOS_THRESHOLD_DATA structures: Added QOSTYPE_RTCP_JB_HIGH, QOSTYPE_RTCP_JB_LOW, and QOSTYPE_RTCP_SCS values for eIPM_QOS_TYPE. IPM_SESSION_INFOEX structure: New for enhanced RTCP reporting.

Document No.	Publication Date	Description of Revisions
05-2257-014	December 2008	Added programming guide content (05-2330-007) to create a combined API Programming Guide and Library Reference document. Revisions to the 05-2330-007 and 05-2257-013 documents are listed below. Reserving Resources for Audio Coders chapter: New. Using the AMR-NB Audio Coder chapter: New. Using AMR-NB and G.711 Audio Over Nb UP chapter: New. H.263 Using RFC 2429 (RFC 4629) Packetization chapter: New. Using Enhanced Variable Rate Codecs (EVRC) chapter: New. DTMF Handling chapter: Added Setting Receive-only RFC 2833 Mode. Implementing Native T.38 Fax Hairpinning chapter: New. Using the Selective Packet Filtration Method chapter: New. Building Applications chapter: Added note about compiling Linux applications in Required Libraries. ipm_GenerateIFrame( ) function: Updated structure name to IPM_IFRAME_INFO (from IFRAME_INFO). ipm_GetSessionInfo( ) function: Updated description regarding QoS and RTCP statistics. Events chapter: Updated description for IPMEV_GET_SESSION_INFO event. IPM_AUDIO_CODER_INFO structure: Added CODER_TYPE_UDPTL_NATIVE. IPM_IFRAME_INFO structure: Updated structure name and inline function to IPM_IFRAME_INFO (from IFRAME_INFO). IPM_NBUP_PROFILE_INFO structure: Added AMR-NB and G.711 values to eProfileType. IPM_NBUP_SUBFLOW_INFO structure: Added AMR-NB and G.711 values to eFlowSize. IPM_PARM_INFO structure: Added PARMBD_RTP_SOURCE_FILTER. IPM_SESSION_INFO structure: Updated description regarding QoS and RTCP statistics. IPM_VIDEO_CODER_INFO structure: Updated eCoderType values (changed VIDEO_CODING_xxx to CODER_TYPE_xxx); added CODER_TYPE_H263_1998; removed VIDEO_CODING DEFAULT; removed note.

Document No.	Publication Date	Description of Revisions
05-2257-013	August 2008	Function Summary by Category chapter: Added Dialogic® Multimedia Kit for PCIe to Dialogic® IP Media Library API Function Support by Platform table. Function Information chapter: For supported functions, added Dialogic® Multimedia Kit for PCIe to Platform line. ipm_ReceiveDigits( ) function: Removed Dialogic® Multimedia Platform for AdvancedTCA from Platform line (not supported). ipm_StartMedia( ) function: Updated pMediaInfo parameter description (application cannot define local IP address, local RTP/RTCP port, local T.38 port). Removed first caution about calling ipm_Listen( ) after receiving ipm_StartMedia( ) completion event (not applicable). IPM_AUDIO_CODER_INFO data structure: For unFramesPerPkt field, added note that this field controls the RTP packet type for the EVRC codec. For unCoderPayloadType field, clarified description and revised to indicate that the default value of 0 specifies G.711 mu-law. [IPY00044398] Updated coder information in Supported Audio Coder Properties table: for AMR-NB codec, 1 to 10 fpp is supported for Dialogic® HMP Software (previously only 1 fpp was supported); for AMR-NB codec, replaced lowercase ‘k’ with uppercase K for consistency (both are supported); for QCELP, only 1 fpp is supported (removed 2 and 3); for G.726, removed limitation on 16, 24, 40 kbps for Dialogic® Multimedia Platform for AdvancedTCA or MMP (all are supported); for G.723, frame size is 20 for all platforms (removed 10 for MMP). IPM_PARM_INFO data structure: Updated table of parameters; ordered alphabetically; indicated platform support variation for parameters as appropriate. IPM_VIDEO_CODER_INFO_EX data structure: For unBitRate field, revised information when video transcoding is enabled. For eFramesPerSec field, removed frame rate of 25 and 29.97 fps as they are not supported. [IPY00043524]
05-2257-012	December 2007	ipm_GenerateIFrame( ) function: New, added for video fast update. Events chapter: Added IPMEV_GENERATEIFRAME and IPMEV_GENERATEIFRAME_FAIL for video fast update support. IPM_IFRAME_INFO data structure: New, added for video fast update. Error Codes: Added EIPM_ GENERATEIFRAME _INCAPABLE.

Document No.	Publication Date	Description of Revisions
05-2257-011	October 2007	ipm_DisableEvents( ) function: Added EVT_TELEPHONY value for RFC 2833. Deprecated EVT_RFC2833. ipm_EnableEvents( ) function: Added EVT_TELEPHONY value for RFC 2833. Deprecated EVT_RFC2833. ipm_InitResponseSend( ) function: Updated example code for Nb UP. ipm_InitSend( ) function: Updated example code for Nb UP. ipm_StartMedia( ): Updated example code, remote video and local video coder sections. Events chapter: Added IPMEV_TELEPHONY_EVENT for RFC 2833. IPM_AUDIO_CODER_INFO data structure: For Dialogic® Multimedia Platform for AdvancedTCA (MMP), corrected coder information in table of supported audio coder properties: for AMR-NB, fpp is 1 (removed 2,3,4); for G.726, only 32 kbps is supported with frame sizes 10 and 20. IPM_MEDIA data structure: Added NBUPProfileInfo field (previously missing). Added Nb UP values for eMediaType field (previously missing): MEDIATYPE_NBUP_REMOTE_RTP_INFO, MEDIATYPE_NBUP_LOCAL_RTP_INFO, and MEDIATYPE_NBUP_PROFILE_INFO. IPM_NBUP_RFCI_INFO data structure: Updated description of ucID field and added valid values. IPM_TELEPHONY_EVENT_INFO data structure: New, added for RFC 2833. Deprecated IPM_RFC2833_SIGNALID_INFO structure. IPM_TELEPHONY_INFO data structure: New, added for RFC 2833. Deprecated IPM_RFC2833_SIGNALID_INFO structure. IPM_VIDEO_CODER_INFO data structure: Removed unFrameRate and unSamplingRate fields as these are not supported. Added pExtraCoderInfo field. Added values to eCoderType field and noted that CODER_TYPE_H263 is deprecated. IPM_VIDEO_CODER_INFO_EX data structure: New, to support additional video coder properties for MPEG-4 and H.263.
05-2257-010	August 2007	Made global changes to reflect Dialogic brand. As part of rebranding, renamed document from "IP Media Library API for HMP Library Reference" to "Dialogic® IP Media Library API Library Reference". IPM_AUDIO_CODER_INFO data structure: Updated to include CODER_TYPE_EVRC; removed CODER_TYPE_EVRC_4_8k, CODER_TYPE_EVRC_4_8k_NATIVE, CODER_TYPE_EVRC_9_6k, CODER_TYPE_EVRC_4_8k_NATIVE. Also removed CODER_TYPE_QCELP_8K_NATIVE and CODER_TYPE_QCELP_13K_NATIVE. Updated table of supported audio coder properties for AMR, EVRC, QCELP. IPM_AUDIO_CODER_OPTIONS_INFO data structure: Updated unCoderOptions (added value for QCELP); updated unParm1 and nValue1 (removed CODER_OPT_CDMA_MAX_RATE for EVRC and QCELP); updated unParm2 and unValue2; updated example code.

Document No.	Publication Date	Description of Revisions
05-2257-009	May 2007	Function Summary by Category chapter: Added ipm_InitResponseSend( ) and ipm_InitSend( ) to the Media Session Functions category. Added Dialogic® IP Media Library API Function Support by Platform section. Function Information chapter: Added the following new functions for Nb UP support: ipm_InitResponseSend( ) and ipm_InitSend( ). ipm_GetCapabilities function: Corrected example code. ipm_GetLocalMediaInfo( ) function: Added statement about querying RTP/RTCP information. Added caution for Dialogic® Multimedia Platform for AdvancedTCA (Multimedia Platform for AdvancedTCA). ipm_SetRemoteMediaInfo( ) function: Removed from this document. Deprecated function. Replaced by ipm_StartMedia( ). ipm_StartMedia( ) function: Added caution for Multimedia Platform for AdvancedTCA. Updated example code to illustrate video. Events chapter: Added the following events: IPMEV_INIT_COMPLETE, IPMEV_INIT_FAILED, IPMEV_INIT_RESPONSE_SEND, IPMEV_INIT_SEND, IPMEV_INIT_RECEIVED, IPMEV_NACK_SENT. Data Structures chapter: Added the following new structures for Nb UP support: IPM_INIT_FAILURE, IPM_INIT_RECEIVED, IPM_INIT_RESPONSE, IPM_INIT_SEND, IPM_NACK_SENT, IPM_NBUP_INIT_FAILURE, IPM_NBUP_INIT_RECEIVED, IPM_NBUP_INIT_RESPONSE, IPM_NBUP_INIT_SEND, IPM_NBUP_NACK_SENT, IPM_NBUP_PROFILE_INFO, IPM_NBUP_RFCI_INFO, and IPM_NBUP_SUBFLOW_INFO. CT_DEVINFO data structure: Added CT_DFHMPATCA value for ct_devfamily. IPM_AUDIO_CODER_INFO data structure: Added several new coder types, including AMR-NB, EVRC, QCELP, and all _NATIVE coder types. Updated table of supported audio coder properties. IPM_AUDIO_CODER_OPTIONS_INFO data structure: New. IPM_MEDIA data structure: Added AudioCoderOptionsInfo field. Added eMediatype values, including MEDIATYPE_AUDIO_LOCAL_CODER_OPTIONS_INFO and MEDIATYPE_AUDIO_REMOTE_CODER_OPTIONS_INFO. Corrected MEDIATYPE_FAX_SIGNAL value (previously listed as MEDIATYPE_FAX_SIGNAL_INFO). IPM_PARM_INFO data structure: Added eIP_PARM defines, including PARMCH_LATENCYALG_AUDIO, PARMCH_LATENCYFRAMEINIT_AUDIO, and PARMCH_LATENCYFRAMEMAX_AUDIO. IPM_VIDEO_CODER_INFO data structure: Removed unProfileLevelID and szVisualConfiguration fields because they are not yet supported. Error Codes chapter: Updated for ipm_InitResponseSend( ) and ipm_InitSend( ).
05-2257-008	August 2006	ipm_GetCapabilities( ) function: Added.

Document No.	Publication Date	Description of Revisions
05-2257-007	June 2006	Media Session Functions section: Added ipm_SecurityGenMasterKeys( ) function. ipm_GetLocalMediaInfo( ) function: Updated description to clarify eMediaType and unCount as members of data structures referenced by the pMediaInfo function parameter and enumerate the allowed values. Also updated for video support. ipm_SecurityGenMasterKeys( ) function: Added for SRTP support. Events chapter: Updated the IPMEV_QOS_ALARM event description. Events chapter: Added IPMEV_SEC_NOTIFY_EXPIRE_KEY_AUDIO and IPMEV_SEC_NOTIFY_EXPIRE_KEY_VIDEO event descriptions. Data Structures chapter: Added SRTP and video related structures. IPM_AUDIO_CODER_INFO data structure: Added to replace IPM_CODER_INFO. IPM_PARM_INFO data structure: In the PARMCH_ECNLP_ACTIVE parameter description in the table, updated the definition of “NLP”. IPM_QOS_ALARM_DATA data structure: Updated to include SRTP and SRTCP alarms. IPM_QOS_THRESHOLD_DATA data structure: Updated to include SRTP and SRTCP alarms. IPM_MEDIA data structure: Updated for SRTP and video support. IPM_SECURITY_BASE64_KEY data structure: Added for SRTP support. IPM_SECURITY_BINARY_KEY data structure: Added for SRTP support. IPM_SECURITY_INFO data structure: Added for SRTP support. IPM_SECURITY_KEY data structure: Added for SRTP support. IPM_SRTP_PARMS data structure: Added for SRTP support. IPM_VIDEO_CODER_INFO data structure: Added for video support
05-2257-006	December 2005	Media Session Functions section: Added ipm_ModifyMedia( ) function Events section: Added IPMEV_MODIFYMEDIA and IPMEV_MODIFYMEDIA_FAIL ipm_GetLocalMediaInfo( ) function: Removed references to video ipm_ModifyMedia( ) function: Added to support endpoint configuration for half- and full-duplex media streams ipm_SetRemoteMediaInfo( ): Removed detailed information since the function is deprecated ipm_StartMedia( ) function: Added the DATA_IP_INACTIVE option to the eDirection parameter. Rephrased the first caution for greater clarity. IPM_AUDIO_CODER_INFO data structure: Removed from this version IPM_CODER_INFO data structure: Updated the supported coders table; removed the original note and added a note about G.726 usage. IPM_MEDIA data structure: Removed references to video; updated names of elements for audio coders IPM_PARM_INFO data structure: Added the following parameters: PARMBD_RTCPAUDIO_INTERVAL PARMBD_RTPAUDIO_PORT_BASE PARMCH_ECACTIVE PARMCH_ECHOTAIL PARMCH_ECNLP_ACTIVE IPM_VIDEO_CODER_INFO data structure: Removed from this version General: Hid all references to deprecated ipm_SetRemoteMediaInfo( ) function except the function reference page itself.

Document No.	Publication Date	Description of Revisions
05-2257-005	August 2005	ipm_GetLocalMediaInfo( ) function: Updated code example for video media info IPM_AUDIO_CODER_INFO data a structure: Added note about name change; added note to unCoderPayloadType description (PTR#33921) IPM_MEDIA data structure: Added union element for video coders; changed name of element for audio coders IPM_PARM_INFO data structure: Added info on using PARMCH_TOS for DSCP field IPM_RFC2833_SIGNALID_INFO data structure: Restored to document (previous removal was in error) IPM_VIDEO_CODER_INFO: New data structure
05-2257-004	April 2005	Function Summary by Category chapter: Removed unimplemented ipm_SendRFC2388SignalIDToIP function ipm_DisableEvents( ) function page: Removed unsupported EVT_FAXTONE and EVT_T38CALLSTATE event types ipm_EnableEvents( ) function page: Removed unsupported EVT_FAXTONE and EVT_T38CALLSTATE event types ipm_SendRFC2388SignalIDToIP function: Removed as unimplemented ipm_SetQoSThreshold( ) function page: Added Caution on possible failure scenario. Corrected code example Events chapter: Removed unimplemented IPMEV_RFC2833SIGNALRECEIVED event IPM_RFC2833_SIGNALID_INFO data structure section: Removed as unused IPM_QOS_SESSION_INFO data structure page: Clarified descriptions of jitter and lost packets QoS statistics
05-2257-003	September 2004	ipm_GetQoSAlarmStatus( ) function: Corrected code example ipm_GetSessionInfo( ) function: Added info on NULL pointer in asynch mode ipm_GetXmitSlot( ) function: Added info on NULL pointer in asynch mode ipm_Open( ) function: Removed caution that function must be called in synchronous mode ipm_ReceiveDigits( ) function: Clarified usage of data structure. Added caution regarding active RTP session requirement for receiving digits. ipm_SendDigits( ) function: Added info about maximum number of digits. Added caution regarding active RTP session requirement for sending digits. ipm_StartMedia( ) function: Added caution to avoid setting IP address 0.0.0.0 (PTR#32986). Corrected name of completion event. CT_DEVINFO data structure: Removed much information irrelevant to the structure's use with IPML on HMP IPM_DIGIT_INFO data structure: Added info about maximum number of digits and send vs. receive usage differences IPM_PARM_INFO data structure: Removed descriptions of three unsupported RFC2833 parameters IPM_PORT_INFO data structure: Added caution to avoid setting IP address 0.0.0.0 (PTR#32986)

Document No.	Publication Date	Description of Revisions
05-2257-002	April 2004	Made document HMP-specific by removing hardware-specific information, including "IP Media Function Support by Platform" section. ipm_DisableEvents( ) function: Removed two unsupported events. Added EVT_RTCPTIMEOUT and EVT_RTPTIMEOUT QoS alarm events. ipm_EnableEvents( ) function: Removed two unsupported events. Added EVT_RTCPTIMEOUT and EVT_RTPTIMEOUT QoS alarm events. ipm_GetLocalMediaInfo( ) function: Corrected MEDIATYPE... types referred to in the Cautions and Code Example sections ipm_Ping( ): Removed as unsupported ipm_SendDigits( ) function: Documented as supported for HMP ipm_SetQoSThreshold( ) function: Revised code example to use correct minimum value (100) for unTimeInterval ipm_SetRemoteMediaInfo( ) function: Documented function as deprecated. Corrected code example. ipm_StartMedia( ) function: Corrected code example ipm_Stop( ) function: Removed two unsupported eIPM_STOP_OPERATION values IPM_CODER_INFO data structure: Removed hardware-specific enum values and tables of supported coders. Updated table of supported coders. IPM_PARM_INFO data structure: Added PARMCH_RX_ADJVOLUME and PARMCH_RX_ADJVOLUME parameters. Expanded and corrected descriptions of all parameters. Removed hardware-specific enum values. IPM_PING_INFO and IPM_PINGPARM structures: Removed as unsupported IPM_QOS_ALARM_DATA data structure: Corrected names of eIPM_QOS_TYPE enums. Added new enums for RTCP Timeout and RTP Timeout alarms. IPM_QOS_SESSION_INFO data structure: Corrected names of eIPM_QOS_TYPE enums. Added new enums for RTCP Timeout and RTP Timeout alarms. IPM_QOS_THRESHOLD_DATA data structure: Corrected names of eIPM_QOS_TYPE enums. Added new enums for RTCP Timeout and RTP Timeout alarms. Added value ranges for unFaultThreshold. Updated descriptions of all fields. Added table of default parameter values. Error Codes chapter: added EIPM_RESOURCEINUSE
05-2257-001	September 2003	Initial version of HMP-specific document. Much of the information contained in this document was previously published in the *IP Media Library API for Linux and Windows Operating Systems Library Reference*, document number 05-1833-002. Among other small changes, the following changes were made: ipm_GetCTInfo( ): New function CT_DEVINFO: Added to book IPM_PARM_INFO data structure: Added valid value and variable type info for PARMCH_RFC2833EVT_TX_PLT and PARMCH_RFC2833EVT_RX_PLT. Deleted PARMCH_RFC2833MUTE_AUDIO, PARMCH_RFC2833TONE_TX_PLT, and PARMCH_RFC2833TONE_RX_PLT

About This Publication

The following topics provide information about this publication:

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

Purpose

This publication describes the features of the Dialogic® IP Media Library API and provides programming guidelines for those who choose to develop applications using this API library. It also provides a reference to the functions, events, data structures, and error codes in the Dialogic® IP Media Library API.

This guide applies to the Dialogic® IP Media Library API that is provided with the Dialogic® Host Media Processing (HMP) Software, the Dialogic® Multimedia Platform for AdvancedTCA, and the Dialogic® Multimedia Kit for PCIe.

A separate version of the guide is provided for Dialogic® System Release software, as there are some functional differences in the IP Media Library API implementation.

Applicability

This document version (05-2257-016) is published for Dialogic® Host Media Processing (HMP) Software Release 4.1LIN.

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 guide is intended for software developers who will access the Dialogic® IP Media Library API. This may include 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

This publication assumes that you are familiar with the Linux or Windows® operating system and the C programming language.

The information in this guide is organized in two major parts:

- *Programming Guide* content, which describes the features of the Dialogic® IP Media Library API and provides feature implementation guidelines.
- *Library Reference* content, which provides reference information about functions, data structures, events, and error codes.

Related Information

See the following for more information:

- For Dialogic® product documentation, see *http://www.dialogic.com/manuals*
- For Dialogic technical support, see *http://www.dialogic.com/support*
- For Dialogic® product information, see *http://www.dialogic.com*

Product Description 1

This chapter provides an overview of the Dialogic® IP Media Library API software. Topics include:

1.1 Features

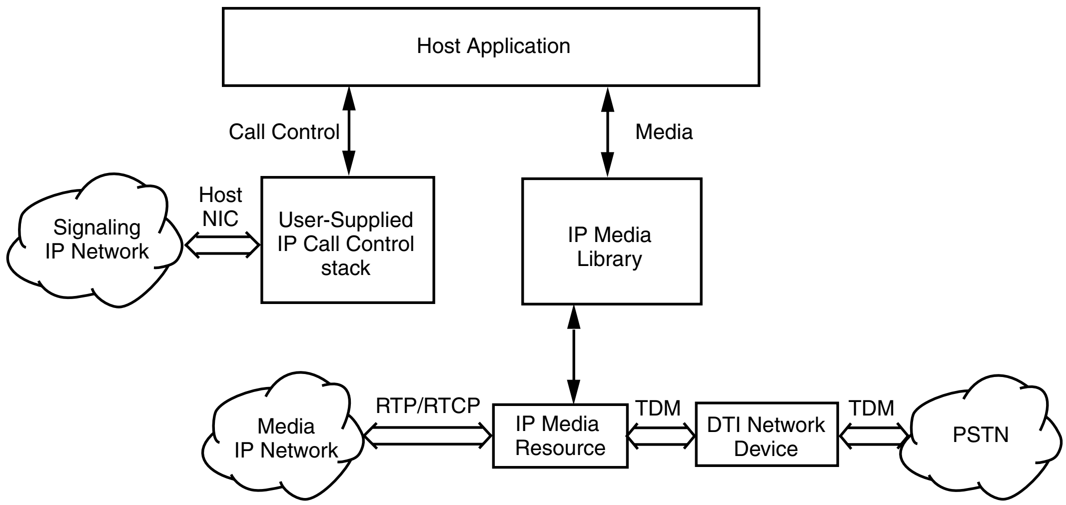
Features of the Dialogic® IP Media Library API software include:

- media resource management, such as open, close, and configure tasks
- media resource operations, such as start, stop, and detect digits
- Quality of Service (QoS) threshold alarm configuration and status reporting
- support of Dialogic® Standard Runtime Library API event management routines for error retrieval
- compatibility with Dialogic® Global Call API library or another call control stack to provide IP call control functionality

1.2 Architecture

Figure 1 shows the Dialogic® IP Media Library API architecture when using a user-supplied call control stack.

Figure 1. IP Media Architecture



1.3 Introduction to the Dialogic® IP Media Library API

The Dialogic® IP Media Library API provides an application programming interface to control the starting and stopping of RTP sessions, transmit and receive DTMF or signals, QoS alarms and their thresholds, and general-purpose device control functions. The library is only used to control media functions. It is not used to control the signaling stack. The application developer may choose to integrate any third party IP signaling stack (H.323, SIP, MGCP, etc.), or implement a proprietary signaling stack solution. The application developer uses the IP signaling stack to initiate or answer calls, and negotiate media characteristics such as coder, frames per packet, destination IP address, etc. Once media characteristics have been negotiated, the application uses Dialogic® IP Media Library API functions to start RTP streaming using the desired media characteristics.

1.4 Relationship with Dialogic® Global Call API Library

The Dialogic® Global Call API library provides a common call control interface that is independent of the underlying network interface technology. While the Dialogic® Global Call API library is primarily used for call establishment and teardown, it also provides capabilities to support applications that use IP technology, such as:

- call control capabilities for establishing calls over an IP network, via the RADVISION H.323 and SIP signaling stacks
- support for IP media control by providing the ability to open and close IP media channels for streaming, using the Dialogic® IP Media Library API software internally (under the hood)

Note: Applications should not mix Dialogic® Global Call API and Dialogic® IP Media Library API library usage of the same ipm_ devices.

Refer to the following Global Call manuals for more details:

- *Dialogic® Global Call IP Technology Guide*
- *Dialogic® Global Call API Programming Guide*
- *Dialogic® Global Call API Library Reference*

1.5 Dialogic® Standard Runtime Library API Support

The Dialogic® IP Media Library API performs event management using the Dialogic® Standard Runtime Library (SRL), which provides a set of common system functions that are applicable to all devices. Dialogic® SRL functions, parameters, and data structures are described in the *Dialogic® Standard Runtime Library API Library Reference*. Use the Dialogic® SRL functions to simplify application development by writing common event handlers to be used by all devices.

1.6 Media Channel Device Naming

To determine available resources, call **ipm_Open( )** on a board device, then call **ATDV_SUBDEVS( )** to get the available resources. (SRL operations are described in the *Dialogic® Standard Runtime Library API Library Reference*.)

To determine available resources in the Windows® environment, use the **sr_getboardcnt( )** function, which returns the number of boards of a particular type. (SRL operations are described in the *Dialogic® Standard Runtime Library API Library Reference*.)

Each IP media channel device follows the naming convention ipmBxCy; where:

- B is followed by the unique logical board number
- C is followed by the number of the media device channel

You may also use the **ipm_Open( )** function to open a board device, ipmBx, where B is followed by the unique logical board number.

Before you can use any of the other Dialogic® IP Media Library API functions on a device, that device must be opened. When the device is opened using **ipm_Open( )**, the function returns a unique device handle. The handle is the only way the device can be identified once it has been opened. The **ipm_Close( )** function closes a device.

Feature Support by Platform 2

This chapter describes high-level features and support for each feature by platform.

This document is used in more than one Dialogic® platform (software release). There may be information in this document that is not supported in a Dialogic® platform. Table 1 lists the high-level features of the Dialogic® IP Media Library API software documented in this current version and lists the Dialogic® platform (software release) that currently supports each feature. This table will be updated as a feature becomes available on a platform.

For function support information, see Section 23.5, "Dialogic® IP Media Library API Function Support by Platform", on page 122.

Note: Using the AMR-NB resource in connection with one or more Dialogic products does not grant the right to practice the AMR-NB standard. To seek a license patent agreement to practice the standard, contact the VoiceAge Corporation at www.voiceage.com/licensing/php.

Table 1. High-Level Feature Support by Platform

Feature	Dialogic® HMP Software 3.0WIN	Dialogic® HMP Software 3.1LIN	Dialogic® HMP Software 4.1LIN	MMP for ATCA	MMK for PCIe	Comments
Coder: AMR-NB audio	NS	S	S	S	S	
Coder: EVRC audio	NS	NS	NS	S	S	
Coder: H.263-1998 (H.263+) native hairpinning	S	S	S	S	S	
Coder: H.263-1998 (H.263+) video from RTSP server to 3G-324M	NS	S	S	S	S	Some restrictions apply. See Chapter 10, "H.263 Using RFC 2429 (RFC 4629) Packetization".
Coder: QCELP audio	NS	NS	NS	S	S	
DTMF handling - RFC 2833 receive-only mode	S	S	S	NS	NS	See Section 13.2.5, "Setting Receive-only RFC 2833 Mode", on page 65.
Echo cancellation	S	S	S	NS	NS	
I-Frame update	NS	NS	S	S	S	See **ipm_GenerateIFrame( )**.
NAT traversal	S	NS	NS	NS	NS	Network Access Translation (NAT).
Native RTP hairpinning	S	S	S	S	S	

Legend:
S = supported; NS = not supported
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
Dialogic® HMP Software 3.0WIN = Dialogic® Host Media Processing Software Release 3.0 for Windows
MMP for ATCA = Dialogic® Multimedia Platform for AdvancedTCA
MMK for PCIe = Dialogic® Multimedia Kit Software for PCIe

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

Feature	Dialogic® HMP Software 3.0WIN	Dialogic® HMP Software 3.1LIN	Dialogic® HMP Software 4.1LIN	MMP for ATCA	MMK for PCIe	Comments
Native RTP play and record	S	S	S	S	S	
Native T.38 hairpinning	S	NS	NS	NS	NS	
Nb UP protocol	NS	S	S	S	S	
Quality of service	S	S	S *	S *	S *	* On these platforms, QoS alarms and events are not supported for video streams.
RTCP reports (enhanced)	NS	S	S	NS	NS	Created using **ipm_GetSessionInfoEx( )**. See Section 17.8.2, "Enhanced RTCP Reports", on page 95.
Secure RTP	S	S	NS	NS	NS	
Selective packet filtration method	S	NS	NS	NS	NS	
Volume control	S	S	S	S	S	

Legend:
S = supported; NS = not supported
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
Dialogic® HMP Software 3.0WIN = Dialogic® Host Media Processing Software Release 3.0 for Windows
MMP for ATCA = Dialogic® Multimedia Platform for AdvancedTCA
MMK for PCIe = Dialogic® Multimedia Kit Software for PCIe

Programming Models 3

This chapter describes the programming models supported by the Dialogic® IP Media Library API software.

The *Dialogic® Standard Runtime Library API Programming Guide* describes different programming models which can be used by applications. The Dialogic® IP Media Library API supports all the programming models described therein.

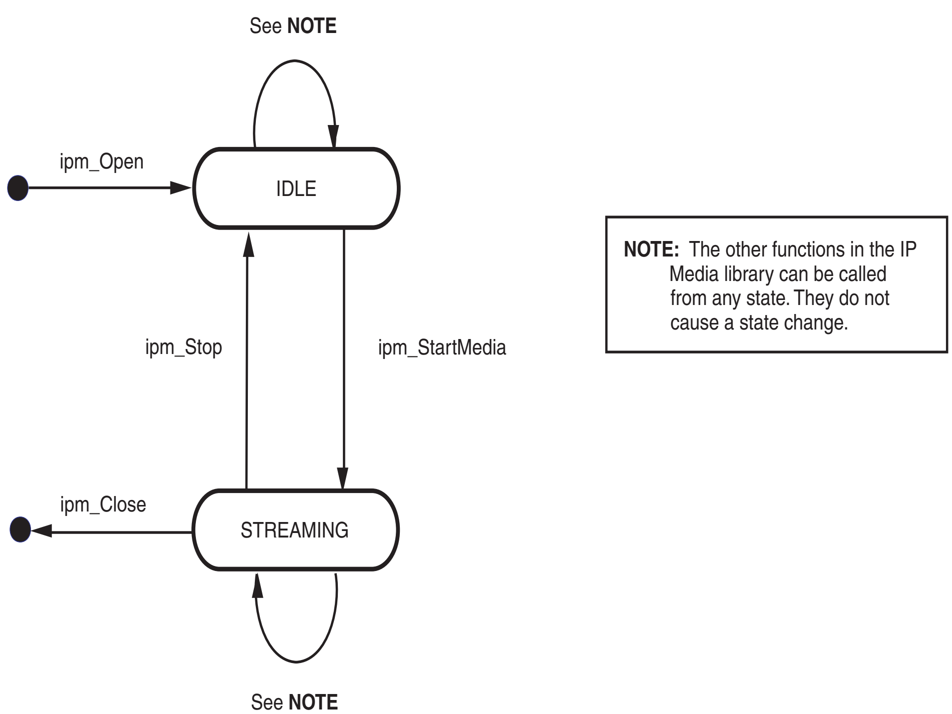
Note: The synchronous programming model is recommended for low-density systems only. For high-density systems, asynchronous programming models provide increased throughput for the application.

State Models 4

This chapter describes a simple Dialogic® IP Media Library API state-based application.

Figure 2 shows a simple application using two channel device states, IDLE and STREAMING.

Figure 2. IP Media Channel State Diagram



Event Handling 5

All IP media events are retrieved using the Dialogic® Standard Runtime Library (SRL) event retrieval mechanisms, including event handlers. The SRL is a device-independent library containing Event Management functions and Standard Attribute functions. This chapter lists SRL functions that are typically used by IP media-based applications.

5.1 Dialogic® Standard Runtime Library Event Management Functions

SRL Event Management functions retrieve and handle device termination events for certain library functions. Applications typically use the following functions:

sr_enbhdlr( )
enables event handler

sr_dishdlr( )
disables event handler

sr_getevtdev( )
gets device handle

sr_getevttype( )
gets event type

sr_waitevt( )
wait for next event

sr_waitevtEx( )
wait for events on certain devices

Note: See the *Dialogic® Standard Runtime Library API Library Reference* for function details.

5.2 Dialogic® Standard Runtime Library Standard Attribute Functions

SRL Standard Attribute functions return general device information, such as the device name or the last error that occurred on the device. Applications typically use the following functions:

ATDV_ERRMSGP( )
pointer to string describing the error that occurred during the last function call on the specified device

ATDV_LASTERR( )
error that occurred during the last function call on a specified device. See the function description for possible errors for the function.

ATDV_NAMEP( )
pointer to device name, for example, ipmBxCy

ATDV_SUBDEVS( )
number of subdevices

Note: See the *Dialogic® Standard Runtime Library API Library Reference* for function details.

Error Handling 6

This chapter describes error handling for the Dialogic® IP Media Library API software.

All Dialogic® IP Media Library API functions return a value that indicates the success or failure of the function call. Success is indicated by a return value of zero or a non-negative number. Failure is indicated by a value of -1.

If a function fails, call the Standard Attribute functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** for the reason for failure. These functions are described in the *Dialogic® Standard Runtime Library API Library Reference*.

If an error occurs during execution of an asynchronous function, the IPMEV_ERROR event is sent to the application. No change of state is triggered by this event. Upon receiving the IPMEV_ERROR event, the application can retrieve the reason for the failure using the standard runtime library functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**.

Reserving Resources for Audio Coders 7

This chapter describes how to reserve resources for audio coders. Topics include:

7.1 Feature Description

When working with audio coders, it is necessary to reserve resources for these coders prior to invoking the **ipm_StartMedia( )** function to enable Dialogic® software to use resources more efficiently. It is also good practice to release these resources after the call is completed.

Several scenarios are provided to explain how to reserve and release audio coder resources. The Dialogic® IP Media Library API (ipm_ API functions) and the Dialogic® Device Management API (dev_ API functions) are used in these scenarios.

Note: The Device Management API _Ex functions described in this chapter are not supported on Dialogic® Host Media Processing (HMP) Software Release 3.0 for Windows.

7.2 Reserve Resources for Outbound Call

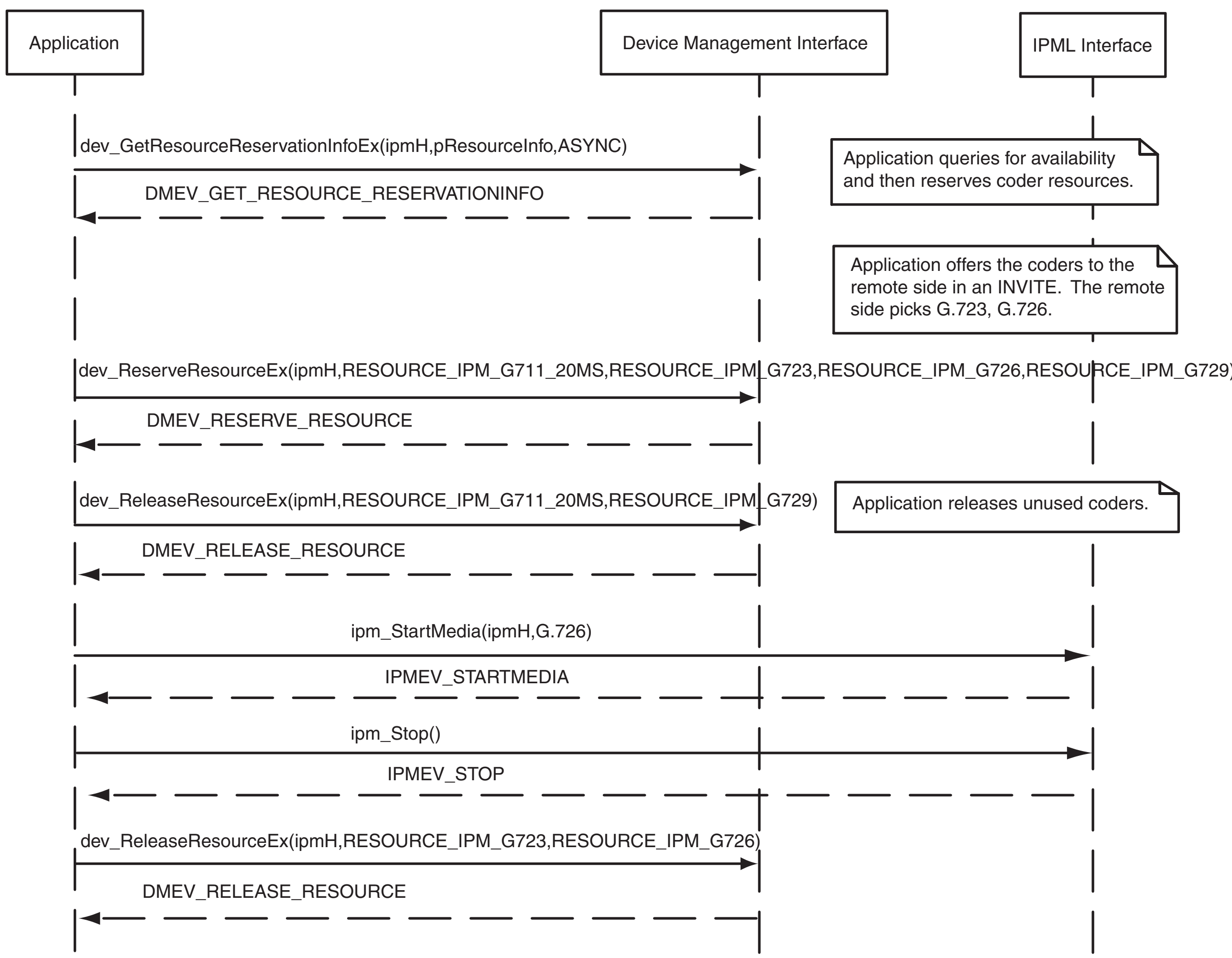
The following steps describe a sample scenario in which an application reserves resources for an outbound call:

1. Issue **dev_GetResourceReservationInfoEx( )** to query and get the current reservation information for the specified resources and device. The information is provided in the DEV_RESOURCE_RESERVATIONINFO_EX structure.
2. Issue **dev_ReserveResourceEx( )** to reserve coders; for example, RESOURCE_IPM_G711_20MS, RESOURCE_IPM_G723, RESOURCE_IPM_G726, and RESOURCE_IPM_G729.
3. Offer the reserved coders to the remote side.
4. The remote side selects G.723 and G.726.

5. Issue **dev_ReleaseResourceEx( )** to release unused resources: RESOURCE_IPM_G711_20MS and RESOURCE_IPM_G729. This leaves RESOURCE_IPM_G723 and RESOURCE_IPM_G726 as the reserved audio coders.
6. Issue **ipm_StartMedia( )** to start media and proceed with the call.
7. After the call is disconnected, issue **ipm_Stop( )**.
8. Issue **dev_ReserveResourceEx( )** to release the reserved coders RESOURCE_IPM_G723 and RESOURCE_IPM_G726. Alternatively, you can release all reserved coders by specifying RESOURCE_IPM_ALL_AUDIO_CODERS.

The following figure illustrates this scenario.

Figure 3. Reserve Resources for Outbound Call



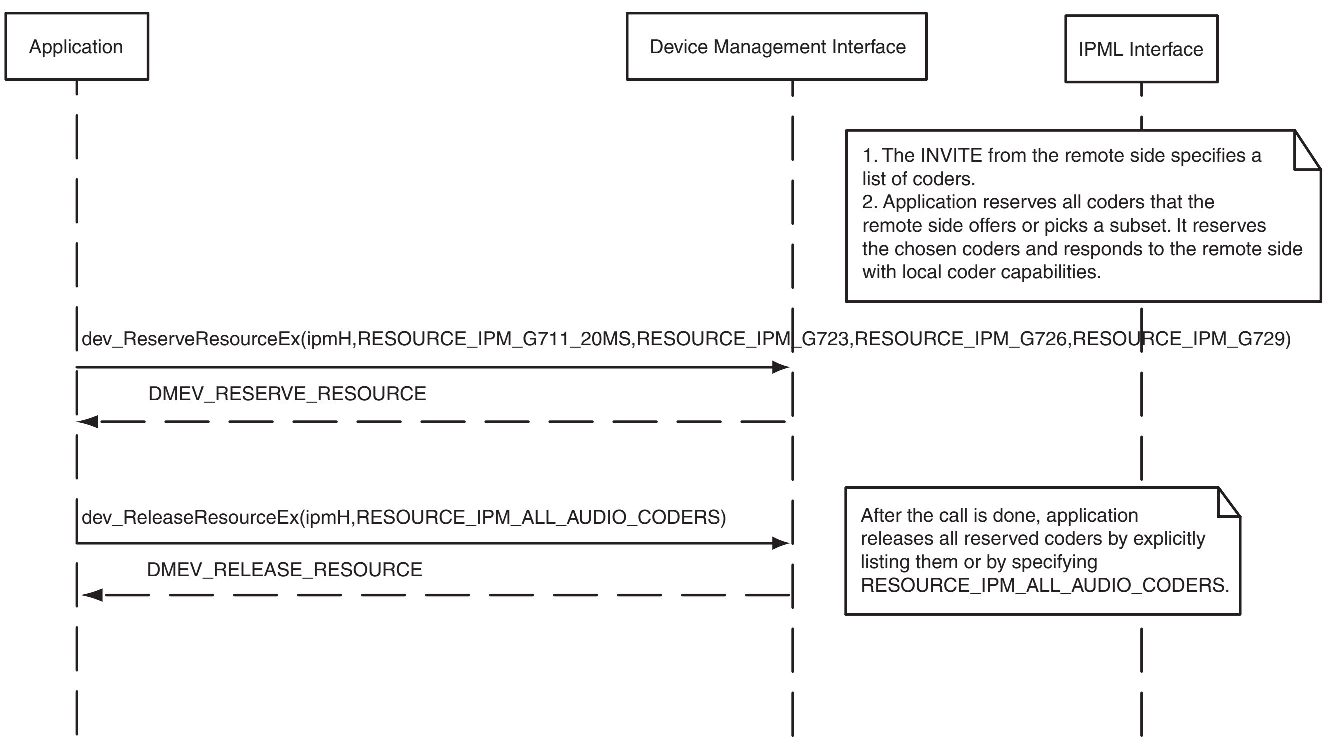
7.3 Reserve Resources for Inbound Call

The following steps describe a sample scenario in which an application reserves resources for an inbound call:

1. Receive an INVITE from the remote side that specifies a list of coders.
2. Reserve all the coders or a subset of the coders that the remote side offered using **dev_ReserveResourceEx( )**.
3. Offer the coders to the remote side.
4. Issue **ipm_StartMedia( )** to start media and proceed with the call.
5. After the call is disconnected, issue **ipm_Stop( )**.
6. Issue **dev_ReleaseResourceEx( )** to release all reserved coders by explicitly listing them or by specifying RESOURCE_IPM_ALL_AUDIO_CODERS.

The following figure illustrates this scenario.

Figure 4. Reserve Resources for Inbound Call



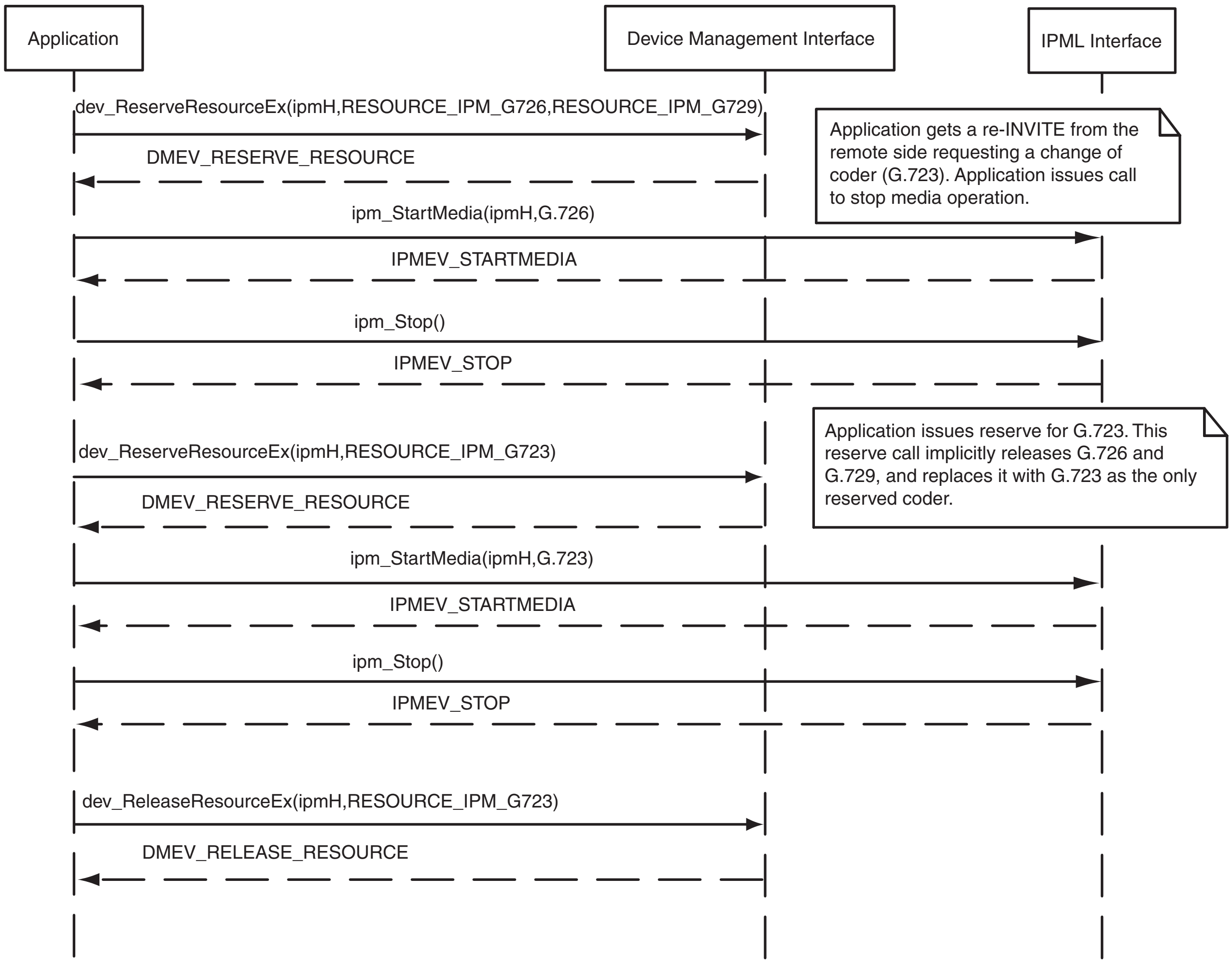
7.4 Release Resources Implicitly by a Subsequent Reservation Call

The following steps describe a sample scenario in which an application releases reserved resources implicitly by making another resource reservation call:

1. Reserve RESOURCE_IPM_G726 and RESOURCE_IPM_G729 resources using **dev_ReserveResourceEx( )**.
2. Use RESOURCE_IPM_G726 for a media operation via **ipm_StartMedia( )**.
3. Receive a re-INVITE message from the remote side requesting a change of coder to G.723.
4. Issue **ipm_Stop( )** to stop the media operation.
5. Issue **dev_ReserveResourceEx( )** to reserve RESOURCE_IPM_G723. This call is successful. This call implicitly releases RESOURCE_IPM_G726 and RESOURCE_IPM_G729, and replaces these coders with RESOURCE_IPM_G723 as the only reserved coder.
6. Issue **ipm_StartMedia( )** to start media and proceed with the call using RESOURCE_IPM_G723.
7. After the call is disconnected, issue **ipm_Stop( )**.
8. Issue **dev_ReleaseResourceEx( )** to release RESOURCE_IPM_G723.

The following figure illustrates this scenario.

Figure 5. Reserve Resources Implicitly



7.5 Handling a Resource Reservation Failure

The following steps describe a sample scenario in which an application handles a resource reservation failure:

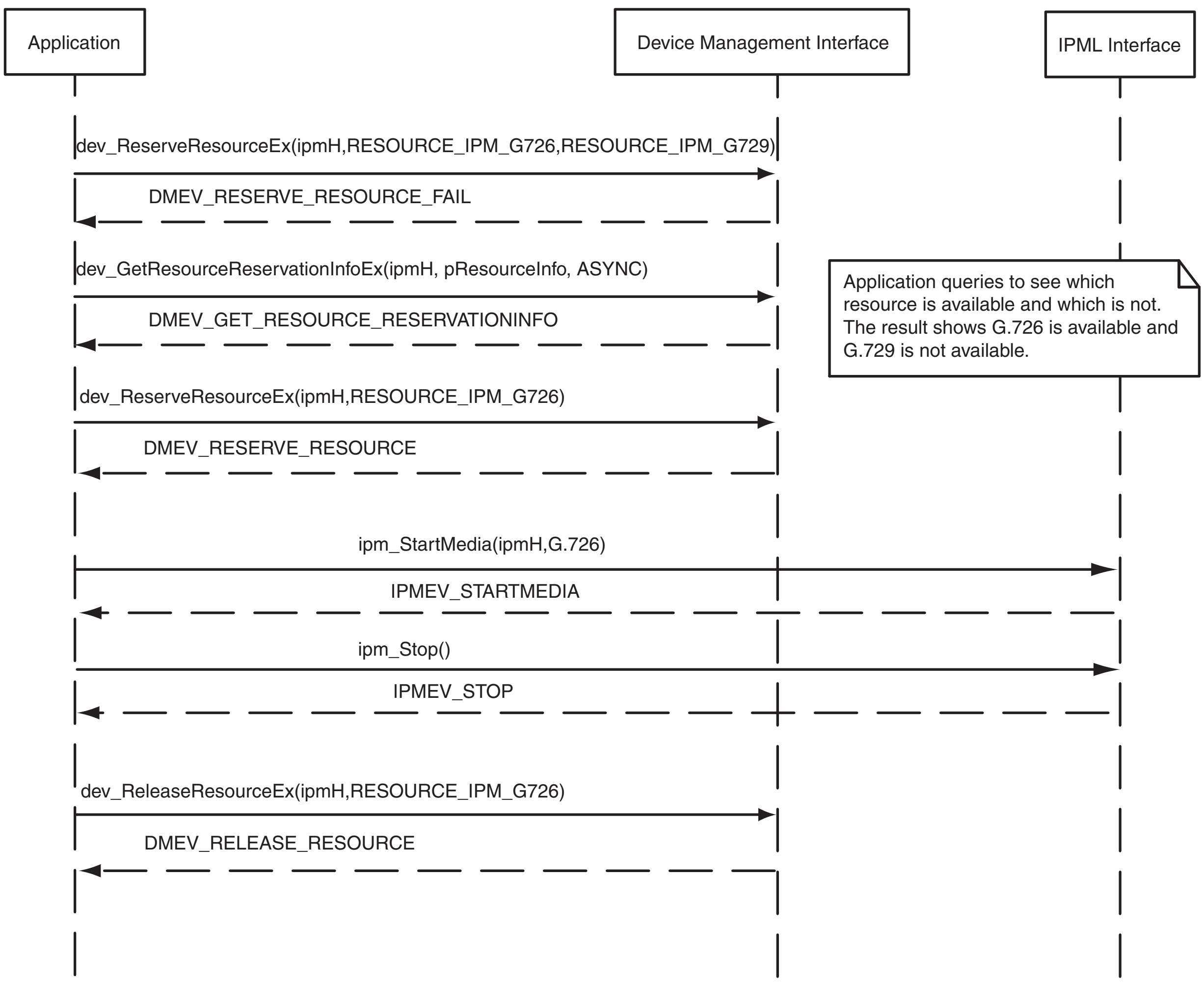
1. The **dev_ReserveResourceEx( )** call to reserve RESOURCE_IPM_G726 and RESOURCE_IPM_G729 resources fails for lack of available resources.

   Note: The reservation call fails when one or more of the resources requested is not available. You can query to check on resource availability prior to issuing a reservation request. Otherwise, you will need to query after the operation fails prior to re-issuing a reservation request.

2. Issue **dev_GetResourceReservationInfoEx( )** to check on resource availability. In this example, RESOURCE_IPM_G726 is available.
3. Issue **dev_ReserveResourceEx( )** to reserve RESOURCE_IPM_G726.
4. Issue **ipm_StartMedia( )** to start media and proceed with the call using RESOURCE_IPM_G726.
5. After the call is disconnected, issue **ipm_Stop( )**.
6. Issue **dev_ReleaseResourceEx( )** to release RESOURCE_IPM_G726.

The following figure illustrates this scenario.

Figure 6. Reserve Resources Failure



7.6 Reservation State after a Subsequent Call to Resource Reservation Fails

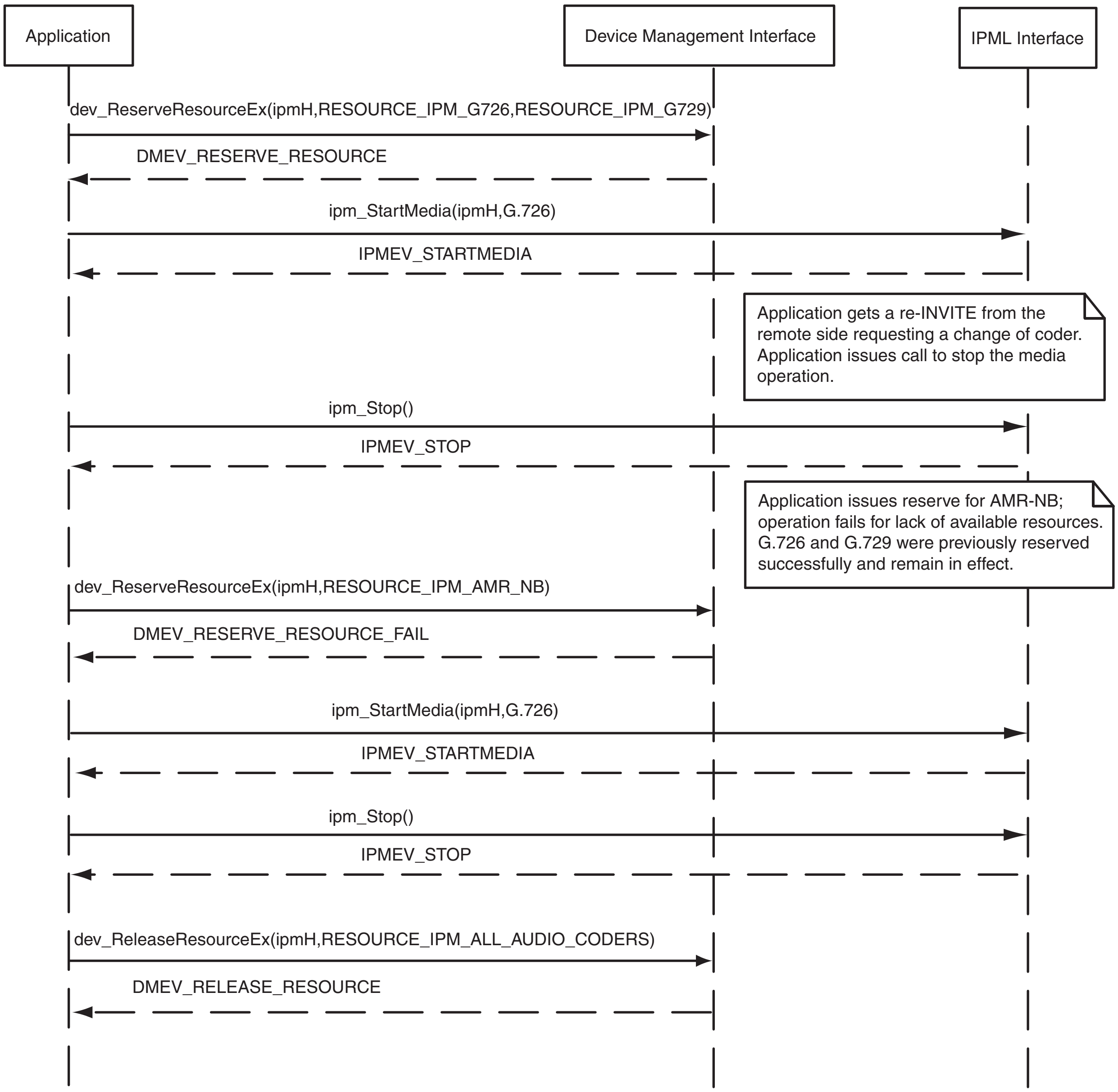
This sample scenario describes the reservation state when an initial resource reservation call succeeds, and a subsequent resource reservation call fails:

1. Issue **dev_ReserveResourceEx( )** to reserve RESOURCE_IPM_G726 and RESOURCE_IPM_G729 resources.
2. Issue **ipm_StartMedia( )** to start media operation using G.726.
3. Receive a re-INVITE request from the remote side to use the AMR-NB coder.
4. Issue **ipm_Stop( )** to stop media operation.
5. Issue **dev_ReserveResourceEx( )** to reserve RESOURCE_IPM_AMR_NB. However, the operation fails for lack of available resources. Because this operation failed, the previous successful resource reservation is preserved and remains in effect. In this case, RESOURCE_IPM_G726 and RESOURCE_IPM_G729 are still reserved.

   Note: Using the AMR-NB resource in connection with one or more Dialogic products does not grant the right to practice the AMR-NB standard. To seek a license patent agreement to practice the standard, contact the VoiceAge Corporation at www.voiceage.com/licensing/php.
6. Reject the re-INVITE from the remote side and continue to use any of the reserved coders.
7. Issue **ipm_StartMedia( )** to start media operation using G.726 or G.729.
8. After the call is disconnected, issue **ipm_Stop( )** to stop media operation.
9. Issue **dev_ReleaseResourceEx( )** to release RESOURCE_IPM_G726 and RESOURCE_IPM_G729. Alternatively, use RESOURCE_IPM_ALL_AUDIO_CODERS to release all the reserved audio coders.

The following figure illustrates this scenario.

Figure 7. Reservation State after a Subsequent Call Fails



Using the AMR-NB Audio Coder 8

This chapter describes the AMR Narrow Band coder and how to implement this coder in an application. Topics include:

8.1 Feature Description

The AMR-NB audio coder is not supported on all releases. For support information, see Chapter 2, “Feature Support by Platform”.

Note: Using the AMR-NB resource in connection with one or more Dialogic products does not grant the right to practice the AMR-NB standard. To seek a license patent agreement to practice the standard, contact the VoiceAge Corporation at www.voiceage.com/licensing/php.

AMR is an adaptive multi-rate speech codec. During operation, both local and remote sides can request a change in the bit rate and dynamically adjust the bandwidth. The protocol uses the following:

Frame Type (FT)
: to indicate the transmitted bit rate

Codec Mode Request (CMR) value
: to request a particular bit rate in every packet

To control the bit rate, AMR assumes that all connections are bi-directional.

This feature is specific to AMR-NB. It excludes support for AMR-WB and AMR-WB+, which are wideband audio, and some other formats not addressed by AMR-NB.

The AMR-NB codec supports the following bit rates:

- 12.2 kbps (GSM EFR)
- 10.2 kbps
- 7.95 kbps
- 7.40 kbps (IS-641)
- 6.70 kbps (PDC-EFR)
- 5.90 kbps
- 5.15 kbps
- 4.75 kbps

- 1.80 kbps (assuming SID frames are continuously transmitted)

  Note: The 1.80 kbit/s rate is not actually a voice signal, but the bit rate consumed when Voice Activation Detection (VAD) is processing a silence.

This feature is only supported when using the Session Initiation Protocol (SIP). None of the available SDP options are currently supported through Dialogic® Global Call software (that is, direct first-party call control). The options are only available using third-party call control (3PCC), where the application is responsible for interpreting received SDP text strings and for constructing all outbound SDP text strings.

8.2 API Library Support

The Dialogic® IP Media Library API allows the application to provide a preferred receive bit rate, a preferred transmit bit rate, and a rule to determine how changes in the received Codec Mode Request (CMR) value control the transmitted bit rate. This API library support is described in the following topics:

- CMR Value (Preferred Receive Bit Rate)
- Preferred Transmit Bit Rate
- CMR Rules
- RTP Payload Format

8.2.1 CMR Value (Preferred Receive Bit Rate)

The CMR value indicates a preferred receive bit rate on the local side. The CMR value is a preference and is communicated to the remote side as a request.

To specify the CMR value, set eMediaType in the IPM_MEDIA structure to MEDIATYPE_AUDIO_LOCAL_CODER_INFO and set eCoderType in the IPM_AUDIO_CODER_INFO structure to a value that corresponds to the preferred receive bit rate (CMR value). For example, to specify a CMR value of 5 which corresponds to a preferred receive bit rate of 7.95 kbit/s, set eCoderType to CODER_TYPE_AMRNB_7_95K.

8.2.2 Preferred Transmit Bit Rate

The preferred transmit bit rate indicates the following:

- The transmit bit rate to be used when a CMR value from the remote side has not been received yet
- The transmit bit rate to be used when the CMR value received from the remote side is 15 (don't care)
- The maximum transmit bit rate when the CMR rule is "CMR Limit" (see CMR Rules).

To specify the preferred transmit bit rate, set eMediaType in the IPM_MEDIA structure to MEDIATYPE_AUDIO_REMOTE_CODER_INFO and set eCoderType in the IPM_AUDIO_CODER_INFO structure to the desired transmit bit rate. For example, to specify a

Frame Type index of 4 which corresponds to a preferred transmit bit rate of 7.4 kbit/s, set eCoderType to CODER_TYPE_AMRNB_7_4K.

8.2.3 CMR Rules

The CMR rules indicate how the transmit bit rate is affected by CMR values received from the remote side. The rules are:

CMR Tracking
: Indicates that the transmit bit rate should follow the CMR value in the received packet.

CMR Limit
: Indicates that the transmit bit rate should follow the CMR value in the received packet as long as it doesn't exceed the preferred transmit bit rate setting.

To specify CMR rules, set eMediaType in the IPM_MEDIA structure to MEDIATYPE_AUDIO_REMOTE_CODER_OPTIONS_INFO and bitwise OR either CODER_OPT_AMR_CMR_TRACK or CODER_OPT_AMR_CMR_LIMIT in the AudioCoderOptionsInfo.unCoderOptions field. AudioCoderOptionsInfo field (of type IPM_AUDIO_CODER_OPTIONS_INFO) is a member of the IPM_MEDIA union.

Note: For AMR-NB, specifying a CMR rule is mandatory and the rules are mutually exclusive. With either rule, a preferred transmit bit rate must also be specified so that the software can transmit at the bit rate before the first packet is received or when a CMR value of 15 (don't care) is received from the remote side.

CMR rules are not applicable to the MEDIATYPE_AUDIO_LOCAL_CODER_OPTIONS_INFO eMediaType.

8.2.4 RTP Payload Format

AMR supports two different formats for the RTP payload:

Bandwidth efficient
: Minimizes the amount of network bandwidth.

Octet-aligned
: Makes the packet parsing easier for the AMR application.

To specify the RTP payload format, set eMediaType in the IPM_MEDIA structure to either MEDIATYPE_AUDIO_REMOTE_CODER_OPTIONS_INFO (for the transmit side) or MEDIATYPE_AUDIO_LOCAL_CODER_OPTIONS_INFO (for the receive side) and bitwise OR either CODER_OPT_AMR_EFFICIENT or CODER_OPT_AMR_OCTET in the AudioCoderOptionsInfo.unCoderOptions field. AudioCoderOptionsInfo (of type IPM_AUDIO_CODER_OPTIONS_INFO) is a member of the IPM_MEDIA union.

Note: For AMR-NB, specifying an RTP payload is mandatory and the formats are mutually exclusive.

8.3 Sample Scenarios

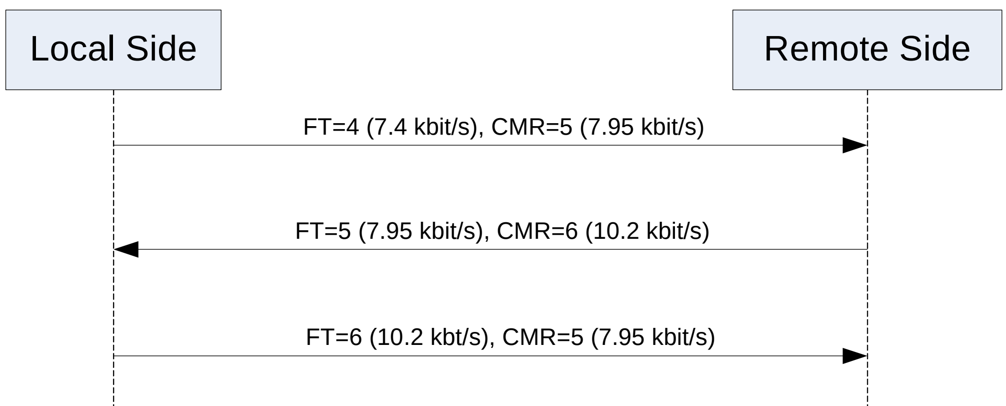
The following sections show sample scenarios for using AMR-NB and for setting CMR and FT values:

- Sample Scenario: Transmit at Bit Rate Requested by Remote Side
- Sample Scenario: Transmit at Bit Rate Less Than Preferred Value

Note: Using the AMR-NB resource in connection with one or more Dialogic products does not grant the right to practice the AMR-NB standard. To seek a license patent agreement to practice the standard, contact the VoiceAge Corporation at www.voiceage.com/licensing/php.

8.3.1 Sample Scenario: Transmit at Bit Rate Requested by Remote Side

In this example, the host application wants to transmit at the bit rate requested by the incoming CMR value. The following diagram depicts this use case, where the local side is the host application. FT refers to Frame Type and CMR refers to Codec Mode Request.



Note: While the diagram above implies an immediate reaction to a CMR from the other side, in reality, the other side's response to a CMR may take a few packets.

The sequence of activities is as follows:

1. The application on the local side:
   a. Sets its preferred bit rate via the remote audio coder settings, in this case 7.4 kbit/s.
   b. Sets its CMR value via the local audio coder settings, in this case 7.95 kbit/s.
   c. Sets the RTP payload format to bandwidth efficient and its CMR rule to "CMR Tracking" via the coder options settings.
   d. Starts RTP streaming using **ipm_StartMedia( )**.
2. The IPM device on the local side transmits at the preferred bit rate, in this case 7.4 kbit/s.
3. The IPM device on the local side receives its first packet from the remote side with a CMR value which is higher than its preferred bit rate. In this case the CMR value of 10.2 kbit/s is higher than the preferred bit rate of 7.4 kbit/s.

4. Since the CMR rule is set to “CMR Tracking”, the IPM device on the local side changes its transmitted bit rate to match the received CMR value, in this case 10.2 kbit/s.

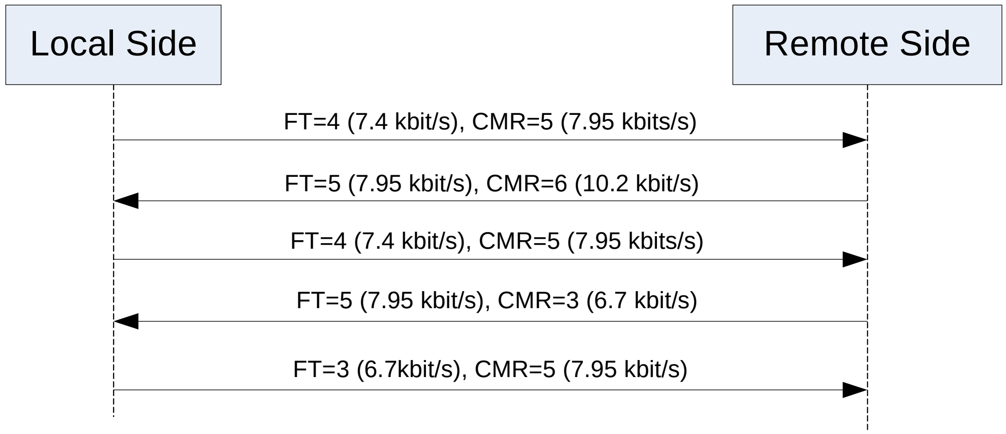
Example Code

The following code demonstrates the configuration required to handle this scenario.

```
...
/* Setup IP address here */

// Local Audio Coder
ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eCoderType = CODER_TYPE_AMRNB_7_95k;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eFrameSize = CODER_FRAMESIZE_20;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unFramesPerPkt = 1;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eVadEnable = CODER_VAD_ENABLE
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unCoderPayloadType = 96;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unRedPayloadType = 0
unCount++;

// Remote Audio Coder
ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eCoderType = CODER_TYPE_AMRNB_7_4k;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eFrameSize = CODER_FRAMESIZE_20;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unFramesPerPkt = 1;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eVadEnable = CODER_VAD_ENABLE
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unCoderPayloadType = 96;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unRedPayloadType = 0
unCount++;

ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_OPTIONS_INFO;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unVersion =
                                                       IPM_AUDIO_CODER_OPTIONS_INFO_VERSION;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unCoderOptions=
                                                       CODER_OPT_AMR_EFFICIENT;
unCount++;

ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_OPTIONS_INFO;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unVersion =
                                                       IPM_AUDIO_CODER_OPTIONS_INFO_VERSION;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unCoderOptions=
                                              CODER_OPT_AMR_CMR_TRACK | CODER_OPT_AMR_EFFICIENT;
unCount++;
ipmMediaInfo.unCount = unCount;
...
```

8.3.2 Sample Scenario: Transmit at Bit Rate Less Than Preferred Value

In this example, the host application wants to limit the transmitted bit rate to less than the preferred value. The following diagram shows this case, where the local side is the host application. FT refers to Frame Type and CMR refers to Codec Mode Request.



Note: While the diagram above implies an immediate reaction to a CMR from the other side, in reality, the other side's response to a CMR may take a few packets.

The sequence of activities is as follows:

1. The application on the local side:
   a. Sets its preferred bit rate via the remote audio coder settings, in this case 7.4 kbit/s.
   b. Sets its CMR value via the local audio coder settings, in this case 7.95 kbit/s.
   c. Sets the RTP payload format to bandwidth efficient and its CMR rule to "CMR Limit" via the coder options settings.
   d. Starts RTP streaming using **ipm_StartMedia( )**.
2. The IPM device on the local side transmits at the preferred bit rate, in this case 7.4 kbit/s.
3. The IPM device on the local side receives its first packet from the remote side with a CMR value which is higher than its preferred bit rate. In this case, the CMR value of 10.2 kbit/s is higher than the preferred bit rate of 7.4 kbit/s.
4. Since its CMR rule is set to "CMR Limit", the IPM device on the local side cannot exceed the preferred bit rate, so it leaves the transmitted bit rate at the preferred value, in this case 7.4 kbit/s.
5. The IPM device on the local side receives a packet from the remote side with a CMR value which is less than its preferred bit rate. In this case, the CMR value of 6.7 kbit/s is less than the preferred bit rate of 7.4 kbit/s.
6. Since the new received CMR value of 6.7 kbit/s does not exceed the limit (7.4 kbit/s), the IPM device on the local side changes its transmitted bit rate to match the received CMR value (6.7 kbit/s).

Example Code

The following code demonstrates the configuration required to handle this scenario.

```
...
/* Setup IP address here */

// Local Audio Coder
ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
```

```
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eCoderType =
CODER_TYPE_AMRNB_7_95k;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eFrameSize = CODER_FRAMESIZE_20;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unFramesPerPkt = 1;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eVadEnable = CODER_VAD_ENABLE
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unCoderPayloadType = 96;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unRedPayloadType = 0
unCount++;

// Remote Audio Coder
ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eCoderType = CODER_TYPE_AMRNB_7_4k;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eFrameSize = CODER_FRAMESIZE_20;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unFramesPerPkt = 1;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eVadEnable = CODER_VAD_ENABLE
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unCoderPayloadType = 96;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unRedPayloadType = 0
unCount++;

ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_OPTIONS_INFO;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo = {0};
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unVersion =
                                             IPM_AUDIO_CODER_OPTIONS_INFO_VERSION;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unCoderOptions=
                                             CODER_OPT_AMR_EFFICIENT;
unCount++;

ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_OPTIONS_INFO;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo = {0};
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unVersion =
                                             IPM_AUDIO_CODER_OPTIONS_INFO_VERSION;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unCoderOptions=
                                             CODER_OPT_AMR_CMR_LIMIT | CODER_OPT_AMR_EFFICIENT;
unCount++
ipmMediaInfo.unCount = unCount;
```

Using AMR-NB and G.711 Audio Over Nb UP 9

This chapter describes how to stream AMR-NB and G.711 audio over Nb UP. Topics include:

9.1 Feature Description

The AMR-NB and G.711 audio over Nb UP feature is not supported on all releases. For support information, see Chapter 2, "Feature Support by Platform".

Note: Using the AMR-NB resource in connection with one or more Dialogic products mentioned herein does not grant the right to practice the AMR-NB standard. To seek a patent license agreement to practice the standard, contact the VoiceAge Corporation at www.voiceage.com/licensing.php.

This feature allows an IP media streaming (IPM) device to stream AMR-NB audio (20 ms, 12.2 Kbps bit rate) and G.711 audio (5 ms and 20 ms, A-law and mu-law) over Nb UP. This data can be streamed to and from a 3G network. A multimedia (MM) device can be connected to the IPM device for native play and record operations. Transcoding is not currently supported.

Setting up AMR-NB over Nb UP or G.711 over Nb UP media sessions is similar to setting up a 3G-324M over Nb UP session. However, the 3G-324M (M3G) component is not used. The following sections provide guidelines for streaming AMR-NB audio or G.711 audio over Nb UP. For information on the Dialogic® 3G-324M API, see the *Dialogic® 3G-324M API Programming Guide and Library Reference*.

9.2 API Library Support

In the Dialogic® IP Media Library API, the following values in the eProfileType field, eIPM_NBUP_PROFILE_TYPE enumeration, of the IPM_NBUP_PROFILE_INFO structure are used for streaming AMR-NB or G.711 audio over Nb UP:

NBUP_PROFILE_AMR_NB
: AMR-NB type connection

NBUP_PROFILE_G711ALAW64K_5MS
: G.711 A-law 5 ms type connection

NBUP_PROFILE_G711ALAW64K_20MS
G.711 A-law 20 ms type connection

NBUP_PROFILE_G711ULAW64K_5MS
G.711 mu-law 5 ms type connection

NBUP_PROFILE_G711ULAW64K_20MS
G.711 mu-law 20 ms type connection

In the Dialogic® IP Media Library API, the following values in the eFlowSize field, eIPM_NBUP_FLOW_SIZE enumeration, of the IPM_NBUP_SUBFLOW_INFO structure are used for streaming AMR-NB or G.711 audio over Nb UP:

NBUP_FLOW_SIZE_39_BITS
use 39 bits (for AMR-NB)

NBUP_FLOW_SIZE_60_BITS
use 60 bits (for AMR-NB)

NBUP_FLOW_SIZE_81_BITS
use 81 bits (for AMR-NB)

NBUP_FLOW_SIZE_103_BITS
use 103 bits (for AMR-NB)

NBUP_FLOW_SIZE_1280_BITS
use 1280 bits (for G.711)

9.3 Guidelines for Streaming AMR-NB Over Nb UP

To stream AMR-NB audio (20 ms, 12.2 Kbps bit rate) over Nb UP, follow these general guidelines:

1. Specify a native connection between the IPM device and the MM device. Native play and record operation takes place using native AMR-NB 12.2 Kbps audio files.
2. Where available, use the inline function to initialize a data structure. For example, use the INIT_IPM_NBUP_PROFILE_INFO inline function to initialize IPM_NBUP_PROFILE_INFO, and the INIT_IPM_NBUP_INIT_SEND inline function to initialize IPM_NBUP_INIT_SEND.
3. Call **ipm_StartMedia( )** to start the session. In the IPM_NBUP_PROFILE_INFO structure, specify NBUP_PROFILE_AMR_NB in the eProfileType field.
4. Call **ipm_InitSend( )** to send an Nb UP initialization message to the remote endpoint. Construct the IPM_NBUP_INIT_SEND structure with a list of two IPM_NBUP_RFCI_INFO structures.

   The first IPM_NBUP_RFCI_INFO structure, which represents the first RFCI, should contain three IPM_NBUP_SUBFLOW_INFO structures. For the first IPM_NBUP_SUBFLOW_INFO structure, specify NBUP_FLOW_SIZE_81_BITS. For the second structure, specify NBUP_FLOW_SIZE_103_BITS. For the third structure, specify NBUP_FLOW_SIZE_60_BITS. This represents the RFCI used for the source rate of AMR-NB 12.2 Kbps.

   The second IPM_NBUP_RFCI_INFO structure, which represents the second RFCI, should contain three IPM_NBUP_SUBFLOW_INFO structures. For the first

IPM_NBUP_SUBFLOW_INFO structure, specify NBUP_FLOW_SIZE_39_BITS. For the second and third structures, specify NBUP_FLOW_SIZE_0_BITS. This represents the RFCI used for the source rate of AMR-NB SID.

The IPMEV_INIT_SEND termination event indicates successful completion of the **ipm_InitSend( )** function.

5. Upon receiving the IPMEV_INIT_RECEIVED event, the remote endpoint calls **ipm_InitResponseSend( )** to respond to the Nb UP message sent by the local endpoint.

   The IPMEV_INIT_RESPONSE_SEND termination event indicates successful completion of the **ipm_InitResponseSend( )** function.

6. Both the remote endpoint and the local endpoint should receive the IPMEV_INIT_COMPLETE unsolicited event, which indicates that the Nb UP session is successfully established. Upon receiving this event, the endpoints may exchange data, for example, using **mm_Play( )** and **mm_Record( )**.

9.4 Guidelines for Streaming G.711 (5 ms) over Nb UP

To stream G.711 (5 ms) audio over Nb UP, follow these general guidelines:

1. Specify a native connection between the IPM device and the MM device. Native play and record operation takes place using native G.711 audio files.

   Note: For play operation, native audio files containing G.711 20 ms format may be used. The Dialogic® product transforms this data to 5 ms format before transmitting to the IP network.

2. Where available, use the inline function to initialize a data structure. For example, use INIT_IPM_NBUP_PROFILE_INFO inline function to initialize IPM_NBUP_PROFILE_INFO, and INIT_IPM_NBUP_INIT_SEND inline function to initialize IPM_NBUP_INIT_SEND.

3. Call **ipm_StartMedia( )** to start the session. In the IPM_NBUP_PROFILE_INFO structure, specify NBUP_PROFILE_G711ALAW64K_5MS or NBUP_PROFILE_G711ULAW64K_5MS in the eProfileType field.

4. Call **ipm_InitSend( )** to send an Nb UP initialization message to the remote party. Construct the IPM_NBUP_INIT_SEND structure with a list of one IPM_NBUP_RFCI_INFO structure.

   The IPM_NBUP_RFCI_INFO structure should contain three IPM_NBUP_SUBFLOW_INFO structures. For the first IPM_NBUP_SUBFLOW_INFO structure, specify NBUP_FLOW_SIZE_320_BITS. For the second and third structures, specify NBUP_FLOW_SIZE_0_BITS. This represents the RFCI used for the source rate of G.711 5 ms.

   The IPMEV_INIT_SEND termination event indicates successful completion of the **ipm_InitSend( )** function.

5. Upon receiving the IPMEV_INIT_RECEIVED event, the remote endpoint calls **ipm_InitResponseSend( )** to respond to the Nb UP message sent by the local endpoint.

   The IPMEV_INIT_RESPONSE_SEND termination event indicates successful completion of the **ipm_InitResponseSend( )** function.

6. Both the remote endpoint and the local endpoint should receive the IPMEV_INIT_COMPLETE unsolicited event, which indicates that the Nb UP session is

successfully established. Upon receiving this event, the endpoints may exchange data, for example, using **mm_Play( )** and **mm_Record( )**.

9.5 Guidelines for Streaming G.711 (20 ms) over Nb UP

To stream G.711 (20 ms) audio over Nb UP, follow these general guidelines:

1. Specify a native connection between the IPM device and the MM device. Native play and record operation takes place using native G.711 audio files.
2. Where available, use the inline function to initialize a data structure. For example, use INIT_IPM_NBUP_PROFILE_INFO inline function to initialize IPM_NBUP_PROFILE_INFO, and INIT_IPM_NBUP_INIT_SEND inline function to initialize IPM_NBUP_INIT_SEND.
3. Call **ipm_StartMedia( )** to start the session. In the IPM_NBUP_PROFILE_INFO structure, specify NBUP_PROFILE_G711ALAW64K_20MS or NBUP_PROFILE_G711ULAW64K_20MS in the eProfileType field.
4. Call **ipm_InitSend( )** to send an Nb UP initialization message to the remote party. Construct the IPM_NBUP_INIT_SEND structure with a list of one IPM_NBUP_RFCI_INFO structure.

   The IPM_NBUP_RFCI_INFO structure should contain three IPM_NBUP_SUBFLOW_INFO structures. For the first IPM_NBUP_SUBFLOW_INFO structure, specify NBUP_FLOW_SIZE_1280_BITS. For the second and third structures, specify NBUP_FLOW_SIZE_0_BITS. This represents the RFCI used for the source rate of G.711 20 ms.

   The IPMEV_INIT_SEND termination event indicates successful completion of the **ipm_InitSend( )** function.
5. Upon receiving the IPMEV_INIT_RECEIVED event, the remote endpoint calls **ipm_InitResponseSend( )** to respond to the Nb UP message sent by the local endpoint.

   The IPMEV_INIT_RESPONSE_SEND termination event indicates successful completion of the **ipm_InitResponseSend( )** function.
6. Both the remote endpoint and the local endpoint should receive the IPMEV_INIT_COMPLETE unsolicited event, which indicates that the Nb UP session is successfully established. Upon receiving this event, the endpoints may exchange data, for example, using **mm_Play( )** and **mm_Record( )**.

H.263 Using RFC 2429 (RFC 4629) Packetization 10

This chapter describes the H.263 video coder using RFC 2429 (RFC 4629) packetization.

10.1 Feature Description

H.263 video coder using RFC 2429 (RFC 4629) packetization is not supported on all releases. For support information, see Chapter 2, “Feature Support by Platform”.

The Dialogic® IP Media Library API supports streaming H.263 Baseline Profile (Profile 0) Level 10 video using RFC 2429 packetization from an RTSP (Real-time Streaming Protocol) Server to 3G-324M calls. Note that this codec is also known as H.263-1998 or H.263+. Only Baseline Profile is supported. No H.263 annexes are supported.

This feature allows an application to natively stream audio and video from an RTSP Server to 3G-324M calls terminated by the Dialogic® product. These 3G-324M calls may be established over the PSTN (for example, E1 circuit-switched connection) or over IP.

For 3G-324M calls established over the PSTN, this feature uses DTI and M3G devices. For 3G-324M calls established over IP, this feature uses IPM and M3G devices. Note that the DTI device and the IPM device establish the transport to the remote endpoint. H.223 multiplexed multimedia data flows between these devices and the M3G device. The M3G device performs the multiplex/demultiplex operations and acts as the portal to the rest of the multimedia server.

The audio streaming formats supported for this feature include AMR-NB and G.723.

For information on the Dialogic® 3G-324M API, see the *Dialogic® 3G-324M API Programming Guide and Library Reference*.

10.2 API Library Support

In the Dialogic® IP Media Library API, the following value in the eCoderType field, eIPM_CODER_TYPE enumeration, of the IPM_VIDEO_CODER_INFO structure is supported:

CODER_TYPE_H263_1998
: H.263-1998 (also known as H.263+) video coder

10.3 Usage Guidelines

The following usage guidelines as well as restrictions and limitations are described for this feature:

- This feature is only supported for the 3PCC/SIP call model.
- Although the video format streamed from the RTSP Server to the product is H.263-1998 using RFC 2429 (RFC 4629) packetization, only Baseline H.263 is supported. Annexes are not supported. Video transcoding is not supported. Specify **native connection** (no transcoding) between devices in the DM_PORT_CONNECT_INFO structure of the Device Management API library.
- Audio transcoding is not supported. Therefore, the audio format selected for the 3G-324M call must be the same as the audio format selected for the RTSP Server. Specify **native connection** (no transcoding) between devices in the DM_PORT_CONNECT_INFO structure of the Device Management API library.
- The RTSP Server must be configured appropriately to interoperate with this feature. The characteristics of the video streamed from the RTSP Server must be compatible with 3G mobile devices and the transport; namely, set to QCIF picture format and constant bit rate (CBR) mode rather than variable bit rate (VBR) mode. Bit rates should not exceed 40 Kbps and frame rates should not exceed 15 fps. A nominal bit rate and frame rate to use is 37.8 Kbps at 7.5 fps.
- This feature only supports half-duplex streaming from the IPM device to the 3G-324M device.
- This feature does not support video play and record operations for IPM calls.
- When switching between video sources for transmission of video to the 3G remote terminal, such as switching between the RTSP Server and the MM device, you must send an I-frame as the first video frame streamed from the new source. Ensure that this takes place by starting a new play from the MM device when it becomes the new source or by starting a new play from the RTSP Server when it becomes the new source.

Using Enhanced Variable Rate Codecs (EVRC) 11

This chapter describes the enhanced variable rate codecs (EVRC) and how to implement this codec in an application.

11.1 Feature Description

EVRC is not supported on all releases. For support information, see Chapter 2, “Feature Support by Platform”.

The Dialogic® IP Media Library API supports enhanced variable rate codecs (EVRC) as specified in RFC 3558, RTP Payload Format for Enhanced Variable Rate Codecs (EVRC) and Selectable Mode Vocoders (SMV).

When using EVRCs, more than one codec data frame may be included in a single interleaved/bundled packet by a sender. This is accomplished by:

Bundling
: A technique used to spread the transmission overhead of the RTP and payload header over multiple vocoder frames.

Interleaving
: A technique used to reduce the listener’s perception of data loss by spreading such a loss over non-consecutive vocoder frames.

EVRC and similar vocoders can compensate for an occasional lost frame, but speech quality degrades exponentially with consecutive frame loss.

11.2 API Library Support

EVRC RTP packet format is controlled by specifying the number of frames per packet (fpp) in the unFramesPerPkt field of the IPM_AUDIO_CODER_INFO structure. Frames per packet values of 1 to 3 are allowed. The packet formats are:

Header-Free format
: Set frames per packet value to 1.

Interleaved/Bundled format
: Set frames per packet value to 2 or more.

See RFC 3558 for details of these formats.

Bundling is specified using the unFramesPerPkt field in the IPM_AUDIO_CODER_INFO structure, setting it to 2 or 3.

Interleaving is specified using the unParm1 field in the IPM_AUDIO_CODER_OPTIONS_INFO structure. Valid values are from 0 to 7; 0 indicates do not use interleaving. Interleaving should only be used when the frames per packet value is 2 or more.

For example code, see the IPM_AUDIO_CODER_OPTIONS_INFO structure.

Configuring for Half- or Full-Duplex Media Streams 12

This chapter describes the Dialogic® IP Media Library API capability for setting up and reconfiguring endpoints for half-duplex and full-duplex media streams.

12.1 Overview of Half- and Full-Duplex Stream Support

When using IP technology, the ability to start a stream session at an endpoint for half-duplex (or full-duplex) media streaming and subsequently update the stream session for full-duplex (or half-duplex) media streaming is a useful capability.

One example that demonstrates this is when providing support for a call control feature known as "Early Media". In IP technology, the establishment of RTP media streaming is normally one of the final steps in establishing and connecting a call. This is in contrast to the Public Switched Telephone Network (PSTN), where call progress signaling is commonly provided to the calling party via audible, in-band call progress tones, such as ringback, busy signal, and SIT tones. When implementing a VoIP gateway, it is often imperative to initiate media (RTP) streaming from the local endpoint to the calling party before the call is connected.

To achieve this functionality using the Dialogic® IP Media Library API, the calling party endpoint can be configured for half-duplex streaming (in the receive direction) prior to call connection to receive call progress signaling. The endpoint can then be reconfigured for full-duplex streaming when the call is connected.

Another useful application of this feature is in the implementation of H.450.4 call hold message flows.

Note: Configuring half-duplex streaming for T.38 fax does not apply, since T.38 fax is inherently full-duplex.

12.2 API Library Support

The **ipm_ModifyMedia( )** function is not supported on all releases. For function support information, see Section 23.5, "Dialogic® IP Media Library API Function Support by Platform", on page 122.

The following Dialogic® IP Media Library API functions are used to configure and reconfigure an endpoint for half-duplex or full-duplex media streaming:

ipm_StartMedia( )
: sets media properties and starts an RTP media streaming session

ipm_ModifyMedia( )
: modifies various properties of an active RTP media streaming session

Both functions contain an **eDirection** parameter that is used to specify if the media stream should be half-duplex or full-duplex. Possible values of the **eDirection** parameter are:

DATA_IP_RECEIVEONLY
: receive RTP and RTCP packets from the IP network, but do not send packets

DATA_IP_SENDONLY
: send RTP and RTCP packets to the IP network, but do not receive packets

DATA_IP_TDM_BIDIRECTIONAL
: full-duplex RTP and RTCP path between IP network and TDM

DATA_IP_INACTIVE
: allow RTCP while blocking RTP or T.38 packets

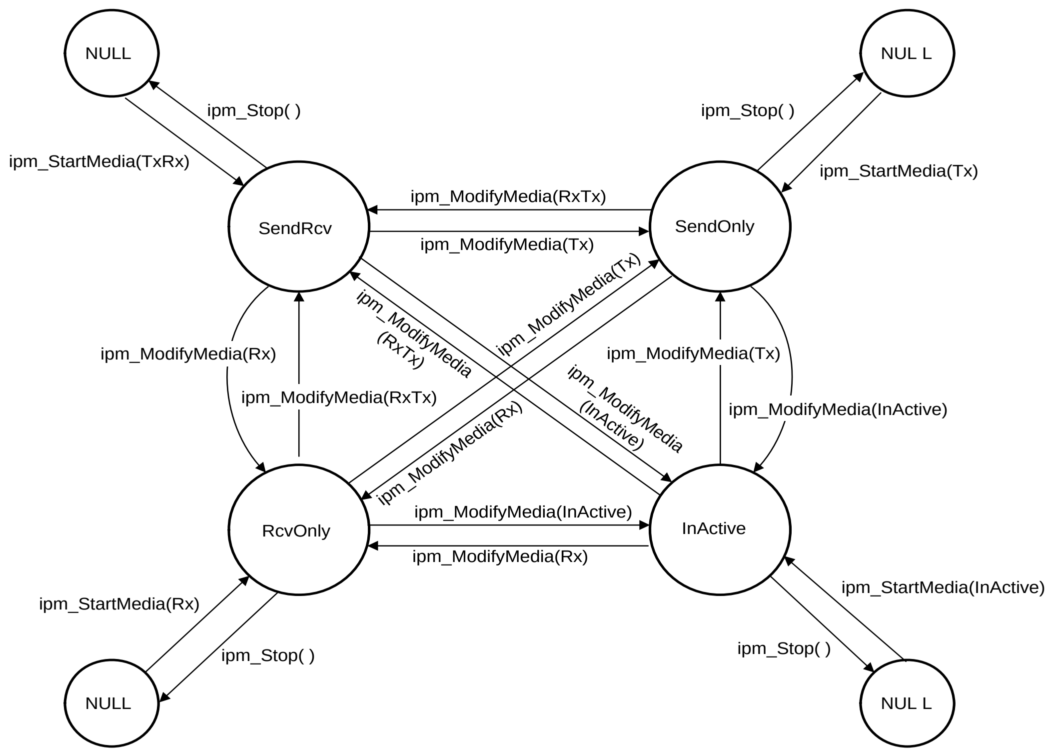
DATA_IP_DIR_NONE
: do not modify the direction of the current session; the previous direction remains in effect. This value is used when changing the coder and/or IP address without changing the direction.

Note: The DATA_IP_DIR_NONE parameter applies only when using the **ipm_ModifyMedia( )** function.

Both functions also have pointers to a media information structure through which coder attributes, such as the coder type, frame size, frames per packet setting etc. can also be configured.

Figure 8 shows the possible endpoint states for media streaming and how the **ipm_StartMedia( )** and **ipm_ModifyMedia( )** functions are used to transition between those states.

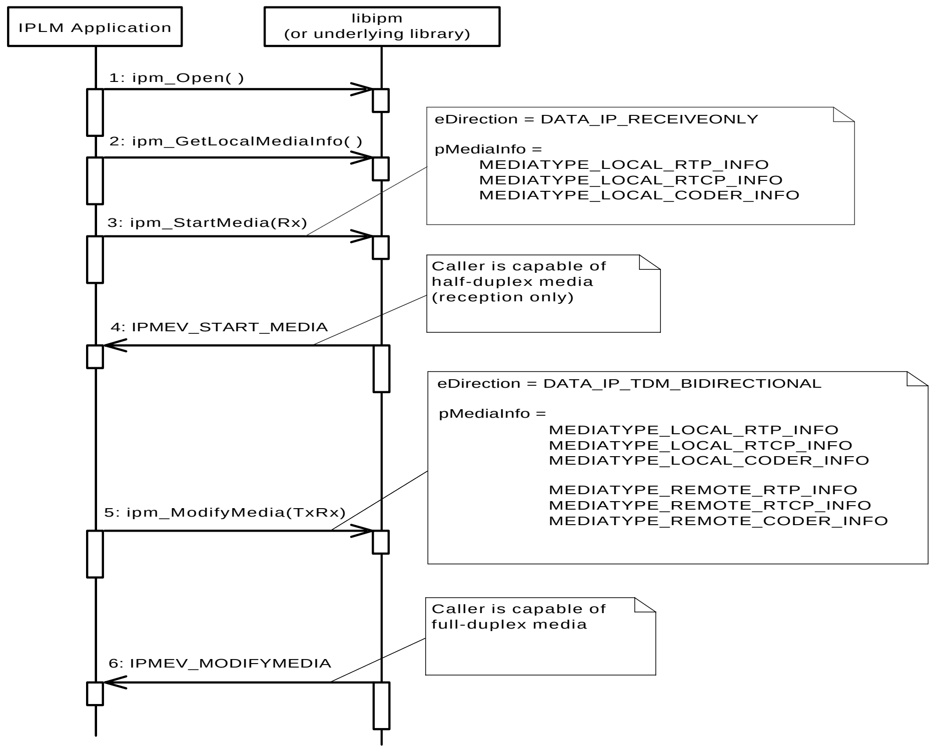
Figure 8. Endpoint Media Streaming State Transitions



12.3 Sample Scenario

Figure 9 shows a scenario in which the **ipm_StartMedia( )** function is used to configure an endpoint for a half-duplex (receive-only) media stream early in the setup sequence. This media stream can be used to receive call setup information, for example call progress tones. Later, the **ipm_ModifyMedia( )** function can be used to reconfigure the endpoint for full-duplex media streaming.

Figure 9. Half- and Full-Duplex Media Streaming Sample Scenario



12.4 Example Code

The following sample code demonstrates how to reconfigure an endpoint from full-duplex media streaming to half-duplex (send only) media streaming. The coder is also changed from G.711 u-law to G.711 A-law.

```
#include <stdio.h>
#include <string>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
   /*
   .
   .
   Main Processing
   .
   .
   */
```

```
/*
Set the media properties for a remote party using IP device handle, nDeviceHandle.
ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
*/

IPM_MEDIA_INFO MediaInfo;
MediaInfo.unCount = 4;

MediaInfo.MediaData[0].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTP_INFO;
MediaInfo.MediaData[0].mediaInfo.PortInfo.unPortId = 2328;
strcpy(MediaInfo.MediaData[0].mediaInfo.PortInfo.cIPAddress, "111.21.0.9");

MediaInfo.MediaData[1].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTCP_INFO;
MediaInfo.MediaData[1].mediaInfo.PortInfo.unPortId = 2329;
strcpy(MediaInfo.MediaData[1].mediaInfo.PortInfo.cIPAddress, "111.41.0.9");

MediaInfo.MediaData[2].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.unCoderPayloadType = 0;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.unRedPayloadType = 0;

MediaInfo.MediaData[3].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.unCoderPayloadType = 0;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.unRedPayloadType = 0;

if (ipm_StartMedia(nDeviceHandle, &MediaInfo, DATA_IP_TDM_BIDIRECTIONAL, EV_SYNC) == -1)
{
   printf("ipm_StartMediaInfo failed for device name = %s with error = %d\n",
   ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
   /*
   .
   .
   Perform Error Processing
   .
   .*/
}
/*
.
. Continue processing
.
*/

MediaInfo.unCount = 2;
MediaInfo.MediaData[0].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ALAW64K;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.unCoderPayloadType = 8;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.unRedPayloadType = 0;

MediaInfo.MediaData[1].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ALAW64K;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.unCoderPayloadType = 8;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.unRedPayloadType = 0;
```

```
   if (ipm_ModifyMedia(nDeviceHandle, &MediaInfo, DATA_IP_SENDONLY, EV_SYNC) == -1)
   {
      printf("ipm_Modify failed for device name = %s with error = %d\n",
      ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
      /*
      .
      .
      Perform Error Processing
      */
   }


   /*
   .
   .
   continue processing
   .
   */

}
```

DTMF Handling 13

This chapter contains guidelines for handling DTMF using the Dialogic® IP Media Library API. Topics include:

13.1 Feature Description

When a session is started on a Dialogic® IPM device, the IPM device receives data from its IP interface and transmits data to the TDM bus. A Dialogic® DTI device receives data from its public switched telephone network (PSTN) interface and transmits to the TDM bus as well. In a gateway configuration, the DTI and IPM devices are configured, via **gc_Listen( )** and **ipm_Listen( )** respectively, to listen to each other and thus create a full-duplex communication path. The IPM device forwards DTMF that it receives on one interface to the other interface. Figure 1, "IP Media Architecture", on page 22 shows the data flow between the Dialogic® IP Media Library API, the IP network, and the PSTN.

When an IPM device receives DTMF from the TDM bus, there are several ways to forward it to the IP interface. The DTMF transfer modes include:

- Encoding the DTMF as audio in the RTP stream (also called in-band)
- Sending the DTMF in the RTP stream via RFC 2833 packets
- Using an application-controlled method (also called out-of-band).

The IPM device can automatically forward the DTMF when either the in-band mode or the RFC 2833 mode has been selected. DTMF is **not** automatically forwarded when the out-of-band mode has been selected. In the out-of-band case, the application must call **ipm_ReceiveDigits( )** and have an IPM_DIGITS_RECEIVED event handler in place. Upon receiving the IPM_DIGITS_RECEIVED event, the DTMF information is contained in the IPM_DIGIT_INFO structure delivered with the event. The application has the responsibility to forward the DTMF via whatever mechanism, open or proprietary, it desires.

The DTMF transfer mode also affects the handling of DTMF that is received from the IP interface:

- In in-band mode, the DTMF is automatically forwarded to the TDM bus.
- In RFC 2833 mode, the DTMF is forwarded to the TDM bus as PCM data.
- In out-of-band mode, the application uses its own mechanism to be notified that a DTMF digit has been received. Then, **ipm_SendDigits( )** is used when necessary to transmit a DTMF digit to the TDM bus.

13.2 Setting DTMF Parameters

This section contains the following topics:

- DTMF Transfer Modes
- Setting In-Band Mode
- Setting Full-Duplex RFC 2833 Mode
- Setting Out-of-Band Mode
- Setting Receive-only RFC 2833 Mode

13.2.1 DTMF Transfer Modes

The Dialogic® IP Media Library API can be used to configure which DTMF transfer mode (in-band, RFC 2833, or out-of-band) is used by the application. The mode is set on a per-channel basis using **ipm_SetParm( )** and the IPM_PARM_INFO data structure.

The eIPM_DTMFXFERMODE enumeration identifies which DTMF mode to use. The following values are supported:

DTMFXFERMODE_INBAND
: DTMF digits are sent and received in-band via standard RTP transcoding. This is the default mode when a channel is opened.

  Note: In-band mode cannot be used when using low bit-rate (LBR) coders.

DTMFXFERMODE_RFC2833
: DTMF digits are sent and received in the RTP stream as defined in RFC 2833.

  Note: Receive-only RFC 2833 mode is also available. See Section 13.2.5, “Setting Receive-only RFC 2833 Mode”, on page 65.

DTMFXFERMODE_OUTOFBAND
: DTMF digits are sent and received outside the RTP stream.

When using RFC 2833, the payload type is specified through the following parameter/value setting in a call to **ipm_SetParm( )**:

PARMCH_RFC2833_EVT_TX_PLT
: Identifies the transmit payload type. The value range for this field is 96 to 127.

PARMCH_RFC2833_EVT_RX_PLT
: Identifies the receive payload type. The value range for this field is 96 to 127.

13.2.2 Setting In-Band Mode

In in-band mode, the DTMF audio is not clamped (not muted) and DTMF digits are sent in the RTP stream. When a channel is opened, the DTMF transfer mode is in-band by default.

Note: In-band mode cannot be used when using low bit-rate coders.

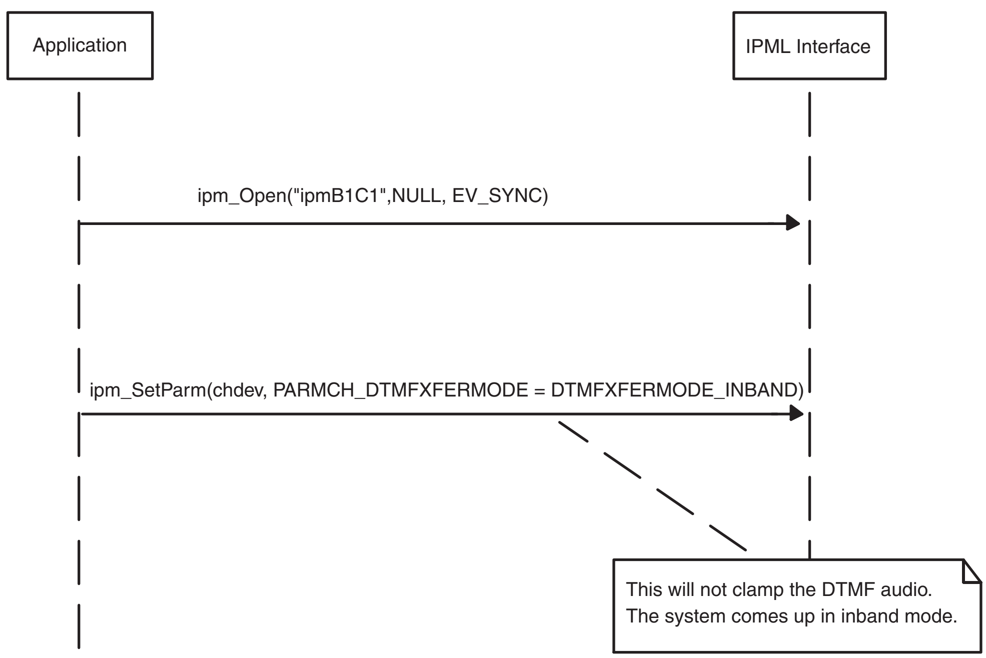
To set up a channel for in-band mode, do the following:

1. Open a channel using **ipm_Open( )**: **ipm_Open**("ipmB1C1",NULL,EV_SYNC).
2. Set the DTMF mode in the IPM_PARM_INFO structure and call **ipm_SetParm( )** as shown below:

```
IPM_PARM_INFO parmInfo;
unsigned long ulParmValue = DTMFXFERMODE_INBAND;
parmInfo.eParm = PARMCH_DTMFXFERMODE;
parmInfo.pvParmValue = &ulParmValue
ipm_SetParm(chdev, &parmInfo, EV_ASYNC)
```

Figure 10 shows a scenario diagram for setting in-band mode.

Figure 10. In-Band DTMF Mode Scenario Diagram



13.2.3 Setting Full-Duplex RFC 2833 Mode

In full-duplex RFC 2833 mode, the DTMF audio is clamped (muted) and DTMF digits are sent in the RTP stream only as RFC 2833 packets. To set up a channel for full-duplex RFC 2833 mode, do the following:

1. Open a channel using **ipm_Open( )**: **ipm_Open**("ipmB1C1",NULL,EV_SYNC).
2. Set the DTMF mode in the IPM_PARM_INFO structure and call **ipm_SetParm( )** as shown below:

```
IPM_PARM_INFO parmInfo;
unsigned long ulParmValue = DTMFXFERMODE_RFC2833;
parmInfo.eParm = PARMCH_DTMFXFERMODE;
parmInfo.pvParmValue = &ulParmValue
ipm_SetParm(chdev, &parmInfo, EV_ASYNC)
```

3. Set up the RFC 2833 event payload on the transmit side as shown below:

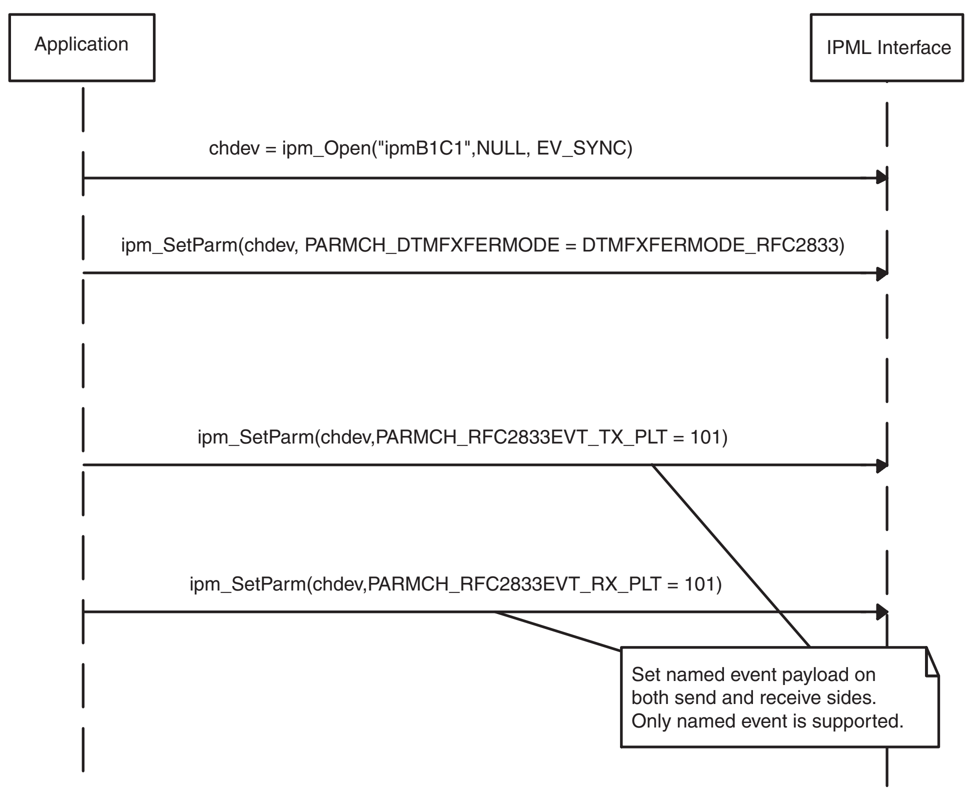
```
IPM_PARM_INFO parmInfo;
unsigned long ulParmValue = 101;
parmInfo.eParm = PARMCH_RFC2833EVT_TX_PLT;
parmInfo.pvParmValue = &ulParmValue
ipm_SetParm(chdev, &parmInfo, EV_ASYNC)
```

4. Set up the RFC 2833 event payload on the receive side as shown below:

```
IPM_PARM_INFO parmInfo;
unsigned long ulParmValue = 101;
parmInfo.eParm = PARMCH_RFC2833EVT_RX_PLT;
parmInfo.pvParmValue = &ulParmValue
ipm_SetParm(chdev, &parmInfo, EV_ASYNC)
```

Figure 11 shows a scenario diagram for setting RFC 2833 mode.

Figure 11. RFC 2833 Scenario Diagram



13.2.4 Setting Out-of-Band Mode

In out-of-band mode, the DTMF audio is automatically clamped (muted) and DTMF digits are not sent in the RTP packets. To set up a channel for out-of-band mode, do the following:

1. Open a channel using **ipm_Open( )**: **ipm_Open**("ipmB1C1",NULL,EV_SYNC).
2. Set the DTMF mode in the IPM_PARM_INFO structure and call **ipm_SetParm( )** as shown below:

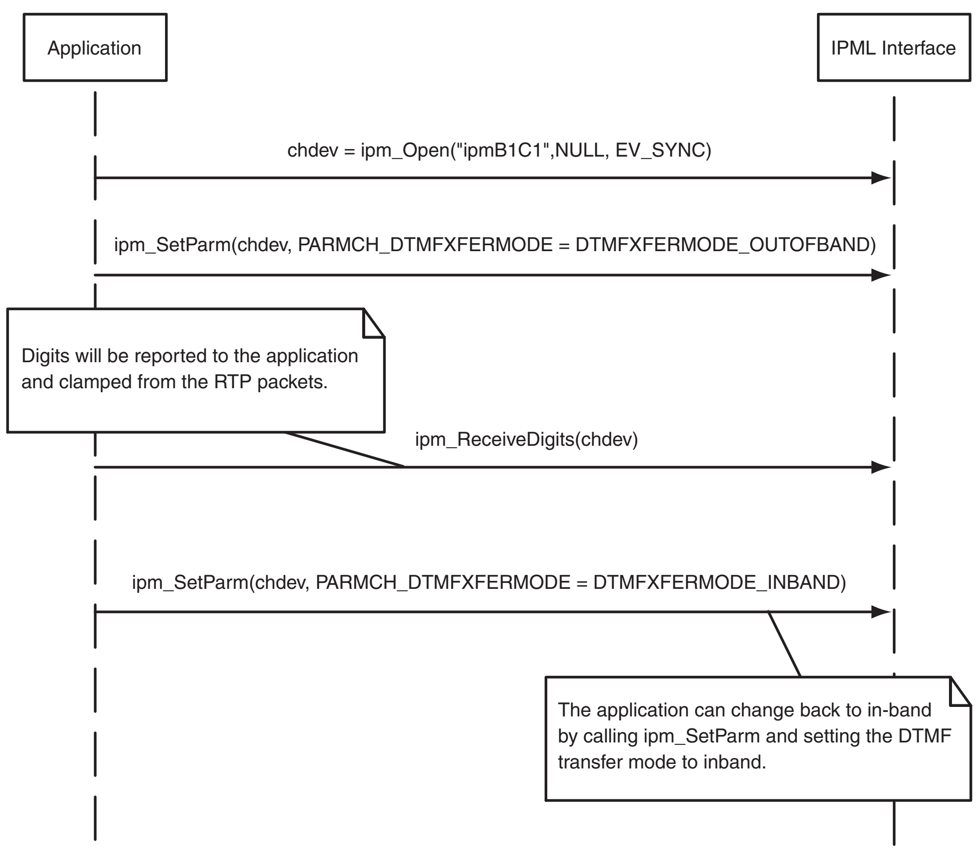
```
IPM_PARM_INFO parmInfo;
unsigned long ulParmValue = DTMFXFERMODE_OUTOFBAND;
parmInfo.eParm = PARMCH_DTMFXFERMODE;
parmInfo.pvParmValue = &ulParmValue
ipm_SetParm(chdev, &parmInfo, EV_ASYNC)
```

3. Call **ipm_ReceiveDigits( )** to have digits reported to the application and clamped from the RTP packets.

To change back to in-band mode, set the PARMCH_DTMFXFERMODE parameter to DTMFXFERMODE_INBAND.

Figure 12 shows a scenario diagram for setting out-of-band mode.

Figure 12. Out-of-Band DTMF Mode Scenario Diagram



13.2.5 Setting Receive-only RFC 2833 Mode

Receive-only RFC 2833 mode is not supported on all releases. For support information, see Chapter 2, "Feature Support by Platform".

In full-duplex RCF 2833 mode, the mechanism used to detect and remove in-band DTMF digits from the audio stream prior to the transmission of audio RTP packets contributes to audio latency. In receive-only RFC 2833 mode, this additional transmit audio latency is reduced.

For the third-party call control model (3PCC), select this mode of operation at runtime using **ipm_SetParm( )** with the appropriate parameter and value prior to invoking **ipm_StartMedia( )**. The parameter to set is PARMCH_DISABLE_TX_TELEPHONY_EVENT in the IPM_PARM_INFO data structure. If needed, this mode can be set on a board basis through the *Hmp.Uconfig* file. For more information, see the *Dialogic® System Configuration Guide.*

For the first-party call control model (1PCC), select this mode using the parameter element IPSET_DTMF in GC_PARM_BLK, which is associated with **gc_SetUserInfo( )** and **gc_SetConfigData( )**. For more information, see the *Dialogic® Global Call IP Technology Guide*.

Example Code Using Dialogic® IP Media Library API

The following example demonstrates how to set or get parameters for receive-only RFC 2833 mode using the Dialogic® IP Media Library API.

```
int disable;
IPM_PARM_INFO ParmInfo;

// Open the IPM Device
int handle;
handle = ipm_Open("ipmB1C1",NULL,EV_SYNC);

// Set the TX disable parameter
disable = 1;
ParmInfo.eParm =  PARMCH_DISABLE_TX_TELEPHONY_EVENT;
ParmInfo.pvParmValue = &disable;
ipm_SetParm(handle, &ParmInfo, EV_SYNC);
```

Example Code Using Dialogic® Global Call API

The following example demonstrates how to set PARMCH_DISABLE_TX_TELEPHONY_EVENT using the Dialogic® Global Call API.

```
int  value  = 1;    // Disable transmit RFC2833 digits
IPM_PARM_INFO ipmParmInfo;
GC_PARM_BLKP parmblkp;

ipmParmInfo.eParm = PARMCH_DISABLE_TX_TELEPHONY_EVENT;
ipmParmInfo.pvParmValue = (void *)&value;

gc_util_insert_parm_ref(&parmblkp, IPSET_CONFIG, IPPARM_IPMPARM, (unsigned
long)sizeof(IPM_PARM_INFO), &ipmParmInfo);
gc_SetUserInfo(GCTGT_GCLIB_CHAN, lineDev, parmblkp, GC_ALLCALLS);
gc_util_delete_parm_blk(parmblkp);
```

For more information, see the *Dialogic® Global Call IP Technology Guide*.

Only the GC_ALLCALLS mode supports this feature.

Download and Startup Parameter Usage

To set receive-only RFC 2833 mode on a board basis, add the following line to the IPVSC [0x40] section of the *Hmp.Uconfig* file:

```
SetParm=0x4019, 1   ! 1 means Disable transmit RFC2833 digits
```

Run the FCDGEN utility to generate the corresponding FCD file. Stop the Dialogic® Services, and then re-start services.

13.3 Notification of DTMF Detection

Notification of DTMF detection depends on the DTMF mode being used. For out-of-band mode, when an incoming DTMF digit is detected (received from the TDM bus), the application receives an unsolicited IPMEV_DIGITS_RECEIVED event. The event data is contained in IPM_DIGIT_INFO. One event is returned for each digit that is received.

13.4 Generating DTMF

Once DTMF mode has been configured, the application can generate DTMF digits using the **ipm_SendDigits( )** function.

Note: The only supported direction for DTMF digit generation is to the TDM bus.

T.38 Fax Server 14

This chapter discusses the T.38 fax server support in the Dialogic® IP Media Library API. Topics include:

14.1 Feature Description

The Dialogic® IP Media Library API supports T.38 fax server capability via the T.38 fax resource. The T.38 fax resource provides the host application the ability to initiate T.38 fax functionality, including modifying the codec from audio to T.38 and T.38 only.

Note: The T.38 fax resource does not support the gateway mode nor does it support T.38 fax relay capability (T.38 packet to V.17/V.27/V.21 fax modem conversion and vice versa). Hence, the fax data cannot be shared on the CT Bus by multiple channels.

Since the T.38 fax server resource has control of the UDP port, unlike the gateway model where the ipm channel controls the UDP port, two additional API functions, **dev_Connect( )** and **dev_Disconnect( )** are needed to associate or disassociate the voice media handle and the fax handle. When **dev_Connect( )** is executed on an ipm channel and a T.38ServerFax resource, the IP media library API translates the ipm_(Get/Start)LocalMediaInfo( ) API call to a T38ServerFax_msg(Get/Set)Parm. As soon as **dev_Disconnect( )** is issued, this translation is stopped and messages are forwarded to the ipm channel. For more information on dev_ functions, see the *Dialogic® Device Management API Library Reference*.

When using third party IP call control engines, specify the following sequence of calls in the application to make and break a T.38 session for sending fax. The Dialogic® IP Media Library API provides the primitives to control media/session parameters.

1. Open an ipm channel using **ipm_Open( )**. For example:
   ```
   ipmDevH1 = ipm_open("ipmB1C1")
   ```
2. Open a dxxx channel to be used for fax using **dx_open( )**. For example:
   ```
   dxDevH1 = dx_open("dxxxB17C3")
   ```
3. Issue **dx_getfeaturelist( )** on the dxxx channel to verify that this channel supports fax. For example:
   ```
   dx_getfeaturelist(dxDevH1,feature_tablep)
   ```
4. Verify that **dx_getfeaturelist( )** returns FT_FAX for ft_fax bitmask in the FEATURE_TABLE structure. For example:
   ```
   if (feature_tablep->ft_fax & FT_FAX)
   ```
5. Open the same dxxx channel using **fx_open( )**. For example:
   ```
   faxdevH1 = fx_open("dxxxB17C3")
   ```

6. Issue **dx_getfeaturelist( )** to determine whether this fax resource supports T.38 fax. For example:
```
if (feature_tablep->ft_fax & FT_FAX_T38UDP)
```
7. To route the fax channel to the ipm channel, use **dev_Connect( )**. For example:
```
ret = dev_Connect(ipmDevH1, faxdevH1, DM_FULLDUP, EV_ASYNC)
```
8. Process the DMEV_CONNECT completion event.

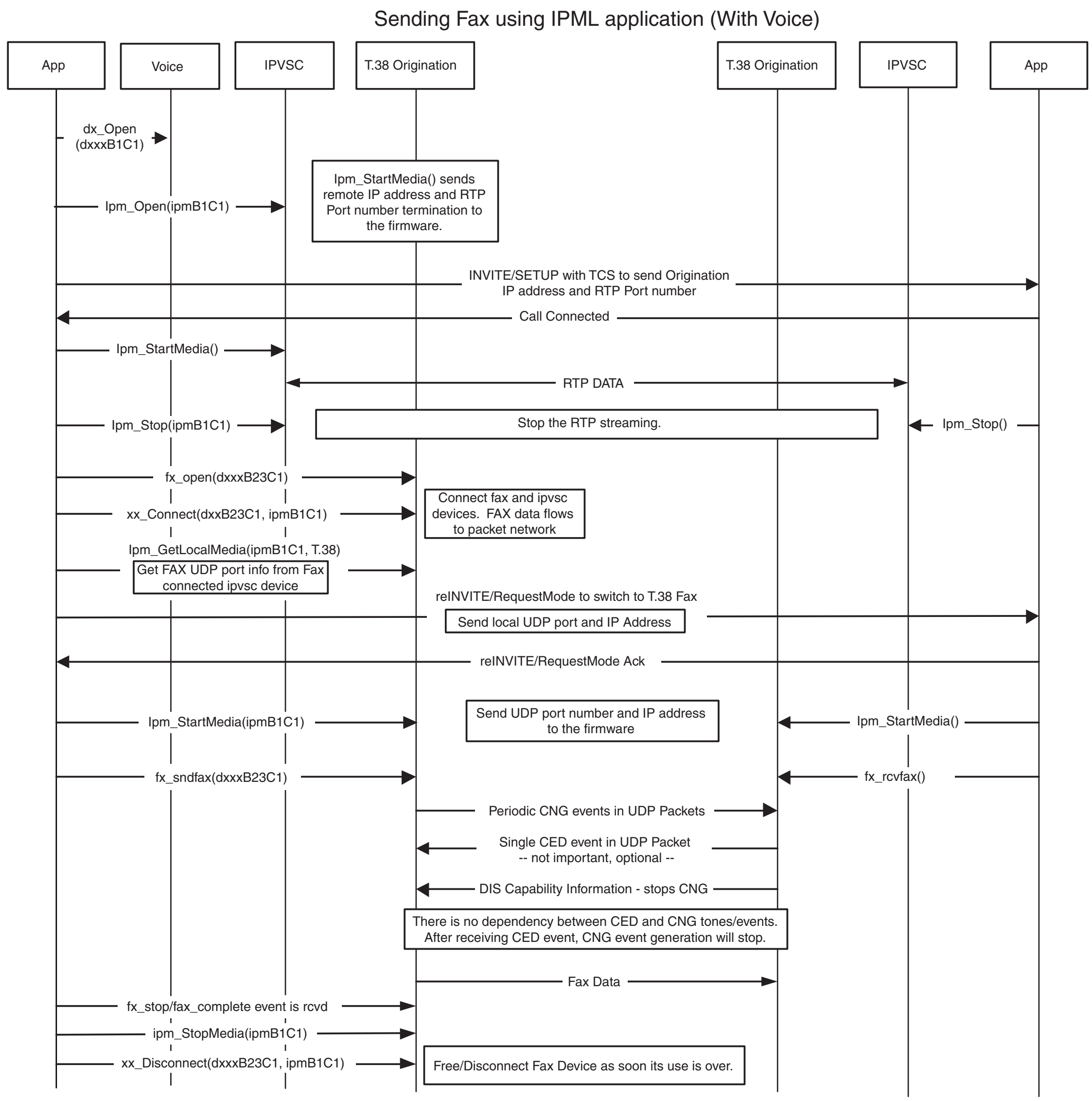
   Note: DM_FULLDUP is the only mode supported when passing T.38 devices because the connection is made logically in both directions.
9. Issue **ipm_GetLocalMediaInfo( )** to get the T.38 port and IP address information. The first media type in the IPM_MEDIA structure must be set to MEDIATYPE_LOCAL_UDPTL_T38_INFO. Process the IPMEV_GET_LOCAL_MEDIA_INFO completion event.
10. Get the remote end IP address and port information, achieved via signaling.
11. Issue **ipm_StartMedia( )** to start media streaming. Specify the remote T.38 information obtained earlier. Process the IPMEV_START_MEDIA completion event.
12. To begin fax transmission, use **fx_sendfax( )**. For example:
```
fx_sendfax(faxdevH1,EV_ASYNC)
```
13. Process the TFX_FAXSEND completion event.
14. When fax transmission is completed, use **ipm_Stop( )** to stop operations on the ipm channel.

For information on dx_ functions, see the Dialogic® Voice API documentation. For information on fx_ functions, see the Dialogic® Fax API documentation.

14.2 Sample Scenario for T.38 Fax Server

The following figure illustrates a T.38 fax server call scenario.

Figure 13. T.38 Fax Server Call Scenario



14.3 Example Code for T.38 Fax Server

The following illustrates example code for T.38 fax server.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <faxlib.h>
#include <ipmlib.h>
#include <devmgmt.h>

static int ipm_handle = -1;
static int fax_handle = -1;

static DF_IOTT iott = {0};
static int fd = 0;
static IPM_MEDIA_INFO info;

static bool ipm_handle_disconnected = false;
static bool fax_handle_disconnected = false;

long IpmEventHandler( unsigned long evthandle )
{
    int evttype = sr_getevttype();

    printf("Event=0x%x SRL handle=0x%x\n",evttype, evthandle);
    switch( evttype )
    {
    case DMEV_CONNECT:
        printf( "DMEV_CONNECT event received.\n" );
        {
            info.MediaData[0].eMediaType = MEDIATYPE_LOCAL_UDPTL_T38_INFO;
            if( ipm_GetLocalMediaInfo( ipm_handle, &info, EV_ASYNC ) == -1 )
            {
                printf( "ipm_GetLocalMediaInfo() failed.\n" );
                exit( 1 );
            }
        }
        break;
    case IPMEV_GET_LOCAL_MEDIA_INFO:
        printf( "IPMEV_GET_LOCAL_MEDIA_INFO event received.\n" );

        {
            info.unCount = 1;
            info.MediaData[0].eMediaType = MEDIATYPE_REMOTE_UDPTL_T38_INFO;
            info.MediaData[0].mediaInfo.PortInfo.unPortId = 2001;// remote IP port
            strcpy( info.MediaData[0].mediaInfo.PortInfo.cIPAddress, "146.152.84.56");
            info.MediaData[1].eMediaType = MEDIATYPE_FAX_SIGNAL;
            info.MediaData[1].mediaInfo.FaxSignal.eToneType = TONE_CED;
            printf("Press enter to continue (ipm_StartMedia)\n");
            //getchar();
            //printf( "calling ipm_StartMedia()\n" );
            if( ipm_StartMedia( ipm_handle, &info, DATA_IP_TDM_BIDIRECTIONAL, EV_ASYNC ) == -1 )
            {
                printf( "ipm_StartMedia() failed.\n" );
                exit( 1 );
            }
            else
            {
                printf("[%s] ipm_StartMedia ok \n", ATDV_NAMEP(ipm_handle));
            }
            //getchar();
            //printf("Press enter to continue (ipm_StartMedia)\n");
        }
        break;
    case DMEV_DISCONNECT:
        printf( "DMEV_DISCONNECT event received.\n" );
```

```
            ipm_handle_disconnected = true;
            if( fax_handle_disconnected )
            {
                return 0;
            }
            break;

        case IPMEV_STARTMEDIA:
            printf( "IPMEV_STARTMEDIA event received.\n" );
            fd = dx_fileopen( "onepg_high.tif", O_RDONLY|O_BINARY );
            if( fd == -1 )
            {
                printf( "dx_fileopen() failed.\n" );
                exit( 1 );
            }
            fx_setiott(&iott, fd, DF_TIFF, DFC_EOM);
            iott.io_type |= IO_EOT;
            iott.io_firstpg = 0;
            iott.io_pgcount = -1;
            iott.io_phdcont = DFC_EOP;
            if( fx_initstat( fax_handle, DF_TX ) == -1 )
            {
                printf( "fx_initstat() failed.\n" );
                exit( 1 );
            }
            if( fx_sendfax( fax_handle, &iott, EV_ASYNC ) == -1 )
            {
                printf( "fx_sendfax() failed.\n" );
                exit( 1 );
            }
            break;

        case IPMEV_STOPPED:
            printf( "IPMEV_STOPPED event received.\n" );
            if( dev_Disconnect( ipm_handle, EV_ASYNC ) == -1 )
            {
                printf( "dev_Disconnect() failed.\n" );
                exit( 1 );
            }

            if( dev_Disconnect( fax_handle, EV_ASYNC ) == -1 )
            {
                printf( "dev_Disconnect() failed.\n" );
                exit( 1 );
            }
            break;
        case IPMEV_ERROR:
            printf( "IPMEV_ERROR event received on IPM channel.\n" );
            exit( -1 );
            break;

        default:
            printf( "Unknown event %d received.\n", evttype );
            break;
        }
        return 0;
    }

    long FaxEventHandler( unsigned long evthandle )
    {
        int evttype = sr_getevttype();

        switch( evttype )
        {
        case TFX_FAXSEND:
            printf( "TFX_FAXSEND event received.\n" );
            if( ipm_Stop( ipm_handle, STOP_ALL, EV_ASYNC ) == -1 )
```

```
        {
            printf( "ipm_Stop() failed.\n" );
            exit( 1 );
        }
        break;

    case TFX_FAXERROR:
        printf( "TFX_FAXERROR event received.\n" );
        exit( 1 );
        break;
    default:
        printf( "Unknown event %d received on fax channel.\n", evttype );
        break;
    }
    return 0;
}

void main()
{
    ipm_handle = ipm_Open( "ipmB1C1", NULL, EV_SYNC );
    if( ipm_handle == -1 )
    {
        printf( "ipm_Open() failed.\n" );
        exit( 1 );
    }

    int vox_handle = dx_open( "dxxxB2C1", 0 );
    if( vox_handle == -1 )
    {
        printf( "dx_open() failed.\n" );
        exit( 1 );
    }

    FEATURE_TABLE feature_table;
    if( dx_getfeaturelist( vox_handle, &feature_table ) == -1 )
    {
        printf( "dx_getfeaturelist() failed.\n" );
        exit( 1 );
    }

    if( dx_close( vox_handle ) == -1 )
    {
        printf( "dx_close() failed.\n" );
        exit( 1 );
    }

    if( feature_table.ft_fax & FT_FAX )
    {
        if( feature_table.ft_fax & FT_FAX_T38UDP)
        {
            fax_handle = fx_open( "dxxxB2C1", 0 );
            if( fax_handle == -1 )
            {
                printf( "fx_open() failed.\n" );
                exit( 1 );
            }
        }
        else
        {
            printf( "Not a T.38 fax device.\n" );
            exit( 1 );
        }
    }
    else
    {
        printf( "Not a fax device.\n" );
        exit( 1 );
```

```
    }
    if( sr_enbhdlr( ipm_handle, EV_ANYEVT, IpmEventHandler ) == -1 )
    {
        printf( "sr_enbhdlr() failed.\n" );
        exit( 1 );
    }

    if( sr_enbhdlr( fax_handle, EV_ANYEVT, FaxEventHandler ) == -1 )
    {
        printf( "sr_enbhdlr() failed.\n" );
        exit( 1 );
    }

    if( dev_Connect( ipm_handle, fax_handle, DM_FULLDUP, EV_ASYNC ) == -1 )
    {
        printf( "dev_Connect() failed.\n" );
        exit( 1 );
    }

    while(1)
    {
        sr_waitevt(-1);
        printf("Got an event\n");
    }

    if( sr_dishdlr( fax_handle, EV_ANYEVT, FaxEventHandler ) == -1 )
    {
        printf( "sr_dishdlr() failed.\n" );
        exit( 1 );
    }

    if( sr_dishdlr( ipm_handle, EV_ANYEVT, IpmEventHandler ) == -1 )
    {
        printf( "sr_dishdlr() failed.\n" );
        exit( 1 );
    }

    if( fx_close( fax_handle ) == -1 )
    {
        printf( "fx_close() failed.\n" );
        exit( 1 );
    }

    if( ipm_Close( ipm_handle, NULL ) == -1 )
    {
        printf( "ipm_Close() failed.\n" );
        exit( 1 );
    }
}
```

Implementing Native T.38 Fax Hairpinning 15

This chapter discusses native T.38 fax hairpinning and provides implementation guidelines. Topics include:

15.1 Feature Description

Native T.38 fax hairpinning is not supported on all releases. For support information, see Chapter 2, "Feature Support by Platform".

Native T.38 fax hairpinning is supported in 3rd Party Call Control (3PCC) SIP.

The native T.38 fax hairpinning feature enables an application to natively route T.38 fax data between the local UDP ports of two IP media devices with established native T.38 fax sessions. UDP packets are routed between these ports and only the source port and the destination port of the UDP header is modified. This means that the T.38 fax payload is not processed in any way and is routed transparently.

Two IP media devices are used to form a native T.38 fax hairpin connection. Each of the IP media devices have IP audio sessions established using the **ipm_StartMedia( )** function and the audio coder type, CODER_TYPE_ UDPTL_NATIVE. Native hairpin connections are formed between the audio ports of the two IP media devices using the **dev_PortConnect( )** function with the DMFL_TRANSCODE_NATIVE flag set.

The **ipm_GetLocalMediaInfo( )** function, with the eMediaType field (IPM_MEDIA structure) set to MEDIATYPE_AUDIO_LOCAL_RTP_INFO, is used to retrieve the local IP address and local port information to be used with the native T.38 hairpinning feature. The RTP port information returned corresponds to the local UDP port. The RTCP port information returned can be ignored since no RTCP port or RTCP-related capabilities apply for the native T.38 hairpinning feature.

Note: QoS alarms are not supported when the audio coder type, CODER_TYPE_UDPTL_NATIVE, is selected.

15.2 Implementation Guidelines

To implement native T.38 hairpinning in your application, follow the guidelines provided in these sections.

- Initializing Structures
- Connecting Devices
- Exchange Media Using ipm_StartMedia( )

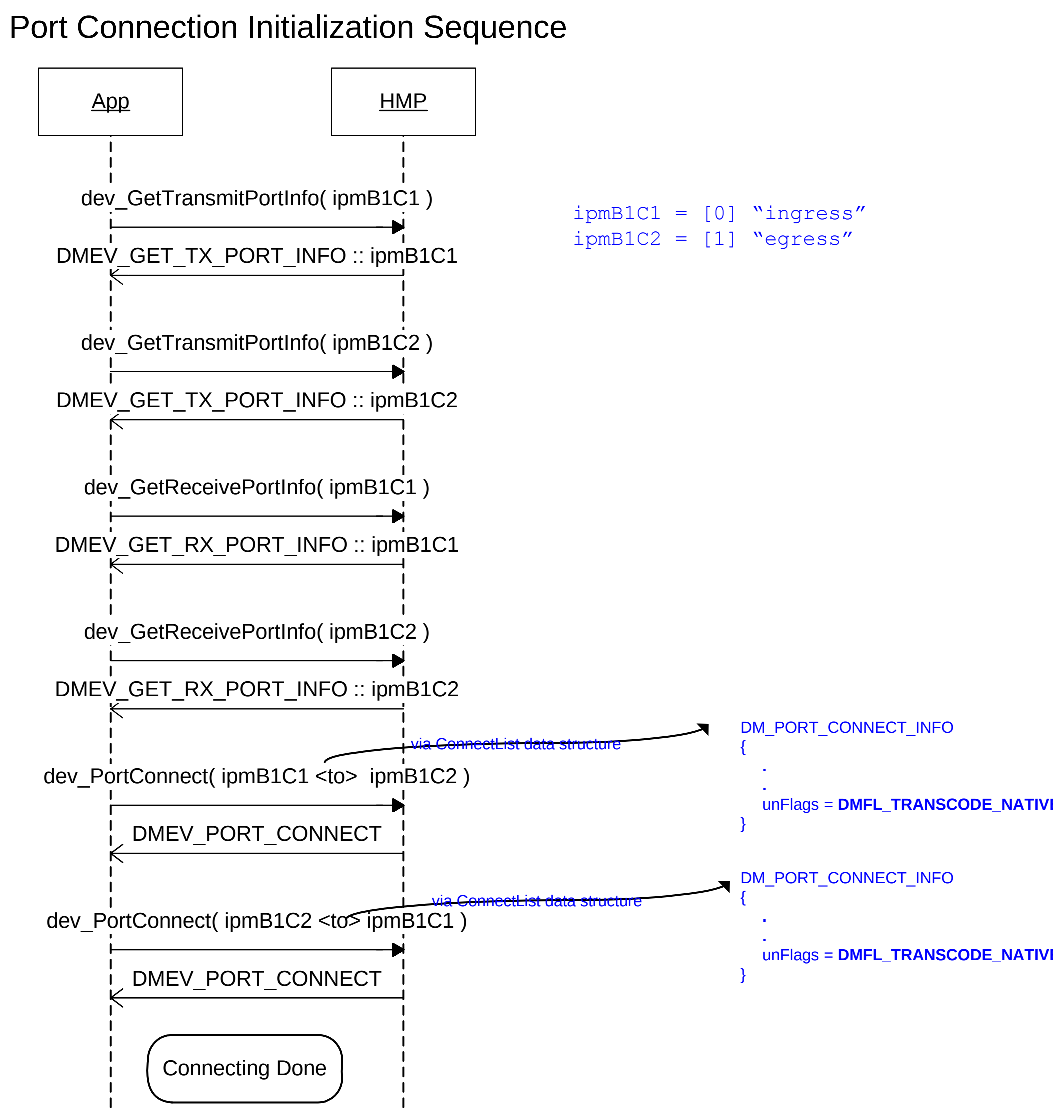
15.2.1 Initializing Structures

Before calling **ipm_StartMedia( )**, initialize data structures: IPM_MEDIA_INFO, IPM_MEDIA, and IPM_AUDIO_CODER_INFO.

15.2.2 Connecting Devices

Once SIP calls have been established, connect IPM devices using the Dialogic® Device Management API. Get device transmit ports and device receive ports for the IPM devices using **dev_GetTransmitPortInfo( )** and **dev_GetReceive PortInfo( )**. Connect IPM devices to each other using **dev_PortConnect( )** with the DMFL_TRANSCODE_NATIVE flag set.

The following figure shows a sequence of calls to establish the connection:



15.2.3 Exchange Media Using ipm_StartMedia( )

The following source code demonstrates the API calls required to initiate the transport of T.38 packets between the two IP media devices.

In this example, the **ipm_GetLocalMediaInfo( )** function collects IP address and port data from each IPM device. Next, the IPM_MEDIA_INFO data structure, passed to **ipm_StartMedia( )**, is populated. The coder type CODER_TYPE_UDPTL_NATIVE indicates native T.38 fax hairpinning.

```
#define NUMCHAN 2

class CHANNEL // in pairs
{
  int             m_ipm_ddd;
  IPM_MEDIA_INFO  m_ReadInfo;
};
CHANNEL l_Chan[NUMCHAN];

int StartT38Hairpin()
{
 int rc;

 IPM_MEDIA_INFO Input;
 IPM_MEDIA_INFO Output;

 Input.unCount = 0;
 Output.unCount = 0;

 i = 0;

 l_Chan[i].m_ReadInfo.unCount=1;
 l_Chan[i].m_ReadInfo.MediaData[0].eMediaType=MEDIATYPE_AUDIO_LOCAL_RTP_INFO;

 rc=ipm_GetLocalMediaInfo(l_Chan[i].m_ipm_ddd ,&(l_Chan[i].m_ReadInfo),EV_SYNC);
 if (rc<0)
  {
   printf("Error Getting Local Media Info channel %i\n",i);
   return -1;;
  }

 Input.MediaData[Input.unCount].eMediaType = MEDIATYPE_REMOTE_RTP_INFO;
 Input.MediaData[Input.unCount].mediaInfo.PortInfo.unPortId =
       l_Chan[i].m_ReadInfo.mediaInfo.PortInfo.unPortId;
 strcpy(Input.MediaData[Input.unCount].mediaInfo.PortInfo.cIPAddress,
       l_Chan[i].m_ReadInfo.mediaInfo.PortInfo.cIPAddress);
 Input.unCount++;

 Input.MediaData[Input.unCount].eMediaType = MEDIATYPE_REMOTE_RTCP_INFO;
 Input.MediaData[Input.unCount].mediaInfo.PortInfo.unPortId =
       l_Chan[i].m_ReadInfo.mediaInfo.PortInfo.unPortId;+1;
 strcpy(Input.MediaData[Input.unCount].mediaInfo.PortInfo.cIPAddress,
       l_Chan[i].m_ReadInfo.mediaInfo.PortInfo.cIPAddress);
 Input.unCount++;

 Input.MediaData[Input.unCount].eMediaType = MEDIATYPE_REMOTE_CODER_INFO;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_UDPTL_NATIVE;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.unFramesPerPkt = 1;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.unCoderPayloadType  = 0;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.unRedPayloadType    = 0;
 Input.unCount++;

 Input.MediaData[Input.unCount].eMediaType = MEDIATYPE_LOCAL_CODER_INFO;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.eCoderType    = CODER_TYPE_UDPTL_NATIVE;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.eFrameSize    = (eIPM_CODER_FRAMESIZE) 30;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.unFramesPerPkt      = 1;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.eVadEnable          = CODER_VAD_DISABLE;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.unCoderPayloadType  = 0;
 Input.MediaData[Input.unCount].mediaInfo.CoderInfo.unRedPayloadType    = 0;
 Input.unCount++;
 i = 1;

 l_Chan[i].m_ReadInfo.unCount=1;
 l_Chan[i].m_ReadInfo.MediaData[0].eMediaType=MEDIATYPE_AUDIO_LOCAL_RTP_INFO;
 rc=ipm_GetLocalMediaInfo(l_Chan[i].m_ipm_ddd,&(l_Chan[i].m_ReadInfo),EV_SYNC);
```

```
 if (rc<0)
  {
    printf("Error Getting Local Media Info channel %i\n",i);
    return -1;;
   }

 Output.MediaData[Output.unCount].eMediaType = MEDIATYPE_REMOTE_RTP_INFO;
 Output.MediaData[Output.unCount].mediaInfo.PortInfo.unPortId =
        l_Chan[i].m_ReadInfo.mediaInfo.PortInfo.unPortId;
 strcpy(Output.MediaData[Output.unCount].mediaInfo.PortInfo.cIPAddress,
        l_Chan[i].m_ReadInfo.mediaInfo.PortInfo.cIPAddress);
 Output.unCount++;

 Output.MediaData[Output.unCount].eMediaType= MEDIATYPE_REMOTE_RTCP_INFO;
 Output.MediaData[Output.unCount].mediaInfo.PortInfo.unPortId =
        l_Chan[i].m_ReadInfo.mediaInfo.PortInfo.unPortId+1;
 strcpy(Output.MediaData[Output.unCount].mediaInfo.PortInfo.cIPAddress,
        l_Chan[i].m_ReadInfo.mediaInfo.PortInfo.cIPAddress);
 Output.unCount++;

 Output.MediaData[Output.unCount].eMediaType= MEDIATYPE_REMOTE_CODER_INFO;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.eCoderType     = CODER_TYPE_UDPTL_NATIVE;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.eFrameSize     = (eIPM_CODER_FRAMESIZE) 30;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.unFramesPerPkt       = 1;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.eVadEnable           = CODER_VAD_DISABLE;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.unCoderPayloadType  = 0;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.unRedPayloadType    = 0;
 Output.unCount++;

 Output.MediaData[Output.unCount].eMediaType= MEDIATYPE_LOCAL_CODER_INFO;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.eCoderType    = CODER_TYPE_UDPTL_NATIVE;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.eFrameSize    = (eIPM_CODER_FRAMESIZE) 30;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.unFramesPerPkt       = 1;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.eVadEnable           = CODER_VAD_DISABLE;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.unCoderPayloadType  = 0;
 Output.MediaData[Output.unCount].mediaInfo.CoderInfo.unRedPayloadType    = 0;
 Output.unCount++;

 rc = ipm_StartMedia(l_Chan[0].m_ipm_ddd, &Output, DATA_IP_TDM_BIDIRECTIONAL, EV_SYNC);
 if (rc == -1)
  {
   printf("ipm_StartMedia failed: %s\n",ATDV_ERRMSGP(l_Chan[0].m_ipm_ddd));
   exit(1);
  }

 rc = ipm_StartMedia(l_Chan[1].m_ipm_ddd, &Input, DATA_IP_TDM_BIDIRECTIONAL, EV_SYNC);
 if (rc == -1)
 {
  printf("ipm_StartMedia failed: %s\n",ATDV_ERRMSGP(l_Chan[1].m_ipm_ddd));
  exit(1);
 }

 return rc;

}
```

15.3 Sample Scenarios

Prior to establishing a native T.38 hairpin between two IP media devices, the application needs to initiate an inbound and outbound SIP call using 3PCC mechanisms. These SIP calls begin as either T.38 media calls or audio calls. Next the application needs to ensure that both IP media devices (and their respective end-points) have established T.38 sessions.

The following sections show 3PCC sample scenarios. Each scenario references the concept of a native audio full-duplex connection and the API call to the **ipm_StartMedia( )** function. See Section 15.2.2, “Connecting Devices”, on page 75 and Section 15.2.3, “Exchange Media Using ipm_StartMedia( )”, on page 76 for more information.

- Scenario 1: INVITE for T.38 Fax Call
- Scenario 2: Re-INVITE for T.38 Fax Call
- Scenario 3: Re-INVITE from Native Hairpin of Audio to Native Hairpin of T.38
- Scenario 4: Re-INVITE from Non-native Hairpin of Audio to Native Hairpin of T.38

15.3.1 Scenario 1: INVITE for T.38 Fax Call

The application wants to respond to an incoming T.38 fax call by hairpinning it to another T.38 fax call. The application initiates an outgoing T.38 fax call and bridges the two calls. Assumptions: the incoming T.38 FAX-A call media session is established on IPM-A and the outgoing T.38 FAX-B call media session is established on IPM-B.

The sequence of activities is as follows:

1. An INVITE for the T.38 FAX-A call is received by the application. IPM-A is selected to receive this call.
2. Native audio full-duplex connections are established between IPM-A and IPM-B using **dev_PortConnect( )**; see Section 15.2.2, “Connecting Devices”, on page 75.
3. An INVITE for the T.38 FAX-B call is sent for IPM-B.
4. A 200 OK is received for the T.38 FAX-B call.
5. **ipm_StartMedia( )** is called for IPM-B to establish the full-duplex native T.38 session for call B.
6. **ipm_StartMedia( )** is called for IPM-A to establish the full-duplex native T.38 session for call A.
7. A 200 OK is sent for the T.38 FAX-A call.

15.3.2 Scenario 2: Re-INVITE for T.38 Fax Call

The application wants to transition from an existing audio session between the media server and a remote terminal to a T.38 session at the request of the remote terminal. An outgoing T.38 call is initiated by the application, and the two calls are bridged. Assumptions: the incoming T.38 FAX-A call media session is established on IPM-A, and the outgoing T.38 FAX-B call media session is established on IPM-B.

The sequence of activities is as follows:

1. There is an existing audio call on IPM-A.
2. A re-INVITE for the T.38 FAX-A call is received by the application for IPM-A.
3. **ipm_Stop( )** is called for IPM-A to end the full-duplex audio session for call A.

4. Previous internal IPM-A connections are broken. Then native audio full-duplex connections are established between IPM-A and IPM-B using **dev_PortConnect( )**; see Section 15.2.2, “Connecting Devices”, on page 75.
5. An INVITE for the T.38 FAX-B call is sent for IPM-B.
6. A 200 OK is received for the T.38 FAX-B call.
7. **ipm_StartMedia( )** is called for IPM-B to establish the full-duplex native T.38 session for call B.
8. **ipm_StartMedia( )** is called for IPM-A to establish the full-duplex native T.38 session for call A.
9. A 200 OK is sent for the T.38 FAX-A call.

15.3.3 Scenario 3: Re-INVITE from Native Hairpin of Audio to Native Hairpin of T.38

Two existing native audio sessions, one on IPM-A and the other on IPM-B, are natively bridged by the application. The application receives a re-INVITE from the remote terminal associated with the session on IPM-A to modify the media to T.38; this re-INVITE is passed on to the remote terminal associated with the session on IPM-B. The remote terminal associated with IPM-B accepts the re-INVITE. The application stops the audio sessions on both IPM-A and IPM-B, and terminates the audio bridge connection between IPM-A and IPM-B. The application then establishes native T.38 fax sessions on both IPM-A and IPM-B, and creates a full-duplex native T.38 hairpin connection between IPM-A and IPM-B. The application then sends the 200 OK to the remote terminal associated with the session on IPM-A.

The sequence of activities is as follows:

1. There is an existing audio call bridged between IPM-A and IPM-B.
2. A re-INVITE for the T.38 FAX-A call is received by the application for IPM-A.
3. A re-INVITE for the T.38 FAX-B call is sent by the application for IPM-B.
4. A 200 OK is received for the T.38 FAX-B call.
5. **ipm_Stop( )** is called for IPM-B to end the full-duplex audio session for call B.
6. The existing audio bridge connection between IPM-A and IPM-B is disconnected.
7. **ipm_Stop( )** is called for IPM-A to end the full-duplex audio session for call A.
8. Native audio full-duplex connections are established between IPM-A and IPM-B using **dev_PortConnect( )**; see Section 15.2.2, “Connecting Devices”, on page 75.
9. **ipm_StartMedia( )** is called for IPM-B to establish the full-duplex native T.38 session for call B.
10. **ipm_StartMedia( )** is called for IPM-A to establish the full-duplex native T.38 session for call A.
11. A 200 OK is sent for the T.38 FAX-A call.

15.3.4 Scenario 4: Re-INVITE from Non-native Hairpin of Audio to Native Hairpin of T.38

Two existing non-native audio sessions, one on IPM-A and the other on IPM-B, are bridged by the application. The application receives a re-INVITE from the remote terminal associated with the session on IPM-A to modify the media to T.38; this re-INVITE is passed on to the remote terminal associated with the session on IPM-B. The remote terminal associated with IPM-B accepts the re-INVITE. The application stops the audio sessions on both IPM-A and IPM-B, and terminates the audio bridge connection between IPM-A and IPM-B. The application then establishes native T.38 fax sessions on both IPM-A and IPM-B, and creates a full-duplex native T.38 hairpin connection between IPM-A and IPM-B. The application then sends the 200 OK to the remote terminal associated with the session on IPM-A.

The sequence of activities is as follows:

1. There is an existing audio call bridged between IPM-A and IPM-B.
2. A re-INVITE for the T.38 FAX-A call is received by the application for IPM-A.
3. A re-INVITE for the T.38 FAX-B call is sent by the application for IPM-B.
4. A 200 OK is received for the T.38 FAX-B call.
5. Native audio full-duplex connections are established between IPM-A and IPM-B using **dev_PortConnect( )**; see Section 15.2.2, “Connecting Devices”, on page 75.
6. **ipm_Stop( )** is called for IPM-B to end the full-duplex audio session for call B.
7. The existing audio bridge connection between IPM-A and IPM-B is disconnected.
8. **ipm_Stop( )** is called for IPM-A to end the full-duplex audio session for call A.
9. Native audio full-duplex connections are established between IPM-A and IPM-B using **dev_PortConnect( )**; see Section 15.2.2, “Connecting Devices”, on page 75.
10. **ipm_StartMedia( )** is called for IPM-B to establish the full-duplex native T.38 session for call B.
11. **ipm_StartMedia( )** is called for IPM-A to establish the full-duplex native T.38 session for call A.
12. A 200 OK is sent for the T.38 FAX-A call.

Using the Selective Packet Filtration Method 16

This chapter describes the selective packet filtration method.

16.1 Feature Description

Selective packet filtration method is not supported on all releases. For support information, see Chapter 2, “Feature Support by Platform”.

With the selective packet filtration method, an application has the ability to filter incoming RTP data based on the remote IP address and port information specified for the RTP session in use.

This feature allows Dialogic® HMP software to ignore all RTP data which does not originate from the remote IP address and port specified in the **ipm_StartMedia( )** function. For example, endpoint A may not stop sending RTP data immediately after receiving a request from the application to terminate the RTP session. This feature allows Dialogic® HMP software to ignore those packets from endpoint A.

The selective filtration method (also called remote terminal IP address verification mode) works on a port and address information pair to either accept or reject a certain RTP packet. If the parameter is enabled, the filter is applied.

When this mode is enabled, Dialogic® HMP software assumes that the IP address and port agreed upon in the call-setup phase as the RTP origin is also the source of all of the RTP packets. For example, if a packet received in the port assigned to this RTP session does not have the origin’s IP address and port, the Dialogic® HMP software will reject it.

Caution: Enabling the selective filtration method violates the requirements of IETF SIP standards, since RFC 3264 calls for an RTP receiver NOT to tie itself exclusively to the negotiated origin’s IP address. For this reason, the selective filtration method should only be used when absolutely necessary.

16.2 API Library Support

To specify the selective filtration method on a board basis, use the **ipm_SetParm( )** function and set the PARMBD_RTP_SOURCE_FILTER parameter in the IPM_PARM_INFO structure. Use 1 to enable filtration and 0 to disable filtration.

The following example shows how to enable the selective filtration method:

```
Setting up the parameter int value=1;
IPM_PARM_INFO Parms;
Parms.eParm=PARMBD_RTP_SOURCE_FILTER ;
Parms.pvParmValue=&value;

rc=ipm_SetParm(brd1,&Parms,EV_SYNC);
```

It is possible to set this mode in the *Hmp.Uconfig* file. For more information, see the *Dialogic® System Configuration Guide*.

Quality of Service (QoS) Alarms and RTCP Reports 17

This chapter describes the QoS alarms and RTCP reports that are supported by the Dialogic® IP Media Library API software. The following topics are discussed:

17.1 QoS Overview

Quality of Service (QoS) alarms are not fully supported on all releases. For support information, see Chapter 2, "Feature Support by Platform".

The public switched telephone network (PSTN) defines quality of service as a particular level of service, for example "toll-like" service. However, quality of service for voice or other media over the Internet Protocol is defined as a continuum of levels, which are affected by packet delay or loss, line congestion, and hardware quality such as microphone quality. The Dialogic® IP Media Library API software is designed to operate along the entire range of quality of service, enabling the application to retrieve information necessary for correct billing.

All QoS parameters supported by the Dialogic® IP Media Library API software are disabled by default. That is, QoS monitoring must be enabled by the application. If desired, the application can set threshold values to monitor the quality of service during sessions. The QoS parameters are measured during time intervals, starting when a session is established. A fault occurs when the measurement of a QoS parameter exceeds a predefined threshold. A recovery occurs when the measurement of a QoS parameter returns to a value that does not exceed the predefined threshold.

To enable and use QoS monitoring in your application, you must follow several steps. Some steps are optional; others are required. These steps are detailed in Section 17.5, "Implementing QoS Alarms", on page 87.

17.2 QoS Alarm Types

All QoS alarms operate on a per-channel basis. That is, a QoS alarm indicates the status of a particular channel during a particular session, not the status of an entire Dialogic® IP media resource board.

The following QoS alarm types are supported in the Dialogic® IP Media Library API software. These names are used in the IPM_QOS_THRESHOLD_DATA structure when setting parameters for the alarms, and in the IPM_QOS_ALARM_DATA structure that is associated with the IPMEV_QOS_ALARM event that is generated when an alarm state transition occurs.

QOSTYPE_JITTER
: QoS alarm for excessive average jitter

QOSTYPE_LOSTPACKETS
: QoS alarm for excessive percentage of lost packets

QOSTYPE_RTCPTIMEOUT
: QoS alarm for RTCP timeout, indicating that RTCP packets are no longer being received. This alarm can also indicate that the network cable is disconnected.

QOSTYPE_RTPTIMEOUT
: QoS alarm for RTP timeout, indicating that RTP packets are no longer being received. This alarm can also indicate that the network cable is disconnected.

QOSTYPE_RTCP_SCS
: QoS alarm for RTCP severely concealed second condition (SCS)

QOSTYPE_RTCP_JB_HIGH
: QoS alarm for RTCP jitter buffer above the threshold

QOSTYPE_RTCP_JB_LOW
: QoS alarm for RTCP jitter buffer below the threshold

The following QoS alarms have been defined for Secure RTP:

QOSTYPE_SEC_AUTH_FAIL_AUDIO
: Secure RTP QoS alarm for authentication failure on RTP audio packets

QOSTYPE_SEC_AUTH_FAIL_VIDEO
: Secure RTP QoS alarm for authentication failure on RTP video packets

QOSTYPE_SEC_PKT_REPLAY_AUDIO
: Secure RTP QoS alarm for replay detection of audio packets

QOSTYPE_SEC_PKT_REPLAY_VIDEO
: Secure RTP QoS alarm for replay detection of video packets

QOSTYPE_SEC_MKI_NOMATCH_AUDIO
: Secure RTP QoS alarm for Master Key Identification (MKI) mis-match on audio packets

QOSTYPE_SEC_MKI_NOMATCH_VIDEO
: Secure RTP QoS alarm for MKI mis-match on video packets

For details on using QoS alarms in your application, see Section 17.5, “Implementing QoS Alarms”, on page 87. For more information on SRTP QoS alarms, see Chapter 21, “Using Secure RTP”.

17.3 QoS Threshold Attributes

All QoS alarm types have one or more threshold attributes, such as time interval and fault threshold, which specify how the system determines when to generate a QoS alarm event.

The threshold attributes listed below are specified in IPM_QOS_THRESHOLD_DATA structures that are contained in an IPM_QOS_THRESHOLD_INFO structure that is passed to **ipm_SetQoSThreshold( )**:

unTimeInterval
: time interval between successive parameter measurements

unDebounceOn
: polling interval for detecting potential alarm fault condition. This interval must be a multiple of unTimeInterval.

unDebounceOff
: polling interval for measuring potential alarm non-fault condition. This interval must be a multiple of unTimeInterval.

unFaultThreshold
: fault threshold value. The meaning and value range of this attribute depend on the alarm type.

unPercentSuccessThreshold
: percentage of poll instances in unDebounceOff interval that the fault threshold must not be exceeded before an “alarm off” event is sent. The granularity for this attribute is the ratio of unTimeInterval to unDebounceOff, expressed as a percentage.

unPercentFailThreshold
: percentage of poll instances in unDebounceOn interval that the fault threshold must be exceeded before an “alarm on” event is set. The granularity for this attribute is the ratio of unTimeInterval to unDebounceOff, expressed as a percentage.

Note: Not all attributes are supported for all alarm types and products. All attributes that are not supported should be set to 0.

The Dialogic® IP Media Library API software provides default values for each threshold attribute that will be used if the application does not specify any threshold values via **ipm_SetQoSThreshold( )**; the specific default values vary by alarm type. See Table 6, “Quality of Service (QoS) Parameter Defaults”, on page 275 for details on the attributes supported and the default values for each QoS alarm type. Note that when an application sets a specific value for a field, it must explicitly set *all* fields in the IPM_QOS_THRESHOLD_DATA structure even when default values are desired for some of the fields.

17.4 QoS Event Types

The following QoS event types are used when calling the **ipm_EnableEvents( )** and **ipm_DisableEvents( )** functions to enable and disable the corresponding QoS alarms.

EVT_JITTER
: event indicating excessive jitter

EVT_LOSTPACKETS
: event indicating excessive percentage of lost packets

EVT_RTCPTIMEOUT
: timeout event indicating RTCP packets are no longer being received

EVT_RTPTIMEOUT
: timeout event indicating RTP packets are no longer being received

EVT_RTCP_SCS
: event used to track RTCP severely concealed second condition (SCS)

EVT_RTCP_JB_HIGH
: event used to track RTCP excessive jitter buffer

EVT_RTCP_JB_LOW
: event used to track RTCP jitter buffer below the threshold

These QoS event types correspond to the QoS alarm types discussed in Section 17.2, "QoS Alarm Types", on page 85. For details on enabling QoS alarms in your application, see the following section, "Implementing QoS Alarms".

17.5 Implementing QoS Alarms

The following steps provide general guidelines for implementing QoS alarms in your application. For details on the Dialogic® IP Media Library API functions and data structures that are mentioned, see Chapter 24, "Function Information" and Chapter 26, "Data Structures".

Note: These steps do not represent every task that must be performed to create a working application but are intended as general guidelines.

1. **Optional steps before enabling a QoS alarm:**
   a. Call **ipm_GetQoSThreshold( )** to retrieve the current settings of QoS parameters on the specified IP channel. QoS parameter default values vary by alarm type and product. For information on QoS parameter default values, see the table in Section 17.3, "QoS Threshold Attributes", on page 86.
   b. If you need to change current QoS parameter values, set up the IPM_QOS_THRESHOLD_INFO structure with desired values. This structure contains one or more IPM_QOS_THRESHOLD_DATA structures. Note that you must explicitly specify the value for *every* parameter in the IPM_QOS_THRESHOLD_DATA structure, even if you want to use the default value for some of those parameters and non-default values for other parameters.
   c. Call **ipm_SetQoSThreshold( )** to use the QoS parameter values set in step 1b.

2. **Enable QoS alarms and start media streaming:**
   a. Call **ipm_EnableEvents( )** to enable QoS monitoring for a list of alarm types.
   b. Call **ipm_StartMedia( )** to start media streaming and begin QoS monitoring.
3. **Monitor QoS alarm notification events:**
   a. When a QoS alarm has been triggered, an IPMEV_QOS_ALARM event is generated by the system. Call the Dialogic® Standard Runtime Library function **sr_getevttype( )** to return the event type.
   b. Use Dialogic® Standard Runtime Library API functions such as **sr_getevtdatap( )** to query the IPM_QOS_ALARM_DATA structure to learn whether the alarm state is on or off.

   Note: For the Dialogic® Host Media Processing (HMP) Software, the system sends a QoS alarm event containing ALARM_STATE_ON when the fault threshold is exceeded and sends a QoS alarm event containing ALARM_STATE_OFF when the threshold returns to the programmed level.
4. **Perform clean-up activities:**
   a. Call **ipm_Stop( )** to stop media streaming.
   b. Call **ipm_DisableEvents( )** to stop QoS parameter monitoring.

For example code that illustrates how to implement QoS alarms, see Section 17.7, "Example Code for QoS Alarm Handling", on page 92.

17.6 QoS Alarm and Alarm Recovery Mechanisms

The information in this section does not apply to the RTP timeout and RTCP timeout alarm types, which do not support the debounce parameters.

To explain how the system monitors, detects, and clears a QoS alarm condition, three scenarios will be presented. In the first scenario, a QoS fault condition is detected but an alarm-on event is not sent to the application. In the second scenario, the QoS fault condition meets all alarm criteria and an alarm-on event is sent. The third scenario expands on the second scenario and describes how the alarm-on condition is cleared.

These scenarios are intended to illustrate the concepts. For easier reference, in the figures, time is shown in seconds rather than in millisecond units.

In the three scenarios, the jitter alarm type is being monitored. The QoS parameters (alarm threshold attribute values) used in these scenarios are:

- unTimeInterval = 1000 ms (1 second)
- unDebounceOn = 4000 ms (4 seconds)
- unDebounceOff = 4000 ms (4 seconds)
- unFaultThreshold = 60 milliseconds
- unPercentFailThreshold = 50 percent
- unPercentSuccessThreshold = 50 percent

From these parameters, the library calculates "count" values for alarm-on and alarm-off debouncing that represent the number of measurements that must fail (or succeed) within a unTimeInterval period before an alarm-on (or alarm-off) event is generated.

For alarm-on debouncing:
count = int((**unDebounceOn/unTimeInterval**) * (**unPercentFailThreshold**/100))
= int((4000/1000) * (50/100))
= int(4 * 0.5)
= 2

For alarm-off debouncing:
count = int((**unDebounceOff/unTimeInterval**) * (**unPercentSuccessThreshold**/100))
= int((4000/1000) * (50/100))
= int(4 * 0.5)
= 2

For example code that uses these QoS parameter values, see Section 17.7, "Example Code for QoS Alarm Handling", on page 92.

Scenario 1: Brief Alarm Condition

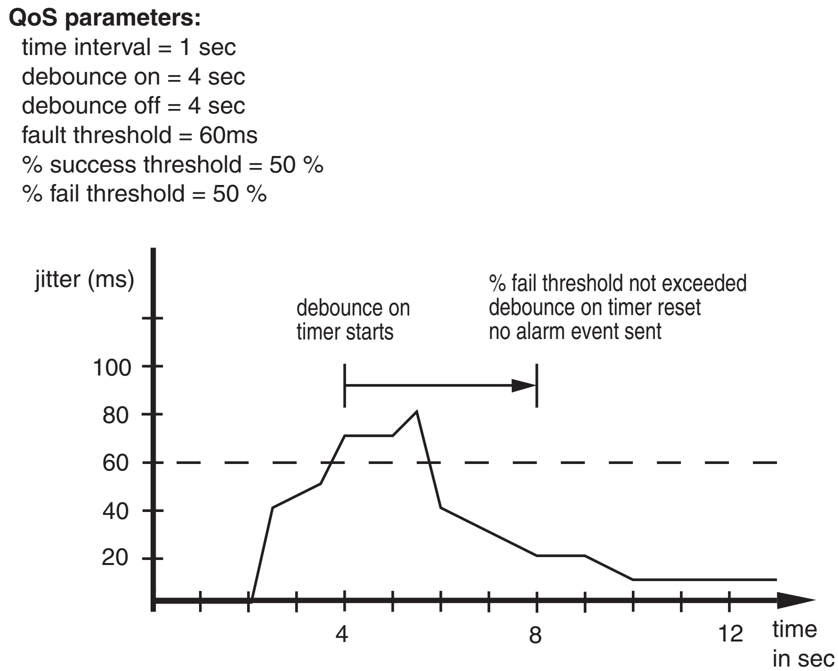
This scenario illustrates that a QoS alarm is triggered, but the alarm condition does not meet all of the specified alarm criteria. An alarm-on event is not sent to the application.

In Figure 14, the time line shows that QoS parameters are measured every time interval (**unTimeInterval** parameter), or every 1 second in this case. When the jitter exceeds the 60ms fault threshold (**unFaultThreshold** parameter), the debounce on timer is kicked off (**unDebounceOn** parameter). In this example, the fault threshold is exceeded at the 4th second.

To determine if this is a true alarm condition, the system continues to monitor the jitter in blocks of 4 seconds (**unDebounceOn** parameter), the debounce on window. If the jitter is below the 60ms fault threshold for more than 50 percent of the time (**unPercentFailThreshold** parameter) in a 4-second block, an alarm-on event is not sent to the application.

In this example, at the end of the 4-second debounce on window (at the 8th second), the percent failure threshold measured is 25 percent; that is, the fault threshold only exceeded the desired fault threshold of 60ms at the 5th second measurement within the 4-second debounce on window. Since the desired percentage failure threshold of 50 percent was not met or exceeded, no alarm-on event is sent to the application. At the end of the 8th second, the debounce on timer is reset.

Figure 14. QoS Scenario 1: Brief Alarm Condition



Scenario 2: True Alarm Condition

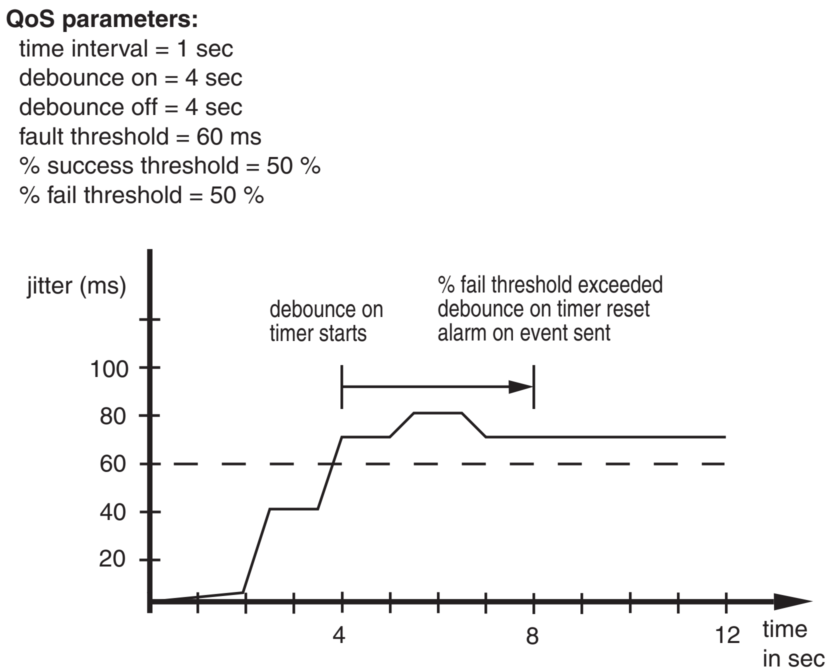
This scenario illustrates that a QoS alarm is triggered, and the alarm condition meets all of the specified alarm criteria. Therefore, an alarm-on event is sent to the application.

In Figure 15, the time line shows that QoS parameters are measured every time interval (**unTimeInterval** parameter), or every 1 second in this case. When the jitter exceeds the 60ms fault threshold (**unFaultThreshold** parameter), the debounce on timer is kicked off (**unDebounceOn** parameter). In this example, the fault threshold is exceeded at the 4th second.

To determine if this is a true alarm condition, the system continues to monitor the jitter in blocks of 4 seconds, the debounce on window (**unDebounceOn** parameter). If the jitter exceeds the 60ms fault threshold for more than 50 percent of the time (**unPercentFailThreshold** parameter) in a 4-second block, an alarm-on event is sent to the application.

In this example, at the end of the 4-second debounce on window (at the 8th second), the percent failure threshold measured is 100 percent; that is, the fault threshold exceeded the desired fault threshold of 60ms at the 5th, 6th, 7th and 8th second measurement within the 4-second debounce on window. Since the desired percentage failure threshold of 50 percent was exceeded, an alarm-on event is sent to the application. At the end of the 8th second, the debounce on timer is reset. See Scenario 3: Alarm Condition Cleared to learn how the system continues to monitor the jitter QoS alarm.

Figure 15. QoS Scenario 2: True Alarm Condition



Scenario 3: Alarm Condition Cleared

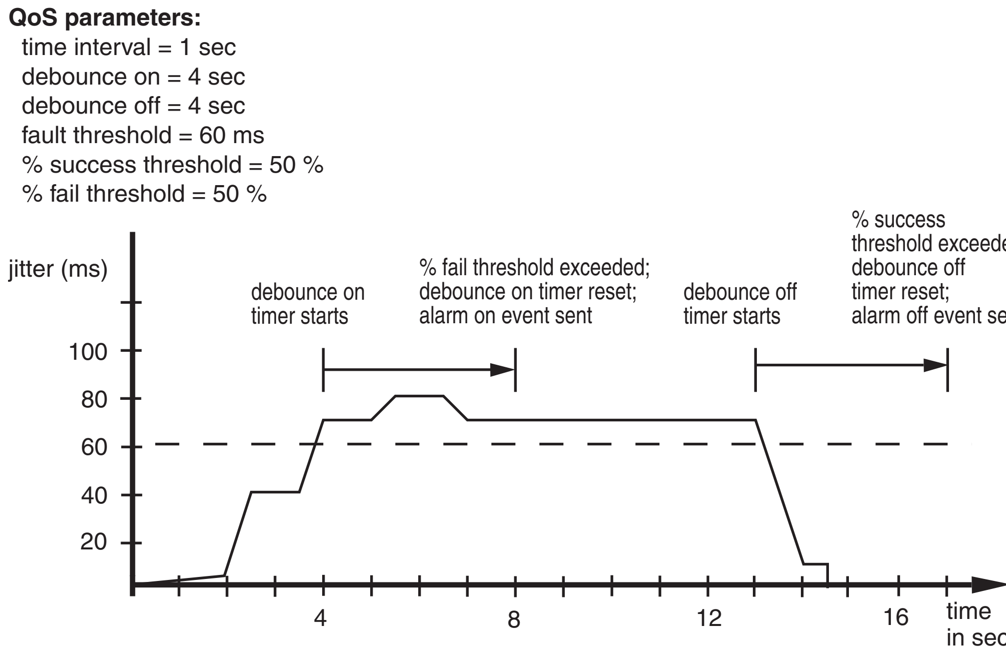
Scenario 3 builds on Scenario 2 to illustrate what happens after an alarm-on event is sent to the application and how the alarm-on condition is cleared.

In Figure 16, an alarm-on event was sent to the application at the 8th second, and the system is now in a QoS failure condition. To determine how long this condition will last, the system resumes monitoring the jitter every time interval (**unTimeInterval** parameter), or every 1 second in this case. When the jitter is less than the 60ms fault threshold (**unFaultThreshold** parameter), the debounce off timer kicks in (**unDebounceOff** parameter). In this example, this condition occurs at the13th second.

To determine if this is a true success condition, the system monitors the jitter in blocks of 4 seconds, the debounce off window (**unDebounceOff** parameter). If the jitter is below the 60ms fault threshold for more than 50 percent of the time (**unPercentSuccessThreshold** parameter) in a 4-second block, an alarm-off event is sent to the application.

In this example, at the end of the 4-second debounce off window (at the 17th second), the percent success threshold measured is 100 percent; that is, the jitter level was below the desired fault threshold of 60ms at the 14th through 17th second measurement within the 4-second debounce off window. Since the desired percentage success threshold of 50 percent was exceeded, an alarm-off event is sent to the application. At the end of the 17th second, the debounce off timer is reset.

Figure 16. QoS Scenario 3: Alarm Condition Cleared



17.7 Example Code for QoS Alarm Handling

The following pseudocode illustrates how you might use QoS alarms in an application. The code enables the following QoS alarm types: jitter, lost packets, RTCP timeout, and RTP timeout. Because the IPM_QOS_THRESHOLD_INFO structure is not filled in for the lost packets alarm type, the default QoS parameter values are used for this alarm. The QoS parameter values for jitter are the same values used in the scenario descriptions in Section 17.6, “QoS Alarm and Alarm Recovery Mechanisms”, on page 88.

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
   int nDeviceHandle;
   IPM_QOS_THRESHOLD_INFO mySetQosThresholdInfo;

   const int nNumEvent = 4;
   eIPM_EVENT myEvents[nNumEvent] ={EVT_LOSTPACKETS,
                                    EVT_JITTER,
                                    EVT_RTPTIMEOUT,
                                    EVT_RTCPTIMEOUT};
```

```
// Register event handler function with srl
sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);
/*
.
.
Main Processing
.
.
.
*/

/*
.
.
    The application can call ipm_GetQoSThreshold() to check the current
    threshold levels for QoS parameters.
.
.
*/

// Change alarm threshold settings for IP device handle, nDeviceHandle.
// ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
// Note:
// 1. You don't have to change all QoS types. In the example below, the lost packet
//    values are not changed.
// 2. For RTP Timeout and RTCP Timeout, the values of all parameters EXCEPT
//    unTimeInterval and unFaultThreshold must be set to ZERO
mySetQosThresholdInfo.unCount = 3;
mySetQosThresholdInfo.QoSThresholdData[0].eQoSType = QOSTYPE_JITTER;
mySetQosThresholdInfo.QoSThresholdData[0].unTimeInterval = 1000; //1sec
mySetQosThresholdInfo.QoSThresholdData[0].unDebounceOn = 4000; //4sec
mySetQosThresholdInfo.QoSThresholdData[0].unDebounceOff = 4000;//4sec
mySetQosThresholdInfo.QoSThresholdData[0].unFaultThreshold = 60;//60ms
mySetQosThresholdInfo.QoSThresholdData[0].unPercentSuccessThreshold = 50;//50%
mySetQosThresholdInfo.QoSThresholdData[0].unPercentFailThreshold = 50;//50%
mySetQosThresholdInfo.QoSThresholdData[1].eQoSType = QOSTYPE_RTPTIMEOUT;
mySetQosThresholdInfo.QoSThresholdData[1].unTimeInterval = 1000;//1sec
mySetQosThresholdInfo.QoSThresholdData[1].unDebounceOn = 0;
mySetQosThresholdInfo.QoSThresholdData[1].unDebounceOff = 0;
mySetQosThresholdInfo.QoSThresholdData[1].unFaultThreshold = 600; //60sec timeout
mySetQosThresholdInfo.QoSThresholdData[1].unPercentSuccessThreshold = 0;
mySetQosThresholdInfo.QoSThresholdData[1].unPercentFailThreshold = 0;
mySetQosThresholdInfo.QoSThresholdData[2].eQoSType = QOSTYPE_RTCPTIMEOUT;
mySetQosThresholdInfo.QoSThresholdData[2].unTimeInterval = 1000;//1sec
mySetQosThresholdInfo.QoSThresholdData[2].unDebounceOn = 0;
mySetQosThresholdInfo.QoSThresholdData[2].unDebounceOff = 0;
mySetQosThresholdInfo.QoSThresholdData[2].unFaultThreshold = 150; //15sec timeout
mySetQosThresholdInfo.QoSThresholdData[2].unPercentSuccessThreshold = 0;
mySetQosThresholdInfo.QoSThresholdData[2].unPercentFailThreshold = 0;

if(ipm_SetQoSThreshold(nDeviceHandle, &mySetQosThresholdInfo, EV_SYNC) == -1)
{
    printf("ipm_SetQoSThreshold failed for device name = %s with error = %d\n",
                ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
    /*
    .
    .
    Perform Error Processing
    .
    .
    */
}
```

```
    // Call ipm_EnableEvent to be notified of possible alarm conditions.

    if(ipm_EnableEvents(nDeviceHandle, myEvents, nNumEvent, EV_SYNC) == -1)
    {
        printf("ipm_EnableEvents failed for device name %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
            /*
            .
            .
                Perform Error Processing
            .
            .
        */
    }

    /*
    .
    . Continue Processing
    .
    */

    // Appplication can disable events if it does not want to be notified.

    if(ipm_DisableEvents(nDeviceHandle, myEvents, nNumEvent, EV_SYNC) == -1)
    {
        printf("ipm_DisableEvents failed for device name %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }

    if(ipm_Close(nDeviceHandle, NULL) == -1)
    {
        printf("----------->ipm_Close() failed for handle = %d\n", nDeviceHandle);
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }
}

void CheckEvent()
{
    int nEventType = sr_getevttype();
    int nDeviceID = sr_getevtdev();
    void *pVoid = sr_getevtdatap();

    switch(nEventType)
    {
    /*
    .
    . List of expected events
    .
    */

    /* When alarm occurs you get this event. */
    case IPMEV_QOS_ALARM:
        {
```

```
                printf("Received IPMEV_QOS_ALARM for device = %s\n",
                    ATDV_NAMEP(nDeviceID));
                IPM_QOS_ALARM_DATA * l_pAlarm = (IPM_QOS_ALARM_DATA*)pVoid;
                switch(l_pAlarm->eQoSType)
                {
                case QOSTYPE_JITTER:
                    printf("Alarm Type = Jitter\n");
                    break;
                case QOSTYPE_LOSTPACKETS:
                    printf("Alarm Type = LostPackets\n");
                    break;
                case QOSTYPE_RTPTIMEOUT:
                    printf("Alarm Type = RTPTimeout\n");
                    break;
                case QOSTYPE_RTCPTIMEOUT:
                    printf("Alarm Type = RTCPTimeout\n");
                    break;
                }
                printf("Alarm state = %s\n", (l_pAlarm->eAlarmState? "On": "Off"));
                break;
            }
        /*
        .
        .
            process other cases.
        .
        .
        */

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
        }
    }
```

17.8 RTCP Reporting

RTCP reporting is discussed in the following topics:

- Basic RTCP Reports
- Enhanced RTCP Reports
- Retrieving an RTCP Report
- Enabling RTCP Reporting System-Wide

17.8.1 Basic RTCP Reports

Basic RTCP reports are obtained by calling **ipm_GetSessionInfo( )**. The RTCP information is represented in the IPM_RTCP_SESSION_INFO portion of the IPM_SESSION_INFO structure passed to the function. The data returned is parsed into data structure elements.

17.8.2 Enhanced RTCP Reports

Enhanced RTCP reports are not supported on all releases. For support information, see Chapter 2, “Feature Support by Platform”.

The Dialogic® IP Media library supports Quality of Service (QoS) alarms and enhanced RTCP reports (RTCP-XR) compliant with RFC 3611 for RTCP Extended Reports and IETF draft RTCP High Resolution VoIP Metrics Report Blocks (RTCP-HR). The latest IETF draft as of this writing is at: http://tools.ietf.org/html/draft-ietf-avt-rtcphr-03.

Enhanced RTCP reporting adds extended and high resolution VoIP metrics to the RTCP packets sent by Dialogic® HMP Software. The new packets convey information beyond that already contained in the reception report blocks of RTCP's Sender Report and Receiver Report packets.

Each enhanced RTCP packet consists of the following:

Basic report blocks as defined by RFC 3550
: These include Sender Report, Receiver Report, and SDES.

An extended report block (XR) as defined by RFC 3611
: This includes the RTCP-XR header block and a set of RTCP-HR defined blocks. The RTCP-HR block includes sub-blocks such as Concealed Seconds metrics, Basic Loss/Discard metrics, Delay/PVD metrics, and Playout metrics. No other block types are supported by this feature.

With enhanced RTCP reporting, it is useful to monitor the following RTCP statistics and raise alarms when they exceed the allowable threshold:

Severely concealed second condition (SCS)
: A severely concealed second condition is defined as a non-overlapping period of one second, during which the cumulative amount of time that has been subject to frame loss exceeds the SCS Threshold of 15 percent. In other words, this refers to any one-second period that has more than 15% of lost data. EVT_RTCP_SCS is the alarm event used to track this condition.

RTCP jitter buffer
: An alarm can be raised when jitter buffer exceeds or falls below the threshold. EVT_RTCP_JB_HIGH and EVT_RTCP_JB_LOW are the alarm events used to track RTCP jitter buffer.

The **ipm_GetSessionInfoEx( )** function enables an application to retrieve the raw data of transmitted or received RTCP packets (which include the old and the new blocks). A separate RTCP parser library is provided and can be used to extract both basic and extended RTCP report information.

17.8.3 Retrieving an RTCP Report

To retrieve an RTCP report, follow these steps:

1. Enable enhanced RTCP reporting by calling **ipm_SetParm( )** with the parameter PARMCH_RTCP_ENHANCED_REPORTING.
2. Specify the frequency of enhanced RTCP reporting events using **ipm_SetParm( )** with the parameter PARMCH_RTCP_ENHANCED_EVENT_FREQ set to a non-zero value.
3. Enable the unsolicited events using the **ipm_EnableEvents( )** function. Enable these event types: EVT_RTCP_REPORT_RECEIVED and EVT_RTCP_REPORT_SENT.

4. When one of the unsolicited events, IPMEV_RTCP_NOTIFY_RECEIVED or IPMEV_RTCP_NOTIFY_SENT, is received by the application, call **ipm_GetSessionInfoEx( )** with the appropriate direction specified.
5. Analyze the IPM_SESSION_INFOEX data structure associated with the completion event IPMEV_GET_SESSION_INFOEX.
6. Use the RTCP parser library provided in *usr/dialogic/demos/rtcpparser* to parse the contents of the compound RTCP packet into individual packets. How-to information for parsing the raw data is also provided.

17.8.4 Enabling RTCP Reporting System-Wide

To enable enhanced RTCP reporting on a system-wide basis, set the values in the [IPVSC] section of the CONFIG file, run fcdgen utility, and then restart the system.

Note: To preserve custom data, it is recommended that you set the custom values in the *HMP.Uconfig* file rather than modifying the CONFIG file directly. For details, see the “Preserving Data in User Configuration Files” topic in the *Dialogic® System Configuration Guide*.

RTCP Enhanced Reporting

Number: 0x401f
Description: Enables transmission of the RTCP-XR (extended) and RTCP-HR (high resolution) blocks in RTCP packets.
Values: 0 for off and 1 for on. Default value: 0

RTCP Event Frequency

Number: 0x4020
Description: Controls how often RTCP reporting events are sent to the application. For example, when set to 5, every fifth RTCP reporting event is sent to the application.
Values: 0 to 255. Default value: 0 (don’t send reporting event).

The following example shows how these parameters are added in the *HMP.Uconfig* file.

```
[IPVSC]

SetParm=0x401f, 1               ! turn on RTCP-XR/HR reporting
SetParm=0x4020, 2               ! receive an event on every second RTCP message
```

See the *Dialogic® System Configuration Guide* for more information on the CONFIG file.

Volume Control 18

This chapter describes the volume adjustment feature which allows an application to adjust the volume level on a Dialogic® IP device. The following topics are covered:

18.1 Volume Control Overview

The Dialogic® IP Media Library API provides the ability to adjust the volume of an inbound and outbound call on a Dialogic® IP device. This volume adjustment value is specified for an IP channel device through the API; possible values are from -32 dB to +31 dB in increments of 1 dB.

The volume adjustment value is a relative change to the nominal value. For example, if the original volume level on a call is 20 dB, then to reduce the volume, you could specify an adjustment value of -6 dB; the volume level on the call would then be 14 dB. To increase the volume, you could specify an adjustment value of +8 dB; the volume level on the call would then be 28 dB. Subsequently, to readjust the volume to 26 dB, you must specify +6 dB. This adjustment is relative to the original nominal value of 20 dB.

18.2 Volume Control Parameters

The **ipm_SetParm( )** function is used to specify the volume adjustment for a Dialogic® IP device in your application. The **ipm_GetParm( )** function returns the value of the volume adjustment for a given Dialogic® IP device. If no volume adjustment has been made, this function returns a zero for the volume adjustment parameters. Both of these functions use the IPM_PARM_INFO structure.

The following parameter types (specified in the IPM_PARM_INFO structure eParm field) are used to adjust the volume level of a call on a Dialogic® IP device:

- PARMCH_RX_ADJVOLUME to adjust the volume level for the inbound side (from IP) of a call
- PARMCH_TX_ADJVOLUME to adjust the volume level for the outbound side (to IP) of a call

18.3 Implementing Volume Control

To implement volume control for a Dialogic® IP device in your application, follow these steps:

Note: These steps do not represent every task that must be performed to create a working application but are intended as general guidelines.

1. Determine the volume adjustment necessary for the Dialogic® IP device; for example, based on your experience with equipment from a particular vendor.
2. Adjust the volume level for the inbound side (from IP) as needed using **ipm_SetParm( )** and the PARMCH_RX_ADJVOLUME parameter in IPM_PARM_INFO structure.
3. Adjust the volume level for the outbound side (to IP) as needed using **ipm_SetParm( )** and the PARMCH_TX_ADJVOLUME parameter in IPM_PARM_INFO structure.
4. Perform streaming activity using **ipm_StartMedia( )**.

   Note: Typically, you adjust the volume level *before* performing a streaming activity over the IP network. However, you can issue the **ipm_SetParm( )** function to change the volume level during an active call.
5. If desired, check the current value of volume level adjustment for a Dialogic® IP device using **ipm_GetParm( )**.
6. If desired, reset the volume to its original value (that is, no adjustment) at call termination using **ipm_SetParm( )** and either PARM_RX_ADJVOL_DEFAULT or PARM_TX_ADJVOL_DEFAULT.

18.4 Volume Control Hints and Tips

The following hints and tips are provided to help you use the volume control feature in your application:

- The volume adjustment value (specified in PARMCH_RX_ADJVOLUME or PARMCH_TX_ADJVOLUME) is applied per IP channel device.
- The volume adjustment value for a Dialogic® IP device remains in effect until it is explicitly changed in the application. Terminating the call or closing the device will not reset the volume level to its default value.
- The adjustment levels specified are absolute values. Each invocation will change the adjustment level to its new value.

18.5 Example Code for Volume Control

The following example illustrates the use of the PARMCH_TX_ADJVOLUME value to decrease the volume by 6 dB for the outbound side of an IP call.

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
```

```
void CheckEvent();

void main()
{
   int nDeviceHandle;
   // Register event handler function with the standard runtime library (SRL)
   sr_enbhdlr( EV_ANYDEV, EV_ANYEVT, (HDLR)CheckEvent);

   /*
   .
    main processing
   .
   */

   /*
    Need to enable three events for IP device handle, nDeviceHandle.
    ASSUMPTION:  A valid nDeviceHandle was obtained from prior call to ipm_Open().
   */
    IPM_PARM_INFO     parmInfo;
    int               parmValue = -6; // decrease nominal volume by 6 dB
    parmInfo.eParm = PARMCH_TX_ADJVOLUME;
    parmInfo.pvParmValue = &ParmValue;
    if ipm_SetParm(nDeviceHandle, &parmInfo, EV_ASYNC) == -1)
    {
      /*
      .
      .Perform error processing
      .
      .
      */
   }

   /*
   .
   . Start media streaming with ipm_StartMedia( )
   .
   */

    // Reset Volume adjust to the channel
    IPM_PARM_INFO  parmInfo;
    int            parmValue = PARM_TX_ADJVOL_DEFAULT;
    parmInfo.eParm = PARMCH_TX_ADJVOLUME;
    parmInfo.pvParmValue = &ParmValue;
    if ( ipm_SetParm(nDeviceHandle, &parmInfo, EV_ASYNC) == -1)
             printf("%s: ipm_SetParm failed.\n", ATDV_NAMEP(nDeviceHandle));
    else
             printf(""%s:  Transmit Volume adjustment has been Reset successfully.\n",
                   ATDV_NAMEP(nDeviceHandle));
    }

    void CheckEvent()
    {
         int nEventType = sr_getevttype();
         int nDeviceID = sr_getevtdev();
         void* pVoid = sr_getevtdatap();

         switch(nEventType)
         {
             .
             .
             case IPMEV_SET_PARM:
                  IPM_PARM_INFO parmInfo;
                  int parmValue = 0;
                  parmInfo.eParm = PARMCH_TX_ADJVOLUME;
                  parmInfo.pvParmValue = &ParmValue;
```

```
                ipm_GetParm(nDeviceHandle, &parmInfo, EV_SYNC) ;
                printf("Outbound Volume for device = %s adjusted",
                       "by = %d db.\n", ATDV_NAMEP(nDeviceID),
                        parmInfo.ParmValue);
                break;

           Default:
                Printf("Received unknown event = %d for device = %s\n",
                       nEventType, ATDV_NAMEP(nDeviceID));
                break;
       }
  }
```

Using Echo Cancellation 19

This chapter describes the Dialogic® IP Media Library API support for echo cancellation, a feature that reduces traces of an output signal (echo) from an incoming signal.

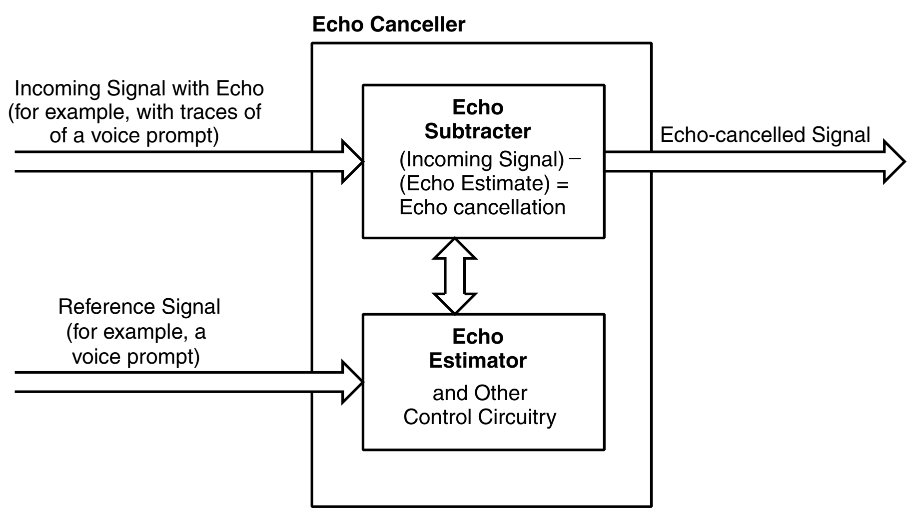
19.1 Overview of Echo Cancellation

Echo cancellation is not supported on all releases. For support information, see Chapter 2, "Feature Support by Platform".

Echo cancellation is a technique used to significantly reduce traces of an outgoing prompt in the incoming signal. These traces are referred to as echo. The echo canceller is the component in the Dialogic® Host Media Processing (HMP) Software responsible for performing echo cancellation.

Figure 17 demonstrates how the echo canceller works. After the incoming signal is processed by the echo canceller, the resulting signal no longer has significant echo and is then sent to the host application.

Figure 17. Echo Cancellation



If echo cancellation is not used, the incoming signal usually contains an echo of the outgoing prompt. Without echo cancellation, an application must ignore all incoming energy until the prompt and its echo terminate.

With echo cancellation, the caller may interrupt the prompt, and the incoming speech signal can be passed to the application.

Echo cancellation may be required when Dialogic® IP media resources receive media streams from the PSTN via the DTI resources of the T1/E1 single span (DNI300TEPHMP) or the T1/E1 quad span (DNI1200TEPHMP) boards. Echo cancellation is not required for media streams received from IP connections, PSTN connections via the T1/E1 dual span (DNI601TEPHMP) board, which provides onboard echo cancellation as part of the DTI resources, or via the DSI162 digital station interface boards.

Echo cancellation on an IP device can be monitored and controlled at run time using the **ipm_GetParm( )** and **ipm_SetParm( )** functions in the Dialogic® IP Media Library API.

19.2 Echo Cancellation Parameters

The **ipm_GetParm( )** and **ipm_SetParm( )** functions use the IPM_PARM_INFO data structure to retrieve and set parameters in general. For echo cancellation, the following parameters are supported:

PARMCH_ECACTIVE
: Enables or disables echo cancellation on an IP device. Possible values are:
- 0 - Disable echo cancellation (default)
- 1 - Enable echo cancellation

PARMCH_ECHOTAIL
: Set or retrieve the echo tail length. The duration of an echo is measured in milliseconds. The echo canceller software can remove some limited number of these milliseconds; this number is known as the tail length of the echo canceller. The longer the tail length, the more echo is cancelled from the incoming signal, but this means more processing power is required. When determining the tail length value, consider the length of the echo delay in the system as well as the overall system configuration. Possible values are in the range 8 to 64 milliseconds (the default is 64 milliseconds).

PARMCH_ECNLP_ACTIVE
: Set or retrieve the Non-Linear Processing (NLP) value. NLP is a process used to block or suppress the residual (echo-cancelled) signal, when there is no near end speech. This process can be used with comfort noise generation (CNG) to produce background noise. Background noise energy estimation is used to adjust the level of comfort noise generated. This allows the speaker to listen to the same level of background noise when the non-linear processor is switched on and off due to double-talk situations or near end speech. A typical usage of this feature is background noise used in dictation applications to let the user know that the application is working. Possible values are:
- 0 - NLP off
- 1 - NLP on (default)

Using NAT Traversal in SIP Media Session 20

This chapter describes Network Address Translation (NAT) in a SIP media session, which enables an application to send RTP/RTCP packets to the correct destination in a NAT environment.

20.1 Feature Description

NAT traversal in SIP media session is not supported on all releases. For support information, see Chapter 2, “Feature Support by Platform”.

NAT is a well-known feature built into many small office or home office networks for security purposes and allows multiple host computers to access the internet through a single public IP address.

In a NAT-enabled environment, the destination information specified in the SDP of the signaling packet cannot be used to send the RTP packet. This is the internal address/port information of the endpoint. Currently Dialogic® HMP software applications have no knowledge of the correct destination information for the RTP/RTCP packets in a NAT-enabled network environment, other than those that come through the SDP.

When enabled, this feature notifies the application of the correct destination information of the originating RTP or RTCP packet. The application uses this information to direct the RTP/RTCP packets appropriately. In order to redirect the packets to the correct destination, the application issues the **ipm_ModifyMedia( )** function with the correct destination information.

20.2 API Library Support

To use the NAT traversal feature and to receive the IPMEV_NOTIFY_ENDPOINTID event that contains endpoint information, enable the EVT_ENDPOINTID_NOTIFY event type through the **ipm_EnableEvents( )** function. The event payload contains the endpoint identification information of the incoming RTP/RTCP packet. The IPMEV_NOTIFY_ENDPOINTID event data is contained in the IPM_ENDPOINTID_INFO structure.

20.3 Example Code 1 for NAT Traversal

This example demonstrates how to enable the EVT_ENDPOINTID_NOTIFY event type.

```
#include <stdio.h>
#include <stdlib.h>
#include "ipmlib.h"
#include "srllib.h"
typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()

{
int nDeviceHandle;
eIPM_EVENT myEvents[1] ={ EVT_ENDPOINTID_NOTIFY };

// Register event handler function with srl
sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);
/*
 .
 .
 Main Processing
 .
 .
 */

/*
 * Need to enable event for IP device handle, nDeviceHandle.
 * ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
 */

if(ipm_EnableEvents(nDeviceHandle, myEvents, 1, EV_ASYNC) == -1)
{
        printf("ipm_EnableEvents failed for device name %s with error = %d\n",
               ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
}

}
void CheckEvent()
{
int nEventType = sr_getevttype();
int nDeviceID = sr_getevtdev();
switch(nEventType)
{
/*
 .
 .
 . List of expected events
 .
 .
 */
/* Expected reply to ipm_EnableEvents() */
   case IPMEV_EVENT_ENABLED:
       printf("Received IPMEV_EVENT_ENABLED for device = %s\n", ATDV_NAMEP(nDeviceID));
       break;
       default:
       printf("Received unknown event = %d for device = %s\n", nEventType,
              ATDV_NAMEP(nDeviceID));
   break;
}

}
```

20.4 Example Code 2 for NAT Traversal

This example demonstrates receiving the IPMEV_NOTIFY_ENDPOINTID event and issuing the **ipm_ModifyMedia( )** function.

```
#include <stdlib.h>
#include "ipmlib.h"
#include "srllib.h"

typedef long int(*HDLR)(unsigned long);
void CheckEvent();


void main()
{
/* Main Processing */
/* ipm_StartMedia() issued */
/* Media session successfully started */
}

void CheckEvent()

{
         int nEventType = sr_getevttype();
         int nDeviceID = sr_getevtdev();
         void* pVoid = sr_getevtdatap();
         IPM_ENDPOINTID_INFO* pEndpointIdInfo;
         char *temp;

         switch(nEventType)
         {
         /*
          .
          .
          . List of expected events
          .
          .
          */

         /* Expected reply to ipm_EnableEvents() */
          case IPMEV_EVENT_ENABLED:
                 printf("Received IPMEV_EVENT_ENABLED for device = %s\n",
                        ATDV_NAMEP(nDeviceID));
                 break;

          case IPMEV_NOTIFY_ENDPOINTID:
                 pEndpointIdInfo = (IPM_ENDPOINTID_INFO *)pVoid;
                 printf("Received IPMEV_ ENDPOINTID_NOTIFY for device =
                         %s\n",ATDV_NAMEP(nDeviceID));
                 printf("MediaType is%d\n", pEndpointIdInfo-> eMediaType);
                 printf("PortId is%d\n", pEndpointIdInfo->unPortId);
                 printf("SSRC is%u\n", pEndpointIdInfo->ulSSRC);

                 /*
                  * Use network function to convert the IP Address from binary form to
                  * dotted decimal form
                  */
                struct in_addr  IPADD;
                IPADD.S_un.S_addr = *((unsigned long *)&pEndpointIdInfo->ucIPAddress);
                temp = inet_ntoa(IPADD);
                printf("IP Address is %s\n", inet_ntoa(IPADD));

                if( (pEndpointIdInfo-> eMediaType == MEDIATYPE_AUDIO_REMOTE_RTP_INFO)
                         || (pEndpointIdInfo->eMediaType == MEDIATYPE_VIDEO_REMOTE_RTP_INFO) )
                {
```

```
                printf("Payload type is%d\n", pEndpointIdInfo->ucPayloadType);
                printf("Payload size is%d\n", pEndpointIdInfo->usPayloadSize);
                printf("Sequence Number is%d\n", pEndpointIdInfo->usSequenceNum);
                printf("Timestamp is%u\n", pEndpointIdInfo->ulTimeStamp);
      }

      /*
       * Send ipm_ModifyMedia() once we have determined that there is a port
       * address information change needed for our media session. The
       * application has to verify that port and address information has really
       * changed before it issues a ipm_ModifyMedia()
       */

    /*
     * Here the example shows only change in RTP information. The user can also
     * change RTCP information in the same ipm_ModifyMedia() call by assuming that
     * the RTCP port will be 1 + the RTP port or wait for another
     * IPMEV_ENDPOINTID_NOTIFY with the RTCP information.
     */
     IPM_MEDIA_INFO MediaInfo;
     MediaInfo.unCount = 1;
     MediaInfo.MediaData[0].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTP_INFO;
     MediaInfo.MediaData[0].mediaInfo.PortInfo.unPortId = pEndpointIdInfo->unPortId;
      {

                printf("ipm_Modify failed for device name = %s with error =
                       %d\n",ATDV_NAMEP(nDeviceID), ATDV_LASTERR(nDeviceID));
                /*
                 .
                 .
                 Perform Error Processing
                 */
       }
      break;

      default:
      printf("Received unknown event = %d for device = %s\n",nEventType,
               ATDV_NAMEP(nDeviceID));
      break;
      }
}
```

Using Secure RTP 21

This chapter describes the Dialogic® IP Media Library API support for Secure RTP, a feature that provides confidentiality and integrity protection for RTP and RTCP traffic. Topics include:

21.1 Overview of Secure RTP

Secure RTP is not supported on all releases. For support information, see Chapter 2, “Feature Support by Platform”.

This section gives a high-level summary of the Secure Real-time Transport Protocol (SRTP) feature. Full details are described in *The Secure Real-time Transport Protocol (SRTP) IETF publication, RFC 3711*, available at http://www.ietf.org/rfc/rfc3711.txt.

Real-time Transport Protocol (RTP) is commonly used for the transmission of real-time audio or video streams in Internet telephony applications. Secure RTP (SRTP) is an enhancement to RTP that provides confidentiality, message authentication, and replay protection for RTP and the companion Real Time Control Protocol (RTCP). “Replay protection” provides protection against an attacker who intercepts and records SRTP packets and later replays them into the packet stream in an attempt to interfere with or garble secure SRTP communications between two endpoints.

SRTP can be thought of as residing between an RTP application and the transport layer. On the sending side, SRTP intercepts an RTP packet and forwards an equivalent SRTP packet. At the receiving side, SRTP receives an SRTP packet and the equivalent RTP packet is passed up the protocol stack.

In SRTP, the payload (and padding) of RTP packets is encrypted. Many different encryption and message authentication algorithms exist, but RFC 3711 specifies the following default pre-defined algorithms:

- For encryption, the pre-defined cipher is the Advanced Encryption Standard (AES) operating in Segmented Integer Counter Mode, sometimes known simply as "Counter Mode". The HMP software does not support the f8-mode.
- For message authentication and integrity, the pre-defined authentication transform is HMAC-SHA1 as described in the *HMAC: Keyed-Hashing for Message Authentication* IETF publication, RFC 2104, available at http://www.ietf.org/rfc/rfc2104.txt.

Using these pre-defined algorithms, the encrypted (SRTP) payload size is only slightly larger than the unencrypted (RTP) payload size.

An important part of any encryption scheme is the generation of the keys used to encrypt the information. This involves the use of master keys and optionally master salt. A master key is a random bit string from which session keys (used directly in the cryptographic transforms) are derived. A master salt is also a random bit string used to provide even greater security. The Dialogic® IP Media Library API includes key generator functionality (the **ipm_SecurityGenMasterKeys( )** function) to provide these keys if necessary.

SRTP may rely on an external key management system to provide the master key and master salt (optional). Alternatively, SRTP can be used in conjunction with the following:

- SIP TLS (Session Initiated Protocol, Transport Layer Security), as described in *The TLS Protocol* IETF publication, RFC 2246, available at http://www.ietf.org/rfc/rfc2246.txt with ancillary information in the IETF draft available at http://www.ietf.org/internet-drafts/draft-gurbani-sip-tls-use-00.txt (expires August, 2006). TLS provides for its own authentication and key management, as well as encryption. TLS can be used to provide a secure way for two devices using SRTP to exchange the necessary setup information, including SRTP keys (using SDP Secure Descriptions, see below).
- Session Description Protocol (SDP) Secure Descriptions, as described in the IETF draft available at http://www.ietf.org/internet-drafts/draft-ietf-mmusic-sdescriptions-12.txt (expires March, 2006). SDP Secure Descriptions are the means by which two endpoints, communicating via SRTP, exchange the keys that enable decryption and authentication. SDP Secure Descriptions define an SDP cryptographic attribute for unicast media streams. The attribute describes a cryptographic key and other parameters that can be used to configure SRTP for a unicast media stream in either a single message or a round-trip exchange. The attribute can be used with a variety of SDP media transports. The SDP crypto attribute requires the services of TLS to secure the SDP message.

SRTP incorporates a "key derivation algorithm" that uses the master key, master salt and packet index to generate the session keys that are used directly for encryption or message authentication. The rate at which new keys are applied, that is, the "key derivation rate" can also be defined.

21.2 Generating Encryption Keys

To reduce the burden on the application developer when developing applications that use SRTP, the IP Media Library includes the **ipm_SecurityGenMasterKeys( )** function that generates master and salt keys. The generated keys can be in one of two formats 1) binary format or 2) Base64-encoded format. Once a Dialogic® IPM device is open, there are no restrictions on when the **ipm_SecurityGenMasterKeys( )** function can be called.

21.3 Starting and Modifying a Media Session that Uses Secure RTP

The **ipm_StartMedia( )** function can be used to start a media streaming session on an IP Media device and the **ipm_ModifyMedia( )** function can be used to modify the properties of an active or suspended media session. A media session can be started as an SRTP session or it can be changed from an RTP to SRTP session or vice versa during the session.

At the transmitting side, the order of the keys that are used for SRTP/SRTCP (from the remote <audio/video> receive list) is the same as the order of the input keys array. Once the lifetime of the first key expires, then the second key is used, and so on. When the lifetime of a key expires, that key is no longer in the device's context.

At the receiving side, a Master Key Identifier (MKI) in each packet indicates the key (from the local <audio/video> receive list) that needs to be used.

The **ipm_ModifyMedia( )** function can also be used to add or change security keys.

21.3.1 Key About to Expire Indication

One of the following asynchronous event is generated when an encryption key is about to expire:

- IPMEV_SEC_NOTIFY_EXPIRE_KEY_AUDIO - for the audio media type
- IPMEV_SEC_NOTIFY_EXPIRE_KEY_VIDEO - for the video media type

The generation of these events to the application can be enabled or disabled using the **ipm_EnableEvents( )** and **ipm_DisableEvents( )** functions. The defines used when enabling or disabling these events are:

- EVT_SEC_NOTIFY_EXPIRE_KEY_AUDIO
- EVT_SEC_NOTIFY_EXPIRE_KEY_VIDEO

The time interval between the generation of the event and the actual expiry of the encryption key is configurable using the **ipm_SetParm( )** function. See Section 21.4, "Retrieving and Modifying Encryption Key Expiry Notification Interval", on page 112.

21.3.2 Maximum Number of Keys

The total number of keys (audio/video, remote receive/local receive) that can be passed via the **ipm_StartMedia( )** or **ipm_ModifyMedia( )** function is limited to IPM_SECURITY_MAX_TOTAL_NUM_KEYS, which is currently set to 20.

In addition, the number of keys for any given media type (local audio, local video, remote audio or remote video) passed along with **ipm_StartMedia( )** or **ipm_ModifyMedia( )** is limited to IPM_SECURITY_MAX_NUM_KEYS, which is currently set to 10.

21.3.3 Usage Restrictions

When using the **ipm_StartMedia( )** function, if the SecurityInfo field (IPM_MEDIA structure) is used, the pParms field (IPM_SECURITY_INFO structure) is required. Also, when using the **ipm_StartMedia( )** function, the eInfoMode field (IPM_SECURITY_INFO structure) must be set to IPM_SECURITY_INFO_MODE_IMMEDIATE.

When using the **ipm_ModifyMedia( )** function, if the SecurityInfo field (IPM_MEDIA structure) is used, the pParms field (IPM_SECURITY_INFO structure) is optional and can be set to NULL. If the pParms field is NULL, the values specified using the **ipm_StartMedia( )** function are used. Also, when using the **ipm_ModifyMedia( )** function, the eInfoMode field (IPM_SECURITY_INFO structure) can be set to one of the following:

- IPM_SECURITY_INFO_MODE_IMMEDIATE - Force new keys to take effect immediately
- IPM_SECURITY_INFO_MODE_WAITFOREXPIRATION - Wait until all the specified key's lifetime has expired before applying new keys

21.3.4 Switching from RTP to SRTP in Mid-session

To switch from RTP to SRTP during a session, use the **ipm_ModifyMedia( )** function and set SecurityInfo (of type IPM_SECURITY_INFO) fields as follows:

- pParms = a non-NULL value
- eInfoMode = IPM_SECURITY_INFO_MODE_IMMEDIATE
- unNumKeys = a non-zero value

21.3.5 Switching from SRTP to RTP in Mid-session

To switch from SRTP to RTP during a session, use the **ipm_ModifyMedia( )** function and set SecurityInfo (of type IPM_SECURITY_INFO) fields as follows:

- pParms = NULL
- eInfoMode = IPM_SECURITY_INFO_MODE_IMMEDIATE
- unNumKeys = 0 (zero)

Note: An alternative way to switch from SRTP to RTP during a session is to use the **ipm_ModifyMedia( )** function and set the usSrtpUnEncryptedFlag, usSrtcpUnEncryptedFlag and usSrtpUnAuthenticatefFlag fields in the associated IPM_SRTP_PARMS structure to 1.

21.3.6 Automatic Validation of Keys

When using the **ipm_StartMedia( )** and **ipm_ModifyMedia( )** functions for SRTP, the validity of the keys is automatically checked against the selected crypto suite (AES_CM_128_HMAC_SHA1_80 or AES_CM_128_HMAC_SHA1_32). If the key lengths do not conform to the crypto suite, an EIPM_BADPARM error is generated.

21.4 Retrieving and Modifying Encryption Key Expiry Notification Interval

The following SRTP parameter values can be retrieved or modified using the **ipm_GetParm( )** and **ipm_SetParm( )** functions:

PARMCH_NOTIFY_EXPIRE_KEY_AUDIO
: Advanced notification time (in multiples of 100 ms units) that the current encryption key for the audio media type is about to expire

PARMCH_NOTIFY_EXPIRE_KEY_VIDEO
: Advanced notification time (in multiples of 100 ms units) that the current encryption key for the video media type is about to expire

These parameters represent the pre-notification duration (in 100 ms increments) for each media type. An asynchronous event is generated that indicates that encryption key will expire in the preset time interval. See Section 21.7, "Events Generated by Secure RTP", on page 113 for more information.

21.5 Retrieving and Resetting Secure RTP Alarms

The SRTP feature uses the QoS alarm mechanism to detect conditions such as authentication failures, packet replay detection and Master Key Identifier (MKI) mismatches. Parameters characterizing these conditions (such as threshold values) can be defined. When one of the alarm conditions is detected, an IPM_QOS_ALARM event is generated to the application.

The **ipm_GetQoSThreshold( )** and **ipm_ResetQoSAlarmStatus( )** functions can be used to retrieve and reset the following alarms related to SRTP:

QOSTYPE_SEC_AUTH_FAIL_AUDIO
: Audio packets detection authentication failure; a measure of the number of audio packets that fail authentication in a given time interval

QOSTYPE_SEC_AUTH_FAIL_VIDEO
: Video packets detection authentication failure; a measure of the number of video packets that fail authentication in a given time interval

QOSTYPE_SEC_PKT_REPLAY_AUDIO
: Audio packets replay detection; a measure of the number of audio replay packets that are detected in a given time interval

QOSTYPE_SEC_PKT_REPLAY_VIDEO
: Video packets replay detection; a measure of the number of video replay packets that are detected in a given time interval

QOSTYPE_SEC_MKI_NOMATCH_AUDIO
: No Master Key Identifier (MKI) match in audio stream detection; a measure of the number of audio packets that have an MKI which does not match the master key in a given time interval

QOSTYPE_SEC_MKI_NOMATCH_VIDEO
No MKI match in video stream detection; a measure of the number of video packets that have an MKI which does not match the master key in a given time interval

Note: The default values that trigger these alarms are given in Table 2, "Secure RTP QoS Alarm Threshold Values", on page 113.

The **ipm_GetSessionInfo( )** function can also be used to retrieve information about these SRTP alarms. The information is retrieve in the same way as other QoS alarm information.

21.6 Retrieving and Setting Threshold Values for Secure RTP Alarms

The **ipm_SetQoSThreshold( )** function can be used to change the threshold values that trigger the Secure RTP alarms (described in Section 21.5, "Retrieving and Resetting Secure RTP Alarms", on page 112). Similarly, the **ipm_GetQoSThreshold( )** function can be used to retrieve the current threshold values that trigger Secure RTP alarms.

Table 2 shows the SRTP QoS alarm threshold values. See the IPM_QOS_THRESHOLD_DATA structure description for more information on the threshold values.

Table 2. Secure RTP QoS Alarm Threshold Values

Alarm Type	Time Interval (ms)	Debounce On (ms)	Debounce Off (ms)	Fault Threshold	% Success Threshold	% Failure Threshold
Audio Authentication	1000	10000	10000	10	20	20
Video Authentication	1000	10000	10000	10	20	20
Audio Packet Replay	1000	10000	10000	10	20	20
Video Packet Replay	1000	10000	10000	10	20	20
Audio MKI match	1000	10000	10000	10	20	20
Video MKI match	1000	10000	10000	10	20	20

21.7 Events Generated by Secure RTP

An application may receive the following notification events during SRTP operation:

IPMEV_SEC_NOTIFY_EXPIRE_KEY_AUDIO
The encryption key for the audio media type is about to expire in the predefined time interval

IPMEV_SEC_NOTIFY_EXPIRE_KEY_VIDEO
The encryption key for the video media type is about to expire in the predefined time interval

See Section 21.4, “Retrieving and Modifying Encryption Key Expiry Notification Interval”, on page 112 for information on setting the parameter values that trigger the alarms and generate the above events.

In addition, an application may receive IPMEV_QOS_ALARM events associated with SRTP QoS alarms. See Section 17.2, “QoS Alarm Types”, on page 85 for more information. The method used to identify the actual event type is similar to that shown for other QoS events in Section 17.7, “Example Code for QoS Alarm Handling”, on page 92 (specifically the **checkEvent( )** function).

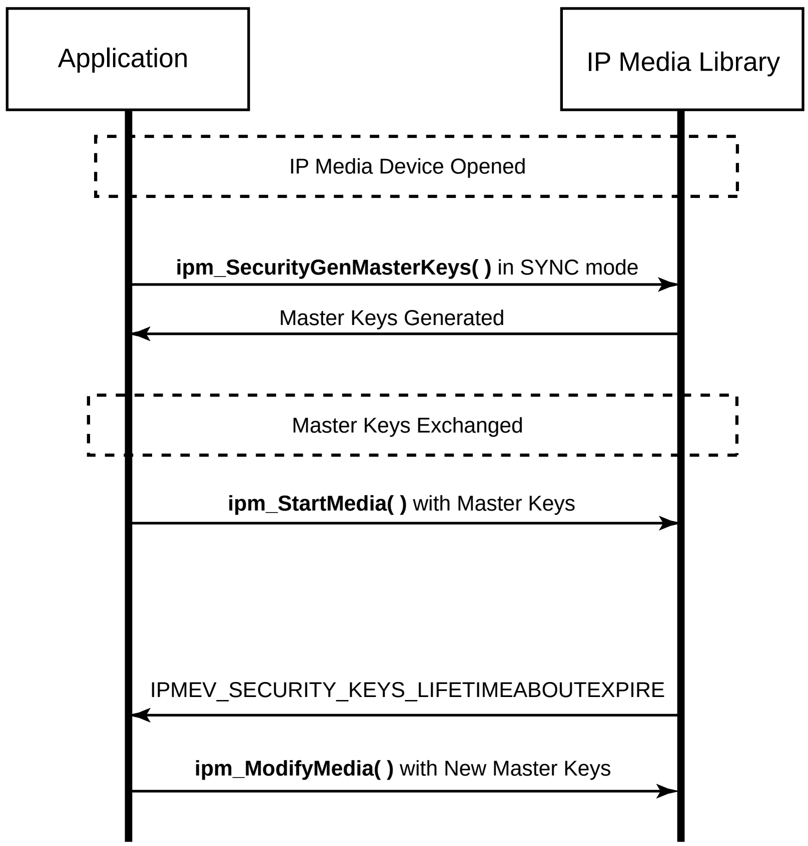
See Section 21.6, “Retrieving and Setting Threshold Values for Secure RTP Alarms”, on page 113 for information on setting parameter values that define when these SRTP QoS alarm events are generated.

You can also use the **ipm_EnableEvents( )** and **ipm_DisableEvents( )** functions to enable or disable the generation of both the notification and QoS alarm events to the application. See Section 21.3.1, “Key About to Expire Indication”, on page 110.

21.8 Use Case for Secure RTP

Figure 18 shows an example SRTP use case scenario.

Figure 18. SRTP Use Case



21.9 Example Code for Secure RTP

The following sample application code demonstrates how to generate master and master salt keys and use the keys in an SRTP media session.

```
/*the following sample code uses SRTP */

#include <stdio.h>
#include <string>
#include <srllib.h>
#include <ipmlib.h>
typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    /*
    .
```

```
.
Main Processing
.
*/

/*
Set the keys for the IP device handle, nDeviceHandle.
ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
*/

char Mkey1[30], bs64Mkey1[35];
char Msalt1[30];

IPM_SRTP_PARMS srtp_parms;
IPM_SECURITY_KEY MasterKeys[2];
IPM_SECURITY_BASE64_KEY Masterbs64Key;
IPM_SECURITY_BINARY_KEY MasterbinKey;

INIT_IPM_SRTP_PARMS(&srtp_parms);
INIT_IPM_SECURITY_BINARY_KEY(&MasterbinKey);

MasterbinKey.pcMasterKey = Mkey1;
MasterbinKey.pcMasterSaltKey = Msalt1;

INIT_IPM_SECURITY_BASE64_KEY(&Masterbs64Key);
Masterbs64Key.pcMasterBase64Key = bs64Mkey1;

INIT_IPM_SECURITY_KEY(&MasterKeys[0]);
MasterKeys[0].eKeyType = IPM_SECURITY_KEYTYPE_BINARY;
MasterKeys[0].pvMasterKey = &MasterbinKey;

INIT_IPM_SECURITY_KEY(&MasterKeys[1]);
MasterKeys[1].eKeyType = IPM_SECURITY_KEYTYPE_BASE64;
MasterKeys[1].pvMasterKey = &Masterbs64Key;


/* Generate the master Key and Master Salt Key for the device */

if ((ipm_SecurityGenMasterKeys(nDeviceHandle, MasterKeys, 1, EV_SYNC) == -1))
{
    printf("ipm_SecurityGenMasterKeys() failed for device name = %s with error = %d\n",
    ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
    /*
    .
    .
    Perform Error Processing
    */
}


/* Masterbs64Key can be filled from SDP */



/*
Set the media properties for a remote party using IP device handle, nDeviceHandle.
ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
*/

IPM_MEDIA_INFO MediaInfo;
MediaInfo.unCount = 6;


MediaInfo.MediaData[0].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTP_INFO;
MediaInfo.MediaData[0].mediaInfo.PortInfo.unPortId = 2328;
strcpy(MediaInfo.MediaData[0].mediaInfo.PortInfo.cIPAddress, "111.21.0.9\n");
```

```
    MediaInfo.MediaData[1].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTCP_INFO;
    MediaInfo.MediaData[1].mediaInfo.PortInfo.unPortId = 2329;
    strcpy(MediaInfo.MediaData[1].mediaInfo.PortInfo.cIPAddress, "111.41.0.9\n");

    MediaInfo.MediaData[2].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
    MediaInfo.MediaData[2].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
    MediaInfo.MediaData[2].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
    MediaInfo.MediaData[2].mediaInfo.CoderInfo.unFramesPerPkt = 1;
    MediaInfo.MediaData[2].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
    MediaInfo.MediaData[2].mediaInfo.CoderInfo.unCoderPayloadType = 0;
    MediaInfo.MediaData[2].mediaInfo.CoderInfo.unRedPayloadType = 0;

    MediaInfo.MediaData[3].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
    MediaInfo.MediaData[3].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
    MediaInfo.MediaData[3].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
    MediaInfo.MediaData[3].mediaInfo.CoderInfo.unFramesPerPkt = 1;
    MediaInfo.MediaData[3].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
    MediaInfo.MediaData[3].mediaInfo.CoderInfo.unCoderPayloadType = 0;
    MediaInfo.MediaData[3].mediaInfo.CoderInfo.unRedPayloadType = 0;


    MediaInfo.MediaData[4].eMediaType = MEDIATYPE_AUDIO_REMOTE_SECURITY_INFO;
    MediaInfo.MediaData[4].mediaInfo.SecurityInfo.unVersion = IPM_SECURITY_INFO_VERSION;
    MediaInfo.MediaData[4].mediaInfo. SecurityInfo.unNumKeys = 1;
    MediaInfo.MediaData[4].mediaInfo. SecurityInfo.pParms = &srtp_parms;
    MediaInfo.MediaData[4].mediaInfo. SecurityInfo.pKeys = &MasterKeys[0];
    MediaInfo.MediaData[4].mediaInfo. SecurityInfo.eInfoMode = IPM_SECURITY_INFO_MODE_IMMEDIATE;

    MediaInfo.MediaData[5].eMediaType = MEDIATYPE_AUDIO_LOCAL_SECURITY_INFO;
    MediaInfo.MediaData[5].mediaInfo. SecurityInfo.unVersion = IPM_SECURITY_INFO_VERSION;
    MediaInfo.MediaData[5].mediaInfo. SecurityInfo.unNumKeys = 1;
    MediaInfo.MediaData[5].mediaInfo. SecurityInfo.pParms = &srtp_parms;
    MediaInfo.MediaData[5].mediaInfo. SecurityInfo.pKeys = &MasterKeys[1];
    MediaInfo.MediaData[5].mediaInfo. SecurityInfo.eInfoMode = IPM_SECURITY_INFO_MODE_IMMEDIATE;


    if (ipm_StartMedia(nDeviceHandle, &MediaInfo, DATA_IP_TDM_BIDIRECTIONAL,EV_SYNC) == -1)
    {
        printf("ipm_StartMediaInfo failed for device name = %s with error = %d\n",
        ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        */
    }
    /*
    .
    . Continue processing
    .
    */

}
```

Building Applications 22

This chapter contains information on how to compile and link your Dialogic® IP Media Library API applications under the Linux and Windows® operating systems. The information is presented in the following topics:

22.1 Compiling and Linking under Linux

The following topics discuss compiling and linking requirements:

- Include Files
- Required Libraries

22.1.1 Include Files

To use Dialogic® IP Media Library API functions in your Linux application, certain include files (also known as header files) and library files are required. You must add statements for these include files in your application. The following header files contain equates that are required for each Linux application that uses the Dialogic® IP Media Library API:

ipmerror.h
: Dialogic® IP Media Library API error header file

ipmlib.h
: Dialogic® IP Media Library API header file

22.1.2 Required Libraries

The following library files must be linked to the application **in the following order**:

libipm.so
: Linking this file is mandatory. Specify `-lipm` in makefile.

libgc.so
: Required only if the application uses R4 Global Call library functions directly, for example, **gc_OpenEx( )**. Specify `-lgc` in makefile.

libdxxx.so
: Required only if the application uses R4 voice library functions directly, for example, **dx_play( )**. Specify `-ldxxx` in makefile.

libsrl.so
: The Dialogic® Standard Runtime Library (SRL) is mandatory. Specify `-lsrl` in makefile.

libpthread.so
POSIX threads system library. Specify `-lpthread` in makefile.

libdl.so
Dynamic Loader system library. Specify `-ldl` in makefile.

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

22.2 Compiling and Linking under Windows®

The following topics discuss compiling and linking requirements:

- Include Files
- Required Libraries

22.2.1 Include Files

To use Dialogic® IP Media Library API functions in your Windows®-based application, certain include files (also known as header files) and library files are required. You must add statements for these include files in your application. The following header files contain equates that are required for each Windows® application that uses the Dialogic® IP Media Library API:

ipmerror.h
Dialogic® IP Media Library API error header file

ipmlib.h
Dialogic® IP Media Library API header file

22.2.2 Required Libraries

The following library files must be linked to the application:

libipm.lib
Linking this file is mandatory.

libgc.lib
Required only if the application uses Dialogic® Global Call API library functions directly, for example, **gc_OpenEx( )**. Use the `-lgc` argument to the system linker.

libdxxxmt.lib
Required only if the application uses R4 voice library functions directly, for example, **dx_play( )**.

libsrlmt.lib
The Dialogic® Standard Runtime Library (SRL) is mandatory.

Function Summary by Category 23

The Dialogic® IP Media Library API contains functions which control and monitor media resources in an IP environment. This chapter contains an overview of the Dialogic® IP Media Library API functions, which are grouped into the categories listed below. This chapter also includes a table listing function support on various platforms.

23.1 System Control Functions

The following functions are used to manage channel, parameter, and event operations:

ipm_Close( )
closes an IP channel

ipm_DisableEvents( )
disables IP notification events

ipm_EnableEvents( )
enables IP notification events

ipm_GetParm( )
returns IP channel parameters

ipm_GetXmitSlot( )
returns TDM time slot information for an IP channel

ipm_Listen( )
connects an IP channel to a TDM time slot

ipm_Open( )
opens an IP channel and returns a handle

ipm_SetParm( )
sets IP channel parameters

ipm_UnListen( )
disconnects an IP channel from a TDM time slot

23.2 I/O (Input/Output) Functions

The following functions are used to transfer digits and data:

ipm_ReceiveDigits( )
: enables the IP channel to receive digits from the TDM bus

ipm_SendDigits( )
: generates supplied digits to the TDM bus

23.3 Media Session Functions

The following functions are used to perform session management:

ipm_GenerateIFrame( )
: generates an I-frame

ipm_GetCTInfo( )
: retrieves information about an IPM device voice channel

ipm_GetLocalMediaInfo( )
: retrieves properties for the local media channel

ipm_GetSessionInfo( )
: retrieves statistics for a session

ipm_GetSessionInfoEx( )
: retrieves RTCP data for a session

ipm_InitResponseSend( )
: sends a response to an IP session

ipm_InitSend( )
: sends an initialization message to a remote party

ipm_ModifyMedia( )
: modifies the properties of an active media session

ipm_SecurityGenMasterKeys( )
: generate master and salt keys

ipm_StartMedia( )
: sets properties for the local and remote media channels and starts the session

ipm_Stop( )
: stops operations on an IP channel

23.4 Quality of Service (QoS) Functions

The following functions are used to control QoS alarms and alarm thresholds:

ipm_GetQoSAlarmStatus( )
: retrieves the ON/OFF state of QoS alarms

ipm_GetQoSThreshold( )
retrieves QoS alarm threshold settings

ipm_ResetQoSAlarmStatus( )
resets QoS alarm to OFF state once it has been triggered

ipm_SetQoSThreshold( )
changes QoS alarm threshold settings

23.5 Dialogic® IP Media Library API Function Support by Platform

The following table provides an alphabetical listing of Dialogic® IP Media Library API functions. The table indicates platform support for each function: Dialogic® Host Media Processing (HMP) Software Release 4.1LIN (HMP 4.1LIN), Dialogic® Host Media Processing (HMP) Software 3.1LIN (HMP 3.1LIN), Dialogic® Host Media Processing (HMP) Software 3.0WIN (HMP 3.0WIN), Dialogic® Multimedia Platform for AdvancedTCA (MMP for ATCA), and Dialogic® Multimedia Kit for PCIe (MMK for PCIe).

Although a function may be supported on all platforms, there may be some differences in usage. For details, see the function reference descriptions in Chapter 24, "Function Information".

Table 3. Dialogic® IP Media Library API Function Support by Platform

Function	HMP 3.0WIN	HMP 3.1LIN	HMP 4.1LIN	MMP for ATCA	MMK for PCIe
ipm_Close( )	S	S	S	S	S
ipm_DisableEvents( )	S	S	S	S	S
ipm_EnableEvents( )	S	S	S	S	S
ipm_GenerateIFrame( )	NS	NS	S	S	S
ipm_GetCapabilities( )	S	S	S	S	S
ipm_GetCTInfo( )	S	S	S	S	S
ipm_GetLocalMediaInfo( ) †	S	S	S	S	S
ipm_GetParm( ) †	S	S	S	S	S
ipm_GetQoSAlarmStatus( )	S	S	S	S	S
ipm_GetQoSThreshold( )	S	S	S	S	S
ipm_GetSessionInfo( )	S	S	S	S	S
ipm_GetSessionInfoEx( )	NS	S	S	NS	NS
ipm_GetXmitSlot( )	S	S	S	S	S
ipm_InitResponseSend( )	NS	S	S	S	S
ipm_InitSend( )	NS	S	S	S	S
Legend: NS = Not Supported, S = Supported, † = Variance between platforms, refer to Function Description for more information.					

Table 3. Dialogic® IP Media Library API Function Support by Platform (Continued)

Function	HMP 3.0WIN	HMP 3.1LIN	HMP 4.1LIN	MMP for ATCA	MMK for PCIe
ipm_Listen( )	S	S	S	S	S
ipm_ModifyMedia( )	S	S	S	NS	NS
ipm_Open( )	S	S	S	S	S
ipm_ReceiveDigits( )	S	S	S	NS	NS
ipm_ResetQoSAlarmStatus( )	S	S	S	NS	NS
ipm_SecurityGenMasterKeys( )	S	S	NS	NS	NS
ipm_SendDigits( )	S	S	S	NS	NS
ipm_SetParm( ) †	S	S	S	S	S
ipm_SetQoSThreshold( )	S	S	S	S	S
ipm_SetRemoteMediaInfo( ) (use **ipm_StartMedia( )**)	deprecated	deprecated	deprecated	deprecated	deprecated
ipm_StartMedia( ) †	S	S	S	S	S
ipm_Stop( )	S	S	S	S	S
ipm_UnListen( )	S	S	S	S	S
Legend: NS = Not Supported, S = Supported, † = Variance between platforms, refer to Function Description for more information.					

Function Information 24

This chapter contains a detailed description of each Dialogic® IP Media Library API function, presented in alphabetical order.

24.1 Function Syntax Conventions

Note: The "Platform" line in the function header table of each function indicates the general platforms supported. For a list of software releases supported, see Table 3, "Dialogic® IP Media Library API Function Support by Platform", on page 122.

The Dialogic® IP Media Library API functions use the following format:

```
ipm_Function (DeviceHandle, Parameter1, Parameter2, …, ParameterN, Mode)
```

where:

ipm_Function
: is the name of the function

DeviceHandle
: is an input field that directs the function to a specific line device

Parameter1, Parameter2, …, ParameterN
: are input or output fields

Mode
: is an input field indicating how the function is executed. This field is applicable to certain functions only. For example, **ipm_Close( )** can only be called synchronously, so Mode is not used. Possible Mode values are:

- EV_ASYNC for asynchronous mode execution. When running asynchronously, the function will return 0 to indicate it has initiated successfully, and will generate a termination event to indicate completion.
- EV_SYNC for synchronous mode execution. When running synchronously, the function will return a 0 to indicate that it has completed successfully.

ipm_Close( )

Name: int ipm_Close(nDeviceHandle, *pCloseInfo)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_CLOSE_INFO *pCloseInfo	• set to NULL

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: System Control

Mode: synchronous only

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

Description

The **ipm_Close( )** function closes an IP channel device and disables the generation of all events.

Parameter	Description
nDeviceHandle	IP Media device handle returned by **ipm_Open( )**
pCloseInfo	set to NULL; reserved for future use

Termination Events

None - this function operates in synchronous mode only.

Cautions

- The **pCloseInfo** pointer is reserved for future use and must be set to NULL.
- Issuing a call to **ipm_Open( )** or **ipm_Close( )** while the device is being used by another process will not affect the current operation of the device. Other handles for that device that exist in the same process or other processes will still be valid. The only process affected by **ipm_Close( )** is the process that called the function.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
    Invalid parameter

EIPM_CONFIG
    Configuration error

EIPM_FWERROR
Firmware error

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

void main()
{
    int nDeviceHandle;

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    Application is shutting down.
    Need to close IP device handle.
    ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
    */
    if(ipm_Close(nDeviceHandle, NULL) == -1)
    {
        printf("----------->ipm_Close() failed for handle = %d\n", nDeviceHandle);
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }

    /*
    .
    .
    Continue cleanup
    .
    .
    */
}
```

See Also

- **ipm_Open( )**

ipm_DisableEvents( )

Name: int ipm_DisableEvents(nDeviceHandle, *pEvents, unNumOfEvents, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
eIPM_EVENT *pEvents	• array of events to disable
unsigned int unNumOfEvents	• number of events to disable
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: System Control

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_DisableEvents( )** function disables IP notification events. Some events are used for Quality of Service (QoS) alarm notifications. Other events are used to indicate status, for example, if an RFC 2833 event has been detected.

Notification events are different from asynchronous function termination events, such as IPMEV_OPEN, which cannot be disabled. Once a particular notification event is successfully disabled, the application is not notified if an event of that type occurs.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pEvents	array of enumerations that specifies the events to disable The eIPM_EVENT data type is an enumeration that defines the following values: • EVT_ENDPOINTID_NOTIFY – Notification containing the identification of the RTP/RTCP endpoint • EVT_JITTER – QoS alarm for excessive average jitter • EVT_LOSTPACKETS – QoS alarm for excessive percentage of lost packets • EVT_RTCP_JB_HIGH – QoS alarm used to track RTCP excessive jitter buffer • EVT_RTCP_JB_LOW – QoS alarm used to track RTCP jitter buffer below the threshold • EVT_RTCP_SCS – QoS alarm used to track RTCP severely concealed second condition (SCS) • EVT_RTCPTIMEOUT – QoS alarm for RTCP inactivity • EVT_RTPTIMEOUT – QoS alarm for RTP inactivity • EVT_SEC_AUTH_FAIL_AUDIO – Secure RTP QoS alarm for authentication failure on audio packets • EVT_SEC_AUTH_FAIL_VIDEO – Secure RTP QoS alarm for authentication failure on video packets • EVT_SEC_MKI_NOMATCH_AUDIO – Secure RTP QoS alarm for MKI mis-match on audio packets • EVT_SEC_MKI_NOMATCH_VIDEO – Secure RTP QoS alarm for MKI mis-match on video packets • EVT_SEC_NOTIFY_EXPIRE_KEY_AUDIO – Secure RTP notification of encryption key for audio about to expire • EVT_SEC_NOTIFY_EXPIRE_KEY_VIDEO – Secure RTP notification of encryption key for video about to expire • EVT_SEC_PKT_REPLAY_AUDIO – Secure RTP QoS alarm for replay detection on audio packets • EVT_SEC_PKT_REPLAY_VIDEO – Secure RTP QoS alarm for replay detection on video packets • EVT_TELEPHONY – RFC 2833 event
unNumOfEvents	number of events to disable (number of enumerations in **pEvents** array)
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_EVENT_DISABLED
: Indicates successful completion; that is, specified events were disabled. This event does not return any data.

IPMEV_ERROR
: Indicates that the function failed.

Cautions

The function fails if **nDeviceHandle** specifies a board device; notification events are only supported for channel devices.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_INTERNAL
: Internal error

EIPM_INV_EVT
: Invalid event

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

EIPM_UNSUPPORTED
: Function unsupported

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    int nDeviceHandle;
    eIPM_EVENT myEvents[2] ={EVT_LOSTPACKETS, EVT_JITTER};
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV, EV_ANYEVT, (HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    Application is shutting down
    Need to disable all enabled events for IP device handle, nDeviceHandle.
    ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open() and
    The events listed in myEvents were enabled sometime earlier.
```

```
    */
    if(ipm_DisableEvents(nDeviceHandle, myEvents, 2, EV_ASYNC) == -1)
    {
        printf("ipm_DisableEvents failed for device name = %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }

    /*
    .
    .
    Continue shut down
    .
    .
    */
}

void CheckEvent()
{
    int nEventType = sr_getevttype();
    int nDeviceID = sr_getevtdev();

    switch(nEventType)
    {
        /*
        .
        .
        . Other events
        .
        .
        */

        /* Expected reply to ipm_DisableEvents */
        case IPMEV_EVENT_DISABLED:
            printf("Received IPMEV_EVENT_DISABLED for device = %s\n",
                ATDV_NAMEP(nDeviceID));
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

■ See Also

- **ipm_EnableEvents( )**

ipm_EnableEvents( )

Name: int ipm_EnableEvents(nDeviceHandle, *pEvents, unNumOfEvents, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
eIPM_EVENT *pEvents	• array of events to enable
unsigned int unNumOfEvents	• number of events to enable
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: System Control

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_EnableEvents**( ) function enables IP notification events. Some events are used for Quality of Service (QoS) notifications on a particular media channel. Other events are used to indicate status, for example, if an RFC 2833 event has been detected.

Notification events (solicited events) are different from asynchronous function termination events, such as IPMEV_OPEN, which cannot be disabled. Once a particular notification event is successfully enabled, the application is notified via SRL event management functions whenever the specified event occurs.

Note: A SUCCESS message returned from a set event request in the IPML library is only a notification that the request was processed, not that all requested events are enabled.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pEvents	array of enumerations that specifies the events to enable The eIPM_EVENT data type is an enumeration that defines the following values: • EVT_ENDPOINTID_NOTIFY – Notification containing the identification of the RTP/RTCP endpoint • EVT_JITTER – QoS alarm for excessive average jitter • EVT_LOSTPACKETS – QoS alarm for excessive percentage of lost packets • EVT_RTCP_JB_HIGH – QoS alarm used to track RTCP excessive jitter buffer • EVT_RTCP_JB_LOW – QoS alarm used to track RTCP jitter buffer below the threshold • EVT_RTCP_SCS – QoS alarm used to track RTCP severely concealed second condition (SCS) • EVT_RTCPTIMEOUT – QoS alarm for RTCP inactivity • EVT_RTPTIMEOUT – QoS alarm for RTP inactivity • EVT_SEC_AUTH_FAIL_AUDIO – Secure RTP QoS alarm for authentication failure on audio packets • EVT_SEC_AUTH_FAIL_VIDEO – Secure RTP QoS alarm for authentication failure on video packets • EVT_SEC_MKI_NOMATCH_AUDIO – Secure RTP QoS alarm for MKI mis-match on audio packets • EVT_SEC_MKI_NOMATCH_VIDEO – Secure RTP QoS alarm for MKI mis-match on video packets • EVT_SEC_NOTIFY_EXPIRE_KEY_AUDIO – Secure RTP notification of encryption key for audio about to expire • EVT_SEC_NOTIFY_EXPIRE_KEY_VIDEO – Secure RTP notification of encryption key for video about to expire • EVT_SEC_PKT_REPLAY_AUDIO – Secure RTP QoS alarm for replay detection on audio packets • EVT_SEC_PKT_REPLAY_VIDEO – Secure RTP QoS alarm for replay detection on video packets • EVT_TELEPHONY – RFC 2833 event
unNumOfEvents	number of events to enable (number of enumerations in **pEvents** array)
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_EVENT_ENABLED
Indicates successful completion; that is, specified events were enabled. This event does not return any data.

IPMEV_ERROR
: Indicates that the function failed.

Cautions

The function fails if **nDeviceHandle** specifies a board device; notification events are only supported for channel devices.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_EVT_EXIST
: Event already enabled

EIPM_EVT_LIST_FULL
: Too many events

EIPM_INTERNAL
: Internal error

EIPM_INV_EVT
: Invalid event

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

EIPM_UNSUPPORTED
: Function unsupported

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    int nDeviceHandle;
    eIPM_EVENT myEvents[2] ={EVT_LOSTPACKETS, EVT_JITTER};
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

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

```
    /*
    Need to enable three events for IP device handle, nDeviceHandle.
    ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
    */
    if(ipm_EnableEvents(nDeviceHandle, myEvents, 2, EV_ASYNC) == -1)
    {
        printf("ipm_EnableEvents failed for device name %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }

    /*
    .
    .
    . Continue Processing
    .
    .
    */
}

void CheckEvent()
{
    int nEventType = sr_getevttype();
    int nDeviceID = sr_getevtdev();

    switch(nEventType)
    {
        /*
        .
        .
        . List of expected events
        .
        .
        */

        /* Expected reply to ipm_EnableEvents() */
        case IPMEV_EVENT_ENABLED:
            printf("Received IPMEV_EVENT_ENABLED for device = %s\n",
                ATDV_NAMEP(nDeviceID));
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

■ See Also

- **ipm_DisableEvents( )**

ipm_GenerateIFrame( )

Name:	int ipm_GenerateIFrame (nDeviceHandle, pFrameInfo, a_usMode)	
Inputs:	int a_nDeviceHandle	• SRL handle of the IP media device
	IPM_IFRAME_INFO * pFrameInfo	• pointer to define structure
	unsigned short a_usMode	• async or sync mode setting
Returns:	0 on success -1 on error	
Includes:	srllib.h ipmlib.h	
Category:	Media Session	
Mode:	Asynchronous or synchronous	
Platform:	Dialogic® HMP Software (Linux), Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe	

■ Description

The **ipm_GenerateIFrame**( ) function requests to the IP media device to generate an I-frame for video fast update.

Note: In order to use this function, the transcoding flag must be ON when connecting an IP media device to any other device. Refer to 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.

The application would typically call the **ipm_GenerateIFrame**( ) function to generate an I-frame when it receives a SIP INFO request to do a video fast update.

Parameter	Description
a_nDeviceHandle	SRL handle of the IP media device
pFrameInfo	structure describing the I-frame information. See the IPM_IFRAME_INFO data structure page for details.
a_usMode	specifies operation mode. Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_GENERATEIFRAME
Indicates successful completion.

IPMEV_GENERATEIFRAME_FAIL
Indicates that the function failed.

Cautions

None.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_GENERATEIFRAME_INCAPABLE
: Incapable of generating an I-frame. Example: Transcoding is not turned ON for the connection between the IP media device and another device.

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
: Invalid state. Example: Streaming is not active on the channel when the function is called.

EIPM_SYSTEM
: System error

EIPM_ GENERATEIFRAME _INCAPABLE
: Incapable of generating an I-frame

Example

```
#include <srllib.h>
#include <ipmlib.h>
#include <ipmerror.h>
#include <errno.h>
void call_setup()
{
        IPM_IFRAME_INFO aIFrameInfo;
        INIT_IPM_IFRAME_INFO(&aIFrameInfo);

        IPM_MEDIA_INFO MediaInfo;
        MediaInfo.unCount = 4;


        /* setup coder info */
        MediaInfo.MediaData[0].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
        MediaInfo.MediaData[1].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
        MediaInfo.MediaData[2].eMediaType = MEDIATYPE_VIDEO_REMOTE_CODER_INFO;
        MediaInfo.MediaData[3].eMediaType = MEDIATYPE_VIDEO_LOCAL_CODER_INFO;
        if(ipm_StartMedia(DeviceHandle,&MediaInfo, DATA_IP_TDM_BIDIRECTIONAL, EV_ASYNC) < 0)
        {
               /*Process error */
        }
        /* Continue processing */
        /* recvd a SIP INFO requesting an I-frame */
        printf("SIP INFO request to generate I-frame");
        if ((ipm_GenerateIFrame(DeviceHandle, &aIFrameInfo ,EV_ASYNC) == -1)
        {
               /*Process error */
        }...
        /* Continue processing */
        }
```

```
        void CheckEvents()
        {
             int nEventType = sr_getevttype();
             int nDeviceID = sr_getevtdev();
             int nError;

           switch (nEventType)
           {
                  /* Expected reply to ipm_GenerateIFrame */
                  case IPMEV_GENERATEIFRAME:
                  printf("Received IPMEV_GENERATEIFRAME for device = %s\n",
                          ATDV_NAMEP(nDeviceID));
                  break;

                  case IPMEV_GENERATEIFRAME_FAIL:
                  nError = ATDV_LASTERR(nDeviceID);
                  printf("Received IPMEV_GENERATEIFRAME_FAIL for device = %s\n",
                          ATDV_NAMEP(nDeviceID));
                  switch (nError)
                  {
                              case EIPM_GENERATEIFRAME_INCAPABLE:
                              printf("IPM incapable of generating iframe\n");
                              break;
                              default:
                              printf("Error=%d\n",nError);
                              break;
                  }
                  break;

                  default:
                  printf("Received unknown event = %d for device = %s\n", nEventType,
                          ATDV_NAMEP(nDeviceID));
                  break;
            }
}
```

■ See Also

None.

ipm_GetCapabilities( )

Name: ipm_GetCapabilities(a_nDeviceHandle, a_CapType, a_num, a_CapabilitiesArray[], a_usMode);

Inputs:

int a_nDeviceHandle	• IP Media device handle
eCAPABILITY_TYPE a_CapType	• capability type to be retrieved
unsigned int a_num	• number of entries in the capability array
IPM_CAPABILITIES a_CapabilitiesArray[]	• capability array
unsigned short a_usMode	• mode of operation

Returns: number of capabilities available
-1 on failure

Includes: srllib.h
ipmlib.h

Category: Media Session

Mode: synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

---

■ Description

The **ipm_GetCapabilities( )** function returns the number of capabilities of the specified type (for example, coders) and details of each capability supported by an active Dialogic® HMP Software license. The number of capabilities available may be greater than the number specified by the **a_num** input parameter, therefore the following rules apply:

- If **a_num** is zero and/or **a_CapabilitiesArray[]** is NULL, this function returns only the number of capabilities available; no capability detail is retrieved.
- If **a_num** is larger than the number of capabilities available (the return value), **a_CapabilitiesArray[]** is filled with details of all capabilities and the remaining allocated memory is unused.
- If **a_num** is smaller than the number of capabilities available (the return value), **a_CapabilitiesArray[]** is filled with details of **a_num** capabilities (that is, as many as will fit); details of the remaining capabilities are not retrieved.

Parameter	Description
n_DeviceHandle	handle of the IP Media device
a_CapType	capability type, for example CAPABILITY_CODERLIST
a_num	the number of entries in the capability array
a_CapabilitiesArray[]	the capability array
a_usMode	operation mode Set to EV_SYNC for synchronous execution

The datatype for the **a_CapabilitiesArray[]** parameter is a union, IPM_CAPABILITIES, which is defined as follows:

```
typedef struct ipm_capabilities_tag
{
   unsigned int version;
   union
   {
      IPM_CODER_INFO Coder;
      // Future types here.
   };
}IPM_CAPABILITIES;
```

In this union, the IPM_AUDIO_CODER_INFO data structure provides coder details such as coder type, frame size, number of frames per packet, VAD enable/disable information and payload-related information.

The datatype for the **a_CapType** parameter is eCAPABILITY_TYPE, an enumeration that is defined as follows:

```
enum eCAPABILTIY_TYPE
{
   CAPABILITY_CODERLIST;
}
```

The **ipm_GetCapabilities( )** function is supported in synchronous mode only. If asynchronous mode (a_usMode = EV_ASYNC) is specified, an error is generated.

■ Cautions

None.

■ Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** or **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_ERROR
: System error

Example

In this example, the first **ipm_GetCapabilities**( ) call retrieves only number of capabilities available (count). That number is then used to allocate the right amount of memory and retrieve details of all the capabilities.

```
#include <ipmlib.h>

unsigned int count;
IPM_CAPABILITIES *caps;
int i;

count=ipm_GetCapabilities(dev,CAPABILITY_CODERLIST,0,NULL,EV_SYNC);
caps=(IPM_CAPABILITIES *)malloc(sizeof(IPM_CAPABILITIES)*count);

// check for memory error here

count=ipm_GetCapabilities(dev,CAPABILITY_CODERLIST,count,caps,EV_SYNC);

for (i=0;i<count;i++)
{
    printf("RFC 1890 Coder Type %ui supported\n",caps[i].Coder.unCoderPayloadType);
}

// Free coder list here

free(caps);
```

See Also

None.

ipm_GetCTInfo( )

Name: int ipm_GetCTInfo(nDeviceHandle, *pCTInfo, usMode)

Inputs:

int nDeviceHandle	• valid channel device handle
CT_DEVINFO *pCTInfo	• pointer to device information structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: ipmlib.h

Category: Media Session

Mode: Synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

Description

The **ipm_GetCTInfo( )** function returns information about a voice channel of an IPM device. This information is contained in a CT_DEVINFO data structure.

Parameter	Description
nDeviceHandle	specifies the valid IP channel handle obtained when the channel was opened using **ipm_Open( )**
pCTInfo	specifies a pointer to the CT_DEVINFO structure that contains the IP channel device information
usMode	operation mode Set to EV_SYNC for synchronous execution. Asynchronous mode is not supported.

Cautions

This function fails if an invalid IP channel handle is specified.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return the following error:

EIPM_BADPARM
Invalid parameter

Example

```
#include <srllib.h>
#include <ipmlib.h>
#include <errno.h>
```

```
main()
{
int chdev; /* Channel device handle */
CT_DEVINFO ct_devinfo; /* Device information structure */
/* Open board 1 channel 1 devices */
if ((chdev = ipm_Open("ipmB1C1", 0)) == -1) {
printf("Cannot open channel ipmB1C1. errno = %d", errno);
exit(1);
}

/* Get Device Information */
if (ipm_GetCTInfo(chdev, &ct_devinfo, EV_SYNC) == -1) {
printf("Error message = %s", ATDV_ERRMSGP(chdev));
exit(1);
}

printf("%s Product Id = 0x%x, Family = %d, Mode = %d, Network = %d, Bus
       mode = %d, Encoding = %d", ATDV_NAMEP(chdev), ct_devinfo.ct_prodid,
       ct_devinfo.ct_devfamily, ct_devinfo.ct_devmode, ct_devinfo.ct_nettype,
       ct_devinfo.ct_busmode, ct_devinfo.ct_busencoding);
}
```

See Also

- **ipm_Open( )**

ipm_GetLocalMediaInfo( )

Name: int ipm_GetLocalMediaInfo(nDeviceHandle, *pMediaInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_MEDIA_INFO *pMediaInfo	• pointer to media information structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: Media Session

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_GetLocalMediaInfo( )** function retrieves properties for the local media channel. This function retrieves the local IP address and port information for the audio and/or video RTP/RTCP ports or T.38 port associated with the specified IP channel. These properties are assigned during firmware download.

The combination of the eMediaType field (in the IPM_MEDIA data structure pointed to by MediaData[ ], which in turn is a field of the IPM_MEDIA_INFO data structure pointed to by **pMediaInfo**) and the unCount field (in the IPM_MEDIA_INFO data structure pointed to by **pMediaInfo**) specifies the media type(s) for which port information is to be retrieved. The allowed combinations are as follows:

- To retrieve RTP port information for both audio and video in a multimedia session, set the eMediaType fields to MEDIATYPE_AUDIO_LOCAL_RTP_INFO and MEDIATYPE_VIDEO_LOCAL_RTP_INFO respectively and unCount to 2. See the code example.
- To retrieve RTP port information for a video only session, set the eMediaType field to MEDIATYPE_VIDEO_LOCAL_RTP_INFO and unCount to 1.
- To retrieve RTP port information for an audio only session, set the eMediaType field to MEDIATYPE_AUDIO_LOCAL_RTP_INFO and unCount to 1.
- To retrieve T.38 fax port information, set the eMediaType field to MEDIATYPE_LOCAL_UDPTL_T38_INFO and unCount to 1.

For audio and/or video media types, only RTP information needs to be queried. RTCP information is automatically provided when querying for RTP.

Note: The RTCP port number is the RTP port number + 1.

Note: It is not possible to retrieve T.38 fax port information together with audio and/or video port information.

To run this function asynchronously, set **mode** to EV_ASYNC. The function returns 0 if successful and the application must wait for the IPMEV_GET_LOCAL_MEDIA_INFO event. Once the event has been returned, use Dialogic® Standard Runtime Library (SRL) functions to retrieve IPM_MEDIA_INFO structure fields.

To run this function synchronously, set **mode** to EV_SYNC. The function returns 0 if successful and the IPM_MEDIA_INFO structure fields will be filled in.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pMediaInfo	pointer to structure that contains local IP address and IP channel port information for audio and/or video RTP / RTCP ports or T.38 port See the IPM_MEDIA_INFO data structure page for details.
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_GET_LOCAL_MEDIA_INFO
: Indicates successful completion, that is, local media information was received. Once the event has been returned, use Dialogic® SRL functions to retrieve IPM_MEDIA_INFO structure fields.

IPMEV_ERROR
: Indicates that the function failed.

■ Cautions

- For the Dialogic® Multimedia Platform for AdvancedTCA, this function returns an invalid state error if a coder is not reserved prior to calling **ipm_GetLocalMediaInfo( )**. All audio and/or video coders must be reserved using the **dev_ReserveResourceEx( )** Dialogic® Device Management Library function. However, this restriction does not apply when using a fax (T.38) media type.

■ Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
System error

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    int nDeviceHandle;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    Get the local IP information for IP device handle, nDeviceHandle.
    ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
    */
    IPM_MEDIA_INFO MediaInfo;
    MediaInfo.unCount = 2;
    MediaInfo.MediaData[0].eMediaType = MEDIATYPE_VIDEO_LOCAL_RTP_INFO;
    // MediaInfo.MediaData[0].eMediaType = MEDIATYPE_LOCAL_UDPTL_T38_INFO;
    MediaInfo.MediaData[1].eMediaType = MEDIATYPE_AUDIO_LOCAL_RTP_INFO;


    if(ipm_GetLocalMediaInfo(nDeviceHandle, &MediaInfo, EV_ASYNC) == -1)
    {
        printf("ipm_GetLocalMediaInfo failed for device name %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }
    /*
    .
    .
    . Continue processing
    .
    .
    */
}

void CheckEvent()
{
    unsigned int i;
    int nDeviceID = sr_getevtdev();
    int nEventType = sr_getevttype();
    void* pVoid = sr_getevtdatap();
    IPM_MEDIA_INFO* pMediaInfo;
```

```
        switch(nEventType)
        {
            /*
            .
            .
            . Other events
            .
            .
            */

            /* Expected reply to ipm_GetLocalMediaInfo */
            case IPMEV_GET_LOCAL_MEDIA_INFO:
                printf("Received IPMEV_GET_LOCAL_MEDIA_INFO for device name = %s\n",
                    ATDV_NAMEP(nDeviceID));
                pMediaInfo = (IPM_MEDIA_INFO*)pVoid;
                for(i=0; i<pMediaInfo->unCount; i++)
                {
                   switch(pMediaInfo->MediaData[i].eMediaType)
                   {
                      case MEDIATYPE_VIDEO_LOCAL_RTP_INFO:
                         printf("MediaType=MEDIATYPE_VIDEO_LOCAL_RTP_INFO\n");
                         printf("PortId=%d\n",pMediaInfo->MediaData[i].mediaInfo.PortInfo.unPortId);
                         printf("IP=%s\n",pMediaInfo>MediaData[i].mediaInfo.PortInfo.cIPAddress);
                         break;
                      case MEDIATYPE_VIDEO_LOCAL_RTCP_INFO:
                         printf("MediaType=MEDIATYPE_VIDEO_LOCAL_RTCP_INFO\n");
                         printf("PortId=%d\n",pMediaInfo->MediaData[i].mediaInfo.PortInfo.unPortId);
                         printf("IP=%s\n",pMediaInfo>MediaData[i].mediaInfo.PortInfo.cIPAddress);
                         break;
                      case MEDIATYPE_AUDIO_LOCAL_RTP_INFO:
                         printf("MediaType=MEDIATYPE_AUDIO_LOCAL_RTP_INFO\n");
                         printf("PortId=%d\n",pMediaInfo->MediaData[i].mediaInfo.PortInfo.unPortId);
                         printf("IP=%s\n",pMediaInfo>MediaData[i].mediaInfo.PortInfo.cIPAddress);
                         break;
                      case MEDIATYPE_AUDIO_LOCAL_RTP_INFO:
                         printf("MediaType=MEDIATYPE_AUDIO_LOCAL_RTP_INFO\n");
                         printf("PortId=%d\n",pMediaInfo->MediaData[i].mediaInfo.PortInfo.unPortId);
                         printf("IP=%s\n",pMediaInfo>MediaData[i].mediaInfo.PortInfo.cIPAddress);
                         break;
                    }
                break;

            default:
                printf("Received unknown event = %d for device name = %s\n",
                    nEventType, ATDV_NAMEP(nDeviceID));
                break;
        }
    }
```

■ See Also

None

ipm_GetParm( )

Name: int ipm_GetParm(nDeviceHandle, *pParmInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_PARM_INFO *pParmInfo	• pointer to parameter info structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: System Control

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_GetParm( )** function retrieves the current value of a parameter.

To run this function asynchronously, set mode to EV_ASYNC. The function returns 0 if successful and the application must wait for the IPMEV_GETPARM event. Once the event has been returned, use Dialogic® Standard Runtime Library (SRL) functions to retrieve parameter values.

To run this function synchronously, set mode to EV_SYNC. The function returns 0 if successful and the IPM_PARM_INFO structure fields will be filled in with the retrieved parameter information.

Parameter	Description
nDeviceHandle	handle of the IP media device
***pParmInfo**	pointer to structure that contains IP channel parameter values See the IPM_PARM_INFO data structure page for details.
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_GET_PARM
Indicates successful completion. Use Dialogic® SRL functions to retrieve IPM_PARM_INFO structure fields.

IPMEV_ERROR
Indicates that the function failed.

Cautions

None

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_FWERROR
: Firmware error

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

void CheckEvent();
typedef long int(*HDLR)(unsigned long);

void main()
{
    int nDeviceHandle;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    ASSUMPTION: A valid nDeviceHandle was obtained from prior
    call to ipm_Open().
    */
    IPM_PARM_INFO ParmInfo;
    unsigned long ulParmValue = 0;
    ParmInfo.eParm = PARMCH_ECHOTAIL;
    ParmInfo.pvParmValue = &ulParmValue;
    if (ipm_GetParm(nDeviceHandle, &ParmInfo, EV_ASYNC)==-1)
    {
        printf("ipm_GetParm failed for device name %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }

    ulParmValue = 0;
    ParmInfo.eParm = PARMCH_ECHOTAIL;
```

```
    if (ipm_GetParm(nDeviceHandle, &ParmInfo, EV_SYNC)==-1)
    {
        printf("%s: ipm_GetParm failed..exiting..!!!\n", ATDV_NAMEP(nDeviceHandle));
    }
    else
    {
        printf("%s: ipm_GetParm(parm=0x%x,value=0x%x) ok %\n", ATDV_NAMEP(nDeviceHandle),
            ParmInfo.eParm, ulParmValue );
    }

    /*
    .
    .
    . continue
    .
    .
    */
}

void CheckEvent()
{
    int nEventType = sr_getevttype();
    int nDeviceID = sr_getevtdev();
    void* pVoid = sr_getevtdatap();
    IPM_PARM_INFO* pParmInfo;

    switch(nEventType)
    {
        /*
        .
        .
        . Other events
        .
        .
        */

        /* Expected reply to ipm_GetQoSAlarmStatus */
        case IPMEV_GET_PARM:
            pParmInfo = (IPM_PARM_INFO*) pVoid;
            printf("Received IPMEV_GETPARM for device = %s\n",
                ATDV_NAMEP(nDeviceID));
            printf("%s: parm=0x%x, ok %\n", ATDV_NAMEP(nDeviceID),
                pParmInfo->eParm);
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

■ See Also

- **ipm_SetParm( )**

ipm_GetQoSAlarmStatus( )

Name: int ipm_GetQoSAlarmStatus(nDeviceHandle, *pQoSAlarmInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_QOS_ALARM_STATUS *pQoSAlarmInfo	• pointer to QoS alarm status structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: QoS

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_GetQoSAlarmStatus( )** function retrieves the ON/OFF state of Quality of Service (QoS) alarms that report the status of a media channel.

This function returns the status of media channel QoS alarms that are enumerated in eIPM_QOS_TYPE. This function does not return the status of board-level alarms.

Use **ipm_ResetQoSAlarmStatus( )** to reset the QoS alarm state.

Parameter	Description
nDeviceHandle	handle of the IP Media channel device
pQoSAlarmInfo	pointer to structure that contains alarm identifier and alarm status values See IPM_QOS_ALARM_STATUS for details.
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_GET_QOS_ALARM_STATUS
Indicates successful completion. Use SRL functions to retrieve IPM_QOS_ALARM_STATUS structure fields.

IPMEV_ERROR
Indicates that the function failed.

Cautions

None.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>
void CheckEvent();
typedef long int(*HDLR)(unsigned long);

void main()
{
   int nDeviceHandle;
   // Register event handler function with srl
   sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

   /*
   .
   .
    Main Processing
   .
   .
   */
   /*
    Query the alarm status for IP device handle, nDeviceHandle.
    ASSUMPTION:  A valid nDeviceHandle was obtained from prior call to ipm_Open().
   */
   IPM_QOS_ALARM_STATUS AlarmStatus;
   AlarmStatus.unAlarmCount = 1;
   AlarmStatus.QoSData[0].eQoSType = QOSTYPE_LOSTPACKETS;

   if(ipm_GetQoSAlarmStatus(nDeviceHandle, &AlarmStatus, EV_ASYNC) == -1)
   {
      printf("ipm_GetQoSAlarmStatus failed for device name %s with error = %d\n",
             ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
      /*
      .
      .
```

```
        Perform Error Processing
        .
        .
        */
    }

    /*
    .
    .
    continue
    .
    .
    */
}

void CheckEvent()
{
    int i;
    int nEventType = sr_getevttype();
    int nDeviceID  = sr_getevtdev();
    void* pVoid    = sr_getevtdatap();
    IPM_QOS_ALARM_STATUS* pmyAlarmStatus;
    switch(nEventType)
    {
        /*
        .
        .
        Other events
        .
        .
        */
        /* Expected reply to ipm_GetQoSAlarmStatus */
        case IPMEV_GET_QOS_ALARM_STATUS:
            pmyAlarmStatus = (IPM_QOS_ALARM_STATUS*)pVoid;
            printf("Received IPMEV_GET_QOS_ALARM_STATUS for device = %s\n",
                   ATDV_NAMEP(nDeviceID));
            for(i=0; i<pmyAlarmStatus->unAlarmCount; ++i)
            {
                switch(pmyAlarmStatus->QoSData[i].eQoSType)
                {
                    case QOSTYPE_LOSTPACKETS:
                        printf("  LOSTPACKETS = %d\n",l_myAlarmStatus.QoSData[i].eAlarmState);
                        break;
                    case QOSTYPE_JITTER:
                        printf("  JITTER = %d\n",l_myAlarmStatus.QoSData[i].eAlarmState);
                        break;
                }
            }
            break;
        default:
            printf("Received unknown event = %d  for device = %s\n",
                   nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

■ See Also

- **ipm_ResetQoSAlarmStatus( )**

ipm_GetQoSThreshold( )

Name: int ipm_GetQoSThreshold(nDeviceHandle, *pQoSThresholdInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_QOS_THRESHOLD_INFO *pQoSThresholdInfo	• pointer to QoS alarm threshold structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: QoS

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_GetQoSThreshold**( ) function retrieves alarm threshold settings for Quality of Service (QoS) alarms that report the status of media channels.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pQoSThresholdInfo	pointer to IPM_QOS_THRESHOLD_INFO structure which contains one or more IPM_QOS_THRESHOLD_DATA structures
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_GET_QOS_THRESHOLD_INFO
: Indicates successful completion. Use SRL functions to retrieve IPM_QOS_THRESHOLD_INFO structure fields.

IPMEV_ERROR
: Indicates that the function failed.

■ Cautions

- The IPM_QOS_THRESHOLD_INFO structure specifies the QoS Alarm Identifier thresholds. The application may use this structure to get statistics for only specified QoS types. Use SRL functions to retrieve IPM_QOS_THRESHOLD_INFO structure fields.

- If **ipm_GetQoSThreshold**( ) is called synchronously, the IPM_QOS_THRESHOLD_INFO structure is both an input and output parameter. If **ipm_GetQoSThreshold**( ) is called asynchronously, the structure is used only as an input parameter. To retrieve all the QoS threshold settings, in both synchronous and asynchronous modes, set the **unCount** field in IPM_QOS_THRESHOLD_INFO structure to 0.
- This function does not apply to board-level alarms because these alarms do not have settable threshold values.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR**( ) and **ATDV_ERRMSGP**( ) to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

void CheckEvent();
typedef long int(*HDLR)(unsigned long);

void main()
{
   int nDeviceHandle;
   // Register event handler function with srl
   sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

   /*
   .
   .
   Main Processing
   .
   .
   */

   /*
   Query the alarm threshold settings for IP device handle, nDeviceHandle.
   ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
   */
   IPM_QOS_THRESHOLD_INFO myThresholdInfo;
```

```
   myThresholdInfo.unCount = 0;
   if(ipm_GetQoSThreshold(nDeviceHandle, &myThresholdInfo, EV_ASYNC) == -1)
   {
      printf("ipm_GetQoSThreshold failed for device name = %s with error = %d\n",
          ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));

      /*
      .
      .
      Perform Error Processing
      .
      .
      */
   }

   /*
   .
   .
   . continue
   .
   .
   */
}

void CheckEvent()
{
   unsigned int i;
   int nEventType = sr_getevttype();
   int nDeviceID = sr_getevtdev();
   void* pVoid = sr_getevtdatap();
   IPM_QOS_THRESHOLD_INFO* pThresholdInfo;

   switch(nEventType)
   {
      /*
      .
      .
      . Other events
      .
      .
      */

      /* Expected reply to ipm_GetQoSThreshold */
      case IPMEV_GET_QOS_THRESHOLD_INFO:
         pThresholdInfo = (IPM_QOS_THRESHOLD_INFO*)pVoid;
         printf("Received IPMEV_GET_QOS_THRESHOLD_INFO for device = %s\n",
             ATDV_NAMEP(nDeviceID));
         for(i=0; i<pThresholdInfo->unCount; ++i)
         {
            switch(pThresholdInfo->QoSThresholdData[i].eQoSType)
            {

               case QOSTYPE_LOSTPACKETS:
                  printf("QOSTYPE_LOSTPACKETS\n");
                  printf("unTimeInterval = %d\n",
                         pThresholdInfo->QoSThresholdData[i].unTimeInterval);
                  printf("unDebounceOn = %d\n",
                         pThresholdInfo->QoSThresholdData[i].unDebounceOn);
                  printf("unDebounceOff = %d\n",
                         pThresholdInfo->QoSThresholdData[i].unDebounceOff);
                  printf("unFaultThreshold = %d\n",
                         pThresholdInfo->QoSThresholdData[i].unFaultThreshold);
                  printf("unPercentSuccessThreshold = %d\n",
                         pThresholdInfo->QoSThresholdData[i].unPercentSuccessThreshold);
                  printf("unPercentFailThreshold = %d\n",
                         pThresholdInfo->QoSThresholdData[i].unPercentFailThreshold);
                  break;
```

```
                case QOSTYPE_JITTER:
                   printf("QOSTYPE_JITTER\n");
                   printf("unTimeInterval = %d\n",
                          pThresholdInfo->QoSThresholdData[i].unTimeInterval);
                   printf("unDebounceOn = %d\n",
                          pThresholdInfo->QoSThresholdData[i].unDebounceOn);
                   printf("unDebounceOff = %d\n",
                          pThresholdInfo->QoSThresholdData[i].unDebounceOff);
                   printf("unFaultThreshold = %d\n",
                          pThresholdInfo->QoSThresholdData[i].unFaultThreshold);
                   printf("unPercentSuccessThreshold = %d\n",
                          pThresholdInfo->QoSThresholdData[i].unPercentSuccessThreshold);
                   printf("unPercentFailThreshold = %d\n",
                          pThresholdInfo->QoSThresholdData[i].unPercentFailThreshold);
                   break;
             }
          }
          break;

       default:
          printf("Received unknown event = %d for device = %s\n",
                 nEventType, ATDV_NAMEP(nDeviceID));
          break;
    }
}
```

See Also

- **ipm_SetQoSThreshold( )**

ipm_GetSessionInfo( )

Name: int ipm_GetSessionInfo(nDeviceHandle, *pSessionInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_SESSION_INFO *pSessionInfo	• pointer to session info structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: Media Session

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_GetSessionInfo( )** function returns QoS and RTCP statistics for a media session in progress. The data returned is parsed into data structure elements.

If a media session has been initiated by calling **ipm_StartMedia( )**, the data returned by **ipm_GetSessionInfo( )** is for the current session. If **ipm_GetSessionInfo( )** is called between media sessions—that is, after **ipm_Stop( )** terminates the session and before **ipm_StartMedia( )** is called to start a new session—only RTCP statistics are returned for the previous media session. QoS statistics are not available between media sessions.

To receive QoS statistics, QoS monitoring must be enabled. For more information, see **ipm_EnableEvents( )** and Chapter 17, "Quality of Service (QoS) Alarms and RTCP Reports".

Parameter	Description
nDeviceHandle	handle of the IP Media device
pSessionInfo	pointer to structure that contains Quality of Service (QoS) information about the previous IP session. This parameter can be NULL if the function is called in the asynchronous mode. See IPM_SESSION_INFO for details.
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_GET_SESSION_INFO
Indicates successful completion; that is, the structure containing session statistics was filled in. Use SRL functions to retrieve IPM_SESSION_INFO structure fields.

IPMEV_ERROR
Indicates that the function failed.

Cautions

- The application can call **ipm_GetQoSAlarmStatus( )** to retrieve alarm information for the current session.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
Invalid parameter

EIPM_INTERNAL
Internal error

EIPM_INV_MODE
Invalid mode

EIPM_INV_STATE
Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
System error

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    int nDeviceHandle;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    Get the current session information for IP device handle, nDeviceHandle.
    ASSUMPTION: nDeviceHandle was obtained from a prior call to ipm_Open().
    Also, ipm_StartMedia() was successfully called some time earlier.
    */
    if(ipm_GetSessionInfo(nDeviceHandle, NULL, EV_ASYNC) == -1)
    {
        printf("ipm_GetSessionInfo failed for device name = %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
```

```
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }

    /*
    .
    .
    . Continue processing
    .
    .
    */
}

void CheckEvent()
{
    unsigned int i;
    IPM_SESSION_INFO* pIPSessionInfo;
    int nDeviceID = sr_getevtdev();
    int nEventType = sr_getevttype();
    void* pVoid = sr_getevtdatap();

    switch(nEventType)
    {
        /*
        .
        .
        . Other events
        .
        .
        */

        /* Expected reply to ipm_GetSessionInfo */
        case IPMEV_GET_SESSION_INFO:
            pIPSessionInfo = (IPM_SESSION_INFO*)pVoid;
            printf("Received IPMEV_GET_SESSION_INFO for device = %s\n",
                ATDV_NAMEP(nDeviceID));
            printf("RtcpInfo.unLocalSR_TimeStamp=%d\n",
                pIPSessionInfo->RtcpInfo.unLocalSR_TimeStamp);
            printf("RtcpInfo.unLocalSR_TxPackets=%d\n",
                pIPSessionInfo->RtcpInfo.unLocalSR_TxPackets);
            printf("RtcpInfo.unLocalSR_TxOctets=%d\n",
                pIPSessionInfo->RtcpInfo.unLocalSR_TxOctets);
            printf("RtcpInfo.unLocalSR_SendIndication=%d\n",
                pIPSessionInfo->RtcpInfo.unLocalSR_SendIndication);
            printf("RtcpInfo.unLocalRR_FractionLost=%d\n",
                pIPSessionInfo->RtcpInfo.unLocalRR_FractionLost);
            printf("RtcpInfo.unLocalRR_CumulativeLost=%d\n",
                pIPSessionInfo->RtcpInfo.unLocalRR_CumulativeLost);
            printf("RtcpInfo.unLocalRR_SeqNumber=%d\n",
                pIPSessionInfo->RtcpInfo.unLocalRR_SeqNumber);
            printf("RtcpInfo.unLocalRR_ValidInfo=%d\n",
                pIPSessionInfo->RtcpInfo.unLocalRR_ValidInfo);
            printf("RtcpInfo.unRemoteSR_TimeStamp=%d\n",
                pIPSessionInfo->RtcpInfo.unRemoteSR_TimeStamp);
            printf("RtcpInfo.unRemoteSR_TxPackets=%d\n",
                pIPSessionInfo->RtcpInfo.unRemoteSR_TxPackets);
            printf("RtcpInfo.unRemoteSR_TxOctets=%d\n",
                pIPSessionInfo->RtcpInfo.unRemoteSR_TxOctets);
            printf("RtcpInfo.unRemoteSR_SendIndication=%d\n",
                pIPSessionInfo->RtcpInfo.unRemoteSR_SendIndication);
            printf("RtcpInfo.unRemoteRR_FractionLost=%d\n",
                pIPSessionInfo->RtcpInfo.unRemoteRR_FractionLost);
```

```
            printf("RtcpInfo.unRemoteRR_CumulativeLost=%d\n",
                pIPSessionInfo->RtcpInfo.unRemoteRR_CumulativeLost);
            printf("RtcpInfo.unRemoteRR_SeqNumber=%d\n",
                pIPSessionInfo->RtcpInfo.unRemoteRR_SeqNumber);
            printf("RtcpInfo.unRemoteRR_ValidInfo=%d\n",
                pIPSessionInfo->RtcpInfo.unRemoteRR_ValidInfo);

            for(i = 0; i< pIPSessionInfo->unQoSInfoCount; ++i)
            {
                printf("Session QOS Type=%d\n", pIPSessionInfo->QoSInfo[i].eQoSType);
                printf("Session QOS Data=%d\n", pIPSessionInfo->QoSInfo[i].unData);
            }
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
             break;
    }
}
```

■ See Also

- **ipm_GetQoSAlarmStatus( )**
- **ipm_GetSessionInfoEx( )**
- **ipm_StartMedia( )**

ipm_GetSessionInfoEx( )

Name: int ipm_GetSessionInfoEx(nDeviceHandle, usDirection)

Inputs:

int nDeviceHandle	• IP Media device handle
unsigned short usDirection	• direction of RTCP report

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: Media Session

Mode: asynchronous

Platform: Dialogic® HMP Software (Linux)

■ Description

The **ipm_GetSessionInfoEx( )** function retrieves RTCP data for a media session if one is in progress; otherwise, it retrieves data for the previous session. This function can retrieve both incoming and outgoing RTCP data that may include extended (RTCP-XR) and high resolution VoIP metrics (RTCP-HR).

The RTCP packets are in a raw, unrendered format. The RTCP report is returned as an event data associated with IPMEV_GET_SESSIONINFOEX.

If a media session has been initiated by calling **ipm_StartMedia( )**, the data returned by **ipm_GetSessionInfoEx( )** is for the current session. If **ipm_GetSessionInfoEx( )** is called between media sessions—that is, after **ipm_Stop( )** terminates the session and before **ipm_StartMedia( )** is called to start a new session—the data returned is for that previous media session.

Before using this function, you can enable events for enhanced RTCP reporting using **ipm_EnableEvents( )**.

Parameter	Description
nDeviceHandle	handle of the IP Media device
usDirection	direction of RTCP report requested. Valid values are: IPM_RTCP_DIR_INCOMING and IPM_RTCP_DIR_OUTGOING.

■ Termination Events

IPMEV_GET_SESSION_INFOEX
Indicates successful completion: the IPM_SESSION_INFOEX structure that contains enhanced RTCP statistics was filled in. Use SRL functions to retrieve event data associated with the structure.

IPMEV_ERROR
Indicates that the function failed.

Cautions

None.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_INTERNAL
: Internal error

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    int nDeviceHandle;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    // Enable RTCP reporting

    IPM_PARM_INFO ParmInfo;
    unsigned long ulParmValue = 1
    ParmInfo.eParm = PARMCH_ RTCP_ENHANCED_REPORTING;
    ParmInfo.pvParmValue = &ulParmValue;

    if(ipm_SetParm(nDeviceHandle, &ParmInfo, EV_ASYNC)==-1)
    {
      // Error Processing
    }
             ………………..
             ………………..
    // Event Processing

    void CheckEvent()
    {
    // Set frequency of RTCP reporting events
```

```
 printf("Setting Event Rate\n");
    Parms.eParm=PARMCH_RTCP_ENHANCED_EVENT_FREQ;
    value=5; // send every fifth event
    rc=ipm_SetParm(l_Channels[i].m_ipm_ddd,&Parms,EV_ASYNC);
    if (rc<0)
    {
            printf("Error setting event rate parm\n");
            return -1;
    }

int nEventType = sr_getevttype();
int nDeviceID = sr_getevtdev();
void* pVoid = sr_getevtdatap();
switch(nEventType)
{
    /* Expected reply to ipm_GetQoSAlarmStatus */
    case IPMEV_SET_PARM:
        printf("Received IPMEV_SETPARM for device = %s\n"ATDV_NAMEP(nDeviceID));
        break;
    case IPMEV_RTCP_NOTIFY_RECEIVED:
        printf("%i IPMEV_RTCP_NOTIFY_RECEIVED\n",index);
        rc=ipm_GetSessionInfoEx(l_Channels[index].m_ipm_ddd,1);
        if (rc == -1) {
           printf("ipm_GetSessionInfoEx failed: channel %i
             %s\n",index,ATDV_ERRMSGP(l_Channels[index].m_ipm_ddd));
    }
break;

case IPMEV_RTCP_NOTIFY_SENT:
    printf("%i IPMEV_RTCP_NOTIFY_SENT\n",index);
    rc=ipm_GetSessionInfoEx(l_Channels[index].m_ipm_ddd,2);
    if (rc == -1) {
       printf("ipm_GetSessionInfoEx failed: channel %i
         %s\n",index,ATDV_ERRMSGP(l_Channels[index].m_ipm_ddd));
}
break;

case IPMEV_GET_SESSION_INFOEX:
    printf("Received IPMEV_GET_SESSION_INFOEX for device =
       %s\n",ATDV_NAMEP(l_Channels[index].m_ipm_ddd));
    pSessionInfoEx = (IPM_SESSION_INFOEX *)pVoid;
    //pSessionInfoEx = (unsigned char *)pVoid;
    if (pSessionInfoEx!=NULL)

int i;
            unsigned char *data;
            unsigned int version;

            printf("version=%d\n",pSessionInfoEx->unVersion);
            printf("direction=%u\n",pSessionInfoEx->unRTCPDirection);
            printf("DataLength=%u\n",pSessionInfoEx->unRTCPDataLength);
            data=(unsigned char*)pSessionInfoEx->pRTCPReport;

            if((pSessionInfoEx->unRTCPDataLength > 0) &&
               (pSessionInfoEx->unVersion == IPM_SESSION_INFOEX_VERSION_0))
            {
              data = (unsigned char *)malloc(pSessionInfoEx->unRTCPDataLength *
                sizeof(unsigned char));
              memcpy(data, pSessionInfoEx->pRTCPReport, pSessionInfoEx->unRTCPDataLength);
              for (i=0;i<pSessionInfoEx->unRTCPDataLength;i++)
              {
                    printf("%i=%02x\n",i,data[i]);
              }
            }
    }
```

■ See Also

- **ipm_GetSessionInfo( )**

ipm_GetXmitSlot( )

Name: int ipm_GetXmitSlot(nDeviceHandle, *pTimeslotInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
SC_TSINFO *pTimeslotInfo	• pointer to time slot info structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: System Control

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_GetXmitSlot( )** function returns TDM time slot information for an IP channel.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pTimeslotInfo	pointer to structure that describes the time slot number, time slot type, and bus encoding format. This parameter can be NULL if the function is called in the asynchronous mode. See SC_TSINFO for details.
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_GET_XMITTS_INFO
Indicates successful completion. Use SRL functions to retrieve SC_TSINFO structure fields.

IPMEV_ERROR
Indicates that the function failed.

■ Cautions

None

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_FWERROR
: Firmware error

EIPM_INTERNAL
: Internal error

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

void CheckEvent();
typedef long int(*HDLR)(unsigned long);

void main()
{
    int nDeviceHandle;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    Get the timeslot information for IP device handle, nDeviceHandle.
    ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
    */
    if(ipm_GetXmitSlot(nDeviceHandle, NULL, EV_ASYNC) == -1)
    {
        printf("ipm_GetXmitSlot failed for device name = %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }
```

```
    /*
    .
    .
    . continue
    .
    .
    */
}

void CheckEvent()
{
    int nEventType = sr_getevttype();
    int nDeviceID = sr_getevtdev();
    void* pVoid = sr_getevtdatap();
    SC_TSINFO* pTimeSlotInfo;

    switch(nEventType)
    {
        /*
        .
        .
        . Other events
        .
        .
        */

        /* Expected reply to ipm_GetXmitSlot */
        case IPMEV_GET_XMITTS_INFO:
            pTimeSlotInfo = (SC_TSINFO*)pVoid;
            printf("Received IPMEV_GET_XMITTS_INFO for device = %s\n",
                ATDV_NAMEP(nDeviceID));
            printf("Timeslot number %d\n", *(pTimeSlotInfo->sc_tsarrayp));
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

■ See Also

None

ipm_InitResponseSend( )

Name: int ipm_InitResponseSend (a_nDeviceHandle, pInitRsp)

Inputs:

int a_nDeviceHandle	• SRL handle of the IPM device
const PIPM_NBUP_INIT_RESPONSE a_pInitRsp	• structure describing the Nb UP response

Returns: 0 on success
-1 on error

Includes: ipmlib.h

Category: Media Session

Mode: asynchronous

Platform: Dialogic® HMP Software (Linux), Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

---

■ Description

The **ipm_InitResponseSend( )** function attempts to send a response to an IP session (for example, Nb UP session) invitation. The **ipm_StartMedia( )** function must have been previously called with the appropriate connection type, such as Nb UP.

Parameter	Description
a_nDeviceHandle	SRL handle of the IP media device
a_pInitRsp	structure describing the Nb UP response. See IPM_NBUP_INIT_RESPONSE for details.

■ Events

If the function returns 0, it can generate any of the following events:

IPMEV_INIT_RESPONSE_SEND
Indicates successful completion of sending the initialization response message. It does not indicate whether the event was received by the remote party. If IPMEV_INIT_RESPONSE_SEND is received, one of the following unsolicited events is subsequently reported:

- IPMEV_INIT_COMPLETE – Unsolicited event reported when the Nb UP session has been successfully negotiated with the remote party. Both parties may now begin exchanging data.
- IPMEV_INIT_FAILED – Unsolicited event reported upon encountering an unexpected error during initialization. Event Data: IPM_NBUP_INIT_FAILURE structure.

IPMEV_ERROR
Indicates that the function failed.

Cautions

None.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_BUSY
: Channel is busy

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include "srllib.h"
#include "dxxxlib.h"
#include "ipmlib.h"

static int g_hIPM = -1;
static IPM_MEDIA_INFO g_media, g_localMedia;
static bool g_mediaStarted;


/***********************************************************************
* Dialogic API function callers
***********************************************************************/
bool
GetLocalMediaInfo ()
{
  g_localMedia.unCount = 1;
  g_localMedia.MediaData[0].eMediaType = MEDIATYPE_NBUP_LOCAL_RTP_INFO;
  printf("Calling ipm_GetLocalMediaInfo()\n");
  if (ipm_GetLocalMediaInfo (g_hIPM, &g_localMedia, EV_ASYNC) == -1)
    {
      printf ("ipm_GetLocalMediaInfo failed\n");
      return false;
    }
  return true;
}

bool
StartMedia ()
{
```

```
  // Set up remote RTP Media Info
  memset(&g_media, 0, sizeof(IPM_MEDIA_INFO));
  g_media.unCount = 2;
  g_media.MediaData[0].eMediaType = MEDIATYPE_NBUP_REMOTE_RTP_INFO;
  g_media.MediaData[0].mediaInfo.PortInfo.unPortId = 40960;
  //    g_localMedia.MediaData[0].mediaInfo.PortInfo.unPortId;
  strcpy (g_media.MediaData[0].mediaInfo.PortInfo.cIPAddress,
     g_localMedia.MediaData[0].mediaInfo.PortInfo.cIPAddress);
  // Set up NBUP info
  g_media.MediaData[1].eMediaType = MEDIATYPE_NBUP_PROFILE_INFO;
  INIT_IPM_NBUP_PROFILE_INFO (&(g_media.MediaData[1].mediaInfo.NBUPProfileInfo));
  g_media.MediaData[1].mediaInfo.NBUPProfileInfo.eProfileType = NBUP_PROFILE_3G324M;
  g_media.MediaData[1].mediaInfo.NBUPProfileInfo.ucTxPLT = 111;
  g_media.MediaData[1].mediaInfo.NBUPProfileInfo.ucRxPLT = 111;
  printf("Remote IP address - %s   Remote RTP port - %d\n",
      g_media.MediaData[0].mediaInfo.PortInfo.cIPAddress,
      g_media.MediaData[0].mediaInfo.PortInfo.unPortId);
  printf("Calling ipm_StartMedia()\n");
  if (ipm_StartMedia (g_hIPM, &g_media, DATA_IP_TDM_BIDIRECTIONAL, EV_ASYNC) == -1)
    {
      printf ("ipm_StartMedia failed\n");
      return false;
    }
  g_mediaStarted = true;
  return true;
}

bool
InitResponseSend ()
{
  IPM_INIT_RESPONSE initResp;
  IPM_NBUP_INIT_RESPONSE nbupResp;
  // Initialize the main structure
  INIT_IPM_INIT_RESPONSE (&initResp);
  initResp.eProtocol = RTP_PROTOCOL_NBUP;
  initResp.data.pNBUP = &nbupResp;
  // Initialize the NBUP specific structure with an ACK response
  INIT_IPM_NBUP_INIT_RESPONSE (&nbupResp);
  nbupResp.eResponse = NBUP_INIT_RESPONSE_ACK;
  printf("Calling ipm_InitResonseSend()\n");
  if (ipm_InitResponseSend (g_hIPM, &initResp) == -1)
    {
      printf ("ipm_InitResponseSend failed\n");
      return false;
    }
  return true;
}

bool
EnableEvents ()
{
  const int nEvents = 4;
  eIPM_EVENT events[nEvents] = {
    EVT_INIT_RECEIVED,
    EVT_PROCEDURE_DONE,
    EVT_SEND_FAILED,
    EVT_NACK_SENT
  };
  printf("Calling ipm_EnableEvents()\n");
  if (ipm_EnableEvents (g_hIPM, events, nEvents, EV_ASYNC) == -1)
    {
      printf ("ipm_EnableEvents failed\n");
      return false;
    }
  return true;
}
```

```
/**********************************************************************
Event handler for Inbound IPM device
NOTE: Returning 1 will cause waitevt(-1) to stop blocking
and exit the state machine.
**********************************************************************/
long
IpmEventHandlerInbound (void *dummy)
{
// Get the event info from SRL
  int evttype = sr_getevttype ();
  int evtdev = sr_getevtdev ();
  if (evtdev == -1)
    return 0;
  printf("Event rcvd - 0x%x", evttype);

  switch (evttype)
    {
    case IPMEV_EVENT_ENABLED:
      printf (" - IPMEV_EVENT_ENABLED\n");
      // Get local Media Info to be used in ipm_StartMedia
      if (!GetLocalMediaInfo ())
   {
     //return 0;
     //return 1;
   }
      break;
    case IPMEV_GET_LOCAL_MEDIA_INFO:
      printf (" - IPMEV_GET_LOCAL_MEDIA_INFO\n");
      {
   // Search for the local media info in the returned data structure
   IPM_MEDIA_INFO *pMediaInfo = (IPM_MEDIA_INFO *) sr_getevtdatap ();
   for (unsigned int ii = 0; ii < pMediaInfo->unCount; ii++)
     {
       switch (pMediaInfo->MediaData[ii].eMediaType)
         {
         case MEDIATYPE_NBUP_LOCAL_RTP_INFO:
      // Found local media info in the returned event, store it away.
      memcpy (&(g_localMedia.MediaData[0]),
         &(pMediaInfo->MediaData[ii]), sizeof (IPM_MEDIA));
      // Call StartMedia.  Should not need this "already started" check...
        if (!g_mediaStarted)
        if (!StartMedia ())
        {
          return 1;
        }
                 break;
         default:
      printf ("Unexpected media type %0x%x found.\n",
         pMediaInfo->MediaData[ii].eMediaType);
      break;
         }
     }
   break;
    case IPMEV_STARTMEDIA:
   printf (" - IPMEV_STARTMEDIA\n");
   // Sit and wait for an inbound NBUP session request
   break;
      case IPMEV_INIT_RECEIVED:
   printf (" - IPMEV_INIT_RECEIVED\n");
   {
     // Check to see what type of invite we received
     PIPM_INIT_RECEIVED pRecv = (PIPM_INIT_RECEIVED) sr_getevtdatap ();
     if (!pRecv)
       {
         printf ("Invalid pointer returned from sr_getevtdatap()");
         return 1;
       }
```

```
      if (pRecv->eProtocol == RTP_PROTOCOL_NBUP)
        {
          // We've received an NBUP invitation. Send an ACK to start the session
          if (!InitResponseSend ())
       {
         return 1;
       }
        }
      else
        {
          printf ("Unsupported protocol (%d) in init request.",
             pRecv->eProtocol);
          return 1;
        }
    }
    break;
       case IPMEV_INIT_RESPONSE_SEND:
    printf (" - IPMEV_INIT_RESPONSE_SEND\n");
    // Just the termination event for ipm_InitResponseSend
    break;
       case IPMEV_INIT_COMPLETE:
    printf (" - IPMEV_INIT_COMPLETE event received"
       " - NBUP CONNECTION SUCCESSFUL\n");
    // This means all has passed and we have an NBUP session.
    // Just exit out of our sample scenario here
    return 1;
       case IPMEV_INIT_FAILED:
    // Something prevented us from connecting
    printf (" - IPMEV_INIT_FAILED\n");
    {
      PIPM_INIT_FAILURE pInfo = (PIPM_INIT_FAILURE) sr_getevtdatap ();
      if (pInfo)
        {
          if (pInfo->eProtocol == RTP_PROTOCOL_NBUP)
       {
         PIPM_NBUP_INIT_FAILURE pNBUP =
           (PIPM_NBUP_INIT_FAILURE) (pInfo->data.pNBUP);
         printf ("Error cause: 0x%X\n", pNBUP->unCause);
       }
          else
       {
         printf ("Unexpected protocol %d found.\n",
            pInfo->eProtocol);
       }
        }
      else
        {
         printf ("Bad pointer returned from sr_getevtdatap()\n");
       }
    }
    return 1;
       case IPMEV_ERROR:
    printf (" - IPMEV_ERROR %d: %s\n", ATDV_LASTERR (g_hIPM),
       ATDV_ERRMSGP (g_hIPM));
    return 1;
       default:
    printf ("WARNING: Unknown event 0x%x received.\n", evttype);
    break;
       }         // end switch
       return 0;
     }
   return 0;
 }
/************************************************************************
MAIN procedure
************************************************************************/
int main ()
```

```
{
  g_mediaStarted = false;
  printf("Calling ipm_Open() on %s\n", ipmB1C2);
  g_hIPM = ipm_Open (ipmB1C2, NULL, EV_SYNC);
  if (g_hIPM == -1)
    {
      printf ("ipm_Open() failed.\n");
      exit (1);
    }
  if (sr_enbhdlr (g_hIPM, EV_ANYEVT, IpmEventHandlerInbound) == -1)
    {
      printf ("sr_enbhdlr() failed.\n");
      exit (1);
    }
  // Enable the NBUP events and kick off the state machine
  if (!EnableEvents ())
    {
      exit (1);
    }
  // Wait for complete
  sr_waitevt (-1);

  printf ("NbUP data can now be sent\n");
  sleep (4);

  // Shutdown
  printf("Exiting...\n");
  printf("Calling ipm_Stop()\n");
  if (ipm_Stop (g_hIPM, STOP_ALL, EV_SYNC == -1))
    {
      printf ("ipm_Stop failed.\n");
    }
  if (sr_dishdlr (g_hIPM, EV_ANYEVT, IpmEventHandlerInbound) == -1)
    {
      printf ("sr_dishdlr() failed.\n");
    }

  printf("Calling ipm_Close()\n");
  if (ipm_Close (g_hIPM, NULL) == -1)
    {
      printf ("ipm_Close() failed.\n");
    }
}
```

■ See Also

- **ipm_InitSend( )**

ipm_InitSend( )

Name:	int ipm_InitSend (a_nDeviceHandle, *a_pInitInfo)	
Inputs:	int a_nDeviceHandle	• SRL handle of the IPM device
	const IPM_INIT_SEND a_pInitInfo	• structure describing the Nb UP initialization parameters
Returns:	0 on success -1 on error	
Includes:	ipmlib.h	
Category:	Media Session	
Mode:	asynchronous	
Platform:	Dialogic® HMP Software (Linux), Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe	

■ Description

The **ipm_InitSend**( ) function attempts to send an IP protocol (for example, Nb UP) message to a remote party. The **ipm_StartMedia( )** function must have been previously called with the appropriate connection type, such as Nb UP.

Parameter	Description
a_nDeviceHandle	SRL handle of the IP media device
a_pInitInfo	structure describing the Nb UP initialization parameters. See IPM_INIT_SEND for details.

■ Events

If the function returns 0, it can generate any of the following events:

IPMEV_INIT_SEND
: Indicates successful completion of sending the initialization message. It does not indicate whether the event was received by remote party or what the response was. If IPMEV_INIT_SEND is received, one of the following unsolicited events is subsequently reported:
  - IPMEV_INIT_COMPLETE – Unsolicited event reported when the Nb UP session has been successfully negotiated with the remote party. Both parties may now begin exchanging data.
  - IPMEV_INIT_FAILED – Unsolicited event reported upon encountering an unexpected error or a negative response (NACK) from the remote party. Event Data: IPM_INIT_FAILURE structure.

IPMEV_ERROR
: Indicates that the function failed.

Cautions

None.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** functions to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_BUSY
: Channel is busy

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

Example

```
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include "srllib.h"
#include "dxxxlib.h"
#include "ipmlib.h"

static int g_hIPM = -1;
static IPM_MEDIA_INFO g_media, g_localMedia;
static bool g_mediaStarted;


/***************************************************************************
* Dialogic API function callers
***************************************************************************/
  bool GetLocalMediaInfo ()
{
  g_localMedia.unCount = 1;
  g_localMedia.MediaData[0].eMediaType = MEDIATYPE_NBUP_LOCAL_RTP_INFO;
  printf ("Calling ipm_GetLocalMediaInfo()\n");
  if (ipm_GetLocalMediaInfo (g_hIPM, &g_localMedia, EV_ASYNC) == -1)

    {
      printf ("ipm_GetLocalMediaInfo failed\n");
      return false;
    }
  return true;
}

bool StartMedia ()
{
  // Set up remote RTP Media Info
  memset(&g_media, 0, sizeof(IPM_MEDIA_INFO));
```

```
  g_media.unCount = 2;
  g_media.MediaData[0].eMediaType = MEDIATYPE_NBUP_REMOTE_RTP_INFO;
  g_media.MediaData[0].mediaInfo.PortInfo.unPortId = 40962;
  strcpy (g_media.MediaData[0].mediaInfo.PortInfo.cIPAddress,
     g_localMedia.MediaData[0].mediaInfo.PortInfo.cIPAddress);

  // Set up NBUP info
  g_media.MediaData[1].eMediaType = MEDIATYPE_NBUP_PROFILE_INFO;
  INIT_IPM_NBUP_PROFILE_INFO (&(g_media.MediaData[1].mediaInfo.NBUPProfileInfo));
  g_media.MediaData[1].mediaInfo.NBUPProfileInfo.eProfileType = NBUP_PROFILE_3G324M;
  g_media.MediaData[1].mediaInfo.NBUPProfileInfo.ucTxPLT = 111;
  g_media.MediaData[1].mediaInfo.NBUPProfileInfo.ucRxPLT = 111;
  printf("Remote IP address - %s        Remote RTP port - %d\n",
              g_media.MediaData[0].mediaInfo.PortInfo.cIPAddress,
              g_media.MediaData[0].mediaInfo.PortInfo.unPortId);
  printf("Calling ipm_StartMedia()\n");
  if (ipm_StartMedia
      (g_hIPM, &g_media, DATA_IP_TDM_BIDIRECTIONAL, EV_ASYNC) == -1)

    {
      printf ("ipm_StartMedia failed\n");
      return false;
    }
  g_mediaStarted = true;
  return true;
}

bool InitSend ()
{

  // Initiate the NBUP session
  IPM_INIT_SEND initSend;
  IPM_NBUP_INIT_SEND nbupSend;

  // Initialize the main structure
  INIT_IPM_INIT_SEND (&initSend);
  initSend.eProtocol = RTP_PROTOCOL_NBUP;
  initSend.data.pNBUP = &nbupSend;

  // Initialize the NBUP specific structure
  INIT_IPM_NBUP_INIT_SEND (&nbupSend, 1, 1);

  // Init function handles memory allocation for us
  nbupSend.pRFCIs[0].ucID = (unsigned char) 0x44;
  nbupSend.pRFCIs[0].pSubFlows[0].eFlowSize = NBUP_FLOW_SIZE_320_BITS;
  printf("Calling ipm_initSend()\n");
  if (ipm_InitSend (g_hIPM, &initSend) == -1)

    {

      // De-allocate memory allocated for us
      FREE_IPM_NBUP_INIT_SEND (&nbupSend);
      printf ("ipm_InitSend failed\n");
      return false;
    }

  // De-allocate memory allocated for us
  FREE_IPM_NBUP_INIT_SEND (&nbupSend);
  return true;
}

bool EnableEvents ()
{
  const int nEvents = 4;
  eIPM_EVENT events[nEvents] =
    {
      EVT_INIT_RECEIVED, EVT_PROCEDURE_DONE, EVT_SEND_FAILED,
```

```
      EVT_NACK_SENT };
  printf("Calling ipm_EnableEvents()\n");
  if (ipm_EnableEvents (g_hIPM, events, nEvents, EV_ASYNC) == -1)

    {
      printf ("ipm_EnableEvents failed\n");
      return false;
    }
  return true;
}


  /**********************************************************************
  Event handler for Outbound IPM device
  NOTE: Returning 1 will cause waitevt(-1) to stop blocking
  and exit the state machine.
  **********************************************************************/
  long
  IpmEventHandlerOutbound (void *dummy)
  {

    unsigned long evthandle = sr_getevtdev();

    // Get the event info from SRL
    int evttype = sr_getevttype ();
    int evtdev = sr_getevtdev ();
    if (evtdev == -1)
      return 0;
    printf("Event rcvd - 0x%x", evttype);
    switch (evttype)

      {
      case IPMEV_EVENT_ENABLED:
        printf (" - IPMEV_EVENT_ENABLED\n");

        // Get local Media Info to be used in ipm_StartMedia
        if (!GetLocalMediaInfo ())

     {
       //return 1;
     }
        break;
      case IPMEV_GET_LOCAL_MEDIA_INFO:
        printf (" - IPMEV_GET_LOCAL_MEDIA_INFO\n");

        {

     // Search for the local media info in the returned data structure
     {
             IPM_MEDIA_INFO * pMediaInfo = (IPM_MEDIA_INFO *) sr_getevtdatap ();
       for (unsigned int ii = 0; ii < pMediaInfo->unCount; ii++)

         {
           switch (pMediaInfo->MediaData[ii].eMediaType)

        {
        case MEDIATYPE_NBUP_LOCAL_RTP_INFO:

          // Found local media info in the returned event, store it away.
          memcpy (&(g_localMedia.MediaData[0]),
              &(pMediaInfo->MediaData[ii]), sizeof (IPM_MEDIA));

          // Call StartMedia.  Should not need this "already started" check.
            if (!g_mediaStarted)
            if (!StartMedia ())
            {
              return 1;
```

```
          }
             break;
       default:
         printf ("Unexpected media type 0x%x found.\n",
            pMediaInfo->MediaData[ii].eMediaType);
         break;
       }
        }
    }
       }
       break;
     case IPMEV_STARTMEDIA:
       printf (" - IPMEV_STARTMEDIA event received\n");

       // Initiate the NBUP session
       if (!InitSend ())
    {
      return 1;
    }
       break;
     case IPMEV_INIT_SEND:
       printf (" - IPMEV_INIT_SEND\n");

       // Just the termination event for ipm_InitSend - take no action
       break;
     case IPMEV_INIT_COMPLETE:
       printf (" - IPMEV_INIT_COMPLETE"
          " - NBUP CONNECTION SUCCESSFUL\n");

       // This means all has passed and we have an NBUP session.
       // Just exit out of our sample scenario here
       return 1;
     default:
       printf ("WARNING: Unknown event 0x%x received.\n", evttype);
       break;
     }
  return 0;
}

/**************************************************************************
MAIN procedure
**************************************************************************/
int main ()
{
  g_mediaStarted = false;
  printf("Calling ipm_Open on  %s\n", ipmB1C1);
  g_hIPM = ipm_Open (ipmB1C1, NULL, EV_SYNC);
  if (g_hIPM == -1)

    {
      printf ("ipm_Open failed.\n");
      exit (1);
    }
  if (sr_enbhdlr (g_hIPM, EV_ANYEVT, IpmEventHandlerOutbound) == -1)

    {
      printf ("sr_enbhdlr() failed.\n");
      exit (1);
    }

// Enable the NBUP events and kick off the state machine
    if (!EnableEvents ())

    {
      exit (1);
    }
```

```
// Wait for complete
    sr_waitevt (-1);

  printf ("NbUP data can now be sent\n");
  sleep (4);

// Shutdown
    printf("Exiting...\n");
    printf("Calling ipm_Stop()\n");
    if (ipm_Stop (g_hIPM, STOP_ALL, EV_SYNC == -1))

    {
      printf ("ipm_Stop failed.\n");
    }
  if (sr_dishdlr (g_hIPM, EV_ANYEVT, IpmEventHandlerOutbound) == -1)

    {
      printf ("sr_dishdlr() failed.\n");
    }
  printf("Calling ipm_Close()\n");
  if (ipm_Close (g_hIPM, NULL) == -1)

    {
      printf ("ipm_Close() failed.\n");
    }
}
```

See Also

- **ipm_InitResponseSend( )**

ipm_Listen( )

Name: int ipm_Listen(nDeviceHandle, *pTimeslotInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
SC_TSINFO *pTimeslotInfo	• pointer to time slot info structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: System Control

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_Listen( )** function connects an IP channel to a TDM time slot, enabling data to flow between the TDM time slot and the IP network or the host.

ipm_Listen( ) uses the information stored in the SC_TSINFO structure to connect the receive channel on the device to an available TDM bus time slot in the specified list of time slots. The time slot number is returned in the SC_TSINFO structure. The receive channel remains connected to the TDM bus time slot until **ipm_UnListen( )** is called or **ipm_Listen( )** is called with a different time slot.

If **ipm_Listen( )** is called to connect to a different TDM time slot, the firmware automatically breaks an existing connection and reconnects it to the new time slot. In this case, the application does not need to call the **ipm_UnListen( )** function.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pTimeslotInfo	pointer to structure that describes the time slot number, time slot type, and bus encoding format See SC_TSINFO for details.
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_LISTEN
Indicates successful completion; that is, an IP channel was connected to the specified TDM time slot. This event does not return any data.

IPMEV_ERROR
: Indicates the function failed.

■ Cautions

- The IP Media library allows **ipm_Listen( )** and **ipm_UnListen( )** to be called either synchronously or asynchronously. Other Dialogic libraries may not support asynchronous execution of the similar **xx_Listen** and **xx_UnListen** functions.

■ Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_FWERROR
: Firmware error

EIPM_INTERNAL
: Internal error

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    int nDeviceHandle;
    SC_TSINFO IPTimeSlotInfo;
    long lTimeSlot;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    Tell IP device handle, nDeviceHandle, to listen to timeslot 10.
    ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
    */
    lTimeSlot = 10;
    IPTimeSlotInfo.sc_tsarrayp = &lTimeSlot;
    IPTimeSlotInfo.sc_numts = 1;
```

```
    if(ipm_Listen(nDeviceHandle, &IPTimeSlotInfo, EV_ASYNC) == -1)
    {
        printf("ipm_Listen failed for device name = %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }

    /*
    .
    .
    . Continue processing
    .
    .
    */
}

void CheckEvent()
{
    int nDeviceID = sr_getevtdev();
    int nEventType = sr_getevttype();

    switch(nEventType)
    {
        /*
        .
        .
        . Other events
        .
        .
        */

        /* Expected reply to ipm_Listen */
        case IPMEV_LISTEN:
            printf("Received IPMEV_LISTEN for device = %s\n", ATDV_NAMEP(nDeviceID));
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

■ See Also

- **ipm_UnListen( )**

ipm_ModifyMedia( )

Name:	int ipm_ModifyMedia(nDeviceHandle, *pMediaInfo, eDirection, usMode)	
Inputs:	int nDeviceHandle	• IP Media device handle
	IPM_MEDIA_INFO *pMediaInfo	• pointer to media information structure
	eIPM_DATA_DIRECTION eDirection	• data flow direction
	unsigned short usMode	• async or sync mode setting
Returns:	0 on success -1 on failure	
Includes:	srllib.h ipmlib.h	
Category:	Media Session	
Mode:	asynchronous or synchronous	
Platform:	Dialogic® HMP Software	

■ Description

The **ipm_ModifyMedia( )** function modifies various properties of an active media session. This function allows the application to modify the following media session properties:

- direction of the media stream
- remote IP address and port

For this function to complete successfully, the stream associated with the IP device must be in either active or suspended mode.

The media session properties are changed on the local endpoint as soon the function is called, and this may result in a perceptible artifact (for example, a click or a brief silence) until the remote endpoint makes the corresponding change. For example, if the coder is being changed by the function call, the local endpoint begins transmitting packets using the new coder and stops accepting packets that it receives which use the old coder as soon as the function executes.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pMediaInfo	pointer to structure that contains local channel RTP/RTCP ports and IP address information (or T.38 port and IP address information) See the IPM_MEDIA_INFO data structure page for details.

Parameter	Description
eDirection	media operation enumeration The eIPM_DATA_DIRECTION data type is an enumeration which defines the following values: • DATA_IP_RECEIVEONLY – receive data from the IP network but do not send data • DATA_IP_SENDONLY – send data to the IP network but do not receive data • DATA_IP_TDM_BIDIRECTIONAL – full duplex data path (streaming media) between IP network and TDM • DATA_IP_INACTIVE – allow RTCP while blocking RTP or T.38 packets
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

Termination Events

In asynchronous mode, the function returns 0 if the operation was initiated successfully.

Completion of the operation is indicated by receipt of a termination event:

IPMEV_MODIFYMEDIA
: Indicates successful completion; that is, modified media information was set and the session has been started.

IPMEV_MODIFYMEDIA_FAIL
: Indicates that the modify media operation failed. The characteristics of the media session remain as they were before the function was called.

Cautions

- An application must ensure that the streaming properties at two endpoints are in sync. For example, changing coder properties at one endpoint without synchronization with the other endpoint will cause dropped RTP packets.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter; invalid coder; invalid count of media information; T.38 is in session; multicast IP address specified; multicast server or client direction.

EIPM_FWERR
: error in lower-level software

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
Invalid state. Initial command did not complete before another function call was made. Stream is idle.

EIPM_SYSTEM
System error

The **ipm_ModifyMedia( )** function returns security errors in the following circumstances:

- The total number of keys has exceeded the given limit (IPM_SECURITY_MAX_TOTAL_NUM_KEYS, which is set to 20).
- The number of keys for any given media type exceeds the given limit (IPM_SECURITY_MAX_NUM_KEYS, which is set to 10).
- The size of a generated key does not match the key size (master key or master salt) in the specified crypto suite (see Table 7, "Crypto Suite Parameter Values", on page 287).
- The Secure RTP feature is not available.
- All Secure RTP resources are allocated.
- Any of the structure version fields are invalid.

Example

The following sample code changes the coder from G.711 mu-law to G.711 A-law and also changes the direction.

```
#include <stdio.h>
#include <string>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
   /*
   .
   .
   Main Processing
   .
   .
   */

   /*
   Set the media properties for a remote party using IP device handle, nDeviceHandle.
   ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
   */

   IPM_MEDIA_INFO MediaInfo;
   MediaInfo.unCount = 4;

   MediaInfo.MediaData[0].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTP_INFO;
   MediaInfo.MediaData[0].mediaInfo.PortInfo.unPortId = 2328;
   strcpy(MediaInfo.MediaData[0].mediaInfo.PortInfo.cIPAddress, "111.21.0.9");

   MediaInfo.MediaData[1].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTCP_INFO;
   MediaInfo.MediaData[1].mediaInfo.PortInfo.unPortId = 2329;
   strcpy(MediaInfo.MediaData[1].mediaInfo.PortInfo.cIPAddress, "111.41.0.9");
```

```
MediaInfo.MediaData[2].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.unCoderPayloadType = 0;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.unRedPayloadType = 0;

MediaInfo.MediaData[3].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.unCoderPayloadType = 0;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.unRedPayloadType = 0;

if (ipm_StartMedia(nDeviceHandle, &MediaInfo, DATA_IP_TDM_BIDIRECTIONAL, EV_SYNC) == -1)
{
   printf("ipm_StartMediaInfo failed for device name = %s with error = %d\n",
   ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
   /*
   .
   .
   Perform Error Processing
   .
   .*/
}
/*
.
. Continue processing
.
*/

MediaInfo.unCount = 2;
MediaInfo.MediaData[0].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ALAW64K;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.unCoderPayloadType = 0;
MediaInfo.MediaData[0].mediaInfo.CoderInfo.unRedPayloadType = 0;

MediaInfo.MediaData[1].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ALAW64K;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.unCoderPayloadType = 0;
MediaInfo.MediaData[1].mediaInfo.CoderInfo.unRedPayloadType = 0;


if (ipm_ModifyMedia(nDeviceHandle, &MediaInfo, DATA_IP_SENDONLY, EV_SYNC) == -1)
{
   printf("ipm_Modify failed for device name = %s with error = %d\n",
   ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
   /*
   .
   .
   Perform Error Processing
   */
}
```

```
    /*
    .
    .
    continue processing
    .
    */

}
```

See Also

- **ipm_StartMedia( )**

ipm_Open( )

Name: int ipm_Open(*szDevName, *pOpenInfo, usMode)

Inputs:	const char *szDeviceName	• device name pointer
	IPM_OPEN_INFO *pOpenInfo	• set to NULL
	unsigned short usMode	• async or sync mode setting

Returns: device handle if successful
-1 on failure

Includes: srllib.h
ipmlib.h

Category: System Control

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_Open( )** function opens an IP channel or board device and returns a unique device handle to identify the physical device that performs the media transfer. All subsequent references to the opened device must be made using the handle until the device is closed.

The IP Media library allows **ipm_Open( )** to be called either synchronously or asynchronously.

If **ipm_Open( )** is called synchronously and no errors are received, the device handle that is returned is valid and may be used by the application.

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

If **ipm_Open( )** is called asynchronously and IPMEV_ERROR is returned, a device handle is also returned. The application must call **ipm_Close( )** using the handle returned by **ipm_Open( )**.

Parameter	Description
szDeviceName	pointer to device name to open IP Media channel device: **ipmBxCy** where **x** is the unique logical board number and **y** is the media device channel number. Board device: **ipmBx** where **x** is the unique logical board number.
pOpenInfo	set to NULL; reserved for future use
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

Termination Events

IPMEV_OPEN
: Indicates successful completion; that is, an IP channel was opened and the device handle is valid. This event does not return any data.

IPMEV_ERROR
: Indicates that the function failed.

Cautions

- Two different applications (running in separate processes) cannot use the same IP media device (ipmBxCx). In other words, multiple calls to **ipm_Open( )** on the same IP media device are not allowed.
- The **pOpenInfo** pointer is reserved for future use and must be set to NULL.
- If this function is called asynchronously and IPMEV_ERROR is received, the application must call **ipm_Close( )** using the handle returned by **ipm_Open( )**.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EINVAL
: Invalid argument (system-level error)

ENOMEM
: Memory allocation failure (system-level error)

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    char cDevName[10];
    int nDeviceHandle;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    . Create a Thread that waits on srl events, this
    . thread will execute the WorkerThread function
    .
    .
    */

    /*
    Open IP channel ipmB1C1
    */
    sprintf(cDevName,"ipmB1C%d", 1);
        if((nDeviceHandle = ipm_Open(cDevName, NULL, EV_ASYNC)) == -1)
```

```
        {
            printf("ipm_Open failed for device name = %s\n", cDevName);
            /*
            .
            .
            Perform Error Processing
            .
            .
            */
        }
    /*
    .
    .
    . continue Main Processing
    .
    .
    */
}

void CheckEvent()
{
    int nDeviceID = sr_getevtdev();
    int nEventType = sr_getevttype();

    switch(nEventType)
    {
        /*
        .
        .
        . Other events
        .
        .
        */

        /* Expected reply to ipm_Open */
        case IPMEV_OPEN:
            printf("Received IPMEV_OPEN for device = %s\n", ATDV_NAMEP(nDeviceID));
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

■ See Also

- **ipm_Close( )**

ipm_ReceiveDigits( )

Name: int ipm_ReceiveDigits(nDeviceHandle, *pDigitInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_DIGIT_INFO *pDigitInfo	• pointer to digit info structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: I/O

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software

■ Description

The **ipm_ReceiveDigits( )** function enables the IP channel to receive digits from the TDM bus. The receive operation continues until **ipm_Stop( )** is called with the eSTOP_RECEIVE_DIGITS flag set.

Note that digits are *always* received asynchronously, even though this function may be called in either asynchronous or synchronous mode. If this function is called synchronously and returns 0, it does not indicate that the digits have been received; instead, it only indicates that the function was successfully processed by the firmware. The application must enable event reporting and check for IPMEV_DIGITS_RECEIVED events.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pDigitInfo	pointer to data structure that contains digit information. The application must set the direction and type of digits before calling the function. On return, the function sets the unNumberOfDigits field to indicate how many IPMEV_DIGITS_RECEIVED events the application must process. See IPM_DIGIT_INFO for details. **Note**: The cDigits[MAX_IP_DIGITS] field in the data structure pointed to by **pDigitInfo** is not used by the **ipm_ReceiveDigits( )** function in either SYNC or ASYNC mode. The incoming digit is delivered asynchronously in the IPM_DIGIT_INFO data structure associated with an IPMEV_DIGITS_RECEIVED event.
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_RECEIVE_DIGITS
Indicates function was successfully processed but does **not** indicate that digits were received. This event does not return data.

IPMEV_ERROR
Indicates that the function failed.

Note: IPMEV_DIGITS_RECEIVED is an unsolicited event that may be reported after the **ipm_ReceiveDigits( )** function is called either synchronously or asynchronously. An event is reported for each digit that was received. The event data indicates the digit origin via the eIPM_DIGIT_DIRECTION enumeration.

■ Cautions

- The only supported value for IPM_DIGIT_INFO.eIPM_DIGIT_DIRECTION is to receive digits from the TDM bus.
- The **ipm_ReceiveDigits( )** function returns valid data only if the digits are being transmitted in out-of-band mode. For more information on setting DTMF mode, see the *Dialogic® IP Media Library API Programming Guide*.
- Digits are only received if there is an active RTP session; if two IPM devices are directly routed together, you must establish an RTP session before digits can be sent and received.

■ Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
Invalid parameter

EIPM_INTERNAL
Internal error

EIPM_INV_STATE
Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
System error

■ Example

```
#include <ipmlib.h>
#include <srllib.h>
#include <stdio.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    int nDeviceHandle;
    IPM_DIGIT_INFO myDigitInfo;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);
```

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

    /*
     Enable an IP device handle, nDeviceHandle, to receive a specified set of digits.
     ASSUMPTION:  A valid nDeviceHandle was obtained from prior call to ipm_Open().
    */

    myDigitInfo.eDigitType = DIGIT_ALPHA_NUMERIC;
    myDigitInfo.eDigitDirection = DIGIT_TDM;

    if(ipm_ReceiveDigits(nDeviceHandle, &myDigitInfo, EV_ASYNC) == -1)
    {
        printf("ipm_ReceiveDigits failed for device name = %s with error = %d\n",
              ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }

    /*
    .
    .
     Continue processing
    .
    .
    */
}

void CheckEvent()
{

    IPM_DIGIT_INFO *pDigitInfo;
    int nDeviceID  = sr_getevtdev();
    int nEventType = sr_getevttype();
    void* pVoid    = sr_getevtdatap();

    switch(nEventType)
    {
        /*
        .
        .
        . Other events
        .
        .
        */

        //Successful reply to ipm_ReceiveDigits()
        case IPMEV_RECEIVE_DIGITS:
            printf("Received IPMEV_RECEIVE_DIGITS for device = %s\n",
                ATDV_NAMEP(nDeviceID));
            break;
```

```
            //Unsolicited event, retrieve digits
            case IPMEV_DIGITS_RECEIVED:
                printf("Received IPM_DIGITS_RECEIVED for device = %s\n",
                    ATDV_NAMEP(nDeviceID));
                pDigitInfo = (IPM_DIGIT_INFO*)pVoid;
                printf("Number of digits = %d, digit=%s on device %s\n",
                    pDigitInfo->unNumberOfDigits, pDigitInfo->cDigits,
                    ATDV_NAMEP(nDeviceID));
                break;

            default:
                printf("Received unknown event = %d for device = %s\n",
                    nEventType, ATDV_NAMEP(nDeviceID));
                break;
        }
    }
```

■ See Also

- **ipm_SendDigits( )**

ipm_ResetQoSAlarmStatus( )

Name: int ipm_ResetQoSAlarmStatus(nDeviceHandle, *pQoSAlarmInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_QOS_ALARM_STATUS *pQoSAlarmInfo	• pointer to QoS alarm structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: QoS

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software

■ Description

The **ipm_ResetQoSAlarmStatus( )** function resets to the OFF state one or more Quality of Service (QoS) alarms that report the status of a media channel.This function does not apply to board-level alarms.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pQoSAlarmInfo	pointer to IPM_QOS_ALARM_STATUS structure which contains one or more IPM_QOS_ALARM_DATA structures
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_RESET_QOS_ALARM_STATUS
Indicates successful completion; that is, specified QoS alarm(s) has been reset to OFF. This event does not return data.

IPMEV_ERROR
Indicates that the function failed.

■ Cautions

None

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    int nDeviceHandle;
    IPM_QOS_ALARM_STATUS myAlarmStatus;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    Reset the QOSTYPE_JITTER alarm for IP device handle, nDeviceHandle.
    NOTE: nDeviceHandle was obtained from prior call to ipm_Open()
    */
    myAlarmStatus.unAlarmCount = 1;
    myAlarmStatus.QoSData[0].eQoSType = QOSTYPE_JITTER;
    if(ipm_ResetQoSAlarmStatus(nDeviceHandle, &myAlarmStatus, EV_ASYNC) == -1)
    {
        printf("ipm_ResetQoSAlarmStatus failed for device name = %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
         Perform Error Processing
        .
        .
        */
    }
```

```
    /*
    .
    .
    Continue Processing
    .
    .
    */
}

void CheckEvent()
{
    int nEventType = sr_getevttype();
    int nDeviceID = sr_getevtdev();

    switch(nEventType)
    {
        /*
        .
        .
        Other events
        .
        .
        */

        /* Expected reply to ipm_ResetQoSAlarmStatus */
        case IPMEV_RESET_QOS_ALARM_STATUS:
            printf("Received IPMEV_RESET_QOS_ALARM_STATUS for device = %s\n",
                ATDV_NAMEP(nDeviceID));
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

■ See Also

- **ipm_GetQoSAlarmStatus( )**

ipm_SecurityGenMasterKeys( )

Name: int ipm_SecurityGenMasterKeys(nDeviceHandle, *pKeys, usNumKeys, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_SECURITY_KEY *pKeys	• pointer to security key structure
unsigned short usNumKeys	• number of keys to be generated
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: Media Session

Mode: synchronous

Platform: Dialogic® HMP Software

■ Description

The **ipm_SecurityGenMasterKeys( )** function generates master and salt keys. The security keys can be generated either in binary format or Base64-encoded format. This function can be called any time after opening an IP Media device. The function is supported in synchronous mode only.

Parameter	Description
nDeviceHandle	handle of the IP Media device
***pKeys**	A pointer to an array of IPM_SECURITY_KEY structures. Applications need to allocate the memory for each IPM_SECURITY_KEY structure and fill in the version of the structures, type of the key that needs to be generated, pointers to the corresponding key structure, master key length, and master salt key length. This function does not modify any other fields of the structure except the keys generated.
usNumKeys	indicates the number of keys that need to be generated
usMode	operation mode, which must be set to EV_SYNC for synchronous execution

■ Termination Events

None.

■ Cautions

The application is expected to perform all memory allocation for the requested keys, that includes the array of IPM_SECURITY_KEY structures as well as for the individual keys pointed to by the pvMasterKey field in each IPM_SECURITY_KEY structure.

Errors

The function returns an error if:

1. Any of the structure version fields are invalid.

2. The pKeys field in the IPM_SECURITY_INFO structure or the pvMasterKey field in the IPM_SECURITY_KEY structure is NULL.

3. The pcMasterKey or pcMasterSaltKey fields in the IPM_SECURITY_BINARY_KEY structure or the pcMasterBase64Key field in the IPM_SECURITY_BASE64_KEY structure is NULL.

4. The key lengths are not valid.

5. The device handle is not valid.

6. The mode is not EV_SYNC.

Example

```
/*the following sample code uses SRTP */

#include <stdio.h>
#include <string>
#include <srllib.h>
#include <ipmlib.h>
typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    /*
    .
    .
    Main Processing
    .
    */

    /*
    Set the keys for the IP device handle, nDeviceHandle.
    ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
    */

    char Mkey1[30], bs64Mkey1[40];
    char Msalt1[30];

    IPM_SRTP_PARMS srtp_parms;
    IPM_SECURITY_KEY MasterKeys[2];
    IPM_SECURITY_BASE64_KEY Masterbs64Key;
    IPM_SECURITY_BINARY_KEY MasterbinKey;

    INIT_IPM_SRTP_PARMS(&srtp_parms);
    INIT_IPM_SECURITY_BINARY_KEY(&MasterbinKey);

    MasterbinKey.pcMasterKey = Mkey1;
    MasterbinKey.pcMasterSaltKey = Msalt1;

    INIT_IPM_SECURITY_BASE64_KEY(&Masterbs64Key);
    Masterbs64Key.pcMasterBase64Key = bs64Mkey1;

    INIT_IPM_SECURITY_KEY(&MasterKeys[0]);
```

```
MasterKeys[0].eKeyType = IPM_SECURITY_KEYTYPE_BINARY;
MasterKeys[0].pvMasterKey = &MasterbinKey;

INIT_IPM_SECURITY_KEY(&MasterKeys[1]);
MasterKeys[1].eKeyType = IPM_SECURITY_KEYTYPE_BASE64;
MasterKeys[1].pvMasterKey = &Masterbs64Key;


/* Generate the master Key and Master Salt Key for the device */

if ((ipm_SecurityGenMasterKeys(nDeviceHandle, MasterKeys, 1, EV_SYNC) == -1))
{
    printf("ipm_SecurityGenMasterKeys() failed for device name = %s with error = %d\n",
    ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
    /*
    .
    .
    Perform Error Processing
    */
}


/* Masterbs64Key can be filled from SDP */



/*
Set the media properties for a remote party using IP device handle, nDeviceHandle.
ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
*/

IPM_MEDIA_INFO MediaInfo;
MediaInfo.unCount = 6;


MediaInfo.MediaData[0].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTP_INFO;
MediaInfo.MediaData[0].mediaInfo.PortInfo.unPortId = 2328;
strcpy(MediaInfo.MediaData[0].mediaInfo.PortInfo.cIPAddress, "111.21.0.9\n");

MediaInfo.MediaData[1].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTCP_INFO;
MediaInfo.MediaData[1].mediaInfo.PortInfo.unPortId = 2329;
strcpy(MediaInfo.MediaData[1].mediaInfo.PortInfo.cIPAddress, "111.41.0.9\n");

MediaInfo.MediaData[2].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.unCoderPayloadType = 0;
MediaInfo.MediaData[2].mediaInfo.CoderInfo.unRedPayloadType = 0;

MediaInfo.MediaData[3].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.eFrameSize = (eIPM_CODER_FRAMESIZE) 30;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.unFramesPerPkt = 1;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.unCoderPayloadType = 0;
MediaInfo.MediaData[3].mediaInfo.CoderInfo.unRedPayloadType = 0;


MediaInfo.MediaData[4].eMediaType = MEDIATYPE_AUDIO_REMOTE_SECURITY_INFO;
MediaInfo.MediaData[4].mediaInfo.SecurityInfo.unVersion = IPM_SECURITY_INFO_VERSION;
MediaInfo.MediaData[4].mediaInfo. SecurityInfo.unNumKeys = 1;
MediaInfo.MediaData[4].mediaInfo. SecurityInfo.pParms = &srtp_parms;
MediaInfo.MediaData[4].mediaInfo. SecurityInfo.pKeys = &MasterKeys[0];
MediaInfo.MediaData[4].mediaInfo. SecurityInfo.eInfoMode = IPM_SECURITY_INFO_MODE_IMMEDIATE;
```

```
    MediaInfo.MediaData[5].eMediaType = MEDIATYPE_AUDIO_LOCAL_SECURITY_INFO;
    MediaInfo.MediaData[5].mediaInfo. SecurityInfo.unVersion = IPM_SECURITY_INFO_VERSION;
    MediaInfo.MediaData[5].mediaInfo. SecurityInfo.unNumKeys = 1;
    MediaInfo.MediaData[5].mediaInfo. SecurityInfo.pParms = &srtp_parms;
    MediaInfo.MediaData[5].mediaInfo. SecurityInfo.pKeys = &MasterKeys[1];
    MediaInfo.MediaData[5].mediaInfo. SecurityInfo.eInfoMode = IPM_SECURITY_INFO_MODE_IMMEDIATE;


    if (ipm_StartMedia(nDeviceHandle, &MediaInfo, DATA_IP_TDM_BIDIRECTIONAL,EV_SYNC) == -1)
    {
        printf("ipm_StartMediaInfo failed for device name = %s with error = %d\n",
        ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        */
    }
    /*
    .
    . Continue processing
    .
    */


}
```

■ See Also

- **ipm_StartMedia( )**
- **ipm_ModifyMedia( )**

ipm_SendDigits( )

Name: int ipm_SendDigits(nDeviceHandle, *pDigitInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_DIGIT_INFO *pDigitInfo	• pointer to digit info structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: I/O

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software

Description

The **ipm_SendDigits**( ) function generates the supplied digits to the TDM bus.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pDigitInfo	pointer to structure that contains digit type, direction, and digits; see IPM_DIGIT_INFO for details. Note that the application must fill in the digit type, direction, number of digits, and the actual digits to be sent. The maximum number of digits is 16.
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

Termination Events

IPMEV_SEND_DIGITS
Indicates successful completion; that is, the supplied digits were sent. This event does not return data.

IPMEV_ERROR
Indicates that the function failed.

Cautions

- If this function is called synchronously and returns 0, it does not indicate that the digits have been sent, but only that the function was successfully processed by the firmware. The application must enable event reporting and check for the IPMEV_SEND_DIGITS event.

- The only supported value for IPM_DIGIT_INFO.eIPM_DIGIT_DIRECTION is to send digits toward the TDM bus.
- Digits are only exchanged if there is an active RTP session; if two IPM devices are directly routed together, you must establish an RTP session before digits can be sent and received.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

Example

```
#include <stdio.h>
#include <string.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    int nDeviceHandle;
    IPM_DIGIT_INFO myDigitInfo;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    Generate a set of digits using IP device handle, nDeviceHandle.
    ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
    */
    myDigitInfo.eDigitType = DIGIT_ALPHA_NUMERIC;
    myDigitInfo.eDigitDirection = DIGIT_TDM;
    strcpy(myDigitInfo.cDigits,"1234567890123456");
    myDigitInfo.unNumberOfDigits = 16;
    if(ipm_SendDigits(nDeviceHandle, &myDigitInfo, EV_ASYNC) == -1)
    {
        printf("ipm_SendDigits failed for device name = %s with error = %d\n",
```

```
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }

    /*
    .
    .
    . Continue Main processing
    .
    .
    */
}

void CheckEvent()
{
    int nDeviceID = sr_getevtdev();
    int nEventType = sr_getevttype();
    void* pVoid = sr_getevtdatap();

    switch(nEventType)
    {
        /*
        .
        .
        . Other events
        .
        .
        */

        //Successful reply to ipm_SendDigits()
        case IPMEV_SEND_DIGITS:
            printf("Received IPMEV_SEND_DIGITS for device = %s\n", ATDV_NAMEP(nDeviceID));
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

See Also

- **ipm_ReceiveDigits( )**

ipm_SetParm( )

Name: int ipm_SetParm(nDeviceHandle, *pParmInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_PARM_INFO *pParmInfo	• pointer to parameter info structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: System Control

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_SetParm( )** function sets values for the specified parameter.

Parameter	Description
nDeviceHandle	handle of the IP media device
pParmInfo	pointer to structure that contains IP channel parameter values See the IPM_PARM_INFO data structure page for details.
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_SET_PARM
Indicates successful completion; that is, the supplied IP channel parameter was modified.

IPMEV_ERROR
Indicates that the function failed.

■ Cautions

None

■ Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_FWERROR
: Firmware error

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

void CheckEvent();
typedef long int(*HDLR)(unsigned long);

void main()
{
    int nDeviceHandle;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    ASSUMPTION: A valid nDeviceHandle was obtained from prior
    call to ipm_Open().
    */
    IPM_PARM_INFO ParmInfo;
    unsigned long ulParmValue = ECHO_TAIL_16;
    ParmInfo.eParm = PARMCH_ECHOTAIL;
    ParmInfo.pvParmValue = &ulParmValue;
    if(ipm_SetParm(nDeviceHandle, &ParmInfo, EV_ASYNC)==-1)
    {
        printf("ipm_SetParm failed for device name %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }

    /*
    .
    .
    continue
    .
    .
    */
}
```

```
void CheckEvent()
{
    int nEventType = sr_getevttype();
    int nDeviceID = sr_getevtdev();
    void* pVoid = sr_getevtdatap();

    switch(nEventType)
    {
        /*
        .
        .
        Other events
        .
        .
        */

        /* Expected reply to ipm_GetQoSAlarmStatus */
        case IPMEV_SET_PARM:
            printf("Received IPMEV_SETPARM for device = %s\n",
                ATDV_NAMEP(nDeviceID));
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

See Also

- **ipm_GetParm( )**

ipm_SetQoSThreshold( )

Name: int ipm_SetQoSThreshold(nDeviceHandle, *pQoSThresholdInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media channel device handle
IPM_QOS_THRESHOLD_INFO *pQoSThresholdInfo	• pointer to QoS alarm threshold structure
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: QoS

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_SetQoSThreshold( )** function changes alarm threshold settings for Quality of Service (QoS) alarms that report the status of a media channel.

This function can be called at any time, including when a session is in progress.

Parameter	Description
nDeviceHandle	handle of the IP Media channel device
pQoSThresholdInfo	pointer to IPM_QOS_THRESHOLD_INFO structure which contains one or more IPM_QOS_THRESHOLD_DATA structures with the threshold settings to be set. Note that when an application needs to specify any given field in an IPM_QOS_THRESHOLD_DATA structure, it must populate *all* fields in the structure even if those fields are to remain at their default values.
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_SET_QOS_THRESHOLD_INFO
: Indicates successful completion; that is, alarm QoS threshold levels were modified. Use SRL functions to retrieve IPM_QOS_THRESHOLD_INFO structure fields.

IPMEV_ERROR
: Indicates that the function failed.

Cautions

If an application exits without calling **ipm_UnListen( )** to clean up voice device routings, the **ipm_SetQoSThreshold( )** function may fail if it is called after the application is restarted but before a new routing of the IPM device to a voice device is established.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

void CheckEvent();
typedef long int(*HDLR)(unsigned long);

void main()
{
    int nDeviceHandle;
    IPM_QOS_THRESHOLD_INFO mySetQosThresholdInfo;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    Change two alarm threshold settings for IP device handle, nDeviceHandle.
    ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
    */
    mySetQosThresholdInfo.unCount = 2;
    mySetQosThresholdInfo.QoSThresholdData[0].eQoSType = QOSTYPE_LOSTPACKETS;
    mySetQosThresholdInfo.QoSThresholdData[0].unTimeInterval = 100;
    mySetQosThresholdInfo.QoSThresholdData[0].unDebounceOn = 100;
    mySetQosThresholdInfo.QoSThresholdData[0].unDebounceOff = 100;
    mySetQosThresholdInfo.QoSThresholdData[0].unFaultThreshold = 20;
    mySetQosThresholdInfo.QoSThresholdData[0].unPercentSuccessThreshold = 60;
```

```
    mySetQosThresholdInfo.QoSThresholdData[0].unPercentFailThreshold = 40;
    mySetQosThresholdInfo.QoSThresholdData[1].eQoSType = QOSTYPE_JITTER;
    mySetQosThresholdInfo.QoSThresholdData[1].unTimeInterval = 100;
    mySetQosThresholdInfo.QoSThresholdData[1].unDebounceOn = 200;
    mySetQosThresholdInfo.QoSThresholdData[1].unDebounceOff = 600;
    mySetQosThresholdInfo.QoSThresholdData[1].unFaultThreshold = 60;
    mySetQosThresholdInfo.QoSThresholdData[1].unPercentSuccessThreshold = 60;
    mySetQosThresholdInfo.QoSThresholdData[1].unPercentFailThreshold = 40;

    if(ipm_SetQoSThreshold(nDeviceHandle, &mySetQosThresholdInfo, EV_ASYNC) == -1)
    {
        printf("ipm_SetQoSThreshold failed for device name = %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }

    /*
    .
    .
    . continue
    .
    .
    */
}

void CheckEvent()
{
    //Get event type and associated data
    int nEventType = sr_getevttype();
    int nDeviceID = sr_getevtdev();

    switch(nEventType)
    {
        /*
        .
        .
        . Other events
        .
        .
        */

        /* Expected reply to ipm_SetQoSThreshold */
        case IPMEV_SET_QOS_THRESHOLD_INFO:
            printf("Received IPMEV_SET_QOS_THRESHOLD_INFO for device = %s\n",
                ATDV_NAMEP(nDeviceID));
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

■ See Also

- **ipm_GetQoSThreshold( )**

ipm_StartMedia( )

Name: int ipm_StartMedia(nDeviceHandle, *pMediaInfo, eDirection, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
IPM_MEDIA_INFO *pMediaInfo	• pointer to media information structure
eIPM_DATA_DIRECTION eDirection	• data flow direction
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: Media Session

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_StartMedia( )** function sets media properties and starts the session. This function allows the application to set the remote and local connectivity selections. **ipm_StartMedia( )** also starts RTP streaming. The remote RTP/ RTCP port information and coder information is provided in the IPM_MEDIA_INFO structure.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pMediaInfo	media information data structure; see IPM_MEDIA_INFO for details Applications can define the following: • local transmit coder and remote transmit coder • remote RTP/RTCP port • remote IP address • remote T.38 port ***Note:*** The application cannot define the local IP address. The IP address is defined by operating system functions.

Parameter	Description
eDirection	media operation enumeration The eIPM_DATA_DIRECTION data type is an enumeration which defines the following values: • DATA_IP_RECEIVEONLY – receive data from the IP network but do not send data • DATA_IP_SENDONLY – send data to the IP network but do not receive data • DATA_IP_TDM_BIDIRECTIONAL – full duplex data path (streaming media) between IP network and TDM
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

Termination Events

IPMEV_STARTMEDIA
: Indicates successful completion; that is, media information was set and the session has been started.

IPMEV_ERROR
: Indicates that the function failed.

Cautions

- Do not set the IP address to 0.0.0.0, because this may lead to a hung port.
- For the Dialogic® Multimedia Platform for AdvancedTCA, this function returns an invalid state error if a coder is not reserved prior to calling **ipm_GetLocalMediaInfo( )**. All audio and/or video coders must be reserved using the **dev_ReserveResourceEx( )** Dialogic® Device Management Library function. However, this restriction does not apply when using a fax (T.38) media type.

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_BUSY
: Channel is busy

EIPM_INTERNAL
: Internal error

EIPM_INV_MODE
: Invalid mode

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
System error

The **ipm_StartMedia**( ) function returns security errors in the following circumstances:

- The total number of keys has exceeded the given limit (IPM_SECURITY_MAX_TOTAL_NUM_KEYS, which is set to 20).
- The number of keys for any given media type exceeds the given limit (IPM_SECURITY_MAX_NUM_KEYS, which is set to 10).
- The size of a generated key does not match the key size (master key or master salt) in the specified crypto suite (see Table 7, “Crypto Suite Parameter Values”, on page 287).
- The Secure RTP feature is not available.
- All Secure RTP resources are allocated.
- Any of the structure version fields are invalid.

■ Example

```
#include <stdio.h>
#include <string>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
     int nDeviceHandle, nMediaCnt = 0;
     // Register event handler function with srl
     IPM_MEDIA_INFO MediaInfo;

     sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);
     /*
     .
     .
     Main Processing
     .
     .
     */
     /*
     Set the media properties for a remote party using IP device handle, nDeviceHandle.
     ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
     */

     /* clear the RemoteMediaInfo structures */
     memset(&MediaInfo, 0, sizeof(IPM_MEDIA_INFO));

     /* remote audio */
     MediaInfo.MediaData[nMediaCnt].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
     MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
     MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.eFrameSize =
              (eIPM_CODER_FRAMESIZE)m_nAudioFrameSize;
     MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.unFramesPerPkt = 1;
     MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
     MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.unCoderPayloadType = 0;
     MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.unRedPayloadType = 0;

     /* local audio */
     ++nMediaCnt;
     MediaInfo.MediaData[nMediaCnt].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
     MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.eCoderType = CODER_TYPE_G711ULAW64K;
```

```
    MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.eFrameSize =
             (eIPM_CODER_FRAMESIZE)m_nAudioFrameSize;
    MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.unFramesPerPkt = 1;
    MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.eVadEnable = CODER_VAD_DISABLE;
    MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.unCoderPayloadType = 0;
    MediaInfo.MediaData[nMediaCnt].mediaInfo.CoderInfo.unRedPayloadType = 0;

    /* remote video */
    ++nMediaCnt;
    MediaInfo.MediaData[nMediaCnt].eMediaType = MEDIATYPE_VIDEO_REMOTE_CODER_INFO;
    INIT_IPM_VIDEO_CODER_INFO(&MediaInfo.MediaData[nMediaCnt].mediaInfo.VideoCoderInfo);
    MediaInfo.MediaData[nMediaCnt].mediaInfo.VideoCoderInfo.eCoderType = VIDEO_CODING_H263;
    MediaInfo.MediaData[nMediaCnt].mediaInfo.VideoCoderInfo.unCoderPayloadType = 100;


    /* local video coder */
    ++nMediaCnt;
    MediaInfo.MediaData[nMediaCnt].eMediaType = MEDIATYPE_VIDEO_LOCAL_CODER_INFO;
    INIT_IPM_VIDEO_CODER_INFO(&MediaInfo.MediaData[nMediaCnt].mediaInfo.VideoCoderInfo);
    MediaInfo.MediaData[nMediaCnt].mediaInfo.VideoCoderInfo.eCoderType = VIDEO_CODING_H263;
    MediaInfo.MediaData[nMediaCnt].mediaInfo.VideoCoderInfo.unCoderPayloadType = 100;


    /* remote audio port */
    ++nMediaCnt;
    MediaInfo.MediaData[nMediaCnt].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTP_INFO;
    strcpy(MediaInfo.MediaData[nMediaCnt].mediaInfo.PortInfo.cIPAddress,"192.168.0.9");
    MediaInfo.MediaData[nMediaCnt].mediaInfo.PortInfo.unPortId = 2328;

    ++nMediaCnt;
    MediaInfo.MediaData[nMediaCnt].eMediaType = MEDIATYPE_AUDIO_REMOTE_RTCP_INFO;
    strcpy(MediaInfo.MediaData[nMediaCnt].mediaInfo.PortInfo.cIPAddress,"192.168.0.9");
    MediaInfo.MediaData[nMediaCnt].mediaInfo.PortInfo.unPortId = 2329;

    /* remote video port */
    ++nMediaCnt;
    MediaInfo.MediaData[nMediaCnt].eMediaType = MEDIATYPE_VIDEO_REMOTE_RTP_INFO;
    strcpy(MediaInfo.MediaData[nMediaCnt].mediaInfo.PortInfo.cIPAddress,"192.168.0.9");
    MediaInfo.MediaData[nMediaCnt].mediaInfo.PortInfo.unPortId = 2340;

    ++nMediaCnt;
    MediaInfo.MediaData[nMediaCnt].eMediaType = MEDIATYPE_VIDEO_REMOTE_RTCP_INFO;
    strcpy(MediaInfo.MediaData[nMediaCnt].mediaInfo.PortInfo.cIPAddress,"192.168.0.9");
    MediaInfo.MediaData[nMediaCnt].mediaInfo.PortInfo.unPortId = 2341;

    MediaInfo.unCount = nMediaCnt + 1;

    if(ipm_StartMedia(nDeviceHandle, &MediaInfo, DATA_IP_TDM_BIDIRECTIONAL, EV_ASYNC) == -1)
    {
        printf("ipm_StartMediaInfo failed for device name = %s with error = %d\n",
                 ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        Perform Error Processing
        .
        .
        */
    }

    /*
    .
    .
    Continue processing
    .
    .
    */
}
```

```
void CheckEvent()
{
     int nDeviceID = sr_getevtdev();
     int nEventType = sr_getevttype();
     switch(nEventType)
     {
          /*
          .
          .
          Other events
          .
          .
          */
          /* Expected reply to ipm_StartMedia */
          case IPMEV_STARTMEDIA:
               printf("Received IPMEV_STARTMEDIA for device = %s\n", ATDV_NAMEP(nDeviceID));
               break;

          default:
               printf("Received unknown event = %d for device = %s\n",
                         nEventType, ATDV_NAMEP(nDeviceID));
               break;
     }
}
```

See Also

- **ipm_ModifyMedia( )**
- **ipm_Stop( )**

ipm_Stop( )

Name: int ipm_Stop(nDeviceHandle, eOperation, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
eIPM_STOP_OPERATION eOperation	• operation to be stopped
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: Media Session

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_Stop( )** function stops operations on the specified IP channel.

To run this function asynchronously, set **mode** to EV_ASYNC. The function returns 0 if successful and the application must wait for the IPMEV_STOPPED event.

Parameter	Description
nDeviceHandle	handle of the IP Media device
eOperation	the type of operation(s) to stop; only one value can be set at a time The eIPM_STOP_OPERATION data type is an enumeration that defines the following values: • STOP_RECEIVE_DIGITS – stop receiving digits • STOP_MEDIA – operation of media streaming session. This enumeration disconnects the session. The application must call **ipm_StartMedia( )** to start a new session. • STOP_ALL – stop all operations
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_STOPPED
Indicates that activity of the type specified in **eOperation** has terminated on this channel. This event does not return data.

IPMEV_ERROR
Indicates that the function failed.

Cautions

None

Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_FWERROR
: Firmware error

Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    int nDeviceHandle;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    Application needs to stop a current session on IP device handle, nDeviceHandle
    ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open()
    and a session has been started by calling ipm_StartMedia() some time earlier.
    */
    if(ipm_Stop(nDeviceHandle, STOP_ALL, EV_ASYNC) == -1)
    {
        printf("ipm_Stop failed for device name = %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }
```

```
    /*
    .
    .
    Continue Processing
    .
    .
    */
}

void CheckEvent()
{
    int nEventType = sr_getevttype();
    int nDeviceID = sr_getevtdev();

    switch(nEventType)
    {
        /*
        .
        .
        List of expected events
        .
        .
        */

        /* Expected reply from ipm_Stop() */
        case IPMEV_STOPPED:
            printf("Received IPMEV_STOPPED for device = %s\n", ATDV_NAMEP(nDeviceID));
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

■ See Also

- **ipm_UnListen( )**

ipm_UnListen( )

Name: int ipm_UnListen(nDeviceHandle, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle
unsigned short usMode	• async or sync mode setting

Returns: 0 on success
-1 on failure

Includes: srllib.h
ipmlib.h

Category: System Control

Mode: asynchronous or synchronous

Platform: Dialogic® HMP Software, Dialogic® Multimedia Platform for AdvancedTCA, Dialogic® Multimedia Kit for PCIe

■ Description

The **ipm_UnListen( )** function stops listening to the TDM time slot specified in a previous call to **ipm_Listen( )**.

If **ipm_Listen( )** is called to connect to a different TDM time slot, the firmware automatically breaks an existing connection and reconnects it to the new time slot. In this case, the application does not need to call the **ipm_UnListen( )** function.

Parameter	Description
nDeviceHandle	handle of the IP Media device
usMode	operation mode Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution.

■ Termination Events

IPMEV_UNLISTEN
: Indicates successful completion; that is, the IP channel was disconnected from the specified TDM time slot. This event does not return data.

IPMEV_ERROR
: Indicates that the function failed.

■ Cautions

- The IP Media library allows **ipm_Listen( )** and **ipm_UnListen( )** to be called either synchronously or asynchronously. Other Dialogic libraries may not support asynchronous execution of the similar xx_Listen and xx_UnListen functions.

■ Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EIPM_BADPARM
: Invalid parameter

EIPM_FWERROR
: Firmware error

EIPM_INTERNAL
: Internal error

EIPM_INV_STATE
: Invalid state. Initial command did not complete before another function call was made.

EIPM_SYSTEM
: System error

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <ipmlib.h>

typedef long int(*HDLR)(unsigned long);
void CheckEvent();

void main()
{
    int nDeviceHandle;
    // Register event handler function with srl
    sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);

    /*
    .
    .
    Main Processing
    .
    .
    */

    /*
    Stop an IP device handle, nDeviceHandle, from listening to a time slot.
    ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
    */
    if(ipm_UnListen(nDeviceHandle, EV_ASYNC) == -1)
    {
        printf("ipm_UnListen failed for device name = %s with error = %d\n",
            ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
        /*
        .
        .
        Perform Error Processing
        .
        .
        */
    }
```

```
    /*
    .
    .
    Continue processing
    .
    .
    */
}

void CheckEvent()
{
    int nEventType = sr_getevttype();
    int nDeviceID = sr_getevtdev();

    switch(nEventType)
    {
        /*
        .
        .
        Other events
        .
        .
        */

        /*Expected reply from ipm_UnListen*/
        case IPMEV_UNLISTEN:
            printf("Received IPMEV_UNLISTEN for device = %s\n", ATDV_NAMEP(nDeviceID));
            break;

        default:
            printf("Received unknown event = %d for device = %s\n",
                nEventType, ATDV_NAMEP(nDeviceID));
            break;
    }
}
```

■ See Also

- **ipm_Listen( )**
- **ipm_Stop( )**

Events 25

This chapter describes the events that are returned by the Dialogic® IP Media Library API functions. The function descriptions in Chapter 24, "Function Information" lists the function's termination events for asynchronous operations.

There are three types of events returned by the Dialogic® IP Media Library API functions:

- events returned after the termination of a function call, called termination events
- unsolicited events triggered by external events
- notification events requested (solicited) by the application

Applications can enable or disable certain notification events for Quality of Service (QoS) information. The notification events supported by the Dialogic® IP Media Library API are enabled and disabled via the function calls **ipm_EnableEvents( )** and **ipm_DisableEvents( )**, respectively.

The following events, listed in alphabetical order, may be returned by the Dialogic® IP Media Library API software. Use **sr_waitevt( )**, **sr_enbhdlr( )** or other SRL functions to collect an event code, depending on the programming model in use. For more information, see the *Dialogic® Standard Runtime Library API Library Reference*.

IPMEV_DIGITS_RECEIVED
: Unsolicited event for **ipm_ReceiveDigits( )** in either synchronous or asynchronous mode. One event is returned for each digit that is received. Event contains digit data in IPM_DIGIT_INFO data structure.

IPMEV_ERROR
: Generic unsuccessful termination event. This event may be generated on any handle when there is an error. No data is returned in the event.

IPMEV_EVENT_DISABLED
: Successful termination event for **ipm_DisableEvents( )**. Indicates that IP notification events specified in function call have been disabled. No data is returned in the event.

IPMEV_EVENT_ENABLED
: Successful termination event for **ipm_EnableEvents( )**. Indicates that IP notification events specified in the function call have been enabled. No data is returned in the event.

IPMEV_FAXTONE
: Unsolicited event enabled via **ipm_EnableEvents( )**. Event is returned when fax tone is detected on TDM. Contains fax tone information in an IPM_FAX_SIGNAL data structure.

IPMEV_GENERATEIFRAME
: Successful termination event for **ipm_GenerateIFrame( )**. Contains I-frame information in the IPM_IFRAME_INFO structure.

IPMEV_GENERATEIFRAME_FAIL
: Termination event for **ipm_GenerateIFrame( )**. Indicates that the application is incapable of generating an I-frame.

IPMEV_GET_LOCAL_MEDIA_INFO
: Successful termination event for **ipm_GetLocalMediaInfo( )**. Contains requested local media information in an IPM_MEDIA_INFO structure.

IPMEV_GET_PARM
: Successful termination event for **ipm_GetParm( )**. Contains requested IP channel parameters in an IPM_PARM_INFO structure.

IPMEV_GET_QOS_ALARM_STATUS
: Successful termination event for **ipm_GetQoSAlarmStatus( )**. Contains requested alarm status information in an IPM_QOS_ALARM_STATUS data structure.

IPMEV_GET_QOS_THRESHOLD_INFO
: Successful termination event for **ipm_GetQoSThreshold( )**. Contains requested alarm threshold settings in an IPM_QOS_THRESHOLD_INFO data structure.

IPMEV_GET_SESSION_INFO
: Successful termination event for **ipm_GetSessionInfo( )**. Contains QoS and RTCP statistics in an IPM_SESSION_INFO data structure.

IPMEV_GET_XMITTS_INFO
: Successful termination event for **ipm_GetXmitSlot( )**. Contains requested TDM time slot information in an SC_TSINFO data structure.

IPMEV_INIT_COMPLETE
: Unsolicited event for **ipm_InitResponseSend( )** and **ipm_InitSend( )** reported when the Nb UP session has been successfully negotiated with the remote party. Both parties may now begin exchanging data. See also IPMEV_INIT_SEND and IPMEV_INIT_RESPONSE_SEND.

IPMEV_INIT_FAILED
: Unsolicited event for **ipm_InitResponseSend( )** and **ipm_InitSend( )** reported upon encountering an unexpected error during initialization or a negative response (NACK) from the remote party. Event Data: IPM_INIT_FAILURE structure. See also IPMEV_INIT_SEND and IPMEV_INIT_RESPONSE_SEND.

IPMEV_INIT_RECEIVED
: Unsolicited event telling the application that there is an incoming request for an Nb UP session. The application may respond to this message by calling **ipm_InitResponseSend( )**.

IPMEV_INIT_RESPONSE_SEND
: Successful termination event for **ipm_InitResponseSend( )**. Indicates successful completion of sending the initialization response message. It does not indicate whether the event was received by the remote party. If IPMEV_INIT_RESPONSE_SEND is received, one of the following unsolicited events is subsequently reported: IPMEV_INIT_COMPLETE or IPMEV_INIT_FAILED.

IPMEV_INIT_SEND
: Successful termination event for **ipm_InitSend( )**. Indicates successful completion of sending the initialization message. It does not indicate whether the event was received by remote party or what the response was. If IPMEV_INIT_SEND is received, one of the following unsolicited events is subsequently reported: IPMEV_INIT_COMPLETE or IPMEV_INIT_FAILED.

IPMEV_LISTEN
: Successful termination event for **ipm_Listen( )**. Indicates time slot routing was successfully completed. No data is returned in the event.

IPMEV_MODIFYMEDIA
: Successful termination event for **ipm_ModifyMedia( )**. Indicates change of media characteristics was successfully completed. No data is returned in the event.

IPMEV_MODIFYMEDIA_FAIL
: Unsuccessful termination event for **ipm_ModifyMedia( )**. Indicates that the media session was not changed.

IPMEV_NACK_SENT
: Unsolicited event telling the application that the IPML library was forced to send a NACK message to the other end of the connection. An example is if a request comes in that the library does not understand. This is different from sending a NACK response to an IPM_INIT_RECEIVED event.

IPMEV_NOTIFY_ENDPOINTID
: Unsolicited event enabled via **ipm_EnableEvents( )**. The event payload contains identification information of the RTP/RTCP endpoint.

IPMEV_OPEN
: Successful termination event for **ipm_Open( )**. Indicates IP channel was successfully opened and device handle is valid. No data is returned in the event.

IPMEV_QOS_ALARM
: Unsolicited event enabled via **ipm_EnableEvents( )**. Event is returned when desired QoS alarm triggers. See the code example (specifically the CheckEvent( ) function definition) in Section 17.7, "Example Code for QoS Alarm Handling", on page 92 for information on determining which alarm triggered the event.

IPMEV_RECEIVE_DIGITS
: Successful termination event for **ipm_ReceiveDigits( )**. Indicates channel has been enabled to receive digits. No data is returned in the event.

  Note: IPMEV_DIGITS_RECEIVED is used to indicate when digit transfer has occurred.

IPMEV_RESET_QOS_ALARM_STATUS
: Successful termination event for **ipm_ResetQoSAlarmStatus( )**. Indicates specified QoS alarms have been reset to OFF state. No data is returned in the event.

IPMEV_RTCP_NOTIFY_RECEIVED
: Unsolicited event enabled via **ipm_EnableEvents( )**. Indicates incoming RTCP reporting data.

IPMEV_RTCP_NOTIFY_SENT
: Unsolicited event enabled via **ipm_EnableEvents( )**. Indicates outgoing RTCP reporting data.

IPMEV_SEC_NOTIFY_EXPIRE_KEY_AUDIO
: Unsolicited event enabled via **ipm_EnableEvents( )**. Provides notification that the encryption key for the audio media type is about to expire in the predefined time interval. The time interval can be set using the **ipm_SetParm( )** function and the **PARMCH_NOTIFY_EXPIRE_KEY_AUDIO** parameter.

IPMEV_SEC_NOTIFY_EXPIRE_KEY_VIDEO
: Unsolicited event enabled via **ipm_EnableEvents( )**. Provides notification that the encryption key for the video media type is about to expire in the predefined time interval. The time interval can be set using the **ipm_SetParm( )** function and the **PARMCH_NOTIFY_EXPIRE_KEY_VIDEO** parameter.

IPMEV_SEND_DIGITS
: Successful termination event for **ipm_SendDigits( )**. Indicates supplied digits were sent successfully. No data is returned in the event.

IPMEV_SET_PARM
: Successful termination event for **ipm_SetParm( )**. Indicates IP channel parameters have been modified. No data is returned in the event.

IPMEV_SET_QOS_THRESHOLD_INFO
: Successful termination event for **ipm_SetQoSThreshold( )**. Indicates requested changes to QoS alarm threshold levels have been made. The updated threshold information is returned in an IPM_QOS_THRESHOLD_INFO data structure.

IPMEV_STARTMEDIA
: Successful termination event for **ipm_StartMedia( )**. Indicates media channel information has been set and session has been started. No data is returned in the event.

IPMEV_STOPPED
: Successful termination event for **ipm_Stop( )**. Indicates all on-going activity on the IP channel has terminated. No data is returned in the event.

IPMEV_T38CALLSTATE
: Unsolicited event enabled via **ipm_EnableEvents( )**. Event is returned when T.38 call state changes. Event data is an eIPM_T38CALLSTATE enumeration identifying the new call state.

IPMEV_TELEPHONY_EVENT
: Unsolicited event enabled via **ipm_EnableEvents( )**. Event is generated when RFC 2833 signal is detected on IP. Event contains signal data in an IPM_TELEPHONY_EVENT_INFO data structure.

IPMEV_UNLISTEN
: Successful termination event for **ipm_UnListen( )**. Indicates IP channel was disconnected from TDM time slot. No data is returned in the event.

Data Structures 26

This chapter describes the data structures and fields used by the Dialogic® IP Media library (IPML) functions. These structures are used to control the operation of functions and to return information. The fields are listed in the sequence in which they are defined in the data structure.

CT_DEVINFO

```
typedef struct ct_devinfo {
   unsigned long  ct_prodid;       /* product ID */
   unsigned char  ct_devfamily;    /* device family */
   unsigned char  ct_devmode;      /* device mode */
   unsigned char  ct_nettype;      /* network interface */
   unsigned char  ct_busmode;      /* bus architecture */
   unsigned char  ct_busencoding;  /* bus encoding */
   union {
           unsigned char ct_RFU[7];     /* reserved */
           struct {
                 unsigned char ct_prottype;
           } ct_net_devinfo;
   } ct_ext_devinfo;
} CT_DEVINFO;
```

■ Description

The CT_DEVINFO data structure supplies information about a device. This structure is used by the **ipm_GetCTInfo( )** function. On return from the function, CT_DEVINFO contains the relevant device and device configuration information.

The valid values for each field of the CT_DEVINFO structure are defined in *ctinfo.h*, which is referenced by *ipmlib.h*. The following descriptions indicate only the values that are relevant when using this structure with the Dialogic® IP Media library. Note that this same data structure definition is used in other Dialogic® API libraries where many additional values may be used.

■ Field Descriptions

The fields of the CT_DEVINFO data structure are described as follows:

ct_prodid
: contains a valid product identification number for the device

ct_devfamily
: specifies the device family; possible values are:
  - CT_DFHMPDM3 – HMP (Host Media Processing) device
  - CT_DFHMPATCA – Multimedia Platform for AdvancedTCA device

ct_devmode
: specifies the device mode; possible values are:
  - CT_DMNETWORK – DM3 network device

ct_nettype
: specifies the type of network interface for the device; possible values are:
  - CT_NTIPT – IP connectivity

ct_busmode
: specifies the bus architecture used to communicate with other devices in the system; possible values are:
  - CT_BMSCBUS – TDM bus architecture

ct_busencoding
describes the PCM encoding used on the bus; possible values are:

- CT_BEULAW – mu-law encoding
- CT_BEALAW – A-law encoding

ct.ext_devinfo.ct_RFU
not used

ct_ext_devinfo.ct_net_devinfo.ct_prottype
not used

Example

See the Example section for **ipm_GetCTInfo( )**.

IPM_AUDIO_CODER_INFO

```
typedef struct ipm_audio_coder_info_tag
{
   eIPM_CODER_TYPE         eCoderType;         /* The coder Type                 */
   eIPM_CODER_FRAMESIZE    eFrameSize;         /* Frame size supported           */
   unsigned int            unFramesPerPkt;     /* No. of Frames per packet       */
   eIPM_CODER_VAD          eVadEnable;         /* VAD enable/disable flag        */
   unsigned int            unCoderPayloadType; /* Type of coder payload supported */
   unsigned int            unRedPayloadType;   /* Type of Redundancy Payload     */

} IPM_AUDIO_CODER_INFO, *PIPM_AUDIO_CODER_INFO;
typedef IPM_AUDIO_CODER_INFO  IPM_CODER_INFO;
typedef PIPM_AUDIO_CODER_INFO  PIPM_CODER_INFO;
```

■ Description

This structure contains the coder properties that will be used in an audio IP session. IPM_AUDIO_CODER_INFO is a child of IPM_MEDIA, which is a child of the IPM_MEDIA_INFO structure. IPM_MEDIA_INFO is used by the **ipm_GetLocalMediaInfo( )** and **ipm_StartMedia( )** functions.

Note: The IPM_AUDIO_CODER_INFO data structure was previously defined under the name IPM_CODER_INFO. The *ipmlib.h* header file includes typedefs that ensure backward compatibility for application code that uses the older name for the data structure.

Note: Using the AMR-NB resource in connection with one or more Dialogic products does not grant the right to practice the AMR-NB standard. To seek a license patent agreement to practice the standard, contact the VoiceAge Corporation at www.voiceage.com/licensing/php.

■ Field Descriptions

The fields of the IPM_AUDIO_CODER_INFO data structure are described as follows. Refer to Table 4 for coder-specific guidelines for filling in these fields.

eCoderType
: type of coder to be used for streaming media operations

The following values are defined:

- CODER_TYPE_AMRNB_4_75K – GSM AMR-NB, 4.75 kbps, CMR type 0
- CODER_TYPE_AMRNB_4_75K_NATIVE – GSM AMR-NB, 4.75 kbps, CMR type 0
- CODER_TYPE_AMRNB_5_15K – GSM AMR-NB, 5.15 kbps, CMR type 1
- CODER_TYPE_AMRNB_5_15K_NATIVE – GSM AMR-NB, 5.15 kbps, CMR type 1
- CODER_TYPE_AMRNB_5_9K – GSM AMR-NB, 5.9 kbps, CMR type 2
- CODER_TYPE_AMRNB_5_9K_NATIVE – GSM AMR-NB, 5.9 kbps, CMR type 2
- CODER_TYPE_AMRNB_6_7K – GSM AMR-NB, 6.7 kbps, CMR type 3
- CODER_TYPE_AMRNB_6_7K_NATIVE – GSM AMR-NB, 6.7 kbps, CMR type 3
- CODER_TYPE_AMRNB_7_4K – GSM AMR-NB, 7.4 kbps, CMR type 4
- CODER_TYPE_AMRNB_7_4K_NATIVE – GSM AMR-NB, 7.4 kbps, CMR type 4
- CODER_TYPE_AMRNB_7_95K – GSM AMR-NB, 7.95 kbps, CMR type 5
- CODER_TYPE_AMRNB_7_95K_NATIVE – GSM AMR-NB, 7.95 kbps, CMR type 5
- CODER_TYPE_AMRNB_10_2K – GSM AMR-NB, 10.2 kbps, CMR type 6
- CODER_TYPE_AMRNB_10_2K_NATIVE – GSM AMR-NB, 10.2 kbps, CMR type 6
- CODER_TYPE_AMRNB_12_2K – GSM AMR-NB, 12.2 kbps, CMR type 7

- CODER_TYPE_AMRNB_12_2K_NATIVE – GSM AMR-NB, 12.2 kbps, CMR type 7
- CODER_TYPE_AMRNB_NONE – Don't care, CMR type 15. Indicates that the software does not care what bit rate it receives; the eCoderType field sets the CMR value in the transmitted packet. This value is not supported for the MEDIATYPE_AUDIO_REMOTE_CODER_INFO media type. This value is only supported for the MEDIATYPE_AUDIO_LOCAL_CODER_INFO media type.
- CODER_TYPE_EVRC – EVRC
- CODER_TYPE_G711ALAW64K – G.711, A-law, 64 kbps
- CODER_TYPE_G711ALAW64K_NATIVE – G.711, A-law, 64 kbps
- CODER_TYPE_G711ULAW64K – G.711, mu-law, 64 kbps
- CODER_TYPE_G711ULAW64K_NATIVE – G.711, mu-law, 64 kbps
- CODER_TYPE_G7231_5_3K – G.723.1, 5.3 kbps
- CODER_TYPE_G7231_5_3K_NATIVE – G.723.1, 5.3 kbps
- CODER_TYPE_G7231_6_3K – G.723.1, 6.3 kbps
- CODER_TYPE_G7231_6_3K_NATIVE – G.723.1, 6.3 kbps
- CODER_TYPE_G726_16K – G.726, 16 kbps, 2 bits/sample
- CODER_TYPE_G726_16K_NATIVE – G.726, 16 kbps, 2 bits/sample
- CODER_TYPE_G726_24K – G.726, 24 kbps, 3 bits/sample
- CODER_TYPE_G726_24K_NATIVE – G.726, 24 kbps, 3 bits/sample
- CODER_TYPE_G726_32K – G.726, 32 kbps, 4 bits/sample
- CODER_TYPE_G726_32K_NATIVE – G.726, 32 kbps, 4 bits/sample
- CODER_TYPE_G726_40K – G.726, 40 kbps, 5 bits/sample
- CODER_TYPE_G726_40K_NATIVE – G.726, 40 kbps, 5 bits/sample
- CODER_TYPE_G729 – G.729
- CODER_TYPE_G729_NATIVE – G.729
- CODER_TYPE_G729ANNEXA – G.729 Annex A
- CODER_TYPE_G729ANNEXA_NATIVE – G.729 Annex A
- CODER_TYPE_G729ANNEXB – G.729 Annex B
- CODER_TYPE_G729ANNEXB_NATIVE – G.729 Annex B
- CODER_TYPE_G729ANNEXAWANNEXB – G.729 Annex A with Annex B
- CODER_TYPE_G729ANNEXAWANNEXB_NATIVE – G.729 Annex A with Annex B
- CODER_TYPE_QCELP_8K – QCELP, 8 kbps
- CODER_TYPE_QCELP_13K – QCELP, 13 kbps
- CODER_TYPE_UDPTL_NATIVE – native T.38 fax hairpin

Note: An IPM device using a coder type that includes _NATIVE as part of the value name for the audio coder is not capable of transcoding audio and can therefore only be used in support of native features such as native play and record operations and native hairpinning. An IPM device using a coder type that does **not** include _NATIVE as part of the value name for the audio coder is capable of transcoding audio and also supports native features.

eFrameSize

size of frame for coders that support multiple frame sizes, such as G.711 and G.726. (Other coders have a predefined, standard value for the frame size and have a user-programmable frames per packet field in the IPM_CODER_INFO data structure.) When packets are sent in both directions (that is, when the call to **ipm_StartMedia( )** or **ipm_SetRemoteMediaInfo( )** specifies **eDirection** = DATA_IP_TDM_BIDIRECTIONAL), the application must know the frame size of incoming packets and use eFrameSize to specify that value.

The eIPM_CODER_FRAMESIZE data type is an enumeration which defines the following values, and coder-specific values are listed in Table 4:

- CODER_FRAMESIZE_10 – frame size = 10 ms
- CODER_FRAMESIZE_20 – frame size = 20 ms
- CODER_FRAMESIZE_30 – frame size = 30 ms

unFramesPerPkt

number of frames per packet. Coder-specific values are listed in Table 4. This field cannot be modified for G.711 coders.

Note: This field controls the RTP packet type for the EVRC codec. If it is set to 1, the RTP packet format will be the Header-Free format. If it is set to any other value, the RTP packet format will use the Interleaved/Bundled format. See RFC 3558 for details of these formats.

eVadEnable

flag for enabling/disabling VAD (Voice Activity Detection)

The eIPM_CODER_VAD data type is an enumeration which defines the following values, and coder-specific values are listed in Table 4:

- CODER_VAD_DISABLE – VAD is OFF
- CODER_VAD_ENABLE – VAD is ON

unCoderPayloadType

RTP header payload type using RFC 3551 (supersedes RFC 1890) standard definitions. See RFC 3551 for details of the coder payload types.

The application must set a value that is compatible with the coder type that is specified in the eCoderType field before calling **ipm_StartMedia**( ) or **ipm_ModifyMedia**( ). The application is responsible for negotiating this value between the two endpoints. This may be set to any value for non-standard coders or if the application does not require interoperability with third-party applications.

Values: 0 to 127. 96 to 127 is the dynamic range.

Default value: 0 (specifies G.711 mu-law)

Note: If not using G.711 mu-law, the application must set the appropriate value.

unRedPayloadType

RTP header redundancy payload type using RFC 2198 definitions for redundant packets. The application is responsible for negotiating this value between the two endpoints. This may be set to any value from 96 to 127.

Table 4. Supported Audio Coder Properties

eCoderType	Frame Size (ms)	Frames per Packet (fpp)	eVadEnable Value
CODER_TYPE_AMRNB_4_75K	fixed at 20	1-10	Must be CODER_VAD_ENABLE
CODER_TYPE_AMRNB_5_15K	fixed at 20	1-10	Must be CODER_VAD_ENABLE
CODER_TYPE_AMRNB_5_9K	fixed at 20	1-10	Must be CODER_VAD_ENABLE
CODER_TYPE_AMRNB_6_7K	fixed at 20	1-10	Must be CODER_VAD_ENABLE

NOTES:
HMP = Dialogic® HMP Software; MMP = Dialogic® Multimedia Platform for AdvancedTCA; MMK = Dialogic® Multimedia Kit for PCIe
* Not supported on Dialogic® HMP Software.
** Supported on Dialogic® HMP Software 3.0 Windows only.
† On Dialogic® HMP Software, using fpp value of 1 is not recommended because it requires increased MIPS usage.
‡ Applications must explicitly enable VAD even though G.729a+b implicitly supports VAD.

Table 4. Supported Audio Coder Properties (Continued)

eCoderType	Frame Size (ms)	Frames per Packet (fpp)	eVadEnable Value
CODER_TYPE_AMRNB_7_4K	fixed at 20	1-10	Must be CODER_VAD_ENABLE
CODER_TYPE_AMRNB_7_95K	fixed at 20	1-10	Must be CODER_VAD_ENABLE
CODER_TYPE_AMRNB_10_2K	fixed at 20	1-10	Must be CODER_VAD_ENABLE
CODER_TYPE_AMRNB_12_2K	fixed at 20	1-10	Must be CODER_VAD_ENABLE
CODER_TYPE_AMRNB_NONE	fixed at 20	1-10	Must be CODER_VAD_ENABLE
CODER_TYPE_EVRC *	fixed at 20	1, 2, or 3	Must be CODER_VAD_DISABLE
CODER_TYPE_G711ALAW64K	10, 20, or 30	1	Must be CODER_VAD_DISABLE
CODER_TYPE_G711ULAW64K	10, 20, or 30	1	Must be CODER_VAD_DISABLE
CODER_TYPE_G723_1_5_3K	30	HMP: 1, 2, 3 MMP/MMK: 1, 2	Either value
CODER_TYPE_G723_1_6_3K	30	HMP: 1, 2, 3 MMP/MMK: 1, 2	Either value
CODER_TYPE_G726_16K	20	1, 2, or 3	Must be CODER_VAD_DISABLE
CODER_TYPE_G726_24K	20	1, 2, or 3	Must be CODER_VAD_DISABLE
CODER_TYPE_G726_32K	20	1, 2, or 3	Must be CODER_VAD_DISABLE
CODER_TYPE_G726_40K	20	1, 2, or 3	Must be CODER_VAD_DISABLE
CODER_TYPE_G729ANNEXA	10	1†, 2, 3, or 4	Must be CODER_VAD_DISABLE
CODER_TYPE_G729ANNEXAWANNEXB	10	1†, 2, 3, or 4	Must be CODER_VAD_ENABLE ‡
CODER_TYPE_QCELP_8K *	20	1	Must be CODER_VAD_DISABLE
CODER_TYPE_QCELP_13K *	20	1	Must be CODER_VAD_DISABLE
CODER_TYPE_UDPTL_NATIVE **	30	1	Must be CODER_VAD_DISABLE

NOTES:
HMP = Dialogic® HMP Software; MMP = Dialogic® Multimedia Platform for AdvancedTCA; MMK = Dialogic® Multimedia Kit for PCIe
* Not supported on Dialogic® HMP Software.
** Supported on Dialogic® HMP Software 3.0 Windows only.
† On Dialogic® HMP Software, using fpp value of 1 is not recommended because it requires increased MIPS usage.
‡ Applications must explicitly enable VAD even though G.729a+b implicitly supports VAD.

IPM_AUDIO_CODER_OPTIONS_INFO

```
typedef struct ipm_audio_coder_options_info_tag
{
      unsigned int unVersion;
      unsigned int unCoderOptions;
      unsigned int unParm1;
      int nValue1;
      unsigned int unParm2;
      int nValue2;
} IPM_AUDIO_CODER_OPTIONS_INFO;
```

■ Description

This data structure provides additional options for audio coders, such as AMR, EVRC, and QCELP.

Note: All unused fields in the IPM_AUDIO_CODER_OPTIONS_INFO structure must be set to 0.

Note: Using the AMR-NB resource in connection with one or more Dialogic products does not grant the right to practice the AMR-NB standard. To seek a license patent agreement to practice the standard, contact the VoiceAge Corporation at www.voiceage.com/licensing/php.

■ Field Descriptions

The fields of the IPM_AUDIO_CODER_OPTIONS_INFO data structure are described as follows:

Note: For EVRC and QCELP, where the media type is MEDIATYPE_AUDIO_LOCAL_CODER_OPTIONS_INFO, all field values are ignored by the firmware.

unVersion
: set to IPM_AUDIO_CODER_OPTIONS_INFO_VERSION

unCoderOptions
: For AMR coders where the media type is MEDIATYPE_AUDIO_LOCAL_CODER_OPTIONS_INFO, valid values are:

- CODER_OPT_AMR_EFFICIENT - Minimize the amount of network bandwidth
  or
  CODER_OPT_AMR_OCTET - Make packet parsing easier for the AMR application

For AMR coders where the media type is MEDIATYPE_AUDIO_REMOTE_CODER_OPTIONS_INFO, valid values that can be ORed together are:

- CODER_OPT_AMR_EFFICIENT - Minimize the amount of network bandwidth
  or
  CODER_OPT_AMR_OCTET - Make packet parsing easier for the AMR application
- CODER_OPT_AMR_CMR_TRACK - Specify that the transmit bit rate should follow the CMR value in the received packet
  or
  CODER_OPT_AMR_CMR_LIMIT - Specify that the transmit bit rate should follow the

CMR value in the received packet with a maximum value specified by the preferred bit rate.

For EVRC and QCELP, set to CODER_OPT_SIGNALING_OFF.

unParm1
: set to a value from the eIPM_CODER_OPTION_PARMS enumeration

For AMR, set to 0.

For EVRC, where the media type is MEDIATYPE_AUDIO_REMOTE_CODER_OPTIONS_INFO, valid values are:

- CODER_OPT_INTERLEAVE_LENGTH - Reduce the listener's perception of data loss by spreading such a loss over non-consecutive vocoder frames.
- CODER_OPT_CDMA_RATE_REDUC - Rate reduction. Bit rate is varied to achieve a variety of average bit rates for more flexibility in bandwidth usage.

For QCELP, where the media type is MEDIATYPE_AUDIO_REMOTE_CODER_OPTIONS_INFO, valid values are:

- CODER_OPT_CDMA_RATE_REDUC - Rate reduction. Bit rate is varied to achieve a variety of average bit rates for more flexibility in bandwidth usage.

nValue1
: The value set here is for the parameter specified in unParm1.

For AMR, set to 0.

For EVRC:

- where unParm1=CODER_OPT_INTERLEAVE_LENGTH, possible values are in the range 0 to 7. Default value is 0.
- where unParm1=CODER_OPT_CDMA_RATE_REDUC, possible values are 0 and 4. Default value is 0.

For QCELP 8 kbps:

- where unParm1=CODER_OPT_CDMA_RATE_REDUC, possible values are 0 and 4. Default value is 0.

For QCELP 13 kbps:

- where unParm1=CODER_OPT_CDMA_RATE_REDUC, possible values are 0, 1, 2, 3, and 4. Default value is 0.

unParm2
: For AMR and EVRC, set to 0.

nValue2
: For AMR and EVRC, set to 0.

■ Example

The following code example shows how to set options when using an EVRC coder type.

```
... /* Setup IP address here */

// Local Audio Coder
```

```
ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eCoderType = CODER_TYPE_EVRC;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eFrameSize = CODER_FRAMESIZE_20;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unFramesPerPkt = 2;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eVadEnable = CODER_VAD_DISABLE;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unCoderPayloadType = 96;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unRedPayloadType = 0;

unCount++;

// Remote Audio Coder

ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eCoderType = CODER_TYPE_EVRC;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eFrameSize = CODER_FRAMESIZE_20;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unFramesPerPkt = 2;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eVadEnable = CODER_VAD_DISABLE;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unCoderPayloadType = 96;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.unRedPayloadType = 0 ;

unCount++;

ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_OPTIONS_INFO;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unVersion =
IPM_AUDIO_CODER_OPTIONS_INFO_VERSION;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unCoderOptions =
CODER_OPT_SIGNALING_OFF;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unParm1 =
CODER_OPT_INTERLEAVE_LENGTH;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.nValue1 = 6;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.unParm2 = 0;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderOptionsInfo.nValue2 = 0;

unCount++;

ipmMediaInfo.unCount = unCount;
```

IPM_CLOSE_INFO

■ Description

This structure is used by the **ipm_Close( )** function.

Note: This structure is reserved for future use. NULL must be passed.

IPM_DIGIT_INFO

```
typedef struct ipm_digit_info_tag
{
    eIPM_DIGIT_TYPE eDigitType;              /* Type of digits */
    eIPM_DIGIT_DIRECTION eDigitDirection;    /* The direction of flow of digits */
    char          cDigits[MAX_IPM_DIGITS];   /* the digits */
    unsigned int  unNumberOfDigits;          /* Number of digits */
    unsigned int  unTimeStamp;
    unsigned int  unExpirationTime;
    unsigned int  unDuration;

} IPM_DIGIT_INFO, *PIPM_DIGIT_INFO;
```

Description

This structure is used to send and receive digits over the TDM bus using the **ipm_SendDigits( )** and **ipm_ReceiveDigits( )** functions. If your application makes a **ipm_SendDigits( )** call, it must fill in the digit type, direction, number of digits, and the actual digits to be sent. If your application makes a **ipm_ReceiveDigits( )** call, all fields are filled in upon successful return.

Field Descriptions

The fields of the IPM_DIGIT_INFO data structure are described as follows:

eDigitType
: must be set to DIGIT_ALPHA_NUMERIC

  The eIPM_DIGIT_TYPE data type is an enumeration which identifies the type of digit. The enumeration defines the following value:
  - DIGIT_ALPHA_NUMERIC – alphanumeric digits

eDigitDirection
: must be set to set to DIGIT_TDM

  The eIPM_DIGIT_DIRECTION data type is an enumeration which identifies the direction of digit flow. The enumeration defines the following value:
  - DIGIT_TDM – digits are sent to or received from the TDM bus

cDigits[MAX_IPM_DIGITS]
: when sending digits, the actual digits to be sent; not used when receiving digits

unNumberOfDigits
: number of digits being sent or received. When sending digits via **ipm_SendDigits( )**, this field indicates the number of digits to be sent; the maximum number of digits that may be sent is 16. When receiving digits via **ipm_ReceiveDigits( )**, upon return the function sets this field to the actual number of digits to be received via asynchronous events.

unTimeStamp
: reserved for future use; set to 0

unExpirationTime
: reserved for future use; set to 0

unDuration
: reserved for future use; set to 0

IPM_ENDPOINTID_INFO

```
typedef struct
{
      unsigned int       unVersion;
      eIPM_MEDIA_TYPE    eMediaType;                 /* Type of media */
      unsigned short     unPortId;                   /* Port ID */
      unsigned char      cIPAddress[IP_ADDR_SIZE];   /* IP Address */
      unsigned long      ulSSRC;                     /* Identifier in the packet */
      unsigned char      ucPayloadType;              /* RTP only */
      unsigned short     usPayloadSize;              /* RTP only */
      unsigned short     usSequenceNum;              /* RTP only */
      unsigned long      ulTimeStamp;                /* RTP only */
} IPM_ENDPOINTID_INFO;
```

■ Description

The IPM_ENDPOINTID_INFO data structure contains information about the endpoint. This structure is used by the Network Address Translation (NAT) traversal feature.

■ Field Descriptions

The fields of the IPM_ENDPOINTID_INFO data structure are described as follows:

UnVersion
: version of the structure. Used to ensure that an application is binary compatible with future changes to this data structure.

eMediaType
: Specifies the type of media type. Valid values are:
  - MEDIATYPE_AUDIO_REMOTE_RTP_INFO
  - MEDIATYPE_AUDIO_REMOTE_RTCP_INFO
  - MEDIATYPE_VIDEO_REMOTE_RTP_INFO
  - MEDIATYPE_VIDEO_REMOTE_RTCP_INFO

unPortId
: Specifies the RTP or RTCP port information to be sent to the application. Valid port ranges are per standards for both RTP and RTCP packets.

cIPAddress
: Specifies the IP address of the endpoint. Valid addresses are according to standards. This field applies to both RTP and RTCP packets.

ulSSRC
: Specifies the unique source identifier that appears along with the RTP or RTCP packets.

ucPayloadType
: Specifies the payload type. This field applies to RTP packets only. For RTCP packets, the event data is set to zero.

usPayloadSize
: Specifies the size of the RTP payload. This field applies to RTP packets only. For RTCP packets, the event data is set to zero.

ulTimeStamp

Specifies the time stamp indicated in the RTP packet. This field applies to RTP packets only and is set to zero for RTCP packets.

IPM_EVENT_INFO

```
typedef struct ipm_event_info_tag
{
    unsigned int unCount;        /* number of following structures */
    void         *pEventData;    /* Data associated with the event */
} IPM_EVENT_INFO, *PIPM_EVENT_INFO;
```

■ Description

This structure is used for IP event notification. See Chapter 25, “Events” for more information.

■ Field Descriptions

The fields of the IPM_EVENT_INFO data structure are described as follows:

unCount
: number of data structures pointed to

*pEventData
: pointer to structure containing event-specific data

IPM_FAX_SIGNAL

```
typedef struct sc_tsinfo {
   eIPM_TONE eToneType;
   unsigned int unToneDuration;

} IPM_FAX_SIGNAL, *PIPM_FAX_SIGNAL;
```

■ Description

This structure defines the tone information detected by the gateway. IPM_FAX_SIGNAL is a child of IPM_MEDIA, which is a child of the IPM_MEDIA_INFO structure. The structure is used by the **ipm_GetLocalMediaInfo( )**, **ipm_ModifyMedia( )** and **ipm_StartMedia( )** functions.

■ Field Descriptions

The fields of the IPM_FAX_SIGNAL data structure are described as follows:

eToneType
: identifies type of tone to generate. The following values are defined for the eIPM_TONE enumeration:
  - TONE_NONE – no tone
  - TONE_CNG – calling (CNG) tone. Tone produced by fax machines when calling another fax machine.
  - TONE_CED – called terminal identification (CED) tone. Tone produced by fax machine when answering a call.

unToneDuration
: duration of tone to generate

IPM_IFRAME_INFO

```
typedef struct iframeinfo
{
    unsigned int          unVersion;
} IPM_IFRAME_INFO;
```

■ Description

This structure contains the I-frame information and is used by the **ipm_GenerateIFrame( )** function.

The INIT_IPM_IFRAME_INFO inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_IFRAME_INFO 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.

IPM_INIT_FAILURE

```
typedef struct IPM_INIT_FAILURE_tag
{
      unsigned int                          unVersion;
      eIPM_RTP_PROTOCOL                     eProtocol;
      union
      {
              PIPM_NBUP_INIT_FAILURE        pNBUP;
      } data;
} IPM_INIT_FAILURE, *PIPM_INIT_FAILURE;
```

Description

The IPM_INIT_FAILURE structure describes a failure. This structure is the wrapper structure around protocol-specific structures (that is, the structure is a generic structure that contains more specific data). This structure is used by **ipm_InitSend( )** and **ipm_InitResponseSend( )**.

The INIT_IPM_INIT_FAILURE inline function is provided to initialize the structure.

Field Descriptions

The fields of the IPM_INIT_FAILURE 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.

eProtocol
: type of RTP encapsulation protocol being described. The eIPM_RTP_PROTOCOL data type is an enumeration that defines the following value:
  - RTP_PROTOCOL_NBUP – describes an Nb UP protocol

data
: data structure containing information about this initialization message. See IPM_NBUP_INIT_SEND for details.

IPM_INIT_RECEIVED

```
typedef struct IPM_INIT_RECEIVED_tag
{
      unsigned int                      unVersion;
      eIPM_RTP_PROTOCOL                 eProtocol;

      union
      {
           IPM_NBUP_INIT_RECEIVED      *pNBUP;
      } data;
} IPM_INIT_RECEIVED, *PIPM_INIT_RECEIVED;
```

■ Description

The IPM_INIT_RECEIVED structure is the wrapper structure around protocol-specific structures (that is, the structure is a generic structure that contains more specific data). This structure is included as part of unsolicited IPMEV_INIT_RECEIVED event (enabled by EVT_INIT_RECEIVED).

The INIT_IPM_INIT_RECEIVED inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_INIT_RECEIVED 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.

eProtocol
: type of RTP encapsulation protocol being described. The eIPM_RTP_PROTOCOL data type is an enumeration that defines the following value:
  - RTP_PROTOCOL_NBUP – describes an Nb UP protocol

data
: data structure containing information about this initialization request. See IPM_NBUP_INIT_RECEIVED structure for details.

IPM_INIT_RESPONSE

```
typedef struct IPM_INIT_RESPONSE_tag
{
      unsigned int                      unVersion;
      eIPM_RTP_PROTOCOL                 eProtocol;
      union
      {
            PIPM_NBUP_INIT_RESPONSE     pNBUP;
      } data;
} IPM_INIT_RESPONSE, *PIPM_INIT_RESPONSE;
```

Description

The IPM_INIT_RESPONSE structure is the wrapper structure around protocol-specific structures (that is, the structure is a generic structure that contains more specific data). This structure is used by **ipm_InitResponseSend( )**.

The INIT_IPM_INIT_RESPONSE inline function is provided to initialize the structure.

Field Descriptions

The fields of the IPM_INIT_RESPONSE 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.

eProtocol
: type of RTP encapsulation protocol being described. The eIPM_RTP_PROTOCOL data type is an enumeration that defines the following value:
  - RTP_PROTOCOL_NBUP – describes an Nb UP protocol

data
: data structure containing information about this initialization response. See IPM_NBUP_INIT_RESPONSE structure for details.

IPM_INIT_SEND

```
typedef struct IPM_INIT_SEND_tag
{
      unsigned int                    unVersion;
      eIPM_RTP_PROTOCOL               eProtocol;
      union
      {
             PIPM_NBUP_INIT_SEND        pNBUP;
      } data;
} IPM_INIT_SEND, *PIPM_INIT_SEND;
```

■ Description

The IPM_INIT_SEND structure is the wrapper structure around protocol-specific structures (that is, the structure is a generic structure that contains more specific data). This structure is used by **ipm_InitSend( )**.

The INIT_IPM_INIT_SEND inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_INIT_SEND 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.

eProtocol
: type of RTP encapsulation protocol being described. The eIPM_RTP_PROTOCOL data type is an enumeration that defines the following value:
  - RTP_PROTOCOL_NBUP – describes an Nb UP protocol

data
: data structure containing information about this initialization message. See IPM_NBUP_INIT_SEND for details.

IPM_MEDIA

```
struct IPM_MEDIA_tag
{
   eIPM_MEDIA_TYPE eMediaType;
   union
   {
      IPM_PORT_INFO           PortInfo;          /* RTP port information     */
      IPM_AUDIO_CODER_INFO    AudioCoderInfo;     /* Audio coder information  */
      IPM_FAX_SIGNAL          FaxSignal;          /* Fax signal information   */
      IPM_VIDEO_CODER_INFO    VideoCoderInfo      /* Video coder information  */
      IPM_SECURITY_INFO       SecurityInfo        /* Security information     */
      IPM_AUDIO_CODER_OPTIONS_INFO  AudioCoderOptionsInfo;
                                          /* Extended Audio Coder Information */
      IPM_NBUP_PROFILE_INFO  NBUPProfileInfo;     /* NBUP Profile Information */
   }mediaInfo;
} IPM_MEDIA, *PIPM_MEDIA;
```

■ Description

This structure contains information about RTP / RTCP ports, coders, security information and fax signals. It is a parent structure of:

- IPM_PORT_INFO
- IPM_AUDIO_CODER_INFO
- IPM_FAX_SIGNAL
- IPM_VIDEO_CODER_INFO
- IPM_SECURITY_INFO
- IPM_AUDIO_CODER_OPTIONS_INFO
- IPM_NBUP_PROFILE_INFO

This structure is a child of the IPM_MEDIA_INFO structure which is used by the **ipm_GetLocalMediaInfo( )**, **ipm_ModifyMedia( )** and **ipm_StartMedia( )** functions.

Fields and values related to security are not supported on all releases; for support information, see Chapter 2, “Feature Support by Platform”.

■ Field Descriptions

The fields of the IPM_MEDIA data structure are described as follows:

eMediaType
: type of media used to start or modify an IP session

The eIPM_MEDIA_TYPE data type is an enumeration which defines the following values:

Audio values:

- MEDIATYPE_AUDIO_LOCAL_CODER_INFO – local receive coder information
- MEDIATYPE_AUDIO_LOCAL_CODER_OPTIONS_INFO – options for audio coders at the local side
- MEDIATYPE_AUDIO_LOCAL_RTCP_INFO – local RTCP audio port information
- MEDIATYPE_AUDIO_LOCAL_RTP_INFO – local RTP audio port information

- MEDIATYPE_AUDIO_LOCAL_SECURITY_INFO – local receive security audio information
- MEDIATYPE_AUDIO_REMOTE_CODER_INFO – remote receive coder information
- MEDIATYPE_AUDIO_REMOTE_CODER_OPTIONS_INFO – options for audio coders at the remote side
- MEDIATYPE_AUDIO_REMOTE_RTCP_INFO – remote RTCP audio port information
- MEDIATYPE_AUDIO_REMOTE_RTP_INFO – remote RTP audio port information
- MEDIATYPE_AUDIO_REMOTE_SECURITY_INFO – remote receive audio security information

Video values:

- MEDIATYPE_VIDEO_LOCAL_CODER_INFO – local receive video coder information
- MEDIATYPE_VIDEO_LOCAL_RTCP_INFO – local RTCP video port information
- MEDIATYPE_VIDEO_LOCAL_RTP_INFO – local RTP video port information
- MEDIATYPE_VIDEO_LOCAL_SECURITY_INFO – local receive video security information
- MEDIATYPE_VIDEO_REMOTE_CODER_INFO – remote receive video coder information
- MEDIATYPE_VIDEO_REMOTE_RTCP_INFO – remote RTCP video port information
- MEDIATYPE_VIDEO_REMOTE_RTP_INFO – remote RTP video port information
- MEDIATYPE_VIDEO_REMOTE_SECURITY_INFO – remote receive video security information

Nb UP values:

- MEDIATYPE_NBUP_REMOTE_RTP_INFO – Nb UP remote RTP information
- MEDIATYPE_NBUP_LOCAL_RTP_INFO – Nb UP local RTP information
- MEDIATYPE_NBUP_PROFILE_INFO – Nb UP profile information

Fax values:

- MEDIATYPE_FAX_SIGNAL – fax signal information to be transmitted to IP during fax transmissions
- MEDIATYPE_LOCAL_UDPTL_T38_INFO – local UDP packet T.38 information
- MEDIATYPE_REMOTE_UDPTL_T38_INFO – remote UDP packet T.38 information

Note: The following eMediaType equates are also provided for backward compatibility (the generic names are equated with audio port values). However, it is recommended that the AUDIO values be used:

```
#define MEDIATYPE_LOCAL_CODER_INFO MEDIATYPE_AUDIO_LOCAL_CODER_INFO
#define MEDIATYPE_LOCAL_RTCP_INFO MEDIATYPE_AUDIO_LOCAL_RTCP_INFO
#define MEDIATYPE_LOCAL_RTP_INFO MEDIATYPE_AUDIO_LOCAL_RTP_INFO
#define MEDIATYPE_REMOTE_CODER_INFO MEDIATYPE_AUDIO_REMOTE_CODER_INFO
#define MEDIATYPE_REMOTE_RTCP_INFO MEDIATYPE_AUDIO_REMOTE_RTCP_INFO
#define MEDIATYPE_REMOTE_RTP_INFO MEDIATYPE_AUDIO_REMOTE_RTP_INFO
```

PortInfo
: reference to RTP port information data structure, type IPM_PORT_INFO

AudioCoderInfo
: reference to audio coder information data structure, type IPM_AUDIO_CODER_INFO

FaxSignal
: reference to fax signal data structure, type IPM_FAX_SIGNAL

VideoCoderInfo
: reference to video coder information data structure, type IPM_VIDEO_CODER_INFO

SecurityInfo
: reference to security information data structure, type IPM_SECURITY_INFO

AudioCoderOptionsInfo
: reference to extended audio coder information data structure, type IPM_AUDIO_CODER_OPTIONS_INFO

NBUPProfileInfo
: reference to Nb UP profile information data structure, type IPM_NBUP_PROFILE_INFO

IPM_MEDIA_INFO

```
typedef struct ipm_media_info_tag
{
    unsigned int  unCount;
    IPM_MEDIA     MediaData[MAX_MEDIA_INFO];

} IPM_MEDIA_INFO, *PIPM_MEDIA_INFO;
```

■ Description

This structure contains IP media session information for various kinds of media information elements, for example, RTP, RTCP and TDM. This structure is the parent of the IPM_MEDIA structure and is used by the **ipm_GetLocalMediaInfo( )**, **ipm_ModifyMedia( )**, and **ipm_StartMedia( )** functions.

■ Field Descriptions

The fields of the IPM_MEDIA_INFO data structure are described as follows:

unCount
: number of media data structures to follow
maximum number of structures = MAX_MEDIA_INFO

MediaData
: reference to IPM_MEDIA structures

IPM_NACK_SENT

```
typedef struct IPM_NACK_SENT_tag
{
       unsigned int                unVersion;
       eIPM_RTP_PROTOCOL           eProtocol;
       union
       {
            PIPM_NBUP_NACK_SENT    pNBUP;
       } data;
} IPM_NACK_SENT, *PIPM_NACK_SENT;
```

■ Description

The IPM_NACK_SENT structure is the wrapper structure around protocol-specific structures (that is, the structure is a generic structure that contains more specific data). This structure is included as part of unsolicited IPMEV_NACK_SENT event (enabled by EVT_NACK_SENT).

The INIT_IPM_NACK_SENT inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_NACK_SENT 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.

eProtocol
: type of RTP encapsulation protocol being described. The eIPM_RTP_PROTOCOL data type is an enumeration that defines the following value:
  - RTP_PROTOCOL_NBUP – describes an Nb UP protocol

data
: data structure containing information about why the NACK response was sent. See IPM_NBUP_NACK_SENT structure for details.

IPM_NBUP_INIT_FAILURE

```
typedef struct IPM_NBUP_INIT_FAILURE_tag
{
      unsigned int            unVersion;
      unsigned int            unCause;
} IPM_NBUP_INIT_FAILURE, *PIPM_NBUP_INIT_FAILURE;
```

■ Description

The IPM_NBUP_INIT_FAILURE structure identifies the cause of failure to establish an Nb UP session. This structure is a child of IPM_INIT_FAILURE.

The INIT_IPM_NBUP_INIT_FAILURE inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_NBUP_INIT_ FAILURE 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.

unCause
: cause values describing possible reasons for failure:
  - NBUP_ERR_CRCHDR – CRC error of frame header
  - NBUP_ERR_CRCPYLD – CRC error of frame payload
  - NBUP_ERR_UNEXP_FRAMENUM – Unexpected frame number
  - NBUP_ERR_FRAMELOSS – Frame loss
  - NBUP_ERR_UNKNOWN_PDUTYPE – PDU type unknown
  - NBUP_ERR_UNKNOWN_PROCEDURE – Unknown procedure
  - NBUP_ERR_UNKNOWN_RESERVED – Unknown reserved value
  - NBUP_ERR_UNKNOWN_FIELD – Unknown field
  - NBUP_ERR_FRAMESHORT – Frame too short
  - NBUP_ERR_MISSING_FIELDS – Missing fields
  - NBUP_ERR_UNEXP_PDUTYPE – Unexpected PDU type
  - NBUP_ERR_UNEXP_PROCEDURE – Unexpected procedure
  - NBUP_ERR_UNEXP_RFCI – Unexpected RFCI
  - NBUP_ERR_UNEXP_VALUE – Unexpected value
  - NBUP_ERR_INITIALISATION_0 – Initialization failure
  - NBUP_ERR_INITIALISATION_1 – Initialization failure (network error, timer expiry)
  - NBUP_ERR_INITIALISATION_2 – Initialization failure (Iu UP function error, repeated NACK).
  - NBUP_ERR_RATECONTROL_FAILURE – Rate control failure
  - NBUP_ERR_ERROREVENT_FAILURE – Error event failure
  - NBUP_ERR_NOTSUP_TIMEALIGN – Time Alignment not supported
  - NBUP_ERR_TIMEALIGN_ERROR – Requested Time Alignment not possible
  - NBUP_ERR_NOTSUP_VERSION – Iu UP Mode version not supported

IPM_NBUP_INIT_RECEIVED

Description

The IPM_NBUP_INIT_RECEIVED structure and related macros are aliases of the IPM_NBUP_INIT_SEND structure and macros. The data in the incoming Nb UP request will always be the same as the data in the outbound Nb UP request. They are defined as follows:

```
#define IPM_NBUP_INIT_RECEIVED                   IPM_NBUP_INIT_SEND
#define INIT_IPM_NBUP_INIT_RECEIVED              INIT_IPM_NBUP_INIT_SEND
#define FREE_IPM_NBUP_INIT_RECEIVED              FREE_IPM_NBUP_INIT_SEND
```

IPM_NBUP_INIT_RESPONSE

```
typedef struct IPM_NBUP_INIT_RESPONSE_tag
{
       unsigned int                     unVersion;
       eIPM_NBUP_INIT_RESPONSE          eResponse;         /* Type of response */
} IPM_NBUP_INIT_RESPONSE, *PIPM_NBUP_INIT_RESPONSE;
```

■ Description

The IPM_NBUP_INIT_RESPONSE structure identifies the type of response to send to an Nb UP request. This structure is a child of the IPM_INIT_RESPONSE structure.

The INIT_IPM_NBUP_INIT_RESPONSE inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_NBUP_INIT_RESPONSE data structure is 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.

eResponse
: response to an Nb UP request. The eIPM_NBUP_INIT_RESPONSE_TYPE data type is an enumeration that defines the following values:
  - NBUP_INIT_RESPONSE_ACK – acknowledge the response and accept the invitation
  - NBUP_INIT_RESPONSE_NACK – send a NACK response to deny the invitation

IPM_NBUP_INIT_SEND

```
typedef struct ipm_NBUP_INIT_SEND_tag
{
    unsigned int                unVersion;
    unsigned int                unSubFlowCount;     /* Number of subflows per RFCI */
    unsigned int                unRFCICount;        /* Number of RFCI structures */
    IPM_NBUP_RFCI_INFO          *pRFCIs;            /* RFCI structures */
} IPM_NBUP_INIT_SEND, *PIPM_NBUP_INIT_SEND;
```

Description

The IPM_NBUP_INIT_SEND structure describes Nb UP initialization items. Since each RFCI must have the same number of subflows, the count is maintained here. This structure is a child of IPM_INIT_SEND structure. The IPM_NBUP_INIT_RECEIVED structure is an alias of the IPM_NBUP_INIT_SEND.

The INIT_IPM_NBUP_INIT_SEND inline function is provided to initialize any dynamic memory needed by the structure and to fill in the values provided by the user.

The FREE_IPM_NBUP_INIT_SEND inline function is provided to de-allocate any dynamic memory allocated by INIT.

Field Descriptions

The fields of the IPM_NBUP_INIT_SEND 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.

unSubFlowCount
: number of SubFlow structures contained in each RFCI structure

unRFCICount
: number of RFCI structures contained in pRFCIs

pRFCIs
: pointer to an array of IPM_NBUP_RFCI_INFO structures

IPM_NBUP_NACK_SENT

```
typedef struct IPM_NBUP_NACK_SENT_tag
{
      unsigned int                    unVersion;
      eIPM_NBUP_NACK_PROCEDURE        eProcedure;
      unsigned int                    unCause;
} IPM_NBUP_NACK_SENT, *PIPM_NBUP_NACK_SENT;
```

■ Description

The IPM_NBUP_NACK_SENT structure identifies the type of NACK response to send to an Nb UP request. This structure is a child of the IPM_NACK_SENT structure.

The INIT_IPM_NBUP_NACK_SENT inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_NBUP_NACK_SENT 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.

eProcedure
: enumeration describing possible reasons for the NACK response. The eIPM_NBUP_NACK_PROCEDURE data type is an enumeration that defines the following values:
- NBUP_NACK_INITIALIZE – error during initialization
- NBUP_NACK_RATE_CONTROL – rate control problem
- NBUP_NACK_TIME_ALIGNMENT – time alignment error

unCause
: cause values describing possible reasons for NACK cause.
- NBUP_ERR_CRCHDR – CRC error of frame header
- NBUP_ERR_CRCPYLD – CRC error of frame payload
- NBUP_ERR_UNEXP_FRAMENUM – Unexpected frame number
- NBUP_ERR_FRAMELOSS – Frame loss
- NBUP_ERR_UNKNOWN_PDUTYPE – PDU type unknown
- NBUP_ERR_UNKNOWN_PROCEDURE – Unknown procedure
- NBUP_ERR_UNKNOWN_RESERVED – Unknown reserved value
- NBUP_ERR_UNKNOWN_FIELD – Unknown field
- NBUP_ERR_FRAMESHORT – Frame too short
- NBUP_ERR_MISSING_FIELDS – Missing fields
- NBUP_ERR_UNEXP_PDUTYPE – Unexpected PDU type
- NBUP_ERR_UNEXP_PROCEDURE – Unexpected procedure
- NBUP_ERR_UNEXP_RFCI – Unexpected RFCI
- NBUP_ERR_UNEXP_VALUE – Unexpected value
- NBUP_ERR_INITIALISATION_0 – Initialization failure
- NBUP_ERR_INITIALISATION_1 – Initialization failure (network error, timer expiry)

- NBUP_ERR_INITIALISATION_2 – Initialization failure (Iu UP function error, repeated NACK).
- NBUP_ERR_RATECONTROL_FAILURE – Rate control failure
- NBUP_ERR_ERROREVENT_FAILURE – Error event failure
- NBUP_ERR_NOTSUP_TIMEALIGN – Time Alignment not supported
- NBUP_ERR_TIMEALIGN_ERROR – Requested Time Alignment not possible
- NBUP_ERR_NOTSUP_VERSION – Iu UP Mode version not supported

IPM_NBUP_PROFILE_INFO

```
typedef struct ipm_nbup_profile_info_tag
{
      unsigned int                      unVersion;
      eIPM_NBUP_PROFILE_TYPE            eProfileType;
} IPM_NBUP_PROFILE_INFO, *PIPM_NBUP_PROFILE_INFO;
```

■ Description

The IPM_NBUP_PROFILE_INFO structure describes the type of Nb UP profile being requested in the call to **ipm_StartMedia( )**. This structure is a child of IPM_MEDIA.

The INIT_IPM_NBUP_PROFILE_INFO inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_NBUP_PROFILE_INFO 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.

eProfileType
: the type of profile to set up in **ipm_StartMedia( )**. The eIPM_NBUP_PROFILE_TYPE data type is an enumeration that defines the following values:

- NBUP_PROFILE_3G324M – 3G-324M type connection
- NBUP_PROFILE_AMR_NB – AMR-NB type connection
- NBUP_PROFILE_G711ALAW64K_5MS – G.711 A-law 5 ms type connection
- NBUP_PROFILE_G711ALAW64K_20MS – G.711 A-law 20 ms type connection
- NBUP_PROFILE_G711ULAW64K_5MS – G.711 mu-law 5 ms type connection
- NBUP_PROFILE_G711ULAW64K_20MS – G.711 mu-law 20 ms type connection

IPM_NBUP_RFCI_INFO

```
typedef struct ipm_NBUP_RFCI_INFO_tag
{
       unsigned int                unVersion;
       unsigned char               ucID;
       IPM_NBUP_SUBFLOW_INFO       *pSubFlows;
} IPM_NBUP_RFCI_INFO, *PIPM_NBUP_RFCI_INFO;
```

■ Description

This structure describes the RFCI items in the Nb UP initialization message. This structure is a child of IPM_NBUP_INIT_SEND structure.

The INIT_IPM_NBUP_RFCI_INFO inline function is provided to initialize any dynamic memory needed by the structure and to fill in the values provided by the user.

The FREE_IPM_NBUP_RFCI_INFO inline function is provided to de-allocate any dynamic memory allocated by INIT.

■ Field Descriptions

The fields of the IPM_NBUP_RFCI_INFO 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.

ucID
: RAB subflow combination identifier (RFCI) for this defined subflow combination. Valid values are:
  - 0 - 62
  - 63 (RFCI not applicable)

pSubFlows
: a list of subflow structures. The count of subflows is maintained by the value in the IPM_NBUP_SUBFLOW_INFO structure.

IPM_NBUP_SUBFLOW_INFO

```
typedef struct IPM_NBUP_SUBFLOW_INFO_tag
{
      unsigned int                      unVersion;
      eIPM_NBUP_FLOW_SIZE               eFlowSize;
} IPM_NBUP_SUBFLOW_INFO, *PIPM_NBUP_SUBFLOW_INFO;
```

■ Description

This structure describes the attributes of a subflow. This structure is a child of IPM_NBUP_RFCI_INFO structure.

The INIT_IPM_NBUP_SUBFLOW_INFO inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_NBUP_SUBFLOW_INFO 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.

eFlowSize
: number of bits to use for each of the flows in the IPM_NBUP_SUBFLOW_INFO data type structure. The eIPM_NBUP_FLOW_SIZE data type is an enumeration that defines the following values:
- NBUP_FLOW_SIZE_0_BITS – use 0 bits
- NBUP_FLOW_SIZE_320_BITS – use 320 bits
- NBUP_FLOW_SIZE_39_BITS – use 39 bits (for AMR-NB)
- NBUP_FLOW_SIZE_60_BITS – use 60 bits (for AMR-NB)
- NBUP_FLOW_SIZE_81_BITS – use 81 bits (for AMR-NB)
- NBUP_FLOW_SIZE_103_BITS – use 103 bits (for AMR-NB)
- NBUP_FLOW_SIZE_1280_BITS – use 1280 bits (for G.711)

IPM_OPEN_INFO

■ Description

This structure is used by the **ipm_Open( )** function.

Note: This structure is reserved for future use.

IPM_PARM_INFO

```
typedef struct ipm_param_info_tag
{
    eIPM_PARM      eParm;          /* the parameter to set or get */
    void           *pvParmValue;   /* pointer to value of parameter */
} IPM_PARM_INFO, *PIPM_PARM_INFO;
```

■ Description

This structure is used to set or retrieve parameters for an IP channel. The structure is used by the **ipm_GetParm( )** and **ipm_SetParm( )** functions.

■ Field Descriptions

The fields of the IPM_PARM_INFO data structure are described as follows:

eIPM_PARM
: type of parameter to set or get. See Table 5 for supported types and corresponding values. In the table, MMP refers to Dialogic® Multimedia Platform for AdvancedTCA and MMK refers to Dialogic® Multimedia Kit for PCIe.

pvParmValue
: pointer to the value of the parameter. See Table 5 for supported values for each parameter type.

Table 5. eIPM_PARM Parameters and Values

eIPM_PARM Define	Description and Values
PARMBD_RTCPAUDIO_INTERVAL	Supported on Dialogic® HMP Software only. Specifies the RTCP calculation frequency for audio streams. Type: integer. Valid values: 1 to 15. Default: 5. **Note:** 1. This parameter is a board-level parameter; the setting is applied to all IP devices on a virtual board. 2. If the value is changed, the new value is immediately applied to the virtual board. 3. This parameter applies to audio RTP and RTCP packets only.
PARMBD_RTP_SOURCE_FILTER	Supported on Dialogic® HMP Software (HMP 3.0) only. Specifies the filtering of RTP packets at runtime based on the remote IP address and port information for the RTP session in use. Type: Integer. Valid values: 0 (off) and 1 (on). Default: 0.

Table 5. eIPM_PARM Parameters and Values (Continued)

eIPM_PARM Define	Description and Values
PARMBD_RTPAUDIO_PORT_BASE	Supported on Dialogic® HMP Software only. Specifies the RTP base port number for audio streams. Type: integer. Valid values: 0 to 65535. Default: 49152. **Note:** 1. This parameter is a board-level parameter; the setting is applied to all IP devices on a virtual board. 2. If the RTP base port number is changed while a virtual board's channels are in an active call (streaming), the new value does not take effect on the active channels until the calls end. However, modifying the RTP base port number on an IP device with active channels causes unexpected behavior on the active calls. 3. This parameter applies to audio RTP and RTCP packets only.
PARMCH_DISABLE_TX_TELEPHONY_EVENT	Supported on Dialogic® HMP Software only. Enables/disables transmit RFC 2833 digits. • 1 – disable transmission • 0 – enable transmission
PARMCH_DTMFXFERMODE	Specifies DTMF transfer mode. Type: eIPM_DTMFXFERMODE (enumeration). Valid values: • DTMFXFERMODE_INBAND – in-band (default) • DTMFXFERMODE_OUTOFBAND – out-of-band • DTMFXFERMODE_RFC2833 – RFC 2833 **Note:** In order for DTMF event reporting to occur, you must set out-of-band signaling on the receive side.
PARMCH_ECACTIVE	Supported on Dialogic® HMP Software only. Enables/disables echo cancellation. Type: eIPM_ACTIVE (enumeration). Valid values: • ECACTIVE_OFF (default) • ECACTIVE_ON
PARMCH_ECHOTAIL	Supported on Dialogic® HMP Software only. Specifies echo tail length value. Type: eIPM_ECHO_TAIL (enumeration) Valid values: • ECHO_TAIL_NONE • ECHO_TAIL_8 • ECHO_TAIL_16 (default) • ECHO_TAIL_32 • ECHO_TAIL_64
PARMCH_ECNLP_ACTIVE	Supported on Dialogic® HMP Software only. Enables/disables Non-Linear Processing (NLP) value for echo cancellation. NLP is a process in which signals that have a level below a defined threshold are replaced by comfort noise and signals that have a level above the threshold are passed unmodified. Type: Integer. Values: 0 or 1. Default: 1.

Table 5. eIPM_PARM Parameters and Values (Continued)

eIPM_PARM Define	Description and Values
PARMCH_LATENCYALG_AUDIO	Supported on Dialogic® Multimedia Platform for AdvancedTCA (MMP) only. Defines the mode of operation of the jitter buffer. By default, the jitter buffer uses "adaptive mode", which means latency can grow until the available number of frames (PARMCH_LATENCYFRAMEMAX_AUDIO) is depleted. Using fixed mode, the number of frames buffered can grow up to 1.5 times the starting value (PARMCH_LATENCYFRAMEINIT_AUDIO). Type: Integer. Values: 0 (adaptive mode) or 1 (fixed mode). Default: 0. **Note:** Use adaptive mode to minimize audio loss due to abnormal packet reception conditions. Use fixed mode to minimize latency growth by sacrificing audio quality.
PARMCH_LATENCYFRAMEINIT_AUDIO	Supported on MMP only. Specifies the amount of Packet Loss Recovery (PLR) latency (delay) that can be introduced by defining the initial number of frames that can be buffered. This parameter defines the starting value for initial latency. When fixed mode is enabled (see PARMCH_LATENCYALG_AUDIO), the number of frames buffered can grow up to 1.5 times the starting value. For example, if the parameter value is 6, the maximum number of frames that can be buffered is 9. The PLR module attempts to restore packets arriving at the receive end as close as possible to their original time-stamped positions. Arriving packets are decomposed into individual frames, each with a unique time stamp. Each new frame is then stored in a jitter buffer before it is sent to the decoder. This is done to allow packets arriving out of order to be inserted in the queue in the correct order. The size of this jitter buffer is defined by the number of frames stored and is controlled by the PARMCH_LATENCYFRAMEINIT_AUDIO, PARMCH_LATENCYFRAMEMAX_AUDIO, and PARMCH_LATENCYALG_AUDIO parameters. Type: Integer. Valid values: 1 to PARMCH_LATENCYFRAMEMAX_AUDIO (frames). Default: 6. **Note:** While the number of frames to be buffered should be set as high as possible for best quality, too high a value will add unnecessary latency to the system. Generally, the number of frames buffered should be the same size or slightly larger than the number of frames per packet.

Table 5. eIPM_PARM Parameters and Values (Continued)

eIPM_PARM Define	Description and Values
PARMCH_LATENCYFRAMEMAX_AUDIO	Supported on MMP only. Defines the maximum number of frames to be buffered in the Packet Loss Recovery (PLR) frame list. This parameter adds latency only when the buffer is already filled and additional frames arrive before there is space in the buffer. This provides for bursts of packets to arrive, which would have to be discarded otherwise. The PLR module attempts to restore packets arriving at the receive end as close as possible to their original time-stamped positions. Arriving packets are decomposed into individual frames, each with a unique time stamp. Each new frame is then stored in a jitter buffer before it is sent to the decoder. This is done to allow packets arriving out of order to be inserted in the queue in the correct order. The size of this jitter buffer is defined by the number of frames stored and is controlled by the PARMCH_LATENCYFRAMEINIT_AUDIO, PARMCH_LATENCYFRAMEMAX_AUDIO, and PARMCH_LATENCYALG_AUDIO parameters. Type: Integer. Valid values: 30 to 200 (frames). Default: 100. **Note:** 1. On Windows® systems, the upper limit of the value is dependent on the channel density and the kernel memory availability. 2. While the number of frames to be buffered should be set as high as possible for best quality, too high a value will add unnecessary latency to the system. Generally, the number of frames buffered should be the same size or slightly larger than the number of frames per packet.
PARMCH_NOTIFY_EXPIRE_KEY_AUDIO	Supported on Dialogic® HMP Software only. For Secure RTP (SRTP), specifies the time before encryption key expiry at which notification is provided that the encryption key for the audio media type is about to expire. Type: Integer. Values: In 100 ms units. Default: 1 (100 ms). **Note:** If the default setting for a given device is changed at any time after download, the new settings for that device are persistent beyond the lifetime of the application (even if the device is closed and re-opened). The settings stay in effect until either the next download or until the settings are overridden with another ipm_SetParm( ) call.
PARMCH_NOTIFY_EXPIRE_KEY_VIDEO	Supported on Dialogic® HMP Software only. For Secure RTP (SRTP), specifies the time before encryption key expiry at which notification is provided that the encryption key for the video media type is about to expire. Type: Integer. Values: In 100 ms units. Default: 1 (100 ms). **Note:** If the default setting for a given device is changed at any time after download, the new settings for that device are persistent beyond the lifetime of the application (even if the device is closed and re-opened). The settings stay in effect until either the next download or until the settings are overridden with another **ipm_SetParm( )** call.

Table 5. eIPM_PARM Parameters and Values (Continued)

eIPM_PARM Define	Description and Values
PARMCH_RFC2833EVT_RX_PLT	Specifies RFC 2833 event receive payload type. Type: unsigned char. Valid values: 96 to 127. Default: 101. **Note:** On MMP and MMK, setting only applies to the next **ipm_StartMedia( )** even if set during an active RTP session.
PARMCH_RFC2833EVT_TX_PLT	Specifies RFC 2833 event transmit payload type. Type: unsigned char. Valid values: 96 to 127. Default: 101. **Note:** On MMP and MMK, setting only applies to the next **ipm_StartMedia( )** even if set during an active RTP session.
PARMCH_RTCP_ENHANCED_EVENT_FREQ	Controls how often RTCP reporting events are sent to the application. For example, when set to 5, every fifth RTCP reporting event is sent to the application. Type: integer. Valid values: 0 to 255. Default value: 0 (don't send reporting event)
PARMCH_RTCP_ENHANCED_REPORTING	Enables/disables enhanced RTCP reporting. Type: integer. Valid values: 0 or 1. Default value: 0
PARMCH_RX_ADJVOLUME	Specifies volume level adjustment for inbound (from IP) side in 1 dB increments. Type: integer. Valid values: -32 to 31. Default: 0 (no adjustment).
PARMCH_TOS	Specifies type of service in IPv4 headers. This can be either a 7-bit TOS field or a 6-bit DSCP field for Differentiated Services per RFC2474. Type: char. Valid values: 0 to 255. Default: 0. **Note:** On MMP and MMK, setting only applies to the next **ipm_StartMedia( )** even if set during an active RTP session.
PARMCH_TTL	Specifies time-to-live for multicast. Type: char. Valid values: 0 to 255. Default: 1. **Note:** On MMP and MMK, setting only applies to the next **ipm_StartMedia( )** even if set during an active RTP session.
PARMCH_TX_ADJVOLUME	Specifies volume level adjustment for outbound (to IP) side in 1 dB increments. Type: integer. Valid values: -32 to 31. Default: 0 (no adjustment).

IPM_PORT_INFO

```
typedef struct ipm_port_info_tag
{
   unsigned int   unPortId;                   /* The Port ID */
   char           cIPAddress[IP_ADDR_SIZE];   /* IP Address  */

} IPM_PORT_INFO, *PIPM_PORT_INFO;
```

■ Description

This structure contains RTP, RTCP, and T.38 UDP port properties. It is a child of IPM_MEDIA, which is a child of the IPM_MEDIA_INFO structure that is used by **ipm_GetLocalMediaInfo( )** **ipm_ModifyMedia( )** and **ipm_StartMedia( )**.

■ Field Descriptions

The fields of the IPM_PORT_INFO data structure are described as follows:

unPortId
: port identifier

cIPAddress[IP_ADDR_SIZE]
: null-terminated IP address of the port in standard dotted decimal string format; for example, 192.168.0.1

Note: Avoid setting IP address 0.0.0.0 when using **ipm_StartMedia( )** because this may cause a hung port.

IPM_QOS_ALARM_DATA

```
typedef struct ipm_qos_alarm_data_tag
{
   eIPM_QOS_TYPE     eQoSType;          /* The QOS parameter type  */
   eIPM_ALARM_STATE  eAlarmState;       /* indicate if On/Off */

} IPM_QOS_ALARM_DATA, *PIPM_QOS_ALARM_DATA;
```

■ Description

This structure is used to retrieve data associated with QoS alarms, as reported in IPMEV_QOS_ALARM events. It is also a child of the IPM_QOS_ALARM_STATUS structure, which is used by the **ipm_GetQoSAlarmStatus( )** and **ipm_ResetQoSAlarmStatus( )** functions.

The library generates a IPMEV_QOS_ALARM alarm event with ALARM_STATE_ON when a QoS fault threshold is exceeded, and it generates a generates a QoS alarm event with ALARM_STATE_OFF when the fault measurement returns to a sub-threshold level.

■ Field Descriptions

The fields of the IPM_QOS_ALARM_DATA data structure are described as follows:

eQoSType

identifies the alarm event that has occurred

The eIPM_QOS_TYPE data type is an enumeration which defines the following values:

- QOSTYPE_JITTER – QoS alarm for excessive average jitter
- QOSTYPE_LOSTPACKETS – excessive lost packets
- QOSTYPE_RTCP_JB_HIGH – RTCP jitter buffer above the threshold
- QOSTYPE_RTCP_JB_LOW – RTCP jitter buffer below the threshold
- QOSTYPE_RTCP_SCS – RTCP severely concealed second condition
- QOSTYPE_RTCPTIMEOUT – RTCP inactivity
- QOSTYPE_RTPTIMEOUT – RTP inactivity

The following values are defined for Secure RTP (SRTP):

- QOSTYPE_SEC_AUTH_FAIL_AUDIO – excessive number of audio packets failing authentication in a given time interval
- QOSTYPE_SEC_AUTH_FAIL_VIDEO – excessive number of video packets failing authentication in a given time interval
- QOSTYPE_SEC_PKT_REPLAY_AUDIO – excessive number of audio replay packets detected in a given time interval
- QOSTYPE_SEC_PKT_REPLAY_VIDEO – excessive number of video replay packets detected in the given time interval
- QOSTYPE_SEC_MKI_NOMATCH_AUDIO – excessive number of audio packets with an MKI that did not match the list of MKI values on the receiving side in a given time interval
- QOSTYPE_SEC_MKI_NOMATCH_VIDEO – excessive number of video packets with an MKI that did not match the list of MKI values on the receiving side in a given time interval

eAlarmState
alarm on / off flag

The eIPM_ALARM_STATE data type is an enumeration which defines the following values:

- ALARM_STATE_OFF – alarm is OFF
- ALARM_STATE_ON – alarm is ON

IPM_QOS_ALARM_STATUS

```
typedef struct ipm_qos_alarm_status_tag
{
   unsigned int unAlarmCount;
   IPM_QOS_ALARM_DATA QoSData[MAX_ALARM];

} IPM_QOS_ALARM_STATUS, *PIPM_QOS_ALARM_STATUS;
```

■ Description

This structure contains the status of QoS alarms for an IP channel. It is the parent of IPM_QOS_ALARM_DATA and is used by **ipm_GetQoSAlarmStatus( )** and **ipm_ResetQoSAlarmStatus( )**.

■ Field Descriptions

The fields of the IPM_QOS_ALARM_STATUS data structure are described as follows:

unAlarmCount
: number of QoSData structures to follow
: maximum number of alarms = MAX_ALARM

QoSData
: reference to alarm data information structure IPM_QOS_ALARM_DATA

IPM_QOS_SESSION_INFO

```
typedef struct ipm_qos_session_info_tag
{
      eIPM_QOS_TYPE  eQoSType;
      unsigned int   unData;

} IPM_QOS_SESSION_INFO, *PIPM_QOS_SESSION_INFO;
```

■ Description

This structure reports statistical Quality of Service information for an IP session. It is a child of the IPM_SESSION_INFO structure which is filled in when **ipm_GetSessionInfo( )** returns successfully.

■ Field Descriptions

The fields of the IPM_QOS_SESSION_INFO data structure are described as follows:

eQoSType
: identifies the QoS alarm to retrieve statistics for

The eIPM_QOS_TYPE data type is an enumeration which defines the following values:

- QOSTYPE_JITTER – average jitter (in msec) since beginning of call
- QOSTYPE_LOSTPACKETS – percentage of lost packets since beginning of call
- QOSTYPE_RTCP_JB_HIGH – RTCP jitter buffer above the threshold
- QOSTYPE_RTCP_JB_LOW – RTCP jitter buffer below the threshold
- QOSTYPE_RTCP_SCS – RTCP severely concealed second condition
- QOSTYPE_RTCPTIMEOUT – RTCP inactivity (in msec)
- QOSTYPE_RTPTIMEOUT – RTP inactivity (in msec)

The following values are defined for Secure RTP (SRTP):

- QOSTYPE_SEC_AUTH_FAIL_AUDIO – number of audio packets failing authentication since the beginning of the call
- QOSTYPE_SEC_AUTH_FAIL_VIDEO – number of video packets failing authentication since the beginning of the call
- QOSTYPE_SEC_PKT_REPLAY_AUDIO – number of audio replay packets detected since the beginning of the call
- QOSTYPE_SEC_PKT_REPLAY_VIDEO – number of video replay packets detected since the beginning of the call
- QOSTYPE_SEC_MKI_NOMATCH_AUDIO – number of audio packets with an MKI that did not match the list of MKI values on the receiving side since the beginning of the call
- QOSTYPE_SEC_MKI_NOMATCH_VIDEO – number of video packets with an MKI that did not match the list of MKI values on the receiving side since the beginning of the call

unData
: value of the QoS parameter

IPM_QOS_THRESHOLD_DATA

```
typedef struct ipm_qos_threshold_data_tag
{
    eIPM_QOS_TYPE eQoSType;
    unsigned int  unTimeInterval;
    unsigned int  unDebounceOn;
    unsigned int  unDebounceOff;
    unsigned int  unFaultThreshold;
    unsigned int  unPercentSuccessThreshold;
    unsigned int  unPercentFailThreshold;

} IPM_QOS_THRESHOLD_DATA, *PIPM_QOS_THRESHOLD_DATA;
```

■ Description

This structure contains the threshold values for QoS alarms for an IP channel. It is a child of the IPM_QOS_THRESHOLD_INFO structure which is used by **ipm_GetQoSThreshold( )** and **ipm_SetQoSThreshold( )**. When enabling a QoS alarm, default threshold and timing values as shown in Table 6 are used unless **ipm_SetQoSThreshold( )** is specified to set non-default values. Note that when an application sets a specific value for any field of a IPM_QOS_THRESOLD_DATA structure, it must explicitly set *all* fields in the structure even when default values are desired for some of the fields.

■ Field Descriptions

The fields of the IPM_QOS_THRESHOLD_DATA data structure are described as follows:

eQoSType
: type of QoS parameter to measure

  The eIPM_QOS_TYPE data type is an enumeration which defines the following values:
  - QOSTYPE_JITTER – jitter
  - QOSTYPE_LOSTPACKETS – lost packets
  - QOSTYPE_RTCP_JB_HIGH – RTCP jitter buffer above the threshold
  - QOSTYPE_RTCP_JB_LOW – RTCP jitter buffer below the threshold
  - QOSTYPE_RTCP_SCS – RTCP severely concealed second condition (SCS)
  - QOSTYPE_RTCPTIMEOUT – RTCP inactivity
  - QOSTYPE_RTPTIMEOUT – RTP inactivity
  - QOSTYPE_SEC_AUTH_FAIL_AUDIO – authentication failure on audio packets
  - QOSTYPE_SEC_AUTH_FAIL_VIDEO – authentication failure on video packets
  - QOSTYPE_SEC_PKT_REPLAY_AUDIO – detection of audio packet replay
  - QOSTYPE_SEC_PKT_REPLAY_VIDEO – detection of video packet replay
  - QOSTYPE_SEC_MKI_NOMATCH_AUDIO – unknown MKI value in audio packets
  - QOSTYPE_SEC_MKI_NOMATCH_VIDEO – unknown MKI value in video packets

unTimeInterval
: time interval (in ms) between successive parameter measurements. Value should be set to a multiple of 100; other values are rounded to the nearest hundred.

  Note: Value must be greater than unFaultThreshold for the jitter QoS type.

unDebounceOn
: time interval for detecting potential alarm fault condition. Must be set to a value that is a multiple of unTimeInterval; other values are rounded down to the next lower multiple of unTimeInterval.

  Note: This field is not used for RTCP and RTP Timeout alarms and must be set to 0.

unDebounceOff
: time interval for detecting potential alarm non-fault condition. Must be set to a value that is a multiple of unTimeInterval; other values are rounded down to the next lower multiple of unTimeInterval.

  Note: This field is not used for RTCP and RTP Timeout alarms and must be set to 0.

unFaultThreshold
: fault threshold parameter. The meaning and value range of this field depends on the QoS Type:

- QOSTYPE_JITTER – allowable average jitter, in ms. Range: 0 to 1000 (ms)
- QOSTYPE_LOSTPACKET – allowable percentage of lost packets. Range: 0 to 100 (%)
- QOSTYPE_RTCP_JB_HIGH – allowable RTCP jitter buffer above the threshold. Range: 0 to 65535.
- QOSTYPE_RTCP_JB_LOW – allowable RTCP jitter buffer below the threshold. Range: 0 to 65535.
- QOSTYPE_RTCP_SCS – allowable RTCP severely concealed second condition (SCS). Range: 0 to 65535.
- QOSTYPE_RTCPTIMEOUT – allowable RTCP inactive interval before an alarm is sent, in units of 100 ms. Range: 50 to 1200 (x100 ms)
- QOSTYPE_RTPTIMEOUT – allowable RTP inactive interval before an alarm is sent, in units of 100 ms. Range: 50 to 1200 (x100 ms)
- QOSTYPE_SEC_AUTH_FAIL_AUDIO – allowed number of audio packet authentication failures before an alarm is sent.
- QOSTYPE_SEC_AUTH_FAIL_VIDEO – allowed number of video packet authentication failures before an alarm is sent.
- QOSTYPE_SEC_PKT_REPLAY_AUDIO – allowed number of audio replay packets detected before an alarm is sent.
- QOSTYPE_SEC_PKT_REPLAY_VIDEO – allowed number of video replay packets detected before an alarm is sent.
- QOSTYPE_SEC_MKI_NOMATCH_AUDIO – allowed number of audio packets with MKI mismatches detected before an alarm is sent.
- QOSTYPE_SEC_MKI_NOMATCH_VIDEO – allowed number of video packets with MKI mismatches detected before an alarm is sent.

unPercentSuccessThreshold
: percentage of poll instances in unDebounceOff time interval that the fault threshold must not be exceeded before an “alarm off” event is sent. Allowed values correspond to multiples of the ratio of unDebounceOff to unTimeInterval (i.e., the inverse of the number of poll instances) expressed as an integer percentage; other values are truncated to the next lower percentage multiple.

  Note: This parameter is not used for RTCP and RTP Timeout alarms and must be set to 0.

unPercentFailThreshold
: percentage of poll instances in unDebounceOn time interval that the fault threshold must be exceeded before an “alarm on” event is sent. Allowed values correspond to multiples of the ratio of unDebounceOn to unTimeInterval (i.e., the inverse of the number of poll instances)

expressed as a integer percentage; other values are truncated to the next lower percentage multiple.

Note: This parameter is not used for RTCP and RTP Timeout alarms and must be set to 0.

Table 6. Quality of Service (QoS) Parameter Defaults

QoS Type	Time Interval (ms)	Debounce On (ms)	Debounce Off (ms)	Fault Threshold [1]	% Success Threshold	% Fail Threshold
Jitter	5000	20000	60000	60 (ms)	25	25
Lost Packets	1000	10000	10000	20 (%)	40	40
RTCP Jitter Buffer High	1000	10000	10000	10 (ms)	20	20
RTCP Jitter Buffer Low	1000	10000	10000	10 (ms)	20	20
RTCP SCS	1000	10000	10000	10 (ms)	20	20
RTCP Timeout	1000	0	0	250 (x100ms = 25sec)	0	0
RTP Timeout	1000	0	0	1200 (x100ms = 120sec)	0	0
Audio Authentication Failure	1000	10000	10000	10 packets	20	20
Video Authentication Failure	1000	10000	10000	10 packets	20	20
Audio Packet Replay	1000	10000	10000	10 packets	20	20
Video Packet Replay	1000	10000	10000	10 packets	20	20
Audio MKI No Match	1000	10000	10000	10 packets	20	20
Video MKI No Match	1000	10000	10000	10 packets	20	20

Notes:
1. Units for Fault Threshold are different for different QoS Types. See unit indications in table cells.

QoS debouncing is calculated as an integer number of parameter measurements that must exceed (or fall below) the fault threshold within the debounce interval before an alarm-on (or alarm-off) event is generated. The calculation uses the following formulas:

For QoS alarm-on debouncing:

count = int(int(unDebounceOn/unTimeInterval) * (unPercentFailThreshold/100))

For QoS alarm-off debouncing:

count = int(int(unDebounceOff/unTimeInterval) * (unPercentSuccessThreshold/100))

IPM_QOS_THRESHOLD_INFO

```
typedef struct ipm_qos_threshold_info_tag
{
   unsigned int unCount;
   IPM_QOS_THRESHOLD_DATA QoSThresholdData[MAX_QOS_THRESHOLD];

} IPM_QOS_THRESHOLD_INFO, *PIPM_QOS_THRESHOLD_INFO;
```

■ Description

This structure is used to set and get the threshold values for QoS alarms for a single IP channel. It is the parent of IPM_QOS_THRESHOLD_DATA and is used by **ipm_GetQoSThreshold( )** and **ipm_SetQoSThreshold( )**.

■ Field Descriptions

The fields of the IPM_QOS_THRESHOLD_INFO data structure are described as follows:

unCount
: number of IPM_QOS_THRESHOLD_DATA structures to follow; maximum = MAX_QOS_THRESHOLD

QosThresholdData
: array of structures containing alarm trigger settings

IPM_RTCP_SESSION_INFO

```
typedef struct ipm_rtcp_session_info_tag
{
   unsigned int    unLocalSR_TimeStamp;
   unsigned int    unLocalSR_TxPackets;
   unsigned int    unLocalSR_TxOctets;
   unsigned int    unLocalSR_SendIndication;
   unsigned int    unLocalRR_FractionLost;
   unsigned int    unLocalRR_CumulativeLost;
   unsigned int    unLocalRR_SeqNumber;
   unsigned int    unLocalRR_ValidInfo;
   unsigned int    unRemoteSR_TimeStamp;
   unsigned int    unRemoteSR_TxPackets;
   unsigned int    unRemoteSR_TxOctets;
   unsigned int    unRemoteSR_SendIndication;
   unsigned int    unRemoteRR_FractionLost;
   unsigned int    unRemoteRR_CumulativeLost;
   unsigned int    unRemoteRR_SeqNumber;
   unsigned int    unRemoteRR_ValidInfo;

} IPM_RTCP_SESSION_INFO, *PIPM_RTCP_SESSION_INFO;
```

Description

This structure contains RTCP information for the session. It is a child of the IPM_SESSION_INFO structure which is filled in when **ipm_GetSessionInfo( )** returns successfully.

Field Descriptions

The fields of the IPM_RTCP_SESSION_INFO data structure are described as follows:

unLocalSR_TimeStamp
: time stamp of the RTCP packet transmission from the local sender

unLocalSR_TxPackets
: number of packets sent by the local sender

unLocalSR_TxOctets
: number of bytes sent by the local sender

unLocalSR_SendIndication
: local sender report has changed since the last transmission. Values may be either:
  - FALSE
  - TRUE

unLocalRR_FractionLost
: percentage of packets lost, as computed by the local receiver

unLocalRR_CumulativeLost
: number of packets lost, as computed by the local receiver

unLocalRR_SeqNumber
: last sequence number received from the local receiver

unLocalRR_ValidInfo
: reserved for future use

unRemoteSR_TimeStamp
time stamp of the RTCP packet transmission from the remote sender

unRemoteSR_TxPackets
number of packets sent by the remote sender

unRemoteSR_TxOctets
number of bytes sent by the remote sender

unRemoteSR_SendIndication
remote sender report has changed since the last transmission. Values may be either:
- FALSE
- TRUE

unRemoteRR_FractionLost
percentage of packets lost, as computed by the remote receiver

unRemoteRR_CumulativeLost
number of packets lost, as computed by the remote receiver

unRemoteRR_SeqNumber
last sequence number received from the remote receiver

unRemoteRR_ValidInfo
reserved for future use

IPM_SECURITY_BASE64_KEY

```
typedef struct ipm_security_base64_key_tag
{
    unsigned int    unVersion;
    char            *pcMasterBase64Key;
} IPM_SECURITY_BASE64_KEY, *PIPM_SECURITY_BASE64_KEY;
```

■ Description

This structure contains information relating to Base64-encoded security keys. When the security key type being used is Base64, this structure is a child of the IPM_SECURITY_KEY structure, which in turn is a child of the IPM_SECURITY_INFO structure, which in turn is a child of the IPM_MEDIA structure, which in turn is a child of the IPM_MEDIA_INFO structure used by the **ipm_StartMedia( )** and **ipm_ModifyMedia( )** functions.

The INIT_IPM_SECURITY_BASE64_KEY inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_SECURITY_BASE64_KEY data structure are described as follows:

unVersion
the version of the data structure

pcMasterBase64Key
a character string that is the Base64-encoded master key

Note: pcMasterBase64Key should point to a character array whose length is at least that defined by (unMasterKeyLength + unMasterSaltKeyLength)/6 and rounded up.

IPM_SECURITY_BINARY_KEY

```
typedef struct ipm_security_binary_key_tag
{
    unsigned int    unVersion;
    char            *pcMasterKey;
    char            *pcMasterSaltKey;
} IPM_SECURITY_BINARY_KEY, *PIPM_SECURITY_BINARY_KEY;
```

Description

This structure contains information relating to binary security keys. When the security key type being used is binary, this structure is a child of the IPM_SECURITY_KEY structure, which in turn is a child of the IPM_SECURITY_INFO structure, which in turn is a child of the IPM_MEDIA structure, which in turn is a child of the IPM_MEDIA_INFO structure used by the **ipm_StartMedia( )** and **ipm_ModifyMedia( )** functions.

The IPM_SECURITY_BINARY_KEY inline function is provided to initialize the structure.

Field Descriptions

The fields of the IPM_SECURITY_BINARY_KEY data structure are described as follows:

unVersion
: the version of the data structure

pcMasterKey
: a character string that is the binary master key

Note: pcMasterKey should point to a character array whose length is at least that defined by unMasterKeyLength/8.

pcMasterSaltKey
: a character string that is the binary master salt key

Note: pcMasterSaltKey should point to a character array whose length is at least that defined by unMasterSaltKeyLength/8.

IPM_SECURITY_INFO

```
typedef struct ipm_security_info_tag {
    unsigned int             unVersion;
    unsigned int             unNumKeys;
    eIPM_SECURITY_INFO_MODE  eInfoMode;
    IPM_SECURITY_KEY         *pKeys;
    IPM_SRTP_PARMS           *pParms;
} IPM_SECURITY_INFO, *PIPM_SECURITY_INFO;
```

Description

This structure contains security information. It is a child of the IPM_MEDIA structure, which in turn is a child of the IPM_MEDIA_INFO structure used by the **ipm_StartMedia( )** and **ipm_ModifyMedia( )** functions.

The INIT_IPM_SECURITY_INFO inline function is provided to initialize the structure.

Field Descriptions

The fields of the IPM_SECURITY_INFO data structure are described as follows:

unVersion
: version of the IPM_SECURITY_INFO structure

unNumKeys
: number of keys passed

eInfoMode
: specifies the mode of operation. Currently, there are two modes of operation as defined in the eIPM_SECURITY_INFO_MODE enumeration:

```
typedef enum eipm_security_info_mode_tag
{
    IPM_SECURITY_INFO_MODE_IMMEDIATE             = 1,
    IPM_SECURITY_INFO_MODE_WAITFOREXPIRATION     = 2
} eIPM_SECURITY_INFO_MODE;
```

The meaning of each value is as follows:

- IPM_SECURITY_INFO_MODE_IMMEDIATE (default) – apply new security keys immediately
- IPM_SECURITY_INFO_MODE_WAITFOREXPIRATION – wait until all the specified key lifetimes have expired before applying new keys

pKeys
: pointer to array of IPM_SECURITY_KEY structures. The number of keys this pointer can point to is specified in unNumKeys.

pParms
: pointer to an IPM_SRTP_PARMS structure. The same secure RTP (SRTP) parameter values are applied to all the keys specified.

IPM_SECURITY_KEY

```
typedef struct ipm_security_key_tag
{
    UINT64                  unSRTPLifeTime;
    unsigned int            unVersion;
    unsigned int            unSRTCPLifeTime;
    unsigned int            unMkiLength;
    unsigned int            unMkiValue;
    unsigned int            unMasterKeyLength;
    unsigned int            unMasterSaltKeyLength;
    eIPM_SECURITY_KEY_TYPE  eKeyType;
    void                    *pvMasterKey;
} IPM_SECURITY_KEY, *PIPM_SECURITY_KEY;
```

Description

This structure contains security key information. It is used by the **ipm_SecurityGenMasterKeys( )** function when generating master and salt keys. It is also a child of the IPM_SECURITY_INFO structure, which in turn is a child of the IPM_MEDIA structure, which in turn is a child of the IPM_MEDIA_INFO structure used by the **ipm_StartMedia( )** and **ipm_ModifyMedia( )** functions.

Currently, two types of security keys are supported:

- Binary Keys - the master key and master salt keys are binary and are two separate keys
- Base64-Encoded Keys - the master key and master salt key are combined (as specified in the *Session Description Protocol Security Descriptions for Media Streams* IETF draft) and Base64-encoded

The INIT_IPM_SECURITY_KEY inline function is provided to initialize the structure.

Field Descriptions

The fields of the IPM_SECURITY_KEY data structure are described as follows:

unSRTPLifeTime
: The lifetime of the master keys, that is, the maximum number of SRTP packets that need to be secured with the master key

unVersion
: The version of the structure

unSRTCPLifeTime
: The lifetime of the master keys, that is, the maximum number of SRTCP packets that need to be secured with the master key

unMkiLength
: The length (in bytes) of the Master Key Identifier (MKI) associated with the SRTP master key. The maximum value supported is 4. If unMkiLength is specified as 0 (zero), no MKI is attached to the SRTP packet and the MKI value is ignored.

unMkiValue
: The MKI value is the MKI that needs to be attached to the SRTP packets if the MKI length is non-zero. The maximum value is limited to 4 bytes.

unMasterKeyLength
: The length (in bits) of the master key. The only value supported currently is 128.

unMasterSaltKeyLength
: The length (in bits) of the master salt key. The only value supported currently is 112.

eKeyType
: Identifies the type of key. Two key types are currently supported, binary and Base64-encoded keys. The eIPM_SECURITY_KEY_TYPE datatype is an enumeration that defines the supported key types as follows:

```
typedef enum
{
    IPM_SECURITY_KEYTYPE_BINARY = 1,
    IPM_SECURITY_KEYTYPE_BASE64 = 2
} eIPM_SECURITY_KEY_TYPE;
```

pvMasterKey
: A void pointer to a key structure that corresponds with the key type. Currently, this is a pointer to an IPM_SECURITY_BINARY_KEY structure or an IPM_SECURITY_BASE64_KEY structure depending on the key type specified in the eKeyType field.

IPM_SESSION_INFO

```
typedef struct ipm_session_info_tag
{
    IPM_RTCP_SESSION_INFO  RtcpInfo;
    unsigned int           unQoSInfoCount;
    IPM_QOS_SESSION_INFO   QoSInfo[MAX_QOS_SESSION];

} IPM_SESSION_INFO, *PIPM_SESSION_INFO;
```

■ Description

This structure is a parent structure of the IPM_RTCP_SESSION_INFO and IPM_QOS_SESSION_INFO structures, and it is used by the **ipm_GetSessionInfo( )** function.

■ Field Descriptions

The fields of the IPM_SESSION_INFO data structure are described as follows:

RtcpInfo
: reference to RTCP session information structure IPM_RTCP_SESSION_INFO

unQoSInfoCount
: number of IPM_QOS_SESSION_INFO structures to follow;
maximum sessions = MAX_QOS_SESSION

QoSInfo
: reference to QoS session information structure IPM_QOS_SESSION_INFO

IPM_SESSION_INFOEX

```
typedef struct ipm_session_infoex_tag
{
    unsigned int            unVersion;
    unsigned short          unRTCPDirection;
    unsigned int            unRTCPDataLength;
    void*                   pRTCPReport;

} IPM_SESSION_INFOEX, *PIPM_SESSION_INFOEX;
```

Description

This structure contains RTCP data. It is used by the **ipm_GetSessionInfoEx( )** function.

Field Descriptions

The fields of the IPM_SESSION_INFOEX 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.

unRTCPDirection
: direction of the RTCP report requested (incoming or outgoing)

unRTCPDataLength
: length of the RTCP report

pRTCPReport
: points to the beginning of the RTCP data (that is, the first byte of the UDP payload in an RTCP packet)

IPM_SRTP_PARMS

```
typedef struct ipm_srtp_parm_tag {
    unsigned int            unVersion;
    eIPM_CRYPTO_SUITE       eCryptoSuite;
    unsigned short          usKeyDerivationRate;
    unsigned short          usWindowSizeHint;
    unsigned short          usSrtpUnEncryptedFlag;
    unsigned short          usSrtcpUnEncryptedFlag;
    unsigned short          usSrtpUnAuthenticatedFlag;
    unsigned int            unSsrc;
    unsigned int            unRoc;
    unsigned int            unSeqNum;
} IPM_SRTP_PARMS, * PIPM_SRTP_PARMS;
```

■ Description

This structure contains parameter information for Secure RTP (SRTP). It is a child of the IPM_SECURITY_INFO structure, which in turn is a child of the IPM_MEDIA structure, which in turn is a child of the IPM_MEDIA_INFO structure used by the **ipm_StartMedia( )** and **ipm_ModifyMedia( )** functions.

The INIT_IPM_SRTP_PARMS inline function is provided to initialize the structure.

The default values shown in the inline function equate to the following:

- IPM_SRTP_PARMS_VERSION
- IPM_CRYPTO_SUITE_DEFAULT (IPM_CRYPTO_AES_CM_128_HMAC_SHA1_80)
- IPM_KDR_DEFAULT (0)
- IPM_WINDOW_SIZE_HINT_DEFAULT (64)
- IPM_SRTP_UNENCRYPTED_DEFAULT (0)
- IPM_SRTCP_UNENCRYPTED_DEFAULT (0)
- IPM_SRTP_UNAUTHENTICATED_DEFAULT (0)
- IPM_SSRC_DEFAULT (0)
- IPM_ROC_DEFAULT (0)
- IPM_SEQNUM_DEFAULT (0xFFFFFFFF)

■ Field Descriptions

The fields of the IPM_SRTP_PARMS data structure are described as follows:

unVersion
: version of the IPM_SRTP_PARMS structure

eCryptoSuite
: Crypto suite used for authentication. Possible values are defined by the eIPM_CRYPTO_SUITE enumeration:

```
typedef enum
{
    IPM_CRYPTO_AES_CM_128_HMAC_SHA1_80 = 1,
    IPM_CRYPTO_AES_CM_128_HMAC_SHA1_32 = 2
} eIPM_CRYPTO_SUITE;
```

The enumeration values reflect the following crypto suite parameters as described in the *Session Description Protocol Security Descriptions for Media Streams* IETF draft.

Table 7. Crypto Suite Parameter Values

Characteristic	AES_CM_128_HMAC_SHA1_80	AES_CM_128_HMAC_SHA1_32
Master Key Length	128 bits	128 bits
Salt Value	112 bits	112 bits
Default Lifetime	2^{31} packets	2^{31} packets
Cipher	AES counter mode	AES counter mode
Encryption Key	128 bits	128 bits
MAC	HMAC-SHA1	HMAC-SHA1
Authentication Tag	80 bits	32 bits
SRTP Auth Key Length	160 bits	160 bits
SRTCP Auth Key Length	160 bits	160 bits

usKeyDerivationRate
: defines the Key Derivation Rate (KDR), that is the rate at which session keys are derived from the master key and master salt key. If KDR is set to 0 (zero, the default), the key is derived only once. This field takes an integer value in the range 1 to 24, which corresponds to a KDR value in the range 2^1 to 2^{24}.

usWindowSizeHint
: the SRTP_WINDOW_SIZE parameter that protects against replay attacks (that is, the capturing of a packet and later reinsertion into a stream). The default value (and minimum value) is 64.

usSrtpUnEncryptedFlag
: flag for enabling or disabling the encryption of SRTP packets or the use of the NULL cipher in SRTP. Possible values are 1 (SRTP messages are **not** encrypted) or 0 (SRTP messages are encrypted). The default value is 0.

usSrtcpUnEncryptedFlag
: flag for enabling or disabling the encryption of SRTCP packets or the use of the NULL cipher in SRTCP. Possible values are 1 (SRTCP messages are not encrypted) or 0 (SRTCP messages are encrypted). The default value is 0.

usSrtpUnAuthenticatedFlag
: flag for enabling or disabling SRTP authentication. Possible values are 1 (SRTP messages are not authenticated) or 0 (SRTP messages are authenticated). The default value is 0.

unSsrc
: specifies how the Synchronized Source (SSRC) identifier is obtained. Possible values are:

- 0, obtain the SSRC that is either automatically generated by HMP software for outgoing RTP packets or taken from inbound SRTP packets (default)
- 1, obtain by negotiation via SDP (passed to IP Media library)

Note: The unSsrc field is not currently supported.

unRoc

specifies how the Rollover Counter (ROC) is obtained. Possible values are:

- 0, estimate the ROC from SRTP packets (default)
- 1, obtain by negotiation via SDP (passed to IP Media library)

Note: The unRoc field is not currently supported.

unSeqNum

specifies the starting sequence number for SRTP packets. By default, the sequence number is taken from the SRTP packet. If the starting sequence number is negotiated, the same value can be passed to the IP Media library. The default is 0xFFFFFFFF.

Note: The unSeqNum field is not currently supported.

IPM_TELEPHONY_EVENT_INFO

```
typedef struct ipm_telephony_event_info_tag
{
    unsigned int            unVersion;          /* Structure version for library use only */
    eIPM_TELEPHONY_EVENT_ID  eTelephonyEventID;  /* The named event usually DTMF named event */
    short                   sVolume;            /* The power level for the DTMF event tone*/
    unsigned short          usDuration;         /* Duration for the DTMF digit in ms*/
} IPM_TELEPHONY_EVENT_INFO, *PIPM_TELEPHONY_EVENT_INFO;
```

Description

The IPM_TELEPHONY_EVENT_INFO data structure contains detailed information about a telephony event, for example a DTMF event. This structure is a child structure of the IPM_TELEPHONY_INFO data structure.

Field Descriptions

The fields of the IPM_TELEPHONY_EVENT_INFO data structure are described as follows:

UnVersion
: version of the IPM_TELEPHONY_EVENT_INFO structure. This field is used by the IP Media Library for checking the backward binary compatibility of future versions of the data structure.

eTelephonyEventID
: a named event, typically a DTMF named event. The datatype of the telephony_event field is an eIPM_TELEPHONY_EVENT_ID enumeration that lists all possible tone signal identifiers as described in RFC 2833. The eIPM_TELEPHONY_EVENT_ID is an enumeration with values listed as follows:

- SIGNAL_ID_EVENT_DTMF_0
- SIGNAL_ID_EVENT_DTMF_1
- SIGNAL_ID_EVENT_DTMF_2
- SIGNAL_ID_EVENT_DTMF_3
- SIGNAL_ID_EVENT_DTMF_4
- SIGNAL_ID_EVENT_DTMF_5
- SIGNAL_ID_EVENT_DTMF_6
- SIGNAL_ID_EVENT_DTMF_7
- SIGNAL_ID_EVENT_DTMF_8
- SIGNAL_ID_EVENT_DTMF_9
- SIGNAL_ID_EVENT_DTMF_STAR
- SIGNAL_ID_EVENT_DTMF_POUND
- SIGNAL_ID_EVENT_DTMF_A
- SIGNAL_ID_EVENT_DTMF_B
- SIGNAL_ID_EVENT_DTMF_C
- SIGNAL_ID_EVENT_DTMF_D

sVolume
: the power level associated with the DTMF event tone

usDuration
: the duration of the DTMF digit in milliseconds

IPM_TELEPHONY_INFO

```
typedef struct ipm_telephony_info_tag
{
   unsigned long              UnVersion;      /* Structure version for library use only */
   eIPM_TELEPHONY_INFO_TYPE   eTelInfoType;   /* RFC2833 Info type - named event or tone */
   union
   {
      IPM_TELEPHONY_EVENT_INFO  TelEvtInfo;   /* RFC2833 named event info eg. DTMF digit * /
      IPM_TELEPHONY_TONE_INFO   TelToneInfo;  /* RFC2833 non-standard tone signal Information */
   }TelephonyInfo;
} IPM_TELEPHONY_INFO, *PIPM_TELEPHONY_INFO;
```

■ Description

This structure contains telephony information (such as RFC 2833 information) that is to be transferred over an IP network.

■ Field Descriptions

The fields of the IPM_TELEPHONY_INFO data structure are described as follows:

UnVersion
: version of the IPM_TELEPHONY_INFO structure. This field is used by the IP Media Library for checking the backward binary compatibility of future versions of the data structure.

eTelInfoType
: the information type; for example, an RFC 2833 named event or tone

The eIPM_TELEPHONY_INFO_TYPE data type is an enumeration which defines the following values:

- TEL_INFOTYPE_EVENT – indicates that the union in this structure is the IPM_TELEPHONY_EVENT_INFO structure.
- TEL_INFOTYPE_TONE – indicates that the union in this structure is the IPM_TELEPHONY_TONE_INFO structure. Reserved for future use.

TelephonyInfo.TelToneInfo
: non-standard tone signal information; for example, an RFC 2833 non-standard tone. Reserved for future use.

TelephonyInfo.TelEvtInfo
: named event information; for example, RFC 2833 DTMF digit. See IPM_TELEPHONY_EVENT_INFO for more information.

■ Example

```
#include <stdio.h>
#include <string.h>
#include <srllib.h>
#include <ipmlib.h>

void main()
{
   int nDeviceHandle;
   eIPM_EVENT myEvents[2] ={EVT_TELEPHONY, EVT_JITTER};
   // Register event handler function with srl
   sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,(HDLR)CheckEvent);
   /*  Main Processing
```

```
     */
    /* Need to enable two events for IP device handle, nDeviceHandle.
        ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
    */
    if(ipm_EnableEvents(nDeviceHandle, myEvents, 2, EV_ASYNC) == -1)
    {
       printf("ipm_EnableEvents failed for device name %s with error = %d\n",
       ATDV_NAMEP(nDeviceHandle), ATDV_LASTERR(nDeviceHandle));
       /*  Perform Error Processing
           ..
      */
    }
    /* Continue Processing */
}

void CheckEvent()
{
    int nEventType = sr_getevttype();
    int nDeviceID = sr_getevtdev();
    void *pVoid   = sr_getevtdatap();
    IPM_TELEPHONY_INFO *pTelInfo;

    switch(nEventType)
    {
        /* List of expected events */
        /* Expected reply to ipm_EnableEvents() */
     case IPMEV_EVENT_ENABLED:
           printf("Received IPMEV_EVENT_ENABLED for device = %s\n",  ATDV_NAMEP(nDeviceID));
       break;

        /* Received unsolicited Telephony event (RFC2833 info). */
     case IPMEV_TELEPHONY_EVENT:
            printf("Received IPMEV_TELEPHONY_EVENT for device name = %s\n",
ATDV_NAMEP(nDeviceID));
            pTelInfo = (IPM_TELEPHONY_INFO*)pVoid;

            switch (pTelInfo->eTelInfoType)
            {
            case TEL_INFOTYPE_EVENT:
                  printf("Telephony Info Type = RFC2833 NAMED EVENT INFO!!\n");
                  printf("Telephony Named Event ID = %d\n", pTelInfo-
>TelEvtInfo.eTelephonyEventID);
                  printf("Named Event Volume = %d dB\n", pTelInfo->TelEvtInfo.sVolume);
                  printf("Named Event Duration = %d ms\n", pTelInfo->TelEvtInfo.usDuration);
              break;
            case TEL_INFOTYPE_TONE:
                  printf("Telephony info type TEL_INFOTYPE_TONE is not supported.\n");
              break;
            default:
                  printf("Unknown telephony info type.\n");
              break;
            }
       break;

     default:
           printf("Received unknown event = %d for device = %s\n", nEventType,
ATDV_NAMEP(nDeviceID));
       break;
    }
}
```

IPM_VIDEO_CODER_INFO

```
typedef struct ipm_video_coder_info_tag
{
   unsigned int             unVersion;           /* structure version */
   eIPM_CODER_TYPE          eCoderType;          /* The coder Type */
   unsigned int             unCoderPayloadType;  /* Type of coder payload supported */
   PIPM_VIDEO_CODER_INFO_EX  pExtraCoderInfo;     /* Additional video coder parameters */

} IPM_VIDEO_CODER_INFO, *PIPM_VIDEO_CODER_INFO;
```

■ Description

This structure contains the coder properties that will be used in a multimedia (video) IP session. IPM_VIDEO_CODER_INFO is a child of the IPM_MEDIA union, which is a child of the IPM_MEDIA_INFO structure that is used by the **ipm_GetLocalMediaInfo( )**, **ipm_ModifyMedia( )**, and **ipm_StartMedia( )** functions.

The INIT_IPM_VIDEO_CODER_INFO inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_VIDEO_CODER_INFO data structure are described as follows.

unVersion
: version number 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.

eCoderType
: type of coder to be used for streaming media operations. The eIPM_CODER_TYPE data type is an enumeration that defines the following values:
  - CODER_TYPE_H263 – H.263 video coder
  - CODER_TYPE_H263_1998 – H.263-1998 video coder
  - CODER_TYPE_MP4V_ES – MPEG-4 video coder

unCoderPayloadType
: RTP header payload type using RFC 1890 standard definitions. The application is responsible for negotiating this value between the two endpoints. This may be set to any value for non-standard coders or if the application does not require interoperability with third-party applications. Values: 0 to 127; 96 to 127 is the dynamic payload range.

pExtraCoderInfo
: pointer to IPM_VIDEO_CODER_INFO_EX structure which specifies additional video coder properties

IPM_VIDEO_CODER_INFO_EX

```
typedef struct ipm_video_coder_info_ex_tag
{
    unsigned int            unVersion;            /* structure version */
    eVIDEO_PROFILE          eProfile;             /* Profile ID */
    eVIDEO_LEVEL            eLevel;               /* Ignored for MPEG4 */
    eVIDEO_IMAGE_WIDTH      eImageWidth;          /* Image width */
    eVIDEO_IMAGE_HEIGHT     eImageHeight;         /* Image height */
    unsigned int            unBitRate;            /* Bits per second */
    eVIDEO_FRAMESPERSEC     eFramesPerSec;        /* Frames per second */
    eVIDEO_SAMPLING_RATE    eSamplingRate;        /* Samples per second */
    unsigned int            unVisualConfigSize;   /* Size of szVisualConfiguration string
                                                     in bytes */
    unsigned char*          szVisualConfiguration; /* Representation of visual
                                                     configuration */
}IPM_VIDEO_CODER_INFO_EX, *PIPM_VIDEO_CODER_INFO_EX;
```

■ Description

This structure contains additional coder properties that will be used in a multimedia (video) IP session. This structure is a child of the IPM_VIDEO_CODER_INFO structure. It is used by the **ipm_GetLocalMediaInfo( )**, **ipm_ModifyMedia( )**, and **ipm_StartMedia( )** functions.

The INIT_IPM_VIDEO_CODER_INFO_EX inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the IPM_VIDEO_CODER_INFO_EX data structure are described as follows.

unVersion
: version number 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.

eProfile
: video profile ID for H.263 and MPEG-4; also level ID for MPEG-4. The eVIDEO_PROFILE data type is an enumeration that defines the following values:

- VIDEO_PROFILE_0_H263 – H.263
- VIDEO_PROFILE_LEVEL_SP0_MPEG4 – MPEG-4 Simple Profile Level 0
- VIDEO_PROFILE_LEVEL_SP1_MPEG4 – MPEG-4 Simple Profile Level 1
- VIDEO_PROFILE_LEVEL_SP2_MPEG4 – MPEG-4 Simple Profile Level 2
- VIDEO_PROFILE_LEVEL_SP3_MPEG4 – MPEG-4 Simple Profile Level 3
- VIDEO_PROFILE_DEFAULT – default profile: H.263

eLevel
: video signal level for H.263; not applicable to MPEG-4. The eVIDEO_LEVEL data type is an enumeration that defines the following values:

Note: Set a video signal level that is appropriate for the frame size and frame rate. No field validation is performed by the software.

- VIDEO_LEVEL_10_H263 – video signal level 10
- VIDEO_LEVEL_20_H263 – video signal level 20
- VIDEO_LEVEL_30_H263 – video signal level 30
- VIDEO_LEVEL_DEFAULT – default level: video signal level 10

eImageWidth
: width of video image, in pixels per line. The eVIDEO_IMAGE_WIDTH data type is an enumeration that defines the following values:
  - VIDEO_IMAGE_WIDTH_128 – 128 pixels per line (sub-QCIF)
  - VIDEO_IMAGE_WIDTH_176 – 176 pixels per line (QCIF)
  - VIDEO_IMAGE_WIDTH_352 – 352 pixels per line (CIF)
  - VIDEO_IMAGE_WIDTH_DEFAULT – default image width: 176 pixels per line

eImageHeight
: height of video image, in number of lines. The eVIDEO_IMAGE_HEIGHT data type is an enumeration that defines the following values:
  - VIDEO_IMAGE_HEIGHT_96 – 96 lines (sub-QCIF)
  - VIDEO_IMAGE_HEIGHT_144 – 144 lines (QCIF)
  - VIDEO_IMAGE_HEIGHT_288 – 288 lines (CIF)
  - VIDEO_IMAGE_HEIGHT_DEFAULT – default image height: 144 lines

unBitRate
: the output bit rate of the video signal. Valid values depend on whether video transcoding is enabled or not. (Video transcoding is enabled if the DMFL_TRANSCODE_ON flag is set for video port connections when **dev_PortConnect( )** is called).

  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 the transmitted RTP video stream and sets the video bit rate for this stream. 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.

eFramesPerSec
: video frame rate in frames per second. The eVIDEO_FRAMESPERSEC data type is an enumeration that defines the following values:
  - 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 – default frame rate: 15 fps

eSamplingRate
: sampling rate of media stream, in kHz. The eVIDEO_SAMPLING_RATE data type is an enumeration that defines the following values:
  - VIDEO_SAMPLING_RATE_90000 – sampling rate of 90 kHz
  - VIDEO_SAMPLING_RATE_DEFAULT – default sampling rate: 90 kHz

unVisualConfigsize
: size, in bytes, of the visual configuration data specified in szVisualConfiguration. Set to 0 if no visual configuration data is to be specified.

szVisualConfiguration

hexadecimal representation of octet string that expresses the visual configuration data (not null-terminated). Set to NULL if no visual configuration data is to be specified.

SC_TSINFO

```
typedef struct sc_tsinfo {
   unsigned long   sc_numts;
   long            *sc_tsarrayp;
} SC_TSINFO;
```

Description

This structure defines the TDM bus (CT Bus) time slot information. It is used by **ipm_GetXmitSlot( )**, **ipm_Listen( )**, **ipm_StartMedia( )**, and **ipm_GetLocalMediaInfo( )**.

Field Descriptions

The fields of the SC_TSINFO data structure are described as follows:

sc_numts
: number of time slots to follow; must be set to 1 for this release

sc_tsarrayp
: time slot ID number

Error Codes 27

This chapter describes the error/cause codes supported by the Dialogic® IP Media software error library, *ipmerror.h*. All Dialogic® IP Media library functions return a value that indicates the success or failure of the function call. Success is indicated by a return value of zero or a non-negative number. Failure is indicated by a value of -1.

If a function fails, call the Standard Attribute functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** for the reason for failure. These functions are described in the *Dialogic® Standard Runtime Library API Library Reference*.

If an error occurs during execution of an asynchronous function, the IPMEV_ERROR event is sent to the application. No change of state is triggered by this event. Upon receiving the IPMEV_ERROR event, the application can retrieve the reason for the failure using the SRL functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**.

The IP Media software error library contains the following error codes, listed in alphabetical order. The list also identifies the functions that may return the particular error code.

EIPM_BADPARM
: Bad argument or parameter. All IP Media library functions except **ipm_Open( )**.

EIPM_BUSY
: Device busy. May be returned by: **ipm_InitResponseSend( )**, **ipm_InitSend( )**, **ipm_StartMedia( )**

EIPM_CONFIG
: Configuration error. May be returned by: **ipm_Close( )**

EIPM_EVT_EXIST
: Event already enabled. May be returned by: **ipm_EnableEvents( )**

EIPM_EVT_LIST_FULL
: Too many events. May be returned by: **ipm_EnableEvents( )**

EIPM_FWERROR
: Firmware error. May be returned by: **ipm_Close( )**, **ipm_GetParm( )**, **ipm_GetXmitSlot( )**, **ipm_Listen( )**, **ipm_SetParm( )**, **ipm_Stop( )**, **ipm_UnListen( )**

EIPM_ GENERATEIFRAME _INCAPABLE
: Incapable of generating an I-frame. May be returned by: **ipm_GenerateIFrame( )**

EIPM_INTERNAL
: Internal error. May be returned by: **ipm_DisableEvents( )**, **ipm_EnableEvents( )**, **ipm_GetLocalMediaInfo( )**, **ipm_GetQoSAlarmStatus( )**, **ipm_GetQoSThreshold( )**, **ipm_GetSessionInfo( )**, **ipm_GetXmitSlot( )**, **ipm_InitResponseSend( )**, **ipm_InitSend( )**, **ipm_Listen( )**, **ipm_ReceiveDigits( )**, **ipm_ResetQoSAlarmStatus( )**, **ipm_SendDigits( )**, **ipm_SetQoSThreshold( )**, **ipm_StartMedia( )**, **ipm_UnListen( )**

EIPM_INTERNAL_INIT
: Internal initialization error.

EIPM_INV_DEVNAME
Invalid device name.

EIPM_INV_EVT
Invalid event. May be returned by: **ipm_DisableEvents( )**, **ipm_EnableEvents( )**

EIPM_INV_MODE
Invalid mode. May be returned by: **ipm_GetLocalMediaInfo( )**, **ipm_GetQoSAlarmStatus( )**, **ipm_GetQoSThreshold( )**, **ipm_GetSessionInfo( )**, **ipm_InitResponseSend( )**, **ipm_InitSend( )**, **ipm_ResetQoSAlarmStatus( )**, **ipm_SendDigits( )**, **ipm_SetQoSThreshold( )**, **ipm_StartMedia( )**

EIPM_INV_STATE
Invalid state. Error indicates that initial command did not complete before another function call was made. May be returned by: **ipm_DisableEvents( )**, **ipm_EnableEvents( )**, **ipm_GetLocalMediaInfo( )**, **ipm_GetQoSAlarmStatus( )**, **ipm_GetQoSThreshold( )**, **ipm_GetSessionInfo( )**, **ipm_GetXmitSlot( )**, **ipm_InitResponseSend( )**, **ipm_InitSend( )**, **ipm_Listen( )**, **ipm_ReceiveDigits( )**, **ipm_ResetQoSAlarmStatus( )**, **ipm_SendDigits( )**, **ipm_SetQoSThreshold( )**, **ipm_StartMedia( )**, **ipm_UnListen( )**

EIPM_NOERROR
No error.

EIPM_NOMEMORY
Memory allocation error.

EIPM_RESOURCEINUSE
Resource in use or not available.

EIPM_SRL
SRL error.

EIPM_SRL_SYNC_TIMEOUT
SRL timeout.

EIPM_SYSTEM
System error. May be returned by: **ipm_DisableEvents( )**, **ipm_EnableEvents( )**, **ipm_GetLocalMediaInfo( )**, **ipm_GetQoSAlarmStatus( )**, **ipm_GetQoSThreshold( )**, **ipm_GetSessionInfo( )**, **ipm_GetXmitSlot( )**, **ipm_InitResponseSend( )**, **ipm_InitSend( )**, **ipm_Listen( )**, **ipm_ReceiveDigits( )**, **ipm_ResetQoSAlarmStatus( )**, **ipm_SendDigits( )**, **ipm_SetQoSThreshold( )**, **ipm_StartMedia( )**, **ipm_UnListen( )**

EIPM_TIMEOUT
Timeout.

EIPM_UNSUPPORTED
Function unsupported. May be returned by: **ipm_DisableEvents( )**, **ipm_EnableEvents( )**

Glossary

Codec: see COder/DECoder

COder/DECoder: A circuit used to convert analog voice data to digital and digital voice data to analog audio.

Computer Telephony (CT): Adding computer intelligence to the making, receiving, and managing of telephone calls.

DTMF: Dual-Tone Multi-Frequency

Dual-Tone Multi-Frequency: A way of signaling consisting of a push-button or touch-tone dial that sends out a sound consisting of two discrete tones that are picked up and interpreted by telephone switches (either PBXs or central offices).

Emitting Gateway: called by a G3FE. It initiates IFT service for the calling G3FE and connects to a Receiving Gateway.

E1: The 2.048 Mbps digital carrier system common in Europe.

FCD file: An ASCII file that lists any non-default parameter settings that are necessary to configure a Dialogic® DM3 hardware/firmware product for a particular feature set. The downloader utility reads this file, and for each parameter listed generates and sends the DM3 message necessary to set that parameter value.

Frame: A set of SCbus/CT Bus timeslots which are grouped together for synchronization purposes. The period of a frame is fixed (at 125 µsec) so that the number of time slots per frame depends on the SCbus/CT Bus data rate.

G3FE: Group 3 Fax Equipment. A traditional fax machine with analog PSTN interface.

Gatekeeper: An H.323 entity on the Internet that provides address translation and control access to the network for H.323 Terminals and Gateways. The Gatekeeper may also provide other services to the H.323 terminals and Gateways, such as bandwidth management and locating Gateways.

Gateway: A device that converts data into the IP protocol. It often refers to a voice-to-IP device that converts an analog voice stream, or a digitized version of the voice, into IP packets.

H.323: A set of International Telecommunication Union (ITU) standards that define a framework for the transmission of real-time voice communications through Internet protocol (IP)-based packet-switched networks. The H.323 standards define a gateway and a gatekeeper for customers who need their existing IP networks to support voice communications.

IAF: Internet Aware Fax. The combination of a G3FE and a T.38 gateway.

IFP: Internet Facsimile Protocol

IFT: Internet Facsimile Transfer

International Telecommunications Union (ITU): An organization established by the United Nations to set telecommunications standards, allocate frequencies to various uses, and hold trade shows every four years.

Internet: An inter-network of networks interconnected by bridges or routers. LANs described in H.323 may be considered part of such inter-networks.

Internet Protocol (IP): The network layer protocol of the transmission control protocol/Internet protocol (TCP/IP) suite. Defined in STD 5, Request for Comments (RFC) 791. It is a connectionless, best-effort packet switching protocol.

Internet Service Provider (ISP): A vendor who provides direct access to the Internet.

Internet Telephony: The transmission of voice over an Internet Protocol (IP) network. Also called Voice over IP (VoIP), IP telephony enables users to make telephone calls over the Internet, intranets, or private Local Area Networks (LANs) and Wide Area Networks (WANs) that use the Transmission Control Protocol/Internet Protocol (TCP/IP).

ITU: See International Telecommunications Union.

Jitter: The deviation of a transmission signal in time or phase. It can introduce errors and loss of synchronization in high-speed synchronous communications.

NIC (Network Interface Card): Adapter card inserted into computer that contains necessary software and electronics to enable a station to communicate over network.

PCD file: An ASCII text file that contains product or platform configuration description information that is used by the DM3 downloader utility program. Each of these files identifies the hardware configuration and firmware modules that make up a specific hardware/firmware product. Each type of Dialogic DM3-based product used in a system requires a product-specific PCD file.

PSTN: see Public Switched Telephone Network

Public Switched Telephone Network: The telecommunications network commonly accessed by standard telephones, key systems, Private Branch Exchange (PBX) trunks and data equipment.

Reliable Channel: A transport connection used for reliable transmission of an information stream from its source to one or more destinations.

Reliable Transmission: Transmission of messages from a sender to a receiver using connection-mode data transmission. The transmission service guarantees sequenced, error-free, flow-controlled transmission of messages to the receiver for the duration of the transport connection.

RTCP: Real Time Control Protocol

RTP: Real Time Protocol

SIP: Session Initiation Protocol: an Internet standard specified by the Internet Engineering Task Force (IETF) in RFC 3261. SIP is used to initiate, manage, and terminate interactive sessions between one or more users on the Internet.

T1: A digital transmission link with a capacity of 1.544 Mbps used in North America. Typically channeled into 24 digital subscriber level zeros (DS0s), each capable of carrying a single voice conversation or data stream. T1 uses two pairs of twisted pair wires.

TCP: see Transmission Control Protocol

Terminal: An H.323 Terminal is an endpoint on the local area network which provides for real-time, two-way communications with another H.323 terminal, Gateway, or Multipoint Control Unit. This communication consists of control, indications, audio, moving color video pictures, and/or data between the two terminals. A terminal may provide speech only, speech and data, speech and video, or speech, data, and video.

Transmission Control Protocol: The TCP/IP standard transport level protocol that provides the reliable, full duplex, stream service on which many application protocols depend. TCP allows a process on one machine to send a stream of data to a process on another. It is connection-oriented in the sense that before transmitting data, participants must establish a connection.

UDP: see User Datagram Protocol

UDPTL: Facsimile UDP Transport Layer protocol

User Datagram Protocol: The TCP/IP standard protocol that allows an application program on one machine to send a datagram to an application program on another machine. Conceptually, the important difference between UDP datagrams and IP datagrams is that UDP includes a protocol port number, allowing the sender to distinguish among multiple destinations on the remote machine.

VAD: Voice Activity Detection