Dialogic® PowerMedia™ HMP for Windows Release 3.0

Release Notes

February 2024 05-2508-131

www.dialogic.com

Copyright and Legal Notice

All contents of this document are furnished for informational use only and are subject to change without notice and do not represent a commitment on the part of Enghouse and its affiliates or subsidiaries ("Enghouse"). Reasonable effort is made to ensure the accuracy of the information contained in the document. However, Enghouse does not warrant the accuracy of this information and cannot accept responsibility for errors, inaccuracies or omissions that may be contained in this document.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH DIALOGIC® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN A SIGNED AGREEMENT BETWEEN YOU AND ENGHOUSE, ENGHOUSE ASSUMES NO LIABILITY WHATSOEVER, AND ENGHOUSE DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF ENGHOUSE PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHT OF A THIRD PARTY.

Enghouse products are not intended for use in certain safety-affecting situations.

Due to differing national regulations and approval requirements, certain Enghouse products may be suitable for use only in specific countries, and thus may not function properly in other countries. You are responsible for ensuring that your use of such products occurs only in the countries where such use is suitable. For information on specific products, contact Enghouse at legal.operations@enghouse.com

It is possible that the use or implementation of any one of the concepts, applications, or ideas described in this document, in marketing collateral produced by or on web pages maintained by Enghouse may infringe one or more patents or other intellectual property rights owned by third parties. Enghouse does not provide any intellectual property licenses with the sale of Enghouse products other than a license to use such product in accordance with intellectual property owned or validly licensed by Enghouse and no such licenses are provided except pursuant to a signed agreement with Enghouse. More detailed information about such intellectual property is available from Enghouse's legal department at **80 Tiverton Court, Suite 800 Markham, Ontario L3R 0G4**.

Enghouse encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.

Dialogic, Dialogic Pro, DialogicOne, Dialogic Buzz, Brooktrout, BorderNet, PowerMedia, PowerVille, PowerNova, ControlSwitch, I-Gate, Veraz, Cantata, TruFax, and NMS Communications, among others as well as related logos, are either registered trademarks or trademarks of Enghouse and its affiliates or subsidiaries. Enghouse's trademarks may be used publicly only with permission from Enghouse. Such permission may only be granted by Enghouse legal department at **80 Tiverton Court, Suite 800 Markham, Ontario L3R 0G4**. Any authorized use of Enghouse's trademarks will be subject to full respect of the trademark guidelines published by Enghouse from time to time and any use of Enghouse's trademarks requires proper acknowledgement.

The names of actual companies and products mentioned herein are the trademarks of their respective owners.

This document discusses one or more open source products, systems and/or releases. Enghouse is not responsible for your decision to use open source in connection with Enghouse products (including without limitation those referred to herein), nor is Enghouse responsible for any present or future effects such usage might have, including without limitation effects on your products, your business, or your intellectual property rights.

Table of Contents

1. Revision History

This section summarizes the changes made in this and, if applicable, each previously published version of the Release Notes for PowerMedia™ HMP for Windows Release 3.0, which is a document that is planned to be periodically updated throughout the lifetime of the release.

Revision	Release Date	Notes
05-2508-131	February 2024	Updates to support PowerMedia HMP for Windows 3.0 Service Update 556. Release Issues: HMP-1725, HMP-1681
05-2508-130	December 2023	Updates to support PowerMedia HMP for Windows 3.0 Service Update 554. Release Issues: HMP-1715
05-2508-129	August 2023	Updates to support PowerMedia HMP for Windows 3.0 Service Update 551. Post Release Developments: • Multitrack audio wav recording • Support for GCP as of Service Update 550 • MM streaming support for AMR, AMR-WB and EVS Release Issues: HMP-1695, HMP-1694, HMP-1664, HMP-1424
05-2508-128	May 2023	Updates to support PowerMedia HMP for Windows 3.0 Service Update 550. Release Issues: HMP-1665, HMP-1654, HMP-1640, HMP-1630, HMP-1625.

Revision	Release Date	Notes
05-2508-127	March 2023	Updates to support PowerMedia HMP for Windows 3.0 Service Update 548. Post Release Developments: Added the following new features: • Secure RTP With First Party Call Control • Ipv6 Call Control Release Issues: HMP-1621, HMP-1601, HMP-1522, HMP-1436, HMP-1150.
05-2508-126	December 2022	Updates to support PowerMedia HMP for Windows 3.0 Service Update 545. Release Issues: HMP-1558, HMP-1537, HMP-1521
05-2508-125	September 2022	Updates to support PowerMedia HMP for Windows 3.0 Service Update 543. Release Issues: Added the following resolved defects: HMP-1523, HMP-1517, HMP-1507, HMP-1506, HMP-1500, HMP-1499, HMP-1494.
05-2508-124	March 2022	Updates to support PowerMedia HMP for Windows 3.0 Service Update 540. Added Section 4 that incorporates new API information. Release Issues: Added the following resolved defects: HMP-1472, HMP-1468.

Revision	Release Date	Notes
05-2508-123	December 2021	Updates to support PowerMedia HMP for Windows 3.0 Service Update 538. Post Release Developments: Added the following new features: • NAT Traversal for 1PCC • AMR2 • Transmit RFC 2833/RFC 4733 Tone Events • Microsoft Windows 11 and Windows Server 2022 Release Issues: Added the following resolved defects: HMP-1408, HMP-1332, HMP-1126.
05-2508-122	June 2021	Archived previous revision of the document. Updates to support PowerMedia HMP for Windows 3.0 Service Update 533. Release Issues: Added the following resolved defects: HMP-1317, HMP-1311, HMP-1245, HMP-1244, HMP-1234, HMP-1223, HMP-1149, HMP-1144, HMP-1141, HMP-1136, HMP-965.

2. Post Release Developments

This section describes significant changes after the general availability release.

Media Streaming Support for AMR, AMR-WB and EVS Codecs

HMP 3.0 Windows Service Update 551 adds support for AMR, AMR-WB and EVS codecs when using the media streaming API functions. For more information regarding media streaming APIs, refer to the *Dialogic® Multimedia API Programming Guide and Library Reference*.

Multitrack Audio Wave Recording

With Service Update 551, HMP for Windows adds support for audio recording to dual-track (stereo) .wav files. This multitrack record feature enables applications to record two separate audio sources into different tracks. This feature can be utilized by call centers, E911 applications, banking applications, and monitoring applications to record two audio callers, such as agent and client, as different tracks rather than recording the mixed output of an audio conference. An additional use case of this feature enables applications to capture an audio recording of the HMP system input and output of the caller (i.e., what the caller hears and what the caller says) in a single dual-track (stereo) .wav file.

For more information, refer to the Documentation Updates section later in this document.

Google Cloud Platform (GCP) Support

Dialogic® PowerMedia™ HMP for Windows Release 3.0 Service Update 550 adds support for Google Cloud Platform (GCP) instances running Microsoft Windows Server 2022 Datacenter or later. HMP has been qualified on the e2-medium, n2-standard, n2d-standard, t2d-standard, and c3-highcpu instance types. Two or more VCPU are supported; the required instance size is dependent on the number of ports in use during run time and depends on HMP and application performance requirements.

Note: Depending on use case, an application may need to configure SIP headers ("Contact", "From") using the external IP address of the instance. When using SIP in 1PCC mode, this can be accomplished by using the NAT Traversal Support for 1PCC Applications in Cloud Environments released with Service Update 538.

Secure RTP With First Party Call Control

With Service Update 548, HMP for Windows adds functionality to support Secure RTP calls when using the first party call control (1PCC) model.

For more information, refer to the Documentation Updates section later in this document.

IPv6 Call Control

With Service Update 548, HMP for Windows adds functionality to support IPv6 call control.

For more information, refer to the Documentation Updates section later in this document.

Driver Certification Updates with HMP Windows Service Update 538

HMP Windows SU538 contains updated hardware driver certificates issued by DigiCert. The new root certificate DigiCert Trusted Root G4 must be installed prior to updating to HMP Windows SU538. The G4 certificate is installed automatically as long as "Automatic Root Certificate Update" in the Local Group Policy is enabled.

NAT Traversal Support for 1PCC Applications in Cloud Environments

With Service Update 538, HMP for Windows adds NAT Traversal functionality that enables SIP and RTP access to the public network in cloud environments where a media server only has access to a network interface with a local IP address.

For more information, refer to the Documentation Updates section later in this document.

AMR2 Audio Codec Support

With Service Update 538, HMP for Windows adds support for the AMR2 codec. The AMR2 codec is a restricted subset of AMR codec functionality provided for VoLTE compatibility with older UMTS networks.

For more information, refer to the Documentation Updates section later in this document.

Transmit RFC 2833/RFC 4733 Tone Events

With Service Update 538, HMP for Windows adds support for sending RFC 2833/RFC 4733 telephony events. This feature allows an application to transmit a sequence of both DTMF and non-DTMF telephony events, including hookflash, over an IP network.

For more information, refer to the Documentation Updates section later in this document.

Microsoft Windows 11 and Windows Server 2022 Support

With Service Update 538, HMP for Windows has been qualified to run on Microsoft Windows 11 and Windows Server 2022.

Increased Channel Density

With Service Update 533, Dialogic PowerMedia HMP for Windows Release 3.0 now supports up to 3000 ports of audio.

3. Release Issues

Issue Type	Defect No.	SU No.	Product or Com-ponent	Description
Resolved	HMP-1725	556	Driver	An issue introduced in SU550 that caused delayed or missing events and crashes on some VMWare configurations has been addressed.
Resolved	HMP-1681	556	Driver	An update was made to address a crash in the SSP component when parsing RTCP packets.
Resolved	HMP-1715	554	IPHOST	An issue introduced in SU550 that impacted SIP call rates/density has been addressed.
Resolved	HMP-1695	551	IPHOST	An update was made to prevent additional characters from being included in the 200 OK SDP a=crypto line.
Resolved	HMP-1694	551	IPHOST	An update was made to address an IPHOST library exception when accepting a ReINVITE during a 1PCC SRTP call.
Resolved	HMP-1664	551	IPHOST	An issue that caused SIP TLS connections to fail has been addressed.
Resolved	HMP-1424	551	Driver	An update was made to address a memory leak when mismatched dynamic payload types are used. Note that packets with an incorrect payload type are dropped and not processed.
Resolved	HMP-1665	550	IPHOST	An issue that caused the SIP stack to leak call resources when handling SIP forking has been addressed.
Resolved	HMP-1654	550	IPHOST	An update was made to prevent adding crypto lines to the SDP for outbound calls.
Resolved	HMP-1640	550	IPHOST	An issue that caused an application segfault when no IPv6 proxy was specified has been addressed.
Resolved	HMP-1630	550	IPHOST	An issue that prevented calls from being answered has been addressed.

Issue Type	Defect No.	SU No.	Product or Com-ponent	Description
Resolved	HMP-1625	550	IPHOST	An issue was addressed that prevented use of Elliptic Curve Diffie-Hellman ciphers for SIP TLS.
Resolved	HMP-1621	548	Driver	An issue introduced in SU 545 that caused connection failures between DX devices and thinblade timeslots has been addressed.
Resolved	HMP-1601	548	HMP	An issue was addressed that caused unexpected bit rate changes in RTP streams encoded in EVS.
Resolved	HMP-1522	548	HMP	An issue was addressed that prevented RTCP receiver reports from being sent in RECVEIVEONLY mode.
Resolved	HMP-1436	548	Host Library	An update was made to return an IPMEV_ERROR - "Invalid Parameter" error from ipm_StartMedia() when licensed resources are all in use.
Resolved	HMP-1150	548	IPHOST	An issue that prevented IPv6 calls through an outbound proxy has been addressed.
Resolved	HMP-1558	545	IPHOST	An issue introduced in SU 543 that caused SIP supervised transfer calls to fail has been addressed.
Resolved	HMP-1537	545	IPHOST	An update was made to ensure that SDP is included in SIP 200 OK responses when processing inbound calls.
Resolved	HMP-1521	545	Driver	An issue that caused delayed audio during hair pinned call scenarios has been addressed.
Resolved	HMP-1523	543	IPHOST	An update was made to increase the SIP SDP buffer size to prevent calls from being rejected with "413 Request Entity Too Large" error.
Resolved	HMP-1517	543	HMP	An update was made to address a crash during network driver buffer allocation.
Resolved	HMP-1507	543	HMP	An issue that caused ipm_ModifyMedia() to fail when a Native codec was selected has been addressed.

Issue Type	Defect No.	SU No.	Product or Com-ponent	Description
Resolved	HMP-1506	543	Fax	An issue that caused ipm_StartMedia() failures when using fax T.38 has been addressed.
Resolved	HMP-1500	543	IPHOST	An update was made to ensure that SIP INVITE messages sent after a CANCEL contain the proper SDP.
Resolved	HMP-1499	543	Host Library	An issue that caused an exception during a call to gc_stop() has been addressed.
Resolved	HMP-1494	543	IPHOST	An update was made to ensure that SIP headers containing ":" are properly encoded.
Resolved	HMP-1472	540	IPHOST	An issue that caused SIP registration using Digest Authentication to fail has been resolved.
Resolved	HMP-1468	540	Install	An update was made to ensure that HMP drivers are signed with an updated certificate.
Resolved	HMP-1408	538	IPHOST	An issue that caused an application crash when the SIP "Supported:" header included a comma-separated list was addressed.
Resolved	HMP-1332	538	IPHOST	An update was made so that the SIP "identity:" and "identity-info:" headers can be read and set from an application.
Resolved	HMP-1126	538	HMP	An issue was addressed that prevented IPM completion events from being sent to an application.
Resolved	HMP-1328	538	Fax	An issue that occurred during fax receive that resulted in corrupted files approximately 3% of the time has been resolved.
Resolved	HMP-1311	538	IPHOST	An issue that caused SIP INVITE to be rejected with 400 "Sip Parser Error" for certain SIP "Identity" header formats was addressed.
Resolved	HMP-1317	533	IPHOST	The issue that caused a SIP INVITE that followed a cancelled call not to include SDP was addressed

Issue Type	Defect No.	SU No.	Product or Com-ponent	Description
Resolved	HMP-1245	533	Firmware	An issue that caused an HMP start failure when setting the PrmRFC2833TimeStampSkew parameter was addressed.
Resolved	HMP-1244	533	Firmware	The issue that caused MultiMedia recordings to fail when using the MultiMedia Streaming I/O interface was fixed.
Resolved	HMP-1234	533	Host Library	The library exception that occurred when using the MultiMedia Streaming I/O interface was addressed.
Resolved	HMP-1223	533	IPHOST	The issue that caused incorrect handling of the SIP REINVITE message when multiple RFC2833 lines are present in the SDP was fixed.
Resolved	HMP-1149	533	IPHOST	An issue that caused the gc_AcceptModifyCall () to fail when a REINVITE with remote port change is received was addressed.
Resolved	HMP-1144	533	IPHOST	A fix to allow modification of the SDP content of subsequent SIP OPTIONS messages was made.
Resolved	HMP-1141	533	IPHOST	An issue that occurred when calling gc_SetAuthenticationInfo() with an empty Identity element was fixed.
Resolved	HMP-1136	533	Firmware	An issue that caused SIP calls over an IPv6 network to fail with GCEV_TASKFAIL was addressed.
Resolved	HMP-965	533	IPHOST	An issue that caused the gc_OpenEx() function to fail on systems with IPv6 addresses was fixed.

4. Documentation Updates

NAT Traversal Feature

NAT Traversal functionality enables SIP and RTP access to the public network in cloud environments where a media server only has access to a network interface with a local IP address.

SIP and SDP address translation is configured on gc_Start() by setting the "nat_external_sip_address" and "nat_external_rtp_address" fields in the IP_VIRTBOARD structure.

The SIP external address is used to replace the host part of the addresses in the "From" header and the top "Via" header in outbound SIP request messages. The host part of the address is replaced in the "Contact" header in outbound SIP request messages and outbound SIP response messages. The SIP external address is used in 1PCC and 3PCC operating modes.

An application can use IPSET_SIP_MSGINFO / IPPARM_SIP_HDR to add SIP headers. The application must translate addresses for header types that aren't known to GlobalCall.

The RTP external address is used to replace the host part of the addresses on the o= and c= lines in outbound SDP. The RTP external address is used in 1PCC operating mode only.

The "audio_rtp_base_port" field in the IP_VIRTBOARD structure is used to configure unique UDP port ranges for the IPM devices in 1PCC operating mode when multiple media servers share one public IP address.

In a cloud environment where the media server has a local IP address, only the SIP and SDP external addresses must be configured. The SIP and SDP external addresses are set in the "nat_external_sip_address" and "nat_external_rtp_address" fields of the IP_VIRTBOARD structure.

IP_VIRTBOARD Additions for NAT traversal

The following parameters have been added to the IP_VIRTBOARD structure to support NAT Traversal feature. For more information regarding the IP_VIRBOARD structure, refer to the *Dialogic® Global Call IP Technology Guide*.

nat_external_sip_address (structure version >= 0x118 only)
: Specifies the host address that will replace the host address in From, Contact and Via headers in outbound SIP messages. The value can be any string, e.g. an IPv4 address, an IPv6 address or an FQDN. SIP address translation is disabled by default. This field applies to 1PCC and 3PCC operating modes.

nat_external_rtp_address (structure version >= 0x118 only)
: Specifies the host address that will replace the host addresses on the c= and o= SDP lines in all outbound SDP. The value must be an IPv4 address or an IPv6 address. SDP address translation is disabled by default. This field applies to 1PCC operating mode only.

audio_rtp_base_port (structure version >= 0x118 only)
: Sets the IPM base UDP port. The default value is 0 which means the default IPM base UDP port will be used. This field applies to 1PCC operating mode only.

Configuring Multiple Servers Sharing a Single Public Address

Multiple media servers can also share a single public IP address. Forwarding rules are configured on the NAT device for each media server. Each media server's UDP and TCP ports are configured so they don't overlap.

Media server 1 configuration

SIP UDP port 5060

SIP TCP port 5060

RTP / RTP base UDP port 20000

Media server 2 configuration

SIP UDP port 5070

SIP TCP port 5070

RTP / RTP base UDP port 30000

SIP and RTP to media server 1

Public		Private
SIP 172.1.1.10:5060	->	192.168.1.20:5060
RTP 172.1.1.10:20000	->	192.168.1.20:20000

SIP and RTP to media server 2

Public		Private
SIP 172.1.1.10:5070	->	192.168.1.30:5070
RTP 172.1.1.10:30000	->	192.168.1.30:30000

The following IP_VIRTBOARD fields are used to configure the network interface IP address, UDP ports and TCP port for SIP on gc_Start():

localIP

localIPv6

localIPv6_iface_name

sip_signaling_port

audio_rtp_port_base

AMR2 Codec Support

The AMR2 codec is a restricted subset of AMR codec functionality provided for VoLTE compatibility with older UMTS networks. The use of AMR2 promotes Tandem Free Operation (TFO) and Transcoder Free Operation (TrFO) when a legacy network utilizes a restricted subset of AMR modes.

Support for AMR2 and AMR Mode Change Restrictions are specified as an optional, but recommended requirement by the IMS VoLTE specification IR.92, "IMS Profile for Voice and SMS". AMR2 provides compatibility with multiple AMR codec types, including FR AMR, HR AMR, UMTS AMR, and OHR AMR.

RFC 4867 and 3GPP TS 26.114 describe AMR2 and how the AMR2 features are negotiated using SDP. The IPM device provides the media path processing required for the following SDP parameters:

mode-set

mode-change-period

mode-change-capability

mode-change-neighbor

The following new macros are bitwise OR'd into the unCoderOptions field of the existing IPM_AUDIO_CODER_OPTIONS_INFO structure to configure AMR2.

CODER_OPT_AMR_MODE_CHANGE_NEIGHBOR(neighbor) where "neighbor" is 0 or 1 as defined in RFC 4867.

CODER_OPT_AMR_MODE_CHANGE_PERIOD(period) where "period" is 1 or 2 as defined in RFC 4867.

CODER_OPT_AMR_MODE_SET(set) where each bit in "set" indicates a bitrate in the mode-set. The bit definitions are as follows.

Bit Definition for AMR

Bit	Bitrate (kbps)
0	4.75
1	5.15
2	5.90
3	6.70
4	7.40
5	7.95
6	10.2
7	12.2

Bit Definition for AMR-WB

Bit	Bitrate (kbps)
0	6.60
1	8.85
2	12.65
3	14.25
4	15.85
5	18.25
6	19.85
7	23.05
8	23.85

CMR rule and packing mode configuration are supported in previous releases. The following macros have been added to this release.

CODER_OPT_AMR_CMR_RULE(rule) where "rule" is CODER_OPT_AMR_CMR_TRACK or CODER_OPT_AMR_CMR_LIMIT

CODER_OPT_AMR_PACKING_MODE(mode) where "mode" is CODER_OPT_AMR_OCTET or CODER_OPT_AMR_EFFICIENT

Example AMR2 configuration

The following example demonstrates the IPM REMOTE configuration for an endpoint that includes the following AMR2 related attributes in SDP.

```
a=fmtp:96 mode-set=0,2,4,7; mode-change-period=2, \
  mode-change-neighbor=1; mode-change-capability=2
```

Example code

```
...
/* Setup IP address here */
// Remote Audio Coder
ipmMediaInfo.MediaData[unCount].eMediaType = MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
ipmMediaInfo.MediaData[unCount].mediaInfo.AudioCoderInfo.eCoderType =
CODER_TYPE_AMRNB_12_2k;
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.unRedPayloType = 0;
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_RULE(CODER_OPT_AMR_CMR_TRACK) |
                    CODER_OPT_AMR_PACKING_MODE(CODER_OPT_AMR_EFFICIENT) |
                    CODER_OPT_AMR_MODE_CHANGE_NEIGHBOR(1) |
                    CODER_OPT_AMR_MODE_CHANGE_PERIOD(2) |
                    CODER_OPT_AMR_MODE_SET(0x95);
unCount++
 ipmMediaInfo.unCount = unCount;
```

For more information on IP Media codec configuration, refer to the *Dialogic® IP Media Library API Programming Guide and Library Reference.*

Transmit RFC 2833/RFC 4733 Tone Events

This feature allows an application to transmit a sequence of both DTMF and non-DTMF telephony events over an IP network by calling ipm_SendTelephonySignals( ). This can be used in generating all RFC 4733 tone event definitions (0-255) beyond the initial set of DTMF telephony events (0-15) used to represent digits 0-9, A-D, *, #. This can also be used to generate a non-DTMF telephony event, such as a Hookflash event, and DTMF RFC 2833/RFC 4733 RTP telephony events based on WebRTC signaling events in a WebRTC Gateway application. The feature also allows an application to support the modem and text tone event definitions specified in RFC 4734 (https://tools.ietf.org/html/rfc4734), or channel oriented signaling tone events specified in RFC 5244 (https://tools.ietf.org/html/rfc5244).

The RFC 4733 (https://tools.ietf.org/html/rfc4733) recommendation specifies the "RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals" and obsoletes the original RFC 2833 specification. The send/receive RFC 4733 tone event capability is integrated into the standard DTMF digit generation and detection API when RFC 2833/RFC 4733 mode is negotiated upon SDP media session establishment.

Note: ipm_SendTelephonySignals() is only available when audio is encoded. It's not available for native audio.

New DTMF Transfer Mode

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.

DTMFXFERMODE_RFC2833_APP

This transfer mode is the same as DTMFXERMODE_RFC2833 except that inband tones are not converted into RFC 2833 DTMF events by the IPM transmitter. The tones are still clamped. In this mode, telephony events are generated exclusively by the application when ipm_SendTelephonySignals() is called.

Function Information

Name: int ipm_SendTelephonySignals (nDeviceHandle, *pDigitInfo, usMode)

Inputs:

int nDeviceHandle	• IP Media device handle IPM_TELEPHONY_SEQUENCE_INFO
*pInfo	• pointer to 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

Description

The ipm_SendTelephonySignals( ) function instructs the IPM device to generate a sequence of RFC 2833/RFC 4733 telephony events over an IP network. The on/off time and volume of each telephony event is configurable.

The transfer mode must be set to DTMFXERMODE_RFC2833 or DTMFXERMODE_RFC2833_APP for the telephony events to be transmitted on the network. Refer to the ipm_SetParm( ) for more information.

Parameter	Description
nDeviceHandle	handle of the IP Media device
pInfo	pointer to the IPM_TELEPHONY_SEQUENCE_INFO structure
usMode	operation mode. Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

Termination Events

IPMEV_SEND_TELEPHONY_SIGNALS

Indicates successful completion. The given telephony event sequence has been transmitted to the remote endpoint. If ipm_Stop( ) is called while a sequence is being generated,

generation is stopped immediately and the IPMEV_SEND_TELEPHONY_SIGNALS termination event is generated, followed by the IPMEV_STOP termination event.

IPMEV_SEND_TELEPHONY_SIGNALS_FAIL

Indicates that the function failed. See the "Errors" section below for a list of error codes.

Cautions

ipm_SendTelephonySignals() is only available when audio is encoded. It's not available for native audio.

When the transfer mode is set to DTMFXFERMODE_RFC2833, inband tones that are converted to telephony events will conflict with telephony events that are generated by the application at the same time.

The DTMFXFERMODE_RFC2833_APP mode disables telephony event generation from inband tones on the transmit side. While in DTMFXFERMODE_RFC2833_APP mode, an application can detect inbound tones or telephony events using a DX device or detect inbound telephony events using IPM telephony event reporting. The detected tones/events can be regenerated using ipm_SendTelephonySignals( ).

Errors

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

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.

If the IPMEV_SEND_TELEPHONY_SIGNALS_FAIL termination event is received, call ATDV_LASTERR( ) and ATDV_ERRMSGP( ) to return one of the following errors:

EIPM_INVALID_EVENT_ID

The event ID is not in the range 0 through 255.

EIPM_INVALID_VOLUME

The volume is not in the range 0 through 63.

EIPM_INVALID_OPTIONS

Options value is wrong.

EIPM_INVALID_SIGNAL_TYPE

Unrecognized signal type. The signal type must be "event".

EIPM_PAYLOAD_TYPE_NOT_IMPLEMENTED

Per sequence/per event RTP payload type is not implemented. The telephony event payload type must be set by the PARMCH_RFC2833EVT_TX_PLT to ipm_SetParm( ).

EIPM_PTIME_NOT_IMPLEMENTED

Per sequence/per event ptime is not implemented. The frame time of the codec selected in ipm_StartMedia( ) is always used.

EIPM_CLOCK_RATE_NOT_IMPLEMENTED

Per sequence/per event clock rate is not implemented. The RTP clock rate of the codec selected in ipm_StartMedia( ) is always used.

EIPM_TONE_NOT_IMPLEMENTED

This function can only generate telephony events. Telephony tones can not be generated by this function.

EIPM_OUT_OF_RANGE

An event ID in the "telephony event ID string" is too large. The maximum value is 255.

EIPM_TOO_MANY_DIGITS

An event ID in the "telephony event ID string" contains too many digits. The maximum number of digits is 3.

EIPM_INVALID_CHARACTER

The "telephony event ID string" contains an invalid character. Valid characters are comma and decimal digits.

Example 1

In this example, the "telephony event ID string" is used to generate the sequence.

```
void SendTelephonySignalsExample1(int handle)
{
     IPM_TELEPHONY_SEQUENCE_INFO seq;
```

```
    //    The second parameter to the INIT function is set to zero since it
    doesn't apply

    //    to strEventIDs. It's only used for the signal array.

    INIT_IPM_TELEPHONY_SEQUENCE_INFO(&seq, 0);

    // These apply to each event in the event ID string.

    seq.sVolume = 7; // dBm0/sign dropped as defined in RFC 4733 seq.usDuration =
    200; // milliseconds

    //    This is the duration of the gap between events in the event ID string.
    seq.unInterval = 100; // milliseconds

    //    The event ID string is a comma separated list of telephony event IDs.
    The following

    //    list includes DTMF 1,2,3 and hook flash.

    seq.strEventIDs = "1,2,3,16";

    if( ipm_SendTelephonySignals(handle, &seq, EV_ASYNC) == -1 )

    {

    printf("ipm_SendTelephonySignals() failed, %s (%ld)\n", ATDV_ERRMSGP(handle),
    ATDV_LASTERR(handle));

    }

    FREE_IPM_TELEPHONY_SEQUENCE_INFO(&seq); // this must be called
}
```

Example 2

In this example, the “telephony signal array” is used to generate the sequence. The volume and duration can be configured for each event.

```
void SendTelephonySignalsExample2(int handle)

{

    IPM_TELEPHONY_SEQUENCE_INFO seq;

    IPM_TELEPHONY_EVENT_INFO      pEventInfo;

    //    The "count" parameter is set to 3 since there are 3 elements in the
    telephony

    //    event array defined below.

    INIT_IPM_TELEPHONY_SEQUENCE_INFO(&seq, 3);

    // These are the default values that will be used for the event array
    elements.

    seq.sVolume = 7;

    seq.usDuration    = 200;

    seq.unInterval    = 100;

    //    The volume and duration aren't set in this element, so the default
    values

    //    above are used.
```

```
        pEventInfo = INIT_IPM_TELEPHONY_EVENT_INFO(&seq);

        pEventInfo->eTelephonyEventID = SIGNAL_ID_EVENT_DTMF_0;

        //    The volume and duration set on the next two events override the volume,
        duration

        //    and interval values set above.

        pEventInfo = INIT_IPM_TELEPHONY_EVENT_INFO(&seq);

        pEventInfo->eTelephonyEventID = SIGNAL_ID_OFF; // pseudo event ID to insert a
        gap between events

        pEventInfo->usDuration  = 150;

        pEventInfo = INIT_IPM_TELEPHONY_EVENT_INFO(&seq);

        pEventInfo->eTelephonyEventID = SIGNAL_ID_EVENT_DTMF_1;

        pEventInfo->sVolume     = 5;

        pEventInfo->usDuration = 185; // this is rounded up to the next frame period,
        e.g. 200 milliseconds for 20 millisecond G.711

        if( ipm_SendTelephonySignals(handle_, &seq, EV_ASYNC) == -1 )

        {

        printf("ipm_SendTelephonySignals() failed, %s (%ld)\n", ATDV_ERRMSGP(handle_),
        ATDV_LASTERR(handle_));

        }

        FREE_IPM_TELEPHONY_SEQUENCE_INFO(&seq); // this must be called
    }
```

For more information on IP Media API functionality, refer to the Dialogic® IP Media Library API Programming Guide and Library Reference.

Secure RTP With First Party Call Control

Secure RTP (http://www.ietf.org/rfc/rfc3711.txt) is a method that allows for secure encrypted transmission of RTP data between endpoints. Secure RTP functionality has been previously supported with HMP when using third party call control (3PCC) mode. In 3PCC mode, the application is responsible for selecting the encryption method, key generation, negotiation, and state transitions between the endpoints. This HMP release adds functionality for supporting Secure RTP when using first party call control (1PCC) configuration. In 1PCC mode, these steps are managed within the HMP GlobalCall libraries. Secure RTP can be used in conjunction with SIP TLS (https://www.rfc-editor.org/rfc/rfc5246.txt) to provide a secure method for two endpoints using SRTP to exchange the necessary setup information, including SRTP keys.

Supported crypto suites

HMP supports the following crypto suites through 1PCC:

- AES_CM_128_HMAC_SHA1_80
- AES_CM_128_HMAC_SHA1_32
- AES_CM_256_HMAC_SHA1_80
- AES_CM_256_HMAC_SHA1_32

Enabling 1PCC SRTP

The feature is enabled by including a GC_PARM_BLK with parameter set ID GCSET_CHAN_CAPABILITY and parameter ID of IPPARM_ENABLE_SRTP_1PCC. Setting the value to IP_DISABLE disables Secure RTP (default), setting to IP_ENABLE enables Secure RTP.

The HMP license on the target system must include Encryption (SRTP / TLS) in the license configuration to enable support for this feature. An "IPERR_BAD_PARAM" is returned when enabling the feature if Encryption is not included in the HMP license.

Example code

…

```
/* Enable Secure RTP */
gc_util_insert_parm_val(&gcParmBlk, GCSET_CHAN_CAPABILITY,IPPARM_ENABLE_SRTP_1PCC,
 sizeof(long), IP_ENABLE);
if (gc_SetUserInfo(GCTGT_GCLIB_CHAN, lineDev , gcParmBlk, GC_ALLCALLS) < 0) {
   printf("Error: gc_SetUserInfo() returned error enabling Secure RTP\n");
}
```

Outbound calls

When the SRTP feature is enabled, the GlobalCall library will generate a new Master key for each call. The crypto information is included in the SDP offer provided in the SIP INVITE message. The remote side responds with its crypto information in its response. If the negotiation is successful, RTP in both directions will be encrypted. If the remote side does not provide its crypto information in its response, the call will be rejected per RFC4568, section 7.1.2 (https://www.ietf.org/rfc/rfc4568.txt)

The table below shows documents HMP behavior during outbound call negotiation.

Local	Remote response	Results
SRTP Disabled	SDP without SRTP	Invoke regular RTP call handling
SRTP Enabled	SDP with SRTP	Invoke Secure RTP call handling
SRTP Disabled	SDP with SRTP	If remote responds with SRTP crypto, then local side will CANCEL the call and return GCEV_DISCONNECTED event sent to application with reason IPEC_InternalReasonSRTPCryptoMismatch.
SRTP Enabled	SDP without SRTP	Per RFC, if the remote does not support SRTP then it should reject the call. If it accepts without sending SRTP crypto back then local side will CANCEL the call and return GCEV_DISCONNECTED event sent to application with reason IPEC_InternalReasonSRTPCryptoMismatch.

Receiving calls

When the SRTP feature is enabled, the GlobalCall library will generate a new Master key for each call. When a new incoming call is received containing crypto information in the SDP, GlobalCall will respond with its crypto information in the SDP answer provided in the SIP response. If a call is received that does not include crypto information, GlobalCall will not include crypto in its response. The resulting call will not utilize Secure RTP.

The table below shows documents HMP behavior during inbound call negotiation.

Local	Remote sends	Results
SRTP Disabled	Invite without SRTP	Invoke regular RTP call handling
SRTP Enabled	Invite with SRTP	Invoke Secure RTP call handling
SRTP Disabled	Invite with SRTP	Send 488 "Not Acceptable Here" response to remote and return GCEV_DISCONNECTED event sent to application.
SRTP Enabled	Invite without SRTP	Invoke regular RTP call handling (no SRTP)

Feature notes

- While HMP IP Media Library implementation supports the ability to specify multiple encryption keys, 1PCC Secure RTP feature utilizes a single key per call. Multiple key rotation is not supported.
- For more information related to Secure RTP, see Chapter 21 of the "Dialogic® IP Media Library API Programming Guide and Library Reference" https://www.dialogic.com/-/media/manuals/docs/ip_media_api_hmp_v16.pdf
- For more information related to SIP TLS, see Chapter 4, Section 29 of the "Dialogic® Global Call IP Technology Guide" https://www.dialogic.com/-/media/manuals/docs/globalcall_for_ip_hmp_v12.pdf

IPv6 Call Control

Support for IPv6 call control has been enabled for HMP Windows. The feature is enabled through the IP_VIRTBOARD structure.

IP_VIRTBOARD

- E_SIP_IPv6 - Enables the application to use IPv6. The default is disabled.
- localIPv6 - Specifies the local IPv6 address to be used in SIP signaling.
- localIPv6_iface_name - For Link-Local IPv6 address only. Specifies the network interface to use when sending IPv6 packets. The value can be an interface name or a scope identifier string value.

The IPPARM_SDP_IP_TYPE parameter in the IPSET_SDP parameter set is used to specify the SDP address type (IPv4 or IPv6 for RTP/RTCP addresses) in the SIP SDP offer/answer model. The default value is IPv4 addressing for backward compatibility. Use gc_SetUserInfo( ) to specify arguments for a single call (GC_SINGLECALL) or for all calls

(GC_ALLCALLS) on a line device. The gc_SetConfigData( ) function is not used with this parameter.

IPPARM_SDP_IP_TYPE

Specifies the IP address type to use in SDP:

- USE_IPv4 – (Default) Only IPv4 addressing is accepted in incoming/outgoing SDP.
- USE_IPv6 – Only IPv6 addressing is accepted in incoming SDP.
- PREFER_IPv6 – IPv6 addressing is used when sending an SDP offer. When receiving an SDP offer based on IPv4, the SIP stack will use IPv4 SDP for that connection.

Multitrack Audio Wave Recording Feature

The multitrack record feature enables applications to record two separate audio sources into a dual-track (stereo) wave file. This feature can be utilized by call centers, E911 applications, banking applications, and monitoring applications to record two audio callers, such as agent and client, as different tracks rather than recording the mixed output of an audio conference. An additional use case of this feature enables applications to capture an audio recording of the HMP system input and output of the caller (i.e., what the caller hears and what the caller says) in a single dual-track (stereo) .wav file.

The two main use cases supported by HMP for the multitrack record feature in this release are individual party multitrack transaction recording and two-party multitrack recording, which are described in the following sections.

Individual Party Multitrack Recording

The individual multitrack transaction recording use case enables applications to record the audio of the caller speaking and the audio that the caller hears in the same file as two different tracks.

This feature provides the ability to record the system output sent to a user without the need to do packet capture on the network to get the audio as it is heard by the caller. The recording of what a caller hears includes all of the different sources that occur during a call, such as audio from another caller, output of a conference, or output from a play file. This provides the ability to record the audio a caller hears without the need to put all sources through a conference mixer.

Two-Party Multitrack Recording

The two-party recording use case enables applications to record two sources, such as two call parties, as two separate tracks in a single .wav file. The resulting file has each audio source in a separate track, which can be played back together or separated by source.

Providing recordings as multitrack recordings has unique advantages over single mixed audio recordings. A dual-track (stereo) .wav file can be played back on standard players as a stereo file with synchronized audio between the two parties. Additionally, a multitrack file also allows the audio of each individual participant track to be easily separated. Separating the audio allows post processing of the individual caller's audio that may not be possible with a mixed conference output where voices cannot easily be separated.

For example, individual tracks can be sent to speech analytics software to get an accurate per participant transcript or to analyze the speech characteristics of a caller or agent.

MM_MEDIA_AUDIO Updates

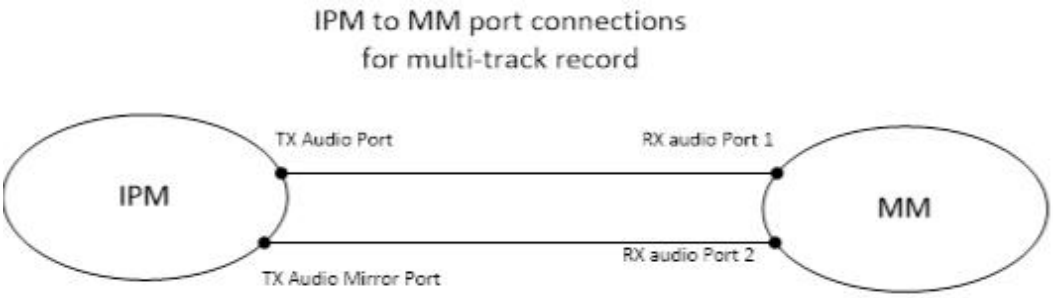
Updates are needed to the MM_MEDIA_AUDIO structure that is passed into the mm_Record( ) function to start the recording. The following is an example.

```
INIT_MM_MEDIA_ITEM_LIST(&m_audioMediaList);

INIT_MM_MEDIA_AUDIO(&m_audioMediaList.item.audio);

m_audioMediaList.ItemChain          = EMM_ITEM_EOT;

m_audioMediaList.item.audio.codec= m_mmAudioCodecRecord;

m_audioMediaList.item.audio.unMode = MM_MODE_AUD_BEEPINITIATED |
MM_MODE_MULTI_TRACK_RECORD; m_audioMediaList.item.audio.unOffset = 0;

m_audioMediaList.item.audio.szFileName = m_audioRecordFileName.c_str();
m_audioMediaList.item.audio.unAccessMode = MM_MEDIA_ACCESS_MODE_FILE;
m_audioMediaList.item.audio.unNumTracks = 2
```

When using ORing in MM_MODE_MULTI_TRACK_RECORD in the unMode field and setting the umNumTracks field to a value of 2, a 2 track .wav file will be recorded.

Port Connections

With a single track recording, there is one port connected to the MM device. When executing a multitrack recording, there are 2 ports that are connected to the MM device. The following figure shows a use case in which both the transmit and receive streams of an IPM device will be recorded to a multitrack .wav file.



This use case makes use of "mirrored" port that mirrors the data coming into the RX port of an IPM device. This RX data is what will be sent out over the IP network. The "tx mirror" port mirrors the data from the RX side back into a TX port for transmission to other HMP components.

TX Mirror Port

The "tx mirror" port is represented in the device management API using the port type of DM_PORT_MEDIA_TYPE_AUDIO_MIRROR. A port of this type will be returned in the event data provided by a call to dev_GetTransmitPorts( ) on an IPM device. This port will be provided in addition to the existing port of type DM_PORT_MEDIA_TYPE_AUDIO.

Connecting Two TX Ports to MM

In addition to adding the "tx mirror" port to IPM, an additional port was added to MM to support a second transmitter to be recorded. The second RX port is accessed by ORing in the value of DMFL_TRANSCODE_USE_SECOND_PORT in the unFlags field. The following is

sample code to show how to retrieve the “tx mirror” and connect both the TX audio port and the TX mirror audio port to the MM device.

```
#include <srllib.h>
#include <ipmlib.h>
#include <mmlib.h>
#include <port_connect.h>
#include <string.h>
#include <iostream>
using namespace std;
int main(int argc, char** argv)
{
	int rc;
	int ipmDev,mmDev;
	long evt;
	void* evt_data;
	const char szDev1[] = "ipmB1C1";
	const char szDev2[] = "mmB1C1";
	unsigned int index = 0;
	ipmDev = -1;
	mmDev = -1;
	try
	{
	// Open device (ipm)
	ipmDev = ipm_Open(szDev1, NULL, EV_ASYNC);
	if (-1 == ipmDev)
	{
		cout << "ipm_Open error";
		cout << " handle = " << ipmDev << endl;
		throw 1;
	}
	sr_waitevt(-1);
	evt = sr_getevttype();
	if (IPMEV_OPEN != evt)
	{
		cout << "ipm_Open error";
		cout << " event = " << evt << endl;
		throw 2;
	}
	cout << "ipm_Open() completed successfully" << endl;
	// Open device (mm)
	mmDev = mm_Open(szDev2, NULL, NULL);
```

```
if (-1 == mmDev)
{
       cout << "mm_Open error";
       cout << " handle = " << mmDev << endl;
       throw 3;
}
sr_waitevt(-1);
evt = sr_getevttype();
if (MMEV_OPEN != evt)
{
       cout << "mm_Open error";
       cout << " event = " << evt << endl;
       throw 4;
}
cout << "mm_Open() completed successfully" << endl;
//****** get MM RX ports ******
rc = dev_GetReceivePortInfo(mmDev, NULL);
if (-1 == rc)
{
       cout << "dev_GetReceivePortInfo error";
       cout << " rc = " << rc << endl;
       throw 5;
}
sr_waitevt(-1);
evt = sr_getevttype();
if (DMEV_GET_RX_PORT_INFO != evt)
{
       cout << "dev_GetReceivePortInfo error";
       cout << " event = " << evt << endl;
       throw 4;
}
evt_data = sr_getevtdatap();
int evt_len = sr_getevtlen();
DM_PORT_INFO_LIST mmRxPortList = {};
memcpy(&mmRxPortList, evt_data, evt_len);
cout << "Number of RX ports: " << mmRxPortList.unCount << endl;
DM_PORT_INFO *pmmRxAudioPort = NULL;
for (index = 0; index < mmRxPortList.unCount; index++)
{
       cout << "port type at index " << index << ": "
       << mmRxPortList.port_info[index].port_media_type << endl;
       if (mmRxPortList.port_info[index].port_media_type ==
       DM_PORT_MEDIA_TYPE_AUDIO)
```

```
{
cout << "RX port index " << index << " is of type DM_PORT_MEDIA_TYPE_AUDIO" <<
endl;
pmmRxAudioPort = &mmRxPortList.port_info[index];
}
}
//****** get IPM TX ports ******
rc = dev_GetTransmitPortInfo(ipmDev, NULL);
if (-1 == rc)
{
       cout << "dev_GetTransmitPortInfo error";
       cout << " rc = " << rc << endl;
       throw 5;
}
sr_waitevt(-1);
evt = sr_getevttype();
if (DMEV_GET_TX_PORT_INFO != evt)
{
       cout << "dev_GetTransmitPortInfo error";
       cout << " event = " << evt << endl;
       throw 4;
}
evt_data = sr_getevtdatap();
evt_len = sr_getevtlen();
DM_PORT_INFO_LIST ipmTxPortList = {};
memcpy(&ipmTxPortList, evt_data, evt_len);
cout << "Number of TX ports: " << ipmTxPortList.unCount << endl;
DM_PORT_INFO *pipmTxAudioMirrorPort = NULL;
DM_PORT_INFO *pipmTxAudioPort = NULL;
for (index = 0; index < ipmTxPortList.unCount; index++)
{
       if (ipmTxPortList.port_info[index].port_media_type ==
       DM_PORT_MEDIA_TYPE_AUDIO_MIRROR)
{
cout << "TX port index " << index << " is of
type DM_PORT_MEDIA_TYPE_AUDIO_MIRROR" << endl;
pipmTxAudioMirrorPort = &ipmTxPortList.port_info[index];
}
if (ipmTxPortList.port_info[index].port_media_type == DM_PORT_MEDIA_TYPE_AUDIO)
{
       cout << "TX port index " << index << " is of type
       DM_PORT_MEDIA_TYPE_AUDIO" <<
       endl;
```

```
        pipmTxAudioPort = &ipmTxPortList.port_info[index];
}
}
//***** Connect MM receive and IPM transmit ports (audio and audio
mirror ****************
// the IPM device has to transmit ports. The newly-added port is the called
that
"rx mirror" port
// because it takes that audio that is transmitted out to the IP network and
"mirrors"
back into
// HMP. This allows the outgoing data to be recorded. The rx mirror port has a
media
type of
// DM_PORT_MEDIA_TYPE_AUDIO_MIRROR.
//
// to connect the rx mirror port to the MM device, a new flag bit has been
defined.
// DMFL_TRANSCODE_USE_SECOND_PORT allows a second transmit port to be connected
to an MM
device.
// To use the 2nd port in MM, OR in the DMFL_TRANSCODE_USE_SECOND_PORT flag and
set the
tx port
// to the rx mirror port retrieved with dev_GetTransmitPorts().
DM_PORT_CONNECT_INFO_LIST ConnList;
INIT_DM_PORT_CONNECT_INFO_LIST(&ConnList);
unsigned int count=0;
/* set up MM audio tx to IPM port connections with transcoding enabled
*/
ConnList.port_connect_info[count].unFlags = DMFL_TRANSCODE_ON |
DMFL_TRANSCODE_USE_SECOND_PORT;
ConnList.port_connect_info[count].port_info_tx = *pipmTxAudioMirrorPort;
ConnList.port_connect_info[count].port_info_rx = *pmmRxAudioPort;
count++;
ConnList.port_connect_info[count].unFlags = DMFL_TRANSCODE_ON;
ConnList.port_connect_info[count].port_info_tx =
*pipmTxAudioPort; ConnList.port_connect_info[count].port_info_rx
= *pmmRxAudioPort; count++;
ConnList.unCount = count;
rc = dev_PortConnect(ipmDev, &ConnList, NULL);
if (-1 == rc)
{
        cout << "dev_PortConnect error";
```

```
		cout << " rc = " << rc << endl;
		throw 51;
	}
	sr_waitevt(-1);
	evt = sr_getevttype();
	if (DMEV_PORT_CONNECT != evt)
	{
		cout << "dev_PortConnect error";
		cout << " event = " << evt << endl;
		throw 52;
	}
	cout << "dev_PortConnect completed sucessfully" << endl;
	// disconnect the ports
	rc = dev_PortDisconnect(ipmDev, &ConnList, NULL);
	if (-1 == rc)
	{
		cout << "dev_PortDisconnect error";
		cout << " rc = " << rc << endl;
		throw 51;
	}
	sr_waitevt(-1);
	evt = sr_getevttype();
	if (DMEV_PORT_DISCONNECT != evt)
	{
		cout << "dev_PortDisconnect error";
		cout << " event = " << evt << endl;
		throw 52;
	}
	cout << "dev_PortDisconnect completed sucessfully" << endl;
	}
	catch(int point)
	{
		cout << "error at point: " << point << endl;
		exit(-1);
	}
}
```