Dialogic
Making Innovation Thrive™

Dialogic® Device Management API

Library Reference

October 2009

05-2222-010

Publication Date: October 2009

Document Number: 05-2222-010

Contents

Revision History

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

Document No.	Publication Date	Description of Revisions
05-2222-010	October 2009	dev_Connect( ) function: Indicated that M3G connections are not supported in Dialogic® HMP Software 3.0WIN. dev_GetResultInfo( ) function: Indicated that this function is not supported in Dialogic® HMP Software 3.0WIN. dev_GetResourceReservationInfoEx( ) function: Indicated that this function is not supported in Dialogic® HMP Software 3.0WIN. dev_PortConnect( ) function: Added Dialogic® HMP Software 4.1LIN to Connection Type table in Supported Connections section. dev_ReleaseResourceEx( ) function: Indicated that this function is not supported in Dialogic® HMP Software 3.0WIN. dev_ReserveResourceEx( ) function: Indicated that this function is not supported in Dialogic® HMP Software 3.0WIN.
05-2222-009	November 2008	Function Summary by Category chapter: Emphasized that Ex functions should be used instead of non-Ex functions in Resource Reservation Functions. Added Error Processing Functions (inadvertently omitted in previous version). Function Information chapter: Removed Platform line from function syntax table. In general, functions are supported across platforms, with a few exceptions. dev_Connect( ) function: In Description, added a paragraph to clarify usage. In Asynchronous Operation, replaced dev_ErrorInfo( ) with dev_GetResultInfo( ) for processing failure events. In Supported Connections, updated to show that both synchronous and asynchronous modes are supported for all connections. In Supported Connections, added “M3G Audio Device and CNF”, “M3G Audio Device and Voice / DTI,” and “M3G Control Device and DTI.” [IPY00044585] dev_Disconnect( ) function: In Asynchronous Operation, replaced dev_ErrorInfo( ) with dev_GetResultInfo( ) for processing failure events. In Errors, removed clause about failure event. [IPY00043214] dev_GetResultInfo( ) function: Minor edits in Description and Caution. dev_GetResourceReservationInfoEx( ) function: In Asynchronous Operation, replaced dev_ErrorInfo( ) with dev_GetResultInfo( ) for processing failure events. In Errors, removed clause about failure event from introductory paragraph. Updated asynchronous example code to remove ATDV_LASTERR( ). dev_GetReceivePortInfo( ), dev_GetTransmitPortInfo( ), dev_PortConnect( ) and dev_PortDisconnect( ) functions: In Asynchronous Operation, replaced SRL functions with dev_GetResultInfo( ) for processing failure events. In Errors, removed clause about failure event and SRL functions from introductory paragraph, and replaced with dev_ErrorInfo( ). dev_PortConnect( ) function: Updated to clarify that this function creates half-duplex connections between “internal” ports of the specified device. Added M3G and CNF devices to Multimedia Scenario section and added note about external/internal ports and connections. Added Supported Connections section. [IPY00078800]

Document No.	Publication Date	Description of Revisions
05-2222-009 (cont.)	November 2008	dev_PortDisconnect( ) function: In Description, clarified that these are “internal” ports. dev_ReleaseResourceEx( ), dev_ReserveResourceEx( ) functions: In Asynchronous Operation, replaced dev_ErrorInfo( ) with dev_GetResultInfo( ) for processing failure events. In Errors, removed clause about failure event. Events chapter: Added information about using dev_GetResultInfo( ) for processing failure events. DM_PORT_INFO_LIST structure: Added INIT_DM_PORT_INFO_LIST inline function and updated unVersion field description. Error Codes chapter: Removed details about failure events as this information belongs in the Events chapter.
05-2222-008	June 2008	Function Summary by Category chapter: Added a new Event Handling category. dev_PortConnect( ) function: Updated example code to add a comment about checking transcoding support for video. Also added an if statement for DM_PORT_MEDIA_TYPE_VIDEO. dev_GetResultInfo( ) function: Added this new function. DM_EVENT_INFO structure: Added this new data structure. DM_PORT_CONNECT_INFO structure: Added notes to the unFlags field of this data structure. DM_PORT_INFO structure: Added a note that the DM_PORT_MEDIA_TYPE_NBUP value is deprecated.
05-2222-007	August 2007	Made global changes to reflect Dialogic brand. Function Summary by Category chapter: Removed table of function support by platform section. dev_GetResourceReservationInfoEx() function: Updated Platform line to show support for Dialogic® HMP software; added note in Description section. dev_ReserveResourceEx() function: Updated Platform line to show support for Dialogic® HMP software; added note in Description section. dev_ReleaseResourceEx() function: Updated Platform line to show support for Dialogic® HMP software; added note in Description section.

Document No.	Publication Date	Description of Revisions
05-2222-006	May 2007	Function Summary by Category chapter: Added new functions to Device Connection Functions. Added table of function support by platform. Function Information chapter: Added the following new Device Connection functions: dev_GetReceivePortInfo(), dev_GetTransmitPortInfo(), dev_PortConnect() and dev_PortDisconnect(). dev_Connect() function: Updated example code to include inline functions INIT_MM_AUDIO_CODEC, INIT_MM_VIDEO_CODEC, INIT_MM_PLAY_RECORD_LIST, INIT_MM_PLAY_INFO. dev_GetResourceReservationInfo() function: Updated example code to include inline function INIT_DEV_RESOURCE_RESERVATIONINFO and other edits. dev_GetResourceReservationInfoEx() function: Updated example code to include inline function INIT_DEV_RESOURCE_RESERVATIONINFO_EX and other edits. dev_PortConnect() function: Updated example code to include inline function INIT_DM_PORT_CONNECT_INFO_LIST. dev_ReleaseResourceEx( ) function: Updated example code to include inline function INIT_DEV_RESOURCE_LIST. dev_ReserveResourceEx( ) function: Added caution about cleaning up resources before exiting. Updated example code to include inline function INIT_DEV_RESOURCE_LIST. Events chapter: Added eight new events associated with the new Device Connection functions. Data Structures chapter: Added the following new data structures: DM_CONNECT_STATUS_LIST, DM_PORT_CONNECT_INFO, DM_PORT_CONNECT_INFO_LIST, DM_PORT_INFO, DM_PORT_INFO_LIST. DEV_RESOURCE_LIST structure: Added INIT_DEV_RESOURCE_LIST inline function. Changed version data type from ‘int’ to ‘unsigned int’. DEV_RESOURCE_RESERVATIONINFO structure: Added INIT_DEV_RESOURCE_RESERVATIONINFO inline function. Changed version data type from ‘int’ to ‘unsigned int’. DEV_RESOURCE_RESERVATIONINFO_EX structure: Added INIT_DEV_RESOURCE_RESERVATIONINFO_EX inline function. Changed version data type from ‘int’ to ‘unsigned int’.

Document No.	Publication Date	Description of Revisions
05-2222-005	September 2006	Global change: Revisions included adding new Dialogic® Multimedia Platform for AdvancedTCA references, function operations, data structures, and events. Purpose section: Updated the description of the API to include Dialogic® Multimedia Platform for AdvancedTCA. Function Summary by Category chapter: Specified which Resource Reservation Functions are on Dialogic® HMP software and Dialogic® Multimedia Platform for AdvancedTCA. Function Information chapter: Added three new functions: dev_GetResourceReservationEx( ), dev_ReleaseResourceEx( ), dev_ReserveResourceEx( ). Added Dialogic® Multimedia Platform for AdvancedTCA to dev_Connect( ), dev_Disconnect( ), and dev_ErrorInfo( ) functions. Data Structures chapter: Added three new data structures: DEV_RESOURCE_LIST, DEV_RESOURCE_RESERVATIONINFO_EX, and resourceInfo. Events chapter: Added four new Resource Reservation Events: DMEV_RELEASE_RESOURCE, DMEV_RELEASE_RESOURCE _FAIL, DMEV_RESERVE_RESOURCE, and DMEV_RESERVE_RESOURCE_FAIL. Added dev_GetResourceReservationInfoEx( ) function to existing events DMEV_GET_RESOURCE_RESERVATIONINFO and DMEV_GET_RESOURCE_RESERVATIONINFO_FAIL.
05-2222-004	August 2006	dev_Connect( ) function: Added new connection types to the section on Supported Connections.
05-2222-003	August 2005	Added multimedia features. updated some function operations, and made a few corrections. Purpose section: Updated the description of the API to include ability to connect IP media and multimedia devices. dev_Connect( ) function: Added section on Supported Connections. Removed section on Implicit Disconnection (as well as corresponding caution) as not applicable. Changed Cautions section to indicate that multiple connections are not possible. Added Multimedia Sample and Example A (Multimedia Asynchronous). Corrected the T.38 Sample, which referred to the IPML define MEDIATYPE_LOCAL_T38_INFO instead of MEDIATYPE_LOCAL_UDPTL_T38_INFO. dev_Disconnect( ) function: Changed Cautions section to indicate that disconnecting a device that is not connected generates an error now, rather than being ignored, as occurred previously. Added cross reference to dev_Connect( ) example code. Replaced the T.38 Sample with a cross reference to identical sample in dev_Connect( ).

Document No.	Publication Date	Description of Revisions
05-2222-002	September 2004	dev_ReleaseResource() and dev_ReserveResource() functions: Corrected function header, description, operation, cautions, and example code to indicate that the Resource Reservation operations on the Low Bit Rate codec (resource type **RESOURCE_IPM_LBR**) are supported in synchronous mode only (asynchronous mode is not supported). Resource Reservation Events: Removed the following Resource Reservation events because asynchronous mode is not supported for the Resource Reservation functions: DMEV_RELEASE_RESOURCE DMEV_RELEASE_RESOURCE_FAIL DMEV_RESERVE_RESOURCE DMEV_RESERVE_RESOURCE_FAIL dev_ReleaseResource() function: Reworded caution to say that the function requires the device to be open or else it generates an EIPM_INV_STATE error (deleted “and that it have a resource of the specified type reserved for it”).
05-2222-001	September 2003	Initial version of document.

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 contains reference information for functions, parameters, data structures, values, events, and error codes in the Dialogic® Device Management API. The API provides run-time control and management of configurable system devices, including functions to reserve resources and to manage connections between devices for communication and sharing of resources.

Applicability

This document is published for the following releases: Dialogic® Host Media Processing Software Release 4.1LIN (Dialogic® HMP Software 4.1LIN), Dialogic® Host Media Processing Software Release 3.1LIN (Dialogic® HMP Software 3.1LIN), Dialogic® Host Media Processing Software Release 3.0WIN (Dialogic® HMP Software 3.0WIN), Dialogic® Multimedia Platform for ATCA (MMP for ATCA), and Dialogic® Multimedia Kit for PCIe (MMK for PCIe). The information in this document applies to all releases unless indicated otherwise.

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 information is intended for:

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

- Original Equipment Manufacturers (OEMs)
- End Users

How to Use This Publication

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

The information in this publication is organized as follows:

- Chapter 1, "Function Summary by Category" introduces the categories of functions and provides a brief description of each function.
- Chapter 2, "Function Information" provides an alphabetical reference to all the functions in the Dialogic® Device Management API library.
- Chapter 3, "Events" describes the events that are generated by the Dialogic® Device Management API functions.
- Chapter 4, "Data Structures" describes the data structures used with Dialogic® Device Management API functions, including fields and valid values.
- Chapter 5, "Error Codes" presents a listing of error codes that are returned by the API.

Related Information

See the following for additional information:

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

Function Summary by Category 1

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

1.1 Dialogic® Device Management API Header File

The Dialogic® Device Management API contains functions that provide run-time control and management of configurable system devices. The Dialogic® Device Management API functions, parameters, data structures, values, events, and error codes are mainly defined in the *devmgmt.h* header file. The Dialogic® Device Management API functions have a “dev_” prefix.

Note: The header file also contains other functions, such as those belonging to the Dialogic® Board Management Library, which have a “brd_” prefix. The Dialogic® Board Management Library functions and their associated data belong to a separate API category and are not addressed by this document. Their presence in the header file does not indicate that they are supported.

1.2 Device Connection Functions

Device Connection functions manage the connections between devices, allowing communication and sharing of resources. They include the following functions:

dev_Connect( )
Establishes either a half-duplex or a full-duplex connection for communication between the two specified channel devices.

dev_Disconnect( )
Disconnects or breaks the connection between the receive channel of the specified device and the transmit channel of the device that was associated with it.

dev_GetReceivePortInfo( )
Retrieves device receive port information.

dev_GetTransmitPortInfo( )
Retrieves device transmit port information.

dev_PortConnect( )
Establishes port to port connections.

dev_PortDisconnect( )
Disconnects or breaks the connection between ports.

1.3 Resource Reservation Functions

Resource Reservation functions manage configurable system devices at run time. They provide the ability to reserve low bit rate codecs (e.g., G.723 or G.729) for an IP media device on media processing software.

They include the following functions:

Note: The "**Ex( )**" functions supercede and should be used instead of the non-Ex( ) versions. The Ex( ) functions provide improved information about available resources.

dev_GetResourceReservationInfo( )
Provides the current reservation information for the specified resource (s) and device in a DEV_RESOURCE_RESERVATIONINFO data structure. Superceded by **dev_GetResourceReservationInfoEx( )**.

dev_GetResourceReservationInfoEx( )
Provides the current reservation information for the specified resource(s) and device in the DEV_RESOURCE_RESERVATIONINFO_EX data structure.

dev_ReleaseResource( )
Releases a specified resource previously reserved for the device. Superceded by **dev_ReleaseResourceEx( )**.

dev_ReleaseResourceEx( )
Releases specified resource(s) previously reserved for the device.

dev_ReserveResource( )
Reserves a resource for use by the specified device, such as reserving a low bit rate codec resource (e.g., G.723 or G.729) for an IP media device. Superceded by **dev_ReserveResourceEx( )**.

dev_ReserveResourceEx( )
Reserves resource(s) for use by the specified device, such as reserving a low bit rate codec resource (e.g., G.723 or G.729) for an IP media device.

1.4 Event Handling Functions

Event Handling functions provide event handling information. This category includes the following function:

dev_GetResultInfo( )
Gathers information concerning a given event. This event information may be used for trace logging, debugging, and error handling.

1.5 Error Processing Functions

Error Processing functions provide error processing information. They include the following functions:

dev_ErrorInfo( )
Obtains the error information for the last error in the Dialogic® Device Management API or one of its subsystems and provides it in the DEV_ERRINFO error information structure.

Function Information 2

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

2.1 Function Syntax Conventions

The Dialogic® Device Management API functions use the following format:

```
dev_FunctionName (DeviceHandle, Parameter1, Parameter2, ..., ParameterN, mode)
```

where:

dev_FunctionName
: represents the name of the function. Functions in the Dialogic® Device Management API use the prefix “dev_” in the function name.

DeviceHandle
: is an input parameter that specifies a valid handle obtained for a device when the device was opened

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

mode
: is an input parameter that specifies how the function should be executed, typically either asynchronously or synchronously. Some functions can be executed in only one mode and so do not provide this parameter.

dev_Connect( )

Name:	int dev_Connect (devHandle1, devHandle2, connType, mode)	
Inputs:	int devHandle1	• a valid channel device
	int devHandle2	• a valid channel device
	eCONN_TYPE connType	• type of connection to make between the devices
	unsigned short mode	• asynchronous or synchronous function mode
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h	
Category:	Device Connection	
Mode:	asynchronous or synchronous	

Description

The **dev_Connect( )** function establishes either a half-duplex or a full-duplex connection for communication between the two specified channel devices. If half-duplex communication is used, the first device listens to the second device (i.e., **devHandle1** listens to **devHandle2**). The connection remains until broken by **dev_Disconnect( )**.

Note: The terms *listen* and *receive* are used synonymously.

By default, the **dev_Connect( )** function connects audio linear ports and video native ports. In this case, "linear" means audio in straight PCM format. This is the lowest common denominator from which other forms of compressed audio can be derived using a codec. Thus, audio transcoding is achieved using **dev_Connect( )**. For video, the video stream is available only in its native format (the one in which it was generated), so this function does not perform video transcoding.

Parameter	Description
devHandle1	specifies a valid channel device handle obtained when the channel was opened
devHandle2	specifies a valid channel device handle obtained when the channel was opened
connType	specifies a connection type from among the following valid values: • **DM_FULLDUP** – Specifies full-duplex communication (default) • **DM_HALFDUP –** Specifies half-duplex communication where the first device listens to the second device (i.e., **devHandle1** listens to **devHandle2**)
mode	specifies how the function should be executed. Set this to one of the following: • EV_ASYNC – asynchronously • EV_SYNC – synchronously (default)

■ Supported Connections

The **dev_Connect( )** function can create connections between devices including:

Multimedia and IP Media
: A full-duplex or half-duplex connection between an IP media device and a multimedia device. Requires a valid IP media device handle obtained through the **ipm_Open( )** function and a valid multimedia device handle obtained through the **mm_Open( )** function. Both synchronous and asynchronous modes are supported. In the half-duplex connection, either type of device can listen to the other.

T.38 Fax and IP Media
: A full-duplex connection between an IP media device and a T.38 UDP fax device. Requires a valid T.38 UDP fax device handle obtained through the **fx_open( )** function and a valid IP media device handle obtained through the **ipm_Open( )** function. Both synchronous and asynchronous modes are supported.

CNF Audio Conferencing Party and Voice
: A full-duplex or half-duplex connection between an audio conferencing party device (CNF API) and a voice device. Requires a valid audio conferencing party device handle obtained through the **cnf_OpenParty( )** function and a valid voice device handle obtained through the **dx_open( )** function. Both synchronous and asynchronous modes are supported. In the half-duplex connection, either type of device can listen to the other.

CNF Audio Conferencing Party and IP Media
: A full-duplex or half-duplex connection between an audio conferencing party device (CNF API) and an IP media device. Requires a valid audio conferencing party device handle obtained through the **cnf_OpenParty( )** function and a valid IP media device handle obtained through the **ipm_Open( )** function. Both synchronous and asynchronous modes are supported. In the half-duplex connection, the IP device can listen to the conferencing party device.

CNF Audio Conferencing Party and CNF Audio Conferencing Party
: A full-duplex connection between two audio conferencing party devices (CNF API). Requires valid audio conferencing party device handles obtained through the **cnf_OpenParty( )** function. Both synchronous and asynchronous modes are supported.

CNF Audio Conferencing Party and Digital Network Interface (DTI) Device
: A full-duplex or half-duplex connection between an audio conferencing party device (CNF API) and a DTI device. Requires a valid audio conferencing party device handle obtained through the **cnf_OpenParty( )** function and a valid DTI device handle obtained through the **dt_open( )** function. Both synchronous and asynchronous modes are supported. In the half-duplex connection, either type of device can listen to the other.

M3G Audio Device and CNF Audio Conferencing Party
: A full-duplex or half-duplex connection between a 3G-324M audio device (M3G API) and an audio conferencing party device (CNF API). Requires a valid M3G audio device handle obtained through the **m3g_Open( )** function and a valid device handle obtained through the **cnf_OpenParty( )** function. Both synchronous and asynchronous modes are supported. In a half-duplex connection, either type of device can listen to the other.

: This connection type is not supported in Dialogic® HMP Software 3.0WIN.

M3G Audio Device and Voice / Digital Network Interface (DTI) Device
: A full-duplex or half-duplex connection between a 3G-324M audio device (M3G API) and a voice device or DTI device. Requires a valid M3G audio device handle obtained through the

m3g_Open( ) function and a valid device handle obtained through the **dx_open( )** or **dt_open( )** function. Both synchronous and asynchronous modes are supported. In a half-duplex connection, either type of device can listen to the other.

This connection type is not supported in Dialogic® HMP Software 3.0WIN.

M3G Control Device and Digital Network Interface (DTI) Device
A full-duplex or half-duplex connection between a 3G-324M control device (M3G API) and a DTI device. Requires a valid M3G control device handle obtained through the **m3g_Open( )** function and a valid DTI device handle obtained through the **dt_open( )** function. Both synchronous and asynchronous modes are supported. In a half-duplex connection, either type of device can listen to the other.

This connection type is not supported in Dialogic® HMP Software 3.0WIN.

To break the connection made by **dev_Connect( )**, use the **dev_Disconnect( )** function.

To connect other device types, use the technology-specific routing functions, such as **dx_listen( )** and **dt_listen( )**.

■ Asynchronous Operation

To run this function asynchronously, set the mode parameter to EV_ASYNC. The function returns 0 to indicate it has initiated successfully. The function generates a DMEV_CONNECT termination event for each device to indicate successful completion of the function operation. The function always generates one event for each device regardless of whether the connection type is full-duplex or half-duplex (i.e., a successful half- or full-duplex connection will generate two events). The application program must wait for the completion events that indicate the connection was successful. Use the Dialogic® Standard Runtime Library (SRL) functions to process the termination events. The device handle for the connected device can be obtained from the successful termination event by using the **sr_getevtdev( )** function.

This function generates a DMEV_CONNECT_FAIL error event for each device to indicate failure of the function operation. The function always generates one event for each device regardless of whether the failed connection type is full-duplex or half-duplex. Use **dev_GetResultInfo( )** to retrieve the error information.

■ Synchronous Operation

To run this function synchronously, set the mode parameter to EV_SYNC. This function returns 0 to indicate successful completion and -1 to indicate failure. Use **dev_ErrorInfo( )** to retrieve the error information.

Note: Synchronous operation is not supported for multimedia device connection or disconnection.

■ Cautions

- The **dev_Connect( )** function must be called from the same process that opens the devices and obtains the device handles used in the function.
- To break a connection made by **dev_Connect( )**, you must use the **dev_Disconnect( )** function.
- Multiple connections on a device are not allowed. Once a **dev_Connect( )** has been successfully performed on a device, the device is considered to be connected regardless of

whether the device is listening or being listened to. If you attempt to perform **dev_Connect( )** more than once on a device without first disconnecting the device, the function generates an EDEV_DEVICEBUSY error. This also means that you cannot create a full-duplex connection by performing two half-duplex connections on the same devices. To create a full-duplex connection in this situation, you must first disconnect the half-duplex connection and then create a full-duplex connection.

- If **dev_Connect( )** fails in doing either part of a full-duplex connection, the operation as a whole fails and no connection will be made (i.e., it does not create a half-duplex connection).

■ Multimedia Sample

The following sample programming sequence describes how to connect a multimedia device to an IP media channel using a **half-duplex** connection and then **play** a multimedia clip over IP. It is intended as a basic guideline to show some of the steps involved in general terms.

- Use the **ipm_Open( )** function to open the IP media device and get the device handle.
- Use the **mm_Open( )** function to open the multimedia device and get the device handle.
- Use the **dev_Connect( )** function to make a half-duplex connection (DM_HALFDUP) between the IP media device and the multimedia device, specifying the IP media device as **devHandle1** (listen/receive) and the multimedia device as **devHandle2** (transmit). For playing multimedia, the IP media device (**devHandle1)** must listen to the multimedia device (**devHandle2**).
- Wait for the DMEV_CONNECT events for both the IP media device and the multimedia device to confirm that the **dev_Connect( )** function was successful.
- Set MediaData[0].eMediaType = MEDIATYPE_VIDEO_LOCAL_RTP_INFO. Set MediaData[1].eMediaType = MEDIATYPE_AUDIO_LOCAL_RTP_INFO. Then use the **ipm_GetLocalMediaInfo( )** function and get the local multimedia port and IP address information from the IPMEV_GET_LOCAL_MEDIA_INFO event.
- Obtain the remote end multimedia port and IP address by using Global Call in 3PCC mode for SDP/SIP, or by using a call control framework other than Global Call for other use cases.
- Initialize the IPM_MEDIA_INFO data structure with all media information, including local and remote IP port and address obtained earlier. For full multimedia transmission (audio and video), set eMediaType to the following:
    - MEDIATYPE_AUDIO_LOCAL_RTP_INFO
    - MEDIATYPE_AUDIO_LOCAL_RTCP_INFO
    - MEDIATYPE_AUDIO_LOCAL_CODER_INFO
    - MEDIATYPE_VIDEO_LOCAL_RTP_INFO
    - MEDIATYPE_VIDEO_LOCAL_RTCP_INFO
    - MEDIATYPE_VIDEO_LOCAL_CODER_INFO
    - MEDIATYPE_AUDIO_REMOTE_RTP_INFO
    - MEDIATYPE_AUDIO_REMOTE_RTCP_INFO
    - MEDIATYPE_AUDIO_REMOTE_CODER_INFO
    - MEDIATYPE_VIDEO_REMOTE_RTP_INFO
    - MEDIATYPE_VIDEO_REMOTE_RTCP_INFO
    - MEDIATYPE_VIDEO_REMOTE_CODER_INFO
- Use the **ipm_StartMedia( )** function to start the media session.

- Wait for the IPMEV_STARTMEDIA event to confirm that the **ipm_StartMedia( )** function was successful.
- Initialize the parameters for the **mm_Play( )** function, including a list of multimedia files to play and the runtime control information.
- Use the **mm_Play( )** function to transmit the multimedia data from the multimedia device to the IP media device.
- Wait for the MMEV_PLAY_ACK event to confirm that the **mm_Play( )** function started successfully.
- Wait for the MMEV_PLAY event to confirm that the **mm_Play( )** function completed successfully.
- Use the **ipm_Stop( )** function to tear down the media session.
- Use the **dev_Disconnect( )** function on the IP media device (listening device) to break the half-duplex connection.
- Wait for the DMEV_DISCONNECT event on the IP device.

To **record** multimedia using a **half-duplex** connection, you can use the same procedure but with the following differences:

- When you use the **dev_Connect( )** function to create the half-duplex connection between the IP media device and the multimedia device, specify the multimedia device as **devHandle1** (receive) and the IP media device as **devHandle2** (transmit). For recording, the multimedia device (**devHandle1**) must listen to the IP media device (**devHandle2**).
- Use the **mm_Record( )** function rather than **mm_Play( )**, and wait for the corresponding MMEV_RECORD_ACK and MMEV_RECORD events.
- Use the **dev_Disconnect( )** function on the multimedia device (receive device) to break the half-duplex connection.

Note: If you want to both **play and record** over the same connection, you can use the **dev_Connect( )** function to establish a **full-duplex** connection between the IP media device and the multimedia device (as long as the devices are not already connected). To completely break the full-duplex connection when done, you must call the **dev_Disconnect( )** function twice: once for the IP media device and once for the multimedia device.

■ T.38 Fax Sample

The following sample programming sequence describes how to make and break a T.38 fax session over an IP media channel. It is intended as a basic guideline to show some of the steps involved in general terms.

- Use the **ipm_Open( )** function to open the IP media device and get the device handle.
- Use the **dx_open( )** function to open the voice resource device and get the device handle.
- Use the **dx_getfeaturelist( )** function to get feature information on the voice device handle.
- Check the ft_fax feature table information to see if it is a valid fax device (FT_FAX).
- Use the **fx_open( )** function to open the fax resource device and get the device handle.
- Check the ft_fax feature table information to see if it is a valid T.38 fax device (FT_FAX_T38UDP).
- Use the **dev_Connect( )** function to make a full-duplex connection (DM_FULLDUP) between the IP media device and the fax device.

- Wait for the DMEV_CONNECT events for both the IP media device and the fax device to confirm that the **dev_Connect( )** function was successful.
- Set MediaData[0].eMediaType = MEDIATYPE_LOCAL_UDPTL_T38_INFO, and use the **ipm_GetLocalMediaInfo( )** function to get the local T.38 port and IP address information.
- Wait for the IPMEV_GET_LOCAL_MEDIA_INFO event.
- Obtain the remote end T.38 port and IP address. This would usually be obtained by using a signaling protocol such as H.323 or SIP.
- Use the **ipm_StartMedia( )** function and specify the remote T.38 port and IP address obtained earlier.
- Wait for the IPMEV_STARTMEDIA event to confirm that the **ipm_StartMedia( )** function was successful.
- Use the **fx_sendfax( )** function to start the fax transmission.
- Wait for the TFX_FAXSEND event to confirm that the **fx_sendfax( )** function was successful.
- Use the **ipm_Stop( )** function to conclude the session.
- Use the **dev_Disconnect( )** function on the IP media device and on the fax device to break both sides of the full-duplex connection.

Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information. Possible errors for this function include:

EDEV_DEVICEBUSY
: At least one of the devices specified is currently in use by another Device Management API function call.

EDEV_FAX_SUBSYSTEMERR
: A subsystem error occurred during an internal call to a fax library function because the subsystem function was unable to start (this is not a Device Management API error). See the fax library documentation for the fax error codes and descriptions.

EDEV_INVALIDCONNTYPE
: An invalid connection type (**connType**) was specified (e.g., T.38 UDP fax connection must be full-duplex).

EDEV_INVALIDDEVICEHANDLE
: An invalid device handle was specified. For the **dev_Connect( )** function, the Supported Connections do not allow connection of the specified types of devices. Valid handles are listed in Supported Connections.

EDEV_INVALIDMODE
: An invalid **mode** was specified for executing the function synchronously or asynchronously (EV_SYNC or EV_ASYNC).

EDEV_IPM_SUBSYSTEMERR
: A subsystem error occurred during an internal call to an IP media library function because the subsystem function was unable to start (this is not a Device Management API error). See the IP media library documentation for the IP media error codes and descriptions.

EDEV_MM_SUBSYSTEMERR
A subsystem error occurred during an internal call to a multimedia library function because the subsystem function was unable to start (this is not a Device Management API error). See the multimedia library documentation for the multimedia error codes and descriptions.

See also Chapter 5, “Error Codes” for additional information.

■ Example A (Multimedia Asynchronous)

The following example code shows how the function is used in asynchronous mode.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

#include <srllib.h>
#include <dxxxlib.h>
#include <faxlib.h>
#include <ipmlib.h>
#include <devmgmt.h>
#include <mmlib.h>

static int ipm_handle = -1;
static int mm_handle = -1;

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

static bool ipm_handle_disconnected = false;
static bool mm_handle_disconnected = false;

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

  switch( evttype )
  {
  case DMEV_CONNECT:
    printf( "DMEV_CONNECT event received by IPM device.\n" );
    {
      local_info.MediaData[0].eMediaType=MEDIATYPE_VIDEO_LOCAL_RTP_INFO;
      local_info.MediaData[1].eMediaType=MEDIATYPE_AUDIO_LOCAL_RTP_INFO;

      if( ipm_GetLocalMediaInfo( ipm_handle, &local_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 = 12;

      local_info.MediaData[0].eMediaType=MEDIATYPE_VIDEO_LOCAL_RTP_INFO;
      local_info.MediaData[0].eMediaType=MEDIATYPE_AUDIO_LOCAL_RTP_INFO;

      info.MediaData[0].eMediaType=MEDIATYPE_AUDIO_LOCAL_RTP_INFO;
      info.MediaData[0].mediaInfo.PortInfo.unPortId =
local_info.MediaData[2].mediaInfo.PortInfo.unPortId;
```

```
      strcpy(info.MediaData[0].mediaInfo.PortInfo.cIPAddress,
local_info.MediaData[2].mediaInfo.PortInfo.cIPAddress);

      info.MediaData[1].eMediaType=MEDIATYPE_AUDIO_LOCAL_RTCP_INFO;
      info.MediaData[1].mediaInfo.PortInfo.unPortId =
local_info.MediaData[3].mediaInfo.PortInfo.unPortId;
      strcpy(info.MediaData[1].mediaInfo.PortInfo.cIPAddress,
local_info.MediaData[3].mediaInfo.PortInfo.cIPAddress);

      info.MediaData[2].eMediaType=MEDIATYPE_AUDIO_REMOTE_RTP_INFO;
      info.MediaData[2].mediaInfo.PortInfo.unPortId = 4800;
      strcpy(info.MediaData[2].mediaInfo.PortInfo.cIPAddress, "146.152.86.45");

      info.MediaData[3].eMediaType=MEDIATYPE_AUDIO_REMOTE_RTCP_INFO;
      info.MediaData[3].mediaInfo.PortInfo.unPortId = 4801;
      strcpy(info.MediaData[3].mediaInfo.PortInfo.cIPAddress, "146.152.86.45");

      info.MediaData[4].eMediaType=MEDIATYPE_AUDIO_LOCAL_CODER_INFO;
      // AudioCoderInfo
      info.MediaData[4].mediaInfo.CoderInfo.eCoderType=CODER_TYPE_G711ULAW64K;
      info.MediaData[4].mediaInfo.CoderInfo.eFrameSize=CODER_FRAMESIZE_20;
      info.MediaData[4].mediaInfo.CoderInfo.unFramesPerPkt=1;
      info.MediaData[4].mediaInfo.CoderInfo.eVadEnable=CODER_VAD_DISABLE;
      info.MediaData[4].mediaInfo.CoderInfo.unCoderPayloadType=0;
      info.MediaData[4].mediaInfo.CoderInfo.unRedPayloadType=0;

      info.MediaData[5].eMediaType=MEDIATYPE_AUDIO_REMOTE_CODER_INFO;
      // AudioCoderInfo
      info.MediaData[5].mediaInfo.CoderInfo.eCoderType=CODER_TYPE_G711ULAW64K;
      info.MediaData[5].mediaInfo.CoderInfo.eFrameSize=CODER_FRAMESIZE_20;
      info.MediaData[5].mediaInfo.CoderInfo.unFramesPerPkt=1;
      info.MediaData[5].mediaInfo.CoderInfo.eVadEnable=CODER_VAD_DISABLE;
      info.MediaData[5].mediaInfo.CoderInfo.unCoderPayloadType=0;
      info.MediaData[5].mediaInfo.CoderInfo.unRedPayloadType=0;

      info.MediaData[6].eMediaType=MEDIATYPE_VIDEO_LOCAL_RTP_INFO;
      info.MediaData[6].mediaInfo.PortInfo.unPortId =
           local_info.MediaData[0].mediaInfo.PortInfo.unPortId;
      strcpy(info.MediaData[6].mediaInfo.PortInfo.cIPAddress,
           local_info.MediaData[0].mediaInfo.PortInfo.cIPAddress);

      info.MediaData[7].eMediaType=MEDIATYPE_VIDEO_LOCAL_RTCP_INFO;
      info.MediaData[7].mediaInfo.PortInfo.unPortId =
           local_info.MediaData[1].mediaInfo.PortInfo.unPortId;
      strcpy(info.MediaData[7].mediaInfo.PortInfo.cIPAddress,
           local_info.MediaData[1].mediaInfo.PortInfo.cIPAddress);

      info.MediaData[8].eMediaType=MEDIATYPE_VIDEO_REMOTE_RTP_INFO;
      info.MediaData[8].mediaInfo.PortInfo.unPortId = 4900;
      strcpy(info.MediaData[8].mediaInfo.PortInfo.cIPAddress, "146.152.86.45");

      info.MediaData[9].eMediaType=MEDIATYPE_VIDEO_REMOTE_RTCP_INFO;
      info.MediaData[9].mediaInfo.PortInfo.unPortId = 4901;
      strcpy(info.MediaData[9].mediaInfo.PortInfo.cIPAddress, "146.152.86.45");

      // This is assuming local will always be == remote for coder info...
      info.MediaData[10].eMediaType=MEDIATYPE_VIDEO_LOCAL_CODER_INFO;
      info.MediaData[10].mediaInfo.VideoCoderInfo.unVersion=0;
      info.MediaData[10].mediaInfo.VideoCoderInfo.eCoderType=CODER_TYPE_H263;
      info.MediaData[10].mediaInfo.VideoCoderInfo.unFrameRate = 1500;
      info.MediaData[10].mediaInfo.VideoCoderInfo.unSamplingRate = 90000;
      info.MediaData[10].mediaInfo.VideoCoderInfo.unCoderPayloadType = 34;
      info.MediaData[10].mediaInfo.VideoCoderInfo.unProfileID = 0;
      info.MediaData[10].mediaInfo.VideoCoderInfo.unLevelID = 10;
      info.MediaData[10].mediaInfo.VideoCoderInfo.unSizeofVisualConfigData = 0;
      info.MediaData[10].mediaInfo.VideoCoderInfo.szVisualConfigData = NULL;
      info.MediaData[11].eMediaType=MEDIATYPE_VIDEO_REMOTE_CODER_INFO;
      info.MediaData[11].mediaInfo.VideoCoderInfo.unVersion=0;
      info.MediaData[11].mediaInfo.VideoCoderInfo.eCoderType=CODER_TYPE_H263;
      info.MediaData[11].mediaInfo.VideoCoderInfo.unFrameRate = 1500;
      info.MediaData[11].mediaInfo.VideoCoderInfo.unSamplingRate = 90000;
      info.MediaData[11].mediaInfo.VideoCoderInfo.unCoderPayloadType = 34;
      info.MediaData[11].mediaInfo.VideoCoderInfo.unProfileID = 0;
      info.MediaData[11].mediaInfo.VideoCoderInfo.unLevelID = 10;
      info.MediaData[11].mediaInfo.VideoCoderInfo.unSizeofVisualConfigData = 0;
      info.MediaData[11].mediaInfo.VideoCoderInfo.szVisualConfigData = NULL;

      if(ipm_StartMedia( ipm_handle, &info, DATA_IP_TDM_BIDIRECTIONAL, EV_ASYNC ) == -1 )
      {
        printf( "ipm_StartMedia() failed.\n" );
        exit( 1 );
      }
    }
    break;

  case DMEV_DISCONNECT:
    printf( "DMEV_DISCONNECT event received.\n" );
    ipm_handle_disconnected = true;
    if( mm_handle_disconnected )
    {
      // keep the event. Propogate to waitevt() in Main
      return 1;
    }
    break;

  case IPMEV_STARTMEDIA:
    printf( "IPMEV_STARTMEDIA event received.\n" );
      {
        int  item = 0;

        MM_PLAY_INFO play_info;
        MM_PLAY_RECORD_LIST playlist[2];
        MM_MEDIA_ITEM_LIST mediaitemlist1;
        MM_MEDIA_ITEM_LIST mediaitemlist2;
        MM_AUDIO_CODEC AudioCodecType;
        MM_VIDEO_CODEC VideoCodecType;

        // Create Audio
        INIT_MM_AUDIO_CODEC(&AudioCodecType);
        AudioCodecType.unCoding = 1;
        AudioCodecType.unSampleRate = 8000;
        AudioCodecType.unBitsPerSample = 16;
        mediaitemlist1.item.audio.codec = AudioCodecType;
        mediaitemlist1.item.audio.unMode = 0x0020;        // VOX File
        mediaitemlist1.item.audio.unOffset = 0;
        mediaitemlist1.item.audio.szFileName = "Audio.aud";

        mediaitemlist1.ItemChain = EMM_ITEM_EOT;

        // Create Video
        INIT_MM_VIDEO_CODEC(&VideoCodecType);
        VideoCodecType.Coding = EMM_VIDEO_CODING_DEFAULT;
        VideoCodecType.Profile = EMM_VIDEO_PROFILE_DEFAULT;
        VideoCodecType.Level = EMM_VIDEO_LEVEL_DEFAULT;
        VideoCodecType.ImageWidth = EMM_VIDEO_IMAGE_WIDTH_DEFAULT;
        VideoCodecType.ImageHeight = EMM_VIDEO_IMAGE_HEIGHT_DEFAULT;
        VideoCodecType.BitRate = EMM_VIDEO_BITRATE_DEFAULT;
        VideoCodecType.FramesPerSec = EMM_VIDEO_FRAMESPERSEC_DEFAULT;
        mediaitemlist2.item.video.codec = VideoCodecType;
        mediaitemlist2.item.video.unMode = 0;                // Normal Mode
        mediaitemlist2.item.video.szFileName = "Video.vid";

        mediaitemlist2.ItemChain = EMM_ITEM_EOT;
        INIT_MM_PLAY_RECORD_LIST(&playlist[item]);
        playlist[item].ItemType = EMM_MEDIA_TYPE_AUDIO;
        playlist[item].list = &mediaitemlist1;
        playlist[item].ItemChain = EMM_ITEM_CONT;
        item++;
        INIT_MM_PLAY_RECORD_LIST(&playlist[item]);
        playlist[item].ItemType = EMM_MEDIA_TYPE_VIDEO;
        playlist[item].list = &mediaitemlist2;
        playlist[item].ItemChain = EMM_ITEM_EOT;

        INIT_MM_PLAY_INFO(&play_info);
        play_info.eFileFormat = EMM_FILE_FORMAT_PROPRIETARY;
        play_info.list = playlist;

        mm_Play(mm_handle, &play_info, NULL, NULL);
    }
    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( mm_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( "Unknow event %d received.\n", evttype );
    break;
  }

  return 0;
}

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

  switch( evttype )
  {
  case MMEV_OPEN:
    printf( "MMEV_OPEN event received.\n" );
    break;

  case DMEV_CONNECT:
    printf( "DMEV_CONNECT event received by MM device.\n" );
    break;

  case MMEV_PLAY_ACK:
    printf( "Play has been initiated.\n" );
    break;

  case MMEV_PLAY:
    printf( "Play has finished.\n" );
```

```
      // keep the event. Propogate to waitevt() in Main
      return 1;
      break;

    case DMEV_DISCONNECT:
      printf( "DMEV_DISCONNECT event received.\n" );
      mm_handle_disconnected = true;
      if( ipm_handle_disconnected )
      {
        // keep the event. Propogate to waitevt() in Main
        return 1;
      }
      break;

    default:
      printf( "Unknown event %d received on MM 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 mm_handle = mm_Open("mmB1C1", NULL, NULL);
    if( mm_handle == -1 )
    {
      printf( "mm_open() failed.\n" );
      exit( 1 );
    }

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

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

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

    // Wait for Connection and Multimedia Play to complete
    sr_waitevt(-1);

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

    if( dev_Disconnect( mm_handle, EV_ASYNC ) == -1 )
    {
```

```
    printf( "dev_Disconnect() failed.\n" );
    exit( 1 );
  }

  // Wait for DisConnect to complete
  sr_waitevt(-1);

  if( sr_dishdlr( mm_handle, EV_ANYEVT, MMEventHandler ) == -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( mm_Close( mm_handle, NULL) == -1 )
  {
    printf( "mm_close() failed.\n" );
    exit( 1 );
  }

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

■ Example B (T.38 Fax Asynchronous)

The following example code shows how the function is used in asynchronous mode.

```
#include <stdio.h>
#include <stdlib.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();

  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 = 6001;  // 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;

        if( ipm_StartMedia( ipm_handle, &info, DATA_IP_TDM_BIDIRECTIONAL, EV_ASYNC ) == -1 )
        {
          printf( "ipm_StartMedia() failed.\n" );
          exit( 1 );
        }
      }
      break;

    case DMEV_DISCONNECT:
      printf( "DMEV_DISCONNECT event received.\n" );

      ipm_handle_disconnected = true;

      if( fax_handle_disconnected )
      {
        return 1;
      }
      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( "Unknow 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;

    case DMEV_CONNECT:
      printf( "DMEV_CONNECT event received.\n" );
      break;

    case DMEV_DISCONNECT:
      printf( "DMEV_DISCONNECT event received.\n" );
      fax_handle_disconnected = true;
      if( ipm_handle_disconnected )
      {
        return 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 );
    }

    sr_waitevt(-1);

    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 );
    }
  }
```

■ Example C (T.38 Fax Synchronous)

The following example code shows how the function is used in synchronous mode.

```
#include <srllib.h>
#include <dxxxlib.h>
#include <faxlib.h>
#include <ipmlib.h>
#include <devmgmt.h>

void main()
{
  int FaxHandle = fx_open( "dxxxB1C1", 0 );

  if( FaxHandle == -1 )
  {
    printf( "Can not open fax channel.\n" );
    // Perform system error processing
    exit( 1 );
  }


  int IpmHandle = ipm_Open( "ipmB1C1", 0, EV_SYNC );

  if( IpmHandle == -1 )
```

```
{
  printf( "Can not open IPM handle.\n" );
  // Perform system error processing
  exit( 1 );
}


if( dev_Connect( IpmHandle, FaxHandle, DM_FULLDUP, EV_SYNC ) == -1 )
{
  printf( "dev_Connect() failed.\n" );
  exit( 1 );
}

IPM_MEDIA_INFO info;

...
// Setup IPM_MEDIA_INFO structure

if( ipm_StartMedia( IpmHandle, &info, DATA_IP_TDM_BIDIRECTIONAL, EV_SYNC ) == -1 )
{
  printf( "ipm_StartMedia() failed.\n" );
  exit( 1 );
}

if( fx_initstat( FaxHandle, DF_TX ) == -1 )
{
  printf( "fx_initstat() failed.\n" );
  exit( 1 );
}

DF_IOTT iott;
...
// Setup DF_IOTT entries for sending fax

if( fx_sendfax( FaxHandle, &iott, EV_SYNC ) == -1 )
{
  printf( "fx_sendfax() failed.\n" );
  exit( 1 );
}

if( ipm_Stop( IpmHandle, STOP_ALL, EV_SYNC ) == -1 )
{
  printf( "ipm_Stop() failed.\n" );
  exit( 1 );
}


if( dev_Disconnect( IpmHandle, EV_SYNC ) == -1 )
{
  printf( "dev_Disconnect() for IPM channel failed.\n" );
  exit( 1 );
}

if( dev_Disconnect( FaxHandle, EV_SYNC ) == -1 )
{
  printf( "dev_Disconnect() for Fax channel failed.\n" );
  exit( 1 );
}

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

```
    if( ipm_Close( IpmHandle ) == -1 )
    {
      printf( "ipm_Close() failed.\n" );
    }
  }
```

■ See Also

- **dev_Disconnect( )**

dev_Disconnect( )

Name:	int dev_Disconnect (devHandle, mode)	
Inputs:	int devHandle	• a valid channel device
	unsigned short mode	• asynchronous or synchronous function mode
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h	
Category:	Device Connection	
Mode:	asynchronous or synchronous	

■ Description

The **dev_Disconnect( )** function breaks the connection between the receive channel of the specified device and the transmit channel of the device that was associated with it by means of the **dev_Connect( )** function. To break a full-duplex connection that was originally established between the devices with **dev_Connect( )**, you must call **dev_Disconnect( )** for each device.

To break a half-duplex connection between a multimedia device and an IP media device, you must disconnect the receive side, which is typically the IP media device for an **mm_Play( )** and the multimedia device for an **mm_Record( )**.

Note: The terms *listen* and *receive* are used synonymously.

Parameter	Description
devHandle	specifies a valid channel device handle obtained when the channel was opened
mode	specifies how the function should be executed. Set this to one of the following: • EV_ASYNC – asynchronously • EV_SYNC – synchronously (default)

■ Asynchronous Operation

To run this function asynchronously, set the mode parameter to EV_ASYNC. The function returns 0 to indicate it has initiated successfully. The function generates a DMEV_DISCONNECT termination event to indicate successful completion of the function operation. The application program must wait for the completion event that indicates the disconnection was successful. Use the Dialogic® Standard Runtime Library (SRL) functions to process the termination events.

This function generates a DMEV_DISCONNECT_FAIL error event to indicate failure of the function operation. Use **dev_GetResultInfo( )** to retrieve the error information.

■ Synchronous Operation

To run this function synchronously, set the mode parameter to EV_SYNC. This function returns 0 to indicate successful completion and -1 to indicate failure. Use **dev_ErrorInfo( )** to retrieve the error information.

Note: Synchronous operation is not supported for multimedia device connection or disconnection.

■ Cautions

- The **dev_Disconnect( )** function must be called from the same process that opens the device and obtains the device handle used in the function.
- To break a connection made by **dev_Connect( )**, you must use the **dev_Disconnect( )** function.
- If you attempt to perform **dev_Disconnect( )** on a device that is not connected (for example, if it is called on a device without having successfully used **dev_Connect( )** on the device, or if it is called twice in a row on a device), the function generates an EDEV_NOTCONNECTED error.
- If you have a full-duplex connection that was originally established between the devices with **dev_Connect( )**, and you break only one half of the connection with **dev_Disconnect( )**, a half-duplex connection will remain between the devices until you perform **dev_Disconnect( )** on the other device in the connection.

■ Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information. Possible errors for this function include:

EDEV_DEVICEBUSY
: At least one of the devices specified is currently in use by another Dialogic® Device Management API function call.

EDEV_FAX_SUBSYSTEMERR
: A subsystem error occurred during an internal call to a fax library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error). See the fax library documentation for the fax error codes and descriptions.

EDEV_INVALIDDEVICEHANDLE
: An invalid device handle was specified. For the **dev_Connect( )** function, the Supported Connections do not allow connection of these types of devices. (Valid handles include IP media, multimedia, and T.38 UDP fax devices.)

EDEV_INVALIDMODE
: An invalid **mode** was specified for executing the function synchronously or asynchronously (EV_SYNC or EV_ASYNC).

EDEV_INVALIDSTATE
: Device is in an invalid state for the current function call. For example, the **dev_Disconnect( )** function may have been called before both devices were fully connected by the **dev_Connect( )** function.

EDEV_IPM_SUBSYSTEMERR
: A subsystem error occurred during an internal call to an IP media library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error). See the IP media library documentation for the IP media error codes and descriptions.

EDEV_MM_SUBSYSTEMERR
: A subsystem error occurred during an internal call to a multimedia library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error). See the multimedia library documentation for the multimedia error codes and descriptions.

EDEV_NOTCONNECTED
: An attempt was made to perform **dev_Disconnect( )** on a device that is not connected.

See also Chapter 5, “Error Codes” for additional information.

■ Example (Synchronous/Asynchronous)

For examples that show how the function is used to disconnect devices in synchronous or asynchronous mode, see the example code in the **dev_Connect( )** function.

■ See Also

- **dev_Connect( )**

dev_ErrorInfo( )

Name:	int dev_ErrorInfo (pErrInfo)	
Inputs:	DEV_ERRINFO *pErrInfo	• pointer to error information structure
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h	
Category:	Error Processing	
Mode:	synchronous	

■ Description

The **dev_ErrorInfo( )** function obtains the error information for the last error in the Dialogic® Device Management API or one of its subsystems and provides it in the DEV_ERRINFO error information structure. The error codes returned in the structure are listed in Chapter 5, “Error Codes”.

Parameter	Description
pErrInfo	specifies a pointer to DEV_ERRINFO error information structure. Upon successful completion of the function operation, the structure is filled with results. See the DEV_ERRINFO data structure in Chapter 4, “Data Structures” for more information.

■ Cautions

- Call **dev_ErrorInfo( )** only when a Dialogic® Device Management API function fails; otherwise, the data in the DEV_ERRINFO structure will be invalid.
- If the error is a subsystem error, to identify the error code, you must include the header file for the technology-specific subsystem (e.g., ipmlib.h or faxlib.h).
- The Dialogic® Device Management API errors are thread-specific (they are only in scope for that thread). Subsystem errors are device-specific.

■ Errors

None.

■ Example

The following example code shows how the function is used.

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

```
#include <devmgmt.h>

void main()
{
   int iphandle, faxhandle;
   int retval;
   DEV_ERRINFO error_info;

   faxhandle=fx_open("dxxxB2C1", NULL);
    iphandle=ipm_Open("ipmB1C1", NULL, EV_SYNC);

   if ((faxhandle == -1) || (iphandle == -1))
   {
      /* handle error opening a device */
   }

   /* ... */
   retval=dev_Connect(iphandle, faxhandle, DM_FULLDUP, EV_SYNC);
    if(retval==-1)
   {
      /* The dev_Connect() call failed.  This may be because of an error on either
         the fax or the IP device.  Use dev_ErrorInfo() to find out, and then print
         an error message. */

      if (dev_ErrorInfo(&error_info) != -1)
      {
         switch (error_info.dev_ErrValue)
         {
         case EDEV_INVALIDDEVICEHANDLE:
            printf("Error because of an invalid handle.\n");
            break;
         case EDEV_INVALIDCONNTYPE:
            printf("Error because of an invalid connection type.\n");
            break;
         case EDEV_IPM_SUBSYSTEMERR:
            printf("Error %d in IPM subsystem.\n", error_info.dev_SubSystemErrValue);
            break;
         case EDEV_FAX_SUBSYSTEMERR:
            printf("Error %d in FAX subsystem.\n", error_info.dev_SubSystemErrValue);
            break;
         default:
            printf("Error type %d in dev_Connect()\n", error_info.dev_ErrValue);
            break;
         }

         /* Print out the string error message returned as well */
         printf(" Error during dev_Connect(): %s\n", error_info.dev_Msg);
      }
   }

   /* ... */
   fx_close(faxhandle);
   ipm_Close(iphandle, NULL);

   return 0;
}
```

■ See Also

None.

dev_GetReceivePortInfo( )

Name:	dev_GetReceivePortInfo (devHandle, pUserContext)	
Inputs:	int devHandle	• a valid channel device
	void *pUserContext	• a pointer to user-specific context
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h port_connect.h	
Category:	Device Connection	
Mode:	asynchronous	

■ Description

The **dev_GetReceivePortInfo( )** function retrieves device receive ports information and returns it in the data associated with the DMEV_GET_RX_PORT_INFO event.

Parameter	Description
devHandle	specifies a valid channel device handle obtained when the channel was opened
pUserContext	specifies a user-supplied pointer that can be retrieved using **sr_getUserContext( )** when the completion event is received

■ Asynchronous Operation

The function returns DEV_SUCCESS to indicate it has initiated successfully. The function generates a DMEV_GET_RX_PORT_INFO event to indicate successful completion of the function operation. Use the Dialogic® Standard Runtime Library (SRL) functions to process the termination event.

This function generates a DMEV_GET_RX_PORT_INFO_FAIL event to indicate failure of the function operation. Use the **dev_GetResultInfo( )** function to obtain the error information.

The user-supplied pointer **pUserContext** is returned with either event and can be retrieved using **sr_getUserContext( )**. The pointer to the DM_PORT_INFO_LIST structure is returned with either event and can be retrieved using **sr_getevtdatap( )**.

For more information on SRL functions, see the *Dialogic® Standard Runtime Library API Library Reference*.

Cautions

The **dev_GetReceivePortInfo( )** function must be called from the same process that opens the device and obtains the device handle used in the function.

Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information. Possible errors for this function include:

EDEV_BADPARM
: Invalid argument or parameter

EDEV_INVALIDDEVICEHANDLE
: Invalid device handle specified

EDEV_SUBSYSTEMERR
: Internal error

Example

```
#include <srllib.h>
#include <ipmlib.h>
#include <port_connect.h>
#include <string.h>
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
     int ret;
     int rc;
     int dev1;
     long evt;
     void* evt_data;
     int evt_len;
     const char szDev1[] = "ipmB1C1";

     ret = 0;
     dev1 = -1;
     try
     {

     // Open device (ipm)
     dev1 = ipm_Open(szDev1, NULL, EV_ASYNC);
     if (-1 == dev1) {
          cout << "ipm_Open error";
          cout << " handle = "  << dev1 << endl;
          throw 1;
     }
     sr_waitevt(-1);
     evt = sr_getevttype();
     if (IPMEV_OPEN != evt) {
          cout << "ipm_Open error";
          cout << " event = " << evt << endl;
          throw 2;
     }

     // Obtain Device Receive Ports
     rc = dev_GetReceivePortInfo(dev1, NULL);
     if (-1 == rc) {
```

```
                cout << "dev_GetReceivePortInfo error";
                cout << " rc = " << rc << endl;
                throw 3;
            }
            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 port_info_list1 = {};
            memcpy(&port_info_list1, evt_data, evt_len);

            // Print number of ports
            cout << "Number of RX ports: " << port_info_list1.unCount << endl;

            }
            catch (int point) {
                ret = -1;
                cerr << "Error point #" << point << " reached" << endl;
            }

            if (dev1 != -1) {
                rc = ipm_Close(dev1, NULL);
                dev1 = -1;
            }

            return ret;
        }
```

■ See Also

- **dev_GetTransmitPortInfo( )**

dev_GetResourceReservationInfo( )

Name:	int dev_GetResourceReservationInfo (devHandle, pResourceInfo, mode)	
Inputs:	int devHandle	• a valid channel device
	DEV_RESOURCE_RESERVATIONINFO *pResourceInfo	• pointer to resource reservation information structure
	unsigned short mode	• asynchronous or synchronous function mode
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h	
Category:	Resource Reservation	
Mode:	asynchronous or synchronous	

■ Description

Note: The **dev_GetResourceReservationInfoEx( )** function supercedes and should be used instead of the **dev_GetResourceReservationInfo()** function. The _Ex( ) function provides improved information about available resources.

The **dev_GetResourceReservationInfo( )** function obtains the current reservation information for the specified resource and device and provides it in the resource reservation information structure.

Parameter	Description
devHandle	specifies a valid channel device handle obtained when the channel was opened
pResourceInfo	specifies a pointer to a resource reservation information structure. Before executing the function, set the resourceType field to the resource type for which you want to obtain information. Upon successful completion of the function operation, the structure is filled with results. See the DEV_RESOURCE_RESERVATIONINFO data structure in Chapter 4, “Data Structures” for more information.
mode	specifies how the function should be executed. Set this to one of the following: • EV_ASYNC – asynchronously • EV_SYNC – synchronously

■ Asynchronous Operation

To run this function asynchronously, set the mode parameter to EV_ASYNC. The function returns 0 to indicate it has initiated successfully. The function generates a DMEV_GET_RESOURCE_RESERVATIONINFO termination event to indicate successful completion of the function operation. The application program must process for the completion event that indicates the operation was successful. Use the Dialogic® Standard Runtime Library (SRL) functions to process the termination event.

This function generates a DMEV_GET_RESOURCE_RESERVATIONINFO_FAIL error event to indicate failure of the function operation.

Note: Typically, asynchronous mode allows an application to continue with execution of other code while waiting for a response from the device to a previous request. In the Resource Reservation functions, various operations on the low bit rate codec are handled in a single thread of execution, so in this case, using **synchronous mode** for the function may be sufficient.

■ Synchronous Operation

To run this function synchronously, set the mode parameter to EV_SYNC. This function returns 0 to indicate successful completion and -1 to indicate failure. Use **dev_ErrorInfo( )** to retrieve the error information.

■ Cautions

- This function requires that the device be open; otherwise, it generates a subsystem error (for example, EDEV_IPM_SUBSYSTEMERR.
- If the specified resource is invalid or not available, it generates a subsystem error (for example, EDEV_IPM_SUBSYSTEMERR).

■ Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information. Possible errors for this function include:

EDEV_INVALIDDEVICEHANDLE
: An invalid device handle was specified. For the **dev_Connect( )** function, the Supported Connections do not allow connection of these types of devices. (Valid handles include IP media, multimedia, and T.38 UDP fax devices.)

EDEV_INVALIDMODE
: An invalid **mode** was specified for executing the function synchronously or asynchronously (EV_SYNC or EV_ASYNC).

EDEV_IPM_SUBSYSTEMERR
: A subsystem error occurred during an internal call to an IP media library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error). See the IP media library documentation for the IP media error codes and descriptions.

See also Chapter 5, “Error Codes” for additional information.

■ Example

The following example code shows how the function is used in synchronous mode.

```
#include "srllib.h"
#include "ipmlib.h"
#include "devmgmt.h"

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

void main()
{
   int devHandle; // channel handle
   .
   .
   // Resister event handler thru SRL
   sr_enbhdlr( EV_ANYDEV, EV_ANYEVT, (HDLR)CheckEvent);

   // Open channel
   if ((devHandle = ipm_Open("ipmB1C1",0)) == -1) {
      printf("Cannot open channel\n");
      // Perform system error processing
      exit(1);
   }

   //e.g. total number of RESOURCE_IPM_LBR in the system is 5

   // Reserve Low Bit Rate Codec for the specified channel
   if (dev_ReserveResource(devHandle, RESOURCE_IPM_LBR, EV_SYNC) ==-1)
   {
      printf("Cannot Reserve LBR resource.\n");
      // Perform system error processing
   }

   // Get Low Bit Rate Codec reservation information
   DEV_RESOURCE_RESERVATIONINFO resInfo;

   INIT_DEV_RESOURCE_RESERVATIONINFO(&resInfo);
   resInfo.resourceType = RESOURCE_IPM_LBR;
   if (dev_GetResourceReservationInfo(devHandle, &resInfo, EV_SYNC) ==-1)
   {
      printf("Cannot Get LBR resource reservation information.\n");
      // Perform system error processing
   }
   printf("LBR Usage for %s: ReservationStatus = %s\n, curReservePoolCount = %d,
          maxReservePoolCount = %d\n", ATDV_NAMEP(devHandle), (resInfo.curReserveCount == 1)
          ? "Reserved"  : "Unreserved", resInfo.curReservePoolCount,
          resInfo.maxRecervePoolCount);

   //Output is  "LBR Usage for ipmB1C1: ReservationStatus = Reserved, curReservePoolCount = 1,
                 maxReservePoolCount = 5"
}
```

■ See Also

None.

dev_GetResourceReservationInfoEx( )

Name:	int dev_GetResourceReservationInfoEx(devHandle, pResourceInfo, mode)	
Inputs:	int devHandle	• valid channel device
	DEV_RESOURCE_RESERVATIONINFO_EX *pResourceInfo	• pointer to resource reservation information structure
	unsigned short mode	• asynchronous or synchronous function mode
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h	
Category:	Resource Reservation	
Mode:	asynchronous or synchronous	

■ Description

The **dev_GetResourceReservationInfoEx( )** function obtains the current reservation information for the specified resource(s) and device, and provides it in the resource reservation information structure.

Note: The **dev_GetResourceReservationInfoEx()** function supercedes and should be used instead of the **dev_GetResourceReservationInfo()** function. The _Ex( ) function provides improved information about available resources.

Note: The **dev_GetResourceReservationInfoEx()** function is not supported in Dialogic® HMP Software 3.0WIN.

Parameter	Description
devHandle	specifies a valid channel device handle obtained when the channel was opened
pResourceInfo	specifies a pointer to a resource reservation information structure DEV_RESOURCE_RESERVATIONINFO_EX. Before executing the function, set the rsInfo[i].resourceType field to the resource type for which you want to obtain information. Set the count field to the number of items in rsInfo array that have been filled. Upon successful completion of the function operation, the structure is filled with results.
mode	specifies how the function should be executed. Set this to one of the following: • EV_ASYNC - asynchronously • EV_SYNC - synchronously

Asynchronous Operation

To run this function asynchronously, set the mode parameter to EV_ASYNC. The function returns 0 to indicate it has initiated successfully. The function generates a DMEV_GET_RESOURCE_RESERVATIONINFO termination event to indicate successful completion of the function operation. The application program must process for the completion event that indicates the operation was successful. Use the Dialogic® Standard Runtime Library (SRL) functions to process the termination event.

This function generates a DMEV_GET_RESOURCE_RESERVATIONINFO_FAIL error event to indicate failure of the function operation. Use **dev_GetResultInfo( )** to retrieve the error information.

Synchronous Operation

To run this function synchronously, set the mode parameter to EV_SYNC. This function returns 0 to indicate successful completion and -1 to indicate failure. Use **dev_ErrorInfo( )** to retrieve the error information.

It is better to use asynchronous mode because **dev_GetResourceReservationInfoEx( )** is not executed in a single thread of execution.

Set up the data structure to obtain resource reservation information for all the audio coders, as follows:

```
DEV_RESOURCE_RESERVATIONINFO_EX resInfo;
INIT_DEV_RESOURCE_RESERVATIONINFO_EX(&resInfo);
resInfo.rsInfo[0].resourceType = RESOURCE_IPM_ALL_AUDIO_CODERS;
resInfo.count = 1;
```

Cautions

- This function requires that the device be open; otherwise, it generates a subsystem error (for example, EDEV_IPM_SUBSYSTEMERR).
- If the specified resource is invalid or not available, it generates a subsystem error (for example, EDEV_IPM_SUBSYSTEMERR).

Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information.

Possible errors for this function include:

EDEV_INVALIDDEVICEHANDLE
: An invalid device handle was specified.

EDEV_INVALIDMODE
: An invalid **mode** was specified for executing the function synchronously or asynchronously.

EDEV_IPM_SUBSYSTEMERR
: A subsystem error occurred during an internal call to an IP media library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error).

■ Asynchronous Code Example

```
int main()
{

int nDeviceID; // channel handle
INIT_DEV_RESOURCE_RESERVATIONINFO_EX(&devResourceReservationInfoEx);

// Register event handler function with srl
sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,CheckEvent);

// Open channel
if ((nDeviceID = ipm_Open("ipmB1C1", NULL, EV_SYNC)) == -1)
{
       printf("Cannot open channel\n");
       // Perform system error processing
       return -1;
}

/*
. .
Main Processing
. .
*/

devResourceReservationInfoEx.rsInfo[0].resourceType = RESOURCE_IPM_G726;
devResourceReservationInfoEx.rsInfo[1].resourceType = RESOURCE_IPM_G729;
devResourceReservationInfoEx.count = 2;

if (dev_GetResourceReservationInfoEx(nDeviceID, &devResourceReservationInfoEx, EV_ASYNC) == -1)

        {
                printf("dev_GetResourceReservationInfoEx failed for device name
                %s \n", ATDV_NAMEP(nDeviceID));
                /*
                . .
                Perform Error Processing
                . .
                */
        }
/* Continue processing */
return 0;
}
```

■ Synchronous Code Example

```
#include "srllib.h"
#include "ipmlib.h"
#include "devmgmt.h"

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

void main()
{
int devHandle; // channel handle
int i;
. .
// Open channel
if ((devHandle = ipm_Open("ipmB1C1",0)) == -1) {
printf("Cannot open channel\n");
// Perform system error processing
exit(1);
}
// Get Low Bit Rate Codec reservation information
DEV_RESOURCE_RESERVATIONINFO_EX resInfo;
INIT_DEV_RESOURCE_RESERVATIONINFO_EX(&resInfo);
resInfo.rsInfo[0].resourceType = RESOURCE_IPM_G729;
resInfo.rsInfo[1].resourceType = RESOURCE_IPM_G723;
resInfo.count = 2;

if (dev_GetResourceReservationInfoEx(devHandle, &resInfo, EV_SYNC) ==-1)
{
printf("Cannot Get resource reservation information.\n");
// Perform system error processing
}

printf("Usage for %s:\n",ATDV_NAMEP(devHandle));

for (int i = 0; i < resInfo.count; i++)
{
     printf(" ResourceType = %d: Reserved = %d, availableResourceCount = %d\n",
     resInfo.rsInfo[i].resourceType,
     resInfo.rsInfo[i].curReserveCount,
     resInfo.rsInfo[i].availableResourceCount);

}
..
..
..
/* Continue processing */

]
```

■ See Also

- **dev_ReleaseResourceEx( )**
- **dev_ReserveResourceEx( )**

dev_GetResultInfo( )

Name:	int dev_GetResultInfo(devHandle, eventType, peventData, pdevInfo)	
Inputs:	long devHandle	• SRL device handle
	unsigned long eventType	• SRL event type
	void* peventData	• pointer to SRL event data
	PDM_EVENT_INFO pdevInfo	• pointer to device management event information
Returns:	0 if successful -1 if failure	
Includes:	devmgmt.h	
Category:	Event Handling	
Mode:	Synchronous	

■ Description

The **dev_GetResultInfo( )** function collects information about a given event. For Dialogic® Dialogic® Device Management API events, the event type ID and event data block act as identifiers for extended event information. The **dev_GetResultInfo( )** function uses this data to collect event information and populate a DM_EVENT_INFO structure for the event. The event information may be used for trace logging, debugging, and error handling.

Note: This function is not supported in Dialogic® HMP Software 3.0WIN.

Parameter	Description
devHandle	SRL device handle returned by **sr_getevtdev( )**
eventType	SRL event type returned by **sr_getevttype( )** pointer to SRL event data returned by **sr_getevtdatap( )**
peventData	pointer to device management event
pdevInfo	pointer to the DM_EVENT_INFO structure to be filled with information pertaining to the given event

In order to use this function in a thread of execution other than the one in which the data block was gathered, the application must make a copy of the event data block using the **sr_createevtdatapcopy( )** function, or retrieve the event data block using the **sr_getevtdatapex( )** function with the SR_EVENTDATASCOPE_TAG_USER flag.

Even though the data length returned from the **sr_getevtdatalen( )** function associated with a specific device management library event may be zero, the data pointer will be non-zero. This pointer is a reference pointer in all cases and is used with this function. As a result, the event data block pointer must be used in the call to this function in all cases.

For more information on SRL functions, see the Dialogic® Standard Runtime Library documentation.

Caution

In order to use this function in an execution thread other than the one in which the event data block was gathered, the application must make a copy of the event data block as stated in the Description section. Then the application must call **sr_destroy( )** on the return pointer for either of those functions.

Errors

If this function returns -1 to indicate a failure, use **dev_ErrorInfo( )** to retrieve the reason for the error.

Example

The following example shows a simple retrieval of device management event information.

```
/*
 * ASSUMPTION: An event has been generated by a call to dev_PortConnect() on device
 * with handle a_hDev.
 */
int retrieveEvent(long a_hDev)
{
    Int retCode = -1;

    if (sr_waitevt(10000) == -1)
    {
        printf("wait event failure\n");
        return 0;
    }

    long evttype = sr_getevttype();
    long evtdev = sr_getevtdev();
    void * pevtdata = sr_getevtdatap();

    if (evtdev != a_hDev)
    {
        printf("event for unknown device handle [%ld]\n", evtdev);
    }
    else
    {
        switch(evttype)
        {
        case DMEV_PORT_CONNECT:
            printf("DMEV_PORT_CONNECT event received by device handle [%ld]\n",
                   evtdev);
            retCode = 0;
            break;

        case DMEV_PORT_CONNECT_FAIL:
            DM_EVENT_INFO devInfo;
            INIT_DM_EVENT_INFO(&devInfo);
            if(dev_GetResultInfo(evtdev, evttype, pevtdatap, &devInfo) == -1)
            {
                DEV_ERRINFO errInfo;
                dev_ErrorInfo(&errInfo);
                printf("Error: DMEV_PORT_CONNECT_FAIL event received\n /
                       dev_GetResultInfo() failure: err(%d), sserr(%d) - %s\n”,
                       errInfo.dev_ErrValue,
                       errInfo.dev_SubSystemErrValue,
                       errInfo.dev_Msg);
            }
            else
            {
                printf("Error: DMEV_PORT_CONNECT_FAIL event received\n /
value(%d), subsystem value(%d), message(%s), subsystem message(%s), further information / -
 %s\n",
                        devInfo.nValue,
                        devInfo.nSubSystemValue,
                        devInfo.szMsg,
                        devInfo.szSubSystemMsg,
                        devInfo.szAdditionalInfo);
            }
            break;

        default:
            printf("ERROR: unexpected event received for handle [%ld]: 0x%x\n",
                     evtdev, evttype);
        };
    }
    return retCode;
}
```

■ See Also

None.

dev_GetTransmitPortInfo( )

Name:	dev_GetTransmitPortInfo (devHandle, pUserContext)	
Inputs:	int devHandle	• a valid channel device
	void *pUserContext	• a pointer to user-specific context
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h port_connect.h	
Category:	Device Connection	
Mode:	asynchronous	

■ Description

The **dev_GetTransmitPortInfo( )** function retrieves device transmit ports information and returns it in the data associated with the DMEV_GET_TX_PORT_INFO event.

Parameter	Description
devHandle	specifies a valid channel device handle obtained when the channel was opened
pUserContext	specifies a user-supplied pointer that can be retrieved using **sr_getUserContext( )** when the completion event is received

■ Asynchronous Operation

The function returns DEV_SUCCESS to indicate it has initiated successfully. The function generates a DMEV_GET_TX_PORT_INFO event to indicate successful completion of the function operation. Use the Dialogic® Standard Runtime Library (SRL) functions to process the termination event.

This function generates a DMEV_GET_TX_PORT_INFO_FAIL event to indicate failure of the function operation. Use the **dev_GetResultInfo( )** function to obtain the error information.

The user-supplied pointer **pUserContext** is returned with either event and can be retrieved using **sr_getUserContext( )**. The pointer to the DM_PORT_INFO_LIST structure is returned with either event and can be retrieved using **sr_getevtdatap( )**.

For more information on SRL functions, see the Dialogic® Standard Runtime Library API Library Reference.

Cautions

The **dev_GetTransmitPortInfo( )** function must be called from the same process that opens the device and obtains the device handle used in the function.

Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information. Possible errors for this function include:

EDEV_BADPARM
: Invalid argument or parameter

EDEV_INVALIDDEVICEHANDLE
: Invalid device handle specified

EDEV_SUBSYSTEMERR
: Internal error

Example

```
#include <srllib.h>
#include <ipmlib.h>
#include <port_connect.h>
#include <string.h>
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
     int ret;
     int rc;
     int dev2;
     long evt;
     void* evt_data;
     int evt_len;
     const char szDev2[] = "ipmB1C2";

     ret = 0;
     dev2 = -1;
     try
     {

     // Open device (ipm)
     dev2 = ipm_Open(szDev2, NULL, EV_ASYNC);
     if (-1 == dev2) {
          cout << "ipm_Open error";
          cout << " handle = "  << dev2 << endl;
          throw 1;
     }
     sr_waitevt(-1);
     evt = sr_getevttype();
     if (IPMEV_OPEN != evt) {
          cout << "ipm_Open error";
          cout << " event = " << evt << endl;
          throw 2;
     }

     // Obtain Device Transmit Ports
     rc = dev_GetTransmitPortInfo(dev2, NULL);
     if (-1 == rc) {
```

```
                    cout << "dev_GetReceivePortInfo error";
                    cout << " rc = " << rc << endl;
                    throw 3;
               }
               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();
               int evt_len = sr_getevtlen();
               DM_PORT_INFO_LIST port_info_list1 = {};
               memcpy(&port_info_list1, evt_data, evt_len);

               // Print number of ports
               cout << "Number of TX ports: " << port_info_list1.unCount << endl;

               }
               catch (int point) {
                    ret = -1;
                    cerr << "Error point #" << point << " reached" << endl;
               }

               if (dev2 != -1) {
                    rc = ipm_Close(dev2, NULL);
                    dev2 = -1;
               }

               return ret;
          }
```

■ See Also

- **dev_GetReceivePortInfo( )**

dev_PortConnect( )

Name:	dev_PortConnect (devHandle, pConnectList, pUserContext)	
Inputs:	int devHandle	• a valid channel device
	CPDM_PORT_CONNECT_INFO_LIST pConnectList	• a pointer to the list of connection structures
	void *pUserContext	• a pointer to user-specific context
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h port_connect.h	
Category:	Device Connection	
Mode:	asynchronous	

■ Description

The **dev_PortConnect( )** function creates half-duplex connections between one or more internal transmit ports of the specified device and internal receive ports of another device or the same device, based on the contents of the connection structures. The receive ports are typically ports of other devices, although they can be receive ports of the same device, which would result in a loop-back connection. Use this function for making internal connections between packet interfaces.

The ports discussed in this function reference information for **dev_PortConnect( )** refer to internal ports.

Parameter	Description
devHandle	specifies a valid channel device handle obtained when the channel was opened
pConnectList	specifies a pointer to the list of connection structures, DM_PORT_CONNECT_INFO_LIST
pUserContext	specifies a user-supplied pointer that can be retrieved using **sr_getUserContext( )** when the completion event is received

Connections are created from the transmit ports and receive ports provided in the list of DM_PORT_CONNECT_INFO structures. Connections may be made from a single transmit port to multiple receive ports by repeating the transmit port in the source DM_PORT_INFO_LIST structure. Connections may also be made from a single transmit port to multiple receive ports by calling **dev_PortConnect( )** multiple times using the same transmit ports and different receive ports in the DM_PORT_CONNECT_INFO_LIST structure.

The Dialogic® Device Management API library checks for compatible port pairs before initiating a connection and rejects the request if a mismatch is detected; see Supported Connections.

The **dev_PortConnect( )** function allows granular control over which ports to connect; for example, audio only or video only. If there are multiple receive ports in the port list, the device will simultaneously transmit to the internal receive ports of multiple devices, creating a one-to-many connection. A transcoding flag in DM_PORT_CONNECT_INFO is used to indicate if the connection is native, or if transcoding should be performed; see Supported Connections for more information. See Multimedia Scenario for usage example.

■ Asynchronous Operation

The function returns DEV_SUCCESS to indicate it has initiated successfully. The function generates a DMEV_PORT_CONNECT event to indicate successful completion of the function operation. Use the Dialogic® Standard Runtime Library (SRL) functions to process the termination event.

This function generates a DMEV_PORT_CONNECT_FAIL event to indicate failure of the function operation. Use **dev_GetResultInfo( )** to obtain the error information.

The user-supplied pointer is returned with either event and can be retrieved using **sr_getUserContext( )**. The pointer to the DM_CONNECT_STATUS_LIST structure is returned with either event and can be retrieved using **sr_getevtdatap( )**.

For more information on SRL functions, see the Dialogic® Standard Runtime Library documentation.

■ Multimedia Scenario

The following describes how to establish full-duplex audio and video connections between two devices. In this example, the two devices are the multimedia (MM) device and the IP media (IPM) device. CNF multimedia conferencing and 3G-324M (M3G) devices can be substituted in the example.

Note: These ports are not the external IPM ports that are transmitting/receiving RTP packets; they are internal connections. Therefore, when receiving RTP packets from the outside world, an IPM device will transmit this data internally to another device (in this case an MM device).

- Use **dev_GetTransmitPortInfo( )** and **dev_GetReceivePortInfo( )** to retrieve the internal transmit port and the internal receive port information for the MM device.
- Use **dev_GetTransmitPortInfo( )** and **dev_GetReceivePortInfo( )** to retrieve the internal transmit port and internal receive port information for the IPM device.
- Create a full-duplex connection by calling **dev_PortConnect( )** twice: first to create the connections from the internal transmit ports of the MM device to the internal receive ports of the IPM device, and then again to create the connections from the internal transmit ports of the IPM device to the internal receive ports of the MM device. Indicate the connection type (native or transcoding) in DM_PORT_CONNECT_INFO.

■ Supported Connections

The **dev_PortConnect( )** function can create connections between devices including:

Multimedia and IP Media
: A half-duplex connection between an internal port of an IP media device and an internal port of a multimedia device. Requires a valid IP media device handle obtained through **ipm_Open( )** and a valid multimedia device handle obtained through **mm_Open( )**.

IP Media and IP Media
: A half-duplex connection between internal ports of two IP media devices. Requires a valid IP media device handle obtained through **ipm_Open( )**. Used for hairpinning.

M3G and IP Media
: A half-duplex connection between an internal port of an IP media device and an internal port of a 3G-324M (M3G) device (audio or video). Requires a valid IP media device handle obtained through **ipm_Open( )** and a valid M3G device handle obtained through **m3g_Open( )**.

M3G and Multimedia
: A half-duplex connection between an internal port of a multimedia device and an internal port of a 3G-324M (M3G) device (audio or video). Requires a valid multimedia device handle obtained through **mm_Open( )** and a valid M3G device handle obtained through **m3g_Open( )**.

M3G and CNF Multimedia Conferencing
: A half-duplex connection between an internal port of a 3G-324M (M3G) device and an internal port of a multimedia conferencing device. Requires a valid multimedia conferencing party device handle (MCX) obtained through **cnf_OpenParty( )** and a valid M3G device handle obtained through **m3g_Open( )**.

M3G and M3G
: A half-duplex connection between internal ports of two 3G-324M (M3G) devices (audio or video). Requires valid M3G device handles obtained through **m3g_Open( )**. Used for hairpinning.

The connection types that are supported, native or transcoding, vary by software release. The connection type is specified in the unFlags field of the DM_PORT_CONNECT_INFO structure. The media type such as audio or video is specified in the port_media_type field of the DM_PORT_INFO structure. In the table, HMP 3.0WIN refers to Dialogic® HMP Software 3.0WIN, HMP 3.1LIN refers to Dialogic® HMP Software 3.1LIN, HMP 4.1LIN refers to Dialogic® HMP Software 4.1LIN, MMP for ATCA refers to Dialogic® Multimedia Platform for ATCA, and MMK for PCIe refers to Dialogic® Multimedia Kit for PCIe.

Connection Type	HMP 3.0WIN	HMP 3.1LIN	HMP 4.1LIN	MMK 1.0 for PCIe	MMP 2.0 for ATCA	MMP 1.1 for ATCA
Native audio	S	S	S	S	S	S
Native video	S	S	S	S	S	S
Transcoding audio	N	N	S	S	S	N
Transcoding video	N	N	S	S	S	N

Legend: S=Supported, N=Not supported

Cautions

- The **dev_PortConnect( )** function must be called from the same process that opens the device and obtains the device handle used in the function.
- A call to **dev_PortConnect( )** must complete, as indicated by the termination event, before a second **dev_PortConnect( )** call can be made successfully on the same device; otherwise, the second connection results in an EDEV_INVALIDSTATE error.
- If **dev_PortConnect( )** is unable to complete one or more connections defined by the source and destination DM_PORT_INFO_LIST structures, the function returns the DMEV_PORT_CONNECT_FAIL event. Connections that were successfully completed, however, will not be automatically disconnected. The application can check the status of each connection by retrieving and examining the DM_CONNECT_STATUS_LIST structure.
- If **dev_PortConnect( )** is called on device A and a connection is made to destination port X (of device B), and then **dev_PortConnect( )** is called on device C and a second connection is also made to destination port X (of device B), the data received by device B may be corrupted. The first connection made from device A to port X is not implicitly disconnected when the second **dev_PortConnect( )** call is made.

Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information. Possible errors for this function include:

EDEV_BADPARM
Invalid argument or parameter

EDEV_INVALIDDEVICEHANDLE
Invalid device handle specified

EDEV_SUBSYSTEMERR
Internal error

Example

This example illustrates a half-duplex connection between two devices. The internal transmit ports of ipmB1C2 are connected to the internal receive ports of ipmB1C1.

```
#include <srllib.h>
#include <ipmlib.h>
#include <port_connect.h>
#include <string.h>
#include <iostream>

using namespace std;

int CreateConnectInfoList(
     PDM_PORT_CONNECT_INFO_LIST pconn_lst,
     CPDM_PORT_INFO_LIST pport_lst1,
     CPDM_PORT_INFO_LIST pport_lst2
     )
{
     INIT_DM_PORT_CONNECT_INFO_LIST(&pconn_lst);
     // Loop through all transmit ports of 1st device
     int k = 0;
     int i;
     for (i = 0; i < pport_lst1->unCount; ++i) {
          DM_PORT_MEDIA_TYPE type_tx =
               pport_lst1->port_info[i].port_media_type;
          // find appropriate RX port on 2nd device
          bool bFound = false;
          int j;
          for (j = 0; j < pport_lst2->unCount; ++j) {
               DM_PORT_MEDIA_TYPE type_rx =
                    pport_lst2->port_info[j].port_media_type;
               if (type_tx == type_rx) {
                    bFound = true;
                    break;
               }
          }
          if (!bFound) {
               continue;
          }
          // create element of connect list
          // Check the transcoding support for video (DM_PORT_MEDIA_TYPE_VIDEO) in the software
             release before setting unFlags to DMFL_TRANSCODE_ON. Only set if video transcoding
             is supported.

          if (type_tx == DM_PORT_MEDIA_TYPE_AUDIO)
              info.unFlags = DMFL_TRANSCODE_ON;
          else
              info.unFlags = DMFL_TRANSCODE_NATIVE;
          info.port_info_tx = pport_lst1->port_info[i];
          info.port_info_rx = pport_lst2->port_info[j];
          ++k;
     }
     pconn_lst->unCount = k;
     return k;
}

int main(int argc, char** argv)
{
     int ret;
     int rc;
     int dev1, dev2;
     long evt;
     void* evt_data;
     int evt_len;
     const char szDev1[] = "ipmB1C1";
     const char szDev2[] = "ipmB1C2";

     ret = 0;
     dev1 = -1;
     try
     {

     // Open device (ipmB1C1)
     dev1 = ipm_Open(szDev1, NULL, EV_ASYNC);
     if (-1 == dev1) {
          cout << "ipm_Open error";
          cout << " handle = "  << dev1 << endl;
          throw 11;
     }
     sr_waitevt(-1);
     evt = sr_getevttype();
     if (IPMEV_OPEN != evt) {
          cout << "ipm_Open error";
          cout << " event = " << evt << endl;
          throw 12;
     }

     // Open device (ipmB1C2)
     dev2 = ipm_Open(szDev2, NULL, EV_ASYNC);
     if (-1 == dev2) {
```

```
     cout << "ipm_Open error";
     cout << " handle = "  << dev2 << endl;
     throw 21;
}
sr_waitevt(-1);
evt = sr_getevttype();
if (IPMEV_OPEN != evt) {
     cout << "ipm_Open error";
     cout << " event = " << evt << endl;
     throw 22;
}

// Obtain Device 1 Receive Ports
rc = dev_GetReceivePortInfo(dev1, NULL);
if (-1 == rc) {
     cout << "dev_GetReceivePortInfo error";
     cout << " rc = " << rc << endl;
     throw 13;
}
sr_waitevt(-1);
evt = sr_getevttype();
if (DMEV_GET_RX_PORT_INFO != evt) {
     cout << "dev_GetReceivePortInfo error";
     cout << " event = " << evt << endl;
     throw 14;
}
evt_data = sr_getevtdatap();
evt_len = sr_getevtlen();
DM_PORT_INFO_LIST port_info_list1 = {};
memcpy(&port_info_list1, evt_data, evt_len);

// Print number of ports
cout << "Number of RX ports: "
<< port_info_list1.unCount << endl;

// Obtain Device 2 Transmit Ports
rc = dev_GetTransmitPortInfo(dev2, NULL);
if (-1 == rc) {
     cout << "dev_GetTransmitPortInfo error";
     cout << " rc = " << rc << endl;
     throw 23;
}
sr_waitevt(-1);
evt = sr_getevttype();
if (DMEV_GET_TX_PORT_INFO != evt) {
     cout << "dev_GetTransmitPortInfo error";
     cout << " event = " << evt << endl;
     throw 24;
}
evt_data = sr_getevtdatap();
evt_len = sr_getevtlen();
DM_PORT_INFO_LIST port_info_list2 = {};
memcpy(&port_info_list2, evt_data, evt_len);

DM_PORT_CONNECT_INFO_LIST connectList;
int num_matched_ports;
num_matched_ports = CreateConnectInfoList(
     &connectList,
     &port_info_list2,
     &port_info_list1
);
if (0 == num_matched_ports) {
     cout << "No matched ports found" << endl;
     throw 50;
}
// Connect transmit ports of dev2 to receive ports of dev1
rc = dev_PortConnect(dev2, &connectList, 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;
        }
        /* Ports now connected */

        // Disconnect transmit ports of dev2 from receive ports of dev1
        rc = dev_PortDisconnect(dev2, &connectList, NULL);
        if (-1 == rc) {
             cout << "dev_PortDisconnect error";
             cout << " rc = " << rc << endl;
             throw 61;
        }
        sr_waitevt(-1);
        evt = sr_getevttype();
        if (DMEV_PORT_DISCONNECT != evt) {
             cout << "dev_PortDisconnect error";
             cout << " event = " << evt << endl;
             throw 62;
        }
        /* Ports now disconnected */

        }
        catch (int point) {
             ret = -1;
             cerr << "Error point #" << point << " reached" << endl;
        }

        if (dev1 != -1) {
             rc = ipm_Close(dev1, NULL);
             dev1 = -1;
        }

        if (dev2 != -1) {
             rc = ipm_Close(dev2, NULL);
             dev2 = -1;
        }

        return ret;
   }
```

■ See Also

- **dev_PortDisconnect( )**

dev_PortDisconnect( )

Name:	dev_PortDisconnect (devHandle, pConnectList, pUserContext)	
Inputs:	int devHandle	• a valid channel device
	CPDM_PORT_CONNECT_INFO_LIST pConnectList	• a pointer to the list of connection structures
	void *pUserContext	• a pointer to user-specific context
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h port_connect.h	
Category:	Device Connection	
Mode:	asynchronous	

■ Description

The **dev_PortDisconnect( )** function severs connections between one or more internal transmit ports of the specified device and internal receive ports of another device or the same device, based on the contents of the connection structures.

Parameter	Description
devHandle	specifies a valid channel device handle obtained when the channel was opened
pConnectList	specifies a pointer to the list of connection structures, DM_PORT_CONNECT_INFO_LIST
pUserContext	specifies a user-supplied pointer that can be retrieved using **sr_getUserContext( )** when the completion event is received

■ Asynchronous Operation

The function returns DEV_SUCCESS to indicate it has initiated successfully. The function generates a DMEV_PORT_DISCONNECT event to indicate successful completion of the function operation. Use the Dialogic® Standard Runtime Library (SRL) functions to process the termination event.

This function generates a DMEV_PORT_DISCONNECT_FAIL event to indicate failure of the function operation. Use **dev_GetResultInfo( )** to obtain the error information.

The user-supplied pointer is returned with either event and can be retrieved using **sr_getUserContext( )**. For more information on this function, see the *Dialogic® Standard Runtime Library API Library Reference*.

Cautions

- The **dev_PortDisconnect( )** function must be called from the same process that opens the device and obtains the device handle used in the function.
- In a full-duplex connection, when disconnecting, call **dev_PortDisconnect( )** twice: once to disconnect the transmit ports of device A from the receive ports of device B, and then again to disconnect the transmit ports of device B from the receive ports of device A.

Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information. Possible errors for this function include:

EDEV_BADPARM
: Invalid argument or parameter

EDEV_INVALIDDEVICEHANDLE
: Invalid device handle specified

EDEV_SUBSYSTEMERR
: Internal error

Example

For an example, see **dev_PortConnect( )**.

See Also

- **dev_PortConnect( )**

dev_ReleaseResource( )

Name:	int dev_ReleaseResource (devHandle, resType, mode)	
Inputs:	int devHandle	• a valid channel device
	eDEV_RESOURCE_TYPE resType	• a resource type
	unsigned short mode	• synchronous function mode
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h	
Category:	Resource Reservation	
Mode:	synchronous	

■ Description

Note: The **dev_ReleaseResourceEx()** function supercedes and should be used instead of the **dev_ReleaseResource()** function. The _Ex( ) function provides improved information about available resources.

The **dev_ReleaseResource( )** function releases a specified resource previously reserved for the device. When you release a resource, it returns to the pool of available resources.

Parameter	Description
devHandle	specifies a valid channel device handle obtained when the channel was opened
resType	specifies a resource type. The following is the only valid value: • **RESOURCE_IPM_LBR** – specifies the resource for IP media low bit rate codecs (e.g., G.723 or G.729). A board device handle is not valid when using this resource type; the device handle must be a valid IP media channel device. This resource type is supported in synchronous mode only.
mode	specifies how the function should be executed. For resource type **RESOURCE_IPM_LBR**, set this to: • EV_SYNC – synchronously

■ Synchronous Operation

Resource Reservation operations on the low bit rate codec are handled in a single thread of execution; therefore, resource type **RESOURCE_IPM_LBR** is supported in **synchronous mode** only.

To run this function synchronously, set the mode parameter to EV_SYNC. This function returns 0 to indicate successful completion and -1 to indicate failure. Use **dev_ErrorInfo( )** to retrieve the error information.

■ Cautions

- This function requires that the device be open and that it have a resource of the specified type reserved for it; otherwise, it generates a subsystem error (e.g., EDEV_IPM_SUBSYSTEMERR).
- If the specified resource is actively being used, it cannot be released and generates a subsystem error (e.g., EDEV_IPM_SUBSYSTEMERR).
- Resource type **RESOURCE_IPM_LBR** is not supported in **asynchronous mode** and will not generate the necessary events.
- If you use this function to release the **RESOURCE_IPM_LBR** resource multiple times for the same device (without reserving the resource again), it is ignored. It does not return an error or change the resource pool allocation.
- If you close the device, it releases all resources reserved for it.

■ Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information. Possible errors for this function include:

EDEV_INVALIDDEVICEHANDLE
: An invalid device handle was specified. For the **dev_Connect( )** function, the Supported Connections do not allow connection of these types of devices. (Valid handles include IP media, multimedia, and T.38 UDP fax devices.)

EDEV_INVALIDMODE
: An invalid **mode** was specified for executing the function synchronously or asynchronously (EV_SYNC or EV_ASYNC).

EDEV_IPM_SUBSYSTEMERR
: A subsystem error occurred during an internal call to an IP media library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error). See the IP media library documentation for the IP media error codes and descriptions.

See also Chapter 5, “Error Codes” for additional information.

■ Example

The following example code shows how the function is used in synchronous mode.

```
#include "srllib.h"
#include "ipmlib.h"
#include "devmgmt.h"

void main()
{
   int devHandle; // channel handle
   .
   .

   // Open channel
   if ((devHandle = ipm_Open("ipmB1C1", NULL, EV_SYNC)) == -1)
   {
      printf("Cannot open channel\n");
      exit(1);
   }

   // UnReserve Low Bit Rate Codec for the specified channel
   if (dev_ReleaseResource(devHandle, RESOURCE_IPM_LBR, EV_SYNC) ==-1)
   {
      printf("Cannot Release LBR resource.\n");
      // Perform system error processing
   }
}
```

■ See Also

None.

dev_ReleaseResourceEx( )

Name:	int dev_ReleaseResourceEx(devHandle, pResourceList, mode)	
Inputs:	int devHandle	• valid channel device
	DEV_RESOURCE_LIST *pResourceList	• pointer to resource reservation list structure
	unsigned short mode	• asynchronous or synchronous function mode
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h	
Category:	Resource Reservation	
Mode:	asynchronous or synchronous	

■ Description

The **dev_ReleaseResourceEx( )** function releases specified resource(s) previously reserved for the device. When you release a resource, it returns to the pool of available resources.

When using any of the RESOURCE_IPM types, the IPM device must be stopped when issuing this API call.

Note: The **dev_ReleaseResourceEx()** function supercedes and should be used instead of the **dev_ReleaseResource()** function. The _Ex( ) function provides improved information about available resources.

Note: The **dev_ReleaseResourceEx( )** function is not supported in Dialogic® HMP Software 3.0WIN.

Parameter	Description
devHandle	specifies a valid channel device handle obtained when the channel was opened
pResourceList	pointer to resource reservation structure DEV_RESOURCE_LIST. When using any of the RESOURCE_IPM types, a board device handle is not valid; the device handle must be a valid IP media channel device.
mode	specifies how the function should be executed. Set this to one of the following: • EV_ASYNC - asynchronously • EV_SYNC - synchronously

■ Asynchronous Operation

To run this function asynchronously, set the mode parameter to EV_ASYNC. The function returns 0 to indicate it has initiated successfully. The function generates a

DMEV_RELEASE_RESOURCE termination event to indicate successful completion of the function operation. The application must process for the completion event that indicates the operation was successful. Use the Dialogic® Standard Runtime Library (SRL) functions to process the termination event.

This function generates a DMEV_RELEASE_RESOURCE _FAIL error event to indicate failure of the function operation. Use **dev_GetResultInfo( )** to retrieve the error information.

■ Synchronous Operation

To run this function synchronously, set the mode parameter to EV_SYNC. This function returns 0 to indicate successful completion and -1 to indicate failure. Use **dev_ErrorInfo( )** to retrieve the error information.

Note: It is better to use asynchronous mode because **dev_ReleaseResourceEx( )** is not executed in a single thread of execution as is **dev_ReleaseResource( )**.

■ Cautions

- This function requires that the device be open; otherwise, it generates a subsystem error (e.g., EDEV_IPM_SUBSYSTEMERR).
- If the specified resource is invalid, it generates a subsystem error (e.g., EDEV_IPM_SUBSYSTEMERR).
- If you use this function to release the RESOURCE_IPM_xxx resources multiple times for the same device (without reserving the resource again), it is ignored. It does not return an error or change the resource pool allocation.
- This function requires that the IPM device be idle when the call is issued. Otherwise it generates a subsystem error (e.g., EDEV_IPM_SUBSYSTEMERR).

■ Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information.

Possible errors for this function include:

EDEV_INVALIDDEVICEHANDLE
: An invalid device handle was specified.

EDEV_INVALIDMODE
: An invalid **mode** was specified for executing the function synchronously or asynchronously.

EDEV_IPM_SUBSYSTEMERR
: A subsystem error occurred during an internal call to an IP media library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error).

■ Asynchronous Code Example

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

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

     switch(nEventType)
     {
     /*
     .
     .
     . Other events
     .
     .
     */
                     case DMEV_RELEASE_RESOURCE:
          printf("Received DMEV_RELEASE_RESOURCE for device name = %s\n",
          ATDV_NAMEP(nDeviceID));

                       break;

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

     }
return 0;
}
int main()
{
int devHandle; // channel handle
DEV_RESOURCE_LIST devResourceList;
IPM_MEDIA_INFO MediaInfo;

// Register event handler function with srl
sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,CheckEvent);

// Open channel
if ((devHandle = ipm_Open("ipmB1C1", NULL, EV_SYNC)) == -1)
{
        printf("Cannot open channel\n");
        // Perform system error processing
        return -1;
}

/*
. .
Main Processing
. .
*/
/*
Release G726 G729 coders for IP device handle, nDeviceHandle.
ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
Coders were previously reserved.
*/

INIT_DEV_RESOURCE_LIST(&devResourceList);
devResourceList.rsList[0] = RESOURCE_IPM_G729;
devResourceList.rsList[1] = RESOURCE_IPM_G726;
devResourceList.count = 2;
// Release Codec for the specified channel
        if (dev_ReleaseResourceEx(devHandle, &devResourceList, EV_ASYNC) ==-1)
        {
                printf("Cannot Release Coder resources.\n");
                return 1;

        // Perform system error processing
        }else
                printf("Release succeeded.\n");

        /* Continue processing */

return 0;
}
```

Synchronous Code Example

```
#include "srllib.h"
#include "ipmlib.h"
#include "devmgmt.h"
void main()
{
        int devHandle; // channel handle
        DEV_RESOURCE_LIST devResList;

        // ASSUMPTION: devHandle is a valid device handle obtained from a previous ipm_Open call

        INIT_DEV_RESOURCE_LIST(&devResourceList);
        devResourceList.rsList[0] = RESOURCE_IPM_G726;
        devResourceList.rsList[1] = RESOURCE_IPM_G729;
        devResourceList.count = 2;

        // UnReserve Low Bit Rate Codec for the specified channel

        if (dev_ReleaseResourceEx(devHandle, &devResourceList, EV_SYNC) ==-1)
        {
             printf("Cannot Release resources.\n");
             // Perform system error processing
        }
        else
         printf("Release succeeded.\n");

        /*
        . .
        . Continue processing
        ..
        . .
        */

}
```

See Also

- **dev_GetResourceReservationInfoEx( )**
- **dev_ReserveResourceEx( )**

dev_ReserveResource( )

Name:	int dev_ReserveResource (devHandle, resType, mode)	
Inputs:	int devHandle	• a valid channel device
	eDEV_RESOURCE_TYPE resType	• a resource type
	unsigned short mode	• synchronous function mode
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h	
Category:	Resource Reservation	
Mode:	synchronous	

■ Description

Note: The **dev_ReserveResourceEx( )** function supercedes and should be used instead of the **dev_ReserveResource()** function. The _Ex( ) function provides improved information about available resources.

The **dev_ReserveResource( )** function reserves a resource for use by the specified device. This allows an application to reserve resources during initial setup and can be especially useful for complex setups, where the setup might fail during an intermediate step for lack of a critical resource. In such cases, it is sometimes necessary to backtrack and then retry the operation with an alternate resource. Reserving the resource before-hand ensures that the dependency on the resource is met before proceeding with the setup.

Parameter	Description
devHandle	specifies a valid channel device handle obtained when the channel was opened
resType	specifies a resource type. The following is the only valid value: • **RESOURCE_IPM_LBR** – specifies the resource for IP media low bit rate codecs (e.g., G.723 or G.729). A board device handle is not valid when using this resource type; the device handle must be a valid IP media channel device. This resource type is supported in synchronous mode only.
mode	specifies how the function should be executed. For resource type **RESOURCE_IPM_LBR**, set this to: • EV_SYNC – synchronously

Synchronous Operation

Resource Reservation operations on the low bit rate codec are handled in a single thread of execution; therefore, resource type **RESOURCE_IPM_LBR** is supported in **synchronous mode** only.

To run this function synchronously, set the mode parameter to EV_SYNC. This function returns 0 to indicate successful completion and -1 to indicate failure. Use **dev_ErrorInfo( )** to retrieve the error information.

Cautions

- If you use this function to reserve the **RESOURCE_IPM_LBR** resource multiple times for the same device (without releasing the resource), it is ignored. It does not return an error or change the resource pool allocation.
- This function requires that the device be open; otherwise, it generates a subsystem error (e.g., EDEV_IPM_SUBSYSTEMERR).
- If no resource of the specified type is available, it generates a subsystem error (e.g., EDEV_IPM_SUBSYSTEMERR).
- If you close the device, it releases all resources reserved for it.

Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information. Possible errors for this function include:

EDEV_INVALIDDEVICEHANDLE
: An invalid device handle was specified. For the **dev_Connect( )** function, the Supported Connections do not allow connection of these types of devices. (Valid handles include IP media, multimedia, and T.38 UDP fax devices.)

EDEV_INVALIDMODE
: An invalid **mode** was specified for executing the function synchronously or asynchronously (EV_SYNC or EV_ASYNC).

EDEV_IPM_SUBSYSTEMERR
: A subsystem error occurred during an internal call to an IP media library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error). See the IP media library documentation for the IP media error codes and descriptions.

See also Chapter 5, “Error Codes” for additional information.

Example

The following example code shows how the function is used in synchronous mode.

```
#include "srllib.h"
#include "ipmlib.h"
#include "devmgmt.h"

void main()
{
   int devHandle;       // channel handle
```

```
    .
    .
    // Open channel
    if ((devHandle = ipm_Open("ipmB1C1", NULL, EV_SYNC)) == -1)
    {
       printf("Cannot open channel\n");
       // Perform system error processing
       exit(1);
    }

    // Reserve Low Bit Rate Codec for the specified channel
    if (dev_ReserveResource(devHandle, RESOURCE_IPM_LBR, EV_SYNC) ==-1)
    {
       printf("Cannot Reserve LBR resource.\n");
       // Perform system error processing
    }
}
```

■ See Also

None.

dev_ReserveResourceEx( )

Name:	int dev_ReserveResourceEx(devHandle, pResourceList, mode)	
Inputs:	int devHandle	• valid channel device
	DEV_RESOURCE_LIST *pResourceList	• pointer to resource reservation list structure
	unsigned short mode	• asynchronous or synchronous function mode
Returns:	DEV_SUCCESS if successful -1 if failure	
Includes:	srllib.h devmgmt.h	
Category:	Resource Reservation	
Mode:	asynchronous or synchronous	

■ Description

The **dev_ReserveResourceEx( )** function reserves resource(s) for use by the specified device. This allows an application to reserve resources during initial setup and can be especially useful for complex setups, where the setup might fail during an intermediate step for lack of a critical resource. In such cases, it is sometimes necessary to backtrack and then retry the operation with an alternate resource. Reserving the resource(s) beforehand ensures that the dependency on the resource is met before proceeding with the setup.

When using any of the RESOURCE_IPM types, the IPM device must be idle when issuing this API call. If there is an ongoing streaming operation, it must either be completed or stopped prior to issuing this API call. Also, the application must call **ipm_GetLocalMediaInfo( )** after every **dev_ReserveResourceEx( )** call to RESOURCE_IPM types.

Note: The **dev_ReserveResourceEx()** function supercedes and should be used instead of the **dev_ReserveResource()** function. The _Ex( ) function provides improved information about available resources.

Note: The **dev_ReserveResourceEx( )** function is not supported in Dialogic® HMP Software 3.0WIN.

Parameter	Description
devHandle	specifies a valid channel device handle obtained when the channel was opened

Parameter	Description
pResourceList	pointer to resource reservation structure DEV_RESOURCE_LIST. When using any of the RESOURCE_IPM types, a board device handle is not valid; the device handle must be a valid IP media channel device.
mode	specifies how the function should be executed. Set this to one of the following: • EV_ASYNC - asynchronously • EV_SYNC - synchronously

Asynchronous Operation

To run this function asynchronously, set the mode parameter to EV_ASYNC. The function returns 0 to indicate it has initiated successfully. The function generates a DMEV_RESERVE_RESOURCE termination event to indicate successful completion of the function operation. The application must process for the completion event that indicates the operation was successful. Use the Dialogic® Standard Runtime Library (SRL) functions to process the termination event.

This function generates a DMEV_RESERVE_RESOURCE _FAIL error event to indicate failure of the function operation. Use **dev_GetResultInfo( )** to retrieve the error information.

Synchronous Operation

To run this function synchronously, set the mode parameter to EV_SYNC. This function returns 0 to indicate successful completion and -1 to indicate failure. Use **dev_ErrorInfo( )** to retrieve the error information.

It is better to use asynchronous mode because **dev_ReserveResourceEx( )** is not executed in a single thread of execution as is **dev_ReserveResource( )**.

Cautions

- The coders specified in a call to **dev_ReserveResourceEx( )** override the set of coders previously reserved when the function completes successfully. When the call fails, the set of reserved coders obtained through the previous successful call is still valid.
- The application must call **dev_ReleaseResourceEx( )** to release resources. It can use RESOURCE_IPM_ALL_AUDIO_CODERS to release all audio coders that it has reserved without having to list every single one of them.
- This function requires that the device be open; otherwise, it generates a subsystem error (e.g., EDEV_IPM_SUBSYSTEMERR).
- If the specified resource is unavailable or invalid, it generates a subsystem error (e.g., EDEV_IPM_SUBSYSTEMERR). Use **dev_ErrorInfo( )** to obtain the technology-specific error code.
- The current call to reserve a set of coders replaces any set of coders that were previously reserved. Therefore, the application must send the complete list of coders it needs to reserve.
- This function requires that the IPM device be idle when the call is issued. Otherwise it generates an error (e.g., EDEV_IPM_SUBSYSTEMERR).

- The application should always clean up resources before exiting. The application should stop the RTP session by using **ipm_Stop( )**. Otherwise the next startup will result in **dev_ReserveResourceEx( )** failure.

Errors

If this function returns -1 to indicate failure, use **dev_ErrorInfo( )** to retrieve the error information.

Possible errors for this function include:

EDEV_INVALIDDEVICEHANDLE
: An invalid device handle was specified.

EDEV_INVALIDMODE
: An invalid **mode** was specified for executing the function synchronously or asynchronously.

EDEV_IPM_SUBSYSTEMERR
: A subsystem error occurred during an internal call to an IP media library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error).

EIPM_RESOURCESINUSE
: A resource in use error is returned if all IPM coder resources are in use and not available for reservation.

Asynchronous Code Example

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

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

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

     /* Expected reply to dev_ReserveResourceEx */
     case DMEV_RESERVE_RESOURCE:
          printf("Received DMEV_RESERVE_RESOURCE for device name = %s\n",
          ATDV_NAMEP(nDeviceID));
```

```
            /* It is necessary to call ipm_GetLocalMediaInfo() after a call
            * to dev_ReserveResourceEx().
            * Get the local IP information for IP device handle,
            * nDeviceHandle.
            * ASSUMPTION: A valid nDeviceHandle was obtained from prior call
            * to ipm_Open().
            */

            MediaInfo.unCount = 1;
            MediaInfo.MediaData[0].eMediaType = MEDIATYPE_LOCAL_RTP_INFO;
            if(ipm_GetLocalMediaInfo(nDeviceID, &MediaInfo, EV_ASYNC) == - 1)
            {
            printf("ipm_GetLocalMediaInfo failed for device name %s with error = %d\n",
            ATDV_NAMEP(nDeviceID), ATDV_LASTERR(nDeviceID));
            /*
            . .
            Perform Error Processing
            . .
            */
            }

        break;
/* 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_RTCP_INFO:
                       printf("MediaType=MEDIATYPE_AUDIO_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;
                  }
            }

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

```
}
int main()
{

int devHandle; // channel handle
DEV_RESOURCE_LIST devResourceList;

// Register event handler function with srl
sr_enbhdlr( EV_ANYDEV ,EV_ANYEVT ,CheckEvent);

// Open channel
if ((devHandle = ipm_Open("ipmB1C1", NULL, EV_SYNC)) == -1)
{
         printf("Cannot open channel\n");
         // Perform system error processing
         return -1;
}

/*
. .
Main Processing
. .
*/
/*
Reserve G726 G729 coders for IP device handle, nDeviceHandle.
ASSUMPTION: A valid nDeviceHandle was obtained from prior call to ipm_Open().
*/

INIT_DEV_RESOURCE_LIST(&devResourceList);
devResourceList.rsList[0] = RESOURCE_IPM_G729;
devResourceList.rsList[1] = RESOURCE_IPM_G726;
devResourceList.count = 2;

// Reserve Low Bit Rate Codec for the specified channel
         if (dev_ReserveResourceEx(devHandle, &devResourceList, EV_ASYNC) ==-1)
         {
                 printf("Cannot Reserve LBR Coder resourceS.\n");
                 return 1;

         // Perform system error processing
         }else
                 printf("Reserve succeeded.\n");
        /* Continue processing */

return 0;
}
```

Synchronous Code Example

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

using namespace std;

int main()
{

int devHandle; // channel handle
DEV_RESOURCE_LIST devResourceList;
IPM_MEDIA_INFO MediaInfo;
// Open channel
if ((devHandle = ipm_Open("ipmB1C1", NULL, EV_SYNC)) == -1)
{
//       printf("Cannot open channel\n");
         // Perform system error processing
         return -1;
}

INIT_DEV_RESOURCE_LIST(&devResourceList);
devResourceList.rsList[0] = RESOURCE_IPM_G729;
devResourceList.rsList[1] = RESOURCE_IPM_G726;
devResourceList.count = 2;

// Reserve Low Bit Rate Codec for the specified channel
if (dev_ReserveResourceEx(devHandle, &devResourceList, EV_SYNC) ==-1)
{
      printf("Cannot Reserve LBR Coder resourceS.\n");
// Perform system error processing
}else
      printf("Reserve succeeded.\n");

/* It is necessary to call ipm_GetLocalMediaInfo() after a call
* to dev_ReserveResourceEx().
* Get the local IP information for IP device handle,
* nDeviceHandle.
* ASSUMPTION: A valid nDeviceHandle was obtained from prior call * to ipm_Open().
*/

MediaInfo.unCount = 1;
MediaInfo.MediaData[0].eMediaType = MEDIATYPE_LOCAL_RTP_INFO;
if(ipm_GetLocalMediaInfo(devHandle, &MediaInfo, EV_SYNC) == -1)
{
      printf("ipm_GetLocalMediaInfo failed for device name %s with error =
      %d, %s\n", ATDV_NAMEP(devHandle), ATDV_LASTERR(devHandle),
      ATDV_ERRMSGP(devHandle));
// Perform system error processing

}else
      printf("GetLocalMediaInfo succeeded.\n");

/*

* Continue processing

*/

ipm_Close(devHandle, NULL);

return 0;
}
```

■ See Also

- **dev_GetResourceReservationInfoEx( )**
- **dev_ReleaseResourceEx( )**

Events 3

This chapter describes the events that are generated by the Dialogic® Device Management API functions.

3.1 Overview of Dialogic® Device Management API Events

When running in asynchronous mode, the functions in the Dialogic® Device Management API generate termination events to indicate the result of the function operation. Typically, each function generates a different set of events. The events applicable to a function are documented in Chapter 2, "Function Information".

The Dialogic® Device Management API events contain a "DMEV_" prefix and the failure events are typically identified by a "_FAIL" suffix; for example, DMEV_CONNECT_FAIL. No change of state is triggered by the failure event. If an error occurs during execution of an asynchronous function, a failure event is sent to the application. To retrieve error information for a failure event, use **dev_GetResultInfo( )**.

To collect termination event codes, use Dialogic® Standard Runtime Library (SRL) functions. For detailed information on event handling and management, see the Dialogic® Standard Runtime Library documentation.

3.2 Device Connection Events

The following events are generated by the Dialogic® Device Management API for the Device Connection functions:

DMEV_CONNECT
: Termination event generated for each device specified in the **dev_Connect( )** function to indicate successful completion of the function operation.

DMEV_CONNECT_FAIL
: Termination event generated for each device specified in the **dev_Connect( )** function to indicate failure of the function operation.

DMEV_DISCONNECT
: Termination event generated to indicate successful completion of the **dev_Disconnect( )** function operation.

DMEV_DISCONNECT_FAIL
: Termination event generated to indicate failure of the **dev_Disconnect( )** function operation.

DMEV_GET_RX_PORT_INFO
: Termination event generated to indicate successful completion of the **dev_GetReceivePortInfo( )** function operation.

DMEV_GET_RX_PORT_INFO_FAIL
: Termination event generated to indicate failure of the **dev_GetReceivePortInfo( )** function operation.

DMEV_GET_TX_PORT_INFO
: Termination event generated to indicate successful completion of the **dev_GetTransmitPortInfo( )** function operation.

DMEV_GET_TX_PORT_INFO_FAIL
: Termination event generated to indicate failure of the **dev_GetTransmitPortInfo( )** function operation.

DMEV_PORT_CONNECT
: Termination event generated to indicate successful completion of the **dev_PortConnect( )** function operation.

DMEV_PORT_CONNECT_FAIL
: Termination event generated to indicate failure of the **dev_PortConnect( )** function operation.

DMEV_PORT_DISCONNECT
: Termination event generated to indicate successful completion of the **dev_PortDisconnect( )** function operation.

DMEV_PORT_DISCONNECT_FAIL
: Termination event generated to indicate failure of the **dev_PortDisconnect( )** function operation.

3.3 Resource Reservation Events

The following events are generated by the Dialogic® Device Management API for the Resource Reservation functions:

DMEV_GET_RESOURCE_RESERVATIONINFO
: Termination event generated to indicate successful completion of the **dev_GetResourceReservationInfo( )** and **dev_GetResourceReservationInfoEx( )** function operations.

DMEV_GET_RESOURCE_RESERVATIONINFO_FAIL
: Termination event generated to indicate failure of the **dev_GetResourceReservationInfo( )** and **dev_GetResourceReservationInfoEx( )** function operations

DMEV_RELEASE_RESOURCE
: Termination event to indicate successful completion of the **dev_ReleaseResourceEx( )** function operation.

DMEV_RELEASE_RESOURCE _FAIL
: Termination event generated to indicate failure of the **dev_ReleaseResourceEx( )** function operation.

DMEV_RESERVE_RESOURCE
: Termination event to indicate successful completion of the **dev_ReserveResourceEx( )** function operation.

DMEV_RESERVE_RESOURCE_FAIL
: Termination event generated to indicate failure of the **dev_ReserveResourceEx( )** function operation.

Data Structures 4

This chapter provides information on the data structures used by Dialogic® Device Management API functions. The data structures are used to control the operation of functions and to return information. For each data structure, its definition is given, followed by details on its fields. The following data structures are included in this chapter:

DEV_ERRINFO

```
typedef struct errinfo
{
   int dev_ErrValue;
   int dev_SubSystemErrValue;
   char dev_Msg[DEV_MAXERRMSGSIZE];
} DEV_ERRINFO;
```

■ Description

The DEV_ERRINFO structure is used with the **dev_ErrorInfo( )** function to provide error information for the functions in the Device Management API.

■ Field Descriptions

The fields of the DEV_ERRINFO data structure are described as follows:

dev_ErrValue
: The error value returned for the last error generated by a Device Management API function call. The defines for the valid Device Management API error values are in the *devmgmt.h* header file and have a "EDEV_" prefix; also see Chapter 5, "Error Codes". If the error value returned indicates a subsystem error type, such as DEV_IPM_SUBSYSTEMERR or DEV_FAX_SUBSYSTEMERR, check the dev_SubSystemErrValue field to obtain the subsystem error value.

dev_SubSystemErrValue
: If the dev_ErrValue field indicates a subsystem error type, the dev_SubSystemErrValue field contains the error value returned by the subsystem for the last error generated by a Device Management API function call. The defines for the valid subsystem error values are in the technology-specific subsystem header file, which must be included in your program and used to identify the error. For example, if the dev_ErrValue field returns a DEV_IPM_SUBSYSTEMERR, indicating that an error occurred during an internal call to an IP media library function, the dev_SubSystemErrValue field returns an error value equivalent to an "EIPM_" error define from *ipmlib.h*.

dev_Msg
: The descriptive error message for the error. This is the Device Management API error description, unless dev_ErrValue reports a subsystem error, in which case it is the error description for the subsystem error code.

DEV_RESOURCE_LIST

```
typedef struct resourcelist
{       unsigned int version; // struct version
        int count;            // number of items filled in rslist
        eDEV_RESOURCE_TYPE rsList[MAX_CODER_RESOURCE_TYPE];
} DEV_RESOURCE_LIST;
```

■ Description

The DEV_RESOURCE_LIST structure is used by the **dev_ReserveResourceEx( )** and **dev_ReleaseResourceEx( )** functions to specify a list of coders to be reserved or released. The list of coders is specified in the array of enums rsList and the number entries filled in rsList is specified in the count field.

The INIT_DEV_RESOURCE_LIST inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the DEV_RESOURCE_LIST data structure are described as follows:

version
: The version number of the data structure. Use the inline function to initialize this field to the current version.

count
: The number of rsList elements to follow. Maximum number of coder resource types is defined in MAX_CODER_RESOURCE_TYPE.

rsList
: An array of eDEV_RESOURCE_TYPE elements.

DEV_RESOURCE_RESERVATIONINFO

```
typedef struct getresourceinfo
{
   unsigned int          version;             // struct version
   eDEV_RESOURCE_TYPE   resourceType;        // resource type
   int                  curReserveCount;     // current num. of resourceType reserved for device
   int                  curReservePoolCount; // current number of resourceType reserved in pool
   int                  maxReservePoolCount; // maximum number of resourceType available in pool
} DEV_RESOURCE_RESERVATIONINFO;
```

■ Description

The DEV_RESOURCE_RESERVATIONINFO structure is used with the **dev_GetResourceReservationInfo( )** function to provide resource reservation information.

The INIT_DEV_RESOURCE_RESERVATIONINFO inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the DEV_RESOURCE_RESERVATIONINFO data structure are described as follows:

version
: The version number of the data structure. Use the inline function to initialize this field to the current version.

resourceType
: The resource type for which the reservation information is returned in the data structure. The following is the only valid value:
  - **RESOURCE_IPM_LBR** – specifies the resource for IP media low bit rate codecs. A board device handle is not valid when using this resource type; the device handle must be a valid IP media channel device.

curReserveCount
: The current number of resourceType reserved for the device. Valid values:
  - 0 – No resource of resourceType is reserved for the device.
  - 1 – One resource of resourceType is reserved for the device.
  - n – The specified number of resources of resourceType are reserved for the device.

  Note: Some resource types, such as **RESOURCE_IPM_LBR**, do not permit reservation of more than one resource per device.

curReservePoolCount
: The number of system-wide resources of resourceType currently reserved for devices (that is, the number of reserved resources in the system resource pool).

maxReservePoolCount
: The maximum number of resources of resourceType allowed in the system. For Dialogic® Host Media Processing (HMP) software, the maximum number of **RESOURCE_IPM_LBR**

resources is specified through the Dialogic® HMP software License Manager. (If you change the setting, you must restart the Dialogic® HMP software for it to take effect.)

Note: The number of available system resources of resourceType can be calculated by subtracting curReservePoolCount from maxReservePoolCount.

DEV_RESOURCE_RESERVATIONINFO_EX

```
typedef struct getresourceinfo
{      unsigned int version; // struct version
       int count; // number of items filled in rsInfo
       ResourceInfo rsInfo[MAX_CODER_RESOURCE_TYPE];
} DEV_RESOURCE_RESERVATIONINFO_EX;
```

■ Description

The DEV_RESOURCE_RESERVATIONINFO_EX structure is used with the **dev_GetResourceReservationInfoEx( )** function to provide resource reservation information. See also resourceInfo structure.

The INIT_DEV_RESOURCE_RESERVATIONINFO_EX inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the DEV_RESOURCE_RESERVATIONINFO_EX data structure are described as follows:

version
: The version number of the data structure. Use the inline function to initialize this field to the current version.

count
: The number of resourceInfo data structures to follow. Maximum number of coder resource types is defined in MAX_CODER_RESOURCE_TYPE.

rsInfo
: An array of resourceInfo structures.

DM_CONNECT_STATUS_LIST

```
typedef struct DM_CONNECT_STATUS_LIST
{
     unsigned int         unVersion;
     unsigned int         unCount;
     CONNECT_STATUS       connect_status[MAX_DM_PORT_INFO];
   } DM_CONNECT_STATUS_LIST, *PDM_CONNECT_STATUS_LIST;

typedef const DM_CONNECT_STATUS_LIST* CPDM_CONNECT_STATUS_LIST;
```

■ Description

The DM_CONNECT_STATUS_LIST structure contains the status of each connection being created or severed. It is used with the **dev_PortConnect( )** and **dev_PortDisconnect( )** functions.

The INIT_DM_CONNECT_STATUS_LIST inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the DM_CONNECT_STATUS_LIST data structure are described as follows:

unVersion
: The version number of the data structure. Use the inline function to initialize this field to the current version.

unCount
: The number (1-n) of connect_status elements. Maximum number of values is defined in MAX_DM_PORT_INFO.

connect_status
: The pass or error condition array for each requested connection. Valid values:

- DM_STAT_UNKNOWN
- DM_STAT_CONNECT
- DM_STAT_DISCONNECT
- DM_STAT_CONNECT_FAIL
- DM_STAT_DISCONNECT_FAIL

DM_EVENT_INFO

```
typedef struct devinfo
{
     unsigned int  unVersion;
     int           nValue;
     int           nSubSystemValue;
     char          szMsg[DEV_MAXMSGSIZE];
     char          szSubSystemMsg[DEV_MAXMSGSIZE];
     char          szAdditionalInfo[DEV_MAXMSGSIZE];
} DM_EVENT_INFO, *PDM_EVENT_INFO;

typedef const DM_EVENT_INFO* CPDM_EVENT_INFO;
```

■ Description

The DM_EVENT_INFO data structure is used for transferring event-related information to the application. This is accomplished by passing the device management event data pointer returned by the **sr_getevtdatap( )**, **sr_getevtdatapex( )** or **sr_createevtdatapcopy( )** function to the **dev_GetResultInfo( )** function.

For more information about SRL functions, see the Standard Runtime Library documentation.

The INIT_DM_EVENT_INFO inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the DM_EVENT_INFO data structure are described as follows:

unVersion
: The version number of the data structure. Use the inline function to initialize this field to the current version.

nValue
: An integer code to represent information related to the state of the device management library at the time the event was generated. This may be a general purpose code or an error code. Valid values include:

- DM_EVENT_CODE_SUCCESS
- DM_EVENT_CODE_INVALID_DEVICE_HANDLE
- DM_EVENT_CODE_DX_SUBSYSTEMERR
- DM_EVENT_CODE_IPM_SUBSYSTEMERR
- DM_EVENT_CODE_CNF_SUBSYSTEMERR
- DM_EVENT_CODE_M3G_SUBSYSTEMERR
- DM_EVENT_CODE_MM_SUBSYSTEMERR
- DM_EVENT_CODE_DTI_SUBSYSTEMERR
- DM_EVENT_CODE_T38_SUBSYSTEMERR
- DM_EVENT_CODE_SUBSYSTEMERR
- DM_EVENT_CODE_INVALIDSTATE
- DM_EVENT_CODE_NOTCONNECTED
- DM_EVENT_CODE_MAX

nSubSystemValue
: An integer code to represent information related to the state of the library that owns the given device associated with the event. Values are specific to that library.

szMsg
: A null terminated string containing a translation of the integer code in nValue or meaningful phrase related to nValue.

szSubSystemMsg
: A null terminated string containing a translation of the integer code in nSubSystemValue or meaningful phrase related to nSubSystemValue.

szAdditionalInfo
: A null terminated string potentially containing a more descriptive statement related to the event or cause of the error related to a failure event.

DM_PORT_CONNECT_INFO

```
typedef struct
{
     unsigned int         unVersion;
     unsigned int         unFlags;
     DM_PORT_INFO         port_info_tx;
     DM_PORT_INFO         port_info_rx;
} DM_PORT_CONNECT_INFO, *PDM_PORT_CONNECT_INFO;

typedef const DM_PORT_CONNECT_INFO* CPDM_PORT_CONNECT_INFO;
```

■ Description

The DM_PORT_CONNECT_INFO structure specifies transmit and receive port information for a connection. This structure is a child structure of the DM_PORT_CONNECT_INFO_LIST structure.

The INIT_DM_PORT_CONNECT_INFO inline function is provided to initialize the structure.

■ Field Descriptions

The fields of DM_PORT_CONNECT_INFO data structure are described as follows:

unVersion
: The version number of the data structure. Use the inline function to initialize this field to the current version.

unFlags
: Flags specifying details of the connection to establish:

- DMFL_TRANSCODE_ON - default mode
- DMFL_TRANSCODE_NATIVE - native (no transcoding)

Note: The unFlags value is ignored for ports of type DM_PORT_MEDIA_TYPE_H223 (specified in DM_PORT_INFO). For ports of type DM_PORT_MEDIA_TYPE_VIDEO, make sure your software release supports video transcoding before setting the flag to DMFL_TRANSCODE_ON. See the Supported Connections section in **dev_PortConnect( )** or see the Release Guide for your software release for information on transcoding support. For software releases that do not support video transcoding, set the flag to DMFL_TRANSCODE_NATIVE.

port_info_tx
: Transmit port information, specified in the DM_PORT_INFO structure.

port_info_rx
: Receive port information, specified in the DM_PORT_INFO structure.

DM_PORT_CONNECT_INFO_LIST

```
typedef struct DM_PORT_CONNECT_INFO_LIST
{
     unsigned int          unVersion;
     unsigned int          unCount;
     DM_PORT_CONNECT_INFO   port_connect_info[MAX_DM_PORT_INFO];
} DM_PORT_CONNECT_INFO_LIST, *PDM_PORT_CONNECT_INFO_LIST;

typedef const DM_PORT_CONNECT_INFO_LIST* CPDM_PORT_CONNECT_INFO_LIST;
```

■ Description

The DM_PORT_CONNECT_INFO_LIST structure specifies a list of DM_PORT_CONNECT_INFO structures. It is used with the **dev_PortConnect( )** and **dev_PortDisconnect( )** functions.

The INIT_DM_PORT_CONNECT_INFO_LIST inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the DM_PORT_CONNECT_INFO_LIST data structure are described as follows:

unVersion
: The version number of the data structure. Use the inline function to initialize this field to the current version.

unCount
: The number (1-n) of port_connect_info elements to follow. Maximum number of structures is defined in MAX_DM_PORT_INFO.

port_connect_info
: An array of DM_PORT_CONNECT_INFO structures that specify the details of the connection to establish or tear down.

DM_PORT_INFO

```
typedef struct DM_PORT_INFO
{
     unsigned int              unVersion;
     DM_DEVICE_ID              device_ID;
     DM_PORT_ID                port_ID;
     DM_PORT_MEDIA_TYPE        port_media_type;
} DM_PORT_INFO, *PDM_PORT_INFO;

typedef const DM_PORT_INFO* CPDM_PORT_INFO;
```

■ Description

The DM_PORT_INFO structure contains details about the port used in the connection. It is a child structure of the DM_PORT_INFO_LIST structure.

The INIT_DM_PORT_INFO inline function is provided to initialize the structure.

■ Field Descriptions

The fields of the DM_PORT_INFO data structure are described as follows:

unVersion
: The version number of the data structure. Use the inline function to initialize this field to the current version.

device_ID
: Globally unique device ID which identifies a device. A value of DM_DEVICE_ID_NULL indicates an undefined device.

port_ID
: Locally unique port ID. A value of DM_PORT_ID_NULL indicates an undefined port.

  Note: This field should not be modified.

port_media_type
: Indicates the media type associated with the port. Valid values:

  - DM_PORT_MEDIA_TYPE_NONE
  - DM_PORT_MEDIA_TYPE_AUDIO
  - DM_PORT_MEDIA_TYPE_VIDEO
  - DM_PORT_MEDIA_TYPE_H223

  Note: The DM_PORT_MEDIA_TYPE_NBUP value is deprecated. Use the DM_PORT_MEDIA_TYPE_H223 value instead.

DM_PORT_INFO_LIST

```
typedef struct DM_PORT_INFO_LIST
{
     unsigned int              unVersion;
     unsigned int              unCount;
     DM_PORT_INFO              port_info[MAX_DM_PORT_INFO];
} DM_PORT_INFO_LIST, *PDM_PORT_INFO_LIST;

typedef const DM_PORT_INFO_LIST* CPDM_PORT_INFO_LIST;
```

■ Description

The DM_PORT_INFO_LIST structure specifies a list of DM_PORT_INFO structures. It is used with **dev_GetTransmitPortInfo( )** and **dev_GetReceivePortInfo( )** to return device port information.

The INIT_DM_PORT_INFO_LIST structure is provided to initialize the structure.

■ Field Descriptions

The fields of the DM_PORT_INFO_LIST data structure are described as follows:

unVersion
: The version number of the data structure. Use the inline function to initialize this field to the current version.

unCount
: The number (1-n) of port_info elements that follow.

port_info
: Refers to an array of DM_PORT_INFO data structures.

resourceInfo

```
typedef struct resourceInfo
{      eDEV_RESOURCE_TYPE resourceType; // resource type
       int curReserveCount; // current num. of resourceType reserved for device
       int availableResourceCount; // number of resourceType available in pool
} ResourceInfo;
```

Description

The resourceInfo structure is used within the DEV_RESOURCE_RESERVATIONINFO_EX structure, which is passed in the **dev_GetResourceReservationInfoEx( )** function to provide resource reservation information.

Field Descriptions

The fields of the resourceInfo data structure are described as follows:

resourceType
The resource type for which the reservation information is returned in the data structure resourceInfo. The valid values are as follows:

- RESOURCE_IPM_ALL_AUDIO_CODERS
- RESOURCE_IPM_G711_30MS
- RESOURCE_IPM_G711_20MS
- RESOURCE_IPM_G711_10MS
- RESOURCE_IPM_G723
- RESOURCE_IPM_G726
- RESOURCE_IPM_G729
- RESOURCE_IPM_AMR_NB
- RESOURCE_IPM_EVRC
- RESOURCE_IPM_GSM_EFR

A board device handle is not valid when using these resource types; the device handle must be a valid IP media channel device.

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 http://www.voiceage.com/licensing.php.

curReserveCount
The current number of resourceType reserved for the device. The following values are used:

- 0 – No resource of resourceType is reserved for the device.
- 1 – One resource of resourceType is reserved for the device.
- n – The specified number of resources of resourceType are reserved for the device.

Note: The RESOURCE_IPM_<type> resource types do not permit reservation of more than one resource per device.

availableResourceCount
The number of resources of resourceType available to be reserved in the system. This number depends on the resources reserved and used at runtime.

Error Codes 5

This chapter describes the error codes supported by the Dialogic® Device Management API.

The functions return a value indicating the outcome of the function operation. In most cases, the function returns DEV_SUCCESS (or 0) for a successful outcome and -1 for an unsuccessful outcome or an error. If a function fails, use **dev_ErrorInfo( )** to retrieve the error information.

If an error occurs during execution of an asynchronous function, a failure event is sent to the application. For more information on events, see Chapter 3, "Events".

Notes: ***1.*** Use **dev_ErrorInfo( )** only when a Dialogic® Device Management API function fails; otherwise, the data in the DEV_ERRINFO structure will be invalid.

2. If the error is a subsystem error, to identify the error code, you must include the header file for the technology-specific subsystem (for example, *ipmerror.h*, *ipmlib.h*, and *faxlib.h*).

3. The Dialogic® Device Management API errors are thread-specific (they are only in scope for that thread). Subsystem errors are device-specific.

The API contains the following error codes, listed in alphabetical order.

EDEV_DEVICEBUSY
: At least one of the devices specified is currently in use by another Dialogic® Device Management API function call. This can occur for the Device Connection functions.

EDEV_FAX_SUBSYSTEMERR
: A subsystem error occurred during an internal call to a fax library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error). This error may occur when calling the **dev_Connect( )** function if the connection to the fax device fails, or the **dev_Disconnect( )** function if the disconnection fails. See the fax library documentation for the fax error codes and descriptions.

EDEV_INVALIDCONNTYPE
: An invalid connection type (**connType**) was specified for the **dev_Connect( )** function (for example, T.38 UDP fax connection must be full-duplex).

EDEV_INVALIDDEVICEHANDLE
: An invalid device handle was specified for a Device Connection function or for a Resource Reservation function. For the **dev_Connect( )** function, the Supported Connections do not allow connection of the specified types of devices. Valid handles are listed in Supported Connections.

EDEV_INVALIDMODE
: An invalid **mode** was specified for a function that can be executed synchronously or asynchronously (EV_SYNC or EV_ASYNC).

EDEV_INVALIDSTATE
: Device is in an invalid state for the current function call. For example, the **dev_Disconnect( )** function may have been called before both devices were fully connected by the **dev_Connect( )** function.

EDEV_IPM_SUBSYSTEMERR
A subsystem error occurred during an internal call to an IP media library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error). This error may occur when calling the **dev_Connect( )** function if the connection to the IP media device fails, or the **dev_Disconnect( )** function if the disconnection fails. See the IP media library documentation for the IP media error codes and descriptions.

EDEV_MM_SUBSYSTEMERR
A subsystem error occurred during an internal call to a multimedia library function because the subsystem function was unable to start (this is not a Dialogic® Device Management API error). See the multimedia library documentation for the multimedia error codes and descriptions.

EDEV_NOTCONNECTED
An attempt was made to perform **dev_Disconnect( )** on a device that is not connected.