



# Dialogic<sup>®</sup> Device Management API

Library Reference

---

*August 2007*

Copyright © 2003 - 2007, Dialogic Corporation. All rights reserved. You may not reproduce this document in whole or in part without permission in writing from Dialogic Corporation.

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

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

Dialogic products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

It is possible that the use or implementation of any one of the concepts, applications, or ideas described in this document, in marketing collateral produced by or on web pages maintained by Dialogic may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic does not provide any intellectual property licenses with the sale of Dialogic products other than a license to use such product in accordance with intellectual property owned or validly licensed by Dialogic and no such licenses are provided except pursuant to a signed agreement with Dialogic. More detailed information about such intellectual property is available from Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. **Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.**

Dialogic, Diva, Eicon, Eicon Networks, Eiconcard and SIPcontrol, among others, are either registered trademarks or trademarks of Dialogic. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic's trademarks requires proper acknowledgement. Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries. Other names of actual companies and products mentioned herein are the trademarks of their respective owners.

Publication Date: August 2007

Document Number: 05-2222-007

# Contents

---

Revision History	5
<b>About This Publication</b>	9
Purpose	9
Applicability	9
Intended Audience	10
How to Use This Publication	10
Related Information	10
<b>1 Function Summary by Category</b>	11
1.1 Dialogic® Device Management API Header File	11
1.2 Device Connection Functions	11
1.3 Resource Reservation Functions	12
1.4 Error Processing Functions	12
<b>2 Function Information</b>	13
2.1 Function Syntax Conventions	13
dev_Connect() – connect devices	14
dev_Disconnect() – disconnect devices	31
dev_ErrorInfo() – get error information	34
dev_GetReceivePortInfo() – retrieve device receive ports information	36
dev_GetResourceReservationInfo() – get resource reservation information	39
dev_GetResourceReservationInfoEx() – get resource reservation information	42
dev_GetTransmitPortInfo() – retrieve device transmit ports information	46
dev_PortConnect() – create connections between transmit and receive ports	49
dev_PortDisconnect() – sever connections between transmit and receive ports	55
dev_ReleaseResource() – release a resource	57
dev_ReleaseResourceEx() – release a resource	60
dev_ReserveResource() – reserve a resource	64
dev_ReserveResourceEx() – reserve a resource	67
<b>3 Events</b>	73
3.1 Overview of Dialogic® Device Management API Events	73
3.2 Device Connection Events	73
3.3 Resource Reservation Events	74
<b>4 Data Structures</b>	77
DEV_ERRINFO – error information	78
DEV_RESOURCE_LIST – list of coders to be reserved or released	79
DEV_RESOURCE_RESERVATIONINFO – resource reservation information	80
DEV_RESOURCE_RESERVATIONINFO_EX – resource reservation information	82
DM_CONNECT_STATUS_LIST – status for port connections	83
DM_PORT_CONNECT_INFO – port connection information	84
DM_PORT_CONNECT_INFO_LIST – port connection information list	85

## **Contents**

	DM_PORT_INFO – port information . . . . .	86
	DM_PORT_INFO_LIST – port information list . . . . .	87
	resourceInfo – resource reservation information . . . . .	88
<b>5</b>	<b>Error Codes</b> . . . . .	<b>89</b>

# Revision History

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

Document No.	Publication Date	Description of Revisions
05-2222-007	August 2007	<p>Made global changes to reflect Dialogic brand.</p> <p><a href="#">Function Summary by Category</a> chapter: Removed table of function support by platform section.</p> <p><a href="#">dev_GetResourceReservationInfoEx()</a> function: Updated Platform line to show support for Dialogic® HMP Software; added note in Description section.</p> <p><a href="#">dev_ReleaseResourceEx()</a> function: Updated Platform line to show support for Dialogic® HMP Software; added note in Description section.</p> <p><a href="#">dev_ReleaseResourceEx()</a> function: Updated Platform line to show support for Dialogic® HMP Software; added note in Description section.</p>
05-2222-006	May 2007	<p><a href="#">Function Summary by Category</a> chapter: Added new functions to <a href="#">Device Connection Functions</a>. Added table of function support by platform.</p> <p><a href="#">Function Information</a> chapter: Added the following new Device Connection functions: <a href="#">dev_GetReceivePortInfo()</a>, <a href="#">dev_GetTransmitPortInfo()</a>, <a href="#">dev_PortConnect()</a> and <a href="#">dev_PortDisconnect()</a>.</p> <p><a href="#">dev_Connect()</a> 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.</p> <p><a href="#">dev_GetResourceReservationInfo()</a> function: Updated example code to include inline function INIT_DEV_RESOURCE_RESERVATIONINFO and other edits.</p> <p><a href="#">dev_GetResourceReservationInfoEx()</a> function: Updated example code to include inline function INIT_DEV_RESOURCE_RESERVATIONINFO_EX and other edits.</p> <p><a href="#">dev_PortConnect()</a> function: Updated example code to include inline function INIT_DM_PORT_CONNECT_INFO_LIST.</p> <p><a href="#">dev_ReleaseResourceEx()</a> function: Updated example code to include inline function INIT_DEV_RESOURCE_LIST.</p> <p><a href="#">dev_ReserveResourceEx()</a> function: Added caution about cleaning up resources before exiting. Updated example code to include inline function INIT_DEV_RESOURCE_LIST.</p> <p><a href="#">Events</a> chapter: Added eight new events associated with the new Device Connection functions.</p> <p><a href="#">Data Structures</a> chapter: Added the following new data structures: <a href="#">DM_CONNECT_STATUS_LIST</a>, <a href="#">DM_PORT_CONNECT_INFO</a>, <a href="#">DM_PORT_CONNECT_INFO_LIST</a>, <a href="#">DM_PORT_INFO</a>, <a href="#">DM_PORT_INFO_LIST</a>.</p> <p><a href="#">DEV_RESOURCE_LIST</a> structure: Added INIT_DEV_RESOURCE_LIST inline function. Changed version data type from 'int' to 'unsigned int'.</p> <p><a href="#">DEV_RESOURCE_RESERVATIONINFO</a> structure: Added INIT_DEV_RESOURCE_RESERVATIONINFO inline function. Changed version data type from 'int' to 'unsigned int'.</p> <p><a href="#">DEV_RESOURCE_RESERVATIONINFO_EX</a> structure: Added INIT_DEV_RESOURCE_RESERVATIONINFO_EX inline function. Changed version data type from 'int' to 'unsigned int'.</p>

## Revision History

Document No.	Publication Date	Description of Revisions
05-2222-005	September 2006	<p>Global change: Revisions included adding new Dialogic® Multimedia Platform for AdvancedTCA references, function operations, data structures, and events.</p> <p><b>Purpose</b> section: Updated the description of the API to include Dialogic® Multimedia Platform for AdvancedTCA.</p> <p><b>Function Summary by Category</b> chapter: Specified which <b>Resource Reservation Functions</b> are on Dialogic® HMP Software and Dialogic® Multimedia Platform for AdvancedTCA.</p> <p><b>Function Information</b> chapter: Added three new functions: <b>dev_GetResourceReservationInfoEx( )</b>, <b>dev_ReleaseResourceEx( )</b>, <b>dev_ReserveResourceEx( )</b>. Added Dialogic® Multimedia Platform for AdvancedTCA to <b>dev_Connect( )</b>, <b>dev_Disconnect( )</b>, and <b>dev_ErrorInfo( )</b> functions.</p> <p><b>Data Structures</b> chapter: Added three new data structures: <b>DEV_RESOURCE_LIST</b>, <b>DEV_RESOURCE_RESERVATIONINFO_EX</b>, and <b>resourceInfo</b>.</p> <p><b>Events</b> chapter: Added four new <b>Resource Reservation Events</b>: <b>DMEV_RELEASE_RESOURCE</b>, <b>DMEV_RELEASE_RESOURCE_FAIL</b>, <b>DMEV_RESERVE_RESOURCE</b>, and <b>DMEV_RESERVE_RESOURCE_FAIL</b>. Added <b>dev_GetResourceReservationInfoEx( )</b> function to existing events <b>DMEV_GET_RESOURCE_RESERVATIONINFO</b> and <b>DMEV_GET_RESOURCE_RESERVATIONINFO_FAIL</b>.</p>
05-2222-004	August 2006	<p><b>dev_Connect( )</b> function: Added new connection types to the section on <b>Supported Connections</b>.</p>
05-2222-003	August 2005	<p>Added multimedia features. Added DML error handling, updated some function operations, and made a few corrections.</p> <p><b>Purpose</b> section: Updated the description of the API to include ability to connect IP media and multimedia devices.</p> <p><b>dev_Connect( )</b> function: Added section on <b>Supported Connections</b>. Removed section on Implicit Disconnection (as well as corresponding caution) as not applicable. Changed <b>Cautions</b> section to indicate that multiple connections are not possible. Added <b>Multimedia Sample</b> and <b>Example A (Multimedia Asynchronous)</b>. Corrected the T.38 Sample, which referred to the IPML define <b>MEDIATYPE_LOCAL_T38_INFO</b> instead of <b>MEDIATYPE_LOCAL_UDPTL_T38_INFO</b>.</p> <p><b>dev_Disconnect( )</b> function: Changed <b>Cautions</b> 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 <b>dev_Connect( )</b> example code. Replaced the T.38 Sample with a cross reference to identical sample in <b>dev_Connect( )</b>.</p> <p>Global changes to add DML Error Handling: Added <b>dev_ErrorInfo( )</b> function, <b>DEV_ERRINFO</b> structure, and changed all error codes to DML-specific ones, including the <b>Error Codes</b> chapter.</p>

## Revision History

Document No.	Publication Date	Description of Revisions
05-2222-002	September 2004	<p><a href="#">dev_ReleaseResource( )</a> and <a href="#">dev_ReserveResource( )</a> 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 <b>RESOURCE_IPM_LBR</b>) are supported in synchronous mode only (asynchronous mode is not supported).</p> <p><b>Resource Reservation Events:</b> Removed the following Resource Reservation events because asynchronous mode is not supported for the Resource Reservation functions:</p> <ul style="list-style-type: none"><li>DMEV_RELEASE_RESOURCE</li><li>DMEV_RELEASE_RESOURCE_FAIL</li><li>DMEV_RESERVE_RESOURCE</li><li>DMEV_RESERVE_RESOURCE_FAIL</li></ul> <p><a href="#">dev_ReleaseResource( )</a> 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").</p>
05-2222-001	September 2003	Initial version of document.

## ***Revision History***

# 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 the connections between devices for communication and sharing of resources.

The Dialogic® Device Management API library provides the following functionality:

- Connects IP media and multimedia devices, enabling multimedia (audio/video) communication over IP connections.
- Originates and terminates T.38 fax over IP connections through use of a T.38 fax IP-only resource.
- Reserves low bit rate codecs, such as G.723 or G.729, for an IP media device.

Unless noted otherwise, the information in this document pertains to both Dialogic® Host Media Processing (HMP) Software and Dialogic® Multimedia Platform for AdvancedTCA software, which is based on the Dialogic® HMP Software.

## Applicability

This document version (05-2222-007) is published for Dialogic® Host Media Processing Software Release 3.1LIN (also referred to as Dialogic HMP Software 3.1LIN).

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

## Intended Audience

This 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 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 library.
- [Chapter 3, “Events”](#) describes the events that are generated by the Dialogic® Device Management API functions.
- [Chapter 4, “Data Structures”](#) provides information on the data structures used with Dialogic® Device Management API functions, along with their 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)

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:

- Dialogic® Device Management API Header File ..... 11
- Device Connection Functions ..... 11
- Resource Reservation Functions ..... 12
- Error Processing Functions ..... 12

## 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 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:

**dev\_GetResourceReservationInfo()**

Provides the current reservation information for the specified resource and device in a DEV\_RESOURCE\_RESERVATIONINFO data structure.

**dev\_ReleaseResource()**

Releases a specified resource 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.

**dev\_GetResourceReservationInfoEx()**

Obtains the current reservation information for the specified resource(s) and device, and provides it in the DEV\_RESOURCE\_RESERVATIONINFO\_EX data structure.

**dev\_ReleaseResourceEx()**

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

**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.

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

## 1.4 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 the subsystems employed in the Dialogic® Device Management API function call, and provides it in the DEV\_ERRINFO error information structure.

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.

**Note:** The “Platform” row in the function syntax table indicates the platforms supported for each function. Platforms include Dialogic® Host Media Processing software and Dialogic® Multimedia Platform for AdvancedTCA.

## 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

**Platform:** Dialogic® HMP Software and Dialogic® Multimedia Platform for AdvancedTCA

---

### ■ 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.

Parameter	Description
<b>devHandle1</b>	specifies a valid channel device handle obtained when the channel was opened
<b>devHandle2</b>	specifies a valid channel device handle obtained when the channel was opened
<b>connType</b>	specifies a connection type from among the following valid values: <ul style="list-style-type: none"><li>• <b>DM_FULLDUP</b> – Specifies full duplex communication (default)</li><li>• <b>DM_HALFDUP</b> – Specifies half duplex communication where the first device listens to the second device (i.e., <b>devHandle1</b> listens to <b>devHandle2</b>)</li></ul>
<b>mode</b>	specifies how the function should be executed. Set this to one of the following: <ul style="list-style-type: none"><li>• <b>EV_ASYNC</b> – asynchronously</li><li>• <b>EV_SYNC</b> – synchronously (default)</li></ul>

## ■ Supported Connections

The **dev\_Connect()** function can create the following connections:

### 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. Only asynchronous mode is 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 Conferencing Party and Voice

A full duplex or half duplex connection between a conferencing party device (CNF API) and a voice device. Requires a valid conferencing party device handle obtained through the **cnf\_OpenParty()** function and a valid voice device handle obtained through the **dx\_open()** function. Only asynchronous mode is supported. In the half duplex connection, either type of device can listen to the other.

### CNF Conferencing Party and IP Media

A full duplex or half duplex connection between a conferencing party device (CNF API) and an IP media device. Requires a valid 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 Conferencing Party and CNF Conferencing Party

A full duplex connection between two conferencing party devices (CNF API). Requires valid conferencing party device handles obtained through the **cnf\_OpenParty()** function. Only asynchronous mode is supported.

### CNF Conferencing Party and Digital Network Interface Device

A full duplex or half duplex connection between a conferencing party device (CNF API) and a DTI device. Requires a valid conferencing party device handle obtained through the **cnf\_OpenParty()** function and a valid DTI device handle obtained through the **dt\_open()** function. Only *synchronous* mode is supported. In the half duplex connection, either type of device can listen to the other.

To break the connection made by **dev\_Connect()**, you must use the **dev\_Disconnect()** function.

To connect other device types, the technology-specific routing functions must be used, 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

## ***dev\_Connect()* — connect devices**

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 the Dialogic® Device Management API Error Processing function **dev\_ErrorInfo()** 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 the Dialogic® Device Management API Error Processing function **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).

### ■ Errors

If this function returns -1 to indicate failure, or if it generates a DMEV\_CONNECT\_FAIL error event, use the Dialogic® Device Management API Error Processing function **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\_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 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.

**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.

See also [Chapter 5, “Error Codes”](#) for additional information.

## ■ 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

## ***dev\_Connect()*** — *connect devices*

- 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\_FULLLDUP) 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.

### ■ 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 <dxxlib.h>
#include <faxlib.h>
#include <ipmlib.h>
#include <devmgmt.h>
#include <mmllib.h>

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

## **dev\_Connect() — connect devices**

```
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_getevtttype();

    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. Propagate to waitvt() in Main
        return 1;
    }
    break;

```

## **dev\_Connect() — connect devices**

```
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 );
    }
}

```

## ***dev\_Connect()* — connect devices**

```
}

int mm_handle = mm_Open("mmB1C1", NULL, NULL);
if( mm_handle == -1 )
{
    printf( "mm_open() failed.\n" );
    exit( 1 );
}

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

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

if( sr_dishdr( ipm_handle, EV_ANYEVT, IpEventHandler ) == -1 )
{
    printf( "sr_dishdr() 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 <dxxplib.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 IpEventHandler( 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 )
                {

```

## ***dev\_Connect()* — connect devices**

```
        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_getevtttype();

    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( "dxxB2C1", 0 );
    if( vox_handle == -1 )
    {
        printf( "dx_open() failed.\n" );
        exit( 1 );
    }
}
```

## ***dev\_Connect()* — connect devices**

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

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

if( sr_dishdr( ipm_handle, EV_ANYEVT, IpmEventHandler ) == -1 )
{
    printf( "sr_dishdr() 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_FULLLDUP, 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 )
    {

```

## ***dev\_Connect()*** — connect devices

```
    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()* — disconnect devices**

This function generates a DMEV\_DISCONNECT\_FAIL error event to indicate failure of the function operation. Use the Dialogic® Device Management API Error Processing function [dev\\_ErrorInfo\(\)](#) 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 the Dialogic® Device Management API Error Processing function [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, or if it generates a DMEV\_DISCONNECT\_FAIL error event, use the Dialogic® Device Management API Error Processing function [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

**Platform:** Dialogic® HMP Software and Dialogic® Multimedia Platform for AdvancedTCA

---

### ■ 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
<b>pErrInfo</b>	specifies a pointer to an error information structure. Upon successful completion of the function operation, the structure is filled with results. See the DEV_ERRINFO data structure in <a href="#">Chapter 4, “Data Structures”</a> for more information.

### ■ Cautions

- The **dev\_ErrorInfo( )** function should only be called 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 <dxxplib.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

**Platform:** Dialogic® HMP Software and Dialogic® Multimedia Platform for AdvancedTCA

---

### ■ 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
<b>devHandle</b>	specifies a valid channel device handle obtained when the channel was opened
<b>pUserContext</b>	specifies a user-supplied pointer that can be retrieved using <b>sr_getUserContext()</b> 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 Dialogic® Standard Runtime Library (SRL) functions 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()**.

## retrieve device receive ports information — `dev_GetReceivePortInfo()`

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, or if it generates a `DMEV_GET_RX_PORT_INFO_FAIL` event, use the Dialogic® Standard Runtime Library (SRL) functions `ATDV_LASTERR()` and `ATDV_ERRMSGP()` to retrieve the error information. Possible errors for this function include:

`EDEV_BADPARAM`  
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;
        }
    }
}
```

## ***dev\_GetReceivePortInfo()*** — retrieve device receive ports information

```
        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_getevtttype();
    if (DMEV_GET_RX_PORT_INFO != evt) {
        cout << "dev_GetReceivePortInfo error";
        cout << " event = " << evt << endl;
        throw 4;
    }
    evt_data = sr_getevtdata();
    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\_RESERVAT • pointer to resource reservation information structure  
IONINFO \*pResourceInfo  
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

**Platform:** Dialogic® HMP Software

---

### ■ Description

The **dev\_GetResourceReservationInfo( )** function obtains the current reservation information for the specified resource 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.

Parameter	Description
<b>devHandle</b>	specifies a valid channel device handle obtained when the channel was opened
<b>pResourceInfo</b>	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 <a href="#">Chapter 4, “Data Structures”</a> for more information.
<b>mode</b>	specifies how the function should be executed. Set this to one of the following: <ul style="list-style-type: none"><li>• EV_ASYNC – asynchronously</li><li>• EV_SYNC – synchronously</li></ul>

### ■ 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

## ***dev\_GetResourceReservationInfo()* — get resource reservation information**

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 the Dialogic® Device Management API Error Processing function `dev_ErrorInfo()` to retrieve the error information.

**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 the Dialogic® Device Management API Error Processing function `dev_ErrorInfo()` to retrieve the error information.

### ■ 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 or not available, it generates a subsystem error (e.g., `EDEV_IPM_SUBSYSTEMERR`).

### ■ Errors

If this function returns -1 to indicate failure, or if it generates a `DMEV_GET_RESOURCE_RESERVATIONINFO_FAIL` error event, use the Dialogic® Device Management API Error Processing function `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.

## **get resource reservation information — dev\_GetResourceReservationInfo()**

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

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

void main()
{
    int devHandle; // channel handle
    .
    .
    // Register event handler thru SRL
    sr_enbhdr( 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.maxReservePoolCount);

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

### ■ See Also

- [dev\\_ReserveResource\(\)](#)
- [dev\\_ReleaseResource\(\)](#)

## dev\_GetResourceReservationInfoEx()

**Name:** int dev\_GetResourceReservationInfoEx(devHandle, pResourceInfo, mode)

**Inputs:** int devHandle • valid channel device  
DEV\_RESOURCE\_RESERVAT • pointer to resource reservation information structure  
IONINFO\_EX  
\*pResourceInfo  
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

**Platform:** Dialogic® HMP Software and Dialogic® Multimedia Platform for AdvancedTCA

---

### ■ 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.

Parameter	Description
<b>devHandle</b>	specifies a valid channel device handle obtained when the channel was opened
<b>pResourceInfo</b>	specifies a pointer to a resource reservation information structure. 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.
<b>mode</b>	specifies how the function should be executed. Set this to one of the following: <ul style="list-style-type: none"><li>• EV_ASYNC - asynchronously</li><li>• EV_SYNC - synchronously</li></ul>

### ■ 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

## *get resource reservation information — dev\_GetResourceReservationInfoEx()*

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 the Dialogic® Device Management API error processing function [dev\\_ErrorInfo\(\)](#) 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 the Dialogic® Device Management API error processing function [dev\\_ErrorInfo\(\)](#) to retrieve the error information.

- Notes:**
1. It is better to use asynchronous mode because [dev\\_GetResourceReservationInfoEx\(\)](#) is not executed in a single thread of execution as with functions used with HMP software.
  2. 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 (e.g., EDEV\_IPM\_SUBSYSTEMERR).
- If the specified resource is invalid or not available, it generates a subsystem error (e.g., EDEV\_IPM\_SUBSYSTEMERR).

### ■ Errors

If this function returns -1 to indicate failure, or if it generates a DMEV\_GET\_RESOURCE\_RESERVATIONINFO\_FAIL error event, use the Dialogic® Device Management API error processing function [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).

## ***dev\_GetResourceReservationInfoEx()* — get resource reservation information**

### ■ **Asynchronous Code Example**

```
int main()
{

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

    // Register event handler function with srl
    sr_enbhdr( 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 with error = %d\n", ATDV_NAMEP(nDeviceID), ATDV_LASTERR(nDeviceID));
        /*
        . .
        Perform Error Processing
        . .
        */
    }
    /* Continue processing */
    return 0;
}
```

### ■ **Synchronous Code Example**

```
#include "srllib.h"
#include "ipmlib.h"
#include "devgmt.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 resource reservation information — dev\_GetResourceReservationInfoEx()***

```
// 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(" Resource Type = %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\_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

**Platform:** Dialogic® HMP Software and Dialogic® Multimedia Platform for AdvancedTCA

---

### ■ 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
<b>devHandle</b>	specifies a valid channel device handle obtained when the channel was opened
<b>pUserContext</b>	specifies a user-supplied pointer that can be retrieved using <b>sr_getUserContext()</b> 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 Dialogic® Standard Runtime Library (SRL) functions 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()**.

## retrieve device transmit ports information — `dev_GetTransmitPortInfo()`

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, or if it generates a `DMEV_GET_TX_PORT_INFO_FAIL` event, use the Dialogic® Standard Runtime Library (SRL) functions `ATDV_LASTERR()` and `ATDV_ERRMSGP()` to retrieve the error information. Possible errors for this function include:

`EDEV_BADPARAM`  
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_getevtttype();
        if (IPMEV_OPEN != evt) {
            cout << "ipm_Open error";
            cout << " event = " << evt << endl;
        }
    }
}
```

## ***dev\_GetTransmitPortInfo()*** — retrieve device transmit ports information

```
        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_getevtttype();
    if (DMEV_GET_TX_PORT_INFO != evt) {
        cout << "dev_GetTransmitPortInfo error";
        cout << " event = " << evt << endl;
        throw 4;
    }
    evt_data = sr_getevtdata();
    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:**

<code>int devHandle</code>	• a valid channel device
<code>CPDM_PORT_CONNECT_INFO_LIST pConnectList</code>	• a pointer to the list of connection structures
<code>void *pUserContext</code>	• 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

**Platform:** Dialogic® HMP Software and Dialogic® Multimedia Platform for AdvancedTCA

---

### ■ Description

The `dev_PortConnect()` function creates connections between one or more transmit ports of the specified device and 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.

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.

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

## ***dev\_PortConnect()* — create connections between transmit and receive ports**

### ■ **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 the Dialogic® Standard Runtime Library (SRL) functions 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 the either event and can be retrieved using `sr_getevtdatap()`.

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

### ■ **Multimedia Scenario**

The following describes how to establish full-duplex audio and video connections between two devices. In this case, the two devices are the multimedia (MM) device and the IP media (IPM) device.

- Use `dev_GetTransmitPortInfo()` and `dev_GetReceivePortInfo()` to retrieve the transmit port and the receive port information for the MM device.
- Use `dev_GetTransmitPortInfo()` and `dev_GetReceivePortInfo()` to retrieve the transmit port and receive port information for the IPM device.
- Create a full-duplex connection by calling `dev_PortConnect()` twice: first to create the connections from the transmit ports of the MM device to the receive ports of the IPM device, and then again to create the connections from the transmit ports of the IPM device to the receive ports of the MM device.

### ■ **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.

## ***create connections between transmit and receive ports — dev\_PortConnect()***

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, or if it generates a DMEV\_PORT\_CONNECT\_FAIL event, use the Standard Runtime Library (SRL) functions **ATDV\_LASTERR()** and **ATDV\_ERRMSGP()** to retrieve the error information. Possible errors for this function include:

EDEV\_BADPARAM  
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 transmit ports of ipmB1C2 are connected to the 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
        DM_PORT_CONNECT_INFO& info =
            pconn_lst->port_connect_info[k];
        INIT_DM_PORT_CONNECT_INFO(&info);
    }
}
```

## ***dev\_PortConnect()*** — create connections between transmit and receive ports

```
        info.unFlags = DMFL_TRANSCODE_ON;
        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_getevtttype();
        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_getevtttype();
        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_getevtttype();
        if (DMEV_GET_RX_PORT_INFO != evt) {
            cout << "dev_GetReceivePortInfo error";
            cout << " event = " << evt << endl;
        }
    }
}
```

## **create connections between transmit and receive ports — dev\_PortConnect()**

```
        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) {
```

## ***dev\_PortConnect()*** — *create connections between transmit and receive ports*

```
        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 • a pointer to the list of connection structures  
pConnectList  
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

**Platform:** Dialogic® HMP Software and Dialogic® Multimedia Platform for AdvancedTCA

---

### ■ Description

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

Parameter	Description
<b>devHandle</b>	specifies a valid channel device handle obtained when the channel was opened
<b>pConnectList</b>	specifies a pointer to the list of connection structures, <a href="#">DM_PORT_CONNECT_INFO_LIST</a>
<b>pUserContext</b>	specifies a user-supplied pointer that can be retrieved using <a href="#">sr_getUserContext()</a> 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 the Dialogic® Standard Runtime Library (SRL) functions to obtain the error information.

## ***dev\_PortDisconnect()* — sever connections between transmit and receive ports**

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, or if it generates a DMEV\_PORT\_DISCONNECT\_FAIL event, use the Dialogic® Standard Runtime Library (SRL) functions **ATDV\_LASTERR()** and **ATDV\_ERRMSGP()** to retrieve the error information. Possible errors for this function include:

EDEV\_BADPARAM  
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` • a resource type  
`resType`  
`unsigned short mode` • synchronous function mode

**Returns:** `DEV_SUCCESS` if successful  
`-1` if failure

**Includes:** `srllib.h`  
`devmgmt.h`

**Category:** Resource Reservation

**Mode:** synchronous

**Platform:** Dialogic® HMP Software

### ■ Description

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.

**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.

Parameter	Description
<code>devHandle</code>	specifies a valid channel device handle obtained when the channel was opened
<code>resType</code>	specifies a resource type. The following is the only valid value: <ul style="list-style-type: none"> <li>• <b>RESOURCE_IPM_LBR</b> – 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.</li> </ul>
<code>mode</code>	specifies how the function should be executed. For resource type <b>RESOURCE_IPM_LBR</b> , set this to: <ul style="list-style-type: none"> <li>• <code>EV_SYNC</code> – synchronously</li> </ul>

### ■ 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.

## *dev\_ReleaseResource()* — release a resource

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 the Dialogic® Device Management API Error Processing function `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 the Dialogic® Device Management API Error Processing function `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
    .
    .
}
```

## ***release a resource — dev\_ReleaseResource()***

```
// Open channel
if ((devHandle = ipm_Open("ipmBlC1", 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

- [dev\\_GetResourceReservationInfo\(\)](#)
- [dev\\_ReserveResource\(\)](#)

## dev\_ReleaseResourceEx()

**Name:** int dev\_ReleaseResourceEx(devHandle, pResourceList, mode)

**Inputs:** int devHandle • valid channel device  
DEV\_RESOURCE\_LIST • pointer to resource reservation list structure  
\*pResourceList  
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

**Platform:** Dialogic® HMP Software and Dialogic® Multimedia Platform for AdvancedTCA

---

### ■ 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.

Parameter	Description
<b>devHandle</b>	specifies a valid channel device handle obtained when the channel was opened
<b>pResourceList</b>	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.
<b>mode</b>	specifies how the function should be executed. Set this to one of the following: <ul style="list-style-type: none"><li>• EV_ASYNC - asynchronously</li><li>• EV_SYNC - synchronously</li></ul>

### ■ 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 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\_RELEASE\_RESOURCE\_FAIL error event to indicate failure of the function operation. Use the Dialogic® Device Management API error processing function [dev\\_ErrorInfo\(\)](#) 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 the Dialogic® Device Management API error processing function [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, or if it generates a DMEV\_RELEASE\_RESOURCE\_FAIL error event, use the Dialogic® Device Management API error processing function [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).

## *dev\_ReleaseResourceEx()* — release a resource

### ■ 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.
    */
}
```

## ***release a resource — dev\_ReleaseResourceEx()***

```
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            • a resource type  
          resType  
          unsigned short mode            • synchronous function mode

**Returns:** DEV\_SUCCESS if successful  
          -1 if failure

**Includes:** srllib.h  
          devmgmt.h

**Category:** Resource Reservation

**Mode:** synchronous

**Platform** Dialogic® HMP Software

---

### ■ Description

The **dev\_ReserveResource( )** function reserves a resource for use by the specified device. This allows an application program 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.

**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.

Parameter	Description
<b>devHandle</b>	specifies a valid channel device handle obtained when the channel was opened
<b>resType</b>	specifies a resource type. The following is the only valid value: <ul style="list-style-type: none"><li>• <b>RESOURCE_IPM_LBR</b> – 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.</li></ul>
<b>mode</b>	specifies how the function should be executed. For resource type <b>RESOURCE_IPM_LBR</b> , set this to: <ul style="list-style-type: none"><li>• <b>EV_SYNC</b> – synchronously</li></ul>

## ■ 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 the Dialogic® Device Management API Error Processing function **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 the Dialogic® Device Management API Error Processing function **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()
{
```

## ***dev\_ReserveResource()*** — *reserve a resource*

```
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

- [dev\\_GetResourceReservationInfo\(\)](#)
- [dev\\_ReleaseResource\(\)](#)

## 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

**Platform:** Dialogic® HMP Software and Dialogic® Multimedia Platform for AdvancedTCA

### ■ Description

The **dev\_ReserveResourceEx()** function reserves resource(s) for use by the specified device. This allows an application program 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.

Parameter	Description
<b>devHandle</b>	specifies a valid channel device handle obtained when the channel was opened

## ***dev\_ReserveResourceEx( ) — reserve a resource***

<b>Parameter</b>	<b>Description</b>
<b>pResourceList</b>	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.
<b>mode</b>	specifies how the function should be executed. Set this to one of the following: <ul style="list-style-type: none"><li>• EV_ASYNC - asynchronously</li><li>• EV_SYNC - synchronously</li></ul>

### ■ **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 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\_RESERVE\_RESOURCE\_FAIL error event to indicate failure of the function operation. Use the Dialogic® Device Management API error processing function [dev\\_ErrorInfo\( \)](#) 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 the Dialogic® Device Management API error processing function [dev\\_ErrorInfo\( \)](#) to retrieve the error information.

**Note:** 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 the [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, or if it generates a DMEV\_RESERVE\_RESOURCE\_FAIL error event, use the Dialogic® Device Management API error processing function **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));
    }
}
```

## ***dev\_ReserveResourceEx( ) — reserve a resource***

```
/* 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;
}
}
```

## reserve a resource — dev\_ReserveResourceEx()

```
}
int main()
{

    int devHandle; // channel handle
    DEV_RESOURCE_LIST devResourceList;

    // Register event handler function with srl
    sr_enbhdr( 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;
```

## ***dev\_ReserveResourceEx( ) — reserve a resource***

```
// 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\(\)](#)

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

- [Overview of Dialogic® Device Management API Events](#). . . . . 73
- [Device Connection Events](#). . . . . 73
- [Resource Reservation Events](#). . . . . 74

## 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 different events, and the functions documented in [Chapter 2, “Function Information”](#) describe the events applicable to them.

Termination events are produced when a function running in asynchronous mode terminates, either successfully or unsuccessfully. To collect termination event codes, use Dialogic® Standard Runtime Library (SRL) functions such as `sr_waitevt()` or `sr_enbhdr()`, depending on the programming model in use. 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.

## Events

### 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.

## **Events**

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](#) ..... 78
- [DEV\\_RESOURCE\\_LIST](#) ..... 79
- [DEV\\_RESOURCE\\_RESERVATIONINFO](#) ..... 80
- [DEV\\_RESOURCE\\_RESERVATIONINFO\\_EX](#) ..... 82
- [DM\\_CONNECT\\_STATUS\\_LIST](#) ..... 83
- [DM\\_PORT\\_CONNECT\\_INFO](#) ..... 84
- [DM\\_PORT\\_CONNECT\\_INFO\\_LIST](#) ..... 85
- [DM\\_PORT\\_INFO](#) ..... 86
- [DM\\_PORT\\_INFO\\_LIST](#) ..... 87
- [resourceInfo](#) ..... 88

## DEV\_ERRINFO

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

### ■ Description

This structure is used with the [dev\\_ErrorInfo\(\)](#) function to provide error information for the functions in the Dialogic® 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 Dialogic® Device Management API function call. The defines for the valid Dialogic® Device Management API error values are in the *devmgmt.h* header file and have a “EDEV\_” prefix (they are also listed in [Chapter 5, “Error Codes”](#)). If the error value returned indicates a subsystem error type (e.g., DEV\_IPM\_SUBSYSTEMERR or DEV\_FAX\_SUBSYSTEMERR), you should 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 Dialogic® 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 Dialogic® 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 rslst
    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 = 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 (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.

#### 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:** Some resource types, like **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 (i.e., 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**

***resource reservation information — DEV\_RESOURCE\_RESERVATIONINFO***

resources is specified through the HMP software License Manager. (If you change the setting, you must restart the 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 = 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

This 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\_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

This 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)

#### 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

This 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

This 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_NBUP`

## 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

This structure specifies a list of [DM\\_PORT\\_INFO](#) structures. It is used with **dev\_GetTransmitPortInfo()** and **dev\_GetReceivePortInfo()** to return device port information.

### ■ Field Descriptions

The fields of the DM\_PORT\_INFO\_LIST data structure are described as follows:

#### unVersion

The version number of the data structure. Set this number to the desired version of the structure. This allows you to maintain compatibility when the structure is superseded by a newer version. For the initial version, set to a value of `DM_PORT_CONNECT_INFO_VERSION_0`.

#### 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

This structure is used within DEV\_RESOURCE\_RESERVATIONINFO\_EX, which is passed in the [dev\\_GetResourceReservationInfoEx\(\)](#) function to provide resource reservation information. See also [DEV\\_RESOURCE\\_RESERVATIONINFO\\_EX](#).

### ■ 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.

#### 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:** Some resource types, like RESOURCE\_IPM\_<type>, 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.

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 the value `DEV_SUCCESS` (or 0) for a successful outcome and -1 for an unsuccessful outcome or an error. If an error occurs during execution of an asynchronous function, a failure event is sent to the application. (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 a function fails, use the Dialogic® Device Management API Error Processing function `dev_ErrorInfo()` to retrieve the error information for both the API library and any subsystems.

- Notes:**
1. The `dev_ErrorInfo()` function should only be called 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 (e.g., `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 (e.g., 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 include IP media, multimedia, and T.38 UDP fax devices.)

## EDEV\_INVALIDMODE

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

## **Error Codes**

### **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.