Device Management API for Windows* and Linux* Operating Systems

Library Reference

September 2004

05-2222-002

Publication Date: September 2004

Document Number: 05-2222-002

Intel Converged Communications, Inc.
1515 Route 10
Parsippany, NJ 07054

For **Technical Support**, visit the Intel Telecom Support Resources website at:
http://developer.intel.com/design/telecom/support

For **Products and Services Information**, visit the Intel Telecom Products website at:
http://www.intel.com/design/network/products/telecom

For **Sales Offices** and other contact information, visit the Where to Buy Intel Telecom Products page at:
http://www.intel.com/buy/wtb/wtb1028.htm

Contents

Revision History

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

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

About This Publication

The following topics provide information about this publication.

Purpose

This publication contains reference information for all functions, parameters, data structures, values, events, and error codes in the Device Management API. The API provides run-time control and management of configurable system devices, which includes functions to reserve resources and to manage the connections between devices for communication and sharing of resources. More specifically, the functions enable use of the T.38 fax IP-only resource, which provides the ability to originate and terminate T.38 fax over IP connections only. The API also includes functions to reserve low bit rate codecs (e.g., G.723 or G.729) for an IP media device on Intel® NetStructure™ Host Media Processing (HMP) software.

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 Device Management API functions.
- Chapter 4, "Data Structures" provides information on the data structures used with 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

For related Intel® Dialogic® publications, see the product documentation (known as the on-line bookshelf) provided with the software release or at the following web site: *http://resource.intel.com/telecom/support/documentation/releases/index.htm*.

Function Summary by Category 1

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

1.1 Device Management API Header File

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

Note: The header file also contains other functions, such as those belonging to the Board Management Library, which have a "brd_" prefix. The 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 Management API Device Connection functions manage the connections between devices, allowing communication and sharing of resources. They include the following functions:

dev_ReserveResource( )
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.

1.3 Resource Reservation Functions

Device Management API 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 Intel® NetStructure™ Host Media Processing (HMP) 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 on HMP software.

Function Information 2

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

2.1 Function Syntax Conventions

The 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 Device Management API use the prefix “dev_” in the function name.

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

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

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

dev_Connect( )

Name: int dev_Connect (devHandle1, devHandle2, connType, mode)

Inputs:

int devHandle1	• a valid channel device
int devHandle2	• a valid channel device
eCONN_TYPE connType	• type of connection to make between the devices
unsigned short mode	• asynchronous or synchronous function mode

Returns: DEV_SUCCESS if successful
-1 if failure

Includes: srllib.h
devmgmt.h

Category: Device Connection

Mode: asynchronous or synchronous

Platform HMP

■ 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( )** or by a subsequent **dev_Connect( )**.

Note: For Intel® NetStructure™ Host Media Processing (HMP) software, the **dev_Connect( )** function is supported for creating connections between IP media devices and fax devices only. The **dev_Connect( )** function provides the only means for connecting a T.38 UDP fax channel to an IP media channel on HMP software, and in this case, **a full duplex connection must be used** (**DM_FULLDUP**). To break the fax and IP media connection made by **dev_Connect( )** on HMP software, you must use the **dev_Disconnect( )** function rather than the **fx_unlisten( )** and **ipm_unlisten( )** functions. To connect other device types, the technology-specific routing functions must be used, such as **dx_listen( )** and **dt_listen( )**.

Parameter	Description
devHandle1	specifies a valid channel device handle obtained when the channel was opened
devHandle2	specifies a valid channel device handle obtained when the channel was opened

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

Implicit Disconnection

If **dev_Connect( )** is used when a previous connection exists on one of the devices, depending on the types of connections, it breaks the previous connection before making the new connection. This implicit disconnection occurs because a device cannot listen to more than one device at any given time. So if a full duplex connection is made between dev1 and dev2, they listen to each other and no longer listen to any other device they may have been listening to. If a half duplex connection is made between dev1 and dev2, dev1 listens to dev2 and no longer listens to any other device it may have been listening to.

This is described in more detail for the following cases, where pre-existing connections are broken when creating a new full duplex and half duplex connection with **dev_Connect( )**:

- If you create a full duplex connection between dev1 and dev2 followed by a full duplex or a half duplex connection between dev1 and dev3, it breaks the original dev1 receive connection to the dev2 transmit channel. However, it does not break the dev2 receive connection to the dev1 transmit channel, and if you want to break this, you must issue a **dev_Disconnect( )** on dev2.
- If you create a full duplex connection between dev1 and dev2 followed by a half duplex connection between dev3 and dev1 (so dev3 listens to dev1), it does not break any part of the original full duplex connection between dev1 and dev2. If you want to break these connections, you should issue a **dev_Disconnect( )** on dev1 and dev2 before making the connection between dev3 and dev1.

Asynchronous Operation

To run this function asynchronously, set the mode parameter to EV_ASYNC. The function returns 0 to indicate it has initiated successfully. The function generates a DMEV_CONNECT termination event for each device to indicate successful completion of the function operation. The function always generates one event for each device regardless of whether the connection type is full duplex or half duplex (i.e., a successful full duplex connection will generate two events). The application program must wait for the completion events that indicate the connection was successful. Use the 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 Standard Runtime Library (SRL) functions to obtain 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 Standard Runtime Library (SRL) functions to get the error information.

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 T.38 fax and IP media connection made by **dev_Connect( )** on HMP software, you must use the **dev_Disconnect( )** function.
- When you use the Device Connection functions to connect and disconnect specific devices, use them consistently for the same operations. For example, when routing and unrouting the same devices, do not mix **dev_Connect( )** and **dev_Disconnect( )** with the technology-specific routing functions, such as **dx_listen( )**, **dx_unlisten( )**, **dt_listen( )**, **dt_unlisten( )**, **nr_scroute( )**, and **nr_scunroute( )**.
- In asynchronous mode, a call to **dev_Connect( )** must complete, as indicated by the termination event, before a second **dev_Connect( )** can be made successfully on one of the same devices; otherwise, the second connection results in an EIPM_INV_STATE error. This could occur in a setup where you make multiple connections in a row.
- 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). However, the attempt to make the full duplex connection could result in an implicit disconnection of a previous connection on one of the devices (see Implicit Disconnection). If an implicit disconnection occurs due to a failed **dev_Connect( )**, it is not automatically reconnected or restored to its original state.
- The connect operation must end before performing a second connect on the same device; otherwise, it generates an EIPM_INV_STATE error.

Errors

If this function returns -1 to indicate failure, or if it generates a DMEV_CONNECT_FAIL error event, use the Standard Runtime Library (SRL) functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve the error information. Possible errors for this function include:

EIPM_BADPARM
: Invalid argument or parameter.

EIPM_INV_STATE
: Invalid state. Indicates that the IP media initial command did not complete before another function call was made.

EIPM_SYSTEM
: Error from operating system; use **dx_fileerrno( )** to obtain the error value.

See also Chapter 5, "Error Codes" for additional information.

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 using HMP software. It is intended as a basic guideline to show some of the steps involved in general terms.

- Use the **ipm_open( )** function to open the IP media device and get the device handle.
- Use the **dx_open( )** function to open the voice resource device and get the device handle.
- Use the **dx_getfeaturelist( )** function to get feature information on the voice device handle.
- Check the ft_fax feature table information to see if it is a valid fax device (FT_FAX).
- Use the **fx_open( )** function to open the fax resource device and get the device handle.
- Check the ft_fax feature table information to see if it is a valid T.38 fax device (FT_FAX_T38UDP).
- Use the **dev_Connect( )** function to make a full duplex connection (DM_FULLDUP) between the IP media device and the fax device.
- Wait for the DMEV_CONNECT events for both the IP media device and the fax device to confirm that the **dev_Connect( )** function was successful.
- Set MediaData[0].eMediaType = MEDIATYPE_LOCAL_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_START_MEDIA 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

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

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

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

static int ipm_handle = -1;
```

```
static int fax_handle = -1;

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

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

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

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

    {
      info.MediaData[0].eMediaType = MEDIATYPE_LOCAL_UDPTL_T38_INFO;


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

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

    {
      info.unCount = 1;
      info.MediaData[0].eMediaType = MEDIATYPE_REMOTE_UDPTL_T38_INFO;
      info.MediaData[0].mediaInfo.PortInfo.unPortId = 6001;  // remote IP port
      strcpy( info.MediaData[0].mediaInfo.PortInfo.cIPAddress, "146.152.84.56");

      info.MediaData[1].eMediaType = MEDIATYPE_FAX_SIGNAL;
      info.MediaData[1].mediaInfo.FaxSignal.eToneType = TONE_CED;

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

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

    ipm_handle_disconnected = true;

    if( fax_handle_disconnected )
    {
      return 1;
    }
    break;

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

    fd = dx_fileopen( "onepg_high.tif", O_RDONLY|O_BINARY );

    if( fd == -1 )
```

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

    fx_setiott(&iott, fd, DF_TIFF, DFC_EOM);

    iott.io_type |= IO_EOT;
    iott.io_firstpg = 0;
    iott.io_pgcount = -1;
    iott.io_phdcont = DFC_EOP;

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

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

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

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

  case IPMEV_ERROR:
    printf( "IPMEV_ERROR event received on IPM channel.\n" );
    exit( -1 );
    break;

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

  return 0;
}

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

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

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

```
    break;

  case TFX_FAXERROR:
    printf( "TFX_FAXERROR event received.\n" );
    exit( 1 );
    break;

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

  case DMEV_DISCONNECT:
    printf( "DMEV_DISCONNECT event received.\n" );
    fax_handle_disconnected = true;
    if( ipm_handle_disconnected )
    {
      return 1;
    }
    break;

  default:
    printf( "Unknown event %d received on fax channel.\n", evttype );
    break;
  }

  return 0;
}

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

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

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

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

  if( feature_table.ft_fax & FT_FAX )
  {
    if( feature_table.ft_fax & FT_FAX_T38UDP)
    {
      fax_handle = fx_open( "dxxxB2C1", 0 );

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

```
    }
  }
  else
  {
    printf( "Not a T.38 fax device.\n" );
    exit( 1 );
  }
}
else
{
  printf( "Not a fax device.\n" );
  exit( 1 );
}

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

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

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

sr_waitevt(-1);

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

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

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

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

Example B

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

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

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

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


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

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


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

  IPM_MEDIA_INFO info;

  ...
  // Setup IPM_MEDIA_INFO structure

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

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

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

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

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

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

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

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

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

See Also

- dev_Disconnect( )

dev_Disconnect( )

Name: int dev_Disconnect (devHandle, mode)

Inputs: int devHandle
- a valid channel device

unsigned short mode
- asynchronous or synchronous function mode

Returns: DEV_SUCCESS if successful
-1 if failure

Includes: srllib.h
devmgmt.h

Category: Device Connection

Mode: asynchronous or synchronous

Platform HMP

Description

The **dev_Disconnect( )** function 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 by means of the **dev_Connect( )** function. To break a full duplex connection that was originally established between the devices with **dev_Connect( )**, you must call **dev_Disconnect( )** for each device.

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

On Intel® NetStructure™ Host Media Processing (HMP) software, you must use the **dev_Disconnect( )** function, rather than the **fx_unlisten( )** and **ipm_unlisten( )** functions, to break the full duplex connection between a T.38 UDP fax device and an IP media device originally made by the **dev_Connect( )** function.

Asynchronous Operation

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

This function generates a DMEV_DISCONNECT_FAIL error event to indicate failure of the function operation. Use the Standard Runtime Library (SRL) functions to obtain 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 Standard Runtime Library (SRL) functions to get the error information.

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 T.38 fax and IP media connection made by **dev_Connect( )** on HMP software, you must use the **dev_Disconnect( )** function.
- If you use the **dev_Disconnect( )** function to disconnect a device that is not connected, it is ignored. It does not return an error. 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 will succeed but have no effect.

Errors

If this function returns -1 to indicate failure, or if it generates a DMEV_DISCONNECT_FAIL error event, use the Standard Runtime Library (SRL) functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve the error information. Possible errors for this function include:

EIPM_INV_STATE
: Invalid state. Indicates that the IP media initial command did not complete before another function call was made.

EIPM_SYSTEM
: Error from operating system; use **dx_fileerrno( )** to obtain the error value.

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

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 using HMP software. It is intended as a basic guideline to show some of the steps involved in general terms.

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

- Use the **dev_Connect( )** function to make a full duplex connection (DM_FULLDUP) between the IP media device and the fax device.
- Wait for the DMEV_CONNECT events for both the IP media device and the fax device to confirm that the **dev_Connect( )** function was successful.
- Set MediaData[0].eMediaType = MEDIATYPE_LOCAL_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_START_MEDIA 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

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

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

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

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

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

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

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

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

    {
      info.MediaData[0].eMediaType = MEDIATYPE_LOCAL_UDPTL_T38_INFO;
```

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

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

  {
    info.unCount = 1;
    info.MediaData[0].eMediaType = MEDIATYPE_REMOTE_UDPTL_T38_INFO;
    info.MediaData[0].mediaInfo.PortInfo.unPortId = 6001;  // remote IP port
    strcpy( info.MediaData[0].mediaInfo.PortInfo.cIPAddress, "146.152.84.56");

    info.MediaData[1].eMediaType = MEDIATYPE_FAX_SIGNAL;
    info.MediaData[1].mediaInfo.FaxSignal.eToneType = TONE_CED;

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

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

  ipm_handle_disconnected = true;

  if( fax_handle_disconnected )
  {
    return 1;
  }
  break;

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

  fd = dx_fileopen( "onepg_high.tif", O_RDONLY|O_BINARY );

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

  fx_setiott(&iott, fd, DF_TIFF, DFC_EOM);

  iott.io_type |= IO_EOT;
  iott.io_firstpg = 0;
  iott.io_pgcount = -1;
  iott.io_phdcont = DFC_EOP;

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

  if( fx_sendfax( fax_handle, &iott, EV_ASYNC ) == -1 )
  {
    printf( "fx_sendfax() failed.\n" );
```

```
      exit( 1 );
    }
    break;

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

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

  case IPMEV_ERROR:
    printf( "IPMEV_ERROR event received on IPM channel.\n" );
    exit( -1 );
    break;

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

  return 0;
}

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

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

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

  case TFX_FAXERROR:
    printf( "TFX_FAXERROR event received.\n" );
    exit( 1 );
    break;

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

  case DMEV_DISCONNECT:
    printf( "DMEV_DISCONNECT event received.\n" );
    fax_handle_disconnected = true;
    if( ipm_handle_disconnected )
    {
      return 1;
    }
    break;

  default:
```

```
    printf( "Unknown event %d received on fax channel.\n", evttype );
    break;
  }

  return 0;
}

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

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

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

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

  if( feature_table.ft_fax & FT_FAX )
  {
    if( feature_table.ft_fax & FT_FAX_T38UDP)
    {
      fax_handle = fx_open( "dxxxB2C1", 0 );

      if( fax_handle == -1 )
      {
        printf( "fx_open() failed.\n" );
        exit( 1 );
      }
    }
    else
    {
      printf( "Not a T.38 fax device.\n" );
      exit( 1 );
    }
  }
  else
  {
    printf( "Not a fax device.\n" );
    exit( 1 );
  }

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

  if( sr_enbhdlr( fax_handle, EV_ANYEVT, FaxEventHandler ) == -1 )
```

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

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

  sr_waitevt(-1);

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

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

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

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

Example B

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

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

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

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


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

  if( IpmHandle == -1 )
  {
    printf( "Can not open IPM handle.\n" );
    // Perform system error processing
```

```
    exit( 1 );
  }


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

  IPM_MEDIA_INFO info;

  ...
  // Setup IPM_MEDIA_INFO structure

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

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

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

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

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


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

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

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

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

See Also

- dev_Connect( )

dev_GetResourceReservationInfo( )

Name: int dev_GetResourceReservationInfo (devHandle, pResourceInfo, mode)

Inputs:

int devHandle	• a valid channel device
DEV_RESOURCE_RESERVATIONINFO *pResourceInfo	• pointer to resource reservation information structure
unsigned short mode	• asynchronous or synchronous function mode

Returns: DEV_SUCCESS if successful
-1 if failure

Includes: srllib.h
devmgmt.h

Category: Resource Reservation

Mode: asynchronous or synchronous

Platform HMP

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.

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

Asynchronous Operation

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

Cautions

- This function requires that the device be open; otherwise, it generates an EIPM_INV_STATE error.
- If the specified resource is invalid or not available, it generates an EIPM_BADPARM error.

Errors

If this function returns -1 to indicate failure, or if it generates a DMEV_GET_RESOURCE_RESERVATIONINFO_FAIL error event, use the Standard Runtime Library (SRL) functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve the error information. Possible errors for this function include:

EIPM_BADPARM
: Invalid argument or parameter.

EIPM_INTERNAL
: Internal error.

EIPM_INV_STATE
: Invalid state. Indicates that the IP media initial command did not complete before another function call was made.

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

Example

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

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

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

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

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

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

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

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

   resInfo.version = 0;
   resInfo.resourceType = RESOURCE_IPM_LBR;
   if (dev_GetResourceReservationInfo(devHandle, &resInfo, EV_SYNC) ==-1)
   {
      printf("Cannot Get LBR resource reservation information.\n");
      // Perform system error processing
   }
   printf("LBR Usage for %s: ReservationStatus = %s\n, curReservePoolCount = %d,
          maxReservePoolCount = %d\n", ATDV_NAMEP(devHandle), (resInfo.curReserveCount == 1)
          ? "Reserved"  : "Unreserved", resInfo.curReservePoolCount,
          resInfo.maxRecervePoolCount);

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

See Also

- dev_ReserveResource( )
- dev_ReleaseResource( )

dev_ReleaseResource( )

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

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.

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

Synchronous Operation

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

To run this function synchronously, set the mode parameter to EV_SYNC. This function returns 0 to indicate successful completion and -1 to indicate failure. Use the Standard Runtime Library (SRL) functions to get the error information.

Cautions

- This function requires that the device be open; otherwise, it generates an EIPM_INV_STATE error.
- If the specified resource is actively being used, it cannot be released and generates an EIPM_BUSY error.
- 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 Standard Runtime Library (SRL) functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve the error information. Possible errors for this function include:

EIPM_BADPARM
: Invalid argument or parameter.

EIPM_BUSY
: Device is busy. (The device is currently executing another command.)

EIPM_INTERNAL
: Internal error.

EIPM_INV_STATE
: Invalid state. Indicates that the IP media initial command did not complete before another function call was made.

EIPM_SYSTEM
: Error from operating system; use **dx_fileerrno( )** to obtain the error value.

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

Example

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

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

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

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

```
    }

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

See Also

- dev_GetResourceReservationInfo( )
- dev_ReserveResource( )

dev_ReserveResource( )

Name: int dev_ReserveResource (devHandle, resType, mode)

Inputs:

int devHandle	• a valid channel device
eDEV_RESOURCE_TYPE resType	• a resource type
unsigned short mode	• synchronous function mode

Returns: DEV_SUCCESS if successful
-1 if failure

Includes: srllib.h
devmgmt.h

Category: Resource Reservation

Mode: synchronous

Platform HMP

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.

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

Synchronous Operation

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

To run this function synchronously, set the mode parameter to EV_SYNC. This function returns 0 to indicate successful completion and -1 to indicate failure. Use the Standard Runtime Library (SRL) functions to get 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 EIPM_INV_STATE error.
- If no resource of the specified type is available, it generates an EIPM_RESOURCESINUSE error.
- If you close the device, it releases all resources reserved for it.

Errors

If this function returns -1 to indicate failure, use the Standard Runtime Library (SRL) functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve the error information. Possible errors for this function include:

EIPM_BADPARM
Invalid argument or parameter.

EIPM_BUSY
Device is busy. (The device is currently executing another command.)

EIPM_INTERNAL
Internal error.

EIPM_RESOURCESINUSE
Resource is not available.

EIPM_INV_STATE
Invalid state. Indicates that the IP media initial command did not complete before another function call was made.

EIPM_SYSTEM
Error from operating system; use **dx_fileerrno( )** to obtain the error value.

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

Example

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

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

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

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

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

See Also

- dev_GetResourceReservationInfo( )
- dev_ReleaseResource( )

Events 3

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

3.1 Overview of Device Management API Events

When running in asynchronous mode, the functions in the 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 Standard Runtime Library (SRL) functions such as **sr_waitevt( )** or **sr_enbhdlr( )**, depending on the programming model in use. For detailed information on event handling and management, see the Standard Runtime Library documentation.

3.2 Device Connection Events

The following events are generated by the 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.

3.3 Resource Reservation Events

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

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

DMEV_GET_RESOURCE_RESERVATIONINFO_FAIL
: Termination event generated to indicate failure of the **dev_GetResourceReservationInfo( )** function operation.

Data Structures 4

This chapter provides information on the data structures used by 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_RESOURCE_RESERVATIONINFO

```
typedef struct getresourceinfo
{
   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

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

Field Descriptions

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

version
: 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 0.

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 Intel® NetStructure™ Host Media Processing (HMP) software, the maximum number of **RESOURCE_IPM_LBR** resources is specified through the HMP software License Manager. (If you change the setting, you must restart the Intel® telecom software for it to take effect.)

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

Error Codes 5

This chapter describes the error codes supported by the 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 DEV_FAILURE (or -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 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 (returns -1 or generates an error event), use the Standard Runtime Library (SRL) functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve the error information.

Notes: ***1.*** The error codes are generated by technology subsystems that are used by the Device Management API; for example, by internal function calls to the IP media library. For more information on the subsystem errors, refer to the documentation for the applicable library (e.g., *IP Media Library API for Linux and Windows Operating Systems Library Reference*).

2. To identify the error code, you must include the header file for the appropriate technology-specific subsystem (e.g., *IPMError.h*, *ipmlib.h*, and *faxlib.h*).

3. The technology-specific subsystem errors are device-specific rather than thread-specific.

When using the Device Management API, the following subsystem error codes are possible (they are listed in alphabetical order):

EIPM_BADPARM
: Invalid argument or parameter.

EIPM_BUSY
: Device is busy. (The device is currently executing another command.)

EIPM_INTERNAL
: Internal error.

EIPM_RESOURCESINUSE
: Resource is not available; an error generated by the **dev_ReserveResource( )** function.

EIPM_INV_STATE
: Invalid state. Indicates that the IP media initial command did not complete before another function call was made.

EIPM_SYSTEM
: Error from operating system; use **dx_fileerrno( )** to obtain the error value.