Dialogic®

Dialogic® IP Media Library API

Programming Guide

March 2008

05-2330-003_D

Publication Date: March 2008

Document Number: 05-2330-003_D

Contents

Figures

Tables

Revision History

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

Document No.	Publication Date	Description of Revisions
05-2330-003_D	March 2008	Made global changes to reflect Dialogic brand. Changed title to “Dialogic® IP Media Library API Programming Guide.” Required Libraries: Added note at end of this section.
05-2330-003	August 2005	T.38 Fax Server Scenario figure: Updated to show ipm_Stop( ) at correct time Quality of Service (QoS) Alarms chapter: Updated description of threshold and debounce settings (PTR#35692)
05-2330-002	July 2005	DTMF Handling chapter: Removed references to unsupported IPM_RFC2833MUTE_AUDIO parameter (PTR #33826) Generating DTMF section: Removed information relating to unsupported ipm_SendRFC2833SignalIDToIP( ) function QoS Alarm and Alarm Recovery Mechanisms section: Updated to illustrate jitter instead of lost packets
05-2230-001	April 2004	Initial version of document specific to the Dialogic® Host Media Processing Software product published under this title and document ID number. Much of the information contained in this document was previously published in September 2003 as an HMP-specific document under document number 05-1834-003 and the misleading title *IP Media Library API for Linux and Windows Programming Guide*. Significant changes from that previous document version inlcude: Application Development Guidelines chapter: Replaced by separate feature implementation chapters on DTMF Handling and T.38 Fax Server. Quality of Service (QoS) chapter: Added RTCP and RTP timeout alarm types. Removed DTMF discarded alarm type. Added new scenarios and graphics in QoS Alarm and Alarm Recovery Mechanisms section. Updated example code. Volume Control chapter: Added chapter on new feature.

About This Publication

The following topics provide information about this publication:

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

Purpose

This document provides programming guidelines for the Dialogic® IP Media Library API. This publiction is a companion guide to the *Dialogic® IP Media Library API Library Reference*, which provides details on functions and parameters in the IP media software.

Applicability

This document version (05-2330-003_D) is published for Dialogic® Multimedia Software for AdvancedTCA Releases 1.0, 1.1, and 2.0.

This document may also apply to other Dialogic® software releases. Check the Release Guide for your software release to determine whether this document is supported.

Intended Audience

This guide is intended for software developers who will access and utilize the Dialogic® IP media software. This may include any of the following:

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

How to Use This Publication

Refer to this publication after you have installed the hardware and the system software which includes the IP media software. This publication assumes that you are familiar with the Linux* or Windows* operating system and the C programming language.

The information in this guide is organized as follows:

- Chapter 1, "Product Description", introduces the IP media software and its key features.
- Chapter 2, "Programming Models", describes methods of developing IP media-based applications.
- Chapter 3, "State Models", describes a simple state-based IP media application.
- Chapter 4, "Event Handling", defines an event and describes how to handle an event.
- Chapter 5, "Error Handling", presents information on how to obtain error codes and handle errors.
- Chapter 6, "DTMF Handling", provides information on how to send and receive DTMF digits.
- Chapter 7, "T.38 Fax Server", provides information on implementing a T.38 fax server.
- Chapter 8, "Quality of Service (QoS) Alarms", details how QoS may be used in an application.
- Chapter 9, "Volume Control", describes how to use volume level adjustment on a call.
- Chapter 10, "Building Applications", describes how to compile and link IP media-based applications.

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)

Product Description 1

This chapter provides an overview of the Dialogic® IP Media Library API software. It contains the following sections:

1.1 Features

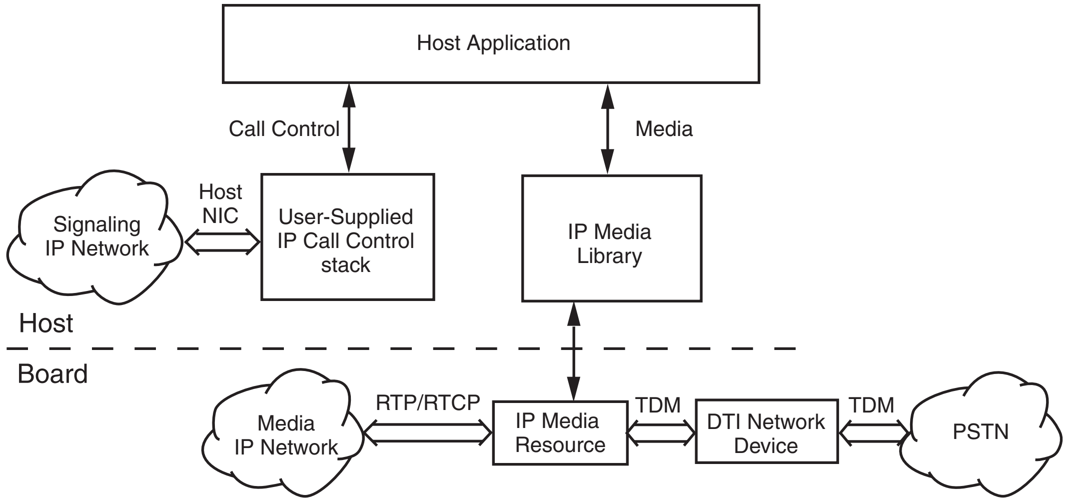
Some of the features of the Dialogic® IP Media Library API software include:

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

1.2 Architecture

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

Figure 1. IP Media Architecture



1.3 Introduction to the Dialogic® IP Media Library API

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

1.4 Relationship with Dialogic® Global Call API Library

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

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

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

Refer to the following Global Call manuals for more details:

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

1.5 Dialogic® Standard Runtime Library API Support

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

1.6 Media Channel Device Naming

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

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

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

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

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

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

Programming Models 2

This chapter describes the programming models supported by the Dialogic® IP media software.

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

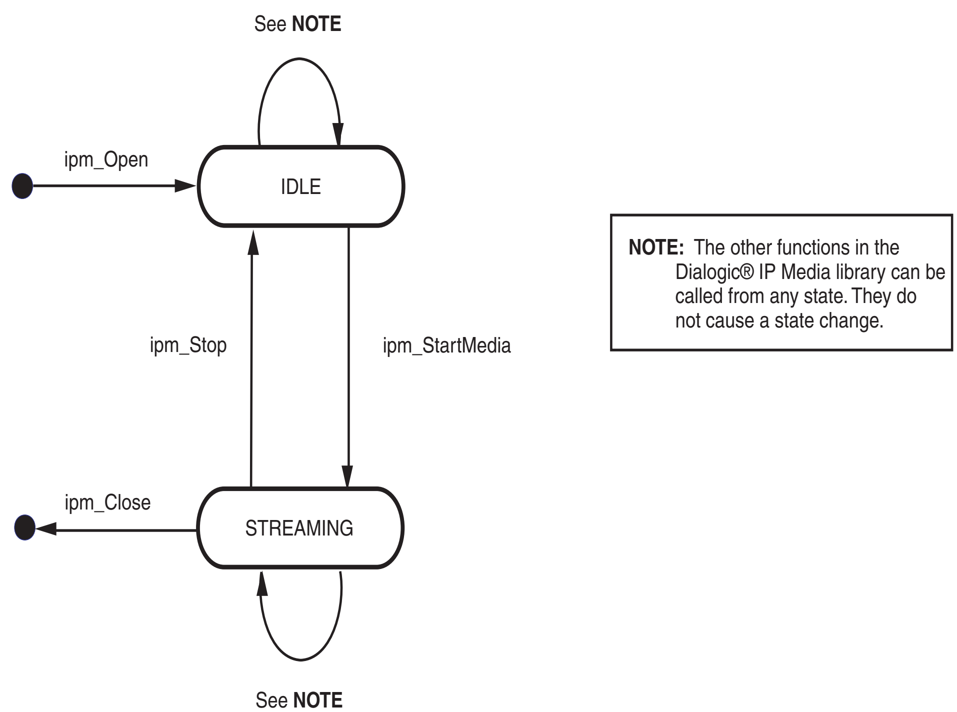
Note: The synchronous programming model is recommended for low density systems only. For high density systems, asynchronous programming models provide increased throughput for the application.

State Models 3

This chapter describes a very simple Dialogic® IP media state-based application.

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

Figure 2. IP Media Channel State Diagram



Event Handling 4

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

4.1 SRL Event Management Functions

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

sr_enbhdlr( )
: enables event handler

sr_dishdlr( )
: disables event handler

sr_getevtdev( )
: gets device handle

sr_getevttype( )
: gets event type

sr_waitevt( )
: wait for next event

sr_waitevtEx( )
: wait for events on certain devices

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

4.2 SRL Standard Attribute Functions

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

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

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

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

ATDV_SUBDEVS( )
number of subdevices

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

Error Handling 5

This chapter describes error handling for the Dialogic® IP media software.

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

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

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

DTMF Handling 6

This chapter contains guidelines for implementing DTMF handling using the Dialogic® IP media library. The following topics are discussed:

6.1 Introduction to DTMF Handling

When a session is started on an IPM device, the IPM device receives data from its IP interface and transmits data towards the TDM bus. A DTI device receives data from its Public Switched Telephone Network (PSTN) interface and transmits towards the TDM bus as well. In a gateway configuration, the DTI and IPM devices will be configured, via **gc_Listen( )** and **ipm_Listen( )** respectively, to listen to each other and thus create a full duplex communication path. The IPM device will forward DTMF that it receives on one interface to the other interface. Figure 1, "IP Media Architecture", on page 12 shows the data flow between the IP media library, the IP network, and the PSTN network.

When an IPM device receives DTMF from the TDM bus, there are several ways to forward it towards the IP interface. These include: forwarding it in the RTP stream (also called in-band), sending in the RTP stream via RFC 2833 packets, and using an application-controlled/defined method (also called out-of-band).

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

When using out-of-band mode, the DTMF is never transmitted in-band. As mentioned earlier, the application has the responsibility to forward the digits.

The setting for DTMF transfer mode also affects the handling of DTMF that is received from the IP interface. When the mode is set to in-band, the DTMF is automatically forwarded to the TDM bus.

When the mode is set to RFC 2833, DTMF is forwarded to the TDM bus as PCM data.

If out-of-band mode has been selected, then the application will use its own mechanism to be notified that a DTMF digit has been received. Then, **ipm_SendDigits( )** is used when necessary to transmit a DTMF digit towards the TDM bus.

6.2 Setting DTMF Parameters

This section contains the following topics:

- DTMF Modes
- Setting In-Band Mode
- Setting RFC 2833 Mode
- Setting Out-of-Band Mode

6.2.1 DTMF Modes

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

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

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

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

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

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

When using RFC2833, the payload type is specified by using the following parameter/value setting in a call to **ipm_SetParm( )**:

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

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

6.2.2 Setting In-Band Mode

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

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

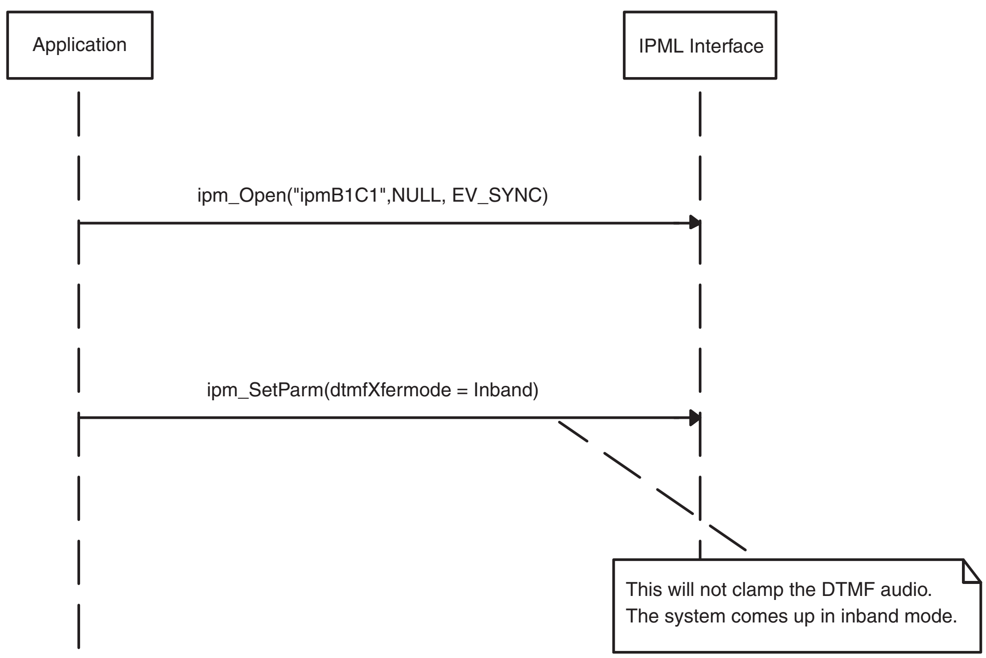
To set up a channel for in-band mode:

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

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

Figure 3 shows a scenario diagram for setting in-band mode.

Figure 3. In-Band Mode Scenario Diagram



6.2.3 Setting RFC 2833 Mode

In RFC2833 mode, the DTMF audio is clamped (muted) and DTMF digits are sent in the RTP stream only as RFC2833 packets. To set up a channel for RFC 2833 mode, do the following:

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

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

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

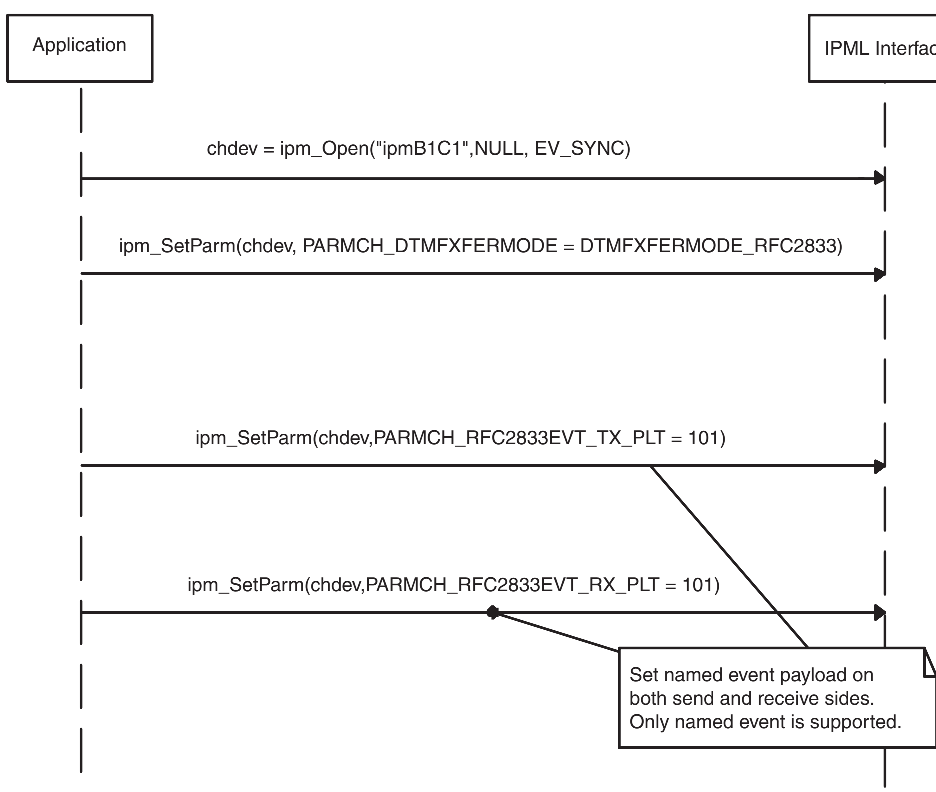
```
IPM_PARM_INFO parmInfo;
unsigned long ulParmValue = 101;
parmInfo.eParm = PARMCH_RFC2833EVT_TX_PLT;
parmInfo.pvParmValue = &ulParmValue
ipm_SetParm(chdev, &parmInfo, EV_ASYNC)
```

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

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

Figure 4 shows a scenario diagram for setting RFC 2833 mode.

Figure 4. RFC 2833 Scenario Diagram



6.2.4 Setting Out-of-Band Mode

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

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

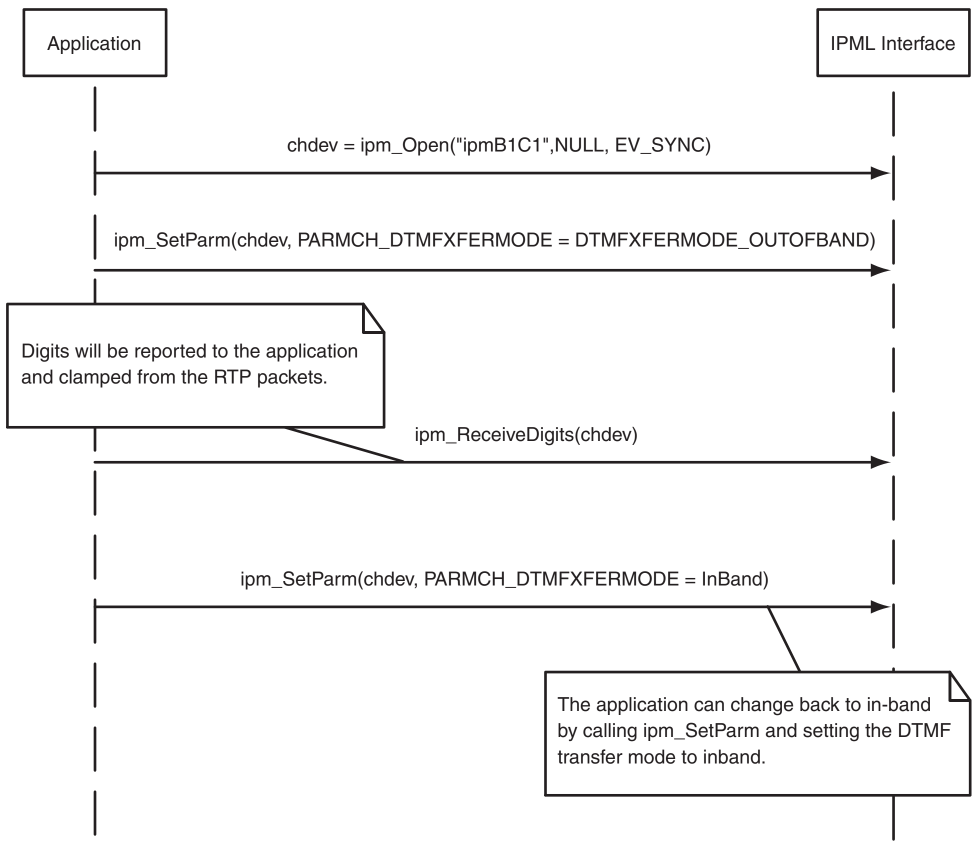
```
IPM_PARM_INFO parmInfo;
unsigned long ulParmValue = DTMFXFERMODE_OUTOFBAND;
parmInfo.eParm = PARMCH_DTMFXFERMODE;
parmInfo.pvParmValue = &ulParmValue
ipm_SetParm(chdev, &parmInfo, EV_ASYNC)
```

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

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

Figure 5 shows a scenario diagram for setting out-of-band mode.

Figure 5. Out-of-Band Mode Scenario Diagram



6.3 Notification of DTMF Detection

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

6.4 Generating DTMF

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

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

T.38 Fax Server 7

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

7.1 Using the T.38 Fax Server

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

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

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

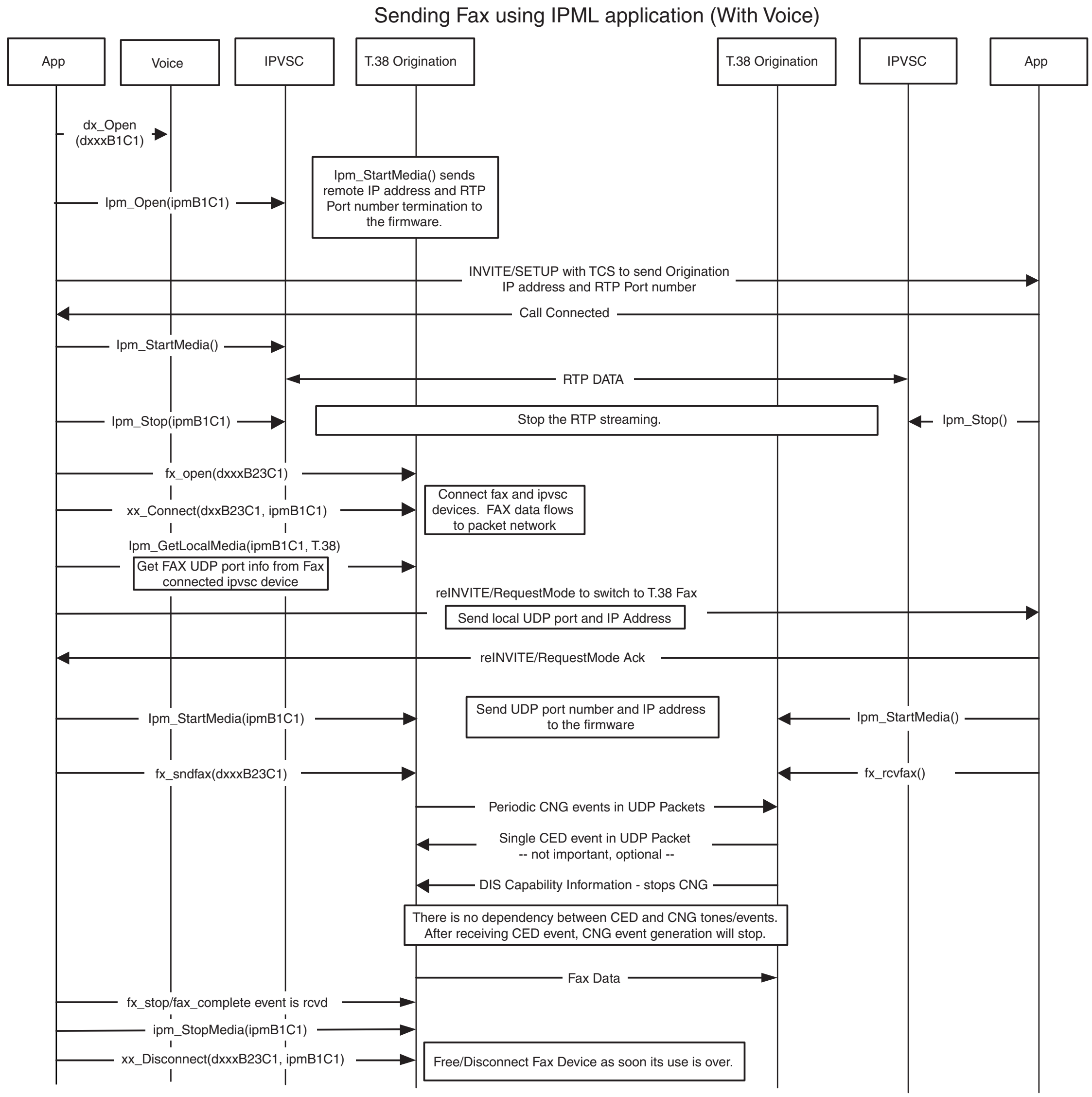
1. Open an ipm channel using **ipm_Open( )**. For example:
   ```
   ipmDevH1 = ipm_open("ipmB1C1")
   ```
2. Open a dxxx channel to be used for fax using **dx_open( )**. For example:
   ```
   dxDevH1 = dx_open("dxxxB17C3")
   ```
3. Issue **dx_getfeaturelist( )** on the dxxx channel to verify that this channel supports fax. For example:
   ```
   dx_getfeaturelist(dxDevH1,feature_tablep)
   ```
4. Verify that **dx_getfeaturelist( )** returns FT_FAX for ft_fax bitmask in the FEATURE_TABLE structure. For example:
   ```
   if (feature_tablep->ft_fax & FT_FAX)
   ```
5. Open the same dxxx channel using **fx_open( )**. For example:
   ```
   faxdevH1 = fx_open("dxxxB17C3")
   ```
6. Issue **dx_getfeaturelist( )** to determine whether this fax resource supports T.38 fax. For example:
   ```
   if (feature_tablep->ft_fax & FT_FAX_T38UDP)
   ```
7. To route the fax channel to the ipm channel, use **dev_Connect( )**. For example:
   ```
   ret = dev_Connect(ipmDevH1, faxdevH1, DM_FULLDUP, EV_ASYNC)
   ```
8. Process the DMEV_CONNECT completion event.

Note: DM_FULLDUP is the only mode supported when passing T.38 devices because the connection is made logically in both directions.

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

7.2 T.38 Fax Server Call Scenario

Figure 6. T.38 Fax Server Scenario



7.3 T.38 Fax Server Example Code

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

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

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

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

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

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

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

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

```
        if( fax_handle_disconnected )
        {
            return 0;
        }
        break;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Quality of Service (QoS) Alarms 8

This chapter describes the QoS alarms that are supported by the Dialogic® IP media software. The following topics are discussed:

8.1 QoS Overview

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

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

To enable and use QoS monitoring in your application, you must follow several steps. Some steps are optional; others are required. These steps are detailed in Section 8.5, "Implementing QoS Alarms", on page 40.

8.2 QoS Alarm Types

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

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

QOSTYPE_JITTER
: QoS alarm for excessive average jitter

QOSTYPE_LOSTPACKETS
: QoS alarm for excessive percentage of lost packets

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

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

For details on using QoS alarms in your application, see Section 8.5, "Implementing QoS Alarms", on page 40.

8.3 QoS Threshold Attributes

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

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

unTimeInterval
: time interval between successive parameter measurements

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

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

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

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

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

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

The IP Media Library software provides default values for each threshold attribute that will be used if the application does not specify any threshold values via **ipm_SetQoSThreshold( )**; the specific default values vary by alarm type. The following table provides details on the attributes supported and the default values for each QoS alarm type. Note that if the application needs to set an explicit value for any of the threshold values for a particular alarm, it must specify the values for *all* fields in the IPM_QOS_THRESHOLD_DATA structure, including those that remain at default values.

For details on the IPM_QOS_THRESHOLD_DATA structure, see the *Dialogic® IP Media Library API Library Reference*.

Table 1. Quality of Service Parameter Defaults for Dialogic® Host Media Processing Software

QoS Type	Time Interval (ms)	Debounce On (ms)	Debounce Off (ms)	Fault Threshold [1]	% Success Threshold	% Fail Threshold
Jitter	5000	20000	60000	60 (ms)	25	25
Lost Packets	1000	10000	10000	20 (%)	40	40
RTCP Timeout	1000	0	0	250 (x100ms = 25sec)	0	0
RTP Timeout	1000	0	0	1200 (x100ms = 120sec)	0	0

Notes:
1. Units for Fault Threshold are different for different QoS Types. See unit indications in table cells.

8.4 QoS Events

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

EVT_JITTER
: event indicating excessive jitter

EVT_LOSTPACKETS
: event indicating excessive percentage of lost packets

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

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

These QoS events correspond to the QoS alarms discussed in Section 8.2, “QoS Alarm Types”, on page 37. For details on enabling QoS alarms in your application, see the following section, “Implementing QoS Alarms”.

8.5 Implementing QoS Alarms

The following steps provide general guidelines for implementing QoS alarms in your application. For details on the Dialogic IP Media Library functions and data structures that are mentioned, see the *Dialogic® IP Media Library API Library Reference*.

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

1. **Optional steps before enabling a QoS alarm:**
   a. Call **ipm_GetQoSThreshold( )** to retrieve the current settings of QoS parameters on the specified IP channel. QoS parameter default values vary by alarm type and product. For information on QoS parameter default values, see the table in Section 8.3, “QoS Threshold Attributes”, on page 38.
   b. If you need to change current QoS parameter values, set up the IPM_QOS_THRESHOLD_INFO structure with desired values. This structure contains one or more IPM_QOS_THRESHOLD_DATA structures. Note that you must explicitly specify the value for *every* parameter in the IPM_QOS_THRESHOLD_DATA structure, even if you want to use the default value for some of those parameters and non-default values for other parameters.
   c. Call **ipm_SetQoSThreshold( )** to use the QoS parameter values set in step 1b.
2. **Enable QoS alarms and start media streaming:**
   a. Call **ipm_EnableEvents( )** to enable QoS monitoring for a list of alarm types.
   b. Call **ipm_StartMedia( )** to start media streaming and begin QoS monitoring.
3. **Monitor QoS alarm notification events:**
   a. When a QoS alarm has been triggered, an IPMEV_QOS_ALARM event is generated by the system. Call the Standard Runtime Library function **sr_getevttype( )** to return the event type.
   b. Use Dialogic® Standard Runtime Library API functions such as **sr_getevtdatap( )** to query the IPM_QOS_ALARM_DATA structure to learn whether the alarm state is on or off.

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

For example code that illustrates how to implement QoS alarms, see Section 8.7, “Example Code for QoS Alarm Handling”, on page 44.

8.6 QoS Alarm and Alarm Recovery Mechanisms

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

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

These scenarios are intended to illustrate the concepts. For easier reference, in the figures, time is shown in seconds rather than in millisecond units. For details on the parameters, see the *Dialogic® IP Media Library API Library Reference*.

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

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

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

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

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

For example code that uses these QoS parameter values, see Section 8.7, "Example Code for QoS Alarm Handling", on page 44.

Scenario 1: Brief Alarm Condition

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

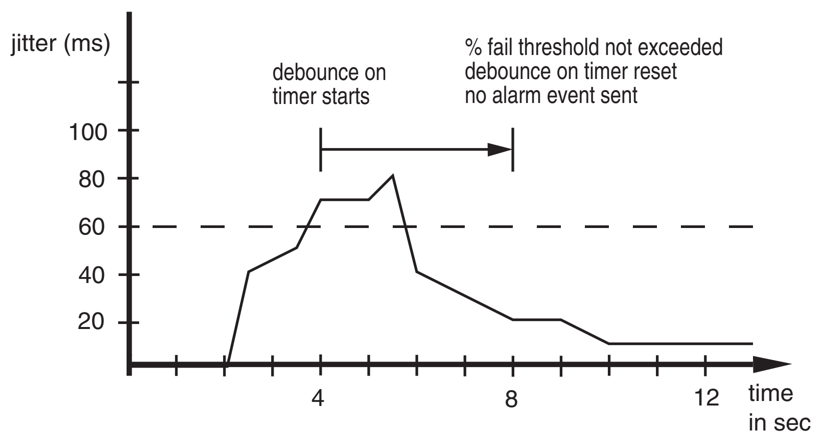
In Figure 7, the time line shows that QoS parameters are measured every time interval (**unTimeInterval** parameter), or every 1 second in this case. When the jitter exceeds the 60ms fault threshold (**unFaultThreshold** parameter), the debounce on timer is kicked off (**unDebounceOn** parameter). In this example, the fault threshold is exceeded at the 4th second.

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

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

Figure 7. QoS Scenario 1: Brief Alarm Condition

QoS parameters:
time interval = 1 sec
debounce on = 4 sec
debounce off = 4 sec
fault threshold = 60ms
% success threshold = 50 %
% fail threshold = 50 %



Scenario 2: True Alarm Condition

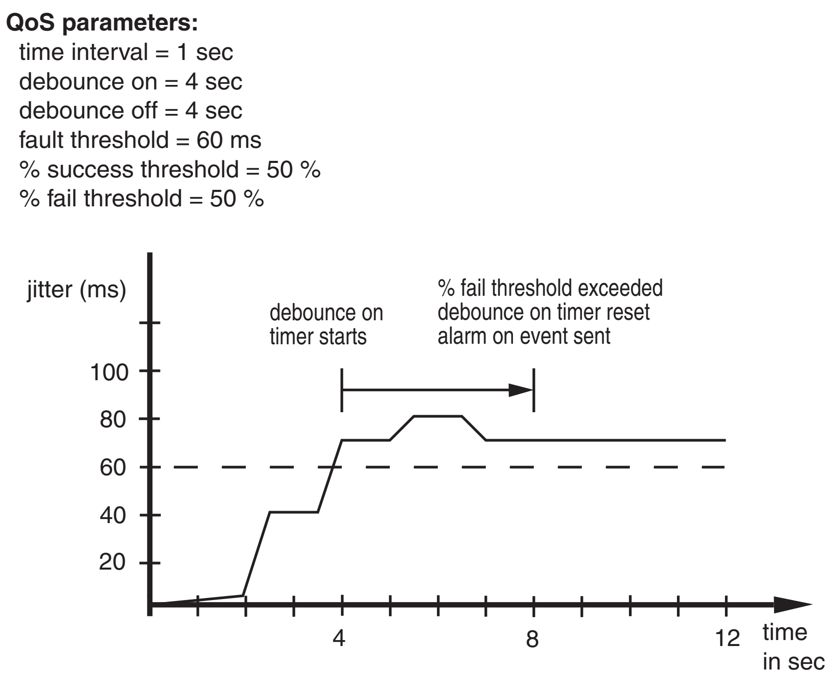
This scenario illustrates that a QoS alarm is triggered, and the alarm condition meets all of the specified alarm criteria. Therefore, an alarm-on event is sent to the application.

In Figure 8, the time line shows that QoS parameters are measured every time interval (**unTimeInterval** parameter), or every 1 second in this case. When the jitter exceeds the 60ms fault threshold (**unFaultThreshold** parameter), the debounce on timer is kicked off (**unDebounceOn** parameter). In this example, the fault threshold is exceeded at the 4th second.

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

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

Figure 8. QoS Scenario 2: True Alarm Condition



Scenario 3: Alarm Condition Cleared

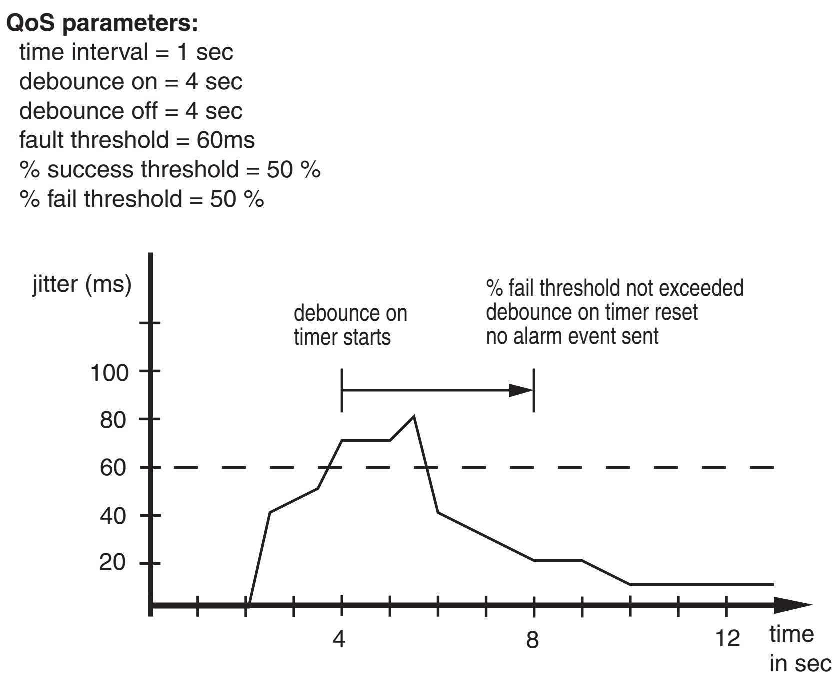
Scenario 3 builds on Scenario 2 to illustrate what happens after an alarm-on event is sent to the application and how the alarm-on condition is cleared.

In Figure 9, an alarm-on event was sent to the application at the 8th second, and the system is now in a QoS failure condition. To determine how long this condition will last, the system resumes monitoring the jitter every time interval (**unTimeInterval** parameter), or every 1 second in this case. When the jitter is less than the 60ms fault threshold (**unFaultThreshold** parameter), the debounce off timer kicks in (**unDebounceOff** parameter). In this example, this condition occurs at the13th second.

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

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

Figure 9. QoS Scenario 3: Alarm Condition Cleared



8.7 Example Code for QoS Alarm Handling

The following pseudocode illustrates how you might use QoS alarms in an application. The code enables the following QoS alarm types: . Because the IPM_QOS_THRESHOLD_INFO structure is not filled in for the lost packets alarm type, the default QoS parameter values are used for this alarm. The QoS parameter values for jitter are the same values used in the scenario descriptions in Section 8.6, “QoS Alarm and Alarm Recovery Mechanisms”, on page 41.

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

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

```
void main()
{
   int nDeviceHandle;
   IPM_QOS_THRESHOLD_INFO mySetQosThresholdInfo;

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

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


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


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

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


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

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

    /*
    .
    . Continue Processing
    .
    */

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

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

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

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

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

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

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

Volume Control 9

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

9.1 Volume Control Overview

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

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

9.2 Volume Control Parameters

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

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

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

For details on these functions and data structure, see the *Dialogic® IP Media Library API Library Reference*.

9.3 Implementing Volume Control

To implement volume control for an IP device in your application, follow these steps:

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

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

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

9.4 Volume Control Hints and Tips

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

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

9.5 Volume Control Example Code

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

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

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

```
void CheckEvent();

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

   /*
   .
    main processing
   .
   */

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

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

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

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

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

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

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

Building Applications 10

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

10.1 Compiling and Linking under Linux

The following topics discuss compiling and linking requirements:

- Include Files
- Required Libraries

10.1.1 Include Files

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

ipmerror.h
: Dialogic IP media library error header file

ipmlib.h
: Dialogic IP media library header file

10.1.2 Required Libraries

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

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

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

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

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

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

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

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

10.2 Compiling and Linking under Windows®

The following topics discuss compiling and linking requirements:

- Include Files
- Required Libraries

10.2.1 Include Files

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

ipmerror.h
IP media library error header file

ipmlib.h
IP media library header file

10.2.2 Required Libraries

The following library files must be linked to the application:

libipm.lib
Linking this file is mandatory.

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

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

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

Index

J

L

M

N

O

P

Q

R

S

T

V