

**Application Note**

**Creating a VoIP Gateway  
with the Dialogic<sup>®</sup> MSP 1010  
Multi-Services Platform and  
Dialogic<sup>®</sup> PowerMedia™ Host  
Media Processing Software**

# Creating a VoIP Gateway with the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software

Application Note

## Executive Summary

This application note describes a VoIP Gateway Demonstration Application that uses the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software to create a VoIP (TDM-to-IP) gateway that routes an incoming PSTN call to SIP. In the application, both MSP Dialogic® MSP SwitchKit® Development Environment Release 10.4.0 and PowerMedia HMP run on the application server, which communicates with the MSP 1010 for performing PSTN and RTP control, and PowerMedia HMP for performing SIP call control.

This application note includes an introduction to the VoIP Gateway Demonstration Application, including required components, the network setup and detailed call flow and source code available for download. The actions required for initializing, setting up and tearing down the call, retrieving events, and building and running the application are also covered.

**Note:** Dialogic® Host Media Processing Software has joined the Dialogic® PowerMedia™ Product Family and is now known as Dialogic® PowerMedia™ Host Media Processing Software (PowerMedia HMP).

# Creating a VoIP Gateway with the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software

Application Note

## Table of Contents

|  |    |
|--|----|
| Introduction to the VoIP Gateway Demonstration Application. . . . .  | 2  |
| Required Components. . . . .   | 2  |
| Network Setup . . . . .  | 3  |
| Call Flow . . . . .  | 4  |
| Initialization and Resource Discovery . . . . .  | 5  |
| Initializing the Dialogic® SwitchKit® API. . . . .   | 5  |
| Initializing and Starting Dialogic® PowerMedia™ Host Media Processing Software<br>Release 3.0WIN . . . . . | 7  |
| Call Setup and Teardown . . . . .  | 8  |
| Event Retrieval. . . . .   | 8  |
| Building and Running the VoIP Gateway Demonstration Application . . . . .                                  | 9  |
| Assumptions and Prerequisites . . . . .  | 9  |
| Building the VoIP Gateway Demonstration Application . . . . .  | 9  |
| Running the VoIP Gateway Demonstration Application . . . . .   | 10 |
| User-Defined Classes and Functions in the VoIP Gateway Demonstration Application . . . . .                 | 10 |
| ADevice Class . . . . .  | 10 |
| CSipDev . . . . .  | 11 |
| CSipMgr . . . . .  | 13 |
| For More Information . . . . .   | 16 |

# Creating a VoIP Gateway with the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software

Application Note

## Introduction to the VoIP Gateway Demonstration Application

The VoIP Gateway Demonstration Application (also called the “application” in this application note) includes an example of a TDM-to-IP scenario. In this scenario, the incoming PSTN call is bridged to the outbound SIP call, so the caller and the SIP user agent called can talk to each other. This is accomplished using the Dialogic® MSP SwitchKit® Development Environment with Dialogic® PowerMedia™ Host Media Processing Software Release 3.0WIN running on the application server, which communicates with the Dialogic® MSP 1010 Multi-Services Platform for performing PSTN and RTP control, and PowerMedia HMP 3.0 for performing SIP call control.

This application note includes the required components for using the application, and steps for setting up the network, initializing the Dialogic® SwitchKit API and Dialogic® Global Call API, setting up and tearing down the call, retrieving events, and building and running the application. For convenience, sample source code is also provided as an example only and it is not intended for production (see “For More Information”).

The developer can implement other, more complex scenarios using the Global Call API-based and SwitchKit API-based classes provided by the application.

## Required Components

The VoIP Gateway Demonstration Application requires the following:

- Dialogic® MSP 1010 Multi-Services Platform (MSP 1010)
- Dialogic® SwitchKit® API (SwitchKit API)
- Dialogic® PowerMedia™ Host Media Processing Software Release 3.0WIN (PowerMedia HMP 3.0 or HMP 3.0)

### ***Dialogic® MSP 1010 Multi-Services Platform***

The MSP 1010 is a multi-functional platform that supports TDM and IP interfaces, as well as fixed and mobile signaling protocols. It has a comprehensive graphical user interface for Operations, Administration, Maintenance, and Provisioning (OAM&P) and a C++ API for developing call processing applications. The MSP 1010 simultaneously supports signaling and bearer traffic over circuit- and packet-switched networks for wireless or wireline call processing applications.

The MSP 1010 supports applications that use Short Message Service (SMS), location-based services, Intelligent Network (IN) technology, and Interactive Voice Response (IVR) technology. It also supports the following protocols:

- ITU and ANSI versions of the SS7 protocol, including MTP2/MTP3, SCCP, TCAP, and ISUP
- Various IN protocols, including CAMEL, WIN, and INAP, as well as the wireless protocols GSM-MAP and ANSI-41
- SIP, using the call processing functionality provided by PowerMedia HMP 3.0

For more detailed information regarding MSP 1010, see “For More Information.”

### ***Dialogic® SwitchKit® API***

The SwitchKit API is a library of functions used to communicate with the Dialogic® Low Level Communications (LLC) and the MSP 1010.

The SwitchKit API provides a framework that allows control over PSTN inbound and outbound calls (SS7 or ISDN). It also manages VoIP media (RTP/RTCP) and instructs the MSP 1010 to start or stop RTP sessions, or to bridge two TDM-to-TDM, TDM-to-IP, or IP-to-IP sessions from either direction together (in this way, the MSP 1010 can receive/initiate two calls and enable both endpoints to communicate).

# Creating a VoIP Gateway with the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software

Application Note

## **Dialogic® PowerMedia™ Host Media Processing Software Release 3.0WIN**

PowerMedia HMP 3.0 (HMP 3.0) is a software package designed for performing IP signaling and media processing tasks on Windows® computing platforms without using specialized hardware.

For call control over IP, HMP 3.0 supports the H.323 and SIP protocols. SIP is supported in First Party Call Control (1PCC) mode, when both signaling and RTP originate and terminate on same HMP 3.0 server, or in Third Party Call Control (3PCC) mode, when an HMP 3.0-based application is used for signaling only; that is, for controlling RTP streams originated by other servers or endpoints.

The VoIP Gateway Demonstration Application uses HMP 3.0 in 3PCC mode. In this mode, HMP 3.0 runs on the application server and is used for SIP signaling, while the MSP 1010 and the SwitchKit API control media operations based on the information (such as VoIP codec, local and remote IP ports, and addresses) provided by the SIP component of HMP 3.0.

HMP 3.0 provides two programming interfaces for signaling:

- The Global Call API is used to create SIP sessions or accept invitations from other SIP endpoints on the network. It provides high level read/write access to all SIP headers and allows for parsing and constructing Session Description Protocol (SDP) data for any SIP functions where SDP is supported.
- The Dialogic® Standard Runtime Library (SRL) API, among other tasks, delivers SIP messages to the application in the form of Global Call API events.

**Note:** The HMP 3.0 APIs are not a part of SwitchKit API or other MSP API packages. They run in their own context and share the same application space with the SwitchKit API, and they are installed separately.

## **Network Setup**

Figure 1 depicts an example network setup for the VoIP Gateway Demonstration Application.

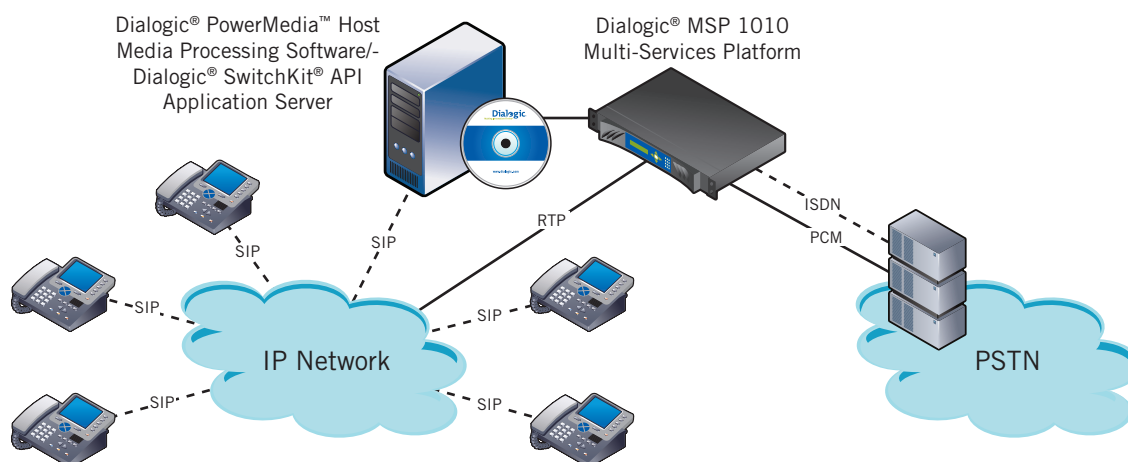


Figure 1. Network Setup for the VoIP Gateway Demonstration Application

## Call Flow

Figure 2 depicts an example call flow implemented by the VoIP Gateway Demonstration Application.

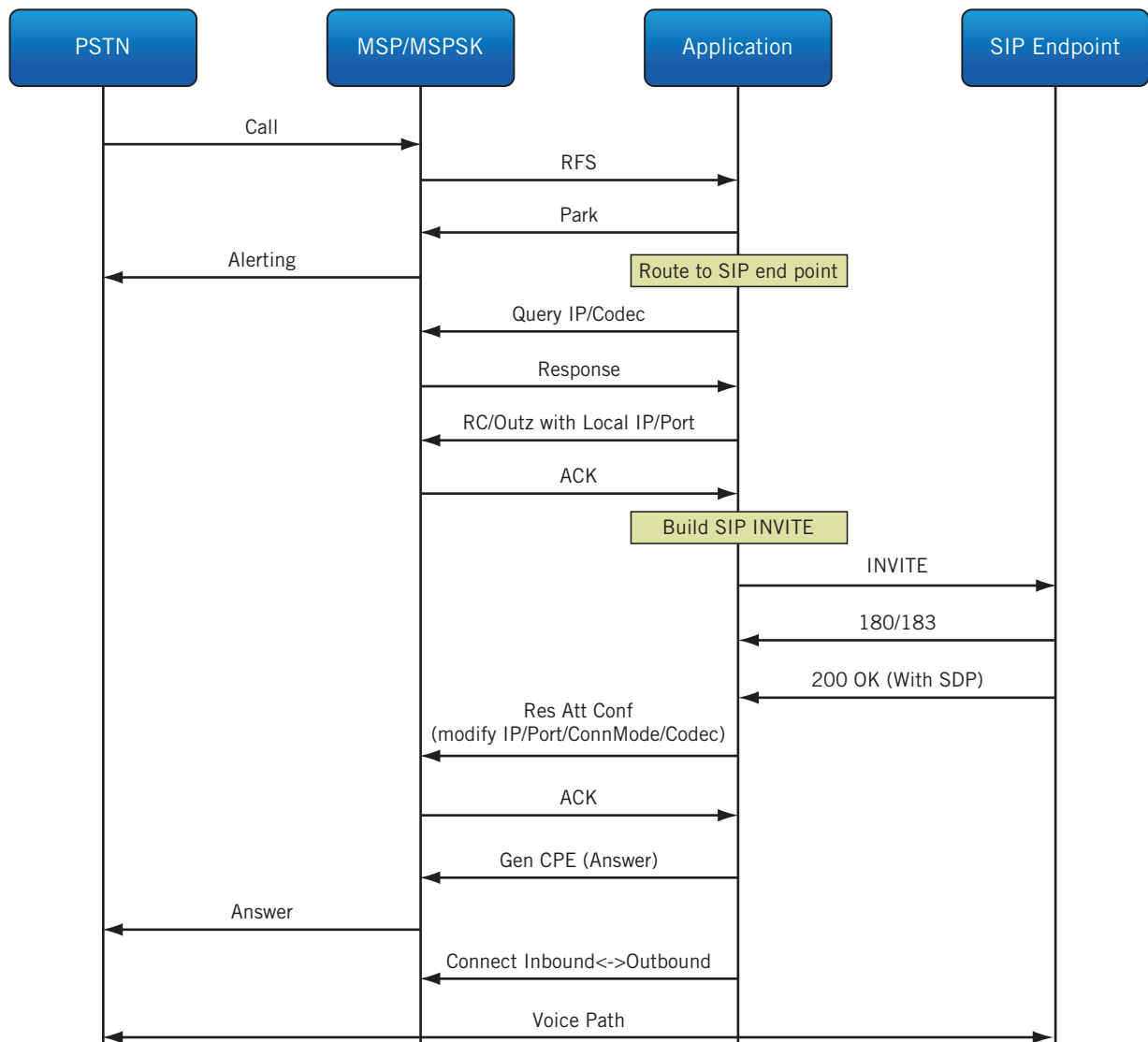


Figure 2. Call Flow for the VoIP Gateway Demonstration Application

The call flow activities are as follows:

1. PSTN caller makes a call that gets routed to the MSP 1010.
2. MSP 1010 detects the call and sends an event notification to the application.
3. The application retrieves the call information, including the call channel, span, calling/called party numbers (if relevant), and so forth.
4. The application parks the call by sending an ALERTING message on the PRI channel, or the corresponding message or tone when configured with another PSTN protocol.
5. The application hears the call ringing and makes an outbound SIP call. To make this call, the application uses the IP address in the AddressList global array, the RTP address/port obtained during application startup, and the Global Call API.
6. When the SIP call is answered, the HMP 3.0 resource retrieves the remote RTP information and passes it to the MSP 1010 via a SwitchKit API call.
7. The MSP 1010 starts a bi-directional RTP stream to the address and port provided by HMP 3.0. The calling and called parties now can talk.
8. When either party drops the call, the other leg is also released by the application, and the RTP stream stops automatically.

## Initialization and Resource Discovery

The initialization and resource discovery phase involves initializing the SwitchKit API and Global Call API.

### Initializing the Dialogic® SwitchKit® API

The VoIP Gateway Demonstration Application initializes the SwitchKit API as follows:

1. The application registers itself with the MSP 1010 by calling `skts_createConnection()` to establish a connection with the LLC and passes its arbitrary ID as an argument:

```
// Establish connections to the LLC(s)
int status = skts_createConnection(
    ConId,// connection ID this is what the application uses to
identify what connection this is
    "NULL",//This is the label, we do not use this in this app
and the API does not either.
    X_APP,// this is the number of the app id the llc identifies this
application by
    0,// isForcedFlag (set to 0)// if the isforcedFlag is set we
will use this id no matter what.
    Host1,// primary LLC ip
    1312,// primary LLC port number (1312 is default)
    Host2,// redundant LLC hostname
    1312);// redundant LLC port number (1312 is default)
if (status != 1)
{
    cout << "Error has occurred creating connection to LLC" <<
endl;
    return -1;
}
```

2. The application notifies the MSP 1010 that it wants to handle a specific channel group by calling `skts_watchChannelGroup()` and specifying the channel group identifier as an argument:

```
// Register for new calls from the specified
// channel group
int status =
skts_watchChannelGroup(ingroupName,aConID);//Tells switchkit to watch
this particular channel group for messages

    if (status != OK) {
        cout << "Error registering for messages from LLC"<<
endl;
```

3. The application calls `QueryMspConfig()` to inquire about VoIP availability:

```
CASEC_ConnectionStatusMsg(connectionStatus)
{
    if (!Connected)
    {

        Connected = true;
        cout<<"Retrieving info for VoIP"<<endl;
        QueryMspConfig(DefaultHandler, ConId);

    }
    return OK;
    break;
}
```

4. LLC returns a `CASEC_NGAConfigureQueryAck` event, which indicates whether or not the MSP 1010 has VoIP capabilities. If the acknowledge is positive and data exists, the module ID, its IP address, subnet mask, and default gateway are stored in the `XLC_NGAConfigureQuery` structure. Otherwise, the application returns an error and must exit.

**Note:** The MSP 1010 must be properly configured to support VoIP.

5. The application calls `QueryIpModuleAttributes()` to initialize the `XLC_ResourceAttributeQuery` structure with the module number and IP parameters obtained in the previous step and passes this information to the MSP 1010:

```
CASEC_UserTimerAck(uta);
{
    if(QueryIpModuleAttributes(VOIP, Offset, DefaultHandler,
ConId) != OK) //module is not configured
    {
        // request IP from the next module
        SKC_UserTimer ut;//we send a user timer here to get back
in the same case and check the next module
        Offset=Offset+1;
        ut.setSeconds(1);
        ut.tsSend(NULL,DefaultHandler, ConId);
    }
    return OK;
```

# Creating a VoIP Gateway with the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software

Application Note

6. Upon successful completion of the QueryIpModuleAttributes request, the MSP 1010 returns a CASEC\_ResourceAttributeQueryAck event.
7. The application calls OnQueryIpModuleAttributes() to process and store the data returned by the CASEC\_ResourceAttributeQueryAck event:

```
CASEC_ResourceAttributeQueryAck(raqa)
{
    if(OnQueryIpModuleAttributes(raqa, VOIP, Offset) != OK) //
negative ack
    {
        printf(" ResourceAttribute received Negative Ack with
status %X :%s\n",raqa->getStatus(),sk_statusText(raqa->getStatus()));
        // check config or increase Offset, otherwise exit.
        // TODO: increase Offset and re-send the query
    }
    else
    {
        g_sipMgr->SetLocalRtpIpAddress(VOIP[Offset].sourceIpAddr);
        if(!g_CapList)
            g_CapList = new list<IP_CAPABILITY *>;
        else
            clearCapList(g_CapList);
        IP_CAPABILITY * l_cap = new IP_CAPABILITY;
        l_cap->type = GCCAPTYPE_AUDIO;
        l_cap->direction = IP_CAP_DIR_LCLTXRX;
        l_cap->payload_type = IP_USE_STANDARD_PAYLOADTYPE;
        l_cap->extra.audio.VAD = GCPV_DISABLE;

        l_cap->extra.audio.frames_per_pkt = 20;
    }
}
```

## Initializing and Starting Dialogic® PowerMedia™ Host Media Processing Software Release 3.0WIN

The VoIP Gateway Demonstration Application initializes the HMP 3.0 as follows:

1. Calls CSipMgr::SetLocalRtpIpAddress() and CSipMgr::SetDtmfXferMode() to initialize a global list of IP capabilities with the pre-stored data.
2. Calls CSipMgr::Init() to start the HMP 3.0.

The application is now ready to process PSTN and SIP calls.

CSipMgr is a user-defined class in the application. For more information, see “CSipMgr” in this application note.

## Call Setup and Teardown

The VoIP Gateway Demonstration Application processes an inbound PSTN or SIP call as follows:

1. The application receives the CASEC\_RFSWithData event in the IngroupHandler(), which indicates an incoming PSTN call.
2. The application calls SIPMGR::GetSipChannel() to retrieve the span/channel number for the channel on which the incoming call arrived.
3. The application puts the call in an alerting state.
4. The application uses the retrieved channel to make an outbound SIP call to the IP address defined for the purposes of the application (xxx).
5. When the SIP call is answered by the remote SIP user agent, Global Call API notifies the application by putting GCEV\_CONNECTED event in SipHandler().

The SIP call is connected, and all the information about local and remote RTP capabilities, including VoIP coder, IP address information, and DTMF transfer mode is now available to the application.

6. The application calls skts\_requestOutseizedChannel() to take the following actions by configuring and sending a route control request XLC\_RouteControl message to the MSP 1010:
  - Sends a request to the MSP 1010 for the available outbound VoIP channel
  - Starts the bi-directional RTP stream between the MSP 1010 and the SIP agent
  - Answers the parked PSTN call with the help of the XLC\_GenerateCallProcessingEvent type object
7. When the PSTN call is connected (positive CASEC\_GenerateCallProcessingEventAck event detected), the PSTN and the RTP (VoIP) legs of the session are connected by constructing and sending an object of the XLC\_Connect class.
8. When the call is dropped by either party, the application receives a disconnect notification in one of the two following ways, depending on whether the call is a PSTN call or a SIP call:
  - For a PSTN call, the application receives a CASEC\_ChannelReleasedWithData event (PSTN SwitchKit API).
  - For a SIP call, the application receives a GCEV\_DISCONNECTED event (SIP HMP 3.0) and releases the caller party and the called party as well. It causes the RTP-TDM bridge to break, and the RTP stream to seize.

## Event Retrieval

The VoIP Gateway Demonstration Application uses four event handlers to process incoming PSTN, SIP, or internal events. These handlers are managed by the functions described in Table 1.

| Event Handler          | Description   |
|------------------------|---|
| DefaultHandler()       | Handles SwitchKit API events that are not handled by any other SwitchKit API event handler. The DefaultHandler() is enabled by the skts_setDefaultHandler() function. |
| IngroupHandler()       | Handles PSTN call-related messages. The IngroupHandler() is enabled by the skts_setGroupHandler() function.   |
| llcConnectionHandler() | Called when the state of a connection to an LLC changes. The llcConnectionHandler() is enabled by the skts_setLLCConnectionHandler() function.                        |
| SipHandler()           | Handles Global Call API events  |

Table 1. Event Handlers and the Functions That Manage Them

# Creating a VoIP Gateway with the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software

Application Note

For the complete syntax of these event handlers and the functions that enable them, see the [SwitchKit® Development Environment Programmer's Guide](#).

At the application start-up, after setting the handlers above, the application enters a loop where it waits for an event to occur by sharing a time between two waiting functions:

- **skts\_rcvAndDispatchAutoStorage()** — Waits for a SwitchKit API event. When a SwitchKit API event is detected, the SwitchKit API framework calls the appropriate event handler for that event.
- **sr\_waitvt()** — Waits for Global Call API events. When a Global Call API event occurs, the application calls the CSipMgr::ProcessEvent() function. This function sends the event to the appropriate CSipDev object for pre-processing and then invokes *SipHandler()* for further, higher-level handling.

## Building and Running the VoIP Gateway Demonstration Application

### Assumptions and Prerequisites

Prior to building and running the VoIP Gateway Demonstration Application, HMP 3.0 must be properly installed on the machine, and the proper license(s) obtained and activated. For installation and licensing procedures, see:

- [Dialogic® Host Media Processing Software](#)
- [Dialogic® Host Media Processing Software Release 3.0WIN](#)

For simplicity, the application uses some global variables with pre-assigned values. The following assumptions are made:

- The inbound channel group used for PSTN traffic is called “Inbound-ISDN”. The name of this channel group can be changed by overriding the **char \*ingroupName** global variable.
- The outbound channel group used for VoIP is called “Outbound-IP”. It is hardcoded in the **\*outroupName** global variable.
- The IP address used by outbound SIP calls is selected from the **char \* ip\_addressList[ ]** array, which currently has one entry: {“10.129.50.138”, NULL}; The user may define a different address list by changing the default address and adding more addresses, so that different SIP calls use different destination addresses. The address list is only used in **I\_sip->MakeCall(ip\_addressList[0])** call; the array index can be incremented dynamically for each new SIP call.
- The IP address of the local machine where the application is running is defined by the **AppServerIpAddr** global variable and used for SIP signaling. If the application server uses more than one Network Interface Controller (NIC), then either IP address can be used for SIP. The IP address must be pingable from the network where the SIP agent is located.
- For connection to the LLC, at least the “**SK\_LLC\_HOST**” environment variable must be defined and populated with the IP address of the primary LLC.
- The firewall on the application server must allow TCP and UDP ports 1720 and 6050 to be used by the application.

### Building the VoIP Gateway Demonstration Application

To build the VoIP Gateway Demonstration Application, link the application to the following libraries:

- SwitchKit API: *SKApi\_ts.lib*
- Global Call API: *libsrlmt.lib libgc.lib* and *SdpApi.lib Ws2\_32.lib*

The project settings or make file must have hmp/inc in the include path and hmp/lib in the library directory list. The location of the SdpApi libraries and include files is arbitrary.

## Running the VoIP Gateway Demonstration Application

To run the VoIP Gateway Demonstration Application, do the following:

1. Prepare the application server for running SwitchKit API-based applications.
2. Read the “Assumptions and Prerequisites” mentioned previously in this application note; configure the application and the environment accordingly.
3. Build the application.
4. Run the application; no command line arguments required.
5. After a few seconds, the message “Ready and waiting to receive calls” must appear on the screen, and the window must stop scrolling down. At this point, test PSTN calls can be started.

## User-Defined Classes and Functions in the VoIP Gateway Demonstration Application

The SwitchKit API defines a collection of high level C++ classes that developers can use for communicating with MSP 1010 servers, controlling PSTN calls, and controlling RTP functions. Unlike the SwitchKit API, the Global Call API is a collection of relatively low-level C-style functions and structures.

To mask low-level SIP aspects and provide an interface more suitable for object-oriented programming, the Demonstration Application defines the following classes and functions to manipulate VoIP signaling:

User-defined classes:

- ADevice
- CSipDev
- CSipMgr

The details for each class and function are described in the following sections.

### ADevice Class

Declared in Device.h; implemented in Device.cpp.

An abstract interface class that serves as a basis (superclass) for the CSipDev class, which is based on the Global Call API.

The ADevice class defines the following public methods for the derived classes:

- setSpan()
- getChannel()
- getSpan()
- setTag()
- getTag()
- Restart()
- DevHandle()
- DeviceState()

**void setSpan(int a\_span)** — Stores a virtual span number of this channel into the `m_spanNumber` private field. The span number is obtained from SwitchKit API during device discovery.

**int getSpan()** — Returns a virtual span number of this channel stored in the `m_spanNumber` private member.

**int getChannel()** — Returns a virtual channel number of the device object. The channel number is assigned to the object during device initialization (from within a specific `Open()` function).

**void setTag(void \* a\_tag) { m\_tag = a\_tag;};** —  
**void \* getTag() const {return m\_tag;};** —

Assigns or returns some arbitrary context to/from the `m_tag` member. This demonstration uses a pointer to a *CallRecord* object, which holds channel-specific information, like a trunk number and a channel number. This tag is mainly used for binding a PSTN channel to a SIP device. For example, when the SIP device receives a remote disconnect event, it uses the trunk/channel information of this tag to drop a call on a paired PSTN channel. The class does not perform a check on the passed argument; the application must make sure to pass a valid context when calling the `setTag()` method.

**void Restart()** —Calls `gc_ResetLineDev` under the hood. Used for bringing the SIP device to an idle state, as an error recovery, or for blocking inbound SIP calls on that specific device. Does not receive arguments; does not return any values. The application must wait for `GCEV_RESETLINEDEV` before performing any further action on this channel.

**inline long DevHandle() const { return (long)m\_hDevHandle;};** —  
**inline int DeviceState() const {return m\_deviceState;};** —

The two inline accessory functions above return a device handle obtained during execution of the `Open()` method, and the current device state, respectively. The device states are defined in derived classes (in this case, in *CSipDev*). The device state depends on a call state and is being updated upon receiving a state transition event in the event handler. For example, receiving the `GCEV_CONNECTED` event case causes the previous `DIALING_STATE` to be changed to a `CONNECTED_STATE`:

```
case GCEV_CONNECTED:
    m_deviceState = CONNECTED_STATE;
    ...
    break;
```

## CSipDev

Declared in `SipDev.h`, implemented in `SipDev.cpp`, implements *ADevice*

Concrete class for making and receiving SIP calls, modifying SIP sessions via re-Invite and setting and retrieving SIP SDP information. Important public methods used by the demonstration include:

- `Open()`
- `Close()`
- `ProcessEvent()`
- `MakeCall()`
- `Drop()`
- `Release()`
- `SetDtmfXferMode()`
- `SetCapabilities()`, `SetDefaultCapabilities`
- `SetLocalPort()`, `SetLocalAddress()`

# Creating a VoIP Gateway with the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software

Application Note

**int Open(int a\_chanNum)** — Initializes a SIP channel and assigns the unique channel number as a device ID. This API calls the gc\_OpenEx() Global Call function, which allocates resources necessary for the SIP instance and returns a unique device handle. This handle is stored as a private data and is internally used for call control or management functions on this device handle. **int Open(int a\_chanNum)** is called only once per channel. Once opened, the device handle should remain open during a lifetime of the application.

Returns 0 (RC\_SUCCESS) on success, -1 (RC\_FAILURE) on error.

**int Close()** — Closes the channel's handle and frees resources. Called by the CDevMgr upon the application's exit.

Returns 0 (RC\_SUCCESS) on success, -1 (RC\_FAILURE) on error.

**int MakeCall(const char \*a\_calledAddress)** — Creates a SIP call to the user specified by the IP address in a\_calledAddress. The address is an ASCII string of format user@ip\_addr | domain. This method internally invokes gc\_MakeCall() function of the Global Call API library and sets the data needed for a successful SIP call.

**int ProcessEvent(METAEVENT \*a\_evt)** — Called from within the **CSipMgr->ProcessEvent()** when the SIP message is received or another Global Call API event occurs. Retrieves the data associated with the event and processes it as appropriate. METAEVENT is a structure returned by the Global Call API library when a Global Call API event is detected. METAEVENT contains an event type, the device (channel) where the event occurs, and the additional data associated with the event. This is how the event processing is cascaded through the application code:

In the event loop of main() function:

```
...
    if(sr_waitevt(10) >= 0) // waiting for a GlobalCall (SIP) event for 10
ms, if occurs
        g_sipMgr->ProcessEvent(); // have the SipMgr to process the event

In CSipMgr::ProcessEvent():

...
    METAEVENT l_event;
    if(gc_GetMetaEvent(&l_event) == -1) // retrieve the METAEVENT
    {
        logOut(ERR1, "gc_GetMetaEvent() failed", 1);
        return -1;
    }
    // determine the SIP channel (CSipDev object) where the event occurred:

    CSipDev * l_evtDev = getDeviceByHandle(l_event.evtdev);
    if(l_evtDev)
        l_evtDev->ProcessEvent(&l_event); // pass this event for the
processing to the CSipDev::ProcessEvent() method
```

And finally, returns this event to the SIP event handler in the main module for a high-level ultimate processing:

```
m_evtHdlr(l_event.evttype, l_evtDev);
```

# Creating a VoIP Gateway with the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software

Application Note

**int Drop(int a\_cause = GC\_NORMAL\_CLEARING)** — Interrupts the current SIP session by sending BYE (forward release) or 200OK on incoming BYE request. The function receives one argument that specifies the reason for disconnect, when the session is interrupted before getting to the connected state. Possible arguments are defined in `gcip_defs.h`, provided with HMP 3.0 SDK, and are prefixed with `IPEC_SIPReason`. When dropping a call in the connected state, the function does not require any arguments.

Returns 0 (`RC_SUCCESS`) on success, -1 (`RC_FAILURE`) on error.

**int Release()** — Must be called upon `GCEV_DROPCALL` event to clean up resources used for the SIP call and to release the memory used by the library to handle the call. Does not receive any arguments.

Returns 0 (`RC_SUCCESS`) on success, -1 (`RC_FAILURE`) on error.

**int SetDtmfXferMode(int a\_mode, int a\_payload = 101)** — Fixes the DTMF transfer mode for a given channel or call. See “CSipMgr” description for more details.

**int SetCapabilities(list<IP\_CAPABILITY \*> \*a\_capList)** — Defines audio capabilities (coders) to be used in the SIP session. Must be called before `MakeCall()`. Receives a list of coders supported by the platform from the `CSipMgr`.

Returns 0 (`RC_SUCCESS`) on success, -1 (`RC_FAILURE`) on error.

**void SetLocalPort(USHORT a\_port), void SetLocalAddress(UCHAR \*a\_ipaddr)** — These methods are called at initialization to inform the object on what port and IP address will be used for RTP on all calls on that channel. The information is obtained and set by the `CSipMgr`.

In addition to the methods described above, the application defines, but does not use, other APIs for processing incoming SIP calls (**Accept()**, **Answer()**), modifying the existing SIP session with Re-Invite (**RequestModifyCall()**, **AcceptModifyCall()**, **RejectModifyCall()**), or handling T.38 Fax calls over IP (**GetSwitchCodecResult()**, **SwitchToFax()**, **SwitchToAudio()**). Even though the application implements a real working code for these methods, they are only included here to demonstrate more SIP scenarios, which the current application does not use. The developer can read detailed information on the scenarios at [Dialogic® Global Call IP Technology Guide](#).

## CSipMgr

Declared in `SipMgr.h`, implemented in `SipMgr.cpp`

This is a singleton class, and contains common data for all Global Call API SIP devices. It also creates all instances of **CSipDev** class and stores them in the private array. The application can set or retrieve the common SIP data, such as a local (MSP 1010) IP address used for RTP on a given trunk, DTMF transfer mode, VoIP coders to be used, and so on, via its public methods, which include:

- `SetRtpPortBase()`
- `SetLocalRtpIpAddress()`
- `SetHandler()`
- `SetDtmfXferMode()`
- `Init()`
- `GetDtmfXferMode()`
- `GetSipChannel()`
- `Shutdown()`

# Creating a VoIP Gateway with the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software

Application Note

**void SetRtpPortBase(unsigned short a\_portB)** — Sets the first RTP port that will be assigned for the first SIP channel. In this application, each SIP device works only with its own port for RTP/RTCP; the RTP port assigned via void SetRtpPortBase() will be included as an audio port in the SIP SDP where appropriate. This is not a requirement of the Global Call API, and each channel may use an arbitrary port in the SDP as long as the MSP 1010 is capable of working with that port. In this application, the ports are fixed to the logical SIP channels. This method receives a UDP port number as an *unsigned short* and does not return any value.

**void SetLocalRtpIpAddress(UCHAR \*a\_addr)** — Sets the IP address for RTP. This address is obtained via the MSP 1010 resource Attribute Query (see the example that follows for details). Along with the RTP port previously described in void **SetRtpPortBase(unsigned short a\_portB)**, the RTP IP address defines UDP connection information that will be passed to the remote User Agent in the SIP SDP block and that will be used for VoIP traffic.

The method receives a character array where the address is specified in a form of ASCII string (like 10.10.10.10). The void SetLocalRtpIpAddress() does not perform checking of the arguments and does not return any value

**int Init(int a\_numOfChannels, char \* a\_ipaddr)** — Must be called to initialize Global Call API libraries and components. By the time the application calls Init(), the UDP port base, local IP address, and Global Call event handler must be set. The Init() receives two arguments: the number of SIP channels the application will be able to use simultaneously, and the IP address of the application server, which will be used for SIP signaling. This API returns -1 on error and 0 on success. Below is an example of CSipMgr initialization sequence:

```
// At the beginning of main():
g_sipMgr = new CSipMgr;
g_sipMgr->SetRtpPortBase(49152);
g_sipMgr->SetHandler(SipHandler);

// In the SK Default handler, when Resource Attribute Query returns the data:

CASEC_ResourceAttributeQueryAck(raqa)
{
    g_sipMgr->SetLocalRtpIpAddress(VOIP[Offset].sourceIpAddress);
    int l_xferMode =
VOIP[Offset].RFC2833?IP_DTMF_TYPE_RFC_2833:IP_DTMF_TYPE_INBAND_RTP;
    g_sipMgr->SetDtmfXferMode(l_xferMode);
    // by now, IP address, RTP port base and codecs are set
    if(g_sipMgr->Init(23, AppServerIpAddress) == -1)
    {
        cout<<"Cannot initialize SIP channels, quitting"<<endl;
        return -1;
    }
    cout<<"Ready and waiting to receive calls"<<endl;
}

return OK;
```

# Creating a VoIP Gateway with the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software

Application Note

**void SetHandler(hmpEvtHdr a\_handler)** — When the application's event loop detects a Global Call API (SIP) event, it passes the event to the *ProcessEvent* method of the CSipMgr. The CSipMgr performs preliminary processing and sends this event back to the application-defined Global Call API event handler for further high-level processing. The address of the event handler to be used is set by this method. The event handler must have the following prototype:

```
void (*hmpEvtHdr)(int, CSipDev *);
```

For example, the Global Call API handler is defined in the main module (IpGateway.cpp) as *void SipHandler(int a\_event, CSipDev \* a\_evtDev)*.

The **SetHandler()** does not return any value.

**void SetDtmfXferMode(int a\_dtmfXferMode)**

**int GetDtmfXferMode()** — DTMF digits may be sent over SIP in different forms, like in-band, as a usual audio signal in a digital form when the RTP packets are filled with a special payload indication of the digit, or out-of-band using SIP INFO or NOTIFY methods. In order to process the digit correctly, the MSP 1010 must know what form of the DTMF will be used in a session, and the Global Call API must set proper SDP information. The SIP Manager obtains the supported DTMF mode from the MSP 1010 Resource Attribute Query and stores it for further use in SDP information. When instructed by the MSP 1010 to start an RTP stream for a given session, the SIP Manager retrieves the DTMF transfer mode from its private storage and passes it to the MSP 1010 along with other information necessary for VoIP, like connect information, coder information, and so on.

DTMF transfer mode is defined in the *gcip\_defs.h* header file, which is a part of Dialogic® SDK for Global Call IP and must be included with Global Call API applications (see *SipDev.h*). The currently supports the following values:

```
IP_DTMF_TYPE_RFC_2833 // 0x02
```

```
IP_DTMF_TYPE_INBAND_RTP // 0x04
```

**CSipDev \* GetSipChannel(int a\_span, int a\_channel)** — Often when a telephony event is detected in the SwitchKit API event handler, the application must take an appropriate action on a corresponding SIP channel (like stopping RTP and disconnecting the SIP session if the PSTN leg hangs up). To each Global Call API (SIP) device and SwitchKit API device, a unique combination of span/channel is assigned at the application startup. By having the channel and span number, the SIP Manager can find the SIP device with these attributes.

The method returns NULL if no SIP devices with the given span/channel can be found. Otherwise, it returns a pointer to the CSipDev object with this span/channel assigned.

**Note:** For simplicity, the current version of the application uses 1 span of SIP devices. So, the Span value in this API is not used and is always considered valid. The search only uses the Channel argument.

**int getSpan()** — Returns a virtual span number of this channel stored in *m\_spanNumber* private member.

**int getChannel()** — Returns a virtual channel number of the device object. The channel number is assigned to the object during device initialization (from within a specific *Open()* function).

**void ShutDown()** — Cleans up resources, de-allocates memory used by the SIP Manager, and unloads Dialogic® libraries from the address space. It must be called right before the exit of the application in order to have a clean teardown. Does not receive arguments; does not return any value.

# Creating a VoIP Gateway with the Dialogic® MSP 1010 Multi-Services Platform and Dialogic® PowerMedia™ Host Media Processing Software

Application Note

## For More Information

A Zip file containing the source code is available for download at [www.dialogic.com/goto/?12067](http://www.dialogic.com/goto/?12067)

*Dialogic® Global Call IP Technology Guide*

*Dialogic® MSP 1010 Multi-Services Platform*

*Dialogic® PowerMedia™ Host Media Processing Software*

*SwitchKit® Development Environment Programmer's Guide*

For detailed information about the Dialogic functions used in this application note, see *Dialogic® MSP 1010 Multi-Services Platform WebHelp*, specifically the chapter called “SwitchKit Programmer Information,” and the *SwitchKit® Threadsafe Library*.

For information about SDP API, please contact [Dialogic Technical Support](#).

[www.dialogic.com](http://www.dialogic.com)

**Dialogic Corporation**  
9800 Cavendish Blvd., 5th floor  
Montreal, Quebec  
CANADA H4M 2V9

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

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

Dialogic may make changes to specifications, product descriptions, and plans at any time, without notice.

Dialogic, PowerMedia, and Switchkit are either registered trademarks or trademarks of Dialogic Corporation. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic's trademarks requires proper acknowledgement.

Windows is a trademark of Microsoft Corporation in the United States and/or other countries. Other names of actual companies and products mentioned herein are the trademarks of their respective owners. Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement their concepts or applications, which licenses may vary from country to country.

Any use case(s) shown and/or described herein represent one or more examples of the various ways, scenarios or environments in which Dialogic® products can be used. Such use case(s) are non-limiting and do not represent recommendations of Dialogic as to whether or how to use Dialogic products.