Dialogic®

Dialogic® Global Call IP

Technology Guide

May 2014

05-2239-012

Publication Date: May 2014

Document Number: 05-2239-012

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
05-2239-012	May 2014	Incorporated documentation updates. IP-Specific Parameters chapter: Added IPPARM_SIP_VIA_HDR_REPLACE row to IPSET_SIP_MSGINFO Parameter Set table in IPSET_SIP_MSGINFO section. IP-Specific Operations chapter: Updated code example in Retrieving SIP Message Header Fields section.
05-2239-011	May 2012	Updated for Dialogic® HMP Software for Linux and for Windows. Incorporated features and documentation updates from Release Updates for Release 3.0, Release 3.1, and Release 4.1.
05-2239-010	March 2008	Updated to correct faulty formatting in some tables.
05-2239-009	November 2007	Made global changes to reflect Dialogic brand.
05-2239-008	August 2006	Updated.
05-2239-007	December 2005	Updated.
05-2239-006	August 2005	Updated.
05-2239-005	April 2005	Updated.
05-2239-004	January 2005	Updated.
05-2239-003	September 2004	Updated.
05-2239-002	April 2004	Updated.
05-2239-001	September 2003	Initial version of document.

About This Publication

The following topics provide information about this publication.

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

Purpose

This publication describes the Dialogic® Global Call API for IP technology as it is implemented in Dialogic® Host Media Processing (HMP) Software. The Dialogic® Global Call API implementation in Dialogic® System Release software is documented in a separate set of documents.

This guide is for users of the Dialogic® Global Call API who are writing applications that use host-based IP H.323 or SIP technology. The Dialogic® Global Call API provides call control capability and supports IP Media control capability. This guide provides Dialogic® Global Call API IP-specific information only and should be used in conjunction with the *Dialogic® Global Call API Programming Guide* and the *Dialogic® Global Call API Library Reference*, which describe the generic behavior of the Dialogic® Global Call API.

Applicability

This document version is published for Dialogic® Host Media Processing (HMP) Software Release 4.1 and 3.0.

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

Intended Audience

This guide is intended for:

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

- Original Equipment Manufacturers (OEMs)

This publication assumes that the audience is familiar with the Windows or Linux operating system and has experience using the C programming language.

How to Use This Publication

This guide is divided into the following chapters:

- Chapter 1, "IP Overview", gives an overview of VoIP technology and brief introductions to the H.323 and SIP standards for novice users.
- Chapter 2, "Dialogic® Global Call API Architecture for IP", describes how Dialogic® Global Call API can be used with IP technology and provides an overview of the architecture.
- Chapter 3, "IP Call Scenarios", provides some call scenarios that are specific to IP technology, including scenarios for the call transfer supplementary service.
- Chapter 4, "IP-Specific Operations", describes how to use Dialogic® Global Call API to perform IP-specific operations, such as setting call related information, registering with a registration server, sending and receiving protocol-specific messages, etc.
- Chapter 5, "Third Party Call Control (3PCC) Operations and Multimedia Support", describes the Dialogic® Global Call API library's support for scenarios where the SIP call control application is not a direct participant in calls.
- Chapter 6, "Building Dialogic® Global Call API IP Applications" provides information for building Dialogic® Global Call API applications that use IP technology.
- Chapter 7, "Debugging Dialogic® Global Call API IP Applications" provides information for debugging Dialogic® Global Call API IP applications using RTF logging facilities.
- Chapter 8, "IP-Specific Function Information", documents functions that are specific to the IP technology and describes additional functionality or limitations for specific Dialogic® Global Call API functions when used with IP technology.
- Chapter 9, "IP-Specific Parameters" provides a reference for IP-specific parameter set IDs and their associated parameter IDs.
- Chapter 10, "IP-Specific Data Structures", provides reference information for data structures that are specific to the use of Dialogic® Global Call API with the IP technology.
- Chapter 11, "IP-Specific Event Cause Codes" describes IP-specific event cause codes.
- Chapter 12, "Supplementary Reference Information" provides supplementary information including technology references and formats for called and calling party addresses for H.323.

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)

IP Overview 1

This chapter provides overview information about the following topics:

1.1 Introduction to VoIP

Voice over IP (VoIP) can be described as the ability to make telephone calls and send faxes over IP-based data networks with a suitable Quality of Service (QoS). The voice information is sent in digital form using discrete packets rather than via dedicated connections as in the circuit-switched Public Switched Telephone Network (PSTN).

Currently, there are two major international groups defining standards for VoIP:

- International Telecommunications Union, Telecommunications Standardization Sector (ITU-T), which has defined the following:
  – Recommendation H.323, covering Packet-based Multimedia Communications Systems (including VoIP)
- Internet Engineering Task Force (IETF), which has defined drafts of the several RFC (Request for Comment) documents, including the following central document:
  – RFC 3261, the Session Initiation Protocol (SIP)

The H.323 recommendation was developed in the mid 1990s and is a mature protocol.

SIP (Session Initiation Protocol) is an emerging protocol for setting up telephony, conferencing, multimedia, and other types of communication sessions on the Internet.

1.2 H.323 Overview

The H.323 specification is an umbrella specification for the implementation of packet-based multimedia over IP networks that cannot guarantee Quality of Service (QoS). This section discusses the following topics about H.323:

- H.323 Entities
- H.323 Protocol Stack
- Codecs
- Basic H.323 Call Scenario
- Registration with a Gatekeeper
- H.323 Call Scenario via a Gateway

1.2.1 H.323 Entities

The H.323 specification defines the entity types in an H.323 network including:

Terminal
: An endpoint on an IP network that supports the real-time, two-way communication with another H.323 entity. A terminal supports multimedia coders/decoders (codecs) and setup and control signaling.

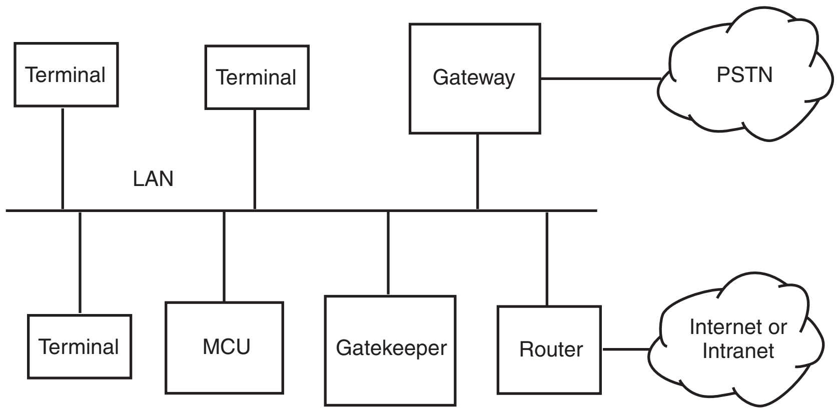
Gateway
: Provides the interface between a packet-based network (for example, an IP network) and a circuit-switched network (for example, the PSTN). A gateway translates communication procedures and formats between networks. It handles call setup and teardown and the compression and packetization of voice information.

Gatekeeper
: Manages a collection of H.323 entities in an H.323 zone controlling access to the network for H.323 terminals, Gateways, and MCUs and providing address translation. A zone can span a wide geographical area and include multiple networks connected by routers and switches. Typically there is only one gatekeeper per zone, but there may be an alternate gatekeeper for backup and load balancing. Typically, endpoints such as terminals, gateways, and other gatekeepers register with the gatekeeper.

Multipoint Control Unit (MCU)
: An endpoint that supports conferences between three or more endpoints. An MCU can be a stand-alone unit or integrated into a terminal, gateway, or gatekeeper. An MCU consists of:
  - Multipoint Controller (MC) – handles control and signaling for conferencing support
  - Multipoint Processor (MP) – receives streams from endpoints, processes them, and returns them to the endpoints in the conference

Figure 1 shows the entities in a typical H.323 network.

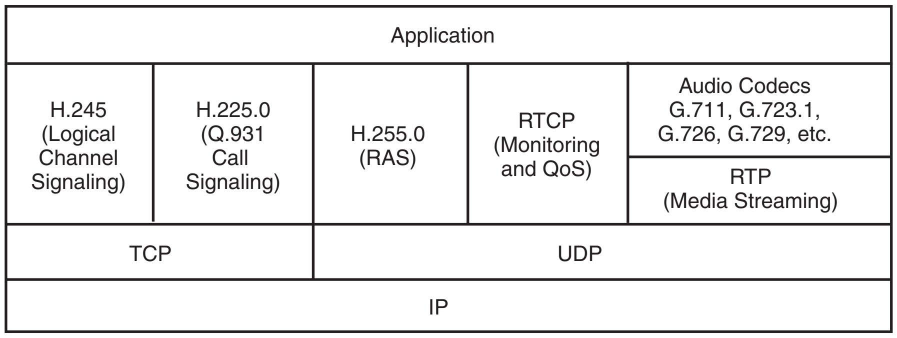
Figure 1. Typical H.323 Network



1.2.2 H.323 Protocol Stack

The H.323 specification is an umbrella specification for the many different protocols that comprise the overall H.323 protocol stack. Figure 2 shows the H.323 protocol stack.

Figure 2. H.323 Protocol Stack


Application
H.245 (Logical Channel Signaling)
H.225.0 (Q.931 Call Signaling)
H.255.0 (RAS)
RTCP (Monitoring and QoS)
Audio Codecs G.711, G.723.1, G.726, G.729, etc.
RTP (Media Streaming)
TCP
UDP
IP

The purpose of each protocol is summarized briefly as follows:

H.245
: Specifies messages for opening and closing channels for media streams, and other commands, requests, and indications.

Q.931
: Defines signaling for call setup and call teardown.

H.225.0
: Specifies messages for call control, including signaling, the packetization and synchronization of media streams, and Registration, Admission, and Status (RAS).

Real Time Protocol (RTP)
: The RTP specification is an IETF draft standard (RFC 1889) that defines the end-to-end transport of real-time data. RTP does not guarantee quality of service (QoS) on the transmission. However, it does provide some techniques to aid the transmission of isochronous data, including:

- information about the type of data being transmitted
- time stamps
- sequence numbers

Real Time Control Protocol (RTCP)
: RTCP is part of the IETF RTP specification (RFC 1889) and defines the end-to-end monitoring of data delivery and QoS by providing information such as:

- jitter, that is, the variance in the delays introduced in transmitting data over a wire
- average packet loss

The H.245, Q.931, and H.225.0 combination provide the signaling for the establishment of a connection, the negotiation of the media format that will be transmitted over the connection, and call teardown at termination. As indicated in Figure 2, the call signaling part of the H.323 protocol is carried over TCP, since TCP guarantees the in-order delivery of packets to the application.

The RTP and RTCP combination is for media handling only. As indicated in Figure 2, the media part of the H.323 protocol is carried over UDP and therefore there is no guarantee that all packets will arrive at the destination and be placed in the correct order.

1.2.3 Codecs

RTP and RTCP data is the payload of a User Datagram Protocol (UDP) packet. Analog signals coming from an endpoint are converted into the payload of UDP packets by codecs (coders/decoders). The codecs perform compression and decompression on the media streams.

Different types of codecs provide varying sound quality. The bit rate of most narrow-band codecs is in the range 1.2 kbps to 64 kbps. The higher the bit rate the better the sound quality. Some of the most popular codecs are:

G.711
: Provides a bit rate of 64 kbps.

G.723.1
: Provides bit rates of either 5.3 or 6.4 kbps. Voice communication using this codec typically exhibits some form of degradation.

G.729
: Provides a bit rate of 8 kbps. This codec is very popular for voice over frame relay and for V.70 voice and data modems.

GSM
: Provides a bit rate of 13 kbps. This codec is based on a telephony standard defined by the European Telecommunications Standards Institute (ETSI). The 13 kbps bit rate is achieved with little degradation of voice-grade audio.

1.2.4 Basic H.323 Call Scenario

A simple H.323 call scenario can be described in five phases:

- Call Setup
- Capability Exchange
- Call Initiation
- Data Exchange
- Call Termination

Calls between two endpoints can be either direct or routed via a gatekeeper. This scenario describes a direct connection where each endpoint is a point of entry and exit of a media flow. The scenario described in this section assumes a slow start connection procedure. See Section 4.2, “Fast and Slow Call Setup Modes”, on page 108 for more information on the difference between the slow start and fast start connection procedure.

The example in this section describes the procedure for placing a call between two endpoints, A and B, each with an IP address on the same subnet.

Call Setup

Establishing a call between two endpoints nominally requires two TCP connections between the endpoints:

- one TCP connection for the call setup (Q.931/H.225 messages)
- one TCP connection for capability exchange and call control (H.245 messages)

In practice, the H.245 channel may not be required thanks to two additional features of the H.323 protocol. H.323 version 2 defines a Fast Start mode that accomplishes the endpoint capability exchange through the use of Fast Start Elements (FSEs) which are "piggy-backed" on Q.931/H.225 call setup messages rather than waiting for an H.245 channel to be established. It is also possible to encapsulate H.245 media control messages within Q.931/H.225 signaling messages using a technique known as *H.245 tunneling*. If tunneling is enabled, one less TCP port is required for incoming connections.

The caller at endpoint A connects to the callee at endpoint B on a well-known port, typically port 1720, and sends the call Setup message as defined in the H.225.0 specification. The Setup message includes:

- message type; in this case, Setup
- bearer capability, which indicates the type of call; for example, audio only
- called party number and address
- calling party number and address
- Protocol Data Unit (PDU), which includes an identifier that indicates which version of H.225.0 should be used along with other information

When endpoint B receives the Setup message, it responds with one of the following messages:

- Release Complete
- Alerting
- Connect
- Call Proceeding

In this case, endpoint B responds with the Alerting message. Endpoint A must receive the Alerting message before its setup timer expires. After sending this message, the user at endpoint B must either accept or refuse the call with a predefined time period. When the user at endpoint B picks up the call, a Connect message is sent to endpoint A and the next phase of the call scenario, capability exchange, can begin.

Capability Exchange

Call control and capability exchange messages, as defined in the H.245 standard, are sent on a second TCP connection. Endpoint A opens this connection on a dynamically allocated port at the endpoint B after receiving the address in one of the following H.225.0 messages:

- Alerting
- Call Proceeding

- Connect

This connection remains active for the entire duration of the call. The control channel is unique for each call between endpoints so that several different media streams can be present.

An H.245 TerminalCapabilitySet message that includes information about the codecs supported by that endpoint is sent from one endpoint to the other. Both endpoints send this message and wait for a reply which can be one of the following messages:

- TerminalCapabilitySetAck - accept the remote endpoints capability
- TerminalCapabilitySetReject - reject the remote endpoints capability

The two endpoints continue to exchange these messages until a capability set that is supported by both endpoints is agreed. When this occurs, the next phase of the call scenario, call initiation, can begin.

Call Initiation

Once the capability setup is agreed, endpoint A and B must set up the voice channels over which the voice data (media stream) will be exchanged. The scenario described here assumes a slow start connection procedure. See Section 4.2, “Fast and Slow Call Setup Modes”, on page 108 for more information on the difference between the slow start and fast start connection procedure.

To open a logical channel at endpoint B, endpoint A sends an H.245 OpenLogicalChannel message to endpoint B. This message specifies the type of data being sent, for example, the codec that will be used. For voice data, the message also includes the port number that endpoint B should use to send RTCP receiver reports. When endpoint B is ready to receive data, it sends an OpenLogicalChannelAck message to endpoint A. This message contains the port number on which endpoint A is to send RTP data and the port number on which endpoint A should send RTCP data.

Endpoint B repeats the process above to indicate which port endpoint A will receive RTP data and send RTCP reports to. Once these ports have been identified, the next phase of the call scenario, data exchange, can begin.

Data Exchange

Endpoint A and endpoint B exchange information in RTP packets that carry the voice data. Periodically, during this exchange both sides send RTCP packets, which are used to monitor the quality of the data exchange. If endpoint A or endpoint B determines that the expected rate of exchange is being degraded due to line problems, H.323 provides capabilities to make adjustments. Once the data exchange has been completed, the next phase of the call scenario, call termination, can begin.

Call Termination

To terminate an H.323 call, one of the endpoints, for example, endpoint A, hangs up. Endpoint A must send an H.245 CloseLogicalChannel message for each channel it has opened with endpoint B. Accordingly, endpoint B must reply to each of those messages with a CloseLogicalChannelAck message. When all the logical channels are closed, endpoint A sends an H.245

EndSessionCommand, waits until it receives the same message from endpoint B, then closes the channel.

Either endpoint (but typically the endpoint that initiates the termination) then sends an H.225.0 ReleaseComplete message over the call signalling channel, which closes that channel and ends the call.

1.2.5 Registration with a Gatekeeper

In a H.323 network, a gatekeeper is an entity that can manage all endpoints that can send or receive calls. Each gatekeeper controls a specific zone and endpoints must register with the gatekeeper to become part of the gatekeeper's zone. The gatekeeper provides call control services to the endpoints in its zone. The primary functions of the gatekeeper are:

- address resolution by translating endpoint aliases to transport addresses
- admission control for authorizing network access
- bandwidth management
- network management (in routed mode)

Endpoints communicate with a gatekeeper using the Registration, Admission, and Status (RAS) protocol. A RAS channel is an unreliable channel that is used to carry RAS messages (as described in the H.255 standard). The RAS protocol covers the following:

- Gatekeeper Discovery
- Endpoint Registration
- Endpoint Deregistration
- Endpoint Location
- Admission, Bandwidth Change and Disengage

Note: The RAS protocol covers status request, resource availability, nonstandard registration messages, unknown message response and request in progress that are not described in any detail in this overview. See *ITU-T Recommendation H.225.0 (09/99)* for more information.

Gatekeeper Discovery

An endpoint uses a process called *gatekeeper discovery* to find a gatekeeper with which it can register. To start this process, the endpoint can multicast a GRQ (gatekeeper request) message to the well-known discovery multicast address for gatekeepers. One or more gatekeepers may respond with a GCF (gatekeeper confirm) message indicating that it can act as a gatekeeper for the endpoint. If a gatekeeper does not want to accept the endpoint, it returns GRJ (gatekeeper reject). If more than one gatekeeper responds with a GCF message, the endpoint can choose which gatekeeper it wants to register with. In order to provide redundancy, a gatekeeper may specify an alternate gatekeeper in the event of a failure in the primary gatekeeper. Provision for the alternate gatekeeper information is provided in the GCF and RCF messages.

Endpoint Registration

An endpoint uses a process called *registration* to join the zone associated with a gatekeeper. In the registration process, the endpoint informs the gatekeeper of its transport, alias addresses, and endpoint type. Endpoints register with the gatekeeper identified in the gatekeeper discovery process described above. Registration can occur before any calls are made or periodically as necessary. An endpoint sends an RRQ (registration request) message to perform registration and in return receives an RCF (registration confirmation) or RRJ (registration reject) message.

Endpoint Deregistration

An endpoint may send an URQ (unregister request) in order to cancel registration. This enables an endpoint to change the alias address associated with its transport address or vice versa. The gatekeeper responds with an UCF (unregister confirm) or URJ (unregister reject) message.

The gatekeeper may also cancel an endpoint's registration by sending a URQ (unregister request) to the endpoint. The endpoint should respond with an UCF (unregister confirm) message. The endpoint should then try to re-register with a gatekeeper, perhaps a new gatekeeper, prior to initiating any calls.

Endpoint Location

An endpoint that has an alias address for another endpoint and would like to determine its contact information may issue a LRQ (location request) message. The LRQ message may be sent to a specific gatekeeper or multicast to the well-known discovery multicast address for gatekeepers. The gatekeeper to which the endpoint to be located is registered will respond with an LCF (location confirm) message. A gatekeeper that is not familiar with the requested endpoint will respond with LRJ (location reject).

Admission, Bandwidth Change and Disengage

The endpoint and gatekeeper exchange messages to provide admission control and bandwidth management functions. The ARQ (admission request) message specifies the requested call bandwidth. The gatekeeper may reduce the requested call bandwidth in the ACF (admission confirm) message. The ARQ message is also used for billing purposes, for example, a gatekeeper may respond with an ACF message just in case the endpoint has an account so the call can be charged. An endpoint or the gatekeeper may attempt to modify the call bandwidth during a call using a BRQ (bandwidth change request) message. An endpoint will send a DRQ (disengage request) message to the gatekeeper at the end of a call.

1.2.6 H.323 Call Scenario via a Gateway

While the call scenario described in Section 1.2.4, "Basic H.323 Call Scenario", on page 24 is useful for explaining the fundamentals of an H.323 call, it is not a realistic call scenario. Most significantly, the IP addresses of both endpoints were defined to be known in the example, while most Internet Service Providers (ISPs) allocate IP addresses to subscribers dynamically. This section describes the fundamentals of a more realistic example that involves a gateway.

A gateway provides a bridge between different technologies; for example, an H.323 gateway (or IP gateway) provides a bridge between an IP network and the PSTN. Figure 3 shows a configuration that uses a gateway. User A is at a terminal, while user B is by a phone connected to the PSTN.

Figure 3. Basic H.323 Network with a Gateway

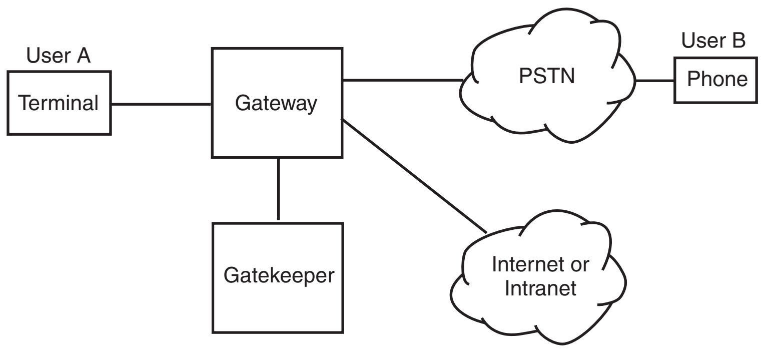


Figure 3 also shows a gatekeeper. The gatekeeper provides network services such as Registration, Admission, and Status (RAS) and address mapping. When a gatekeeper is present, all endpoints managed by the gatekeeper must register with the gatekeeper at startup. The gatekeeper tracks which endpoints are accepting calls. The gatekeeper can perform other functions also, such as redirecting calls. For example, if a user does not answer the phone, the gatekeeper may redirect the call to an answering machine.

The call scenario in this example involves the following phases:

- Establishing Contact with the Gatekeeper
- Requesting Permission to Call
- Call Signaling and Data Exchange
- Call Termination

Establishing Contact with the Gatekeeper

The user at endpoint A attempts to locate a gatekeeper by sending out a Gatekeeper Request (GRQ) message and waiting for a response. When it receives a Gatekeeper Confirm (GCF) message, the endpoint registers with the gatekeeper by sending the Registration Request (RRQ) message and waiting for a Registration Confirm (RCF) message. If more than one gatekeeper responds, endpoint A chooses only one of the responding gatekeepers. The next phase of the call scenario, requesting permission to call, can now begin.

Requesting Permission to Call

After registering with the gatekeeper, endpoint A must request permission from the gatekeeper to initiate the call. To do this, endpoint A sends an Admission Request (ARQ) message to the gatekeeper. This message includes information such as:

- a sequence number
- a gatekeeper assigned identifier

- the type of call; in this case, point-to-point
- the call model to use, either direct or gatekeeper-routed
- the destination address; in this case, the phone number of endpoint B
- an estimation of the amount of bandwidth required. This parameter can be adjusted later by a Bandwidth Request (BRQ) message to the gatekeeper.

If the gatekeeper allows the call to proceed, it sends an Admission Confirm (ACF) message to endpoint A. The ACF message includes the following information:

- the call model used
- the transport address and port to use for call signaling (in this example, the IP address of the gateway)
- the allowed bandwidth

All setup has now been completed and the next phase of the scenario, call signaling and data exchange, can begin.

Call Signaling and Data Exchange

Endpoint A can now send the Setup message to the gateway. Since the destination phone is connected to an analog line (the PSTN), the gateway goes off-hook and dials the phone number using dual tone multifrequency (DTMF) digits. The gateway therefore is converting the H.225.0 signaling into the signaling present on the PSTN. Depending on the location of the gateway, the number dialed may need to be converted. For example, if the gateway is located in Europe, then the international dial prefix will be removed.

As soon as the gateway is notified by the PSTN that the phone at endpoint B is ringing, it sends the H.225.0 Alerting message as a response to endpoint A. As soon as the phone is picked up at endpoint B, the H.225.0 Connect message is sent to endpoint A. As part of the Connect message, a transport address that allows endpoint A to negotiate codecs and media streams with endpoint B is sent.

The H.225.0 and H.245 signaling used to negotiate capability, initiate and call, and exchange data are the same as that described in the basic H.323 call scenario. See the Capability Exchange, Call Initiation, and Data Exchange phases in Section 1.2.4, “Basic H.323 Call Scenario”, on page 24 for more information.

In this example the destination phone is analog, therefore, it requires the gateway to detect the ring, busy, and connect conditions so it can respond appropriately.

Call Termination

As in the basic H.323 call scenario example, the endpoint that hangs up first needs to close all the channels that were open using the H.245 CloseLogicalChannel message. If the gateway terminates first, it sends an H.245 EndSessionCommand message to endpoint A and waits for the same message from endpoint A. The gateway then closes the H.245 channel.

When all channels between endpoint A and the gateway are closed, each must send a DisengageRequest (DRQ) message to the gatekeeper. This message lets the gatekeeper know that the bandwidth is being released. The gatekeeper sends a DisengageConfirm (DCF) message to both endpoint A and the gateway.

1.3 SIP Overview

Session Initiation Protocol (SIP) is an ASCII-based, peer-to-peer protocol designed to provide telephony services over the Internet. The SIP standard was developed by the Internet Engineering Task Force (IETF) and is one of the most commonly used protocols for VoIP implementations. This section discusses the following topics about SIP:

- Advantages of Using SIP
- SIP User Agents and Servers
- Basic SIP Operation
- Basic SIP Call Scenario
- SIP Messages

1.3.1 Advantages of Using SIP

Some of the advantages of using SIP include:

- The SIP protocol stack is smaller and simpler than other commonly used VoIP protocols, such as H.323.
- SIP-based systems are more easily scalable because of the peer-to-peer architecture used. The hardware and software requirements for adding new users to SIP-based systems are greatly reduced.
- Functionality is distributed over different components. Control is decentralized. Changes made to a component have less of an impact on the rest of the system.

1.3.2 SIP User Agents and Servers

User agents (UAs) are appliances or applications, such as SIP phones, residential gateways and software that initiate and receive calls over a SIP network.

Servers are application programs that accept requests, service requests and return responses to those requests. Examples of the different types of servers are:

Location Server
: Used by a SIP redirect or proxy server to obtain information about the location of the called party.

Proxy Server
: An intermediate program that operates as a server and a client and which makes requests on behalf of the client. A proxy server does not initiate new requests, it interprets and possibly modifies a request message before forwarding it to the destination.

Redirect Server
: Accepts a request from a client and maps the address to zero or more new addresses and returns the new addresses to the client. The server does not accept calls or generate SIP requests on behalf of clients.

Registrar Server
: Accepts REGISTER requests from clients. Often, the registrar server is located on the same physical server as the proxy server or redirect server.

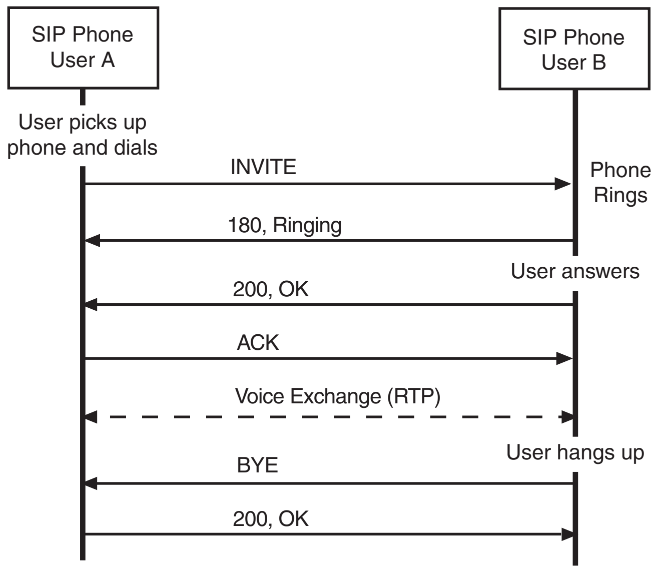
1.3.3 Basic SIP Operation

Callers and callees are identified by SIP addresses. When making a SIP call, a caller first locates the appropriate server and then sends a SIP request. The most common SIP operation is the invitation request. Instead of directly reaching the intended callee, a SIP request may be redirected or may trigger a chain of new SIP requests by proxies. Users can register their location(s) with SIP servers.

1.3.4 Basic SIP Call Scenario

Figure 4 shows the basic SIP call establishment and teardown scenario.

Figure 4. Basic SIP Call Scenario



1.3.5 SIP Messages

In SIP, there are two types of messages:

- SIP Request Messages
- SIP Response Messages

SIP Request Messages

The most commonly used SIP request messages are:

- INVITE
- ACK
- BYE
- REGISTER
- CANCEL
- OPTIONS

For more information on specific SIP request types, see RFC 3261 at *http://ietf.org/rfc/rfc3261.txt*.

SIP Response Messages

SIP response messages are numbered. The first digit in each response number indicates the type of response. The response types are as follows:

1xx	Information responses; for example, 180 Ringing
2xx	Successful responses; for example, 200 OK
3xx	Redirection responses; for example, 302 Moved Temporarily
4xx	Request failure responses; for example, 402 Forbidden
5xx	Server failure responses; for example, 504 Gateway Timeout
6xx	Global failure responses; for example, 600 Busy Everywhere

For more information on SIP response messages, see RFC 3261 at the URL given above.

Dialogic® Global Call API Architecture for IP 2

This chapter discusses the following topics:

2.1 Dialogic® Global Call API over IP Architecture with a Host-Based Stack

The Dialogic® Global Call API provides a common call control interface that is independent of the underlying network interface technology. While Dialogic® Global Call API is primarily concerned with call control, that is, call establishment and teardown, Dialogic® Global Call API provides some additional capabilities to support applications that use IP technology.

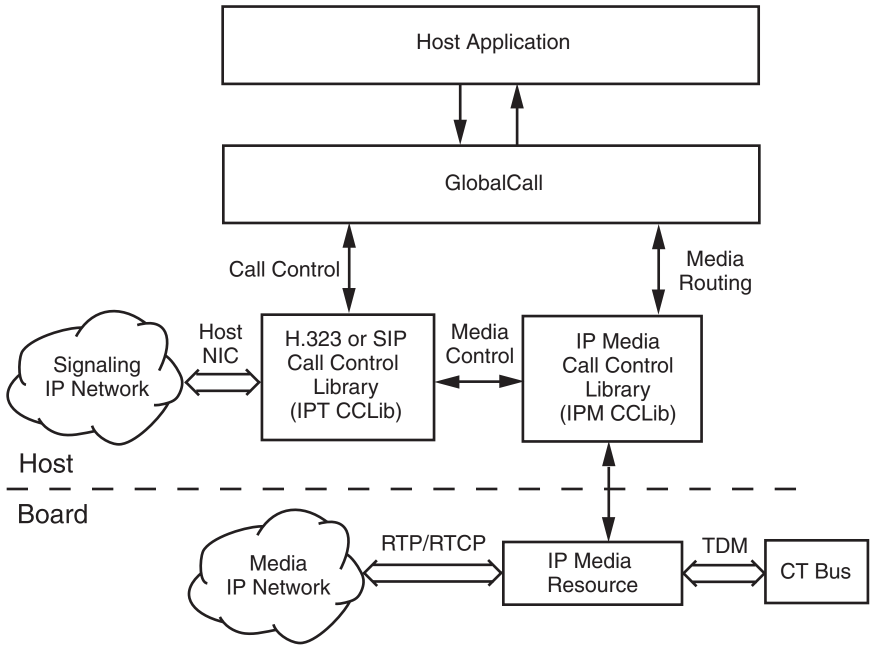
Dialogic® Global Call API support for IP technology includes:

- call control capabilities for establishing calls over an IP network
- support for IP media control by providing the ability to open and close IP media channels for streaming

Dialogic® Global Call API supports a system configuration where both the IP signaling stack and a Dialogic® Host Media Processing (HMP) virtual board, which provides the IP resources for media processing, are running on the host.

Figure 5 shows the Dialogic® Global Call API over IP architecture when using the virtual Dialogic® DM3 board implemented by Dialogic® HMP Software and the host-based stack provided with the system software.

Figure 5. Dialogic® Global Call API Over IP Architecture



To simplify IP Media management by the host application and to provide a consistent look and feel with other Dialogic® Global Call API technology call control libraries, the IP Signaling call control library (IPT CCLib) controls the IP media functionality on the application's behalf.

Note: Dialogic® Global Call API supports the RADVISION H.323 and SIP stacks. If other third-party call control stacks are used, Dialogic® Global Call API cannot be used for IP call control, but the Dialogic® IP Media Library API can be used directly by applications for media resource management. See the *Dialogic® IP Media Library API Programming Guide* and *Dialogic® IP Media Library API Library Reference* for more information.

2.2 Architecture Components

The role of each major component in the architecture is described in the following sections:

- Host Application
- Dialogic® Global Call API
- IP Signaling Call Control Library (IPT CCLib)
- IP Media Call Control Library (IPM CCLib)
- IP Media Resource

2.2.1 Host Application

The host application manages and monitors the IP telephony system operations. Typically the application performs the following tasks:

- initializes Dialogic® Global Call API
- opens and closes IP line devices (used to handle call control)
- opens and closes IP media devices (used to handle media streaming)
- opens and closes public switched telephone network (PSTN) devices
- configures IP media and network devices (capability list, operation mode, etc.)
- performs call control, including making calls, accepting calls, answering calls, dropping calls, releasing calls, and processing call state events
- queries call and device information
- handles PSTN alarms and errors

2.2.2 Dialogic® Global Call API

Dialogic® Global Call API hides technology and protocol-specific information from the host application and acts as an intermediary between the host application and the technology call control libraries. It performs the following tasks:

- performs high-level call control using the underlying call control libraries
- maintains a generic call control state machine based on the function calls used by an application and call control library events
- collects and maintains data relating to resources
- collects and maintains alarm data

2.2.3 IP Signaling Call Control Library (IPT CCLib)

The IP Signaling call control library (IPT CCLib) implements relevant Global Call call control functionality in an IP-specific way. It performs the following tasks:

- controls the H.323 and SIP call control stacks
- manages IP media resources as required by the Dialogic® Global Call API call state model and the IP signaling protocol model
- translates between the Dialogic® Global Call API call model and IP signaling protocol models
- processes Dialogic® Global Call API call control library interface commands
- generates call control library interface events

2.2.4 IP Media Call Control Library (IPM CCLib)

The IP Media Call Control Library (IPM CCLib) performs the following tasks:

- processes Dialogic® Global Call API CCLib commands for the opening, closing, and timeslot routing of IP media devices
- configures QoS (Quality of Service) thresholds
- translates QoS alarm events to Dialogic® Global Call API alarm (GCAMS) events

2.2.5 IP Media Resource

The IP Media Resource processes the IP Media stream. It performs the following tasks:

- encodes PCM data from the TDM bus into IP packets sent to the IP network
- decodes IP packets received from the IP network into PCM data transmitted to the TDM bus
- configures and reports QoS information to the IP Media stream

2.3 Device Types and Usage

This section includes information about device types and usage:

- Device Types Used with IP
- IPT Board Devices
- IPT Network Devices
- IPT Start Parameters

2.3.1 Device Types Used with IP

When using Dialogic® Global Call API with IP technology, a number of different device types are used:

IPT Board Device
: A virtual entity that represents a NIC or NIC address (if one NIC supports more than one IP address). The format of the device name is **iptBx**, where **x** is the logical board number that corresponds to the NIC or NIC address. See Section 2.3.2, “IPT Board Devices”, on page 39 for more information.

IPT Network Device
: Represents a logical channel over which calls can be made. This device is used for call control (call setup and tear down). The format of the device name is **iptBxTy**, where **x** is the logical board number and **y** is the logical channel number. See Section 2.3.3, “IPT Network Devices”, on page 40 for more information.

IP Media Device
: Represents a media resource that is used to control RTP streaming, monitoring Quality of Service (QoS) and the sending and receiving of DTMF digits. The format of the device name is **ipmBxCy**, where **x** is the logical board number and **y** is the logical channel number.

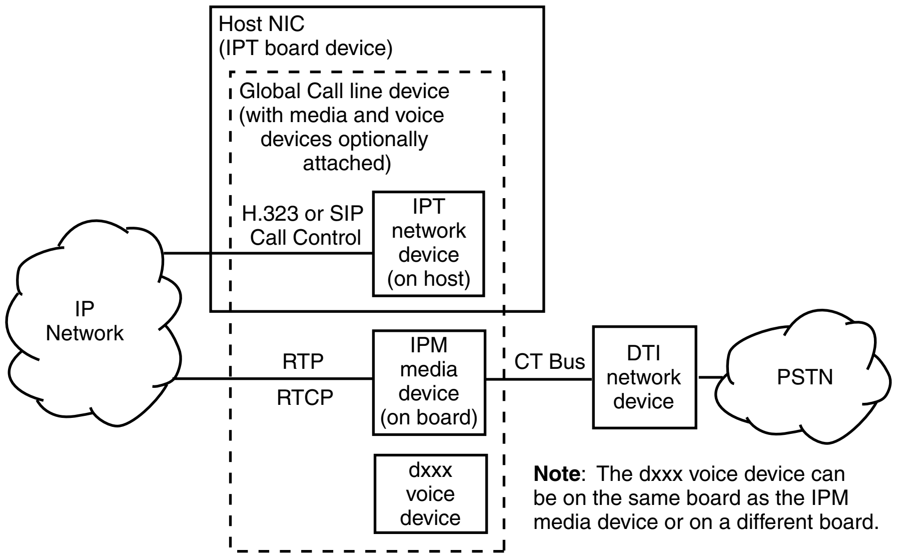
The IPT network device (iptBxTy) and the IP Media device (ipmBxCy) can be opened simultaneously in the same **gc_OpenEx( )** command. If a voice resource is available in the system, for example an IP board that provides voice resources or any other type of board that provides voice resources, a voice device can also be included in the same **gc_OpenEx( )** call to provide voice capabilities on the logical channel. See Section 8.3.18, "gc_OpenEx( ) Variances for IP", on page 575 for more information.

Alternatively, the IPT network device (iptBxTy) and the IP Media device (ipmBxCy) can be opened in separate **gc_OpenEx( )** calls and subsequently attached using the **gc_AttachResource( )** function.

The IP Media device handle, which is required for managing Quality of Service (QoS) alarms for example, can be retrieved using the **gc_GetResourceH( )** function. See Section 4.26, "Managing Quality of Service Alarms", on page 309 for more information.

Figure 6 shows the relationship between the various types of Dialogic® Global Call API devices when a single Host NIC is used.

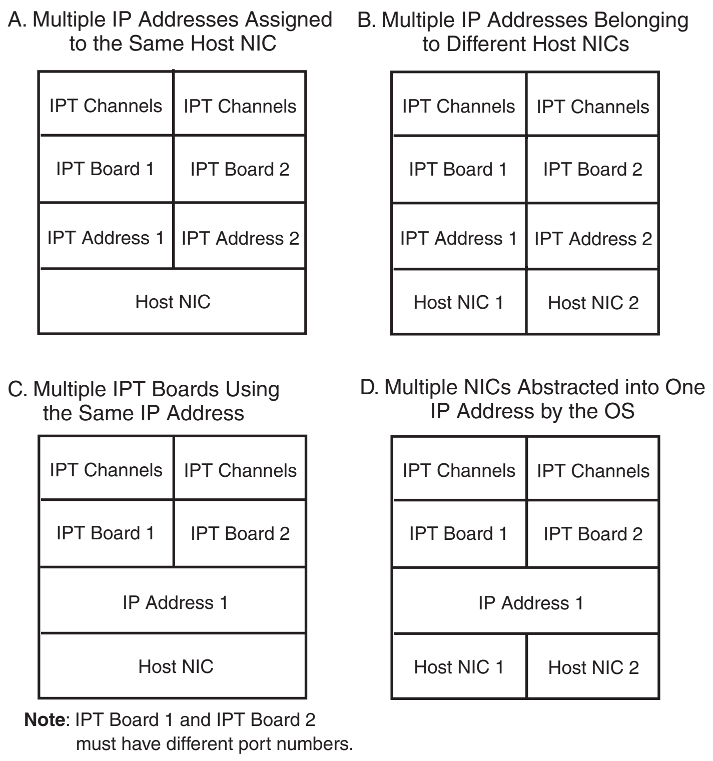
Figure 6. Dialogic® Global Call API Devices


Host NIC
(IPT board device)
Global Call line device (with media and voice devices optionally attached)
H.323 or SIP Call Control
IPT network device (on host)
IP Network
RTP
RTCP
IPM media device (on board)
CT Bus
DTI network device
PSTN
dxxx voice device
Note: The dxxx voice device can be on the same board as the IPM media device or on a different board.

2.3.2 IPT Board Devices

An IPT board device is a virtual entity that corresponds to an IP address and is capable of handling both H.323 and SIP protocols. The application uses the **gc_Start( )** function to bind IP addresses to IPT virtual board devices. Possible configurations are shown in Figure 7. The operating system must support the IP address and underlying layers before the Dialogic® Global Call API application can take advantage of the configurations shown in Figure 7. Up to eight virtual IPT boards can be configured in one system. For each virtual IPT board, it is possible to configure the local address and signaling port (H.323 and SIP), the number of IPT network devices that can be opened simultaneously, etc. See Section 8.3.27, "gc_Start( ) Variances for IP", on page 590 for more information on how to configure IPT board devices.

Figure 7. Configurations for Binding IPT Boards to NIC IP Addresses



Once the IPT board devices are configured, the application can open line devices with the appropriate IPT network device (IPT channel) and optionally IP Media device (IPM channel).

The **gc_SetConfigData( )** function can be used on an IPT board device to apply parameters to all IPT channels associated with the IPT board device. The application can use the **gc_AttachResource( )** and **gc_Detach( )** functions to load balance which host NIC makes a call for a particular IP Media device (IPM channel). It is also possible that the operating system can perform load balancing using the appropriate NIC for call control as shown in Figure 7, configuration D.

The **gc_ReqService( )** function is used on an IPT board device for registration with an H.323 gatekeeper or SIP registrar. See Section 8.3.22, "gc_ReqService( ) Variances for IP", on page 578 for more information.

2.3.3 IPT Network Devices

Dialogic® Global Call API supports three types of IPT network devices:

- H.323 only (P_H323 in the **devicename** string when opening the device)

- SIP only (P_SIP in the **devicename** string when opening the device)
- Dual protocol, H.323 and SIP (P_IP in the **devicename** string when opening the device)

The device type is determined when using the **gc_OpenEx( )** function to open the device. H.323 and SIP only devices are capable of initiating and receiving calls of the selected protocol type only.

Dual protocol devices are capable of initiating and receiving calls using either the H.323 or SIP protocol. The protocol used by a call on a dual protocol device is determined during call setup as follows:

- for outbound calls, by a parameter to the **gc_MakeCall( )** function
- for inbound calls, by calling **gc_GetCallInfo( )** to retrieve the protocol type used. In this case, the application can query the protocol type of the current call after the call is established, that is, as soon as either GCEV_DETECTED (if enabled) or GCEV_OFFERED is received.

2.3.4 IPT Start Parameters

The application determines the number of virtual boards that will be created by the IPT call control library (up to the number of available IP addresses). For each virtual board, the host application will provide the following information:

- number of line devices on the board
- maximum number of IPT devices to be used for H.323 calls (used for H.323 stack allocation)
- maximum number of IPT devices to be used for SIP calls (used for SIP stack allocation)
- board IP address
- signaling port for H.323
- signaling port for SIP
- enable/disable access to SIP message information fields (headers)
- enable/disable MIME-encoded content in SIP messages
- number and size of buffers in MIME memory pool (if MIME feature is enabled)
- enable/disable access to H.323 message information fields
- enable/disable call transfer supplementary service
- set terminal type for H.323
- enable and configure outbound proxy for SIP
- configure SIP transport protocol (enable use of TCP)
- configure SIP request retry behavior
- enable/disable application access to SIP OPTIONS messages
- configure maximum number of SIP registrations

IP Call Scenarios 3

This chapter provides common call control scenarios when using Dialogic® Global Call API with IP technology. Topics include:

3.1 Basic Call Control Scenarios When Using IP Technology

This section provides details of the basic call control scenarios when using IP technology. The scenarios include:

- Basic Call Setup When Using H.323 or SIP
- Basic Call Teardown When Using H.323 or SIP
- Call Setup Scenarios for Early Media

3.1.1 Basic Call Setup When Using H.323 or SIP

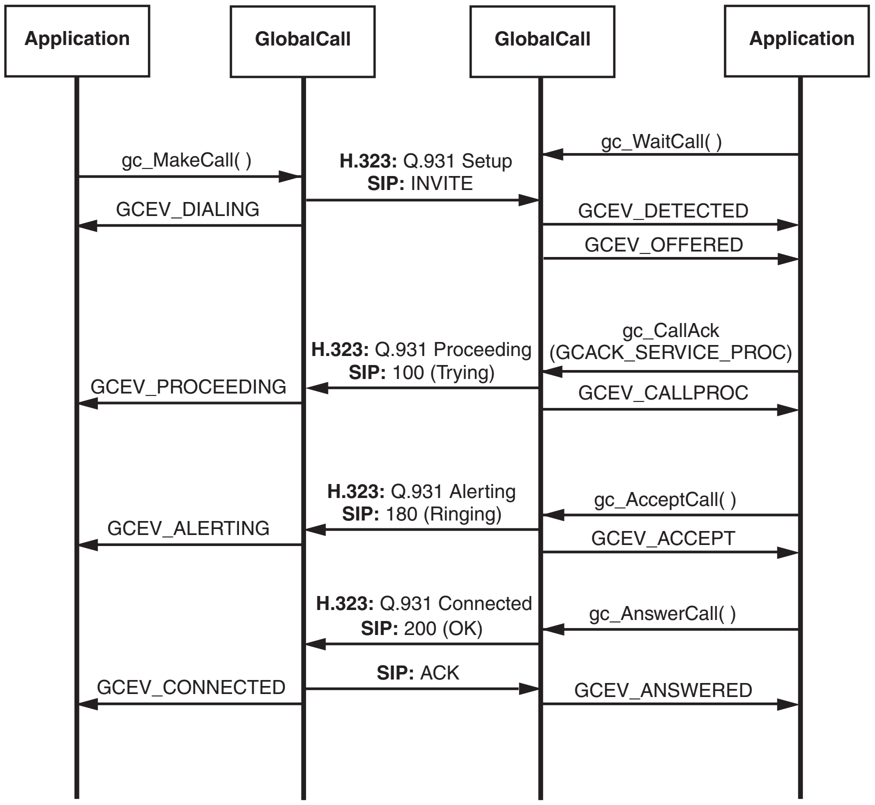
Figure 8 shows the basic call setup sequence when using H.323 or SIP.

Notes: ***1.*** This figure assumes that the network and media channels are already open and a media channel with the appropriate media capabilities is attached to the network channel. See Section 8.3.18, "gc_OpenEx( ) Variances for IP", on page 575 for information on opening and attaching network and media devices and Section 8.3.17, "gc_MakeCall( ) Variances for IP", on page 560 for detailed information on the specification of the destination address etc.

2. Only H.225.0 (Q.931) messages are shown in the sequence below. H.245 messages were omitted in the interest of simplification.

3. The destination address must be a valid address that can be translated by the remote node.

Figure 8. Basic Call Setup When Using H.323 or SIP

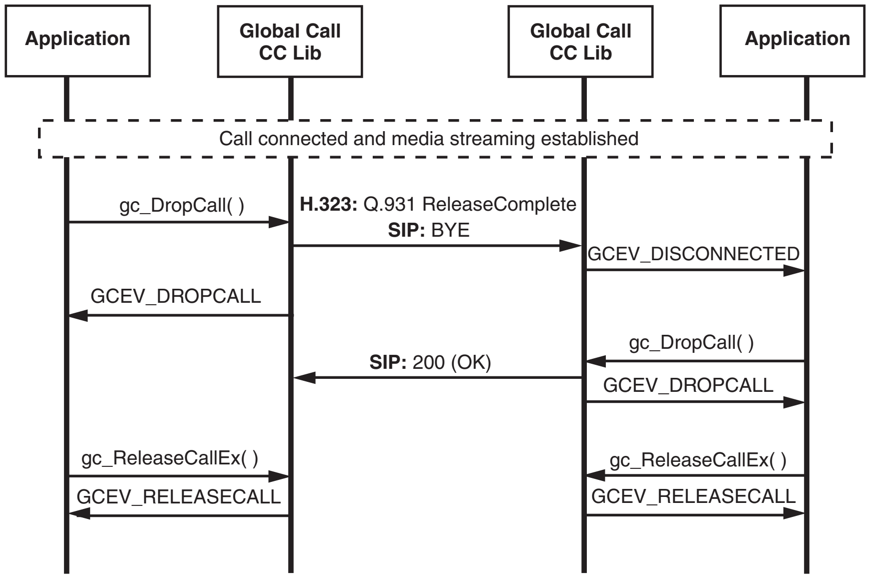


3.1.2 Basic Call Teardown When Using H.323 or SIP

Figure 9 shows the basic call teardown scenario when using Dialogic® Global Call API with H.323 or SIP.

Note: Only H.225.0 (Q.931) messages are shown in the sequence below. H.245 messages were omitted in the interest of simplification.

Figure 9. Basic Call Teardown When Using H.323 or SIP



3.1.3 Call Setup Scenarios for Early Media

When using IP technology, the establishment of RTP media streaming is normally one of the final steps in establishing and connecting a call. This is in contrast to the public switched telephone network (PSTN), where call progress signaling is commonly provided to the calling party via audible, in-band call progress tones, such as ringback, busy signal, and SIT tones. When implementing a VoIP gateway, it is often imperative to initiate media (RTP) streaming from the local endpoint to the calling party before the call is connected. This capability is commonly referred to as *early media*.

The Global Call IP call control library automatically enables media streaming at the earliest possible point in the pre-connect process. This is generally the earliest point at which the remote endpoint provides the remote RTP/RTCP transport addresses and media capabilities. The precise point at which media can be enabled is dependent on a large number of factors, and the following figures illustrate some common best-case scenarios. Each figure illustrates the Dialogic® Global Call API library's behavior from the application's perspective, either in the calling party role or in the called party role.

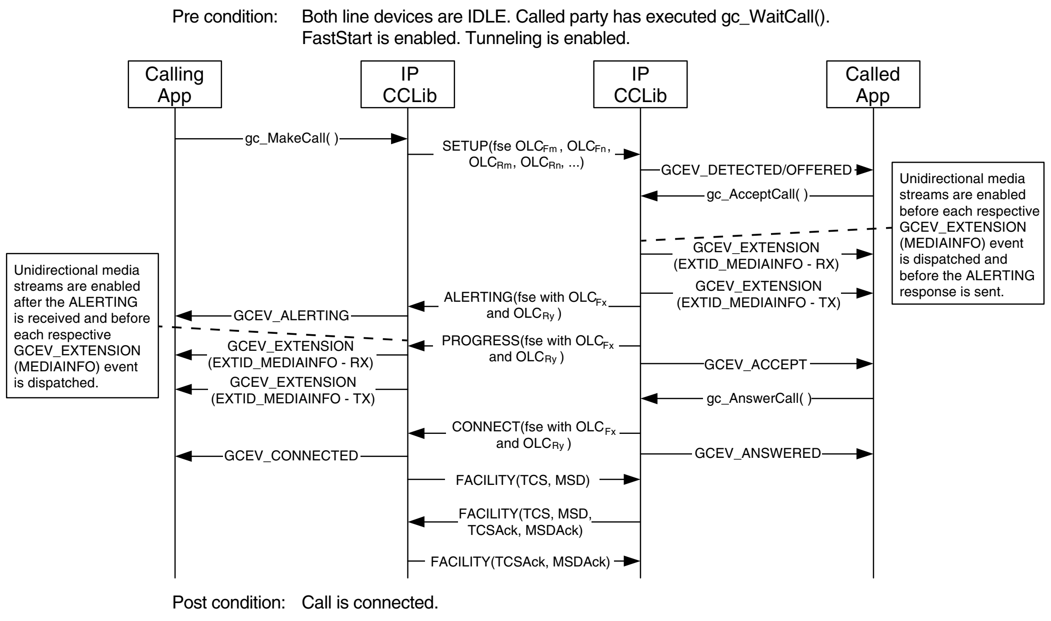
Note that in some cases it is possible to enable streaming in one direction significantly earlier than in the other direction. To take full advantage of this fact, the Global Call IP call control library initially enables a temporary unidirectional connection, then modifies the connection to be full duplex as soon as that is possible.

3.1.3.1 H.323 FastStart Mode

The library's default for H.323 operation enables the Global Call FastStart mode, in which the channel capability information is embedded in a fastStart element (indicated in the figure as "FSE") that can be sent within the messages of the H.225 Setup exchange rather than using the H.245 messages. (This minimizes the number of round-trip message exchanges and avoids the latency of H.245 channel establishment.) As a calling endpoint, the Dialogic® Global Call API library enables media after Alerting is received if the called endpoint supports the fastStart mode. As a called endpoint, the Global Call library enables media in a fastStart connection after the application calls **gc_AcceptCall( )**.

If the calling endpoint sets the MediaWaitForConnect element in the Setup message, the Dialogic® Global Call API library does not enable media transmission for a called endpoint until the Connect message is sent.

Figure 10. H.323 Early Media, FastStart Mode



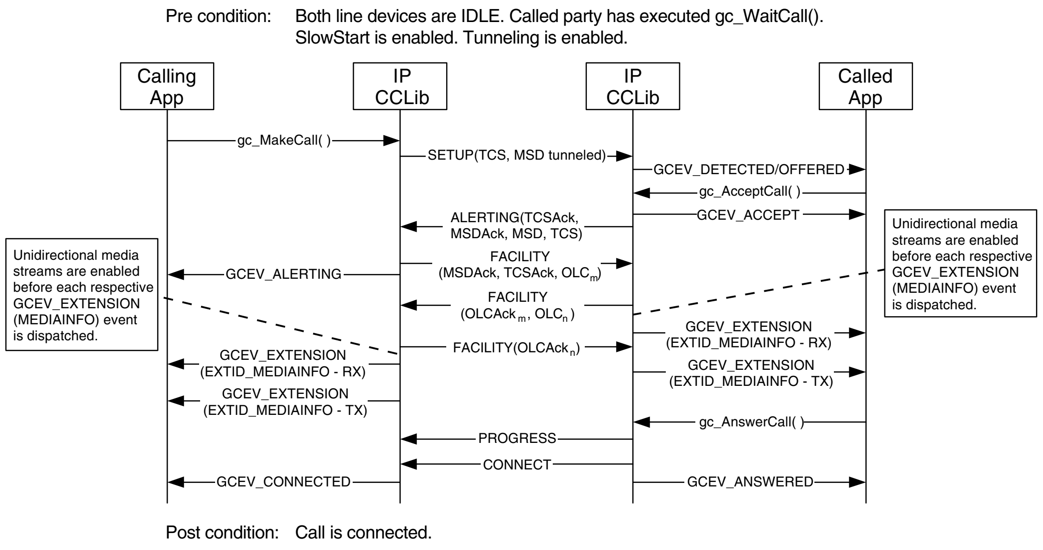
Pre condition: Both line devices are IDLE. Called party has executed gc_WaitCall(). FastStart is enabled. Tunneling is enabled.
Calling App
IP CCLib
IP CCLib
Called App
gc_MakeCall( )
SETUP(fse OLCFm, OLCFn, OLCRm, OLCRn, ...)
GCEV_DETECTED/OFFERED
gc_AcceptCall( )
Unidirectional media streams are enabled before each respective GCEV_EXTENSION (MEDIAINFO) event is dispatched and before the ALERTING response is sent.
GCEV_EXTENSION (EXTID_MEDIAINFO - RX)
GCEV_EXTENSION (EXTID_MEDIAINFO - TX)
Unidirectional media streams are enabled after the ALERTING is received and before each respective GCEV_EXTENSION (MEDIAINFO) event is dispatched.
ALERTING(fse with OLCFx and OLCRy)
GCEV_ALERTING
PROGRESS(fse with OLCFx and OLCRy)
GCEV_EXTENSION (EXTID_MEDIAINFO - RX)
GCEV_EXTENSION (EXTID_MEDIAINFO - TX)
GCEV_ACCEPT
gc_AnswerCall( )
CONNECT(fse with OLCFx and OLCRy)
GCEV_CONNECTED
GCEV_ANSWERED
FACILITY(TCS, MSD)
FACILITY(TCS, MSD, TCSAck, MSDAck)
FACILITY(TCSAck, MSDAck)
Post condition: Call is connected.

3.1.3.2 H.323 SlowStart Mode

Many factors affect the opportunities for early media in H.323 SlowStart mode.

- Unless both endpoints support what is referred to as “early H.245”, early media is not possible in the H.323 SlowStart connection mode.
- When a Global Call application specifies the optional SlowStart mode, or when one endpoint does not support H.323 fastStart mode, media transmission cannot begin at either endpoint until the remote endpoint has sent its Ack to the appropriate OpenLogicalChannel command.
- If the OLCAck that either endpoint receives contains a FlowControlToZero flag parameter that is true, media transmission from that endpoint is not be enabled until a subsequent FlowControl message is received.
- If the calling endpoint sets the MediaWaitForConnect element in the Setup message, the called endpoint does not enable media transmission until the Connect message is sent.

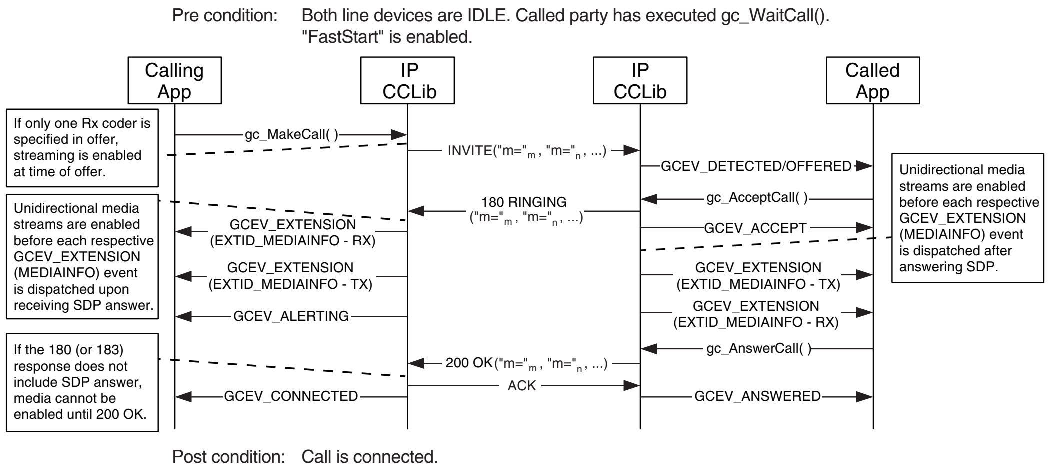
Figure 11. H.323 Early Media, SlowStart Mode with Early H.245 Enabled


Pre condition: Both line devices are IDLE. Called party has executed gc_WaitCall(). SlowStart is enabled. Tunneling is enabled.
Calling App
IP CCLib
IP CCLib
Called App
gc_MakeCall( )
SETUP(TCS, MSD tunneled)
GCEV_DETECTED/OFFERED
gc_AcceptCall( )
ALERTING(TCSAck, MSDAck, MSD, TCS)
GCEV_ACCEPT
Unidirectional media streams are enabled before each respective GCEV_EXTENSION (MEDIAINFO) event is dispatched.
GCEV_ALERTING
FACILITY (MSDAck, TCSAck, OLCm)
FACILITY (OLCAckm, OLCn)
Unidirectional media streams are enabled before each respective GCEV_EXTENSION (MEDIAINFO) event is dispatched.
GCEV_EXTENSION (EXTID_MEDIAINFO - RX)
FACILITY(OLCAckn)
GCEV_EXTENSION (EXTID_MEDIAINFO - RX)
GCEV_EXTENSION (EXTID_MEDIAINFO - TX)
GCEV_EXTENSION (EXTID_MEDIAINFO - TX)
gc_AnswerCall( )
PROGRESS
CONNECT
GCEV_CONNECTED
GCEV_ANSWERED
Post condition: Call is connected.

3.1.3.3 SIP FastStart Mode (Calling UA Offers SDP)

The SIP protocol does not define distinct “fast start” and “slow start” modes as does H.323, but the Dialogic® Global Call API library uses the same FastStart/SlowStart parameter interface to allow applications to specify whether the calling UA offers SDP in its INVITE message or whether it allows the called UA to offer SDP, which SIP refers to as “delayed offer”. In the default “FastStart” mode, the calling endpoint offers SDP and the called UA answers.

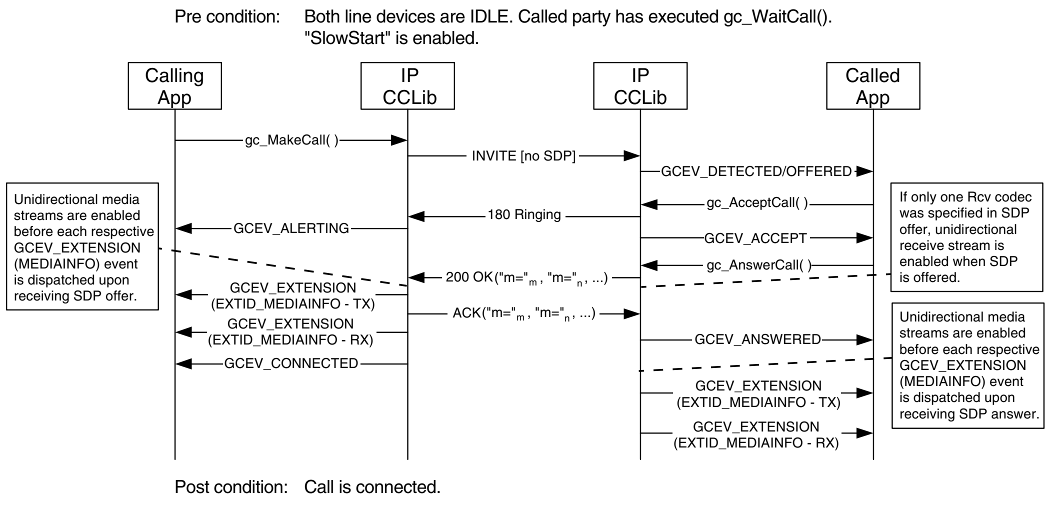
Figure 12. SIP Early Media, Calling UA Offers SDP


Pre condition: Both line devices are IDLE. Called party has executed gc_WaitCall(). "FastStart" is enabled.
Calling App
IP CCLib
IP CCLib
Called App
If only one Rx coder is specified in offer, streaming is enabled at time of offer.
gc_MakeCall( )
INVITE("m="m, "m="n, ...)
GCEV_DETECTED/OFFERED
Unidirectional media streams are enabled before each respective GCEV_EXTENSION (MEDIAINFO) event is dispatched after answering SDP.
gc_AcceptCall( )
180 RINGING ("m="m, "m="n, ...)
Unidirectional media streams are enabled before each respective GCEV_EXTENSION (MEDIAINFO) event is dispatched upon receiving SDP answer.
GCEV_EXTENSION (EXTID_MEDIAINFO - RX)
GCEV_ACCEPT
GCEV_EXTENSION (EXTID_MEDIAINFO - TX)
GCEV_EXTENSION (EXTID_MEDIAINFO - TX)
GCEV_EXTENSION (EXTID_MEDIAINFO - RX)
GCEV_ALERTING
gc_AnswerCall( )
If the 180 (or 183) response does not include SDP answer, media cannot be enabled until 200 OK.
200 OK("m="m, "m="n, ...)
ACK
GCEV_CONNECTED
GCEV_ANSWERED
Post condition: Call is connected.

3.1.3.4 SIP SlowStart Mode (Calling UA Answers SDP)

When a SIP application sets the optional SlowStart parameter, it specifies that the INVITE message it sends will not contain SDP, so that it is up to the called UA to offer SDP which the calling UA will subsequently answer. In SIP terminology, this is known as *delayed offer.*

Figure 13. SIP Early Media, Calling UA Answers SDP


Pre condition: Both line devices are IDLE. Called party has executed gc_WaitCall(). "SlowStart" is enabled.
Calling App
IP CCLib
IP CCLib
Called App
gc_MakeCall( )
INVITE [no SDP]
GCEV_DETECTED/OFFERED
Unidirectional media streams are enabled before each respective GCEV_EXTENSION (MEDIAINFO) event is dispatched upon receiving SDP offer.
gc_AcceptCall( )
If only one Rcv codec was specified in SDP offer, unidirectional receive stream is enabled when SDP is offered.
180 Ringing
GCEV_ALERTING
GCEV_ACCEPT
gc_AnswerCall( )
200 OK("m="m, "m="n, ...)
GCEV_EXTENSION (EXTID_MEDIAINFO - TX)
ACK("m="m, "m="n, ...)
GCEV_EXTENSION (EXTID_MEDIAINFO - RX)
GCEV_ANSWERED
Unidirectional media streams are enabled before each respective GCEV_EXTENSION (MEDIAINFO) event is dispatched upon receiving SDP answer.
GCEV_CONNECTED
GCEV_EXTENSION (EXTID_MEDIAINFO - TX)
GCEV_EXTENSION (EXTID_MEDIAINFO - RX)
Post condition: Call is connected.

3.2 Call Transfer Scenarios When Using H.323

The Dialogic® Global Call API API functions that support IP call transfer are described in the *Dialogic® Global Call API Library Reference*. Information on implementing H.450.2 call transfer can be found in Section 4.38, "T.38 Fax Server", on page 374, and protocol-specific information about the individual call transfer APIs can be found in the subsections of Section 8.3, "Dialogic® Global Call API Function Variances for IP", on page 542.

The following topics describe the call transfer capabilities provided when using the H.450.2 supplementary service with H.323:

- General Conditions for H.450.2 Call Transfers
- Endpoint Behavior in H.450.2 Blind Call Transfers
- Successful H.450.2 Blind Call Transfer Scenario
- Unsuccessful H.450.2 Blind Call Transfer Scenarios
- Endpoint Behavior in H.450.2 Supervised Call Transfer
- Successful H.450.2 Supervised Call Transfer Scenario
- Unsuccessful H.450.2 Supervised Transfer Scenarios

3.2.1 General Conditions for H.450.2 Call Transfers

When performing a call transfer operation, all involved call handles must be on the same stack instance. This imposes the following application restrictions for call transfer operations:

- When performing a supervised call transfer at party A, both the consultation line device and the transferring line device must be on the same virtual board.
- When performing a call transfer (either supervised or blind) at party B, both the transferring line device and the transferred line device must be on the same virtual board.
- When performing a supervised call transfer at party C, both the consultation line device and the transferred-to line device must be on the same virtual board.

3.2.2 Endpoint Behavior in H.450.2 Blind Call Transfers

This section describes the behavior of each of the three endpoints in an H.450.2 blind call transfer. The assumed precondition for supervised call transfer is:

- The transferring endpoint (party A) and the transferred endpoint (party B) are participating in an active call. From the perspective of the Dialogic® Global Call API, party A and party B are both in the GCST_CONNECTED state.

3.2.2.1 Transferring Endpoint (Party A) Role

The transferring endpoint (party A) initiates the blind transfer by calling the **gc_InvokeXfer( )** function, which results in the sending a ctInitiate.Invoke APDU (Application Protocol Data Unit, the type of message used for H.450 supplementary services) within a Facility message. From this point forward, this endpoint is only subsequently notified as to the creation of the transferred call attempt. Note however, that it is not notified as to the end result of the transfer, specifically whether the transfer results in a connection or a no-answer. Instead, the transferring endpoint is only guaranteed notification that the transferred-to endpoint has been alerted to the incoming transferred call offering (that is, ringback). As specified in H.450.2, the ctInitiate.ReturnResult APDU may be returned in either Alerting or Connect. The primary call will also be disconnected remotely via the transferred endpoint (party B) as part of a successful status notification from this endpoint. Both the forward and reverse logical channels will be closed along with their associated audio or data streams. From the Dialogic® Global Call API perspective, the primary call is terminated at the transferring endpoint, as indicated by the GCEV_DISCONNECTED event, implying the endpoint is then responsible for the drop and release of the primary call.

3.2.2.2 Transferred Endpoint (Party B) Role

The endpoint to be transferred (party B) is notified of the request to transfer from the initiating endpoint via the GCEV_REQ_XFER event. Assuming the party to be transferred accepts the transfer request via the **gc_AcceptXfer( )** function, it retrieves the destination address information from the unsolicited transfer request via the GC_REROUTING_INFO structure passed within the GCEV_REQ_XFER event. The endpoint to be transferred then uses the rerouting address information to initiate a call to the new destination party via **gc_MakeCall( )**. From the perspective of the application, this transferred call is treated in the same manner as a normal singular call and the party receives intermediate call state events as to the progress of the call (that is, GCEV_DIALING, GCEV_ALERTING, GCEV_PROCEEDING, and GCEV_CONNECTED). When the transferred endpoint receives its first indication from the transferred-to endpoint (party C) that the call transfer was successful (ctSetup.ReturnResult APDU), the transferred endpoint is notified of the transfer success and implicitly, without user or application initiation, disconnects the primary call with the transferring endpoint.

Assuming the transferred call is answered, the transferred endpoint is then involved in active media streaming with the transferred-to endpoint. Note that the notification of transfer success via the GCEV_XFER_CMPLT event may also arrive with any call progress events, that is, GCEV_ALERTING, GCEV_PROCEEDING, or GCEV_CONNECTED. Although the primary call to the transferring endpoint (party A) is implicitly dropped, the call itself must still be explicitly dropped via **gc_DropCall( )** to resynchronize the local state machine and released via **gc_ReleaseCallEx( )**.

3.2.2.3 Transferred-To Endpoint (Party C) Role

For the most part, from the perspective of the transferred-to endpoint (party C), the transferred call is treated as a typical incoming call. The call is first notified to the application via GCEV_DETECTED or GCEV_OFFERED events at which point the GCRV_XFERCALL cause value provided in the event will alert the application that this call offering is the result of a transfer. At that point, the application may retrieve the typical calling party information about the call. The transferred-to party is then provided the same methods of action as a typical incoming call, namely alerting the transferred endpoint (party B) that the call is proceeding (typical for gateways), ringback notification that the local user is being alerted, or simply answering the call. The only behavior change from this endpoint over typical non-transferred calls, is whether to treat or display the calling party information any differently if it is the result of a transfer. Assuming the transferred call is eventually connected or timed out on no answer, the transferred-to party must eventually drop and release this call as the case for non-transferred call.

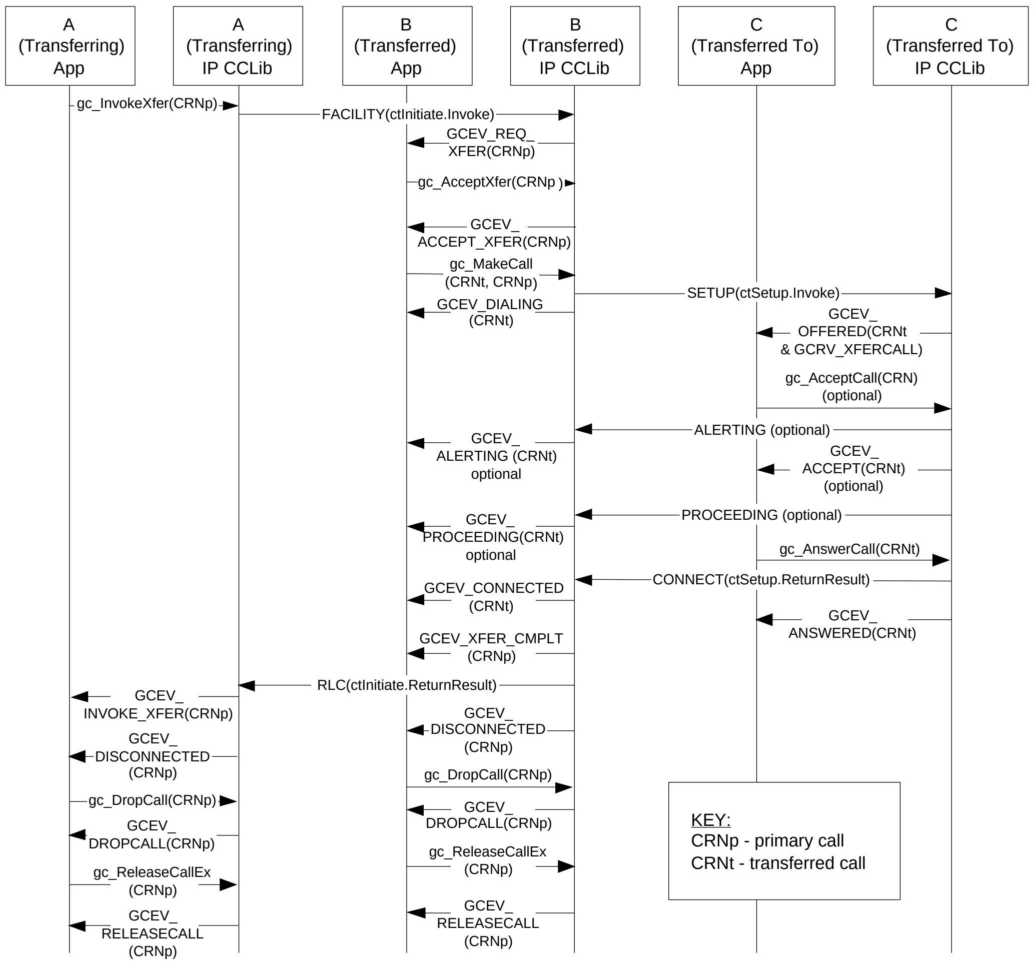
3.2.3 Successful H.450.2 Blind Call Transfer Scenario

As indicated in Figure 14, the precondition for blind transfer is that the transferring endpoint (party A) and the transferred endpoint (party B) are participating in an active (primary) call and are in GCST_CONNECTED from the perspective of the Dialogic® Global Call API. Completion of a successful blind transfer results in the eventual termination of the primary call, and the creation of the transferred call. Note that the connection of the transferred call is not a mandate for the completion of a blind transfer. It is always possible that the transferred call itself may possibly be left unanswered after ringing (Alerting indication) and eventually abandoned and still be considered a *successful* blind transfer from the perspective of the transferring endpoint (party A). Successful blind transfer, in this regard requires only that some response notification (that is, either Alerting or Connect) was received from the transferred-to endpoint.

For simplification purposes, Figure 14 does not indicate the opening and closing of logical channels (and the associated media sessions) because the control procedures are consistent with typical non-transfer related H.323 calls.

Figure 14. Successful H.450.2 Blind Call Transfer

Pre condition: Primary call between A and B is connected (not shown).



Post condition: Transferred call between B and C offered.
Primary call between A and B dropped and released.

3.2.4 Unsuccessful H.450.2 Blind Call Transfer Scenarios

There are a several of scenarios where a blind call transfer may fail. The most common scenarios are described in the following topics:

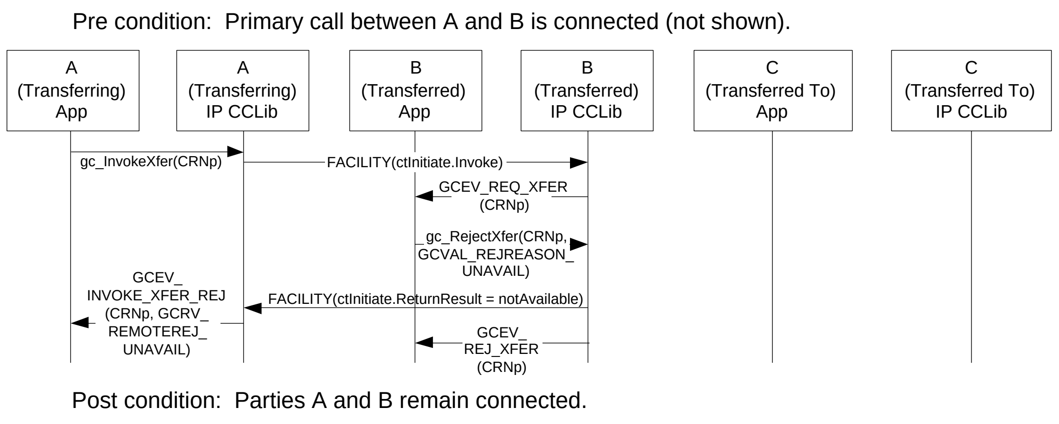
- Party B Rejects Transfer
- No Response From Party B
- No Response From Party C
- Party B Clears Primary Call Before Transfer is Completed
- Party C is Busy When Transfer Attempted

For simplification purposes, none of the following figures indicate the opening and closing of logical channels (and the associated media sessions) because the control procedures are consistent with typical non-transfer related H.323 calls.

3.2.4.1 Party B Rejects Transfer

As indicated in Figure 15, the application at the transferred endpoint (party B) may call the **gc_RejectXfer( )** function to signal via the ctInitiate.ReturnResult APDU that it cannot participate in a transfer. As a result, the GCEV_INVOKE_XFER_REJ termination event is received at transferring endpoint (party A) and the original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

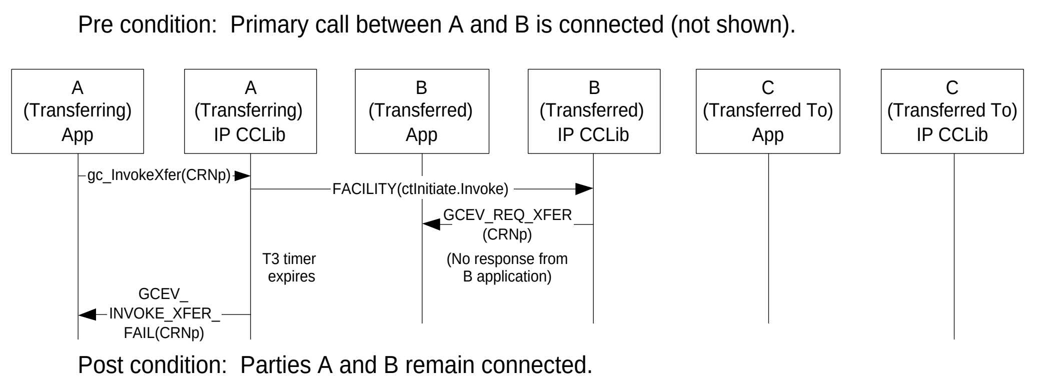
Figure 15. H.450.2 Blind Call Transfer Failure - Party B Rejects Call Transfer


Pre condition: Primary call between A and B is connected (not shown).
A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer(CRNp)
FACILITY(ctInitiate.Invoke)
GCEV_REQ_XFER (CRNp)
gc_RejectXfer(CRNp, GCVAL_REJREASON_UNAVAIL)
GCEV_INVOKE_XFER_REJ (CRNp, GCRV_REMOTEREJ_UNAVAIL)
FACILITY(ctInitiate.ReturnResult = notAvailable)
GCEV_REJ_XFER (CRNp)
Post condition: Parties A and B remain connected.

3.2.4.2 No Response From Party B

As indicated in Figure 16, the transferred endpoint (party B) may not respond to the ctInitiate.ReturnResult APDU which would cause the T3 timer configured as 20 seconds at the transferring endpoint (party A) to expire. As a result, the GCEV_INVOKE_XFER_FAIL termination event would be received at transferring endpoint (party A) and the original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

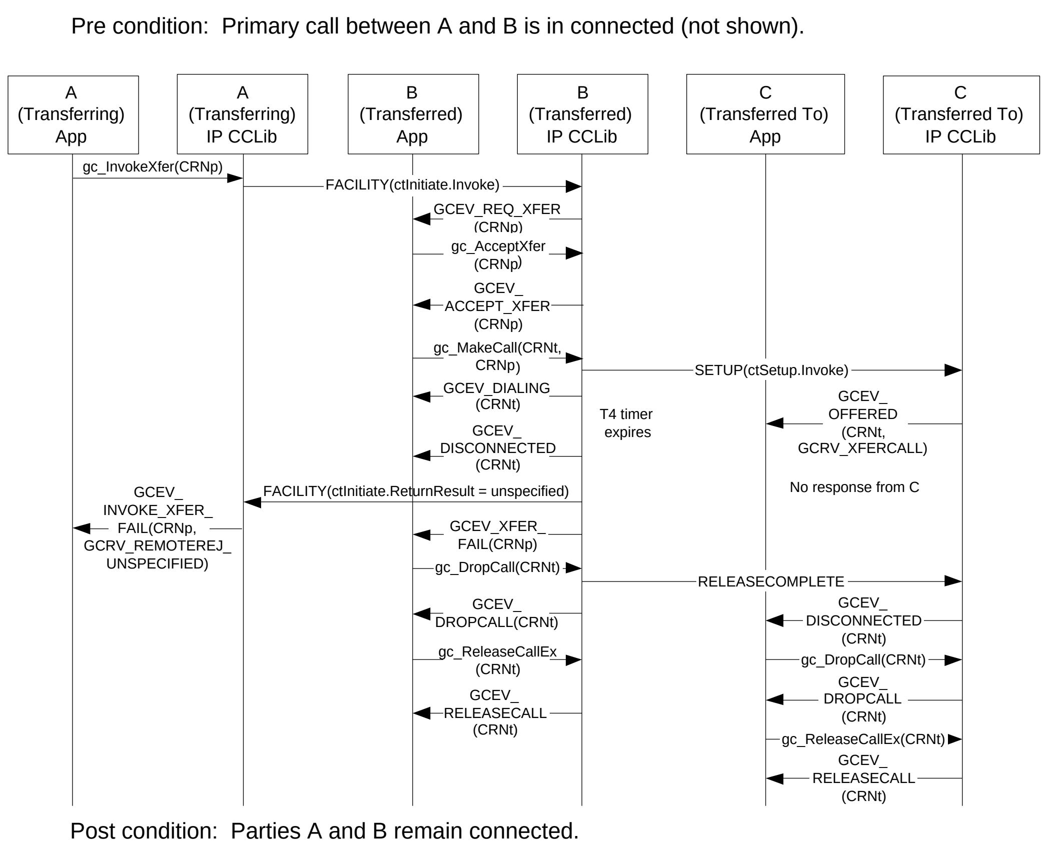
Figure 16. H.450.2 Blind Call Transfer Failure - No Response from Party B


Pre condition: Primary call between A and B is connected (not shown).
A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer(CRNp)
FACILITY(ctInitiate.Invoke)
GCEV_REQ_XFER (CRNp)
T3 timer expires
(No response from B application)
GCEV_ INVOKE_XFER_ FAIL(CRNp)
Post condition: Parties A and B remain connected.

3.2.4.3 No Response From Party C

As indicated in Figure 17, the transferred-to endpoint (party C) may not respond to the incoming call which would cause the T4 timer configured as 20 seconds at the transferred endpoint (party B) to expire. As a result, the transferred endpoint (party B) receives the GCEV_DISCONNECTED event for the transferred call timeout and after sending a ctInitiate.ReturnResult = Unspecified APDU receives the GCEV_XFER_FAIL event on the primary call. Upon receiving the ctInitiate.ReturnResult = Unspecified APDU, the transferring endpoint (party A) is notified by the GCEV_INVOKE_XFER_FAIL termination event and the original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

Figure 17. H.450.2 Blind Call Transfer Failure - No Response from Party C

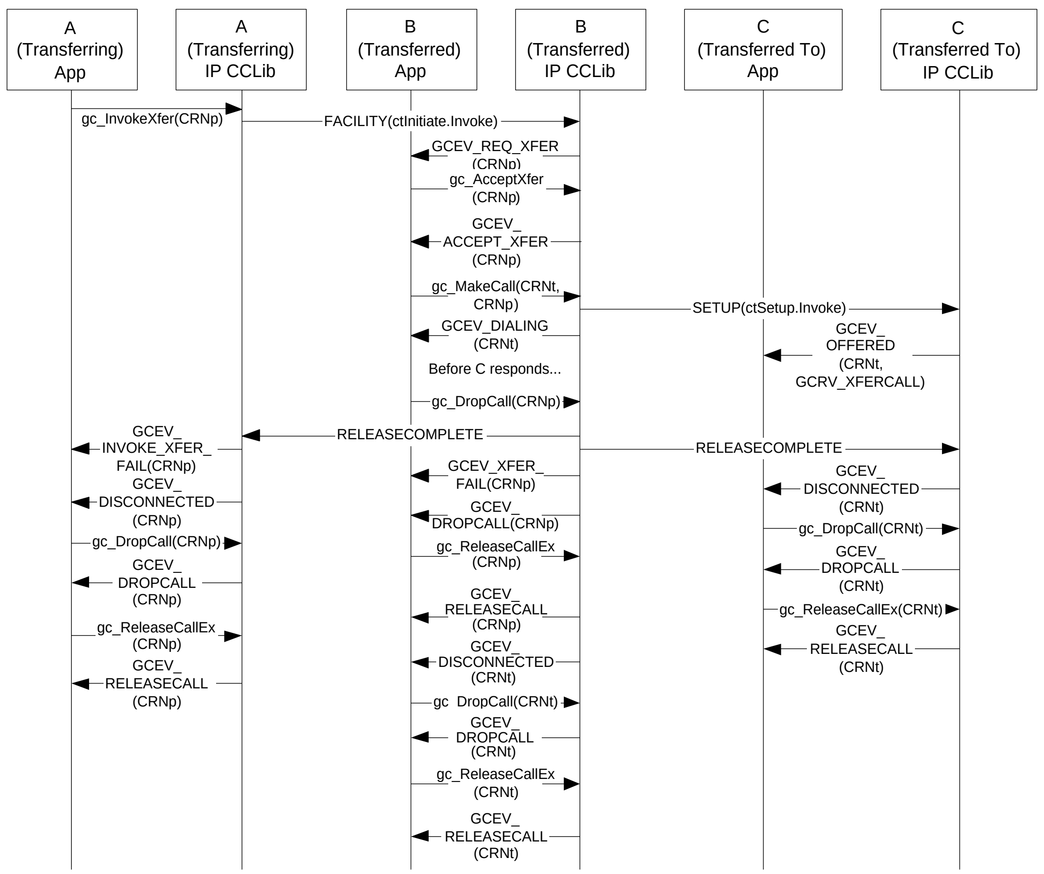


3.2.4.4 Party B Clears Primary Call Before Transfer is Completed

The primary call may be cleared at any time while a blind transfer is in progress. As indicated in Figure 18, the transferred endpoint (party B) may clear the primary call while awaiting acknowledgement from the transferred-to endpoint (party C). As a result, the GCEV_INVOKE_XFER_FAIL termination event is received at transferring endpoint (party A) followed by a GCEV_DISCONNECTED. Similarly, the GCEV_XFER_FAIL termination event is received at the transferred endpoint (party B) followed by a GCEV_DROPCALL. At this point party A must drop and release the call while party B must release the call. The transferred call will also be abandoned implicitly per H.450.2 once the primary call is abandoned. The transferred-to endpoint will receive the GCEV_DISCONNECTED event at which point it must explicitly drop and release the abandoned transferred call attempt. Note that if instead party A initiated the clearing of the primary call while blind transfer is in progress, the only major difference with the corollary is that party B and not A would react to the primary disconnect (in the same manner as before) and explicitly drop the primary call; otherwise, the behavior is identical.

Figure 18. H.450.2 Blind Call Transfer Failure - Party B Clears Primary Call Before Transfer is Completed

Pre condition: Primary call between A and B is in connected (not shown).



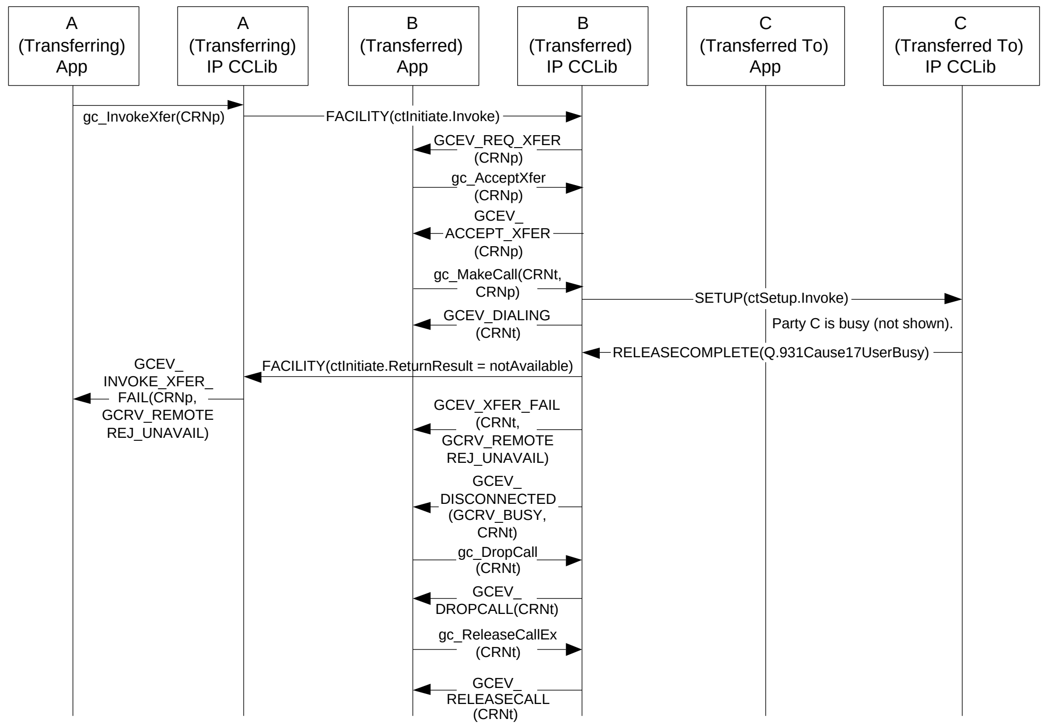
Post condition: Both primary and transferred calls are dropped and released.

3.2.4.5 Party C is Busy When Transfer Attempted

The transferred-to endpoint (party C) may also be busy at the time of transfer (unknown to the transferring endpoint). As indicated in Figure 19, this would result in a Release Complete message with Q.931 Cause 17, User Busy, being returned to the transferred endpoint (party B) indicated to its application via a GCEV_DISCONNECTED event with a cause of GCRV_BUSY. The transferred endpoint (party B) returns a ctInitiate.ReturnError APDU to the transferring endpoint to indicate the transfer failure and in turn must drop the transferred call attempt. As a result, the GCEV_INVOKE_XFER_FAIL termination event is received at transferring endpoint (party A) and the original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

Figure 19. H.450.2 Blind Call Transfer Failure - Party C is Busy When Transfer Attempted

Pre condition: Primary call between A and B is in connected (not shown).
Party C has call connected to another party (not shown).


A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer(CRNp)
FACILITY(ctInitiate.Invoke)
GCEV_REQ_XFER (CRNp)
gc_AcceptXfer (CRNp)
GCEV_ ACCEPT_XFER (CRNp)
gc_MakeCall(CRNt, CRNp)
SETUP(ctSetup.Invoke)
GCEV_DIALING (CRNt)
Party C is busy (not shown).
RELEASECOMPLETE(Q.931Cause17UserBusy)
FACILITY(ctInitiate.ReturnResult = notAvailable)
GCEV_ INVOKE_XFER_ FAIL(CRNp, GCRV_REMOTE REJ_UNAVAIL)
GCEV_XFER_FAIL (CRNt, GCRV_REMOTE REJ_UNAVAIL)
GCEV_ DISCONNECTED (GCRV_BUSY, CRNt)
gc_DropCall (CRNt)
GCEV_ DROPCALL(CRNt)
gc_ReleaseCallEx (CRNt)
GCEV_ RELEASECALL (CRNt)

Post condition: Parties A and B remain connected.
Party C also remains connected (to another party not shown).

3.2.5 Endpoint Behavior in H.450.2 Supervised Call Transfer

This section describes the behavior of each of the three endpoints in a supervised call transfer under H.450.2. The assumed preconditions for supervised call transfer are:

- The transferring endpoint (party A) and the transferred endpoint (party B) are participating in an active call, known as the primary call. From the perspective of the Global Call API, party A and party B are both in the GCST_CONNECTED state.
- The transferring endpoint and the transferred-to endpoint (party C) are participating in an active call, known as the secondary or consultation call. From the perspective of the Global Call call control library, party A and party C are both in the GCST_CONNECTED state. If party C uses Global Call and is not in the connected state when the transfer is invoked, it may fail to receive the Global Call event for the transfer request (GCEV_REQ_INIT_XFER), which will cause it to receive a GCEV_TASKFAIL.

3.2.5.1 Transferring Endpoint (Party A) Role

As in the case of blind transfer, the transferring endpoint initiates the supervised transfer by calling the **gc_InvokeXfer( )** function. The distinction between blind and supervised transfer usage is the addition of the CRN of the secondary (consultation) call. Calling the **gc_InvokeXfer( )** function at the transferring endpoint with two CRN values results in the sending of an ctIdentify.Invoke APDU in a Facility message to the transferred-to endpoint (party C). Once a positive acknowledgement from the transferred-to endpoint is received via a ctIdentify.ReturnResult APDU in a Facility message, the transferring endpoint automatically proceeds to invoke the actual call transfer by sending an ctInitiate.Invoke APDU in a Facility message to the transferred endpoint (party B).

From this point forward, from the perspective of this endpoint, the behavior is similar to that of a blind or unsupervised transfer. The one difference is that the secondary, consultation call is disconnected once the transferred call is answered at the transferred-to endpoint (party C) and must be explicitly dropped and released. Note however, if the transferred call goes unanswered or fails, the secondary call is left active and maintained in the GCST_CONNECTED state.

3.2.5.2 Transferred Endpoint (Party B) Role

The endpoint to be transferred (party B) has no knowledge of the origins of the destination address information a priori in that it was retrieved as a result of a consultation call. Thus, from the perspective of this endpoint, the behavior and handling of supervised transfer is exactly the same as that of blind transfer.

3.2.5.3 Transferred-To Endpoint (Party C) Role

At any point in time after a secondary consultation call is answered by the transferred-to endpoint, a Facility(ctIdentify.Invoke) request may arrive requesting whether it is able to participate in an upcoming transfer as signified by the GCEV_REQ_INIT_XFER event. Assuming that the endpoint is capable, the application calls the **gc_AcceptInitXfer( )** function to accept the transfer along with the intended rerouting number address in the **reroutinginfop** GC_REROUTING_INFO pointer parameter. The IP CCLIB internally returns a newly created callIdentity for the transferred call to use.

From this point forward, the behavior in handling the incoming transferred call from the perspective of this endpoint is identical to that of a blind or unsupervised transfer. The only difference is that the secondary, consultation call is disconnected once the transferred call is answered and must be explicitly dropped and released.

3.2.6 Successful H.450.2 Supervised Call Transfer Scenario

As indicated in Figure 20, the first precondition for supervised H.450.2 transfer is that the transferring endpoint (party A) and the transferred endpoint (party B) are participating in an active call (the primary call) and from the Global Call perspective, in the GCST_CONNECTED state.

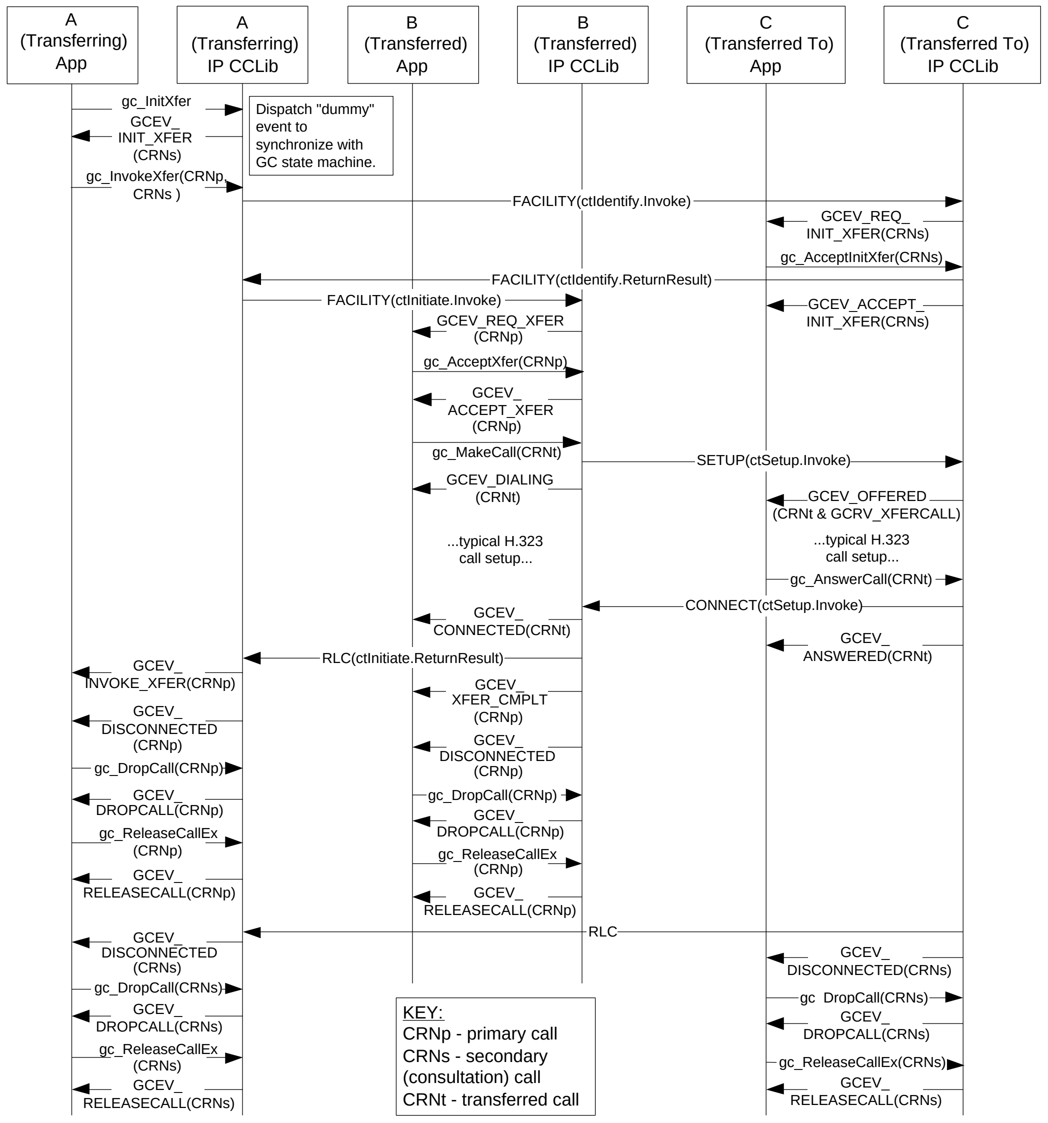
The second precondition for supervised transfer is that a secondary call (the consultation call) from the transferring endpoint (party A) to the transferred-to endpoint (party C) is active and both endpoints are in the GCST_CONNECTED state.

Completion of a successful supervised transfer results in the eventual termination of the primary and secondary (consultation) calls, and the creation of the transferred call. Note that the connection of the transferred call is not a mandate for supervised call transfer. While less likely due to the typical human dialogue on a secondary (consultation) call, it is always possible that the transferred call itself may be left unanswered and eventually abandoned and still be considered a *successful* transfer from the signaling perspective of the transferring endpoint (party A).

For simplification purposes Figure 20 does not indicate the opening and closing of logical channels (and the associated media sessions) because the control procedures are consistent with typical non-transfer related H.323 calls.

Figure 20. Successful H.450.2 Supervised Call Transfer

Pre condition: Primary call between A and B is connected.
Secondary (consultation) call between A and C is connected (not shown).



Post condition: Transferred call between B and C offered (optional whether connected or not).
Primary call between A and B dropped and released.
Secondary (consultation) call between A and C dropped and released.

3.2.7 Unsuccessful H.450.2 Supervised Transfer Scenarios

Note: The same failures that can potentially occur in blind transfer, may take place in supervised transfer as well. See Section 3.2.4, "Unsuccessful H.450.2 Blind Call Transfer Scenarios", on page 53 for more information. The difference being that the secondary, consultation may optionally be cleared as specified in H.450.2.

There are a several other scenarios where a supervised call transfer may fail. The most common scenarios are described in the following topics:

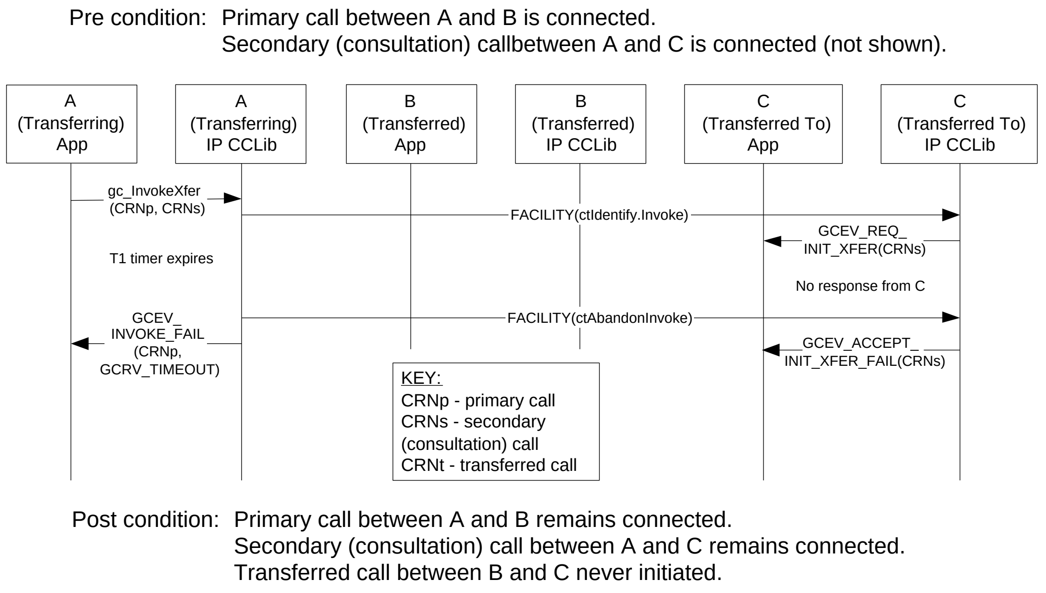
- Party C Timeout
- Party C Rejects the Transfer Request
- Party B Rejects the Transfer Request
- Party B Timeout

For simplification purposes, none of the following figures indicate the opening and closing of logical channels (and the associated media sessions) because the control procedures are consistent with typical non-transfer related H.323 calls.

3.2.7.1 Party C Timeout

As indicated in Figure 21, the user or application at the transferred-to endpoint (party C) may fail to respond to the ctIdentify.Invoke request causing the timer 1 to expire at the transferring endpoint (party A) resulting in an abandoned transfer attempt. As a result, the GCEV_INVOKE_XFER_FAIL termination event is received at transferring endpoint (party A). Both the original primary call and the secondary, consultation call are left connected and in the GCST_CONNECTED state from the perspective of both A and B (primary) and A and C (secondary).

Figure 21. H.450.2 Supervised Call Transfer Failure - Party C Timeout



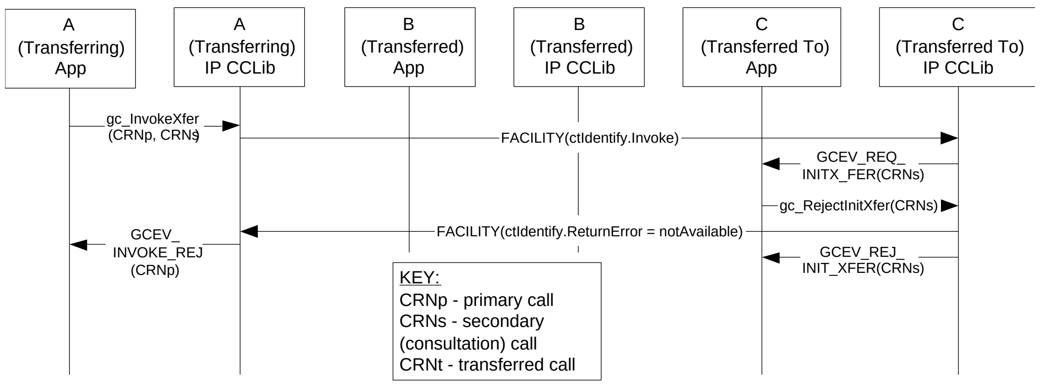
Pre condition: Primary call between A and B is connected.
Secondary (consultation) callbetween A and C is connected (not shown).
A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer (CRNp, CRNs)
FACILITY(ctIdentify.Invoke)
GCEV_REQ_INIT_XFER(CRNs)
T1 timer expires
No response from C
FACILITY(ctAbandonInvoke)
GCEV_INVOKE_FAIL (CRNp, GCRV_TIMEOUT)
GCEV_ACCEPT_INIT_XFER_FAIL(CRNs)
KEY:
CRNp - primary call
CRNs - secondary (consultation) call
CRNt - transferred call
Post condition: Primary call between A and B remains connected.
Secondary (consultation) call between A and C remains connected.
Transferred call between B and C never initiated.

3.2.7.2 Party C Rejects the Transfer Request

As indicated in Figure 22, the user or application at the transferred-to endpoint (party C) may call the **gc_RejectInitXfer( )** function to signal via the ctInIdentify.ReturnResult APDU that it cannot participate in a transfer. As a result, the GCEV_INVOKE_XFER_REJ termination event is received at the transferring endpoint (party A). Both the original primary call and the secondary, consultation call are left connected and in the GCST_CONNECTED state from the perspective of both A and B (primary) and A and C (secondary); GCST_CONNECTED state from the perspective of both A and B.

Figure 22. H.450.2 Supervised Call Transfer Failure - Party C Rejects the Transfer Request

Pre condition: Primary call between A and B is connected.
Secondary (consultation) call between A and C is connected (not shown).

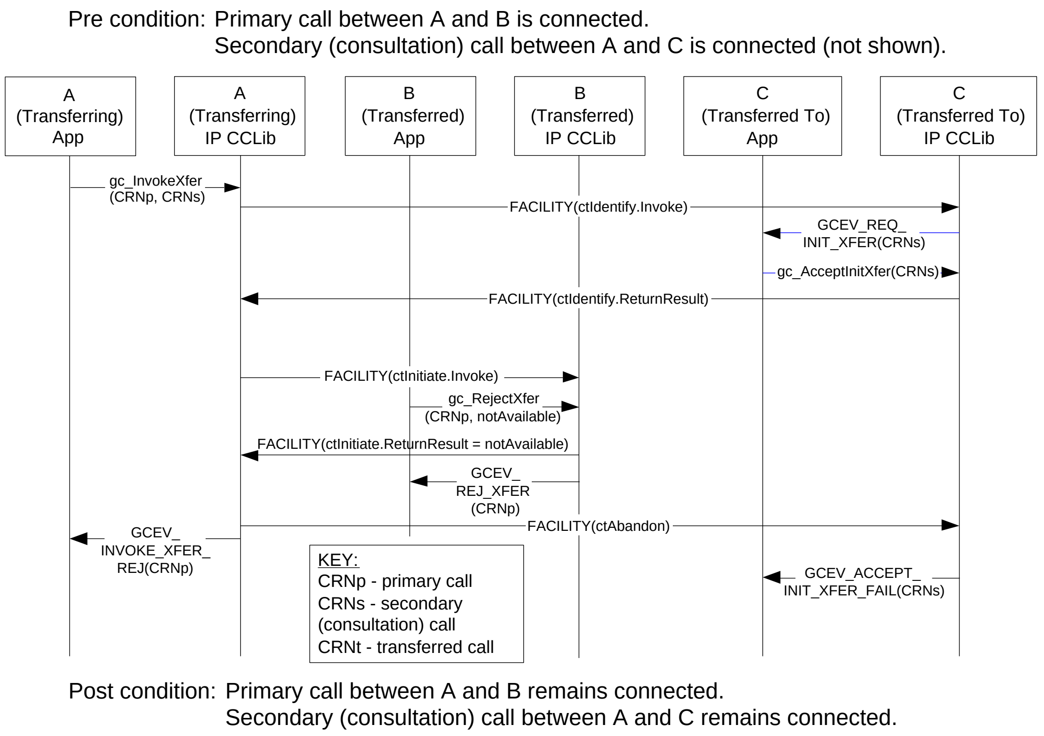

A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer (CRNp, CRNs)
FACILITY(ctIdentify.Invoke)
GCEV_REQ_ INITX_FER(CRNs)
gc_RejectInitXfer(CRNs)
FACILITY(ctIdentify.ReturnError = notAvailable)
GCEV_ INVOKE_REJ (CRNp)
GCEV_REJ_ INIT_XFER(CRNs)
KEY:
CRNp - primary call
CRNs - secondary (consultation) call
CRNt - transferred call

Post condition: Primary call between A and B remains connected.
Secondary (consultation) call between A and C remains connected.

3.2.7.3 Party B Rejects the Transfer Request

As indicated in Figure 23, the user or application at the transferred endpoint (party B) may call the **gc_RejectXfer( )** function to reject the transfer request and signal via the ctInitiate.ReturnResult APDU that it cannot participate in a transfer. As a result, the GCEV_INVOKE_XFER_REJ termination event is received at transferring endpoint (party A). Both the original primary call and the secondary, consultation call are left connected and in the GCST_CONNECTED state from the perspective of both A and B (primary) and A and C (secondary); GCST_CONNECTED state from the perspective of both A and B.

Figure 23. H.450.2 Supervised Call Transfer Failure - Party B Rejects the Transfer Request

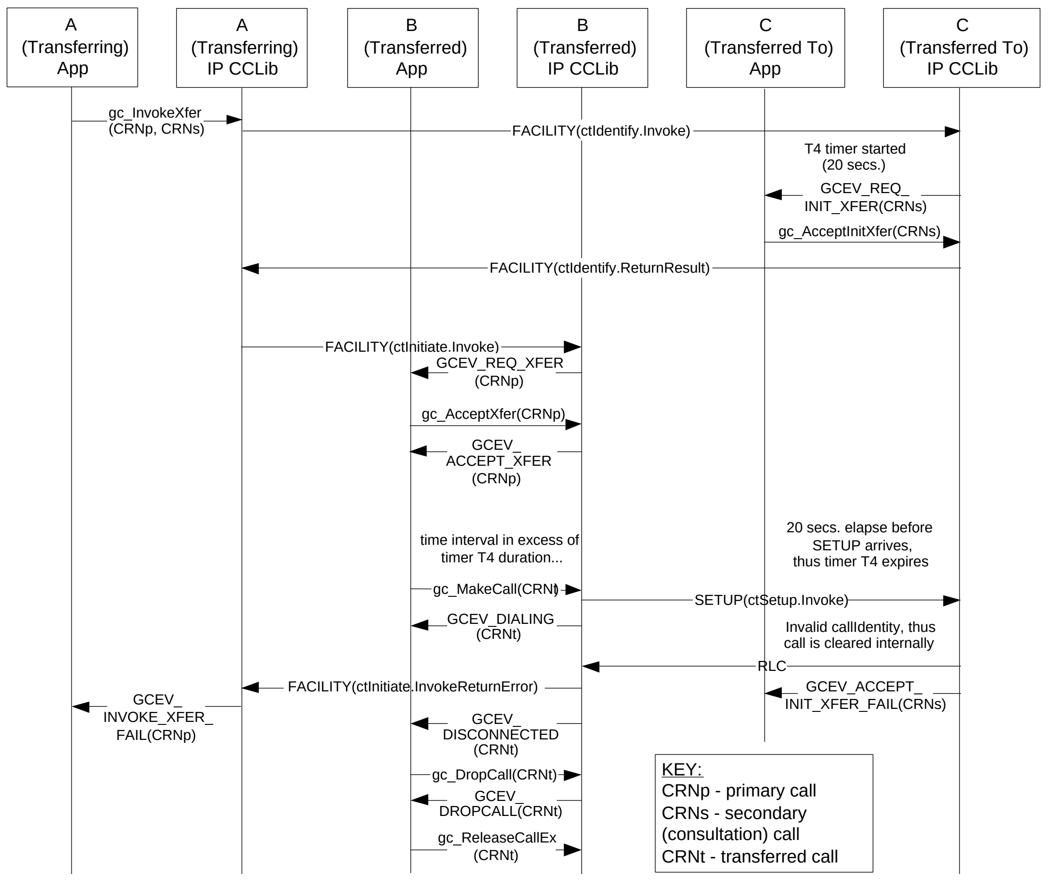

Pre condition: Primary call between A and B is connected.
Secondary (consultation) call between A and C is connected (not shown).
A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer (CRNp, CRNs)
FACILITY(ctIdentify.Invoke)
GCEV_REQ_ INIT_XFER(CRNs)
gc_AcceptInitXfer(CRNs)
FACILITY(ctIdentify.ReturnResult)
FACILITY(ctInitiate.Invoke)
gc_RejectXfer (CRNp, notAvailable)
FACILITY(ctInitiate.ReturnResult = notAvailable)
GCEV_ REJ_XFER (CRNp)
FACILITY(ctAbandon)
GCEV_ INVOKE_XFER_ REJ(CRNp)
GCEV_ACCEPT_ INIT_XFER_FAIL(CRNs)
KEY:
CRNp - primary call
CRNs - secondary (consultation) call
CRNt - transferred call
Post condition: Primary call between A and B remains connected.
Secondary (consultation) call between A and C remains connected.

3.2.7.4 Party B Timeout

As indicated in Figure 24, the user or application at the transferred-to endpoint (party C) may receive the transferred call after the T4 timer expires. If this is the case and the callIdentity is cleared as a result of the T4 expiry, the transferred-to endpoint will clear or reject the transferred call as indicated by a GCEV_DISCONNECTED event at the transferred endpoint (party B) and a GCEV_INVOKE_XFER_FAIL event at the transferring endpoint (party A). Both the original primary call and the secondary, consultation call are left connected and in the GCST_CONNECTED state from the perspective of both A and B (primary) and A and C (secondary); GCST_CONNECTED state from the perspective of both A and B.

Figure 24. H.450.2 Supervised Call Transfer Failure - Party B Timeout

Pre condition: Primary call between A and B is connected.
Secondary (consultation) call between A and C is connected (not shown).


A (Transferring) App
A (Transferring) IP CCLib
B (Transferred) App
B (Transferred) IP CCLib
C (Transferred To) App
C (Transferred To) IP CCLib
gc_InvokeXfer (CRNp, CRNs)
FACILITY(ctIdentify.Invoke)
T4 timer started (20 secs.)
GCEV_REQ_ INIT_XFER(CRNs)
gc_AcceptInitXfer(CRNs)
FACILITY(ctIdentify.ReturnResult)
FACILITY(ctInitiate.Invoke)
GCEV_REQ_XFER (CRNp)
gc_AcceptXfer(CRNp)
GCEV_ ACCEPT_XFER (CRNp)
time interval in excess of timer T4 duration...
20 secs. elapse before SETUP arrives, thus timer T4 expires
gc_MakeCall(CRNt)
SETUP(ctSetup.Invoke)
GCEV_DIALING (CRNt)
Invalid callIdentity, thus call is cleared internally
RLC
FACILITY(ctInitiate.InvokeReturnError)
GCEV_ACCEPT_ INIT_XFER_FAIL(CRNs)
GCEV_ INVOKE_XFER_ FAIL(CRNp)
GCEV_ DISCONNECTED (CRNt)
gc_DropCall(CRNt)
GCEV_ DROPCALL(CRNt)
gc_ReleaseCallEx (CRNt)
KEY:
CRNp - primary call
CRNs - secondary (consultation) call
CRNt - transferred call

Post condition: Primary call between A and B remains connected.
Secondary (consultation) call between A and C remains connected.

3.3 Call Transfer Scenarios When Using SIP

The Dialogic® Global Call API functions that supports IP call transfer are described in the *Global Call API Library Reference*; protocol-specific information about the individual call transfer APIs can be found in the subsections of Section 8.3, "Dialogic® Global Call API Function Variances for IP". General information on implementing call transfer can be found in Section 4.38, "T.38 Fax Server", on page 374, and SIP-specific details can be found in Section 4.5.5, "Call Transfer When Using SIP", on page 141.

The following topics describe the call transfer capabilities provided when using the SIP call transfer supplementary service:

- General Conditions for SIP Call Transfers
- Endpoint Behavior in Unattended SIP Call Transfers
- Successful Unattended SIP Call Transfer Scenarios
- Endpoint Behavior in Attended SIP Transfers
- Successful SIP Attended Call Transfer Scenarios
- Unsuccessful Call Transfer Scenarios

3.3.1 General Conditions for SIP Call Transfers

SIP call transfer uses the REFER method (with NOTIFY support) to reroute a call (a SIP dialog) after the call has been established; in other words, after two endpoints have an established media path.

There are two fundamental types of call transfer:

- Unattended transfer, which is referred to as "blind transfer" in most other technologies and protocols. In this type of transfer the transferring party (called the Transferor in SIP) has a call (or SIP dialog) with the transferred party (called the Transferee in SIP) but not with the transferred-to party (called the Transfer Target in SIP).
- Attended transfer, which is referred to as "supervised transfer" in most other technologies and protocols. In this type of transfer, the Transferor has a dialog with both the Transferee and the Transfer Target.

In its simplest terms, a SIP call transfer involves the Transferor issuing a REFER to the Transferee to cause the Transferee to issue an INVITE to the Transfer Target. The Transferee and Transfer Target negotiate the media without regard to the media that had been negotiated between the Transferor and the Transferee, just as if the Transferee had initiated the INVITE on its own.

Once a transfer request is accepted by the Transferee, the Transferor is not allowed to send another transfer request to the Transferee. Only if a transfer request is rejected or fails is the Transferor allowed to attempt another transfer request to Transferee.

The disposition of the media streams between the Transferor and the Transferee is not altered by the REFER method. A successful REFER transaction does not terminate the session between the Transferor and the Transferee; if those parties wish to terminate their session, they must do so with a subsequent BYE request.

In the SIP call transfer protocol the Transferor is notified when the Transferee accepts the REFER transfer request. The Dialogic® Global Call API Library allows this notification to be signaled to the application as a GCEV_INVOKE_XFER_ACCEPTED event. This event is optional, and is disabled (or masked) by default. The party A application can enable and disable this event at any time after the line device is opened using the **gc_SetConfigData( )** function. See Section 4.5.5.1, "Enabling GCEV_INVOKE_XFER_ACCEPTED Events", on page 141 for details.

When performing a call transfer operation, all involved call handles must be on the same stack instance. This imposes the following application restrictions for call transfer operations

- When performing an attended call transfer at party A, both the consultation line device and the transferring line device must be on the same virtual board.
- When performing a call transfer (either attended or unattended) at party B, both the transferring line device and the transferred line device must be on the same virtual board.
- When performing an attended call transfer at party C, both the consultation line device and the transferred-to line device must be on the same virtual board.

Interoperability Issues

The latest standards for the SIP REFER method are defined in IETF RFC 3515, published in April 2003. The current Global Call implementation is compliant with RFC 3515, but many existing implementations of REFER are based on the previous draft of the REFER method and are not fully compliant. The most significant non-compliance issues are:

- no initial NOTIFY after sending out 202 accept to REFER request
- no subscription state information in NOTIFY message
- no NOTIFY generated by the Transferee (Transferred party) after the call is terminated
- any NOTIFY received by the Transferor (Transferring party) after the subscription is terminated or the call is terminated will be rejected. Note that the subscription can be terminated implicitly after receiving NOTIFY of 180 Ringing.

3.3.2 Endpoint Behavior in Unattended SIP Call Transfers

The precondition for unattended call transfer (commonly referred to as "blind call transfer" in other technologies and protocols) is that the transferring endpoint (party A, or Transferor in SIP terminology) and the transferred endpoint (party B or Transferee in SIP terms) are participating in an active call, known as the primary call. From the perspective of the Dialogic® Global Call API, both parties are in the GCST_CONNECTED state. Completion of a successful unattended transfer results in the eventual termination of the primary call, and the creation of the transferred call between party B and the Transfer Target (party C).

3.3.2.1 Transferor or Transferring Endpoint (party A)

The Transferor (party A) initiates an unattended transfer by calling the **gc_InvokeXfer( )** function on the CRN of the primary call (CRNp), which results in the sending a REFER message to the Transferee (party B). The Refer-To header in the REFER request is constructed from either the char *numberstr or the GC_MAKECALL_BLK *makecallp parameter in the **gc_InvokeXfer( )** function, following the same rules as **gc_MakeCall( )**. The Referred-By header is automatically constructed with the local URI—the same as the From or To header, depending on the direction of the initial call INVITE. Optionally, the Transferor can override the default Referred-By header by inserting a Referred-By header in the **gc_InvokeXfer( )** parm block. Party A will be notified if REFER is accepted or rejected by transferred endpoint (party B).

If party A receives a 2xx response to the REFER (indicating that is was accepted by party B), a GCEV_INVOKE_XFER_ACCEPTED event may optionally be generated. This optional event is disabled by default; after the line device has been opened, the event can be enabled or disabled at any time by use of the **gc_SetConfigData( )** function.

The primary call may be terminated by either party before transferred call is completed. Unlike an H.450.2 transfer, party A in a SIP transfer will **not** get any transfer termination event if party A terminates the primary call before receiving final status from party B. This is because there is no way to be sure if the transfer is successful or if it failed and it is party A's responsibility to update the application transfer states in this case. This is a common scenario in blind transfer where party A does not care about the transferred call status and drops the primary call immediately after receiving a GCEV_INVOKE_XFER_ACCEPTED event.

When the REFER subscription is terminated, party A rejects subsequent NOTIFY messages. Any of the following events terminate the REFER subscription:

- a NOTIFY with subscription state terminated is received
- a NOTIFY of 180 Ringing is received
- a 2xx-6xx final response is received
- the primary call is terminated

If the primary call remains connected and the REFER subscription is alive, party A **may** be notified of the final status of transferred call from party B. The notification of transferred call status is optional depending on party B.

From party A's perspective, a call transfer is considered successful as long as GCEV_INVOKE_XFER_ACCEPTED (if enabled) and GCEV_INVOKE_XFER events are received. If the optional GCEV_INVOKE_XFER_ACCEPTED event type is enabled, that event is generated by receiving a 2xx response (to the REFER request) from party B. The GCEV_INVOKE_XFER event is generated by receiving from party B either a NOTIFY of termination of the REFER subscription or a NOTIFY of 180 Ringing or 2xx final status on the transferred call.

The REFER subscription will be terminated and the primary call will also be disconnected locally immediately after generating a GCEV_INVOKE_XFER event. From the Global Call API perspective, the primary call is terminated at the transferring endpoint as indicated by the GCEV_DISCONNECTED event implying the Transferor endpoint is then responsible for dropping and releasing the primary call.

3.3.2.2 Transferee or Transferred Endpoint (Party B)

The endpoint to be transferred (party B, or Transferee in SIP terms) is notified of the request to transfer from the initiating endpoint via a GCEV_REQ_XFER event on CRNp. If party B accepts the transfer request via **gc_AcceptXfer( )** function call on CRNp, a 202 Accepted response is sent to party A. Sending 202 Accepted to party A starts the REFER subscription, whereupon party B automatically sends a NOTIFY of 100 Trying (with default expiration time of 300 seconds) to party A on CRNp. No further notification of 100 Trying is sent from party B to party A during the call transfer process.

Party B retrieves the destination address information from the unsolicited transfer request via the GC_REROUTING_INFO structure passed with the GCEV_REQ_XFER event. Party B uses the rerouting address information (Refer-To address) to initiate a call to the new destination party via **gc_MakeCall( )** on CRNt. From the perspective of the application, this transferred call is treated in the same manner as a normal singular call and the party receives intermediate call state events as to the progress of the call (e.g., GCEV_DIALING, GCEV_ALERTING, GCEV_PROCEEDING, and GCEV_CONNECTED).

If the CRNp number is included during the **gc_MakeCall( )** on CRNt and the primary call is in the connected state, then a GCEV_XFER_CMPLT event is generated on CRNp once the transferred call is connected. If the CRNp number is not included, there will be no notification to the primary call and/or party A of the transferred call status. The CRNp number must not be included in the **gc_MakeCall()** if primary call was disconnected prior to making transferred call.

When party B receives any provisional response except 100 Trying from Party C and if the REFER subscription is still alive, party B automatically sends NOTIFY to party A with such transferred call status.

When party B receives the indication from party C that the call transfer was successful (200 OK), the party B application is notified of the success via a GCEV_XFER_CMPLT event on CRNp. If the primary call is still connected, party B will notify party A of the transfer status (200 OK) and terminate the REFER subscription. Then party B implicitly, without user/application initiation, disconnects the primary call with party A. Although the primary call to party A is implicitly dropped, the call itself must still be explicitly dropped via **gc_DropCall( )** and released via **gc_ReleaseCallEx( )** to resynchronize the local state machine.

Either the party A or party B application may terminate the primary call after party B accepts the transfer request. If the primary call is terminated by party A before receiving any call transfer termination event (GCEV_INVOKE_XFER or GCEV_INVOKE_XFER_FAIL), party B will not notify party A of the transfer status. If the primary call is terminated by party B before receiving any transferred call provisional or final response from party C, party B *will* send NOTIFY to party A with 200 OK and terminate the REFER subscription before sending BYE to party A.

If the primary call is disconnected before making the transferred call to party C, party B must not include the primary call CRN (CRNp) when making the transferred call to party C. Otherwise, a Global Call error will be returned.

Note that the primary call can be disconnected prior to making the transferred call only during an unattended transfer because the transferred call can be established independently from the primary call. During an attended transfer, the transferred call cannot be established after the primary call is disconnected because the primary call database contains the Replaces information that is required by the transferred call.

If the Referred-By header exists in the REFER message, it is passed to the application via the GCEV_REQ_XFER event if SIP message information access was enabled (by setting the

IP_SIP_MSGINFO_ENABLE in the sip_msginfo_mask field of the IP_VIRTBOARD data structure) when the virtual board was started.

3.3.2.3 Transfer Target or Transferred-To Endpoint (Party C)

From the perspective of party C, the transferred call is, for the most part, treated as a typical incoming call. The call is first notified to the application by a GCEV_DETECTED or GCEV_OFFERED event on CRNt. The GCRV_XFERCALL cause value is provided in the event to alert the application that this call offering is the result of a transfer, but only if the incoming INVITE contains Referred-By or Replaces information indicating a new transferred call. Referred-By and Replaces information, if present, is also attached to GCEV_OFFERED events if SIP header access was enabled (by setting the IP_SIP_MSGINFO_ENABLE value in the sip_msginfo_mask field of the IP_VIRTBOARD data structure) when the virtual board was started.

At that point, the application may retrieve the typical calling party information on CRNt. Party C is then provided the same methods of action as a typical incoming call, namely to alert party B that the call is proceeding (typically for gateways), ringback notification that the local user is being alerted, or simply that the call is answered. The only behavior change from this endpoint over typical non-transferred calls is whether to handle the calling party information any differently because it is the result of a transfer.

3.3.3 Successful Unattended SIP Call Transfer Scenarios

This section describes various scenarios for successful call transfers under the SIP protocol. The scenarios include:

- Successful Transfer with Notification of Connection
- Successful Transfer with Notification of Ringing
- Successful Transfer with Early Termination of REFER Subscription
- Successful Transfer with Primary Call Cleared Prior to Transfer Completion

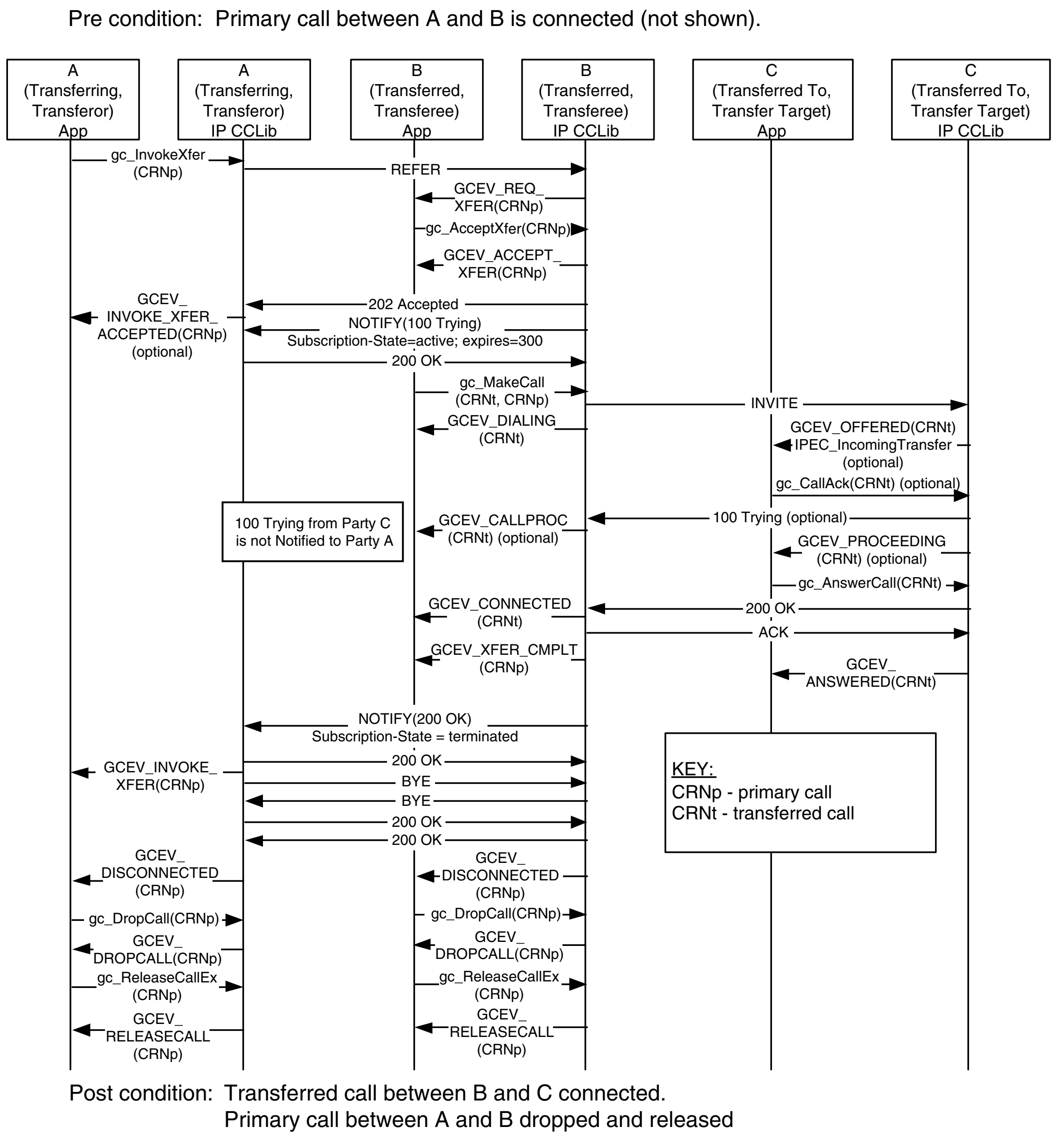
All of the scenarios indicate all three common naming conventions for the three parties involved in a call transfer: parties (A, B, and C), endpoints (transferring, transferred, and transferred-to), and SIP roles (Transferor, Transferee, and Transfer Target). “IP CClib” refers to the call control library and SIP stack portions of Global Call. “Non-Global Call” is used to represent a User Agent that might behave legally but differently than Global Call. Pre and post conditions are explicitly listed in each scenario, but the common pre-condition for all scenarios is that the Transferor (party A) and the Transferee (party B) are participating in an active (primary) call and are in the GCST_CONNECTED state from the perspective of the Global Call API.

All of the following scenarios illustrate the optional GCEV_INVOKE_XFER_ACCEPTED event, which is disabled by default. The party A application can enable and disable this event at any time after the line device is opened using the **gc_SetConfigData( )** function.

3.3.3.1 Successful Transfer with Notification of Connection

Figure 25 illustrates the basic successful scenario, with party A receiving notification from party B after the transferred call between party B and party C has been connected. The SIP dialog for the primary call between party A and party B is automatically disconnected, and both parties then tear down the call using **gc_DropCall( )** and **gc_ReleaseCallEx( )**.

Figure 25. Successful SIP Unattended Call Transfer, Party A Notified with Connection

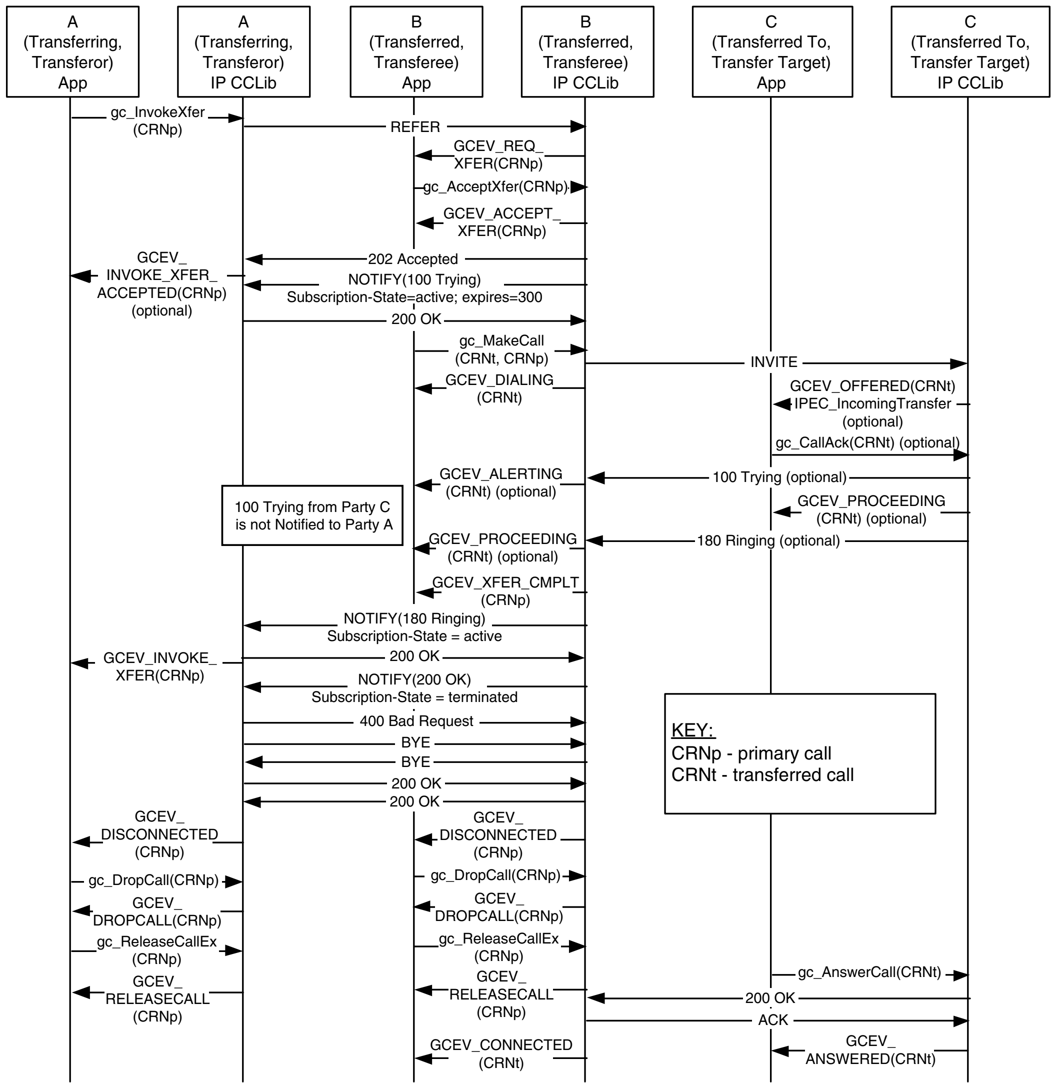

Pre condition: Primary call between A and B is connected (not shown).
A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_REQ_ XFER(CRNp)
gc_AcceptXfer(CRNp)
GCEV_ACCEPT_ XFER(CRNp)
GCEV_ INVOKE_XFER_ ACCEPTED(CRNp) (optional)
202 Accepted
NOTIFY(100 Trying) Subscription-State=active; expires=300
200 OK
gc_MakeCall (CRNt, CRNp)
INVITE
GCEV_DIALING (CRNt)
GCEV_OFFERED(CRNt) IPEC_IncomingTransfer (optional)
gc_CallAck(CRNt) (optional)
100 Trying from Party C is not Notified to Party A
GCEV_CALLPROC (CRNt) (optional)
100 Trying (optional)
GCEV_PROCEEDING (CRNt) (optional)
gc_AnswerCall(CRNt)
GCEV_CONNECTED (CRNt)
200 OK
ACK
GCEV_XFER_CMPLT (CRNp)
GCEV_ ANSWERED(CRNt)
NOTIFY(200 OK) Subscription-State = terminated
KEY: CRNp - primary call CRNt - transferred call
GCEV_INVOKE_ XFER(CRNp)
200 OK
BYE
BYE
200 OK
200 OK
GCEV_ DISCONNECTED (CRNp)
GCEV_ DISCONNECTED (CRNp)
gc_DropCall(CRNp)
gc_DropCall(CRNp)
GCEV_ DROPCALL(CRNp)
GCEV_ DROPCALL(CRNp)
gc_ReleaseCallEx (CRNp)
gc_ReleaseCallEx (CRNp)
GCEV_ RELEASECALL (CRNp)
GCEV_ RELEASECALL (CRNp)
Post condition: Transferred call between B and C connected. Primary call between A and B dropped and released

3.3.3.2 Successful Transfer with Notification of Ringing

Figure 26 illustrates a scenario where party B notifies party A that the transfer has completed as soon as party C responds to the INVITE with a 100 Trying or 180 Ringing. The Call Control Library at Party A disconnects the primary call with party B after the notification and the application then must tear down the call using **gc_DropCall( )** and **gc_ReleaseCallEx( )**.

Figure 26. Successful SIP Unattended Call Transfer, Party A Notified with Ringing

Pre condition: Primary call between A and B is connected (not shown).



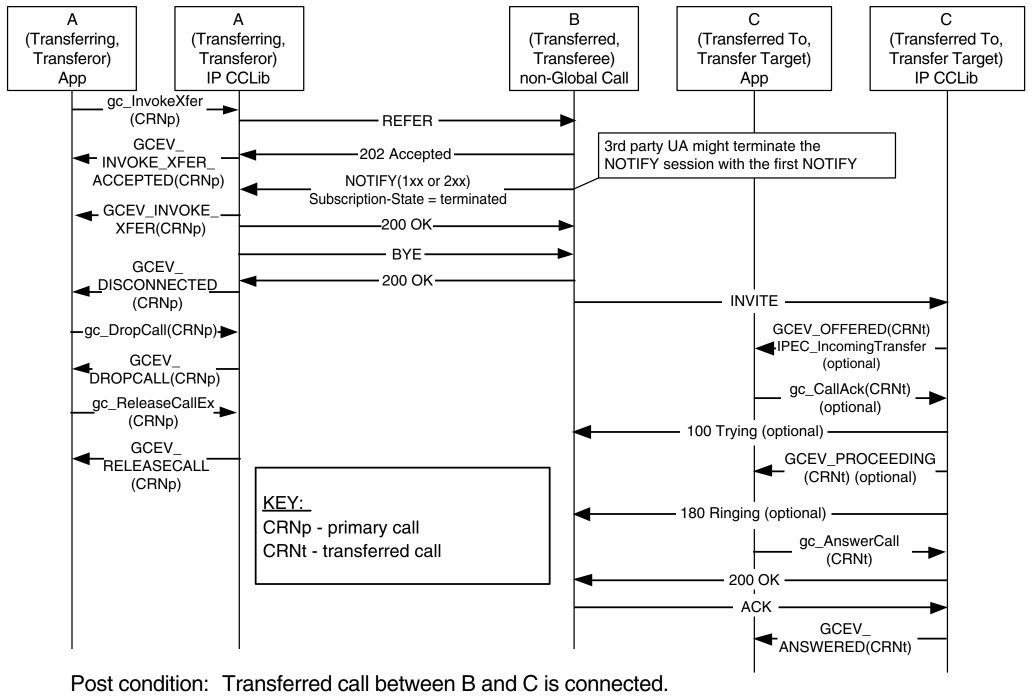
Post condition: Transferred call between B and C is connected.
Primary call between A and B dropped and released

3.3.3.3 Successful Transfer with Early Termination of REFER Subscription

Figure 27 illustrates a valid scenario for which Global Call does not support the party B role. In this scenario, party B terminates the REFER subscription with the first NOTIFY, before party A can be notified of the transferred call status. The Call Control Library at Party A disconnects the primary call with party B after the terminating NOTIFY and the application then must tear down the call using **gc_DropCall( )** and **gc_ReleaseCallEx( )**.

Figure 27. Successful SIP Unattended Call Transfer, Party B Terminates REFER Subscription prior to Notification of Transferred Call Status

Pre condition: Primary call between A and B is connected (not shown).


A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) non-Global Call
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_ INVOKE_XFER_ ACCEPTED(CRNp)
202 Accepted
3rd party UA might terminate the NOTIFY session with the first NOTIFY
NOTIFY(1xx or 2xx) Subscription-State = terminated
GCEV_INVOKE_ XFER(CRNp)
200 OK
BYE
GCEV_ DISCONNECTED (CRNp)
200 OK
INVITE
gc_DropCall(CRNp)
GCEV_OFFERED(CRNt) IPEC_IncomingTransfer (optional)
GCEV_ DROPCALL(CRNp)
gc_CallAck(CRNt) (optional)
gc_ReleaseCallEx (CRNp)
100 Trying (optional)
GCEV_ RELEASECALL (CRNp)
GCEV_PROCEEDING (CRNt) (optional)
KEY:
CRNp - primary call
CRNt - transferred call
180 Ringing (optional)
gc_AnswerCall (CRNt)
200 OK
ACK
GCEV_ ANSWERED(CRNt)

Post condition: Transferred call between B and C is connected.
Primary call between A and B dropped and released

3.3.3.4 Successful Transfer with Primary Call Cleared Prior to Transfer Completion

The SIP protocol supports unattended transfer scenarios where the primary call is cleared or dropped before the transfer completes. In some other technologies and protocols, these scenarios are referred to as “unattended blind transfers” as opposed to “attended blind transfers” where the primary call is maintained until completion. Note that scenarios similar to these are not supported by the H.450.2 protocol.

Figure 28 illustrates a scenario in which party A drops the primary call with party B as soon as it receives notification that party B has accepted the transfer request. In this scenario, party A does not receive any notification that the transfer has completed.

Figure 28. Successful SIP Unattended Call Transfer, Party A Clears Primary Call prior to Transfer Completion

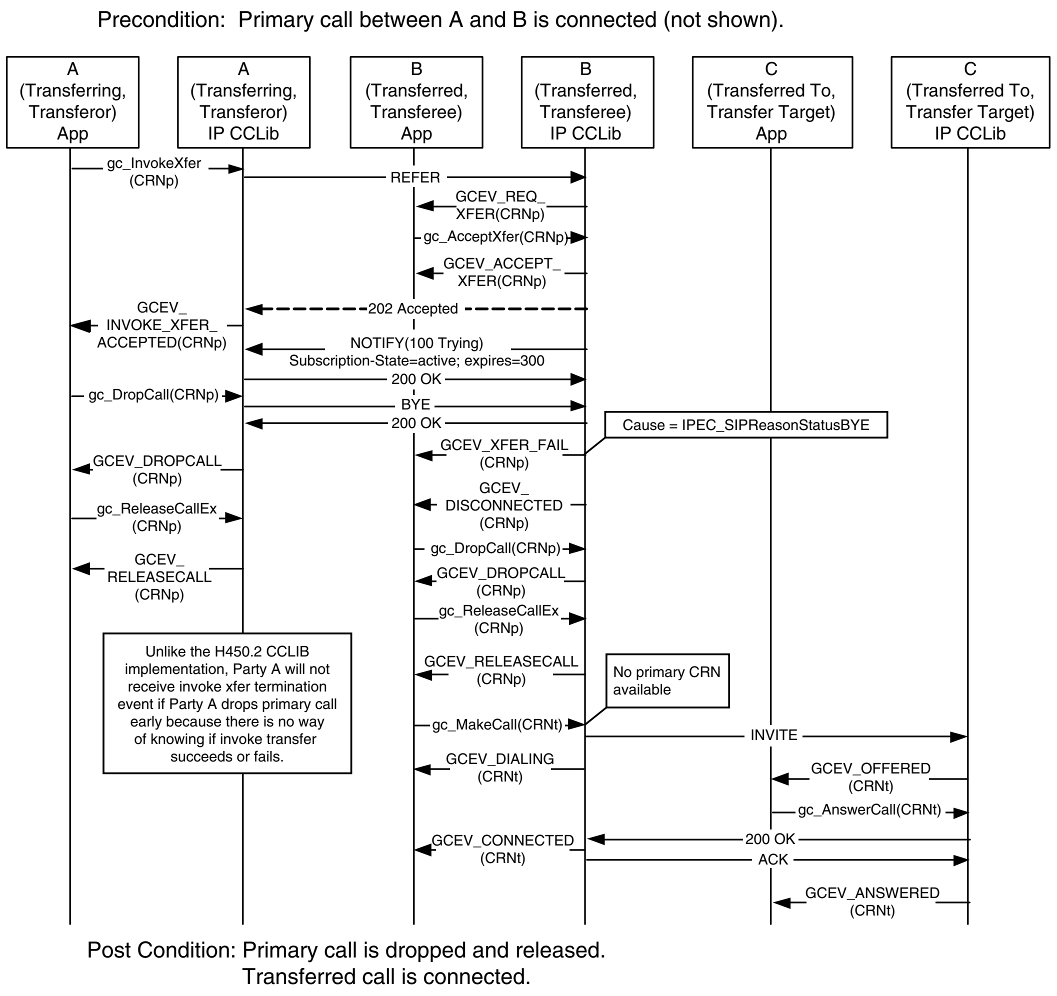
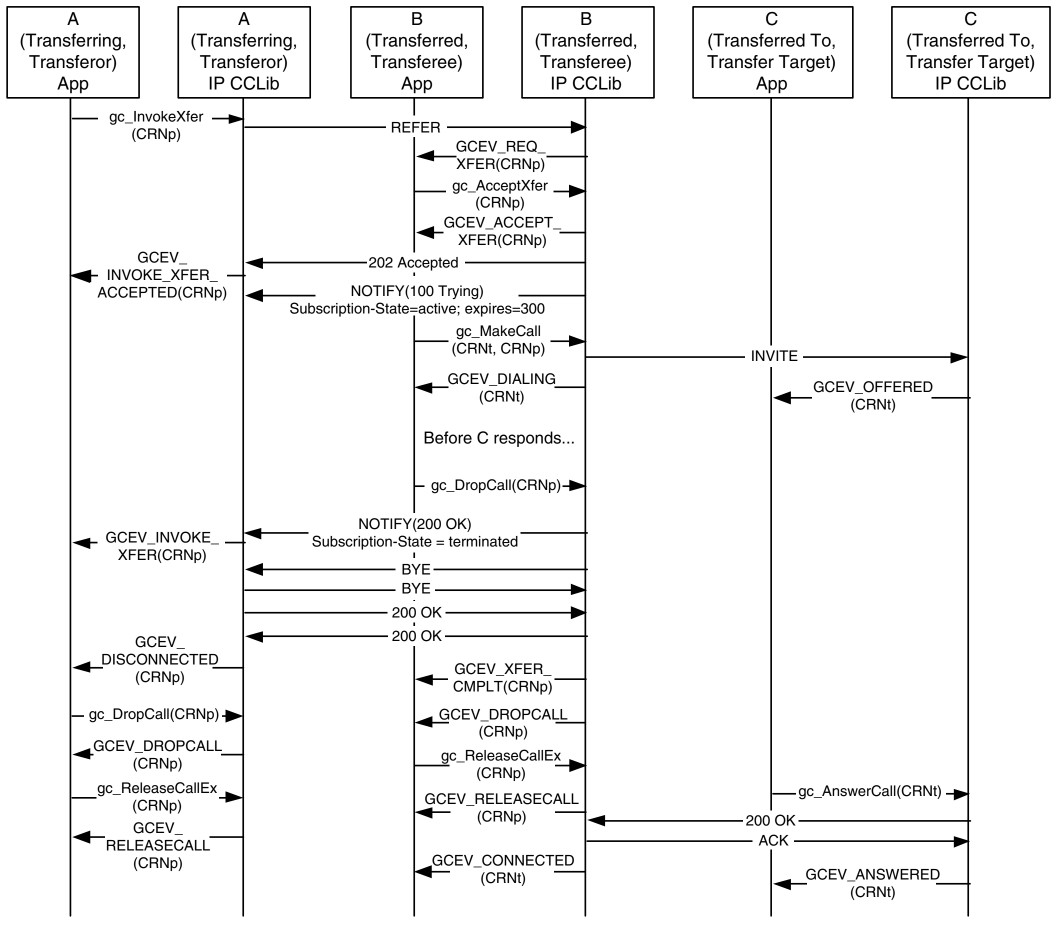

Precondition: Primary call between A and B is connected (not shown).
A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_REQ_XFER(CRNp)
gc_AcceptXfer(CRNp)
GCEV_ACCEPT_XFER(CRNp)
202 Accepted
GCEV_INVOKE_XFER_ACCEPTED(CRNp)
NOTIFY(100 Trying) Subscription-State=active; expires=300
200 OK
gc_DropCall(CRNp)
BYE
200 OK
Cause = IPEC_SIPReasonStatusBYE
GCEV_XFER_FAIL (CRNp)
GCEV_DROPCALL (CRNp)
GCEV_DISCONNECTED (CRNp)
gc_ReleaseCallEx (CRNp)
gc_DropCall(CRNp)
GCEV_RELEASECALL (CRNp)
GCEV_DROPCALL (CRNp)
gc_ReleaseCallEx (CRNp)
Unlike the H450.2 CCLIB implementation, Party A will not receive invoke xfer termination event if Party A drops primary call early because there is no way of knowing if invoke transfer succeeds or fails.
GCEV_RELEASECALL (CRNp)
No primary CRN available
gc_MakeCall(CRNt)
INVITE
GCEV_DIALING (CRNt)
GCEV_OFFERED (CRNt)
gc_AnswerCall(CRNt)
GCEV_CONNECTED (CRNt)
200 OK
ACK
GCEV_ANSWERED (CRNt)
Post Condition: Primary call is dropped and released. Transferred call is connected.

Figure 29 illustrates a scenario in which party B drops the primary call with party A after accepting the transfer request and issuing INVITE to party C, but before receiving any response from party C. In this scenario, party B does notify party A, but this notification only signifies that party B has acted on the transfer request and not that the transfer has actually completed.

Figure 29. Successful SIP Unattended Call Transfer, Party B Clears Primary Call prior to Transfer Completion

Pre condition: Primary call between A and B is connected (not shown).



Post condition: Primary call is dropped and released.
Transferred call is connected.

3.3.4 Endpoint Behavior in Attended SIP Transfers

The assumed preconditions for attended SIP call transfer (commonly referred to as “supervised call transfer” in other technologies and protocols) are:

- The transferring endpoint (party A, or Transferor in SIP terminology) and the transferred endpoint (party B, or Transferee in SIP terms) are participating in an active call, known as the primary call. From the perspective of the Global Call API, party A and party B are both in the GCST_CONNECTED state.
- The Transferor and the transferred-to party (party C or the Transfer Target in SIP terminology) are participating in an active call, known as the secondary or consultation call. From the perspective of the Global Call call control library, party A and party C are both in the GCST_CONNECTED state.

Completion of a successful attended transfer results in the eventual termination of the primary and secondary calls, and the creation of the transferred call between party B and the party C.

3.3.4.1 Transferor or Transferring Endpoint (Party A)

SIP does not support or require a transfer initiation process to obtain the rerouting number as in H.323/H.450.2 supervised transfer. To be consistent with the generic Global Call supervised transfer scenario, the party A application in a SIP attended transfer can call **gc_InitXfer( )**, but no request / response messages will be exchanged between party A and party C as a result. Following this function call, party A always receives a GCEV_INIT_XFER completion event with a dummy rerouting address. To alert party C of incoming transfer process, party A can only notify party C by application data or human interaction outside of SIP protocol.

Just as in the case of unattended transfers, an attended transfer is actually initiated when the Transferor calls the **gc_InvokeXfer( )** function. The difference between unattended and attended transfer usage is the inclusion of the CRN of the secondary (consultation) call as a parameter in the function call. When the Transferor calls **gc_InvokeXfer( )** with two CRN values, a REFER message with a replace parameter in the Refer-To header is sent to the Transferee (party B).

From this point onward, the behavior at this endpoint is similar to that of a unattended transfer, except that the application must also drop the secondary/consultation call at transfer completion. Unlike H.450.2, Global Call will not disconnect the secondary/consultation call once the transferred call is answered at party C.

Because SIP does not require any pre-invocation setup for attended call transfers, the Transferor (party A) can actually treat either of the two active calls as the primary call, and can send the REFER to either of the remote endpoints. This fact provides a recovery mechanism in case one of the remote endpoints does not support the REFER method, as illustrated in the scenarios in the following section.

Protecting and Exposing the Transfer Target

The ability to direct the REFER to either of the parties to which the Transferor provides the opportunity to protect the Transfer Target.

To protect the Transfer Target, the Transferor simply reverses the primary and secondary call CRNs when calling **gc_InvokeXfer( )** to reverse the roles of the two remote parties. The original Transfer Target will now send INVITE to the original Transferee, so that the Transferee is effectively "called back" by the Transfer Target. This has the advantage of hiding information about the original Transfer Target from the original transferee, although the Transferee's experience in this scenario will be different that in current systems PBX or Centrex systems.

To expose the Transfer Target and provide an experience similar to current PBX and Centrex systems, the Transferor uses the secondary call to alert the Transfer Target to the impending transfer, but then disconnects the secondary call and completes the transfer as an unattended transfer. In this case, the **gc_InvokeXfer( )** call only includes the CRN of the primary call.

3.3.4.2 Transferee or Transferred Endpoint (Party B)

This endpoint behaves in the same manner as in unattended transfer with one exception: the INVITE that is sent from Party B to Party C for the transferred call contains a Replaces header that is obtained from the replace parameter in the Refer-To header of the REFER from Party A.

Note that the primary call cannot be disconnected prior to making the transferred call during an attended transfer because the primary call database contains the Replaces information that is required to establish the transferred call.

3.3.4.3 Transfer Target or Transferred-To Endpoint (Party C)

This endpoint behaves in much the same manner as in an unattended transfer with one additional feature and one additional responsibility.

If the Replaces header exists in the incoming INVITE, Global Call automatically matches the Replaces value with any existing connected call on Party C. If a matching call (the secondary or consultation call) is found, that call's CRNs is passed to the application as a GCPARM_SECONDARYCALL_CRN parameter in the GC_PARM_BLK that is attached to the GCEV_OFFERED event.

The party C application must also drop the secondary/consultation call when the transfer completes. Unlike H.450.2 call transfer, Global Call does not automatically disconnect the secondary call once the transferred call answered at the party C.

3.3.5 Successful SIP Attended Call Transfer Scenarios

This section describes the basic scenario for successful SIP call transfer and the scenarios for recovery from two conditions that can block transfer completion. The scenarios include:

- Successful SIP Attended Call Transfer
- Attended Transfer when REFER is Not Globally Supported
- Attended Transfer When Contact URI is Not Globally Routable

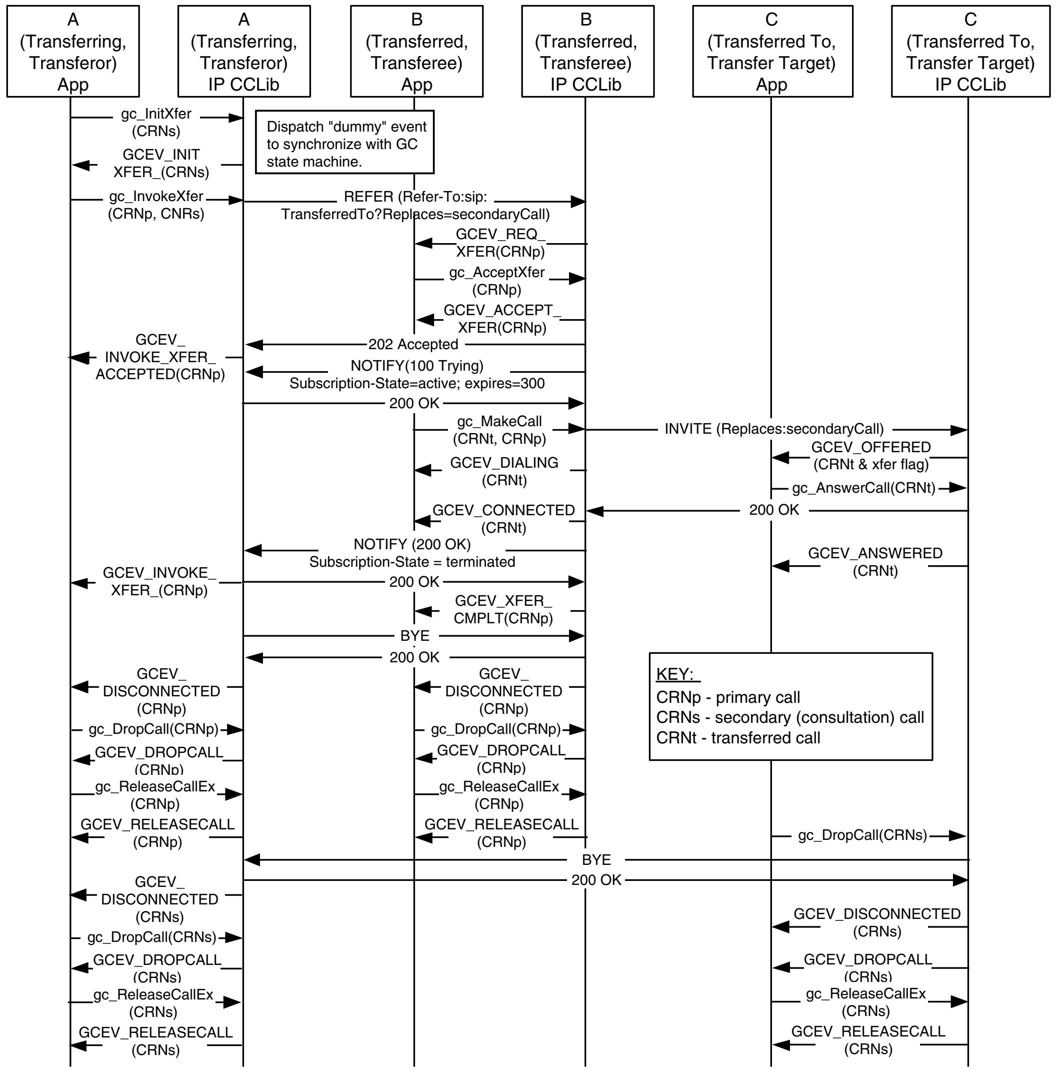
The scenarios all illustrate the optional GCEV_INVOKE_XFER_ACCEPTED event, which is disabled by default. The Transferor application can enable and disable this event at any time after the line device is opened using the **gc_SetConfigData( )** function.

3.3.5.1 Successful SIP Attended Call Transfer

Figure 30 illustrates the basic scenario for successful SIP attended call transfer. The scenario illustrates the use of a **gc_InitXfer( )** function call, which is not required in SIP. The GCEV_INIT_XFER completion event in this case contains a dummy rerouting address.

Figure 30. Successful SIP Attended Call Transfer

Pre condition: Primary call between A and B is connected (not shown).
Secondary (consultation) call between A and C is connected (not shown).

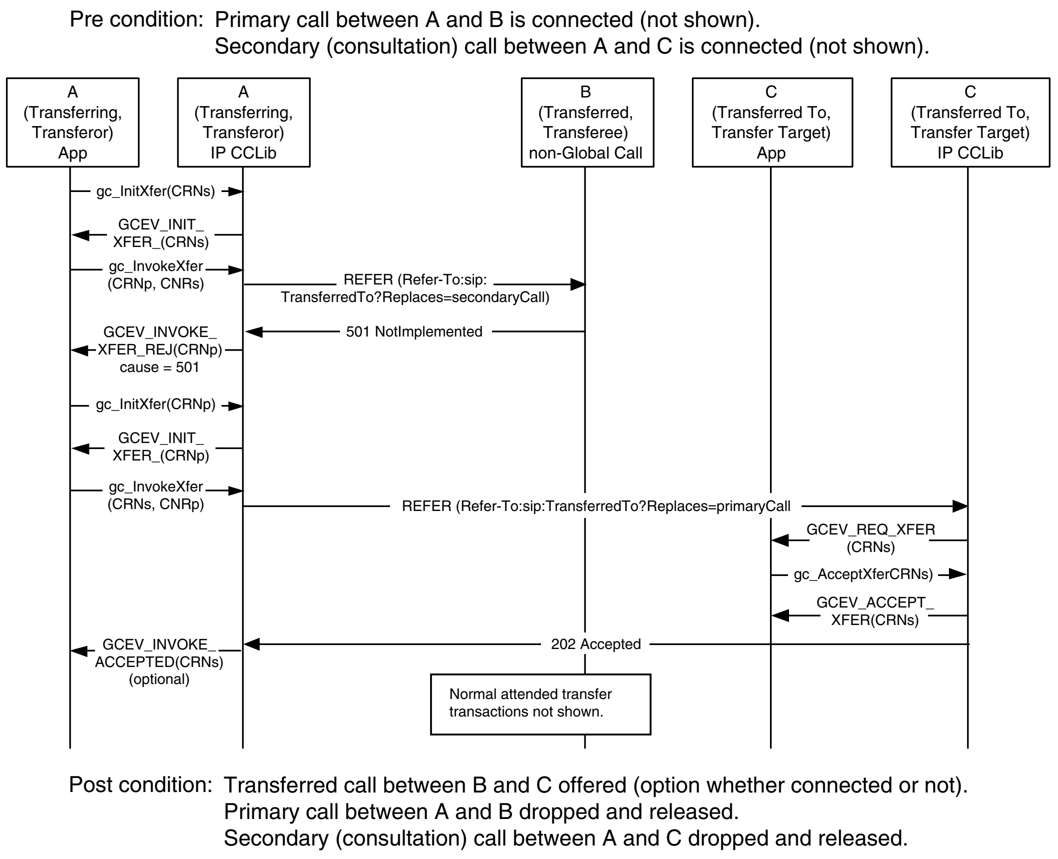


Post condition: Transferred call between B and C offered (option whether connected or not).
Primary call between A and B dropped and released.
Secondary (consultation) call between A and C dropped and released.

3.3.5.2 Attended Transfer when REFER is Not Globally Supported

If protecting or exposing the Transfer Target is not a concern, it is possible to complete an attended transfer when only the Transferor and one other party support REFER. Note that a 405 Method Not Allowed might be returned instead of the 501 Not Implemented response.

Figure 31. SIP Attended Call Transfer, Recovery from REFER Unsupported



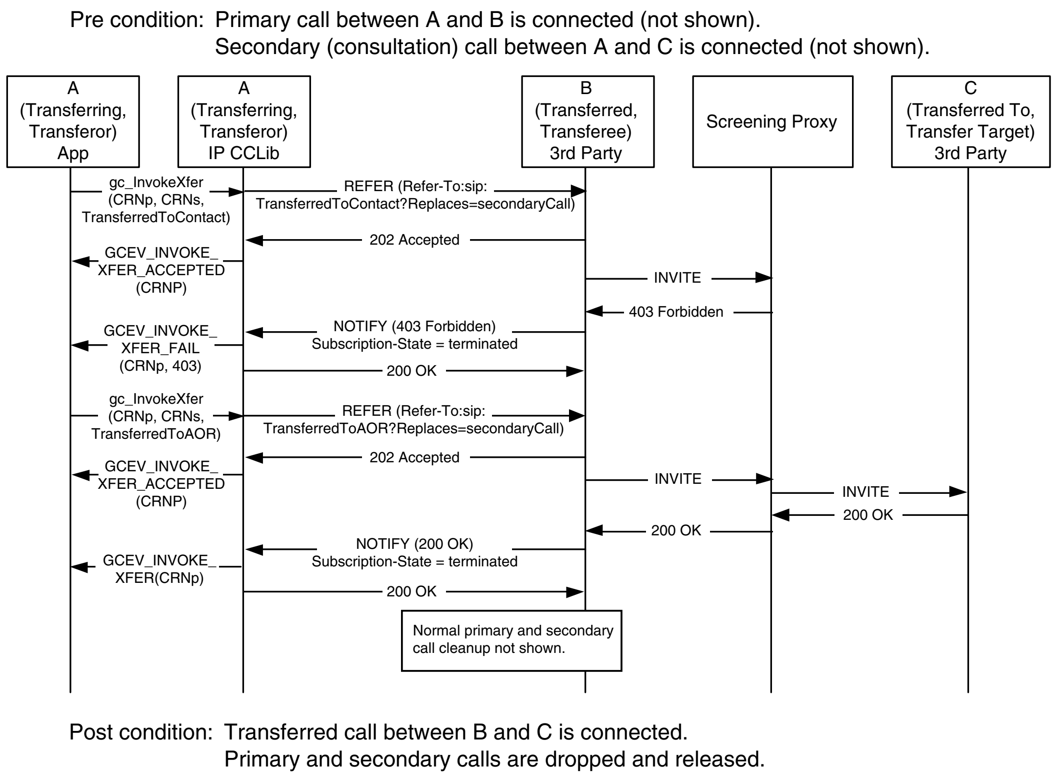
Pre condition: Primary call between A and B is connected (not shown).
Secondary (consultation) call between A and C is connected (not shown).
A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) non-Global Call
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InitXfer(CRNs)
GCEV_INIT_ XFER_(CRNs)
gc_InvokeXfer (CRNp, CNRs)
REFER (Refer-To:sip: TransferredTo?Replaces=secondaryCall)
501 NotImplemented
GCEV_INVOKE_ XFER_REJ(CRNp) cause = 501
gc_InitXfer(CRNp)
GCEV_INIT_ XFER_(CRNp)
gc_InvokeXfer (CRNs, CNRp)
REFER (Refer-To:sip:TransferredTo?Replaces=primaryCall
GCEV_REQ_XFER (CRNs)
gc_AcceptXferCRNs)
GCEV_ACCEPT_ XFER(CRNs)
202 Accepted
GCEV_INVOKE_ ACCEPTED(CRNs) (optional)
Normal attended transfer transactions not shown.
Post condition: Transferred call between B and C offered (option whether connected or not).
Primary call between A and B dropped and released.
Secondary (consultation) call between A and C dropped and released.

3.3.5.3 Attended Transfer When Contact URI is Not Globally Routable

It is a requirement of RFC3261 that a Contact URI be globally routable even outside the dialog. However, due to RFC2543 User Agents and some architectures (NAT/firewall traversal, screening proxies, ALGs, etc.), this will not always be the case. As a result, the methods of attended transfer shown in Figure 30 and Figure 31 may fail since they use the Contact URI in the Refer-To header field. Figure 32 shows such a scenario involving a Screening Proxy in which the transfer initially fails but succeeds on a second try. The failure response (403 Forbidden, 404 Not Found, or a timeout after no response) is communicated back to the Transferor. Since this may be caused by routing problems with the Contact URI, the Transferor retries the REFER, this time with Refer-To containing the Address of Record (AOR) of the Target (the same URI the Transferor used to reach the Transfer Target). However, the use of the AOR URI may result in routing features being activated such as forking or sequential searching which may result in the triggered INVITE reaching the wrong User Agent. To prevent an incorrect UA answering the INVITE, a Require: replaces header field is included in the Refer-To. This ensures that only the UA which matches the Replaces dialog will answer the INVITE, since any incorrect UA which supports Replaces will reply with a 481 and a UA which does not support Replaces will reply with a 420.

Note that there is still no guarantee that the correct endpoint will be reached, and the result of this second REFER may also be a failure. In that case, the Transferor could fall back to unattended transfer or give up on the transfer entirely. Since two REFERs are sent within the dialog, creating two distinct subscriptions, the Transferee uses the 'id' parameter in the Event header field to distinguish notifications for the two subscriptions.

Figure 32. SIP Attended Call Transfer, Recovery from URI Not Routable


Pre condition: Primary call between A and B is connected (not shown).
Secondary (consultation) call between A and C is connected (not shown).
A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) 3rd Party
Screening Proxy
C (Transferred To, Transfer Target) 3rd Party
gc_InvokeXfer (CRNp, CRNs, TransferredToContact)
REFER (Refer-To:sip: TransferredToContact?Replaces=secondaryCall)
202 Accepted
GCEV_INVOKE_ XFER_ACCEPTED (CRNP)
INVITE
403 Forbidden
NOTIFY (403 Forbidden) Subscription-State = terminated
GCEV_INVOKE_ XFER_FAIL (CRNp, 403)
200 OK
gc_InvokeXfer (CRNp, CRNs, TransferredToAOR)
REFER (Refer-To:sip: TransferredToAOR?Replaces=secondaryCall)
202 Accepted
GCEV_INVOKE_ XFER_ACCEPTED (CRNP)
INVITE
INVITE
200 OK
200 OK
NOTIFY (200 OK) Subscription-State = terminated
GCEV_INVOKE_ XFER(CRNp)
200 OK
Normal primary and secondary call cleanup not shown.
Post condition: Transferred call between B and C is connected.
Primary and secondary calls are dropped and released.

3.3.6 Unsuccessful Call Transfer Scenarios

All of the scenarios in this section apply to both unattended (blind) transfer and attended (supervised) SIP call transfers. The **gc_InitXfer( )** function call and the corresponding GCEV_INIT_XFER termination event are “dummy” operations that are only used to synchronize the Global Call state machine and can safely be ignored in this context.

Transfer failures can be caused by any of transfer endpoints as shown in the scenarios. In all cases, the transferring endpoint (Transferor or party A) is notified by a GCEV_INVOKE_XFER_REJ event or a GCEV_INVOKE_XFER_FAIL event. No NOTIFY will be sent from party B to party A if REFER is not accepted by a 202 Accepted response from party B. The primary call and secondary call, if any, remain in the connected state after any transfer failure.

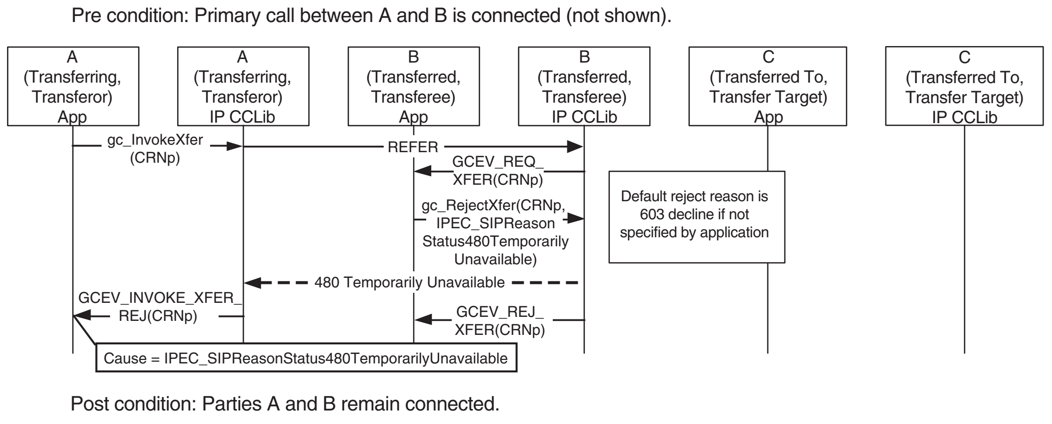
The most common transfer failure scenarios are described in the following topics:

- Party B Rejects Call Transfer
- No Response From Party B
- No Initial NOTIFY after REFER Accepted
- REFER Subscription Expires
- No Response From Party C
- Party B Drops Transferred Call Early
- Party C is Busy When Transfer Attempted

3.3.6.1 Party B Rejects Call Transfer

Figure 33, illustrates a scenario in which the application at the transferred endpoint (Transferee or party B) calls **gc_RejectXfer( )** to signal the Transferor (party A) that it cannot participate in a transfer. The application may specify any valid SIP rejection reason, such as the 480 Temporarily Unavailable shown in the figure; if no reason is specified, the default reason sent is 603 Decline. As a result of the rejection, the GCEV_INVOKE_XFER_REJ termination event is received at the Transferor application (party A). The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both party A and party B.

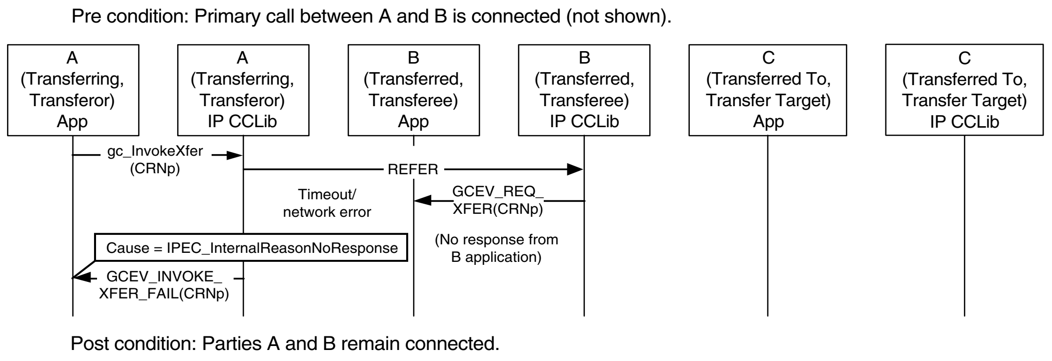
Figure 33. SIP Call Transfer Failure - Party B Rejects Call Transfer


Pre condition: Primary call between A and B is connected (not shown).
A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_REQ_XFER(CRNp)
gc_RejectXfer(CRNp, IPEC_SIPReason Status480Temporarily Unavailable)
Default reject reason is 603 decline if not specified by application
480 Temporarily Unavailable
GCEV_INVOKE_XFER_REJ(CRNp)
GCEV_REJ_XFER(CRNp)
Cause = IPEC_SIPReasonStatus480TemporarilyUnavailable
Post condition: Parties A and B remain connected.

3.3.6.2 No Response From Party B

Figure 34 illustrates a scenario in which the Transferee (party B) does not respond to the REFER, causing the T3 timer at the party A (configured as 20 seconds) to expire. After the timeout, the Transferor application receives the GCEV_INVOKE_XFER_FAIL termination event. The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both party A and party B.

Figure 34. SIP Call Transfer Failure - No Response from Party B

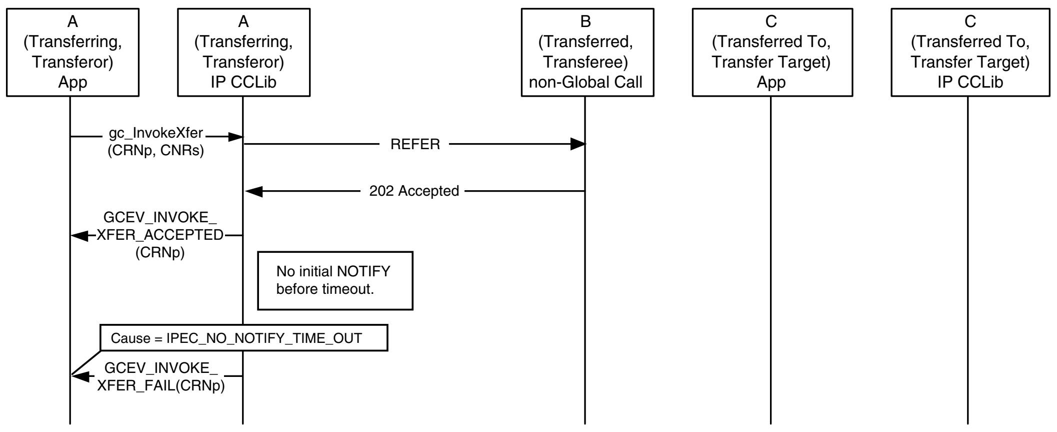

Pre condition: Primary call between A and B is connected (not shown).
A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
Timeout/ network error
GCEV_REQ_XFER(CRNp)
(No response from B application)
Cause = IPEC_InternalReasonNoResponse
GCEV_INVOKE_XFER_FAIL(CRNp)
Post condition: Parties A and B remain connected.

3.3.6.3 No Initial NOTIFY after REFER Accepted

Figure 35 illustrates a scenario in which the Transferee (party B) does not send a NOTIFY after it accepts the REFER, causing the timer at party A to expire. The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both party A and party B.

Figure 35. SIP Call Transfer Failure - No Initial NOTIFY After REFER is Accepted

Pre condition: Primary call between A and B is connected (not shown).

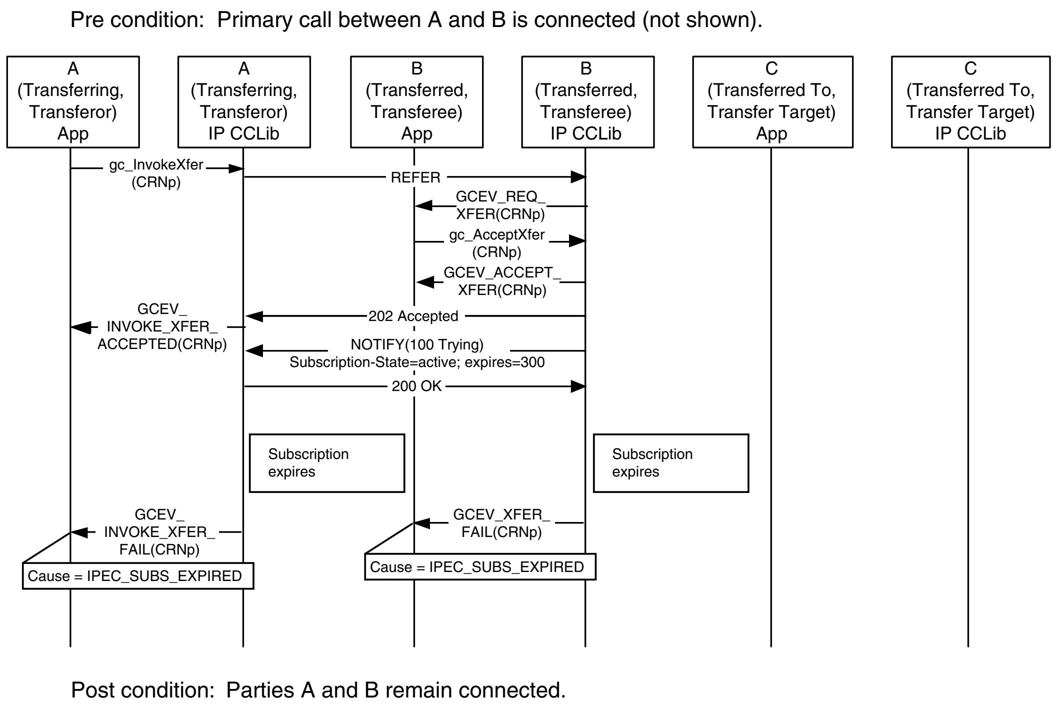

A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) non-Global Call
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp, CNRs)
REFER
202 Accepted
GCEV_INVOKE_ XFER_ACCEPTED (CRNp)
No initial NOTIFY before timeout.
Cause = IPEC_NO_NOTIFY_TIME_OUT
GCEV_INVOKE_ XFER_FAIL(CRNp)

Post condition: Parties A and B remain connected.

3.3.6.4 REFER Subscription Expires

Figure 36 illustrates a scenario in which the REFER subscription expires, causing both party A and party B to time out. After the timeout, the Transferee application receives a GCEV_XFER_FAIL termination event and the Transferor application receives a GCEV_INVOKE_XFER_FAIL termination event. The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both party A and party B.

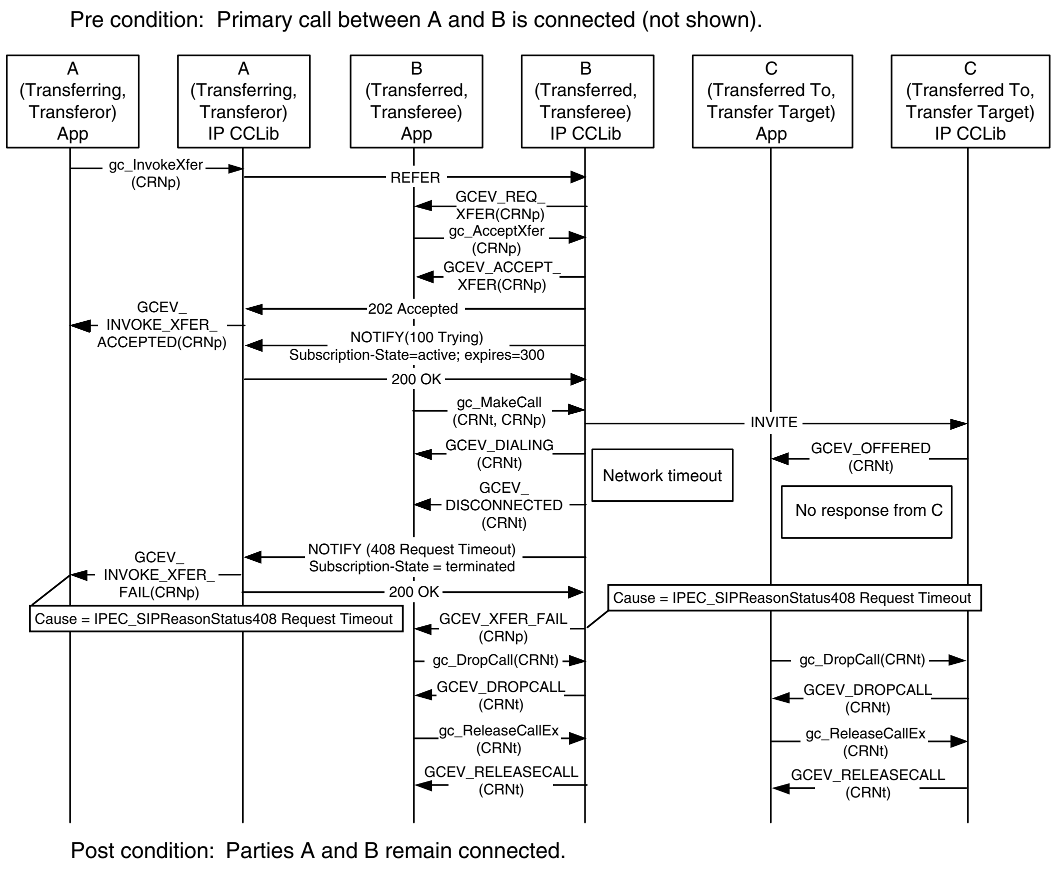
Figure 36. SIP Call Transfer Failure - REFER Subscription Expires


Pre condition: Primary call between A and B is connected (not shown).
A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_REQ_ XFER(CRNp)
gc_AcceptXfer (CRNp)
GCEV_ACCEPT_ XFER(CRNp)
GCEV_ INVOKE_XFER_ ACCEPTED(CRNp)
202 Accepted
NOTIFY(100 Trying) Subscription-State=active; expires=300
200 OK
Subscription expires
Subscription expires
GCEV_ INVOKE_XFER_ FAIL(CRNp)
Cause = IPEC_SUBS_EXPIRED
GCEV_XFER_ FAIL(CRNp)
Cause = IPEC_SUBS_EXPIRED
Post condition: Parties A and B remain connected.

3.3.6.5 No Response From Party C

Figure 37 illustrates a scenario in which the Transfer Target (party C) does not respond to the incoming call from the Transferee (party B) which causes the T4 timer at party B (configured as 20 seconds) to expire. As a result, the Transferee application (party B) receives the GCEV_DISCONNECTED event for the transferred call timeout. The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

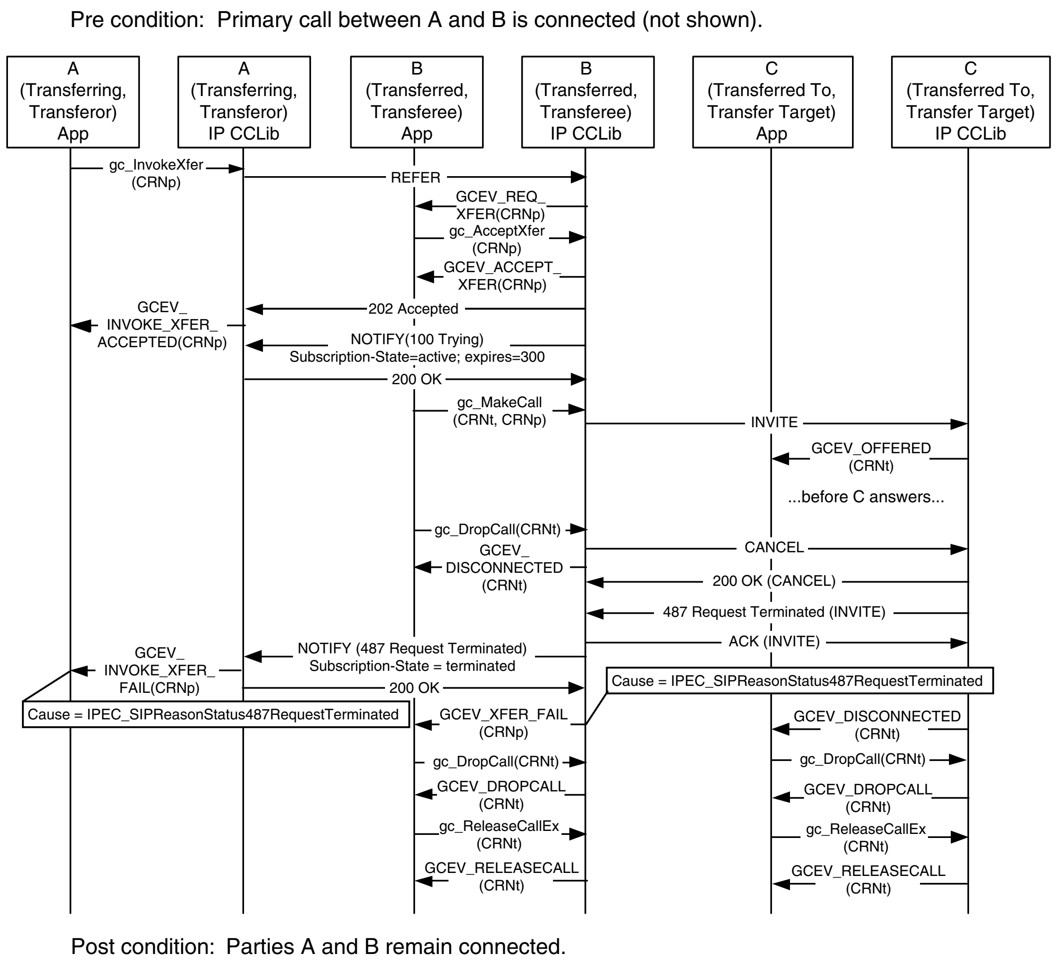
Figure 37. SIP Call Transfer Failure - No Response from Party C


Pre condition: Primary call between A and B is connected (not shown).
A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_REQ_ XFER(CRNp)
gc_AcceptXfer (CRNp)
GCEV_ACCEPT_ XFER(CRNp)
202 Accepted
GCEV_ INVOKE_XFER_ ACCEPTED(CRNp)
NOTIFY(100 Trying) Subscription-State=active; expires=300
200 OK
gc_MakeCall (CRNt, CRNp)
INVITE
GCEV_DIALING (CRNt)
GCEV_OFFERED (CRNt)
Network timeout
No response from C
GCEV_ DISCONNECTED (CRNt)
NOTIFY (408 Request Timeout) Subscription-State = terminated
GCEV_ INVOKE_XFER_ FAIL(CRNp)
200 OK
Cause = IPEC_SIPReasonStatus408 Request Timeout
Cause = IPEC_SIPReasonStatus408 Request Timeout
GCEV_XFER_FAIL (CRNp)
gc_DropCall(CRNt)
gc_DropCall(CRNt)
GCEV_DROPCALL (CRNt)
GCEV_DROPCALL (CRNt)
gc_ReleaseCallEx (CRNt)
gc_ReleaseCallEx (CRNt)
GCEV_RELEASECALL (CRNt)
GCEV_RELEASECALL (CRNt)
Post condition: Parties A and B remain connected.

3.3.6.6 Party B Drops Transferred Call Early

Figure 38 illustrates a scenario in which the Transferee (party B) drops the transferred call before receiving a response to the INVITE it sent to party C. As a result, the GCEV_INVOKE_XFER_FAIL termination event is received at the Transferor (party A) and the GCEV_XFER_FAIL termination event is received a the Transferee (party B). The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both A and B.

Figure 38. SIP Call Transfer Failure - Party B Drops Transferred Call Early



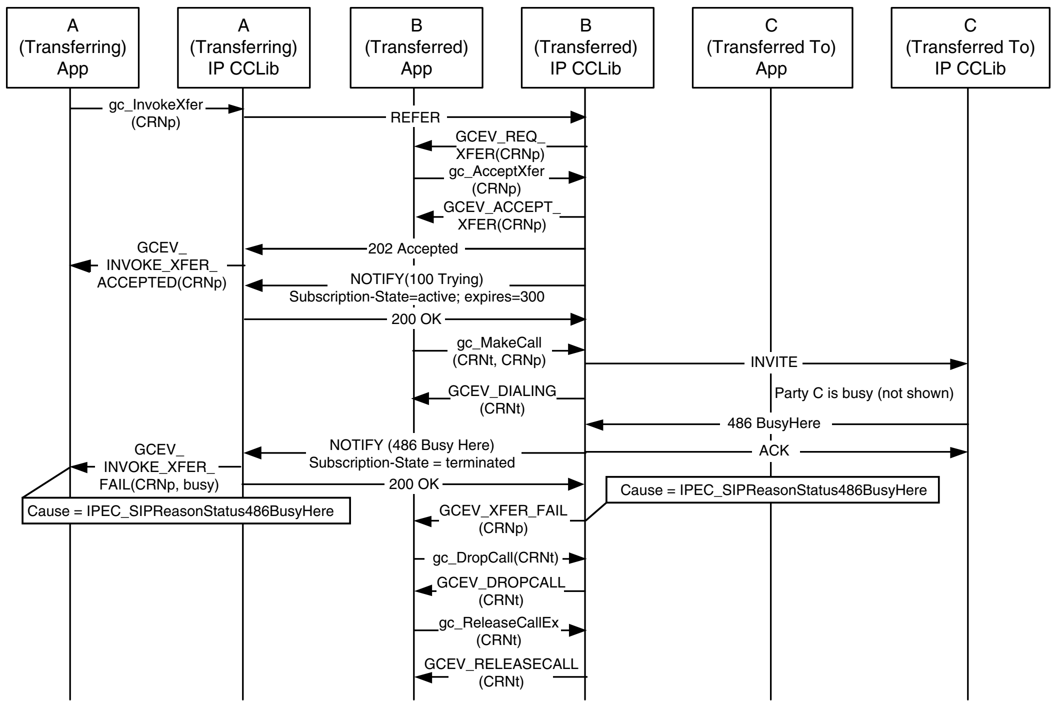
Pre condition: Primary call between A and B is connected (not shown).
A (Transferring, Transferor) App
A (Transferring, Transferor) IP CCLib
B (Transferred, Transferee) App
B (Transferred, Transferee) IP CCLib
C (Transferred To, Transfer Target) App
C (Transferred To, Transfer Target) IP CCLib
gc_InvokeXfer (CRNp)
REFER
GCEV_REQ_XFER(CRNp)
gc_AcceptXfer (CRNp)
GCEV_ACCEPT_XFER(CRNp)
202 Accepted
GCEV_INVOKE_XFER_ACCEPTED(CRNp)
NOTIFY(100 Trying) Subscription-State=active; expires=300
200 OK
gc_MakeCall (CRNt, CRNp)
INVITE
GCEV_OFFERED (CRNt)
...before C answers...
gc_DropCall(CRNt)
CANCEL
GCEV_DISCONNECTED (CRNt)
200 OK (CANCEL)
487 Request Terminated (INVITE)
ACK (INVITE)
GCEV_INVOKE_XFER_FAIL(CRNp)
NOTIFY (487 Request Terminated) Subscription-State = terminated
200 OK
Cause = IPEC_SIPReasonStatus487RequestTerminated
Cause = IPEC_SIPReasonStatus487RequestTerminated
GCEV_XFER_FAIL (CRNp)
GCEV_DISCONNECTED (CRNt)
gc_DropCall(CRNt)
gc_DropCall(CRNt)
GCEV_DROPCALL (CRNt)
GCEV_DROPCALL (CRNt)
gc_ReleaseCallEx (CRNt)
gc_ReleaseCallEx (CRNt)
GCEV_RELEASECALL (CRNt)
GCEV_RELEASECALL (CRNt)
Post condition: Parties A and B remain connected.

3.3.6.7 Party C is Busy When Transfer Attempted

Figure 39 illustrates a scenario in which the Transfer Target (party C) is busy at the time the transfer is requested. (This primarily applies to unattended transfers, since the Transferor would be aware that the Transfer Target is busy in an attended transfer.) In this case, the Transferor (party A) receives a GCEV_INVOKE_XFER_FAIL termination event and the Transferee (party B) receives a GCEV_XFER_FAIL termination event. The original primary call is left connected and in the GCST_CONNECTED state from the perspective of both party A and party B.

Figure 39. SIP Call Transfer Failure - Party C is Busy When Transfer Attempted

Pre condition: Primary call between parties A and B is connected (not shown).
Party C has call connected to another party (not shown).



Post condition: Parties A and B remain connected.
Party C also remains connected (to another party not shown).

3.4 T.38 Fax Server Call Scenarios

Dialogic® Global Call API supports T.38 fax server as described in Section 4.38, "T.38 Fax Server", on page 374. The following scenarios demonstrate the T.38 fax server capabilities provided when using IP technology (both H.323 and SIP):

- Sending T.38 Fax in an Established Audio Session
- Receiving T.38 Fax in an Established Audio Session
- Sending T.38 Fax Without an Established Audio Session
- Receiving T.38 Fax Without an Established Audio Session
- Sending a Request to Switch From T.38 Fax to Audio
- Receiving a Request to Switch From T.38 Fax to Audio
- Terminating a Call After a T.38 Fax Session
- Recovering from a Session Switching Failure

Note: In these scenarios, the application must include T.38 Fax capability either when using **gc_MakeCall( )** for an outbound call or when using **gc_CallAck( )**, **gc_AcceptCall( )**, or **gc_AnswerCall( )** for an inbound call.

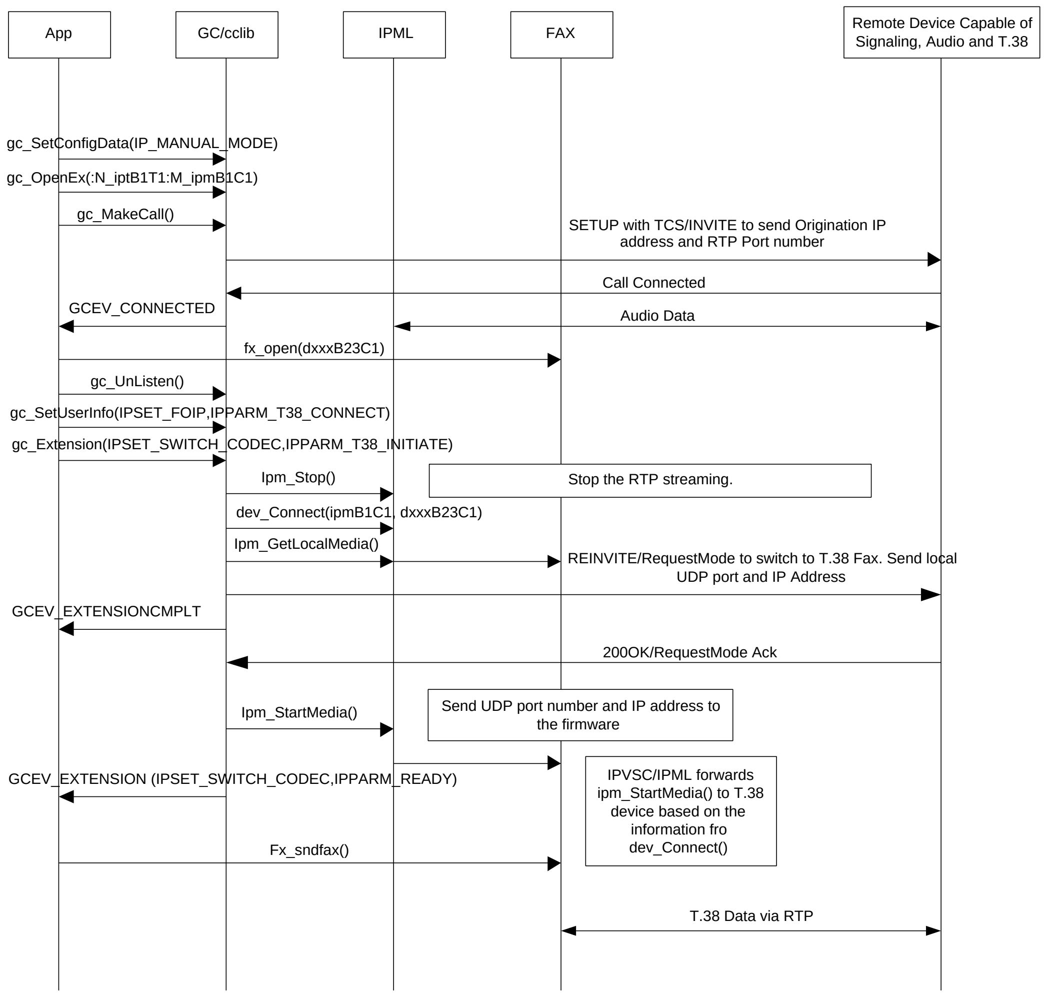
3.4.1 Sending T.38 Fax in an Established Audio Session

In this scenario, the user application uses the Dialogic® Global Call API to open a Media device, configures "Manual" mode of operation and establishes an audio session with the remote device. See Section 4.38.2, "Specifying Manual Operating Mode", on page 376 for more information on manual mode. A T.38 Fax device is then opened and the application switches from an audio session to a T.38 session.

When the application receives notification that the T.38 session is ready, fax information can be sent. Figure 40 shows the scenario diagram.

Note: The application must not use both Dialogic® Global Call API and Dialogic® IP Media Library API functions on the same device. The Dialogic® IP Media Library API calls (ipm_) in Figure 40 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 40. Sending T.38 Fax in an Established Audio Session


App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling, Audio and T.38
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
gc_MakeCall()
SETUP with TCS/INVITE to send Origination IP address and RTP Port number
Call Connected
GCEV_CONNECTED
Audio Data
fx_open(dxxxB23C1)
gc_UnListen()
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_CONNECT)
gc_Extension(IPSET_SWITCH_CODEC,IPPARM_T38_INITIATE)
Ipm_Stop()
Stop the RTP streaming.
dev_Connect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia()
REINVITE/RequestMode to switch to T.38 Fax. Send local UDP port and IP Address
GCEV_EXTENSIONCMPLT
200OK/RequestMode Ack
Ipm_StartMedia()
Send UDP port number and IP address to the firmware
GCEV_EXTENSION (IPSET_SWITCH_CODEC,IPPARM_READY)
IPVSC/IPML forwards ipm_StartMedia() to T.38 device based on the information fro dev_Connect()
Fx_sndfax()
T.38 Data via RTP

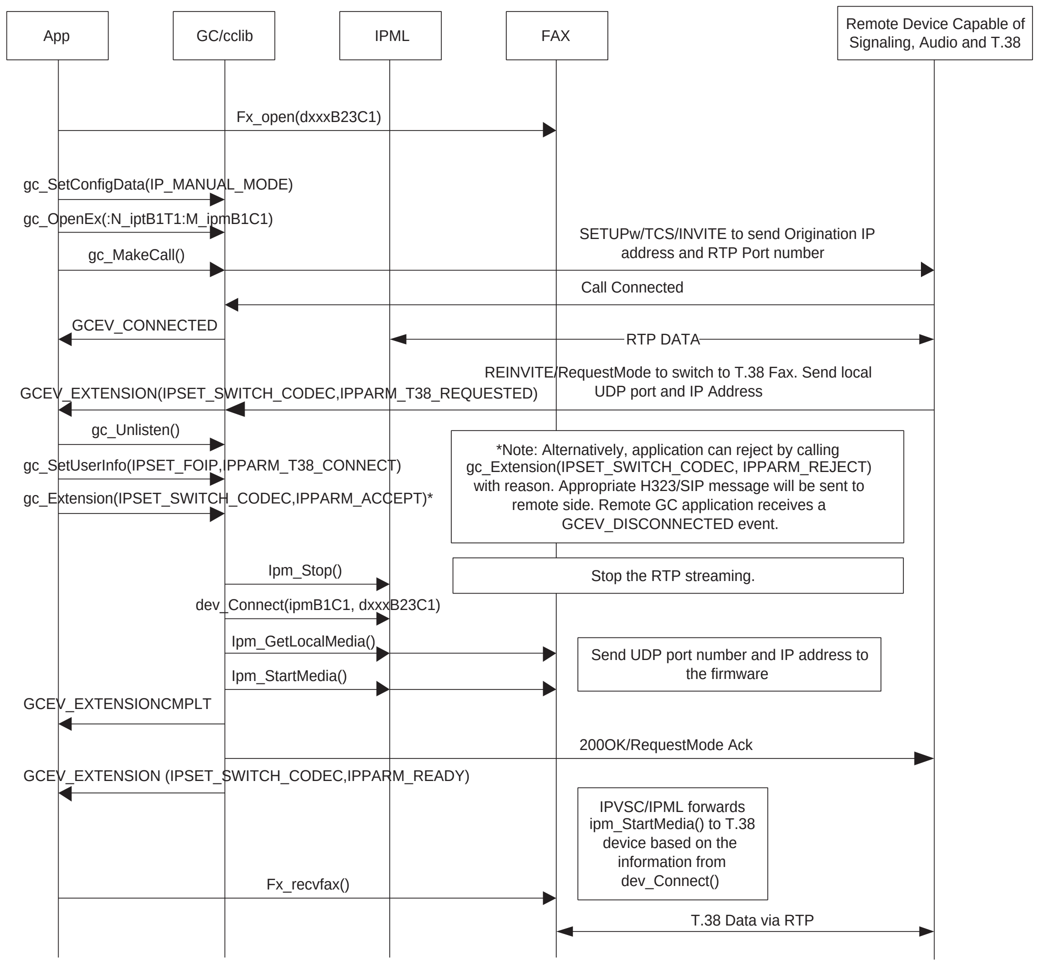
3.4.2 Receiving T.38 Fax in an Established Audio Session

In this scenario, the user application uses the Dialogic® Global Call API to open a Media device, configures "Manual" operating mode and establishes an audio session with the remote device. See Section 4.38.2, "Specifying Manual Operating Mode", on page 376 for more information on manual mode. To prepare to receive fax, the application must also open a T.30 Fax device. During the audio session, the application can be notified of an incoming request to switch from audio to T.38 fax.

The application can choose to accept or reject this request. If the user chooses to accept, Dialogic® Global Call API notifies the application that the T.38 session is ready to receive a fax. Figure 41 shows the scenario diagram.

Note: The application must not use both Dialogic® Global Call API and Dialogic® IP Media Library API functions on the same device. The Dialogic® IP Media Library API calls (ipm_) in Figure 41 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 41. Receiving T.38 Fax in an Established Audio Session


App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling, Audio and T.38
Fx_open(dxxxB23C1)
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
gc_MakeCall()
SETUPw/TCS/INVITE to send Origination IP address and RTP Port number
Call Connected
GCEV_CONNECTED
RTP DATA
REINVITE/RequestMode to switch to T.38 Fax. Send local UDP port and IP Address
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_T38_REQUESTED)
gc_Unlisten()
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_CONNECT)
gc_Extension(IPSET_SWITCH_CODEC,IPPARM_ACCEPT)*
*Note: Alternatively, application can reject by calling gc_Extension(IPSET_SWITCH_CODEC, IPPARM_REJECT) with reason. Appropriate H323/SIP message will be sent to remote side. Remote GC application receives a GCEV_DISCONNECTED event.
Ipm_Stop()
Stop the RTP streaming.
dev_Connect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia()
Send UDP port number and IP address to the firmware
Ipm_StartMedia()
GCEV_EXTENSIONCMPLT
200OK/RequestMode Ack
GCEV_EXTENSION (IPSET_SWITCH_CODEC,IPPARM_READY)
IPVSC/IPML forwards ipm_StartMedia() to T.38 device based on the information from dev_Connect()
Fx_recvfax()
T.38 Data via RTP

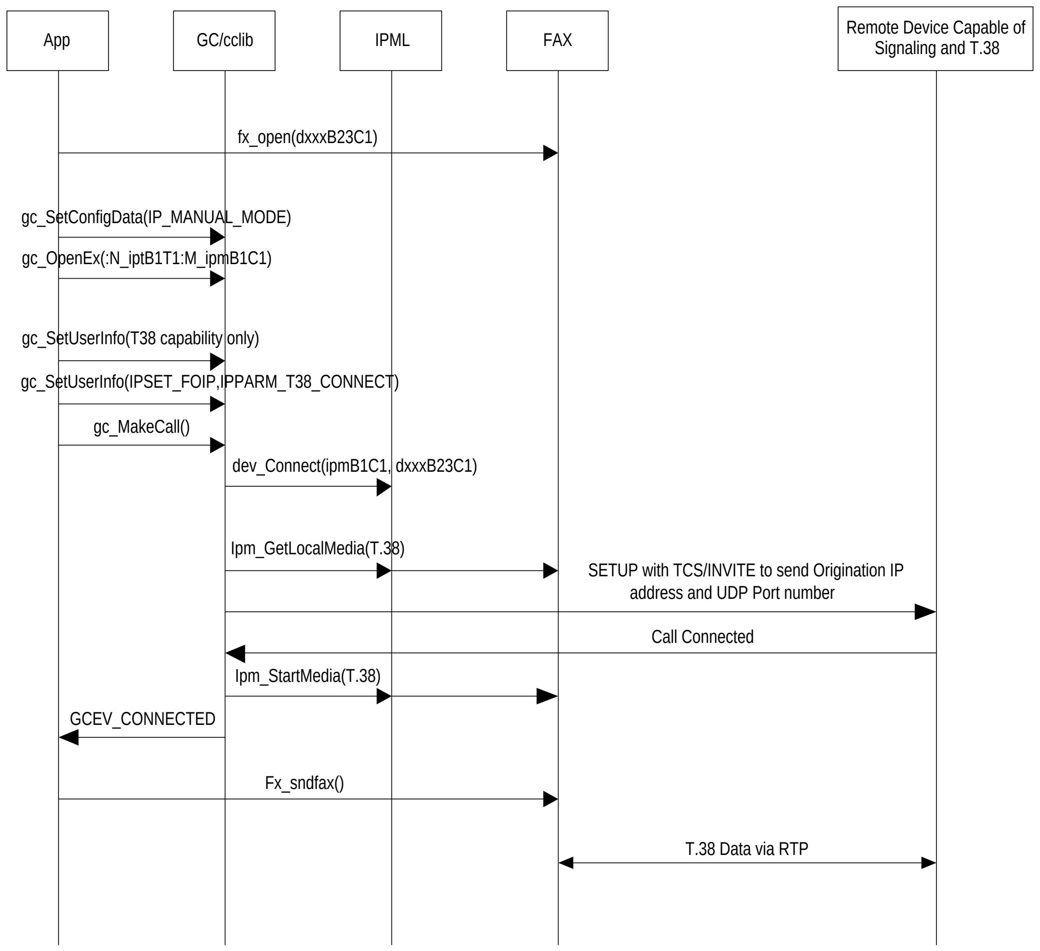
3.4.3 Sending T.38 Fax Without an Established Audio Session

This scenario describes the sending of T.38 Fax in a media session that does not have audio already established. The application first opens a Media device and a T.38 Fax device and configures "Manual" mode of operation. See Section 4.38.2, "Specifying Manual Operating Mode", on page 376 for more information on manual mode. The Dialogic® Global Call API is then used to associate the T.38 Fax device with the IP Media device before making a new T.38 call.

Once the call is connected, the application can send a fax. Figure 42 shows the scenario diagram.

Note: The application must not use both Dialogic® Global Call API and Dialogic® IP Media Library API functions on the same device. The Dialogic® IP Media Library API calls (ipm_) in Figure 42 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 42. Sending T.38 Fax Without an Established Audio Session


App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling and T.38
fx_open(dxxxB23C1)
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
gc_SetUserInfo(T38 capability only)
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_CONNECT)
gc_MakeCall()
dev_Connect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia(T.38)
SETUP with TCS/INVITE to send Origination IP address and UDP Port number
Call Connected
Ipm_StartMedia(T.38)
GCEV_CONNECTED
Fx_sndfax()
T.38 Data via RTP

3.4.4 Receiving T.38 Fax Without an Established Audio Session

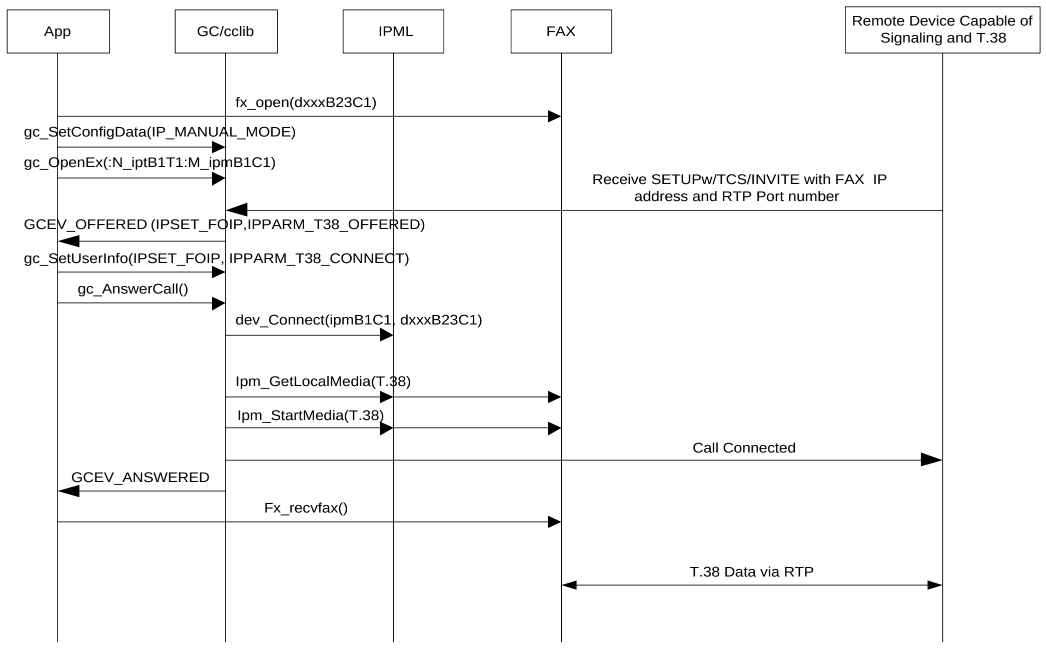
This scenario describes the reception of T.38 Fax in a media session that does not have audio already established. The application first opens a Media device and a T.38 Fax device and configures "Manual" operating mode. See Section 4.38.2, "Specifying Manual Operating Mode", on page 376 for more information on manual mode. When the application receives a T.38 fax request, a GCEV_OFFERED event with T.38 extension information is received.

If the application accepts the call, the T.38 Fax device is associated with the Media device before the T.38 call is answered. Figure 43 shows the scenario diagram.

Notes: ***1.*** The GCEV_OFFERED event with T.38 extension information is only generated if the following requirements are met. For H.323, the incoming message must be a Q.931 Setup message with data terminal capability only. For SIP, the incoming message must be an INVITE message with an SDP that has an image media descriptor only. If this condition is not met, the GCEV_OFFERED event does not include any T.38 extension information. This limitation prevents the T.38 server from receiving the T.38 request in H.323 slow start or in a SIP no SDP INVITE request.

2. The application must not use both Dialogic® Global Call API and Dialogic® IP Media Library API functions on the same device. The Dialogic® IP Media Library API calls (ipm_) in Figure 43 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 43. Receiving T.38 Fax Without an Established Audio Session

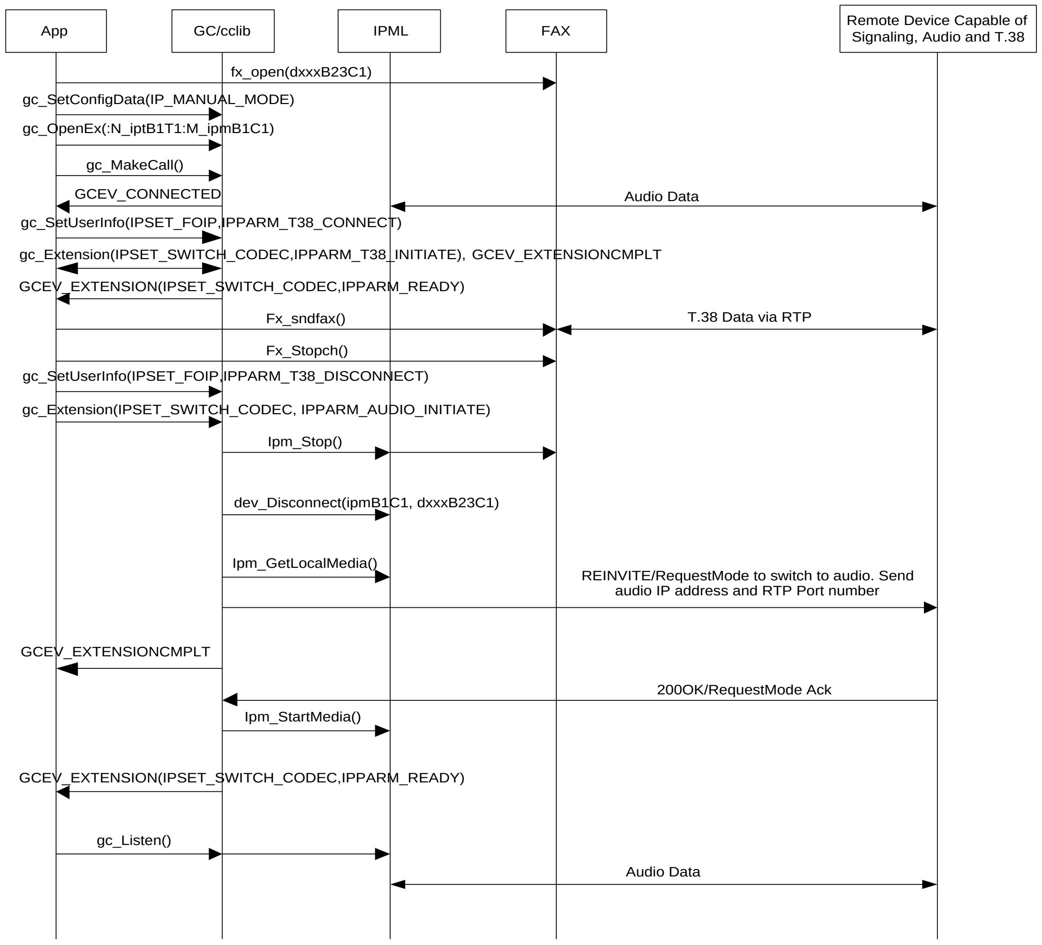

App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling and T.38
fx_open(dxxxB23C1)
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
Receive SETUPw/TCS/INVITE with FAX IP address and RTP Port number
GCEV_OFFERED (IPSET_FOIP,IPPARM_T38_OFFERED)
gc_SetUserInfo(IPSET_FOIP, IPPARM_T38_CONNECT)
gc_AnswerCall()
dev_Connect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia(T.38)
Ipm_StartMedia(T.38)
Call Connected
GCEV_ANSWERED
Fx_recvfax()
T.38 Data via RTP

3.4.5 Sending a Request to Switch From T.38 Fax to Audio

In a fax session, when a fax completes, the application can use the Dialogic® Global Call API to issue a request to switch from a T.38 fax session back to an audio session after disassociating the T.38 Fax device from the Media device. When Dialogic® Global Call API notifies the application that the audio session has been reestablished, the application can once again send and receive audio. Figure 44 shows the scenario diagram.

Note: The application must not use both Dialogic® Global Call API and Dialogic® IP Media Library API functions on the same device. The Dialogic® IP Media Library API calls (ipm_) in Figure 44 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 44. Sending a Request to Switch From T.38 Fax to Audio

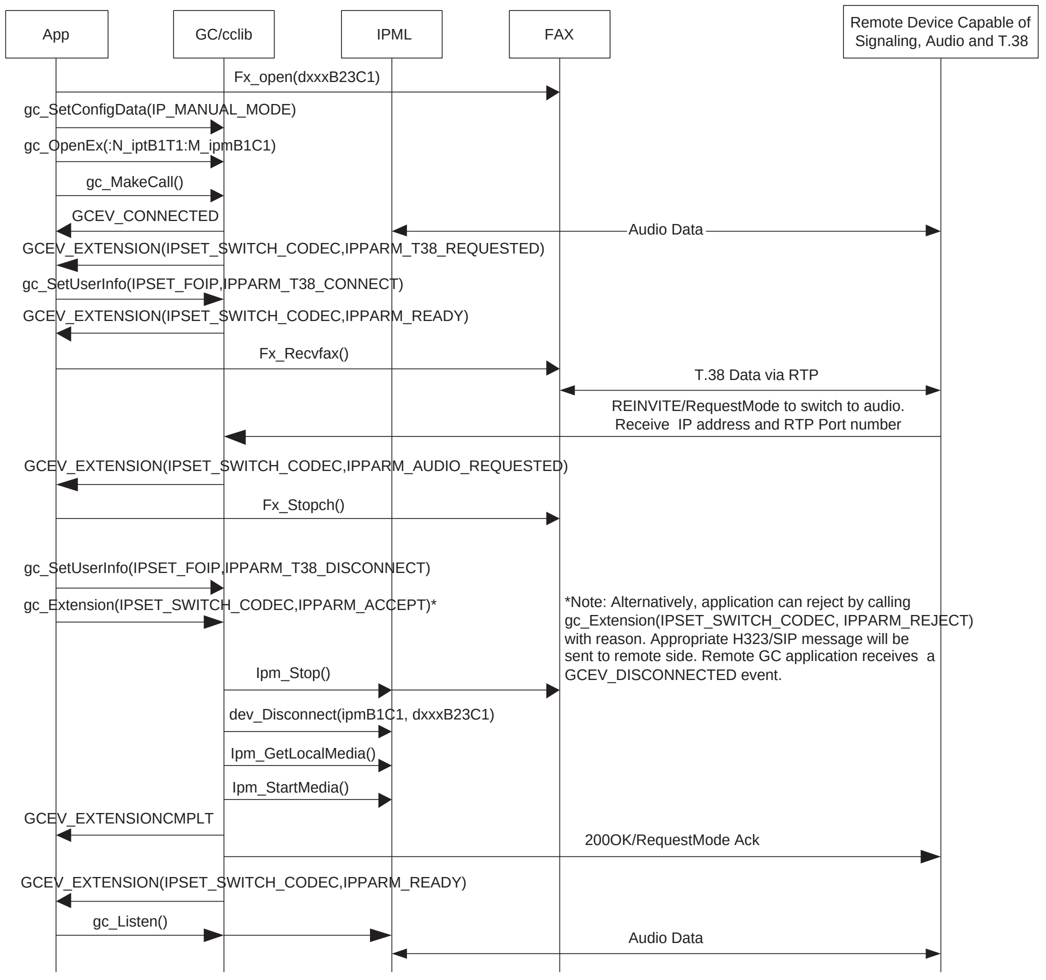

App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling, Audio and T.38
fx_open(dxxxB23C1)
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
gc_MakeCall()
GCEV_CONNECTED
Audio Data
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_CONNECT)
gc_Extension(IPSET_SWITCH_CODEC,IPPARM_T38_INITIATE), GCEV_EXTENSIONCMPLT
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_READY)
Fx_sndfax()
T.38 Data via RTP
Fx_Stopch()
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_DISCONNECT)
gc_Extension(IPSET_SWITCH_CODEC, IPPARM_AUDIO_INITIATE)
Ipm_Stop()
dev_Disconnect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia()
REINVITE/RequestMode to switch to audio. Send audio IP address and RTP Port number
GCEV_EXTENSIONCMPLT
200OK/RequestMode Ack
Ipm_StartMedia()
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_READY)
gc_Listen()
Audio Data

3.4.6 Receiving a Request to Switch From T.38 Fax to Audio

In a fax session, when a fax completes, the application can receive a request to switch from a T.38 fax session back to an audio session. The application can choose to accept the request after disassociating the T.38 Fax device from the Media device. When Dialogic® Global Call API notifies the application that the audio session has been reestablished, the application can once again send and receive audio. Figure 45 shows the scenario diagram.

Note: The application must not use both Dialogic® Global Call API and Dialogic® IP Media Library API functions on the same device. The Dialogic® IP Media Library API calls (ipm_) in Figure 45 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 45. Receiving a Request to Switch From T.38 Fax to Audio


App
GC/cclib
IPML
FAX
Remote Device Capable of Signaling, Audio and T.38
Fx_open(dxxxB23C1)
gc_SetConfigData(IP_MANUAL_MODE)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
gc_MakeCall()
GCEV_CONNECTED
Audio Data
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_T38_REQUESTED)
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_CONNECT)
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_READY)
Fx_Recvfax()
T.38 Data via RTP
REINVITE/RequestMode to switch to audio. Receive IP address and RTP Port number
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_AUDIO_REQUESTED)
Fx_Stopch()
gc_SetUserInfo(IPSET_FOIP,IPPARM_T38_DISCONNECT)
gc_Extension(IPSET_SWITCH_CODEC,IPPARM_ACCEPT)*
*Note: Alternatively, application can reject by calling gc_Extension(IPSET_SWITCH_CODEC, IPPARM_REJECT) with reason. Appropriate H323/SIP message will be sent to remote side. Remote GC application receives a GCEV_DISCONNECTED event.
Ipm_Stop()
dev_Disconnect(ipmB1C1, dxxxB23C1)
Ipm_GetLocalMedia()
Ipm_StartMedia()
GCEV_EXTENSIONCMPLT
200OK/RequestMode Ack
GCEV_EXTENSION(IPSET_SWITCH_CODEC,IPPARM_READY)
gc_Listen()
Audio Data

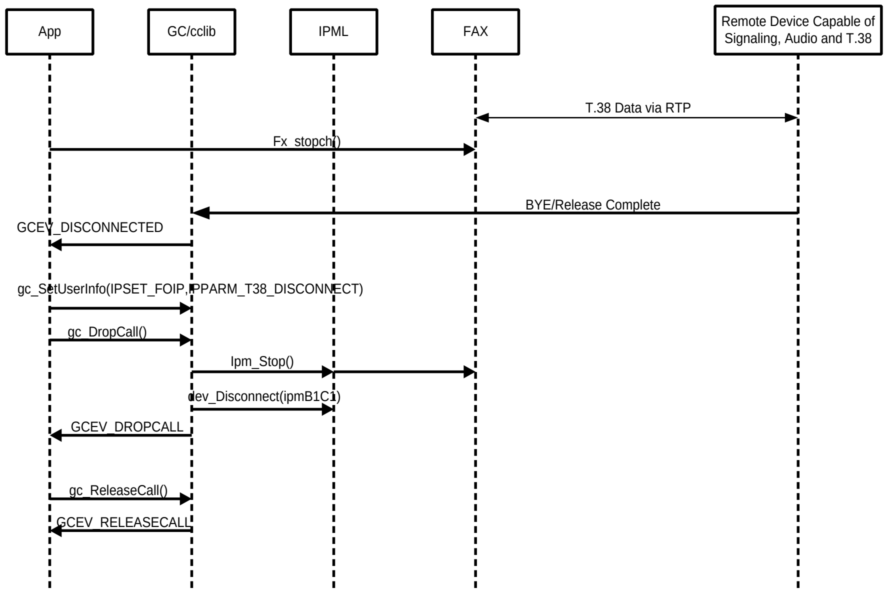
3.4.7 Terminating a Call After a T.38 Fax Session

In any scenario where a T.38 session is established and fax is complete, the application can terminate the call without switching to audio. In either outbound or inbound call termination, the application must disassociate the T.38 Fax device from the Media device before calling **gc_DropCall( )**. This ensures the Media device in the correct state for the next call.

Terminating a call after an audio session follows the normal Global Call call procedures.

Note: The application must not use both Dialogic® Global Call API and Dialogic® IP Media Library API functions on the same device. The Dialogic® IP Media Library API calls (ipm_) in Figure 46 are shown for informational purposes only. Global Call interacts with the IP Media Library on behalf of the application.

Figure 46. Terminating a Call After a T.38 Fax Transfer.



3.4.8 Recovering from a Session Switching Failure

Switching to T.38 Fax or audio may fail due to any a number of reasons, for example, rejection or no response from remote endpoint. It is highly recommended that the application set up a timer for a minimum of 35 seconds for each switching request. If a timeout occurs while waiting for a GCEV_EXTENSION event that has an associated IPPARM_READY parameter, the application has two options:

- Attempt to switch back to original session as if the GCEV_EXTENSION event were received without media capability.
- Terminate the call as if GCEV_EXTENSION event were received without media capability.

If the application times out when switching to T.38 Fax (that is, it does not receive a GCEV_EXTENSION event with an IPPARM_READY parameter within the timeout period), it should follow the scenarios described in Section 3.4.5, "Sending a Request to Switch From T.38 Fax to Audio", on page 95, Section 3.4.6, "Receiving a Request to Switch From T.38 Fax to Audio", on page 96, or Section 3.4.7, "Terminating a Call After a T.38 Fax Session", on page 97.

Note: The application must call the **gc_SetUserInfo( )** function with a GC_PARM_BLK that contains a set ID of IPSET_FOIP and a parameter ID of IPPARM_T38_DISCONNECT to disassociate the devices in any of the scenarios.

If the application times out when switching to audio (that is, it does not receive a GCEV_EXTENSION event with an IPPARM_READY parameter within the timeout period), it should follow the scenarios described in Section 3.4.1, "Sending T.38 Fax in an Established Audio Session", on page 90, Section 3.4.2, "Receiving T.38 Fax in an Established Audio Session", on page 91, or drop the call as if in audio session.

IP-Specific Operations 4

This chapter describes how to use the Dialogic® Global Call API to perform certain operations in an IP environment. These operations include:

4.1 Call Control Library Initialization

Certain system parameters are configurable when using the **gc_Start( )** function to initialize the Dialogic® Global Call API library. Some of these parameters, such as the number of virtual boards and the choice of first party or third party call control operating mode, are set for the entire system, but most of the configuration parameters are set separately for each of the virtual boards in the system.

Among the configuration items that can be set for on a per-virtual board basis are:

- the maximum number of IPT devices available on the virtual board (total, H.323, and SIP)
- the local IP address
- the call signaling ports (H.323 and SIP)
- the terminal type (H.323 only)
- the outbound proxy (SIP only)

In addition, the configuration process is also used to enable certain features that have been added to the Dialogic® Global Call API library as it has evolved in order to ensure backwards compatibility. These features include:

- the call transfer supplementary service
- the ability to access H.323 message information fields and/or SIP message header fields

- the ability to access MIME-encoded message bodies in SIP messages
- the ability to access tunneled signaling messages (TSMs) and/or user-to-user information elements (UU-IEs) in H.323 messages
- the ability to control the transport protocol and retry behavior for SIP messages
- the ability to use Transport Layer Security (TLS) for SIP messages
- the ability to handle SIP OPTIONS requests under application control

System configuration is accomplished using two different data structures, which are initialized to default values and then customized to suit the specific configuration before calling the **gc_Start( )** function. System-level configuration items are set in a IPCCLIB_START_DATA data structure, which also references an array of IP_VIRTBOARD data structures (one per virtual board) that specify board-level configuration items.

The application begins the configuration process by using the **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** functions to initialize the IPCCLIB_START_DATA structure and each of the IP_VIRTBOARD data structures. These initialization functions set default values that can then be overridden with desired values. After setting whatever non-default values it desires (there is no need for the application to set any item that it is leaving at the default value), the application references the IPCCLIB_START_DATA structure from a CCLIB_START_STRUCT structure, which in turn is referenced from the GC_START_STRUCT structure that is passed to the **gc_Start( )** function.

For details on the overall configuration process, including the default values and the allowable values that can be set for each configuration item, see Section 8.3.27, "gc_Start( ) Variances for IP", on page 590, the reference page for IP_VIRTBOARD on page 665, and the reference page for IPCCLIB_START_DATA on page 671. In addition to this overall information, details on how to configure specific capabilities and features (including code snippets showing specific configurations) are provided in the sections of this chapter that document those features, including the following subsections which describe the configuration of the SIP outbound proxy and the SIP transport protocol.

Note: Features that are enabled or configured via the IPCCLIB_START_DATA and IP_VIRTBOARD structures cannot be disabled or reconfigured once the library has been started. All items set in these data structures take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

4.1.1 Setting a SIP Outbound Proxy

When initializing a board device for use with SIP, the application can set an outbound proxy. When such a proxy is set, all outbound requests are sent to the proxy address rather than the IP address of the original Request-URI. The proxy can be set by specifying an IP address or a host name in the IP_VIRTBOARD structure that is used in the **gc_Start( )** function. If both an IP address and a host name are specified in IP_VIRTBOARD, the IP address takes precedence.

Note: A Contact header field in a 200OK response from the called party (User Agent Server, or UAS) resulting from a Dialogic® HMP INVITE will cause the ACK message and all future requests out of Dialogic® HMP Software to bypass the proxy and go directly to the UAS for the remaining of the dialog.

For related information, see also Section 4.33, "Dynamic Selection of Outbound SIP Proxy", on page 366.

The following code snippet illustrates how to set a SIP outbound proxy for a single board:

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..
..
/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data */
INIT_IPCCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);

/* initialize virtual board */
INIT_IP_VIRTBOARD(&virtBoards[0]);

// set outbound proxy by IP Address
virtBoards[0].outbound_proxy_IP.ip_ver = IPVER4;
virtBoards[0].outbound_proxy_IP.u_ipaddr.ipv4 = inet_addr("192.168.1.227");

// set outbound proxy by hostname.
// if outbound proxy is also set by IP address, this is ignored
char OutboundProxyHostName[256];
strcpy(OutboundProxyHostName,"my_outbound_proxy");
virtBoard[0].outbound_proxy_hostname = OutboundProxyHostName;

// set outbound proxy port
virtBoards[0].outbound_proxy_port = 5060;
```

4.1.2 Configuring SIP Transport Protocol

When initializing a board device for use with SIP, the application can enable the use of the TCP transport protocol in addition to the default UDP transport.

When TCP is enabled, the Dialogic® Global Call API library listens for incoming TCP connections as well as UDP connections on the SIP signaling port that is configured for the board.

When TCP is enabled, an outbound message is sent using TCP if any of the following three conditions is true:

- The board device was configured with TCP as the default transport protocol if there is no proxy, or with TCP as the outbound proxy protocol if there is a SIP proxy configured.
- TCP is explicitly specified by setting the string ";transport=tcp" in the Request-URI header field before the message is sent. (Note that this requires the SIP Message Info feature to have been enabled by setting the IP_SIP_MSGINFO_ENABLE mask value in the sip_msginfo_mask field of IP_VIRTBOARD before starting the board.)
- The size of the outgoing message is larger than the configured maximum size for UDP messages, which is 1300 by default.

If none of these conditions is true, UDP is used as the default transport protocol.

Note that network conditions may cause UDP packets to be lost, which can cause SIP messages to be lost. And because SIP does not require some response messages to be retransmitted if the message is lost (1xx informational responses, for example), there are circumstances when the Global Call library is unable to generate a completion event because the expected response is never received. Applications should be written to handle cases caused by missing non-reliable response messages when using UDP transport protocol.

The SIP transport protocol is configured by five fields in the IP_VIRTBOARD structure that is used in the **gc_Start( )** function:

E_SIP_tcpenabled
: Enables TCP support. The default value disables TCP so that all outgoing messages are sent over UDP and incoming TCP messages are refused. No TCP capabilities are available unless this parameter is set to the Enabled value.

E_SIP_OutboundProxyTransport
: Sets the transport protocol that is used by the SIP outbound proxy if the virtual board is configured with a proxy and TCP is enabled. The default value sets UDP as the transport for the proxy. Setting this parameter to the TCP value when TCP is not enabled, or when TCP is enabled but no proxy is configured causes a bad parameter error when **gc_Start( )** is called.

E_SIP_Persistence
: Sets the persistence for TCP connections, with options for no persistence (connection closed after each request), transaction persistence (connection closed when transaction is completed), or user persistence (connection maintained for the lifetime of the user of the transaction). The default is user persistence, which minimizes the number of times that sockets are set up and torn down.

SIP_maxUDPmsgLen
: Sets a maximum size for UDP messages. If TCP is enabled and the application attempts to send a message by UDP that exceeds the configured maximum size (default is 1300 as suggested in RFC3261), TCP transport is automatically used rather than UDP. This size checking may have an undesirable effect on system performance, and a parameter value is defined which disables the feature.

E_SIP_DefaultTransport
: Sets the default transport protocol for requests when there is no SIP outbound proxy. The default value sets UDP as the default transport protocol. Setting this parameter to the TCP value when TCP is not enabled causes a bad parameter error when **gc_Start( )** is called. If TCP is enabled, the application can override the default transport for a specific request by explicitly setting a "transport= " parameter in the Request-URI header field before sending the request.

See the reference page for IP_VIRTBOARD on page 665, for full details on the data structure fields and values.

4.1.2.1 Configuring TCP Transport

With five configuration items controlling TCP transport, the number of possible configuration combinations is clearly very large. The tables in this section list the combinations of configuration parameter settings that are used to achieve various system behaviors. Note that the tables include entries for the outbound proxy configuration, since the transport configuration differs depending on whether or not a proxy is enabled, and the SIP message information mask, which must be configured to allow the application to set the transport for individual requests.

The following code snippet illustrates the general procedure for setting up the IP_VIRTBOARD structure to enable TCP. This specific example sets up a SIP outbound proxy, enables TCP, and sets TCP as the default transport protocol for the proxy for a single board. Note that all data structure fields that are not explicitly set are assumed to contain their default values as configured by the **INIT_IP_VIRTBOARD( )** function.

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..
..
/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data */
INIT_IPCCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);

/* initialize virtual board */
INIT_IP_VIRTBOARD(&virtBoards[0]);

// Enable SIP Message Info to allow transport selection for individual requests
virtBoards[0].ip_sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE;

// set outbound proxy by IP Address
virtBoards[0].outbound_proxy_IP.ip_ver = IPVER4;
virtBoards[0].outbound_proxy_IP.u_ipaddr.ipv4 = inet_addr("192.168.1.227");

// set outbound proxy port
virtBoards[0].outbound_proxy_port = 5060;

//enable and configure TCP for proxy
virtBoards[0].E_SIP_tcpenabled = ENUM_Enabled;
virtBoards[0].E_SIP_OutboundProxyTransport = ENUM_TCP;
virtBoards[0].E_SIP_Persistence = ENUM_PERSISTENCE_TRANSACT_USER;
```

Transport Parameter Combinations without Proxy

All Requests UDP

Parameter	Value
E_SIP_tcpenabled	ENUM_Disabled (default)
E_SIP_OutboundProxyTransport	not set
E_SIP_Persistence	not set
SIP_maxUDPmsgLen	not set
E_SIP_DefaultTransport	not set
outbound_proxy_* fields	IP and hostname both not set
sip_msginfo_mask	any value
transport parameter in Request-URI	not set

All Requests TCP

Parameter	Value
E_SIP_tcpenabled	ENUM_Enabled
E_SIP_OutboundProxyTransport	not set
E_SIP_Persistence	ENUM_PERSISTENCE_TRANSACT_USER (default)
SIP_maxUDPmsgLen	not set
E_SIP_DefaultTransport	ENUM_TCP
outbound_proxy_* fields	IP and hostname both not set
sip_msginfo_mask	any value
transport parameter in Request-URI	not set

Selected Requests TCP

Parameter	Value
E_SIP_tcpenabled	ENUM_Enabled
E_SIP_OutboundProxyTransport	not set
E_SIP_Persistence	ENUM_PERSISTENCE_TRANSACT_USER (default)
SIP_maxUDPmsgLen	1300 (default)
E_SIP_DefaultTransport	ENUM_UDP (default)
outbound_proxy_* fields	IP and hostname both not set
sip_msginfo_mask	includes IP_SIP_MSGINFO_ENABLE
transport parameter in Request-URI	set to ";transport=tcp" on selected requests

Selected Requests UDP

Parameter	Value
E_SIP_tcpenabled	ENUM_Enabled
E_SIP_OutboundProxyTransport	not set
E_SIP_Persistence	ENUM_PERSISTENCE_TRANSACT_USER (default)
SIP_maxUDPmsgLen	not set
E_SIP_DefaultTransport	ENUM_TCP
outbound_proxy_* fields	IP and hostname both not set
sip_msginfo_mask	includes IP_SIP_MSGINFO_ENABLE
transport parameter in Request-URI	set to ";transport=udp" on selected requests

Transport Parameter Combinations with Proxy

All Requests UDP via Proxy

Parameter	Value
E_SIP_tcpenabled	ENUM_Disabled (default)
E_SIP_OutboundProxyTransport	not set
E_SIP_Persistence	not set
SIP_maxUDPmsgLen	not set
E_SIP_DefaultTransport	not set
outbound_proxy_* fields	IP -or- hostname set
sip_msginfo_mask	any value
transport parameter in Request-URI	not set

Requests are sent UDP to the proxy, and the proxy sends the request onward using UDP (unless the proxy resolves the destination as being TCP, based on DNS information).

All Requests TCP via Proxy

Parameter	Value
E_SIP_tcpenabled	ENUM_Enabled
E_SIP_OutboundProxyTransport	ENUM_TCP
E_SIP_Persistence	ENUM_PERSISTENCE_TRANSACT_USER (default)
SIP_maxUDPmsgLen	default (not set)
E_SIP_DefaultTransport	not set
outbound_proxy_ fields	IP -or- hostname set
sip_msginfo_mask	any value
transport parameter in Request-URI	not set

Requests are sent TCP to the proxy, and the proxy sends the request onward using TCP.

Selected Requests TCP via Proxy

Parameter	Value
E_SIP_tcpenabled	ENUM_Enabled
E_SIP_OutboundProxyTransport	ENUM_UDP (default)
E_SIP_Persistence	ENUM_PERSISTENCE_TRANSACT_USER (default)
SIP_maxUDPmsgLen	1300 (default)
E_SIP_DefaultTransport	not set
outbound_proxy_ fields	IP -or- hostname set
sip_msginfo_mask	includes IP_SIP_MSGINFO_ENABLE
transport parameter in Request-URI	set to ";transport=tcp" for selected requests

Selected requests are sent TCP to the proxy, and the proxy sends the request onward using TCP. Other requests are sent UDP to proxy, and are sent onward using UDP (unless the proxy resolves the destination as being TCP, based on DNS information).

Invalid Transport Parameter Combinations

If **TCP is not enabled** (E_SIP_tcpenabled is the default ENUM_Disabled value), the following parameter settings are invalid:

- If E_SIP_OutboundProxyTransport is set to ENUM_TCP, **gc_Start( )** returns an IPERR_BAD_PARM error.
- If E_SIP_DefaultTransport is set to ENUM_TCP, **gc_Start( )** returns an IPERR_BAD_PARM error.
- Setting the Request-URI transport parameter to ";transport=tcp" is invalid but does not produce an error. The invalid header field parameter is ignored, and the request is sent using UDP.

If **TCP is enabled** (E_SIP_tcpenabled is set to ENUM_Enabled), and **no SIP outbound proxy** is set (neither outbound_proxy_IP nor outbound_proxy_hostname is set), the following parameter setting is invalid:

- If E_SIP_OutboundProxyTransport is set to ENUM_TCP, **gc_Start( )** returns an IPERR_BAD_PARM error.

4.1.3 Enabling and Disabling H.245 Tunneling (H.323)

Tunneling is the encapsulation of H.245 media control messages within Q.931/H.225 signaling messages. If tunneling is enabled, one less TCP port is required for incoming connections.

For outgoing calls, the application can enable or disable tunneling by including the following parameter element in the GCLIB_MAKECALL_BLK used by the **gc_MakeCall( )** function:

IPSET_CALLINFO
    IPPARM_H245TUNNELING
    Possible values:
    - IP_H245TUNNELING_ON
    - IP_H245TUNNELING_OFF

For incoming calls, tunneling is enabled by default, but it can be configured on a board device level (where a board device is a virtual entity that corresponds to a NIC or NIC address; see Section 2.3.2, "IPT Board Devices", on page 39). This is done using the **gc_SetConfigData( )** function with target ID of the board device and the parameters above specified in the GC_PARM_BLKP structure associated with the **gc_SetConfigData( )** function.

Note: Tunneling for inbound calls can be configured on a board device basis only (using the **gc_SetConfigData( )** function). Tunneling for inbound calls **cannot** be configured on a per line device or per call basis (using the **gc_SetUserInfo( )** function).

4.2 Fast and Slow Call Setup Modes

The Dialogic® Global Call API library allows applications to specify whether they wish to use signaling techniques that exchange media capabilities as early as possible in the call initiation process. In general, this "fast start" call setup is preferable to the "slow start" setup for several reasons:

- fewer network round trips are required to set up a call
- media streaming may be possible earlier in the pre-connection phase ("early media")
- the local exchange can generate messages when circumstances prevent a connection to the endpoint

4.2.1 Setting the Call Setup Mode

Note: The selection of fast start vs. slow start mode only applies to the first party call control (1PCC) operating mode. If the Dialogic® Global Call API library is initialized in the third party call control (3PCC) operating mode, the IPSET_CALLINFO / IPPARM_CONNECTIONMETHOD parameter that is documented in this section is not supported.

The same Global Call parameter mechanism is used to specify slow start vs. fast start mode for both the H.323 and SIP protocols, even though the result of the mode selection is quite different in the different protocols. See Section 4.2.2, "H.323 Fast Start and Slow Start", on page 109, and Section 4.2.4, "SIP Call Setup Modes", on page 111, for protocol-specific details on the connection modes.

Dialogic® Global Call API applications can set either the fast or slow call setup mode as the default mode for the entire system or for all calls on a given line device, and can also override that default on a call-by-call basis. If the application takes no action to specify the setup mode, the system default is fast start mode.

To specify the slow start mode, either for an individual call or as the default mode, the application inserts the following parameter element in a GC_PARM_BLK:

IPSET_CALLINFO
- IPPARM_CONNECTIONMETHOD
  - value = IP_CONNECTIONMETHOD_SLOWSTART

The scope of the mode setting is determined by which Global Call function the application passes the GC_PARM_BLK to:

- **gc_SetConfigData( )** sets the slow start mode as the default for the entire system (all line devices on all board devices for both H.323 and SIP protocols).
- **gc_SetUserInfo( )** with **duration** = GC_ALLCALLS sets the slow start mode as the default connection mode for H.323 and SIP calls on a given line device.
- **gc_MakeCall( )** with the GC_PARM_BLK in the GCLIB_MAKECALL_BLK structure sets the slow start connection mode for the new call only.

The following code segment illustrates how to insert the parameter that specifies a slow start connection in a GC_PARM_BLK:

```
gc_util_insert_parm_val(&libBblock.ext_datap,
                        IPSET_CALLINFO,
                        IPPARM_CONNECTIONMETHOD,
                        sizeof(char),
                        IP_CONNECTIONMETHOD_SLOWSTART);
```

If the application has previously set the default mode to the slow start mode, it can override that default for an individual call or can reset the default to fast start mode by inserting the following parameter element in a GC_PARM_BLK:

IPSET_CALLINFO
  IPPARM_CONNECTIONMETHOD
  - value = IP_CONNECTIONMETHOD_FASTSTART

Here again, the Global Call function that is used determines the scope of the setting:

- **gc_MakeCall( )** with the GC_PARM_BLK in the GCLIB_MAKECALL_BLK structure sets the fast start connection mode for the new call only.
- **gc_SetUserInfo( )** with **duration** = GC_ALLCALLS resets the default mode to fast start for a given line device for both H.323 and SIP protocols.
- **gc_SetConfigData( )** resets the default mode for the entire system (all line devices on all board devices) to fast start for both protocols.

4.2.2 H.323 Fast Start and Slow Start

H.323 version 2 defines a specific call connection method called *fastStart*, which exchanges endpoint media capabilities much earlier in the setup process than the call connection method defined in H.323 version 1 (a process which then became known as slow start setup). If the remote side supports H.323 version 2 or above, fast start setup can be used; otherwise, slow start setup is used even if the local endpoint attempts to initiate a call using fast start setup.

In H.323 slow start setup, the messages that are used to communicate each endpoint's supported media capabilities are exchanged using the H.245 channel that is established after the H.225 TCP connection, and this introduces significant latency. Media streaming cannot be established until both sides have communicated and negotiated their capabilities in multiple message exchanges. Early media is not possible in H.323 when slow start connection is specified by either party.

Fast start connection, on the other hand, reduces the time required to set up a call to one round-trip of delay after the H.225 TCP connection is established by "piggy-backing" the local endpoint's media capabilities and RTP port in the Q.931 Setup message in a "fastStart element". If the remote side supports fast start connection, it returns the capability parameters in the Alerting, Proceeding, or Connect messages.

Note: In an H.323 fast start call, the fast start element is included in the H.225 Proceeding or Alerting from the remote side only when the application explicitly specifies the coders. If no coder is specified (either a preferred coder or "don't care") before **gc_CallAck( )** and **gc_AcceptCall( )** the fastStart element is not sent out until the Connect (that is, after **gc_AnswerCall( )**).

4.2.3 H.323 Fast Start with Optional H.245 Channel

Because the H.323 fast start mode uses fastStart elements that are embedded in H.225/ Q.931 call setup messages rather than explicit messages on the H.245 channel, the establishment of the H.245 channel becomes optional unless that channel will be needed for other purposes, such as transmission of UII Alphanumeric digits or T.38 fax mode.

When a Global Call application is using the fast start connection mode, it can indicate that the H.245 channel is indeed optional, which allows the call to be considered established earlier. In a normal fast start connection, the Dialogic® Global Call API library does not generate a GCEV_CONNECTED or GCEV_ANSWERED event (to indicate to the application that call establishment is complete) until after the H.245 channel establishment (Phase B) is complete. When the application at the calling party specifies that the H.245 channel is optional, the library generates a GCEV_CONNECTED event as soon as the H.225 call setup (Phase A) is complete unless the remote endpoint has forced the call to fall back to slow start mode. When the application at the called party specifies that the H.245 channel is optional, the library generates a GCEV_ANSWERED event as soon as the H.225 call setup is complete.

The default Global Call behavior is to treat H.245 channel establishment as mandatory (non-optional), so that GCEV_CONNECTED/GCEV_ANSWERED is only generated after the H.245 channel has been established. The application can specify whether the H.245 channel is optional in fast start mode by including the following parameter element in a GC_PARM_BLK block:

IPSET_CALLINFO
  IPPARM_FASTSTART_MANDATORY_H245CH
  with one of the following enumerated values:
  - IP_FASTSTART_MANDATORY_H245CH_ON – H.245 channel establishment is mandatory in fast start connections (default mode)
  - IP_FASTSTART_MANDATORY_H245CH_OFF – H.245 channel establishment is optional in fast start connections

Note: This parameter is ignored for calls that use slow start call setup.

An application can set the H.245 channel establishment mode on a system-wide, per line device, or call-by-call basis, depending on what Global Call function is called to set the parameter:

- **gc_SetConfigData( )** sets the specified H.245 mode for the entire system (all line devices on all board devices).
- **gc_SetUserInfo( )** with **duration** = GC_ALLCALLS sets the specified H.245 mode for a given line device.
- **gc_MakeCall( )** sets the specified H.245 mode for the new call only.

When the application specifies that the H.245 channel is optional, channel establishment proceeds normally with the exchange of MSD and TCS messages and acknowledgements after the library has generated a GCEV_CONNECTED event to the application (assuming that the remote endpoint accepts fast start setup). The application can optionally receive notification of the status of H.245 channel establishment by means of a maskable Global Call extension event. This notification is recommended if the application will require the H.245 channel for any purpose (for example, T.38 fax mode or UII Alphanumeric messages) because an attempt to use the H.245 channel when the channel was not successfully established produces a GCEV_TASKFAIL.

In order to be notified of the completion of H.245 channel establishment (successful or failed), the application must register to receive the corresponding Global Call extension event type. The application must call the **gc_SetConfigData( )** function, passing it a pointer to a GC_PARM_BLK that contains the following parameter:

IPSET_EXTENSIONEVT_MSK
    GCACT_ADDMSK (or GCACT_SETMSK)
        • EXTENSIONEVT_SIGNALING_STATUS

When the application has registered for this event type and the H.245 channel establishment fails, the Dialogic® Global Call API library generates an unsolicited GCEV_EXTENSION event with the extension ID IPEXTID_IPPROTOCOL_STATE. The parameter block associated with this event will contain the following parameter element:

IPSET_IPPROTOCOL_STATE
    IPPARM_EST_CONTROL_FAILED

The application may also call **gc_ResultInfo( )** in this case to retrieve additional information about the cause of the channel establishment failure. The error cause codes that may be returned include:

- IPEC_H245EstChannelFailure_TransportError
- IPEC_H245EstChannelFailure_RemoreReject
- IPEC_H245EstChannelFailure_TCSError
- IPEC_H245EstChannelFailure_MSDError

If the application is using fast start setup mode with optional H.245 channel and the channel establishment fails, and the application then attempts an operation that requires the H.245 channel (for example, sending UII Alphanumeric characters), the library generates a GCEV_TASKFAIL event. The application may call **gc_ResultInfo( )** to retrieve one of the error cause codes listed above.

4.2.4 SIP Call Setup Modes

Unlike H.323, the SIP protocol does not define a “fast start” connection mode. In SIP, the exchange of media capabilities is accomplished via an offer/answer exchange using Session Description Protocol (SDP). This SDP offer/answer exchange can be initiated by either the local or the remote party, and the SDP information can be embedded in any of the request or response messages that are exchanged when establishing a SIP dialog. Normal practice is to include the SDP offer in the INVITE message that initiates a SIP dialog, which corresponds to a “fast start” connection mode. SIP uses the term *delayed offer* to refer to cases where the INVITE does not include the SDP offer, which corresponds to a “slow start” connection mode.

When the calling party in a SIP call uses the default fast start setup mode, the SDP offer is included in the INVITE message that initiates the connection attempt. The remote party then sends an SDP answer in its 200 OK response. (The remote party may optionally include the SDP answer in an informational response such as 180 RINGING, but because informational responses are not reliable messages in SIP the SDP answer will always be included in the reliable 200 OK final response.)

When the calling party in a SIP call specifies the slow start setup mode (delayed offer in SIP terminology), the initial INVITE does not include an SDP offer. Instead, it is left to the remote party to make the SDP offer in its 200 OK. The calling party then sends the SDP answer in its ACK to the 200 OK.

Note: The use of the connection method parameter to control when the SDP offer/answer process is initiated is supported only in the first party call control (1PCC) operating mode. In the third party call control (3PCC) operating mode, the application explicitly controls when the SDP offer and answer are sent and the connection method parameter is not supported.

4.2.5 Retrieving Coder Information from Call Offers

Note: The information in this section only applies when the Global Call IP Call Control library is started in the first party call control (1PCC) operating mode. The capability described in this section is not supported when the library is started in the third party call control (3PCC) operating mode.

Any call offer that is received can potentially contain coder proposal information, in the form of an SDP offer in an INVITE request when using SIP or a fastStart element in a Setup message when using H.323. The IP call control library handles any such proposed coder information internally to begin the coder negotiation process, but it may be useful to the application to access the offered coder information, as well. The call control library can be configured at start-up to provide application access to proposed coder information for SIP or H.323 or both. When this access is enabled and the library accepts a call offer that contains coder proposals, the extra data associated with the GCEV_OFFERED event that is sent to the application will contain one or more additional parameter elements to convey the coder information that was contained in the offer.

4.2.5.1 Enabling Access to “Fast Start” Coder Information

Application access to fast start coder information is a feature that can be disabled or enabled independently for the SIP and H.323 protocols at the time the **gc_Start( )** function is called.

The mandatory **INIT_IP_VIRTBOARD( )** function populates the IP_VIRTBOARD structure with default values. The default values of the sip_msginfo_mask and h323_msginfo_mask fields in the IP_VIRTBOARD structure disable all optional message information access features, including access to coder proposal information. The default values of these data structure fields must be overridden with appropriate values for each ipt board device on which access needs to be enabled. For each of the two message information mask fields, the value that the application sets is typically an OR of two or more defined mask values as described in the reference page for IP_VIRTBOARD on page 665.

The defined mask values that are used to enable access to fast start coder information are:

IP_SIP_FASTSTART_CODERS_IN_OFFERED
: when OR’ed into the sip_msginfo_mask field, enables application access to coder information contained in SDP offers in SIP INVITE requests

IP_H323_FASTSTART_CODERS_IN_OFFERED
: when OR'ed into the h323_msginfo_mask field, enables application access to coder information contained in fastStart elements in H.323 Setup messages

Note: Note that it is not possible to toggle the fast start coder information access between enabled and disabled states. Features that are enabled or configured via the IPCCLIB_START_DATA and IP_VIRTBOARD structures cannot be disabled or reconfigured once the library has been started. All items set in these data structures take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

The following code snippet shows how an application might initialize two virtual boards to enable basic message information access and access to fast start coder information for both SIP and H.323 protocols.

```
.
.
.
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sip_msginfo_mask =
         IP_SIP_MSGINFO_ENABLE | IP_SIP_FASTSTART_CODERS_IN_OFFERED;
         /* override SIP default to enable access to message info and faststart coder info*/
ip_virtboard[1].sip_msginfo_mask =
         IP_SIP_MSGINFO_ENABLE | IP_SIP_FASTSTART_CODERS_IN_OFFERED;
         /* override SIP default to enable access to message info and faststart coder info*/
ip_virtboard[0].h323_msginfo_mask =
         IP_H323_MSGINFO_ENABLE | IP_H323_FASTSTART_CODERS_IN_OFFERED;
         /* override H.323 default to enable access to message info and faststart coder info*/
ip_virtboard[1].h323_msginfo_mask =
         IP_H323_MSGINFO_ENABLE | IP_H323_FASTSTART_CODERS_IN_OFFERED;
         /* override H.323 default to enable access to message info and faststart coder info*/
.
.
.
```

4.2.5.2 Accessing "Fast Start" Coder Information

The Global Call IP call control library includes coder information in the extra data associated with a GCEV_OFFERED event when all of the following conditions are true:

- The library was started with the fast start coder information option enabled for the appropriate protocol (as described in Section 4.2.5.1, "Enabling Access to "Fast Start" Coder Information").
- The fast start mode is enabled (as described in Section 4.2.1, "Setting the Call Setup Mode").
- The call offer is a fast start offer; that is, it includes an SDP offer (SIP) or fastStart element (H.323).
- The SDP offer or fastStart element specifies at least one coder that the library supports.

When all of these conditions are true, the extra data associated with the GCEV_OFFERED event will be a GC_PARM_BLK that contains one or more parameter elements of the following type:

IPSET_CALLINFO
: IPPARM_OFFERED_FASTSTART_CODER
  - value = IP_CAPABILITY data structure

Each such parameter element reflects a coder specification that was contained in the call offer. If the offer contains multiple coder specifications, the order of the parameter elements in the parameter block reflects the order of the specifications in the offer message. This order reflects the remote endpoint's coder preference, with the first specification being the most preferred and the last specification being the least preferred. If any coder properties were left unspecified by the remote end, the matching fields in the corresponding IP_CAPABILITY structure are filled in with the value GCCAP_dontCare.

If any of the four conditions described above is not true, there will be no IPSET_CALLINFO / IPPARM_OFFERED_FASTSTART_CODER parameter element in the parameter block associated with the GCEV_OFFERED.

When the IP_CAPABILITY data structure is used to convey fast start coder information, the direction field of the structure uses the following special value defines:

IP_CAP_DIR_RMTRECEIVE
: Remote coder was specified to be Receive-only.

IP_CAP_DIR_RMTRTPINACTIVE
: Remote coder was specified with "a=inactive", which is used in SIP to inactivate RTP streaming. Only supported when using SIP.

IP_CAP_DIR_RMTRTPRTCPINACTIVE
: Remote coder was specified with RTP address 0.0.0.0, which is used in SIP to inactivate both RTP and RTCP. Only supported when using SIP.

IP_CAP_DIR_RMTTRANSMIT
: Remote coder was specified to be Transmit-only.

IP_CAP_DIR_RMTTXRX
: Remote coder was specified to be capable of both Transmit and Receive.

4.3 Setting Call-Related Information

The Dialogic® Global Call API allows applications to set many items of call-related information. The following topics are presented in this section:

- Overview of Setting Call-Related Information
- Setting Coder Information
- Specifying Nonstandard Data Information (H.323)
- Specifying Nonstandard Control Information (H.323)
- Specifying IPv4 or IPv6 Address

4.3.1 Overview of Setting Call-Related Information

Table 1 summarizes the types of information elements that can be specified, the corresponding set IDs and parameter IDs used to set the information, the functions that can be used to set the information, and an indication of whether the information is supported when using H.323, SIP, or both. For more information on the various parameters, refer to the corresponding parameter set reference section in Chapter 9, “IP-Specific Parameters”.

Table 1. Summary of Call-Related Information that can be Set

Type of Information	Set ID and Parameter IDs	Functions Used to Set Information	SIP/ H.323
Bearer Capability IE	IPSET_CALLINFO • IPPARM_BEARERCAP	**gc_SetUserInfo( )** (GC_SINGLECALL only)	H.323

Call ID (GUID)	IPSET_CALLINFO
	• IPPARM_CALLID
	Note: Setting the Call ID must be done judiciously because it might affect the call control implementation supported by the stack. The Call ID should be treated as a GUID and should be unique at all times.
		gc_SetUserInfo( ) (GC_SINGLECALL only)
		gc_MakeCall( )
			both

Coder Information †	GCSET_CHAN_CAPABILITY
	• IPPARM_LOCAL_CAPABILITY
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			both

Conference Goal	IPSET_CONFERENCE
	• IPPARM_CONFERENCE_GOAL
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			H.323

Connection Method	IPSET_CALLINFO
	• IPPARM_CONNECTIONMETHOD
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			both

DTMF Support	IPSET_DTMF • IPPARM_SUPPORT_DTMF_BITMASK	**gc_SetConfigData( )** **gc_SetUserInfo( )** ††	both

Display Information	IPSET_CALLINFO
	• IPPARM_DISPLAY
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			both

Enabling/Disabling Unsolicited Events	IPSET_EXTENSIONEVT_MSK • GCACT_ADDMSK • GCACT_SETMSK • GCACT_SUBMSK	**gc_SetConfigData( )**	both
Facility IE	IPSET_CALLINFO • IPPARM_FACILITY	**gc_SetUserInfo( )** (GC_SINGLECALL only)	H.323
IPv6 addressing	IPSET_SDP • IPPARM_SDP_IP_TYPE	**gc_SetUserInfo( )**	SIP

MediaWaitFor Connect	IPSET_CALLINFO
	• IPPARM_MEDIAWAITFORCONNECT
		gc_SetUserInfo( ) (GC_SINGLECALL only)
		gc_MakeCall( )
			H.323

† If no transmit or receive coder type is specified, any supported coder type is accepted. The default is “don’t care”; that is, any media coder supported by the platform is valid.
†† The duration parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
††† On the terminating side, can only be set using **gc_SetConfigData( )** on a board device. See Section 4.1.3, “Enabling and Disabling H.245 Tunneling (H.323)”, on page 107 for more information.

Table 1. Summary of Call-Related Information that can be Set (Continued)

Type of Information	Set ID and Parameter IDs	Functions Used to Set Information	SIP/ H.323

Nonstandard Control Information	IPSET_NONSTANDARDCONTROL
	Either:
	• IPPARM_NONSTANDARDDATA_DATA and IPPARM_NONSTANDARDDATA_OBJID
	or
	• IPPARM_NONSTANDARDDATA_DATA and IPPARM_H221NONSTANDARD
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			H.323

Nonstandard Data	IPSET_NONSTANDARDDATA
	Either:
	• IPPARM_NONSTANDARDDATA_DATA and IPPARM_NONSTANDARDDATA_OBJID
	or
	• IPPARM_NONSTANDARDDATA_DATA and IPPARM_H221NONSTANDARD
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			H.323

Phone List	IPSET_CALLINFO
	• IPPARM_PHONELIST
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			both

Presentation Indicator	IPSET_CALLINFO
	• IPPARM_PRESENTATION_IND
		gc_SetUserInfo( ) (GC_SINGLECALL only)
		gc_MakeCall( )
			H.323

SIP Message Information Fields	IPSET_SIP_MSGINFO • IPPARM_CALLID_HDR • IPPARM_CONTACT_DISPLAY • IPPARM_CONTACT_URI • IPPARM_DIVERSION_URI • IPPARM_FROM_DISPLAY • IPPARM_REFERRED_BY • IPPARM_REPLACES • IPPARM_REQUEST_URI • IPPARM_TO_DISPLAY	**gc_SetUserInfo( )** (GC_SINGLECALL only)	SIP
T.38 Fax device association or disassociation with Media device	IPSET_FOIP • IPPARM_T38_CONNECT • IPPARM_T38_DISCONNECT	**gc_SetUserInfo( )** †	both

Tunnelling†††	IPSET_CALLINFO
	• IPPARM_H245TUNNELING
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			H.323

† If no transmit or receive coder type is specified, any supported coder type is accepted. The default is “don’t care”; that is, any media coder supported by the platform is valid.
†† The duration parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
††† On the terminating side, can only be set using **gc_SetConfigData( )** on a board device. See Section 4.1.3, “Enabling and Disabling H.245 Tunneling (H.323)”, on page 107 for more information.

Table 1. Summary of Call-Related Information that can be Set (Continued)

Type of Information	Set ID and Parameter IDs	Functions Used to Set Information	SIP/ H.323
Type of Service: TOS byte / DiffServ field (DSCP) in IPv4 packet header	IPSET_CONFIG • IPPARM_CONFIG_TOS	**gc_SetUserInfo( )** †† **gc_MakeCall( )**	both

User to User Information	IPSET_CALLINFO
	• IPPARM_USERUSER_INFO
		gc_SetConfigData( )
		gc_SetUserInfo( ) ††
		gc_MakeCall( )
			H.323

Vendor Information	IPSET_VENDORINFO • IPPARM_H221NONSTD • IPPARM_VENDOR_PRODUCT_ID • IPPARM_VENDOR_VERSION_ID	**gc_SetConfigData( )**	H.323

† If no transmit or receive coder type is specified, any supported coder type is accepted. The default is “don’t care”; that is, any media coder supported by the platform is valid.
†† The duration parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
††† On the terminating side, can only be set using **gc_SetConfigData( )** on a board device. See Section 4.1.3, “Enabling and Disabling H.245 Tunneling (H.323)”, on page 107 for more information.

4.3.1.1 Setting Call Parameters on a System-Wide Basis

The **gc_SetConfigData( )** function is used to configure call-related parameters, such as coder information, for the entire system. The values set by the **gc_SetConfigData( )** function are used by the call control library as default values for each line device on each board device in the system. These default values are used unless the application overrides them on a per line-device or per-call basis.

See Section 8.3.25, “gc_SetConfigData( ) Variances for IP”, on page 583 for more information about the values of function parameters to set in this context.

4.3.1.2 Setting Call Parameters on a Per Line Device Basis

The **gc_SetUserInfo( )** function (with the **duration** parameter set to GC_ALLCALLS) can be used to set the values of call-related parameters on a per line-device basis. The values set by **gc_SetUserInfo( )** become the new default values for the specified line device and are used by all subsequent calls on that device unless the application overrides them on a per-call basis. See Section 8.3.26, “gc_SetUserInfo( ) Variances for IP”, on page 587 for more information about the values of function parameters to set in this context.

4.3.1.3 Setting Call Parameters on a Per Call Basis

There are two ways to set call parameters on a per-call basis:

- Using **gc_SetUserInfo( )** with the **duration** parameter set to GC_SINGLECALL
- Using **gc_MakeCall( )**

Setting Per-Call Call Parameters Using gc_SetUserInfo( )

The **gc_SetUserInfo( )** function (with the **duration** parameter set to GC_SINGLECALL) can be used to set call parameter values for a single incoming call. This is useful since the **gc_AnswerCall( )** function does not have a parameter to specify a GC_PARM_BLK. At the end of the call, the values set as defaults for the specified line device replace these call-specific values.

If a **gc_MakeCall( )** function is issued after the **gc_SetUserInfo( )**, the values specified in the **gc_MakeCall( )** function override the values specified by the **gc_SetUserInfo( )** function. See Section 8.3.26, "gc_SetUserInfo( ) Variances for IP", on page 587 for more information about the values of function parameters to set in this context.

Setting Per-Call Call Parameters Using gc_MakeCall( )

The **gc_MakeCall( )** function can be used to set call parameter values for a call. The values set are only valid for the duration of the current call. At the end of the call, the values set as default values for the specified line device override the values specified by the **gc_MakeCall( )** function.

See Section 8.3.17, "gc_MakeCall( ) Variances for IP", on page 560 for more information about the values of function parameters to set in this context.

4.3.2 Setting Coder Information

Terminal capabilities are exchanged during call establishment. The terminal capabilities are sent to the remote side as notification of coder supported.

Coder information can be set in the following ways:

- On a system wide basis using **gc_SetConfigData( )**.
- On a per line device basis using **gc_SetUserInfo( )** with a **duration** parameter value of GC_ALLCALLS.
- On a per call basis using **gc_MakeCall( )** or **gc_SetUserInfo( )** with a **duration** parameter value of GC_SINGLECALL.

In each case, a GC_PARM_BLK is set up to contain the coder information. The GC_PARM_BLK must contain the GCSET_CHAN_CAPABILITY parameter set ID with the IPPARM_LOCAL_CAPABILITY parameter ID, which is of type IP_CAPABILITY.

Possible values for fields in the IP_CAPABILITY structure are:

capability
: Specifies the coder type from among the types supported by the particular IP telephony platform; see Table 2 for platform-specific coder types. The following values are defined for the capability field:
  - GCCAP_AUDIO_g711Alaw64k
  - GCCAP_AUDIO_g711Ulaw64k
  - GCCAP_AUDIO_g7231_5_3k (G.723.1 at 5.3 kbps)
  - GCCAP_AUDIO_g7231_6_3k (G.723.1 at 6.3 kbps)
  - GCCAP_AUDIO_g726_16k
  - GCCAP_AUDIO_g726_24k
  - GCCAP_AUDIO_g726_32k
  - GCCAP_AUDIO_g726_40k
  - GCCAP_AUDIO_g729AnnexA
  - GCCAP_AUDIO_g729AnnexAwAnnexB
  - GCCAP_AUDIO_NO_AUDIO
  - GCCAP_DATA_t38UDPFax
  - GCCAP_dontCare – The complete list of coders supported by a product is used when negotiating the coder type to be used. If multiple variations of the same coder are supported by a product, the underlying call control library offers the preferred variant only. For example, if G.711 10ms, 20ms, and 30ms are supported, only the preferred variant, G.711 20 ms, is included.

type
: One of the following:
  - GCCAPTYPE_AUDIO
  - GCCAPTYPE_RDATA

direction
: One of the following:
  - IP_CAP_DIR_LCLTRANSMIT – transmit capability of full-duplex session
  - IP_CAP_DIR_LCLRECEIVE – receive capability of full-duplex session
  - IP_CAP_DIR_LCLRXTX – transmit and receive capability (T.38 only)
  - IP_CAP_DIR_LCLSENDONLY – capability of a half-duplex transmit-only session
  - IP_CAP_DIR_LCLRECVONLY – capability of a half-duplex receive-only session

payload_type
: Not supported. The currently supported coders have static (pre-assigned) payload types defined by standards.

extra
: Reference to a data structure of type IP_AUDIO_CAPABILITY, which contains the following two fields:
  - frames_per_packet – The number of frames per packet.

  Note: For G.711 coders, the extra.frames_per_packet field sets the frame size (in ms) rather than the frames per packet.

  - VAD – Enables or disables VAD.
    Values: GCPV_DISABLE, GCPV_ENABLE, GCCAP_dontCare

  Note: Applications must explicitly set this field to GCPV_ENABLE for the coders that implicitly support only VAD, such as GCCAP_AUDIO_g729AnnexAwAnnexB.

See the reference page for IP_CAPABILITY on page 652 for more information.

Table 2 shows the coders that are supported when using the Dialogic® Global Call API with Dialogic® Host Media Processing (HMP) Software.

Table 2. Coders Supported for Dialogic® Host Media Processing (HMP) Software

Coder and Rate	Dialogic® Global Call API # Define	Frames Per Packet (fpp) and Frame Size (ms)	VAD Support
G.711 A-law	GCCAP_AUDIO_g711Alaw64k	Frame size[1]: 10, 20, or 30 ms Frames per Packet: fixed at 1 fpp	Not supported; must be explicitly disabled
G.711 mu-law	GCCAP_AUDIO_g711Ulaw64k	Frame size[1]: 10, 20, or 30 ms Frames per packet: fixed at 1 fpp	Not supported; must be explicitly disabled
G.723.1 5.3 kbps	GCCAP_AUDIO_g7231_5_3k	Frames per packet: 2 or 3 Frame size: fixed at 30 ms	Supported
G.723.1, 6.3 kbps	GCCAP_AUDIO_g7231_6_3k	Frames per packet: 2 or 3 Frame size: fixed at 30 ms	Supported
G.726, 16 kbps	GCCAP_AUDIO_g726_16k	Frames per packet: 1, 2, or 3 Frame size: fixed at 20 ms	Not supported; must be explicitly disabled
G.726, 24 kbps	GCCAP_AUDIO_g726_24k	Frames per packet: 1, 2, or 3 Frame size: fixed at 20 ms	Not supported; must be explicitly disabled
G.726, 32 kbps	GCCAP_AUDIO_g726_32k	Frames per packet: 1, 2, or 3 Frame size: fixed at 20 ms	Not supported; must be explicitly disabled
G.726, 40 kbps	GCCAP_AUDIO_g726_40k	Frames per packet: 1, 2, or 3 Frame size: fixed at 20 ms	Not supported; must be explicitly disabled
G.729a	GCCAP_AUDIO_g729AnnexA	Frames per packet: 2, 3, or 4 Frame size: fixed at 10 ms	Not supported; must be explicitly disabled
G.729a+b	GCCAP_AUDIO_g729AnnexA wAnnexB	Frames per packet: 2, 3, or 4 Frame Size: fixed at 10 ms	Must be enabled [2]
T.38	GCCAP_DATA_t38UDPFax	Not applicable	Not applicable

Note:
1. For G.711 coders, the frames_per_pkt field of the IP_AUDIO_CAPABILITY structure is actually used to specify the frame size rather than the fpp. See the reference page for IP_AUDIO_CAPABILITY on page 650 for more information.
2. Applications must explicitly specify VAD support even though G.729a+b implicitly supports VAD.

Note: When using low bit-rate (LBR) coders, reliable in-band transmission of DTMF tones is not possible.

4.3.2.1 Specifying Media Capabilities Before Connection

Applications can only specify media capabilities before initial call connection. For an outbound call, capabilities must be set before or with the **gc_MakeCall( )**. For inbound calls, capabilities must be set before or with the **gc_AnswerCall( )**, but it is recommended that they be set before

gc_AcceptCall( ) to get maximum benefit from Global Call's early media support. Capability types can be GCCAPTYPE_AUDIO and/or GCCAPTYPE_RDATA. The session capabilities that can result when different capabilities are set by applications are listed in the Table 3.

Table 3. Capabilities Set by Application

GCCAPTYPE_AUDIO capability set by application	GCCAPTYPE_RDATA capability set by application	Resulting Capability for Initial Connection
Not set	Not set	Any supported audio capability or T.38 fax.
One or more GCCAP_AUDIO_XXX	Not Set	Any specified audio capability. No T.38 fax.
Not Set	GCCAP_DATA_t38UDPFax	T.38 fax only. No audio cpability.
One or more GCCAP_AUDIO_XXX	GCCAP_DATA_t38UDPFax	Any specified audio capability or T.38 fax.
GCCAP_dontCare	Not Set	Any supported audio capability. No T.38 fax.
GCCAP_dontCare	GCCAP_DATA_t38UDPFax	Any supported audio capability or T.38 fax.

4.3.2.2 Resource Allocation When Using Low-Bit Rate Coders

The number of resources available when using G.723 and G.729 coders is limited. When all resources are consumed, depending on the requirements of the application, different behavior may be observed as follows:

- If the application specifies only G.723 and/or G.729 audio coders before **gc_MakeCall( )**, **gc_CallAck( )**, **gc_AcceptCall( )**, or **gc_AnswerCall( )**, the result is a function failure with an error code of IPERR_TXRXRESOURCESINSUFF.
- If the application specifies G.711 with G.723 and/or G.729 audio coders, only the G.711 coder will be provided in the capability set sent to the remote endpoint.
- If the application does not explicitly specify any audio capability, then the G.711 coders (both A-law and µ-law) are included in the capability set sent to the remote endpoint.

LBR coder resources are only released when **gc_ReleaseCallEx( )** is used, regardless of whether the resource was negotiated or not.

Note: When using low bit-rate (LBR) coders, in-band transmission of DTMF tones will not work reliably and should not be attempted.

4.3.3 Specifying Nonstandard Data Information (H.323)

To specify Nonstandard Data information to be included in the H.323 SETUP message, use the **gc_SetUserInfo( )** function with a **duration** parameter of GC_SINGLECALL to preset the information. If the **duration** parameter is set to GC_ALLCALLS, the function fails.

To specify Nonstandard Data, the GC_PARM_BLK pointed by the **infoparmblkp** parameter in the function call must be contain two parameter elements that use the IPSET_NONSTANDARDDATA parameter set ID. The first required parameter element specifies the Nonstandard Data itself, and the second parameter element identifies the type of object identifier to use.

The maximum length of the Global Call parameter used for the Nonstandard Data information is configured at start-up via the max_parm_data_size field in the IPCCLIB_START_DATA structure. The default size is 255 (for backwards compatibility), but applications may configure it to be as large as 4096 bytes. Applications must use the extended **gc_util_..._ex( )** functions to insert or extract any GC_PARM_BLK parameter elements whose data length is defined to be greater than 255.

Note: In practice, applications may not be able to utilize the full maximum length of the nonstandard data parameter element as configured in max_parm_data_size. The H.323 stack limits the overall size of messages to be max_parm_data_size + 512 bytes, and any messages that exceed this limit are truncated without any notification to the application.

The parameter element for the Nonstandard Data data is:

IPSET_NONSTANDARDDATA
  IPPARM_NONSTANDARDDATA_DATA
  - value = Nonstandard Data string, max length = max_parm_data_size (configurable at library start-up)

The parameter element for the Nonstandard Data identifier is one (and only one) of the following:

IPSET_NONSTANDARDDATA
  IPPARM_NONSTANDARDDATA_OBJID
  - value = array of unsigned integers, max length = MAX_NS_PARM_OBJID_LENGTH

IPSET_NONSTANDARDDATA
  IPPARM_H221NONSTANDARD
  - value = IP_H221NONSTANDARD structure

See Section 9.2.19, “IPSET_NONSTANDARDDATA”, on page 631 for more information.

The following code example shown how to set nonstandard data elements:

```
IP_H221NONSTANDARD appH221NonStd;
appH221NonStd.country_code = 181;
appH221NonStd.extension = 31;
appH221NonStd.manufacturer_code = 11;
char* pData = "Data String";
char* pOid = "1 22 333 4444";
choiceOfNSData = 1;/* App decides which type of object identifier to use */

/* setting NS Data */
gc_util_insert_parm_ref_ex(&pParmBlock,
                           IPSET_NONSTANDARDDATA,
                           IPPARM_NONSTANDARDDATA_DATA,
                           (unsigned long)(strlen(pData)+1),
                           pData);
```

```
if (choiceOfNSData) /* App decides the CHOICE of OBJECTIDENTIFIER.
                       It cannot set both objid & H221 */
{
   gc_util_insert_parm_ref(&pParmBlock,
                           IPSET_NONSTANDARDDATA,
                           IPPARM_H221NONSTANDARD,
                           (unsigned char)sizeof(IP_H221NONSTANDARD),
                           &appH221NonStd);
}

else
{
   gc_util_insert_parm_ref(&pParmBlock,
                           IPSET_NONSTANDARDDATA,
                           IPPARM_NONSTANDARDDATA_OBJID,
                           (unsigned char)(strlen(pOid)+1),
                           pOid);
}
```

4.3.4 Specifying Nonstandard Control Information (H.323)

To specify Nonstandard Control information to be included in the H.323 SETUP message, use the **gc_SetUserInfo( )** function with a **duration** parameter of GC_SINGLECALL to preset the information. If the **duration** parameter is set to GC_ALLCALLS, the function fails.

To specify Nonstandard Control data, the GC_PARM_BLK pointed by the **infoparmblkp** function must be set up with two parameter elements that use the IPSET_NONSTANDARDCONTROL parameter set ID. The first required parameter element specifies the Nonstandard Control data itself, and the second parameter element identifies the type of object identifier to use.

The maximum length of the Global Call parameter used for the Nonstandard Control information is configured at start-up via the max_parm_data_size field in the IPCCLIB_START_DATA structure. The default size is 255 (for backwards compatibility), but applications may configure it to be as large as 4096 bytes. Applications must use the extended **gc_util_..._ex( )** functions to insert or extract any GC_PARM_BLK parameter elements whose data length is defined to be greater than 255.

Note: In practice, applications may not be able to utilize the full maximum length of the nonstandard control parameter element as configured in max_parm_data_size. The H.323 stack limits the overall size of messages to be max_parm_data_size + 512 bytes, and any messages that exceed this limit are truncated without any notification to the application.

The parameter element for the Nonstandard Control data is:

IPSET_NONSTANDARDCONTROL
  IPPARM_NONSTANDARDDATA_DATA
  - value = Nonstandard Data string, max length = IPCCLIB_START_DATA.max_parm_data_size (configurable at library start-up)

The parameter element for the Nonstandard Control identifier is one (and only one) of the following:

IPSET_NONSTANDARDCONTROL
  IPPARM_NONSTANDARDDATA_OBJID
  - value = array of unsigned integers, max length = MAX_NS_PARM_OBJID_LENGTH

IPSET_NONSTANDARDCONTROL
  IPPARM_H221NONSTANDARD
  - value = IP_H221NONSTANDARD structure

See Section 9.2.18, "IPSET_NONSTANDARDCONTROL", on page 630 for more information.

The following code example shows how to set nonstandard data elements:

```
IP_H221NONSTANDARD appH221NonStd;
appH221NonStd.country_code = 181;
appH221NonStd.extension = 31;
appH221NonStd.manufacturer_code = 11;
char* pControl = "Control String";
char* pOid = "1 22 333 4444";
choiceOfNSControl = 1; /* App decides which type of object identifier to use */

/* setting NS Control */
gc_util_insert_parm_ref_ex(&pParmBlock,
                           IPSET_NONSTANDARDCONTROL,
                           IPPARM_NONSTANDARDDATA_DATA,
                           (unsigned long)(strlen(pControl)+1),
                           pControl);

   if (choiceOfNSControl)  /* App decide the CHOICE of OBJECTIDENTIFIER.
                              It cannot set both objid & h221 */
   {
      gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_NONSTANDARDCONTROL,
                              IPPARM_H221NONSTANDARD,
                              (unsingned char)sizeof(IP_H221NONSTANDARD),
                              &appH221NonStd);
   }

   else
   {
      gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_NONSTANDARDCONTROL,
                              IPPARM_NONSTANDARDDATA_OBJID,
                              (unsingned char)(strlen(pOid)+1),
                              pOid);
   }
```

4.3.5 Specifying IPv4 or IPv6 Address

The IPv6 addressing feature is not supported in Dialogic® HMP Software Release 3.0.

The IPPARM_SDP_IP_TYPE parameter in the IPSET_SDP parameter set is used to specify the SDP address type (IPv4 or IPv6 for RTP/RTCP addresses) in the SIP SDP offer/answer model. The default value is IPv4 addressing for backward compatibility. Use **gc_SetUserInfo( )** to specify arguments for a single call (GC_SINGLECALL) or for all calls (GC_ALLCALLS) on a line device. The **gc_SetConfigData( )** function is not used with this parameter.

IPPARM_SDP_IP_TYPE
  Specifies the IP address type to use in SDP:
  - USE_IPv4 – (Default) Only IPv4 addressing is accepted in incoming/outgoing SDP.
  - USE_IPv6 – Only IPv6 addressing is accepted in incoming SDP. To use this value, you must configure a valid local media IPv6 address as shown in the example below.

- PREFER_IPv6 – IPv6 addressing is used when sending an SDP offer. When receiving an SDP offer based on IPv4, the SIP stack will adapt itself and use IPv4 SDP for that connection. To use this value, you must configure a valid local media IPv6 address as shown in the example below.

To use IPv6 addressing, enable the feature in the IP_VIRTBOARD structure and configure it in the IP_ADDR structure.

Example

Before using the USE_IPv6 and PREFER_IPv6 values, you must configure a valid local media IPv6 address via the CLI **conf system hmp-rtp-address** command as shown in the following example. (See the *Dialogic® HMP Software for Linux Configuration Guide* for more information on the CLI.)

```
telnet localhost
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
+--------------------------------------------------------------------------+
|                                                                          |
|  .:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::. |
| :::    Dialogic(R) Host Media Processing Software Release 4.1 LIN    ::: |
|  `:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::' |
|                                                                          |
|                                                                          |
| For HELP:                                                                |
|   Use '?' at command prompt                                              |
|   Use '-h' within commands                                               |
|                                                                          |
+--------------------------------------------------------------------------+
Login :admin
Password :*****
CLI> conf system hmp-rtp-address 2001:db8:0:f101::2
updated
updated
CLI> show system
UNKNOWN RMS System
Name:          System Name Not Defined
Location:      System Location Not Defined
Contact:       System Contact Not Defined

Operational Status:     System is ready
Run Status:             Active
Media Last Request:     Start
Media Start Mode:       Manual
Software Media Support: Refer to license details

Uptime:               2 week(s) 1 day(s) 21 hour(s) 30 minute(s) 54 second(s)
Last Boot Reason:     Power on
Services:             7
Form Factor:          RMSModel:   Dialogic(R) Host Media Processing Software Release 4.1 LIN
Vendor:               Dialogic Research Inc.
Serial Number:        Undefined
HMP RTP IPv4 Address: 10.130.4.141
HMP RTP IPv6 Address: 2001:db8:0:f101::2
```

4.4 Connection Phase Messages

In either the SIP or H.323 protocol, a number of messages are exchanged in the connection phase, after one endpoint has initiated a call and before the connection is completed. The Dialogic® Global Call API library and the protocol stack handle most of these messages automatically, without any participation from the application. But the application is able to configure or access some of these messages as described in the following topics:

- Setting and Retrieving Disconnect Cause or Reason Values
- Setting Busy Reason Codes
- SIP Provisional (1xx) Responses
- SIP Redirection (3xx) Response Messages
- SIP Rejection Responses
- Configuring Proceeding Message Generation (H.323)

4.4.1 Setting and Retrieving Disconnect Cause or Reason Values

Use the **cause** parameter in the **gc_DropCall( )** function to specify a disconnect reason/cause to be sent to the remote endpoint.

Note: When using SIP, reasons are only supported when a call is disconnected while in the Offered state.

Use the **gc_ResultInfo( )** function to get the reason/cause of a GCEV_DISCONNECTED event. This reason/cause could be sent from the remote endpoint or it could be the result of an internal error.

IP-specific reason/cause values are specified in the eIP_EC_TYPE enumerator defined in the *gcip_defs.h* header file.

4.4.2 Setting Busy Reason Codes

Both SIP and H.323 define request response codes that can be included in the failure response messages that are sent when a local system cannot take additional incoming sessions. Global Call allows applications to set SIP and H.323 busy code values on a virtual board level.

SIP and H.323 busy codes are configured independently, and the configuration of each can be changed at any time. The busy codes are configured by calling **gc_SetConfigData( )** using the following parameter set ID and parameter ID:

- for SIP: IPSET_SIP_RESPONSE_CODE and IPPARM_BUSY_REASON; see Section 9.2.27, "IPSET_SIP_RESPONSE_CODE", on page 638.
- for H.323: IPSET_H323_RESPONSE_CODE and IPPARM_BUSY_CAUSE; see Section 9.2.8, "IPSET_H323_RESPONSE_CODE", on page 623.

4.4.2.1 Setting SIP Busy Code

For SIP, RFC3261 defines three applicable busy codes:

480 Temporarily Unavailable
: The callee’s end system was contacted successfully, but the callee is currently unavailable. For example, the callee may be not logged in, may be in a state that precludes communication, or may have activated the “do not disturb” feature. This busy code is also returned by a redirect or proxy server that recognizes the user identified by the Request-URI but does not currently have a valid forwarding location for that user.

486 Busy Here
: The callee’s end system was contacted successfully, but the callee is currently not willing or able to take additional calls at this end system. This response should be used if the user could be available elsewhere.

600 Busy Everywhere
: The callee’s end system was contacted successfully, but the callee is busy and does not wish to take the call at this time. This response should be used if the callee knows that no other end system will be available to accept this call.

By default, Global Call automatically responds with a 486 Busy Here when additional incoming call requests arrive after the maximum number of SIP calls per virtual board has been reached. A 480 Temporarily Unavailable or 600 Busy Everywhere reason code can be used instead of the 486 Busy Here if the application explicitly configures the busy code.

To configure the SIP busy reason code, call **gc_SetConfigData( )** with a GC_PARM_BLK that contains the following parameter element:

IPSET_SIP_RESPONSE_CODE
: IPPARM_BUSY_REASON
: Possible values:
  - IPEC_SIPReasonStatus480TemporarilyUnavailable
  - IPEC_SIPReasonStatus486BusyHere (default)
  - IPEC_SIPReasonStatus600BusyEverywhere

The following code snippet illustrates how to configure the SIP busy code:

```
#include "gclib.h"
   .
   .
/* configure SIP Busy Reason Code to 480 Temporarily Available */

GC_PARM_BLKP pParmBlock = NULL;

gc_util_set_insert_parm_val(&pParmBlock,
                            IPSET_SIP_RESPONSE_CODE,
                            IPPARM_BUSY_REASON,
                            sizeof(unsigned short),
                            IPEC_SIPReasonStatus480TemporarilyUnavailable);

gc_SetConfigData(GCTGT_CCLIB_NETIF, board, pParmBlock,
                 0, GCUPDATE_IMMEDIATE, &t, EV_ASYNC);

gc_util_delete_parm_blk(pParmBlock);
```

4.4.2.2 Setting H.323 Busy Code

ITU Recommendation Q.850 defines cause codes that are used for H.323. Among the applicable busy cause definitions are:

Cause 34: No circuit/channel available
  Indicates there is no appropriate circuit/channel currently available to handle the call.

Cause 47: Resource unavailable/unspecified
  Indicates the resource is unavailable when no other cause values in the resource class applies.

To configure the H.323 busy reason code, call **gc_SetConfigData( )** with a GC_PARM_BLK that contains the following parameter element:

IPSET_H323_RESPONSE_CODE
  IPPARM_BUSY_CAUSE
  Typical values:
  - IPEC_Q931Cause34NoCircuitChannelAvailable
  - IPEC_Q931Cause44RequestedCircuitChannelNotAvailable
  - IPEC_Q931Cause47ResourceUnavailableUnspecified

The following code snippet illustrates how to set the H.323 busy code:

```
#include "gclib.h"
   .
   .
/* configure H.323 Busy Reason Code to 34 -  "No Circuit/Channel Available" */

GC_PARM_BLKP pParmBlock = NULL;

gc_util_set_insert_parm_val(&pParmBlock,
                            IPSET_H323_RESPONSE_CODE,
                            IPPARM_BUSY_CAUSE,
                            sizeof(unsigned short),
                            IPEC_Q931Cause34NoCircuitChannelAvailable);

gc_SetConfigData(GCTGT_CCLIB_NETIF, board, pParmBlock,
                 0, GCUPDATE_IMMEDIATE, &t, EV_ASYNC);

gc_util_delete_parm_blk(pParmBlock);
```

4.4.3 SIP Provisional (1xx) Responses

RFC 3261 defines five provisional messages (also called informational messages) that may be sent to the calling party when the server at the called party is performing some further action and does not yet have a definitive response. One of these provisional messages, the 100 Trying message, is uniquely reported to the calling application via the maskable GCEV_PROCEEDING event type. The other four provisional messages, which have response codes in the 18x range, are all reported to the calling application via the same Global Call event type, GCEV_ALERTING. This section describes the mechanisms that Global Call provides to allow applications to differentiate among the 18x provisional responses, which include:

- 180 (Ringing)
- 181 (Call Is Being Forwarded)
- 182 (Queued)

- 183 (Session Progress)

Note: RFC 3261 indicates that the server for the called party may issue more than one 182 Queued response to update the caller about the status of the queued call, but the call control library only generates a GCEV_ALERTING event for the **first** 182 Queued response for a given call.

For all provisional messages, the primary content is the Status-Code in the response's Status-Line, and the technique for retrieving this information is described in Section 4.4.3.1, "Retrieving Status-Code for 18x Provisional Responses".

RFC 3261 specifies that 182 and 183 responses may optionally contain additional information about the call status in the Reason-Phrase of the message's Status-Line. The technique for retrieving this information is described in Section 4.4.3.2, "Retrieving Reason-Phrase from 182 and 183 Provisional Responses".

RFC 3261 also specifies that 183 responses can optionally contain more details about the call progress in message header fields or the message body. Applications can retrieve this information using the generic access mechanisms described in Section 4.9, "Setting and Retrieving SIP Message Header Fields", and Section 4.10, "Sending and Receiving MIME Bodies in SIP Messages (SIP-T)".

For related information, see also Section 4.21, "Receiving Multiple SIP 18x Provisional Responses", on page 287.

4.4.3.1 Retrieving Status-Code for 18x Provisional Responses

When using SIP, each GCEV_ALERTING event will have an associated GC_PARM_BLK that contains the specific status code for the 18x provisional response message in a parameter element of the following type:

IPSET_SIP_RESPOSNSE_CODE
  IPPARM_RECEIVED_RESPONSE_STATUS_CODE
  - value = 3-digit integer retrieved as Status-Code from Status-Line of the received provisional message

4.4.3.2 Retrieving Reason-Phrase from 182 and 183 Provisional Responses

The mechanism provided for retrieving the Reason-Phrase for 182 and 183 provisional response messages is an extension of the generic mechanism for accessing SIP header fields, as described in Section 4.9, "Setting and Retrieving SIP Message Header Fields", even though the Reason-Phrase is not technically a header field.

Applications must first register to receive the Reason-Phrase, using the same technique that is detailed in Section 4.9.2, "Enabling Access to SIP Header Information", on page 181. This registration only needs to be performed once for a board device, and may be performed at any time during the life of an application.

To register to receive the Reason-Phrase, the application first constructs a GC_PARM_BLK that contains the following element:

IPSET_CONFIG
  IPPARM_REGISTER_SIP_HEADER
  - value = "Reason-Phrase"

The application then calls **gc_SetConfigData( )** with this GC_PARM_BLK to register for reception of all the header fields that are identified in the parameter block.

When the Dialogic® Global Call API library receives a 182 or 183 provisional response, it generates a a GCEV_ALERTING event that has an associated GC_PARM_BLK to contain extra data about the event. If the application has previously registered to receive the Reason-Phrase, this GC_PARM_BLK will contain a parameter element as follows:

IPSET_SIP_MSGINFO
  IPPARM_SIP_HDR
  - value = NULL-terminated string which begins with the string “Reason-Phrase:”

Note: Depending on the list of header fields that the application has registered to receive, the GC_PARM_BLK associated with the GCEV_ALERTING event may contain multiple parameter elements that use the IPSET_SIP_MSGINFO / IPPARM_SIP_HDR ID pair. It is the application’s responsibility to parse the value strings of these parameter elements to identify the one that begins with the “Reason-Phrase:” string.

4.4.4 SIP Redirection (3xx) Response Messages

RFC 3261 defines the 3xx range of responses as redirection messages, which can be used by the called party’s server to push alternative routing information back to the originator of an INVITE request. This allows the server to provide information that is useful in locating the target of the request while also taking itself out of the loop for further messaging for the transaction. When the originator of the INVITE request receives a 3xx response, it cancels the original request and issues one or more new requests based on the URI(s) and transport parameters contained in the response.

The supported redirection status codes include:

- 301 (Moved Permanently)
- 302 (Moved Temporarily)
- 305 (Use Proxy)

4.4.4.1 Redirecting an Incoming Call

To redirect an incoming call, the application first prepares a CG_PARM_BLK that contains the alternative contact information to be sent to the originator in the Contact header, then calls **gc_SetUserInfo( )** to set the parameters for the next message. After the parameters are set the application calls **gc_DropCall( )** for the CRN to send the 3xx response; the specific response code that is used is specified via the **cause** parameter using the IPEC_SIPReasonStatus3xx values that are defined in *gcip_defs.h*.

When preparing the parameter block for a redirection response, the application inserts one or more of the following parameter elements into a GC_PARM_BLK:

IPSET_SIP_MSGINFO
  IPPARM_SIP_HDR
  - value = complete Contact header string, starting with “Contact:”

Note: The use of the deprecated IPSET_SIP_MSGINFO / IPPARM_CONTACT_URI parameter ID pair is not recommended because this ID pair only provides access to the URI portion of the Contact header (i.e., without the display string and any parameters), and can only set a single URI. If the GC_PARM_BLK contains one or more IPSET_SIP_MSGINFO / IPPARM_SIP_HDR parameter elements, any element using IPSET_SIP_MSGINFO / IPPARM_CONTACT_URI will be ignored.

If any specific Contact string being set by the application is longer than 255 bytes, the application must use the extended **gc_util_insert_parm_ref_ex( )** function; if the data is less than 255 bytes in length, either **gc_util_insert_parm_ref( )** or **gc_util_insert_parm_ref_ex( )** can be used.

If the application sets more than one Contact header parameter element in the GC_PARM_BLK, the call control library automatically combines then into a single Contact header in a comma-separated value list that reflects the order in which the application specified the separate Contact headers.

RFC 3261 provides detailed information about rules and restrictions for Contact header fields in redirection responses, but a few basic rules are presented here for convenience:

- The Contact header field contains URIs that specify new locations, new user names, or additional transport parameters.
- None of the URIs in the Contact header field can be equal to the one in the Request-URI.
- For a 301 or 302, the response may contain the same location and username that was targeted in the original request, but additional transport parameters to try, such as a different multicast address or a different transport protocol.
- A Contact header field can point to a different resource than the one originally called, and can use any suitable URI (not just SIP URIs).
- Each Contact header field can include an “expires” parameter to indicate how long the URI is valid (in seconds). If this parameter is not provided, the value of the Expires header field determines the length of the validity.

The following code example shows how an application can set two alternative URIs to send in a 302 Moved Temporarily response.

```
void redirectChannel(int channel)
{
   char contact1[] = "Contact: \"forward1\" <sip:forward1@146.152.84.124>;q=0.7;expires=3600";
   char contact2[] = "Contact: \"forward2\" <sip:forward2@146.152.84.124>;q=0.5;expires=60";

   //Set contact header

   GC_PARM_BLKP pParmBlock = NULL;

   gc_util_insert_parm_ref_ex(&pParmBlock,
                              IPSET_SIP_MSGINFO,
                              IPPARM_SIP_HDR,
                              (unsigned long) (strlen(contact1)+1),
                              contact1);
   gc_util_insert_parm_ref_ex(&pParmBlock,
                              IPSET_SIP_MSGINFO,
                              IPPARM_SIP_HDR,
                              (unsigned long) (strlen(contact2)+1),
                              contact2);

   int frc = gc_SetUserInfo(GCTGT_GCLIB_CRN, session[channel].crn,pParmBlock,GC_SINGLECALL);
   if(GC_SUCCESS != frc)
   {
      printf("[%d] gc_SetUserInfo failed\n",channel);
      gc_util_delete_parm_blk(pParmBlock);
      return;
   }

   int rc = gc_DropCall(session[channel].crn,
                        IPEC_SIPReasonStatus302MovedTemporarily,
                        EV_ASYNC);
   if(GC_SUCCESS != rc)
   {
      printf("[%d] gc_DropCall failed \n",channel);
      return;
   }
}
```

The SIP message sent by in this example would look something like the following:

```
SIP/2.0 302 Moved Temporarily
From: HMP-From<sip:146.152.84.1:5060>;tag=52a52b0-0-13c4-28795-17aef347-28795
To: HMP-To<sip:146.152.84.2>;tag=52a5468-0-13c4-28795-783983a2-28795;myname
Call-ID: 52ebbf8-0-13c4-28795-14daf9c6-28795@146.152.84.1
CSeq: 1 INVITE
Via: SIP/2.0/UDP 146.152.84.1:5060;received=146.152.84.2;branch=z9hG4bK-28795-9e19f19-554d9dc4
Supported: replaces
Contact: "forward1" <sip:forward1@146.152.84.124>;q=0.7;expires=3600,"forward2"
 <sip:forward2@146.152.84.124>;q=0.5;expires=60
Content-Length: 0
```

4.4.4.2 Receiving and Handling a Redirect Response

After receiving a GCEV_DISCONNECTED event, the application can check the cause of the event. If the disconnection was because of call redirection, the application can further check the extra data associated with the event for redirect URIs in the form of a Contact header contained in an IPSET_SIP_MSGINFO/IPPARM_SIP_HDR parameter element. After completing the drop call on this channel, the application can make a new call to any of the redirect URIs if it wishes.

According to RFC 3261, applications receiving a 3xx response have great latitude in determining how (or whether) to generate new requests to the redirect URIs. An application can choose which of the suggested URIs to add to its target list, and in what order to add them. The application may generate new requests to the URIs in the target list serially or in parallel. If a new request fails (receives a result code greater than 399), the application should try the next URI in the target list until the call succeeds or until all URIs have produced a failure result. If any of the redirected requests produces a 3xx redirect response, the application can choose to add to its target list any of the URIs that are contained in the 3xx response as long as the URI is not already in the target list.

RFC 3261 recommends that the new requests use the same To, From, and Call-ID used in the original, redirected request, but the application may update the Call-ID if it wishes.

In the following example, the parser assumes the redirect URI is in <> and only returns the first URI in the Contact header.

```
void processEvtHandler()
{
   METAEVENT     metaEvent;
   GC_PARM_BLK  *parmblkp = NULL;
   GC_PARM_DATAPt_gcParmDatap = NULL;

   .
   .
   .
   switch (evtType)
   {
      case GCEV_DISCONNECTED:
      /* check for call redirection */
      if(true == checkCallRedirected())
      {
         parmblkp = (GC_PARM_BLK *) metaEvent.extevtdatap;
         while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
         {
            switch(t_gcParmDatap->set_ID)
            {
               case IPSET_SIP_MSGINFO:
                  switch(t_gcParmDatap->parm_ID)
                  {
                     case IPPARM_SIP_HDR:
                        /* check for first contact URI */
                        Char* addr = checkRedirectedAddress(t_gcParmDatap);
                        if(NULL != addr)
                        {
                           printf("Redirect URI is %s",addr);
                        }
                        break;
                  }
                  break;
            }
         }
      }

      /* continue drop call on this channel */
      .
      .
      .
   }
   .
   .
   .
}

bool checkCallRedirected()
{
   int gcError;    /* GlobalCall Error */
   int ccLibId;    /* CC Library ID */
   long ccError = 0;   /* Call Control Library error code */
   char *GCerrMsg; /* GC pointer to error message string *
   char *errMsg;   /* CCLIB pointer to error message string */

   if(gc_ResultValue( &g_ClaimedMetaEvent, &gcError, &ccLibId, &ccError) == GC_SUCCESS)
   {
      gc_ResultMsg(LIBID_GC, (long) gcError, &GCerrMsg);
      gc_ResultMsg(ccLibId, ccError, &errMsg);
      printf("GC (%d) %s,CC (%ld) %s\n",,gcError,GCerrMsg,ccError,errMsg);
      /check for redirection
      if(IPEC_SIPReasonStatus300MultipleChoices <= ccError &&
          ccError < IPEC_SIPReasonStatus400BadRequest)
      {
         printf("Call is redirected\n");
         return true;
      }
      else
      {
         return false;
      }
   }
   return false;
}

/* Get only the first address in <> */
char* checkRedirectedAddress(GC_PARM_DATA *parmp)
{
   char* ptr;
   char* SipHeaderData=(char*)parmp->value_buf;
   char* HeaderName = NULL;
   char* HeaderData = NULL;
   char* redirectURI = NULL;
   ULONG HeaderDataSize = 0;
   ptr = strchr(SipHeaderData,':');

   if (ptr)
   {
      ptr[0] = '\0';
      HeaderName = SipHeaderData;
      HeaderData = ptr + sizeof(char);
      HeaderDataSize = parmp->value_size - (strlen(HeaderName) + 1);
   }

   if ( HeaderName != NULL &&
        0==_stricmp(HeaderName,"contact") &&
        (HeaderData != NULL)  &&
        (HeaderDataSize != 0) )
   {
      ptr = strchr(HeaderData,'<');
      redirectURI=ptr+sizeof(char);
      ptr = strchr(HeaderData,'>');
      ptr[0]='\0';
      return redirectURI;
   }
   else
   {
      return NULL;
   }
}
```

4.4.5 SIP Rejection Responses

Note: The information in this section only applies when the Dialogic® Global Call IP Call Control library is started in the first party call control (1PCC) operating mode. The capability described in this section is not supported when the library is started in the third party call control (3PCC) operating mode.

There are several circumstances in which the Dialogic® Global Call API library automatically rejects an initial call offer or a subsequent media session proposal.

If the library receives an initial INVITE request that contains an SDP offer which does not specify a codec that is supported on the local media platform, the library automatically sends a 488 (Not Acceptable Here) response. No notification is sent to the application in this situation because no dialog has been established.

If an SDP offer that does not specify a supported codec is received in a re-INVITE request rather than an initial INVITE, the library once again automatically rejects the offer with a 488 (Not Acceptable Here). But in this case the library *does* notify the application (form informational purposes only) with a GCEV_REQ_MODIFY_UNSUPPORTED event.

If the library receives an multimedia SDP offer (i.e., an SDP offer that includes a video media descriptor), the default behavior is to accept the audio portion of the offer (assuming that it is acceptable) but reject the video portion. This is accomplished by setting a port number of 0 in the video media descriptor in the SDP answer that is sent in the 200 OK response.

Applications can optionally configure the library to use an alternative response to SDP offers containing video media descriptors. When the application enables this alternative response, the library automatically rejects an INVITE or re-INVITE that requests a video media session with a 488 (Not Acceptable Here) response. If the SDP offer proposing a video session is contained in a 200 OK rather than an INVITE, the library will ACK the 200 OK but then immediately terminate the call with BYE.

This alternative response to media session proposals is enabled using the **gc_SetConfigData( )** function, passing it a GC_PARM_BLK that includes the following parameter:

IPSET_CONFIG
  IPPARM_1PCC_REJECT_VIDEO
  - value = not used

Once this alternative rejection mode has been enabled, the setting remains in effect until the library is stopped and the application exits.

The following code snippet illustrates how applications would enable the optional rejection mode:

```
gc_util_insert_parm_val(&pParmBlock, IPSET_CONFIG, IPPARM_1PCC_REJECT_VIDEO, sizeof(int), 0);
long t = 0;
int  rc = gc_SetConfigData(GCTGT_CCLIB_NETIF,
                           boarddev,
                           pParmBlock,
                           0,
                           GCUPDATE_IMMEDIATE,
                           &t,
                           EV_ASYNC);
```

4.4.6 Configuring Proceeding Message Generation (H.323)

When using the H.323 protocol, the application can configure whether the Proceeding message is sent under application control (using the **gc_CallAck( )** function) or automatically by the stack. The default behavior is for the stack to send Proceeding automatically.

The generation of the Proceeding message is configured using the **gc_SetConfigData( )** function. To configure the generation of the Proceeding message, the GC_PARM_BLK that is passed to the function must contain the following parameter element:

GCSET_CALL_CONFIG
  GCPARM_CALLPROC
  Possible values:
  - GCCONTROL_APP – The application must use **gc_CallAck( )** to send the Proceeding message. This is the default.
  - GCCONTROL_TCCL – The stack sends the Proceeding message automatically.

4.5 Call Transfer

The Dialogic® Global Call API library provides six APIs specifically for call transfer in the IP technology. These APIs are described in the *Dialogic® Global Call API Library Reference* with protocol-specific variances described in the subsections of Section 8.3, "Dialogic® Global Call API Function Variances for IP". This section describes general considerations for implementing call transfer as well as details specific to H.450.2 (part of the H.323 protocol suite) and SIP protocols. For H.450.2-specific call transfer scenarios see Section 3.2, "Call Transfer Scenarios When Using H.323", on page 49, and for SIP-specific call transfer scenarios, see Section 3.3, "Call Transfer Scenarios When Using SIP", on page 66. The topics covered here include:

- Enabling Call Transfer
- Dialogic® Global Call API Line Devices for Call Transfer
- Incoming Transferred Call
- Call Transfer Glare Condition
- Call Transfer When Using SIP

4.5.1 Enabling Call Transfer

The call transfer supplementary service is a feature that must be enabled at the time the **gc_Start( )** function is called. Both H.450.2 and SIP call transfer services are enabled at the same time. If the application tries to use one of the six IP call transfer functions when call transfer was not enabled, the function call fails with an IPERR_SUP_SERV_DISABLED indication.

The mandatory **INIT_IP_VIRTBOARD( )** function populates the IP_VIRTBOARD structure with default values. The default value of the sup_serv_mask field in the initialized structure disables the call transfer service for both H.323 and SIP protocols. The default sup_serv_mask field value must therefore be overridden with the value IP_SUP_SERV_CALL_XFER for each IPT board device on which call transfer is to be enabled. The following code snippet provides an example for two virtual boards:

```
.
.
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
ip_virtboard[1].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
.
.
```

Note: Features that are enabled or configured via the IP_VIRTBOARD structure cannot be disabled or reconfigured once the library has been started. All items set in this data structure take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

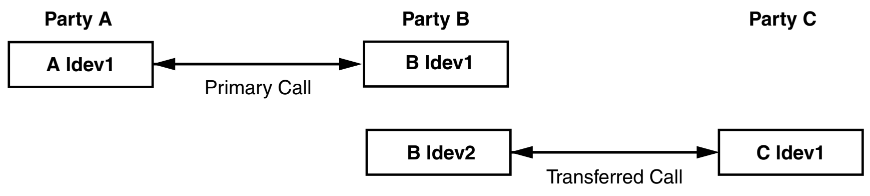
4.5.2 Dialogic® Global Call API Line Devices for Call Transfer

The Dialogic® Global Call API IP architecture is designed so that each RTP transcoder at all times is streaming (xmit and rcv) with only one other endpoint. In order to support call transfers, two Global Call line devices are required at some or all of the endpoints. And because all involved call handles must be on the same stack instance, the following limitations are imposed on call transfers:

- When performing an attended call transfer at party A, both the consultation line device and the transferring line device must be on the same virtual board.
- When performing a call transfer (either attended or unattended) at party B, both the transferring line device and the transferred line device must be on the same virtual board.
- When performing an attended call transfer at party C, both the consultation line device and the transferred-to line device must be on the same virtual board.

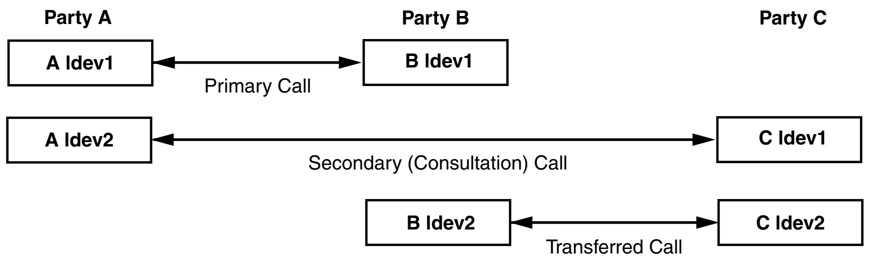
To support blind call transfer, two Dialogic® Global Call API line devices are required at the transferred (party B) endpoint, one for the primary call with the transferring (party A) endpoint and a second to initiate the transferred call to the transferred-to (party C) endpoint. See Figure 47.

Figure 47. Global Call Devices for H.450.2 Blind Call Transfer or SIP Unattended Transfer



To support a successful H.450.2 supervised call transfer or SIP attended call transfer, two Dialogic® Global Call API line devices are eventually utilized at all endpoints. The transferring endpoint or transferor (party A) makes a consultation call to the transferred-to endpoint or transfer target (party C), thus utilizing two line devices at both these endpoints as well. See Figure 48.

Figure 48. Global Call Devices for Supervised Call Transfer



4.5.3 Incoming Transferred Call

The incoming transferred call to party C contains the call control library (CCLIB) cause value of IPEC_IncomingTransfer and a Dialogic® Global Call API library (GC LIB) cause value of GCRV_XFERCALL. The **gc_ResultInfo( )** function can be used to retrieve these values.

In the case of supervised transfer, the associated CRN of the secondary/consultation call is provided. The secondary CRN can be accessed via the extevtdatap pointer within the METAEVENT structure of the GCEV_OFFERED event which references a GC_PARM_BLK. From this parameter block, a data element identified by the SetId/ParmId pair of GCSET_SUPP_XFER and GCPARM_SECONDARYCALL_CRN can be retrieved via the parameter block utility functions to retrieve the secondary call CRN, which is of datatype size CRN (long).

If the transferee address is also provided to party C (optional for H.450.2), it can also be retrieved from this same parameter block, via a data element identified by the SetId/ParmId pair of GCSET_SUPP_XFER and GCPARM_TRANSFERRING_ADDR via the parameter block utility functions as a character array of maximum size GC_ADDRSIZE.

The following code sample demonstrates how to implement this:

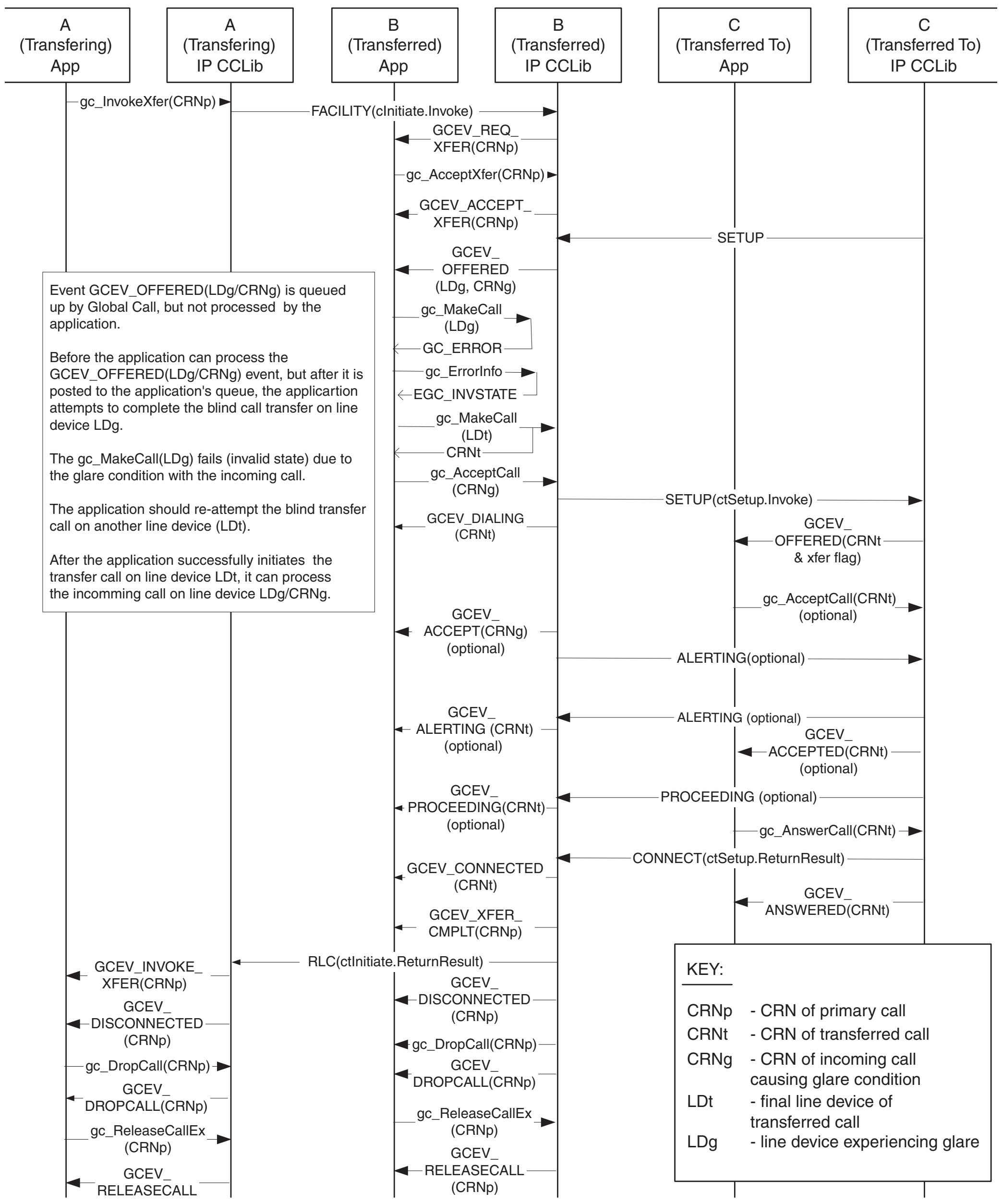
```
   .
   .
   .
case GCEV_OFFERED:
{
   if (metaevent.extevtdatap)
   {
      GC_PARM_BLKP parm_blkp = metaevent.extevtdatap;
      GC_PARM_DATAP curParm = NULL;
      printf("GCEV_OFFERED has parmblk:\n");
      while ((curParm = gc_util_next_parm(parm_blkp, curParm)) != NULL)
      {
         CRN secondaryCRN = 0;
         char transferringAddr[GC_ADDRSIZE];
         printf("SetID: 0x%x  ParmID: 0x%x\n",curParm->set_ID,curParm->parm_ID);

         switch (curParm->parm_ID)
         {
            case GCPARM_SECONDARYCALL_CRN:
               memcpy(&secondaryCRN, curParm->value_buf, curParm->value_size);
               printf("GCPARM_SECONDARYCALL_CRN: 0x%x\n",secondaryCRN);
               break;
            case GCPARM_TRANSFERRING_ADDR:
               memcpy(transferringAddr, curParm->value_buf, curParm->value_size);
               printf("GCPARM_TRANSFERRING_ADDR: %s\n",transferringAddr);
               break;

            default:
               printf("UNEXPECTED PARM_ID: %d\n",curParm->parm_ID);
               break;
         }
      }
   }
break;
.
.
.
```

4.5.4 Call Transfer Glare Condition

Glare can occur on a line device during both blind and supervised call transfer operations. Glare occurs on a line device during call transfer at Party B when the application calls **gc_MakeCall( )** to establish the transferred call (after the application has called **gc_AcceptXfer( )** on the primary CRN). Glare occurs because the CCLIB IP library has chosen the same line device for an incoming call that the application has chosen for establishing the transferred call. The application indication that this glare condition has occurred is that **gc_MakeCall( )** fails with an error indication of EGC_INVSTATE, GCRV_GLARE, or EGC_ILLSTATE. The application should retry the transferred call establishment request on another “available” line device. The application should process the GCEV_OFFERED metaevent on the incoming call/line device that caused the glare “normally” when it is retrieved. The call scenario in Figure 49 describes the glare condition and the appropriate application response.

Figure 49. Call Transfer Glare Condition

Precondition: Primary call between A and B is connected (not shown).



Event GCEV_OFFERED(LDg/CRNg) is queued up by Global Call, but not processed by the application.

Before the application can process the GCEV_OFFERED(LDg/CRNg) event, but after it is posted to the application's queue, the applicartion attempts to complete the blind call transfer on line device LDg.

The gc_MakeCall(LDg) fails (invalid state) due to the glare condition with the incoming call.

The application should re-attempt the blind transfer call on another line device (LDt).

After the application successfully initiates the transfer call on line device LDt, it can process the incomming call on line device LDg/CRNg.

KEY:

CRNp	- CRN of primary call
CRNt	- CRN of transferred call
CRNg	- CRN of incoming call causing glare condition
LDt	- final line device of transferred call
LDg	- line device experiencing glare

Post Condition: Transferred call between B and C completed. Primary call between A and B is dropped and released. Incoming call that causes glare is ringing.

4.5.5 Call Transfer When Using SIP

This section describes specific call transfer procedures when using SIP protocol. For complete SIP-specific call transfer scenarios see Section 3.3, "Call Transfer Scenarios When Using SIP", on page 66. The topics covered here include:

- Enabling GCEV_INVOKE_XFER_ACCEPTED Events
- Invoking an Unattended Call Transfer
- Invoking an Attended Call Transfer
- Processing Asynchronous Call Transfer Events
- Handling a Transfer Request
- Making a Transferred Call

4.5.5.1 Enabling GCEV_INVOKE_XFER_ACCEPTED Events

The following code snippet illustrates how to enable the GCEV_INVOKE_XFER_ACCEPTED event type, which is optionally used to notify the application at party A that party B has accepted a transfer request. This event type is disabled by default. This event can be enabled for an individual line device at any time after the line device is opened. The event is enabled in the party A (Transferor) application, and need only be enabled if the application wishes to receive the events. Note that there is no equivalent event in H.450.2.

```
//enable GCEV_INVOKE_XFER_ACCEPTED event

GC_PARM_BLK *t_pParmBlk = NULL;
long request_id;

gc_util_insert_parm_val(&t_pParmBlk, GCSET_CALLEVENT_MSK, GCACT_ADDMSK,
                        sizeof(long), GCMSK_INVOKEXFER_ACCEPTED);

gc_SetConfigData(GCTGT_GCLIB_CHAN,ldev,t_pParmBlk, 0, GCUPDATE_IMMEDIATE, &request_id, EV_SYNC);

gc_util_delete_parm_blk(t_pParmBlk);
```

Disabling the event is done in exactly the same way except that the parameter ID that is set in the GC_PARM_BLK would be GCACT_SUBMSK instead of GCACT_ADDMSK.

4.5.5.2 Invoking an Unattended Call Transfer

The following code snippet illustrates how to invoke an unattended (blind) transfer on a channel that is in the connected state. In this example, the Refer-To header field of the REFER message that is sent is set to "sip:500@192.168.1.10", while the Referred-By header field is automatically populated by Global Call.

```
int Gc_InvokeXfer(int channel)
{
   INT32  rc;
   GCLIB_MAKECALL_BLK t_gclibmakecallblk;
   GC_MAKECALL_BLK    t_gcmakecallblk = {0};
   char invokeaddr[] = "192.168.1.10";  // party C (TRTSE)
   char phonelist[] = "500";
   /* Invoke transfer */
   memset(&t_gclibmakecallblk, 0, sizeof(GCLIB_MAKECALL_BLK));
   strcpy(t_gclibmakecallblk.destination.address, invokeaddr);
   t_gclibmakecallblk.destination.address_type = GCADDRTYPE_IP;
   t_gclibmakecallblk.destination.address_plan = GCADDRPLAN_UNKNOWN;
   t_gcmakecallblk.gclib = &t_gclibmakecallblk;

   gc_util_insert_parm_ref(&t_pParmBlk, IPSET_CALLINFO, IPPARM_PHONELIST,
                           sizeof(phonelist), phonelist);

   t_gclibmakecallblk.ext_datap = t_pParmBlk;

   rc = gc_InvokeXfer(session[channel].crn, 0, 0, &t_gcmakecallblk, 0, EV_ASYNC);

   gc_util_delete_parm_blk(t_pParmBlk);

   if(GC_SUCCESS != rc)
   {
      printf("GC_APP : [%d] Invoke Xfer failed!!!\n",channel);
      return GC_ERROR;
   }

  return GC_SUCCESS;
}
```

4.5.5.3 Invoking an Attended Call Transfer

Note that it is necessary for the consultation call to be in the connected state at **both** parties before the transfer operation is invoked. If the transferred-to party (party C) is a Global Call application and is not in the connected state when the transfer is invoked, it may fail to receive the Global Call event for the transfer request, which will cause a GCEV_TASKFAIL.

The following code snippet illustrates how a party that is connected to two remote parties, a primary call and a secondary call, invokes a call transfer by sending a REFER to one of the remote parties. The Refer-To, Replaces, and Referred-By header fields in the REFER are automatically filled in by Global Call. Note that the application does not have to specify the Refer-To information in an attended transfer because the secondary call already contains that information.

```
int Gc_InvokeXfer(int primaryChannel, int secondaryChannel)
{
   INT32  rc;

   /* Invoke transfer */
   rc = gc_InvokeXfer(session[primaryChannel].crn, session[secondaryChannel].crn,
                      0, 0, 0, EV_ASYNC);

   if(GC_SUCCESS != rc)
   {
      printf("GC_APP : [%d] Invoke Xfer failed!!!\n",primaryChannel);
      return GC_ERROR;
   }

   return GC_SUCCESS;
}
```

4.5.5.4 Processing Asynchronous Call Transfer Events

The following code snippets illustrate how to handle the asynchronous events that notify applications of the call transfer status as a SIP call transfer proceeds.

```
INT32 processEvtHandler()
{
   METAEVENT    metaEvent;
   GC_PARM_BLK  *parmblkp = NULL;
   :

   int  rc = gc_GetMetaEvent(&metaEvent);
   if (GC_SUCCESS != rc)
   {
      printf("GC_APP : gc_GetMetaEvent() failed\n");
      return rc;
   }

   long evtType = sr_getevttype();
   long evtDev = sr_getevtdev();
   int  g_extIndex = g_lArray[g_evtdev];

   switch (evtType)
   {

      //////////////////////////////////////////
      // Party A events
      //////////////////////////////////////////

      case GCEV_INVOKE_XFER_ACCEPTED:
         // remote party has accepted REFER by 2xx response
         printf("Invoke Transfer Accepted By Remote\n");
         break;

      case GCEV_INVOKE_XFER:
         // remote party has notified transfer success in NOTIFY
         printf("Invoke Transfer Successful\n");
         break;

      case GCEV_INVOKE_XFER_FAIL:
         // Invoke Transfer failed by remote NOTIFY or locally
         PrintEventError(&metaEvent);
         break;

      case GCEV_INVOKE_XFER_REJ:
         // Invoke Transfer Rejected by Remote party
         PrintEventError(&metaEvent);
         break;

      ////////////////////////////////////////
      // Party B events
      ////////////////////////////////////////

      case GCEV_REQ_XFER:
         // Incoming transfer request
         GC_REROUTING_INFO *pRerouteInfo = (GC_REROUTING_INFO *)metaEvent.extevtdatap;
         printf("Reroute number = %s\n", pRerouteInfo->rerouting_num);

         if(NULL != pRerouteInfo->parm_blkp)
         {
            // Handle parm blocks
         }

         strcpy(session[g_extIndex].rerouting_num,pRerouteInfo->rerouting_num);
         session[g_extIndex].rerouting_addrblk = *pRerouteInfo->rerouting_addrblkp;

         GC_HandleXferReq(g_extIndex)
         break;

      case GCEV_ACCEPT_XFER:
         // Accepted incoming transfer request
         break;
      case GCEV_ACCEPT_XFER_FAIL:
         // Failed to accept incoming transfer request
         PrintEventError(&metaEvent);
         break;

      case GCEV_REJ_XFER:
         // Rejected incoming transfer request
         break;

      case GCEV_REJ_XFER_FAIL:
         // Failed to reject incoming transfer request
         PrintEventError(&metaEvent);
         break;

      case GCEV_XFER_CMPLT:
         // completed transferred call
         break;

      case GCEV_XFER_FAIL:
         // Failed to complete the transferred call
         PrintEventError(&metaEvent);
         break;

      /////////////////////////////////////////
      // Party C events
      /////////////////////////////////////////

      case GCEV_OFFERED:
         // Received incoming call
         // Normall incoming call handling
         ...
         break;

      ...
   }
   ...
}



void PrintEventError(METAEVENT* pEvent, long evtDev)
{
   int gcError;     /* GlobalCall Error */
   int ccLibId;     /* CC Library ID */
   long ccError;    /* Call Control Library error code */
   char *GCerrMsg;  /* GC pointer to error message string */
   char *errMsg;    /* CCLIB pointer to error message string */

   if(gc_ResultValue(pEvent, &gcError, &ccLibId, &ccError)    == GC_SUCCESS)
   {
      gc_ResultMsg(LIBID_GC, (long) gcError, &GCerrMsg);
      gc_ResultMsg(ccLibId, ccError, &errMsg);

      printf("Ld 0x%lx, GC (%d) %s, CC (%ld) %s, (%s)\n",
             evtDev, gcError, GCerrMsg, ccError, errMsg, ATDV_NAMEP(evtDev));
   }
}
```

4.5.5.5 Handling a Transfer Request

The following code snippet illustrates how party B handles an incoming transfer request (REFER). Party B can either reject the request or accept it. Note that if no rejection reason is specified, the default reason, 603 Decline, is used.

```
int Gc_HandleXferReq(int channel)
{
   if(session[channel].ConfigFileParm.autoRejectCallXfer)
   {
      printf("GC_APP : [%d] Reject call xfer request\n",channel);
      if(GC_SUCCESS != gc_RejectXfer(session[channel].crn, IPEC_SIPReasonStatus502BadGateway,
                                   0, EV_ASYNC))
      {
         printf("GC_APP : [%d] Reject call xfer failed on device 0x%lx\n", channel,
                session[channel].ldev);
         PrintEventError(g_evtdev);
         return GC_ERROR;
      }
   }
   else
   {
      printf("GC_APP : [%d] Accept call xfer request\n",channel);
      if(GC_SUCCESS != gc_AcceptXfer(session[channel].crn, 0, EV_ASYNC))
      {
         printf("GC_APP : [%d] Accept call xfer failed on device 0x%lx\n", channel,
                 session[channel].ldev);
         PrintEventError(g_evtdev);
         return GC_ERROR;
      }
   }

   return GC_SUCCESS;
}
```

4.5.5.6 Making a Transferred Call

The following code snippet illustrates how party B makes the transferred call to party C after accepting transfer request from party A

```
int Gc_MakeXferCall(int channelPrimary, int channelXfer)
{

   GC_PARM_BLK          * t_pParmBlk = NULL;
   GCLIB_MAKECALL_BLK  t_gclibmakecallblk ;
   GC_MAKECALL_BLK     t_gcmakecallblk = {0};
   t_gcmakecallblk.gclib = &t_gclibmakecallblk;
   int                 channelXfer;

   memset(&t_gclibmakecallblk, 0, sizeof(GCLIB_MAKECALL_BLK));

   gc_util_insert_parm_val(&t_pParmBlk, GCSET_SUPP_XFER, GCPARM_PRIMARYCALL_CRN,
                           sizeof(unsigned long), session[channelPrimary].crn);

   t_gclibmakecallblk.ext_datap = t_pParmBlk;
   t_gclibmakecallblk.destination = session[channelPrimary].rerouting_addrblk;

   int frc = gc_MakeCall(session[channelXfer].ldev, &session[channelXfer].crn,
                         NULL, &t_gcmakecallblk, 0, EV_ASYNC);

   if((GC_SUCCESS != frc) ||(0 == session[channelXfer].crn))
   {
      printf("GC_APP : [%d] Gc_MakeCall failed: : crn 0x%lx\n", channelXfer,
              session[channelXfer].crn);
      PrintGCError(session[channelXfer].ldev);
   }

   gc_util_delete_parm_blk(t_pParmBlk);

   return GC_SUCCESS;
}
```

4.6 Retrieving Current Call-Related Information

To support large numbers of channels, the Dialogic® Global Call API library must perform all operations in asynchronous mode. To support this, an extension function variant allows the retrieval of a parameter as an asynchronous operation.

The retrieval of call-related information is a four step process:

1. Set up a GC_PARM_BLK that identifies which information is to be retrieved. The GC_PARM_BLK includes GC_PARM_DATA blocks. The GC_PARM_DATA blocks specify only the Set_ID and Parm_ID fields, that is, the value_size field is set to 0. The list of GC_PARM_DATA blocks indicate to the call control library the parameters to be retrieved.
2. Use the **gc_Extension( )** function to request the data. The parameters for this call should be specified as follows:
   - **target_type** should be GCTGT_GCLIB_CRN
   - **target_id** should be the actual CRN
   - **ext_id** (extension ID) should be set to IPEXTID_GETINFO
   - **parmblkp** should point to the GC_PARM_BLK set up in step 1
   - **mode** should be set to EV_ASYNC (asynchronous)
3. A GCEV_EXTENSIONCMPLT event is generated in response to the **gc_Extension( )** request. The extevtdatap field in the METAEVENT structure for the GCEV_EXTENSIONCMPLT event is a pointer to an EXTENSIONEVTBLK structure that contains a GC_PARM_BLK with the requested call-related information.
4. Extract the information from the GC_PARM_BLK associated with the GCEV_EXTENSIONCMPLT event. In this case, the GC_PARM_BLK contains real data; that is, the value_size field is not 0, and includes the size of the data following for each parameter requested.

Note: When an application on H.323 is using **gc_Extension( )** to extract information from a GCEV_OFFERED event, the application must ensure that it acknowledges the call within 8 seconds to prevent the offering side from timing out. The timer can be extended by sending PROCEEDING (by calling **gc_CallAck( )**) or ALERTING (by calling **gc_AcceptCall( )**) before extracting the information.

Table 4 shows the parameters that can be retrieved and when the information should be retrieved. The table also identifies which information can be retrieved when using H.323 and which information can be retrieved using SIP. For more information on individual parameters, refer to the corresponding parameter set reference section in Chapter 9, “IP-Specific Parameters”.

Table 4. Retrievable Call Information

Parameter	Set ID and Parameter ID(s)	When Information Can Be Retrieved	Datatype in value_buf Field (see Note 1)	SIP/ H.323
Call ID	IPSET_CALLINFO • IPPARM_CALLID	Any state after Offered or Proceeding	For SIP: string, max. length = MAX_IP_SIP_ CALLID_LENGTH For H.323: array of octets, length = MAX_IP_H323_ CALLID_LENGTH If protocol is unknown, MAX_IP_CALLID_ LENGTH defines the maximum Call ID length for any possible protocol.	both
Bearer Capability IE	IPSET_CALLINFO • IPPARM_BEARERCAP	After Offered	String, max. length = 255	H.323
Call Duration	IPSET_CALLINFO • IPPARM_CALL_DURATION	After Disconnected, before Idle	Unsigned long (value in ms)	H.323
Conference Goal	IPSET_CONFERENCE • IPPARM_CONFERENCE_GOAL	Any state after Offered or Proceeding	Uint[8]	H.323
Conference ID	IPSET_CONFERENCE • IPPARM_CONFERENCE_ID	Any state after Offered or Proceeding	char*, max. length = IP_CONFER ENCE_ID_ LENGTH (16)	H.323
Display Information	IPSET_CALLINFO • IPPARM_DISPLAY	Any state after Offered or Proceeding	char*, max. length = MAX_DISPLAY_ LENGTH (82), null-terminated	both
Facility IE	IPSET_CALLINFO • IPPARM_FACILITY	After Offered (SETUP message), Connected (CONNECT message), or the reception of a Facility message	String, max. length = 255	H.323

Notes:
1. This field is the value_buf field in the GC_PARM_DATA structure associated with the GCEV_EXTENSIONCMPLT event generated in response to the **gc_Extension( )** function requesting the information.
2. Display information, user to user information, phone list, nonstandard data, vendor information and nonstandard control information, and H221 nonstandard information may not be present.
3.Vendor information is included in a Q931 SETUP message received from a peer.
4. The nonstandard object id and nonstandard data parameters described here refer to nonstandard data contained in a SETUP message for example. This should not be confused with the nonstandard data included in protocol messages sent using **gc_Extension( )** which can be retrieved from the metaevent associated with a GCEV_EXTENSION event.

Table 4. Retrievable Call Information (Continued)

Parameter	Set ID and Parameter ID(s)	When Information Can Be Retrieved	Datatype in value_buf Field (see Note 1)	SIP/ H.323
Nonstandard Control (see note 4)	IPSET_NONSTANDARDCONTROL • IPPARM_ NONSTANDARDDATA_DATA and either • IPPARM_ NONSTANDARDDATA_OBJID or • IPPARM_H221NONSTANDARD	See Section 4.6.1, "Retrieving Nonstandard Data From Protocol Messages (H.323)", on page 149 for more information.	String, max length = max_parm_data_size Uint[ ], max length = 40 sizeof(IP_ H221NONSTANDARD)	H.323
Nonstandard Data (see note 4)	IPSET_NONSTANDARDDATA • IPPARM_ NONSTANDARDDATA_DATA and either • IPPARM_ NONSTANDARDDATA_OBJID or • IPPARM_H221NONSTANDARD	See Section 4.6.1, "Retrieving Nonstandard Data From Protocol Messages (H.323)", on page 149 for more information.	String, max length = max_parm_data_size Uint[ ], max length = 40 sizeof(IP_ H221NONSTANDARD)	H.323
Phone List	IPSET_CALLINFO • IPPARM_PHONELIST	Any state after Offered or Proceeding	char*, max. length = 131	both
User to User Information	IPSET_CALLINFO • IPPARM_USERUSER_INFO	Any state after Offered or Proceeding	char*, max. length = MAX_USERUSER_ INFO_LENGTH (131 octets)	H.323
Vendor Product ID	IPSET_VENDORINFO • IPPARM_ VENDOR_PRODUCT_ID	Any state after Offered or Proceeding	char*, max. length = MAX_PRODUCT_ ID_LENGTH (32)	H.323
Vendor Version ID	IPSET_VENDORINFO • IPPARM_ VENDOR_VERSION_ID	Any state after Offered or Proceeding	char*, max. length = MAX_VERSION_ ID_LENGTH (32)	H.323
H.221 Nonstandard Information	IPSET_VENDORINFO • IPPARM_H221NONSTD	Any state after Offered or Proceeding	IP_H221_ NONSTANDARD (see note 4)	H.323

Notes:
1. This field is the value_buf field in the GC_PARM_DATA structure associated with the GCEV_EXTENSIONCMPLT event generated in response to the **gc_Extension( )** function requesting the information.
2. Display information, user to user information, phone list, nonstandard data, vendor information and nonstandard control information, and H221 nonstandard information may not be present.
3.Vendor information is included in a Q931 SETUP message received from a peer.
4. The nonstandard object id and nonstandard data parameters described here refer to nonstandard data contained in a SETUP message for example. This should not be confused with the nonstandard data included in protocol messages sent using **gc_Extension( )** which can be retrieved from the metaevent associated with a GCEV_EXTENSION event.

If an attempt is made to retrieve information in a state in which the information is not available, no error is generated. The GC_PARM_BLK associated with the GCEV_EXTENSIONCMPLT event will not contain the requested information. If phone list and display information are requested and only phone list is available, then only phone list information is available in the GC_PARM_BLK. An error is generated if there is an internal error (such as memory cannot be allocated).

All call information is available until a **gc_ReleaseCallEx( )** is issued.

4.6.1 Retrieving Nonstandard Data From Protocol Messages (H.323)

Any received Q.931 message can include Nonstandard Data. The application can use the **gc_Extension( )** function with an **ext_id** of IPEXTID_GETINFO to retrieve the data while a call is in any state. The **target_type** should be GCTGT_GCLIB_CRN and the **target_id** should be the actual CRN. The information is included with the corresponding GCEV_EXTENSIONCMPLT termination event.

Note: When retrieving nonstandard data, it is only necessary to specify the IPPARM_NONSTANDARDDATA_DATA parameter ID in the extension request. It is not necessary to specify the ID for the nonstandard identifier parameter (that is, IPPARM_NONSTANDARDDATA_OBJID or IPPARM_H221NONSTANDARD). The call control library ensures that the GCEV_EXTENSIONCMPLT event includes all the correct information.

When retrieving nonstandard data from the GC_PARM_BLK associated with the GCEV_EXTENSIONCMPLT event, it is important to use the extended **gc_util_..._ex( )** functions because the IPPARM_NONSTANDARDDATA_DATA parameter is defined to support data that may be longer than 255 bytes. The actual maximum data length is configured by the application via the max_parm_data_size field in the IPCCLIB_START_DATA structure when it initializes the library; the default size is 255, but the application can set any value up to 4096.

4.6.2 Examples of Retrieving Call-Related Information

The following code demonstrates how to do the following:

- create a structure that identifies which information should be retrieved, then use the **gc_Extension( )** with an **extID** of IPEXTID_GETINFO to issue the request
- extract the data from a structure associated with the GCEV_EXTENSIONCMPLT event received as a termination event to the **gc_Extension( )** function

Similar code can be used when using SIP, except that the code must include only information parameters supported by SIP (see Table 4, “Retrievable Call Information”, on page 147).

Specifying Call-Related Information to Retrieve

The following function shows how an application can construct and send a request to retrieve call-related information.

```
int getInfoAsync(CRN crn)
{
   GC_PARM_BLKP gcParmBlk = NULL;
   GC_PARM_BLKP retParmBlk;
   int frc;

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_CALLINFO,
                                 IPPARM_PHONELIST,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_CALLINFO,
                                 IPPARM_CALLID,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_CONFERENCE,
                                 IPPARM_CONFERENCE_ID,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_CONFERENCE,
                                 IPPARM_CONFERENCE_GOAL,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_CALLINFO,
                                 IPPARM_DISPLAY,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_CALLINFO,
                                 IPPARM_USERUSER_INFO,
                                 sizeof(int),1);

   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }
   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_VENDORINFO,
                                 IPPARM_VENDOR_PRODUCT_ID,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_VENDORINFO,
                                 IPPARM_VENDOR_VERSION_ID,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_VENDORINFO,
                                 IPPARM_H221NONSTD,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,/* NS Data: setting this IPPARM implies
                                                retrieval of the complete element */
                                 IPSET_NONSTANDARDDATA,
                                 IPPARM_NONSTANDARDDATA_DATA,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&gcParmBlk,/* NS Control: setting this IPPARM implies
                                                retrieval of the complete element */
                                 IPSET_NONSTANDARDCONTROL,
                                 IPPARM_NONSTANDARDDATA_DATA,
                                 sizeof(int),1);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   frc = gc_Extension(GCTGT_GCLIB_CRN,
                      crn,
                      IPEXTID_GETINFO,
                      gcParmBlk,
                      &retParmBlk,
                      EV_ASYNC);
   if (GC_SUCCESS != frc)
   {
      return GC_ERROR;
   }

   gc_util_delete_parm_blk(gcParmBlk);
   return GC_SUCCESS;
}
```

Extracting Call-Related Information Associated with an Extension Event

The following code demonstrates how an application can extract call information when a GCEV_EXTENSIONCMPLT event is received as a result of a request for call-related information.

```
int OnExtensionAndComplete(GC_PARM_BLKP parm_blk,CRN crn)
{
   GC_PARM_DATA  *parmp = NULL;
   parmp = gc_util_next_parm(parm_blk,parmp);
   if (!parmp)
   {
      return GC_ERROR;
   }

   while (NULL != parmp)
   {
      switch (parmp->set_ID)
      {
         case IPSET_CALLINFO:
            switch (parmp->parm_ID)
            {
                case IPPARM_DISPLAY:
                   if(parmp->value_size != 0)
                   {
                      printf("\tReceived extension data DISPLAY: %s\n", parmp->value_buf);
                   }
                   break;

                case IPPARM_CALLID:
                   /* print the Call ID in parmp->value_buf as array of bytes */
                   for (int count = 0; count < parmp->value_size; count++)
                   {
                      printf("0x%2X ", value_buf[count]);
                   }
                   break;

                case IPPARM_USERUSER_INFO:
                   if(parmp->value_size != 0)
                   {
                      printf("\tReceived extension data UUI: %s\n", parmp->value_buf);
                   }
                   break;

                case IPPARM_PHONELIST:
                   if(parmp->value_size != 0)
                   {
                      printf("\tReceived extension data PHONELIST: %s\n",
                               parmp->value_buf);
                   }
                   break;

                default:
                   printf("\tReceived unknown CALLINFO extension parmID %d\n",
                            parmp->parm_ID);
                   break;
            }/* end switch (parmp->parm_ID) for IPSET_CALLINFO */
            break;

         case IPSET_CONFERENCE:
            switch (parmp->parm_ID)
            {
               case IPPARM_CONFERENCE_GOAL:
                  if(parmp->value_size != 0)
                  {
                     printf("\tReceived extension data IPPARM_CONFERENCE_GOAL: %d\n",
                             (unsigned int)(*(parmp->value_buf)));
                  }
                  break;

               case IPPARM_CONFERENCE_ID:
                  if(parmp->value_size != 0)
                  {
                     printf("\tReceived extension data IPPARM_CONFERENCE_ID: %s\n",
                             parmp->value_buf);
                  }
                  break;

               default:
                  printf("\tReceived unknown CONFERENCE extension parmID %d\n",
                          parmp->parm_ID);
                  break;
            }
            break;

         case IPSET_VENDORINFO:
            switch (parmp->parm_ID)
            {
               case IPPARM_VENDOR_PRODUCT_ID:
                  if(parmp->value_size != 0)
                  {
                     printf("\tReceived extension data  PRODUCT_ID %s\n", parmp->value_buf);
                  }
                  break;

               case IPPARM_VENDOR_VERSION_ID:
                  if(parmp->value_size != 0)
                  {
                     printf("\tReceived extension data  VERSION_ID %s\n", parmp->value_buf);
                  }
                  break;

               case IPPARM_H221NONSTD:
               {
                  if(parmp->value_size == sizeof(IP_H221NONSTANDARD))
                  {
                     IP_H221NONSTANDARD *pH221NonStandard;
                     pH221NonStandard = (IP_H221NONSTANDARD *)(&(parmp->value_buf));
                     printf("\tReceived extension data VENDOR H221NONSTD:
                             CC=%d, Ext=%d, MC=%d\n",
                             pH221NonStandard->country_code,
                             pH221NonStandard->extension,
                             pH221NonStandard->manufacturer_code);
                   }
               }
                   break;

               default:
                  printf("\tReceived unknown VENDORINFO extension parmID %d\n",
                           parmp->parm_ID);
                  break;
            }/* end switch (parmp->parm_ID) for IPSET_VENDORINFO */
            break;

         case IPSET_NONSTANDARDDATA:
            switch (parmp->parm_ID)
            {
               case IPPARM_NONSTANDARDDATA_DATA:
                  printf("\tReceived extension data (NSDATA) DATA: %s\n", parmp->value_buf);
                  break;
               case IPPARM_NONSTANDARDDATA_OBJID:
                  printf("\tReceived extension data (NSDATA) OBJID: %s\n", parmp->value_buf);
                  break;

               case IPPARM_H221NONSTANDARD:
               {
                  if(parmp->value_size == sizeof(IP_H221NONSTANDARD))
                  {
                     IP_H221NONSTANDARD *pH221NonStandard;
                     pH221NonStandard = (IP_H221NONSTANDARD *)(&(parmp->value_buf));
                     printf("\tReceived extension data (NSDATA) h221:CC=%d, Ext=%d, MC=%d\n",
                             pH221NonStandard->country_code,
                             pH221NonStandard->extension,
                             pH221NonStandard->manufacturer_code);
                  }
               }
               break;

               default:
                  printf("\tReceived unknown (NSDATA) extension parmID %d\n",
                          parmp->parm_ID);
                  break;
            }
            break;

         case IPSET_NONSTANDARDCONTROL:
            switch (parmp->parm_ID)
            {
               case IPPARM_NONSTANDARDDATA_DATA:
                  printf("\tReceived extension data (NSCONTROL) DATA: %s\n",
                          parmp->value_buf);
                  break;

               case IPPARM_NONSTANDARDDATA_OBJID:
                  printf("\tReceived extension data (NSCONTROL) OBJID: %s\n",
                          parmp->value_buf);
                  break;

               case IPPARM_H221NONSTANDARD:
               {
                  if(parmp->value_size == sizeof(IP_H221NONSTANDARD))
                  {
                     IP_H221NONSTANDARD *pH221NonStandard;
                     pH221NonStandard = (IP_H221NONSTANDARD *)(&(parmp->value_buf));
                     printf("\tReceived extension data (NSCONTROL) h221:CC=%d, Ext=%d, MC=%d\n",
                             pH221NonStandard->country_code,
                             pH221NonStandard->extension,
                             pH221NonStandard->manufacturer_code);
                  }
               }
               break;

               default:
                  printf("\tReceived unknown (NSCONTROL) extension parmID %d\n",
                          parmp->parm_ID);
                  break;
            }
            break;
```

```
         case IPSET_MSG_Q931:
            switch (parmp->parm_ID)
            {
               case IPPARM_MSGTYPE:
                  switch ((*(int *)(parmp->value_buf)))
                  {
                     case IP_MSGTYPE_Q931_FACILITY:
                        printf("\tReceived extension data IP_MSGTYPE_Q931_FACILITY\n");
                        break;

                     default:
                        printf("\tReceived unknown MSG_Q931 extension parmID %d\n",
                                parmp->parm_ID);
                        break;
                  } /* end  switch ((int)(parmp->value_buf)) */
                    break;
            }/* end switch (parmp->parm_ID) for IPSET_MSG_Q931 */
            break;

         case IPSET_MSG_H245:
            switch (parmp->parm_ID)
            {
               case IPPARM_MSGTYPE:
                  switch ((*(int *)(parmp->value_buf)))
                  {
                     case IP_MSGTYPE_H245_INDICATION:
                        printf("\tReceived extension data IP_MSGTYPE_H245_INDICATION\n");
                        break;

                     default:
                        printf("\tReceived unknown MSG_H245 extension parmID %d\n",
                                parmp->parm_ID);
                        break;
                  }/* end  switch ((int)(parmp->value_buf)) */
                   break;
            }/* end switch (parmp->parm_ID) for IPSET_MSG_H245 */
            break;

         default:
            printf("\t Received unknown extension setID %d\n",parmp->set_ID);
            break;
      }/* end switch (parmp->set_ID) */

      parmp = gc_util_next_parm(parm_blk,parmp);
   }

   return GC_SUCCESS;
}
```

Note: IPPARM_CALLID is a set of bytes and should *not* be interpreted as a string.

Retrieving Call ID

The following code example illustrates how to request Call ID information via a **gc_Extension( )** call.

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_iptB1T1:P_SIP, :N_iptB1T2:P_SIP, etc...)
 * 2. gc_WaitCall() has been issued to wait for a call.
 * 3. gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows) has been called
 *    to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 */
```

```
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>

/*
 * Assume the 'crn' parameter holds the CRN associated
 * with the detected GCEV_OFFERED event.
 */
int request_call_info(CRN crn)
{
   int retval = GC_SUCCESS;
   GC_PARM_BLKP parmblkp = NULL;  /* input parameter block pointer */
   GC_PARM_BLKP retblkp = NULL;   /* pointer for output parameter block (unused) */
   GC_INFO gc_error_info;         /* GlobalCall error information data */

   /* allocate GC_PARM_BLK for Call-ID message parameter */
   gc_util_insert_parm_val(&parmblkp, IPSET_CALLINFO, IPPARM_CALLID, sizeof(int), 1);
   if (parmblkp == NULL)
   {
      /* memory allocation error */
      return(-1);
   }

   /* retrieve the Call-ID from the network */
   if (gc_Extension(GCTGT_GCLIB_CRN, crn, IPEXTID_GETINFO, parmblkp, &retblkp,
                    EV_ASYNC) != GC_SUCCESS)
   {
      /* process error return as shown */
      gc_ErrorInfo( &gc_error_info );
      printf ("Error: gc_Extension() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,
              CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
              crn, gc_error_info.gcValue, gc_error_info.gcMsg, gc_error_info.ccLibId,
              gc_error_info.ccLibName, gc_error_info.ccValue, gc_error_info.ccMsg);
   }

   /* free the parameter block */
   gc_util_delete_parm_blk(parmblkp);

   return (retval);
}
```

Parsing Call ID Information (SIP Protocol)

The following code example illustrates how to parse the Call ID information retrieved via a **gc_Extension( )** call when the SIP protocol is being used.

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_iptB1T1:P_SIP, :N_iptB1T2:P_SIP, etc...)
 * 2. gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows) has been called
 *    to convert the event into metaevent.
 * 3. a GCEV_EXTENSIONCMPLT has been detected.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>

/* Assume the 'crn' parameter holds the CRN associated with the detected GCEV_EXTENSIONCMPLT
 * event, and the 'pEvt' parameter holds a pointer to the detected metaevent.
 */
```

```
int print_call_info(CRN crn, METAEVENT *pEvt)
{
   EXTENSIONEVTBLK *ext_data = NULL;
   GC_PARM_DATA *parmp = NULL;
   GC_PARM_BLK  *parm_blkp;

   if (pEvt)
   {
      if (pEvt->evttype == GCEV_EXTENSIONCMPLT)
      {
         ext_data = (EXTENSIONEVTBLK *)(pEvt->extevtdatap);
      }
   }

   if (!ext_data)
   {
      printf("\tNot a GCEV_EXTENSIONCMPLT event.\n");
      return GC_ERROR;
   }

   parm_blk = &(ext_data->parmblk);

   parmp = gc_util_next_parm(parm_blkp,parmp);
   if (!parmp)
   {
      printf("\tNo data returned in extension event for crn: 0x%lx\n", crn);
      return GC_ERROR;
   }

   while (NULL != parmp)
   {
      switch (parmp->set_ID)
      {
         case IPSET_CALLINFO:
            switch (parmp->parm_ID)
            {
               case IPPARM_CALLID:
                  if(parmp->value_size != 0)
                  {
                     /* Here's where we print the SIP Call ID */
                     printf("\tReceived extension data IPPARM_CALLID: %s\n",
                            parmp->value_buf);
                  }
                  break;

               default:
                  printf("\tReceived unexpected IPSET_CALLINFO parmID %d\n",
                             parmp->parm_ID);
                  break;
            } /* end switch (parmp->parm_ID) */
            break;

         default:
            printf("\t Received unexpected extension setID %d\n",
                   parmp->set_ID);
            break;

      } /* end switch (parmp->set_ID) */

      parmp = gc_util_next_parm(parm_blkp,parmp);
   } /* end while (parmp != NULL) */

   return GC_SUCCESS;
}
```

4.7 Receiving Notification Events

Note: The information in this section only applies when the Dialogic® Global Call API IP call control library is started in the first party call control (1PCC) operating mode. The extension events that provides the capabilities described in this section are not supported when the library is started in the third party call control (3PCC) operating mode.

The Global Call library allows applications to receive unsolicited notification events for several different types of state changes and other transition events.

This section includes the following topics:

- Enabling and Disabling Unsolicited Notification Events
- Getting Media Streaming Status and Connection Information
- Getting Notification of Underlying Protocol State Changes

4.7.1 Enabling and Disabling Unsolicited Notification Events

The application can enable and disable the unsolicited GCEV_EXTENSION notification events associated with certain types of transition events, including:

- media streaming connection state changes (see Section 4.7.2, "Getting Media Streaming Status and Connection Information")
- underlying protocol (Q.931 and H.245) connection state changes (see Section 4.7.3, "Getting Notification of Underlying Protocol State Changes")
- DTMF digit detection (see Section 4.24.2, "Getting Notification of DTMF Detection", on page 300)

4.7.1.1 GCEV_EXTENSION

Enabling and disabling unsolicited GCEV_EXTENSION notification events is done by manipulating the event mask, which has a default value of zero, using the **gc_SetConfigData( )** function. The relevant **gc_SetConfigData( )** function parameter values in this context are:

- **target_type** – GCTGT_CCLIB_NETIF
- **target_id** – IPT board device
- **size** – set to a value of GC_VALUE_LONG
- **target_datap** – a pointer to a GC_PARM_BLK structure that contains the parameters to be configured

The GC_PARM_BLK should contain a parameter element with the IPSET_EXTENSIONEVT_MSK set ID and one of the following parameter IDs:

GCACT_ADDMSK
    Add an event to the mask

GCACT_SUBMSK
    Remove an event from the mask

GCACT_SETMSK
: Set the mask to a specific value

Possible values (corresponding to events that can be added or removed from the mask) are:

EXTENSIONEVT_DTMF_ALPHANUMERIC
: For notification of DTMF digits received in User Input Indication (UII) messages with alphanumeric data. When using SIP, this value is not applicable.

EXTENSIONEVT_SIGNALING_STATUS
: For notification of intermediate protocol state changes in signaling (in H.323, for example, Q.931 Connected and Disconnected) and control (in H.323, for example, H.245 Connected and Disconnected).

EXTENSIONEVT_STREAMING_STATUS
: For notification of the status and configuration information of transmit or receive directions of media streaming including: Tx Connected, Tx Disconnected, Rx Connected, and Rx Disconnected.

4.7.1.2 GCEV_TELEPHONY_EVENT

The application can enable and disable the unsolicited GCEV_TELEPHONY_EVENT notification for every RFC 2833 DTMF digit received. This event delivers raw RFC 2833 information in the form of IPM_TELEPHONY_INFO structure for each DTMF digit received. The IPM_TELEPHONY_INFO structure contains the volume, duration and event ID of the digit. (Refer to the *Dialogic® IP Media Library API Programming Guide and Library Reference* for the structure definition.)

To enable this notification, use the **gc_SetUserInfo( )** function with IPSET_DTMF set ID and the parameter ID:

IPPARM_TELEPHONY_EVENT_DTMF
: Used to enable or disable notification of DTMF digits received within an RTP stream in RFC 2833 format. Notification is via GCEV_TELEPHONY_EVENT for every DTMF received. Values are IP_ENABLE and IP_DISABLE (default).

Notes: ***1.*** Notification is only available on a per line device basis; duration is GC_ALLCALLS.

2. The DTMF transmission mode must be RFC 2833. Refer to Section 4.24.1, “Specifying DTMF Support”, on page 297 for more information.

Example of enabling GCEV_TELEPHONY_EVENT notifications:

```
#include "gcip.h"
#include "gclib.h"
#include "gcip_defs.h"

int enable_dtmf_event(LINEDEV linedev)
{
       GC_PARM_BLKP parmblkp = NULL;

       /* Set the parameter block */
       gc_util_insert_parm_val(&parmblkp, IPSET_DTMF, IPPARM_TELEPHONY_EVENT_DTMF,
       sizeof(int),
       IP_ENABLE);
```

```
            /* Enable GCEV_TELEPHONY_EVENT notification */
            if (gc_SetUserInfo(GCTGT_GCLIB_CHAN,
                         linedev,
                         parmblkp,
                         GC_ALLCALLS) != GC_SUCCESS)

            /* Process error */
       }
          /* Free the parameter block */
          gc_util_delete_parm_blk(parmblkp);

          /* More processing */
          return;
      }
```

4.7.2 Getting Media Streaming Status and Connection Information

The application can receive notification of changes in the status (connection and disconnection) of media streaming in the transmit and receive directions as GC_EXTENSIONEVT events. When the event is a notification of the connection of the media stream in either direction, information about the coders negotiated for that direction and the local and remote RTP addresses is also available.

The events for this notification must be enabled by setting or adding the bitmask value EXTENSIONEVT_SIGNALING_STATUS to the GC_EXTENSIONEVT mask; see Section 4.7.1, "Enabling and Disabling Unsolicited Notification Events", on page 158. Once the events are enabled, when a media streaming connection state changes, the application receives a GCEV_EXTENSION event. The EXTENSIONEVTBLK structure pointed to by the extevtdatap pointer within the GCEV_EXTENSION event will contain the following information for all media streaming status changes:

extID
: IPEXTID_MEDIAINFO

parmblk
: A GC_PARM_BLK containing the protocol connection status with the IPSET_MEDIA_STATE parameter set ID and one of the following parameter IDs:
  - IPPARM_TX_CONNECTED – Media streaming has been initiated in transmit direction. The parameter value is an IP_CAPABILITY structure containing the coder configuration that resulted from the capability exchange with the remote peer.
  - IPPARM_TX_DISCONNECTED – Media streaming has been terminated in transmit direction. No parameter value is used with this parameter ID.
  - IPPARM_RX_CONNECTED – Media streaming has been initiated in receive direction. The parameter value is an IP_CAPABILITY structure containing the coder configuration that resulted from the capability exchange with the remote peer.
  - IPPARM_RX_DISCONNECTED – Media streaming has been terminated in receive direction. No parameter value is used with this parameter ID.
  - IPPARM_TX_SENDONLY – Media streaming has been initiated for a half-duplex transmit-only connection. The parameter value is an IP_CAPABILITY structure containing the coder configuration that resulted from the capability exchange with the remote peer.

- IPPARM_RX_RECVONLY – Media streaming has been initiated for a half-duplex receive-only connection. The parameter value is an IP_CAPABILITY structure containing the coder configuration that resulted from the capability exchange with the remote peer.
- IPPARM_TX_INACTIVE – Media streaming in the transmit direction has been suspended. The parameter value is an IP_CAPABILITY structure containing the coder configuration that resulted from the capability exchange with the remote peer.
- IPPARM_RX_INACTIVE – Media streaming in the receive direction has been suspended. The parameter value is an IP_CAPABILITY structure containing the coder configuration that resulted from the capability exchange with the remote peer.

When the parameter value in the GC_PARM_BLK structure is IPPARM_TX_CONNECTED, indicating that a transmit media stream connection has occurred, the GC_PARM_BLK structure will also contain the local and remote RTP addresses. These addresses are handled as an RTP_ADDR data structure, which contains both the port number and the IP address. The parameter set ID used for the RTP addresses is IPSET_RTP_ADDRESS, and the parameter IDs are IPPARM_LOCAL and IPPARM_REMOTE.

RTP Address and Coder Information Retrieval Example

The following code snippet illustrates how to retrieve the RTP addresses and negotiated coder information from a media stream connection event:

```
//When the event is an extension event:

GC_PARM_BLKP     gcParmBlk;
EXTENSIONEVTBLK  *pextensionBlk;
GC_PARM_DATA     *parmp = NULL;
RTP_ADDR         l_RTA1,l_RTA2;
pextensionBlk = (EXTENSIONEVTBLK *)(m_pMetaEvent->extevtdatap);
gcParmBlk = (&(pextensionBlk->parmblk));

GC_PARM_DATAP l_pParmData;
IP_CAPABILITYl_IPCap;

switch(pextensionBlk->ext_id)
{
   case IPEXTID_MEDIAINFO:

   //get the coder info:
   l_pParmData = gc_util_find_parm(gcParmBlk, IPSET_MEDIA_STATE, IPPARM_TX_CONNECTED);

   if(l_pParmData != NULL)
   {
      memcpy(&l_IPCap, l_pParmData->value_buf, l_pParmData->value_size);

      // get the local RTP address:
      l_pParmData= gc_util_find_parm(gcParmBlk, IPSET_RTP_ADDRESS, IPPARM_LOCAL);
      if(l_pParmData!= NULL)
      {
         memcpy(&l_RTA1,l_pParmData->value_buf,l_pParmData->value_size);
      }

      //get the remote RTP address:
      l_pParmData =gc_util_find_parm(gcParmBlk, IPSET_RTP_ADDRESS, IPPARM_REMOTE);
      if(l_pParmData != NULL)
      {
         memcpy(&l_RTA2, l_pParmData->value_buf, l_pParmData->value_size);
      }
   }
```

```
      else
      {
         //only get tx or rx, not both
         l_pParmData = gc_util_find_parm(gcParmBlk, IPSET_MEDIA_STATE, IPPARM_RX_CONNECTED);
         if(l_pParmData != NULL)
         {
            memcpy(&l_IPCap, l_pParmData->value_buf, l_pParmData->value_size);
         }
      }
   }
```

4.7.3 Getting Notification of Underlying Protocol State Changes

The application can receive notification of intermediate protocol signaling state changes for both H.323 and SIP. The events for this notification must be enabled; see Section 4.7.1, "Enabling and Disabling Unsolicited Notification Events", on page 158.

Once these events are enabled, when a protocol state change occurs, the application receives a GCEV_EXTENSION event. The EXTENSIONEVTBLK structure pointed to by the extevtdatap pointer within the GCEV_EXTENSION event will contain the following information:

extID
: IPEXTID_IPPROTOCOL_STATE

parmblk
: A GC_PARM_BLK containing the protocol connection status with the IPSET_IPPROTOCOL_STATE parameter set ID and one of the following parameter IDs:
  - IPPARM_SIGNALING_CONNECTED – The signaling for the call has been established with the remote endpoint. For example, in H.323, a CONNECT message was received by the caller or a CONNECTACK message was received by the callee.
  - IPPARM_SIGNALING_DISCONNECTED – The signaling for the call has been terminated with the remote endpoint. For example, in H.323, a RELEASE message was received by the terminator or a RELEASECOMPLETE message was received by peer.
  - IPPARM_CONTROL_CONNECTED – Media control signaling for the call has been established with the remote endpoint. For example, in H.323, an OpenLogicalChannel message (for the receive direction) or an OpenLogicalCahnnelAck message (for the transmit direction) was received.
  - IPPARM_CONTROL_DISCONNECTED – Media control signaling for the call has been terminated. For example, in H.323, an EndSession message was received.

  Note: The parameter value field in this GC_PARM_BLK in each case is unused (NULL).

4.8 Modifying an Existing SIP Call via re-INVITE

This section discusses the Dialogic® Global Call API implementation of the SIP re-INVITE method as it applies to first party call control (1PCC). The use of re-INVITE in the context of third party call control (3PCC) is discussed in Chapter 5, "Third Party Call Control (3PCC) Operations and Multimedia Support".

This section includes the following topics:

- Overview of the SIP re-INVITE Method
- Enabling Application Access to re-INVITE Requests

- Receiving SIP re-INVITE Requests
- Responding to SIP re-INVITE Requests
- Sending a SIP re-INVITE Request
- Canceling a Pending re-INVITE Request
- Updating Dialog Properties via re-INVITE
- Implementing Hold and Retrieve via SIP re-INVITE

4.8.1 Overview of the SIP re-INVITE Method

RFC 3261 specifies that User Agents must be able to send and respond to additional INVITE requests after a dialog has been established to allow modification of the dialog or the media session. These subsequent INVITE requests in an existing dialog are known as re-INVITE requests to distinguish them from an initial INVITE request that initiates a new dialog. Re-INVITE requests contain the same Call-ID and To and From tags as the original INVITE request that established the dialog. Either party in a dialog can issue a re-INVITE, and only one re-INVITE can be pending at any given time.

The re-INVITE method is a general purpose mechanism that potentially can be used to modify or update nearly any property of a dialog (notably excluding the header fields that are used to identify the message as a subsequent INVITE rather than a new INVITE) or the associated media session. But it is important to note that different IP telephony platforms support re-INVITE requests to varying degrees. For example, some platforms may only support changing the RTP address while others may also support changing the direction(s) of media streaming or even the codec characteristics. Each endpoint has to determine whether it supports the changes requested in a re-INVITE, and whether it wishes to accept requests that it supports. An endpoint must reject any re-INVITE request that it does not support, and may optionally reject any re-INVITE request for any reason whatsoever.

In first party call control mode (1PCC), the Dialogic® Global Call API library for Dialogic® Host Media Processing Software supports the following capabilities for re-INVITE, which are described in detail in the subsections of this section:

- specifying, changing, or refreshing header field values or parameters for the existing dialog; for example, refreshing expiring Contact information
- changing the DTMF mode
- changing the direction of the streaming; for example, changing from half-duplex to full-duplex streaming
- suspending and resuming streaming to implement hold and retrieve functionality
- changing the RTP port of the remote endpoint

  Note: Global Call does not provide a mechanism for initiating an RTP port change, but Global Call applications can receive and act on port change requests received from non-Global Call applications.
- changing coder properties of the media session; for example, changing from a low bit-rate coder to reduce resource requirements

- changing between audio and T.38 fax modes

  Note: The existing automatic and manual modes for audio/T.38 switching (as described in Section 4.38, "T.38 Fax Server") have used re-INVITE "under the hood" when using the SIP protocol. But when an application has enabled access to re-INVITE requests, audio/T.38 fax mode changes must be handled explicitly by the application, just like any other re-INVITE requests.

4.8.2 Enabling Application Access to re-INVITE Requests

Note: Access to re-INVITE messages must be enabled as described in this section in both 1PCC and 3PCC operating modes.

For backwards compatibility in 1PCC mode, the default behavior of the Dialogic® Global Call API library is to automatically reject all re-INVITE requests it receives that are not related to T.38, and to do so without notifying the application.

In order to have access to received SIP re-INVITE requests, applications must set a specific parameter value using the Global Call **gc_SetConfigData( )** function. To enable the GCEV_REQ_MODIFY_CALL event type that is used to notify applications of re-INVITE requests, the application must include the following parameter element in the GC_PARM_BLK that it passes to the **gc_SetConfigData( )** function:

IPSET_CONFIG
  IPPARM_OPERATING_MODE
  - value = IP_T38_MANUAL_MODIFY_MODE

The following code snippet illustrates how to set this parameter:

```
GC_PARM_BLKP parmblkp = NULL;
long request_id = 0;
gc_util_insert_parm_val(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_OPERATING_MODE,
                        sizeof(int),
                        IP_T38_MANUAL_MODIFY_MODE);

if (gc_SetConfigData(GCTGT_CCLIB_NETIF, boardDev, parmblkp, 0 /*timeout*/,
                     GCUPDATE_IMMEDIATE, &request_id, EV_ASYNC) != GC_SUCCESS)
{
   // handle error…
}
```

In addition to enabling the GCEV_REQ_MODIFY_CALL event for access to received re-INVITE requests, this parameter setting also enables the three **gc_xxxModifyMedia( )** APIs that support re-INVITE functionality. Unless this parameter value is set, any attempt to call one of the **gc_xxxModifyMedia( )** functions will fail with an IPERR_BAD_PARM error code.

4.8.3 Receiving SIP re-INVITE Requests

This section focuses primarily on library behavior in 1PCC operating mode. In 3PCC, the application is responsible for parsing and SDP offers and constructing SDP answers.

RFC3261 specifies that either party in a SIP dialog can initiate a re-INVITE transaction, so Global Call applications must be able to receive and handle incoming re-INVITE requests whenever application access to re-INVITE is enabled.

When the IP Call Control Library receives a re-INVITE request, the library first examines the request to determine whether it specifies media properties that are acceptable by the local endpoint. If the received re-INVITE request specifies media capabilities that are not supported by the local system, the library automatically sends a 488 Not Acceptable Here response to the requesting party and generates a GCEV_REQ_MODIFY_UNSUPPORTED event to the application. This unsolicited event contains a CCLIB cause code of IPEC_SIPReasonStatus488NotAcceptableHere. This event is sent for informational purposes only; the library has already sent the appropriate response to the remote UA, so the local application does not need to take any action upon receiving this informational event.

If the received re-INVITE request does not contain an SDP offer, or if it contains an SDP offer that specifies media capabilities that are supported by the local media device, the call control library automatically sends a 100 Trying response to the requester and generates an unsolicited GCEV_REQ_MODIFY_CALL event to notify the application. The METAEVENT associated with this event contains a pointer to a GC_PARM_BLK structure that the library has populated with the following information from the re-INVITE request:

- a parameter element that indicates the DTMF mode
- parameter elements for any SIP header fields that the application has registered to receive (as described in Section 4.9.4, “Registering SIP Header Fields to be Retrieved”, on page 182)
- one or more parameter elements that contain media session properties that were specified in the received SDP offer (if there was one)
- a parameter element that contains the remote RTP transport address from the received SDP offer (if there was one)

The DTMF mode specified in the re-INVITE may or may not match the properties of the current session. It is the application’s responsibility to determine whether the DTMF mode is different from the current mode, and to decide whether any change being proposed is acceptable. The DTMF mode is contained in a parameter element of the type:

IPSET_DTMF
 IPPARM_SUPPORT_DTMF_BITMASK
 - value = IP_DTMF_TYPE_INBAND_RTP or IP_DTMF_TYPE_RFC_2833

The parameter elements associated with the Call-ID, To, and From headers will contain the same values that were used in the original INVITE request that established the dialog. All other header fields and parameters have potentially been changed, and it is the application’s responsibility to parse and compare the values if appropriate. The header fields that the application has registered to receive are reported in parameter elements of the following type:

IPSET_SIP_MSGINFO
 IPPARM_SIP_HDR
 - value = complete header string, including name, value, and any parameters

If the re-INVITE request contains an SDP offer, the media capabilities proposed in the offer may or may not match the properties of the current media session. It is the application’s responsibility to analyze the media properties proposed in the SDP offer, to determine whether the properties are different from the current session properties, and to decide whether any proposed change is acceptable.

The GC_PARM_BLOCK that is associated with the GCEV_REQ_MODIFY_CALL event may contain any number of parameter elements which identify the supported media properties that were proposed in the request. Each proposed media capability is handled as a parameter element of the following type:

GCSET_CHAN_CAPABILITY
    IPPARM_LOCAL_CAPABILITY
    - value = IP_CAPABILITY data structure

The number of these parameter elements depends on the specifics of what change the re-INVITE is requesting:

- If the SDP offer in the re-INVITE is proposing a full-duplex media session, there will be a pair of GCSET_CHAN_CAPABILITY/IPPARM_LOCAL_CAPABILITY parameter elements for each proposed media capability that is supported on the local platform, one element for each direction. Within each parameter pair, all fields of the of the IP_CAPABILITY structure will be the same except for the direction fields, one of which will be IP_CAP_DIR_LCLRECEIVE and the other IP_CAP_DIR_LCLTRANSMIT.
- If the SDP offer in the re-INVITE is proposing a half-duplex media session, there may be only a single GCSET_CHAN_CAPABILITY/ IPPARM_LOCAL_CAPABILITY element in the parameter block, although multiple elements are possible if multiple coders are being proposed. Within each parameter element, the IP_CAPABILITY.direction field will be either IP_CAP_DIR_LCLRECVONLY or IP_CAP_DIR_LCLSENDONLY.
- If the SDP offer in the re-INVITE is seeking to suspend streaming (to place the call on hold, for example), there may be only a single GCSET_CHAN_CAPABILITY/ IPPARM_LOCAL_CAPABILITY element in the parameter block, although multiple elements are possible. When the re-INVITE is requesting to suspend streaming, the IP_CAPABILITY.direction field will be set to either IP_CAP_DIR_LCLRTPINACTIVE or IP_CAP_DIR_LCLRTPRTCPINACTIVE.
- If the SDP offer in the re-INVITE is proposing a change from audio mode to T.38 fax mode, there will be only one GCSET_CHAN_CAPABILITY/ IPPARM_LOCAL_CAPABILITY element in the parameter block. Within this element, the IP_CAPABILITY.capability field will be GCCAP_DATA_t38UDPFax and the IP_CAPABILITY.direction field will be IP_CAP_DIR_LCLTXRX.

Finally, The GC_PARM_BLK will include a parameter element that contains the remote RTP transport address, which may be the same as the existing address or may be different. It is the application's responsibility to compare the address to determine whether it is different and whether the proposed change is acceptable.

The RTP transport address is handled as a parameter element of the following type:

IPSET_RTP_ADDRESS
    IPPARM_REMOTE
    - value = RTP_ADDR data structure

There will always be at least one of these parameter elements if the re-INVITE request contains an SDP offer (which is the typical case for re-INVITE requests).

Note: SDP does not explicitly communicate RTCP port addresses, but these can be inferred from RTP addresses according to the "plus one" offset convention.

4.8.4 Responding to SIP re-INVITE Requests

This section focuses primarily on library behavior in 1PCC operating mode. In 3PCC, the application is responsible for constructing the SDP answer.

After an application has received an unsolicited GCEV_REQ_MODIFY_CALL event that signals reception of a re-INVITE request, and has retrieved and analyzed the parameter elements from the GC_PARM_BLK associated with the METAEVENT, it is able to accept or reject the proposed change by calling the appropriate Global Call API.

4.8.4.1 Rejecting a SIP re-INVITE Request

When an application determines that it is unable to or does not wish to accept the changes that were proposed in a received re-INVITE request, it simply calls the **gc_RejectModifyCall( )** function to send a final response message with the specified 3xx–6xx reason code. The reason code to send is specified using the appropriate IPEC_SIPReasonStatus… defines as defined in *gcip_defs.h* and documented in Section 11.5, "Failure Response Codes When Using SIP", on page 700.

When the remote user agent acknowledges the rejection response, the library generates a GCEV_REJECT_MODIFY_CALL completion event to notify the application and the media session continues unchanged, just as if a re-INVITE request had never been issued.

If the transmission of the rejection message fails for some reason, the library generates a GCEV_REJECT_MODIFY_CALL_FAIL event. In the case of such a failure, the re-INVITE transaction is still in progress, and the application should make another attempt to respond to the request.

4.8.4.2 Accepting a SIP re-INVITE Request

When an application determines that the changes to the existing dialog or media session that were proposed in a received re-INVITE request are acceptable, it calls the **gc_AcceptModifyCall( )** function to send a 200 OK response. But because the SDP offer contained in a re-INVITE request may contain more than one session proposal, the application has the opportunity to specify which proposal it wishes to accept.

If the application calls **gc_AcceptModifyCall( )** with a NULL pointer as the **parmblkp** parameter, the library uses the codec preferences that were used in the original INVITE dialog to formulate the SDP response. In this case, if the SDP offer in the re-INVITE proposed a codec that the application did not indicate as acceptable in the original INVITE dialog, the library treats the situation as a rejection of the call modification request. In this case, a 488 Not Acceptable Here response is sent to the remote party to terminate the re-INVITE dialog, and a GCEV_REJECT_MODIFY_CALL event is sent to the application.

To formulate a specific SDP answer, an application inserts appropriate media capability parameter elements into the GC_PARM_BLK parameter block that it passes to **gc_AcceptModifyCall( )**. Each parameter element is of the following format:

GCSET_CHAN_CAPABILITY
  IPPARM_LOCAL_CAPABILITY
  - value = IP_CAPABILITY data structure

A full-duplex connection requires two such parameter elements, one for each direction. A half-duplex connection requires one parameter element with the direction field of the IP_CAPABILITY structure set appropriately.

To accept an on-hold request, the application must insert a parameter element with an IP_CAPABILITY structure that contains one of the direction values that specifies inactive streaming. If the application does not specify a capability that matches a proposed capability in the re-INVITE's SDP offer, the library treats the situation as a rejection of the modification request, sends a 488 Not Acceptable Here response to the remote party to terminate the re-INVITE dialog, and generates a GCEV_REJECT_MODIFY_CALL to the application.

If the re-INVITE request specifies a change in the codec, the library makes the change effective on the local media platform as soon as the **gc_AcceptModifyCall( )** function is called. All further packets sent by the local media platform will use the new codec, and only packets using the new codec will be accepted from the remote endpoint. This may cause some audible artifact, such as a click or a brief silence, if the remote endpoint is not able to synchronize codecs promptly.

When the remote UA acknowledges the 200 OK response, the library generates a GCEV_ACCEPT_MODIFY_CALL event to notify the application that the re-invite transaction has completed successfully. If the transmission of the 200 OK message fails for some reason, the library generates a GCEV_ACCEPT_MODIFY_CALL_FAIL event. In the case of such a failure, the re-INVITE transaction is still in progress, and the application should make another attempt to respond to the re-INVITE request.

4.8.5 Sending a SIP re-INVITE Request

This section focuses primarily on library behavior in 1PCC operating mode. In 3PCC, the application is responsible for constructing the SDP offer that will be contained in the re-INVITE.

To send a SIP re-INVITE request, an application begins by constructing a GC_PARM_BLK that contains parameter elements for the dialog and media session properties that it wishes to change. Then the application passes that parameter block in a call to the **gc_ReqModifyCall( )** function. Note that there can be only a single re-INVITE transaction pending at any given time; if there is a re-INVITE already pending (initiated by either endpoint), calling **gc_ReqModifyCall( )** produces an error result.

If a re-INVITE request times out, the library generates a GCEV_MODIFY_CALL_FAIL event to the application with a cause value of IPEC_SIPReasonStatus408RequestTimeout. In compliance with RFC 3261 the 408 timeout condition causes the library to send BYE to terminate the dialog, and it notifies the application of this termination with a GCEV_DISCONNECTED event.

The GC_PARM_BLK that the application constructs may contain three types of parameter elements. There may be an element to specify the DTMF mode, one or more elements to specify SIP header fields to change in order to update the properties of the dialog (such as the Contact or Via information), and one or more elements to specify media capabilities to be included in the SDP offer within the re-INVITE request.

4.8.5.1 Specifying DTMF Mode in a re-INVITE Request

An application may request a change in the DTMF mode in re-INVITE request by inserting a parameter element of the following type in the GC_PARM_BLK it passes to the **gc_ReqModifyCall( )** function:

IPSET_DTMF
  IPPARM_SUPPORT_DTMF_BITMASK
  - value = IP_DTMF_TYPE_INBAND_RTP or IP_DTMF_TYPE_RFC_2833

4.8.5.2 Inserting SIP Header Fields in a re-INVITE Request

SIP header fields to be sent in a re-INVITE are specified using the standard technique. The application simply inserts parameter elements of the following type into the GC_PARM_BLK it passes to **gc_ReqModifyCall( )**:

IPSET_SIP_MSGINFO
  IPPARM_SIP_HDR
  - value = complete header string, including header field name

The header fields are inserted in the SIP message in the same order in which they are inserted into the GC_PARM_BLK. See Section 4.9.5, "Setting SIP Header Fields for Outbound Messages", on page 185 for more details on sending SIP headers.

When setting header fields in SIP re-INVITE requests, there are some restrictions to note:

- Request-URI and Call-ID cannot be set by the application because they are used to identify the request as a subsequent INVITE request (re-INVITE).
- CSeq cannot be set by the application.
- In the From and To headers, the URI and Tag cannot be changed because they are used to identify the request as a re-INVITE. In both cases, the Display and some of the URI parameters *can* be changed, but the application must ensure that the URI and Tag substrings that it includes when specifying the header string are identical to those in the original INVITE.
- Max-Forwards can be set by the application, but if the application does not set it the library automatically sets it to 70.
- Contact and Via can be set by the application, but if the application does not provide them the library automatically inserts the corresponding header field from the last INVITE or 2xx response that the application sent in the current dialog.

All other header fields, including proprietary headers, can be set without restriction.

4.8.5.3 Specifying Media Session Properties in a SIP re-INVITE

If an application wishes to change any media session properties via a re-INVITE request, it must insert appropriate media capability parameter elements into the GC_PARM_BLK that it passes to **gc_ReqModifyCall( )**. If there is no need to change media session properties (for example, when using re-INVITE simply to refresh the Contact information for the dialog), the application can opt to not include media session property parameter elements in the GC_PARM_BLK, in which case the library will use the last SDP answer (that is, the current session properties) when it constructs the re-INVITE.

The parameter elements for media capabilities are of the form:

GCSET_CHAN_CAPABILITY
IPPARM_LOCAL_CAPABILITY
- value = IP_CAPABILITY structure

For a full-duplex media session, the application must insert these capability parameter elements in pairs, one for transmit (IP_CAPABILITY.direction = IP_CAP_DIR_LCLTRANSMIT) and one for receive (IP_CAPABILITY.direction = IP_CAP_DIR_LCLRECEIVE). If multiple session proposals are being included in the SDP offer, the application inserts multiple such pairs of parameter elements in order of codec preference.

For a half-duplex media session, the application inserts a single parameter element with the IP_CAPABILITY.direction field set to either IP_CAP_DIR_LCLTXONLY or IP_CAP_DIR_LCLRXONLY. If multiple session proposals are being included in the SDP offer, the application inserts multiple parameter elements of this type in order of codec preference.

When requesting the remote endpoint to switch from audio mode to T.38 fax mode, the application inserts only a single parameter element with IP_CAPABILITY.capability set to GCCAP_DATA_t38UDPFax and IP_CAPABILITY.direction set to IP_CAP_DIR_LCLTXRX.

When requesting the remote endpoint to suspend streaming to place a call on hold, the application inserts only a single parameter element with IP_CAPABILITY.direction set to either IP_CAP_DIR_LCLRTPINACTIVE (to disable RTP streaming only) or IP_CAP_DIR_LCLRTPRTCPINACTIVE (to disable both RTP and RTCP).

In each case, the IP_CAPABILITY structure must be fully specified. If only one property is being changed (for example, only changing the direction), the remaining fields of the structure must contain the current values for each of the other capability properties.

4.8.6 Canceling a Pending re-INVITE Request

If an application wishes to cancel a pending re-INVITE request, it first inserts a special parameter element into a GC_PARM BLK, then passes that parameter block to **gc_ReqModifyCall( )**.

The parameter element used to cancel a pending re-INVITE is:

IPSET_MSG_SIP
IPPARM_SIP_METHOD
- value = IP_MSGTYPE_SIP_CANCEL

No other parameter elements can be present in the GC_PARM_BLK when canceling a re-INVITE request.

4.8.7 Updating Dialog Properties via re-INVITE

Dialog properties that are specified in SIP message header fields can be updated or changed by sending a re-INVITE request that contains header fields with new values. The most common use of this capability is to provide updated Contact information or to refresh it when the Expires interval is exceeded. Note that either party in a dialog can issue a re-INVITE to refresh or update dialog properties.

As noted earlier in this section, applications cannot change the Call-ID, the URI or Tag in the From and To headers, or the CSeq, since all of these are restricted values in re-INVITE requests.

With the exception of three header fields that the library automatically populates, only the header fields that are explicitly specified by the application will be transmitted in the re-INVITE and updated at the remote endpoint. The Contact and Via headers are automatically populated by the library with the corresponding header values from the last 2xx or INVITE message that was sent by the application in the current dialog unless the application explicitly sets the header in the re-INVITE. The other auto-fill header field is Max-Forwards, which is set to 70 by default.

When the application only needs to send updated header fields (that is, when does not also need to change any media session properties), the simplest approach is for the application to not include any capability elements in the GC_PARM_BLK that it passes to **gc_ReqModifyCall( )**. In this circumstance, the library automatically inserts the last SDP answer in the re-INVITE request that it constructs. Alternatively, the application can explicitly insert the current capabilities in the GC_PARM_BLK.

The following code example illustrates the use of re-INVITE to update the Contact header:

```
.
.
.
#include <gcip.h>
#include <gclib.h>
.
.
.

/* Request Contact refresh:                                             */
/* Assumes: 1) caller has verified call to be in connected state        */
/*          2) caller has enabled event handler for GCEV_MODIFY_CALL_ACK, */
/*             GCEV_MODIFY_CALL_REJ, and GCEV_MODIFY_CALL_FAIL.         */

int refreshToHomeLocation (CRN crn)
{

   char *pContactHeader = "Contact: Rich <r.intelligent@myhomeISP.com>";

   gc_util_insert_parm_ref_ex(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_SIP_HDR,
                              (unsigned long)(strlen(pContactIdHeader) + 1),
                              pContactHeader);

   if (NULL == parmblkp)  return FAILURE;
   if (gc_ReqModifyCall(crn, parmblkp, EV_ASYNC) < 0)  return FAILURE;

   gc_util_delete_parm_blk(parmblkp);

} /* End of function. */
```

4.8.8 Implementing Hold and Retrieve via SIP re-INVITE

Either party in a SIP dialog (calling or called) can put the call on hold by sending a re-INVITE request that contains a specially configured SDP offer that requests the remote endpoint to suspend RTP streaming. SIP standards define two methods for specifying suspension of RTP streaming:

- The newer method of signaling an on-hold request sets the direction attribute in the media description of the SDP offer to “a=inactive”. This method, which is indicated as the preferred method in RFC 3261 suspends only the RTP streaming while leaving the RTCP session active for QoS monitoring.
- The “legacy” method (which is defined in RFC 2543) sets the connection line of the SDP offer to “c=0.0.0.0”. If the remote endpoint accepts this proposal, both RTP and RTCP are disabled.

The Global Call IP call control library supports both methods of suspending media streaming.

4.8.8.1 Suspending RTP Streaming Only

To place an existing call on hold by suspending only the RTP streaming, an application first inserts a specially configured capability parameter element into a GC_PARM_BLK, then passes that parameter block in a call to **gc_ReqModifyCall( )**. The parameter element conforms to the following:

GCSET_CHAN_CAPABILITY
   IPPARM_LOCAL_CAPABILITY
   - value = IP_CAPABILITY data structure with direction field set to IP_CAP_DIR_LCLRTPINACTIVE

All of the other fields in the IP_CAPABILITY structure should be set to the current values for the active media session. The application can start with a copy of the IP_CAPABILITY structure that was retrieved as part of the connection information as described in Section 4.7.2, “Getting Media Streaming Status and Connection Information”, on page 160, and then modify only the direction field before inserting the parameter element into the GC_PARM_BLK.

When suspending streaming, it is only necessary to include a single capability parameter element in the parameter block even if the active call is a full-duplex media session.

4.8.8.2 Suspending RTP and RTCP Streaming

To completely suspend an existing call by deactivating both the RTP streaming and the RTCP session, an application first inserts a specially configured capability parameter element into a GC_PARM_BLK, then passes that parameter block in a call to **gc_ReqModifyCall( )**. The parameter element conforms to the following:

GCSET_CHAN_CAPABILITY
   IPPARM_LOCAL_CAPABILITY
   - value = IP_CAPABILITY data structure with direction field set to IP_CAP_DIR_LCLRTPRTCPINACTIVE

As in the similar case of suspending RTP only, all of the fields in the IP_CAPABILITY structure except for the direction field should be set to the current values for the active media session. The application can start with a copy of the IP_CAPABILITY structure that was retrieved as part of the connection information as described in Section 4.7.2, "Getting Media Streaming Status and Connection Information", on page 160, and then modify only the direction field before inserting the parameter element into the GC_PARM_BLK.

When suspending streaming, it is only necessary to include a single capability parameter element in the parameter block even if the active call is a full-duplex session.

4.8.8.3 Retrieving a Held Call

Retrieving a held call is a matter of sending a re-INVITE with a "normal" SDP offer (non-zero address in the "c=" line and non-inactive direction parameter in the "m=" line).

For a full-duplex connection, a Global Call application does this by inserting a pair of parameter elements that specify media capabilities for receive and transmit directions. The parameter elements are configured as follows:

GCSET_CHAN_CAPABILITY
   IPPARM_LOCAL_CAPABILITY
   - value = IP_CAPABILITY data structure with direction field set to IP_CAP_DIR_LCLRECEIVE

GCSET_CHAN_CAPABILITY
   IPPARM_LOCAL_CAPABILITY
   - value = IP_CAPABILITY data structure with direction field set to IP_CAP_DIR_LCLTRANSMIT

For a half-duplex connection, a Global Call application inserts a single parameter element as follows:

GCSET_CHAN_CAPABILITY
   IPPARM_LOCAL_CAPABILITY
   - value = IP_CAPABILITY data structure with direction field set to IP_CAP_DIR_LCLRECVONLY or IP_CAP_DIR_LCLSENDONLY

Note that there is no requirement that a session must be re-activated in the same mode that it was in when it was inactivated. For example, a session that was in full-duplex mode when it was put on hold can be retrieved from hold as a half-duplex session or vice versa.

If the application wishes to reactivate the held call with the same codec properties as when the call was placed on hold, it must populate all fields of the IP_CAPABILITY structure except the direction with the original values. This can be accomplished by using copies of the IP_CAPABILITY structure that was used in the on-hold re-INVITE request and modifying the direction field in each, or by using both of the IP_CAPABILITY structures that were retrieved as the connection information from the original INVITE dialog (see Section 4.7.2, "Getting Media Streaming Status and Connection Information", on page 160, for details).

Alternatively, the application can modify the properties of the streaming session (for example, changing to a different codec) at the same time that it retrieves the call from hold. To do this, the application simply builds a new pair of parameter elements (or a single element for half duplex) that specify the desired media properties and direction values.

4.9 Setting and Retrieving SIP Message Header Fields

The Dialogic® Global Call API supports the setting and retrieving of SIP message header fields in various SIP message types, including INFO, INVITE, NOTIFY, OPTIONS, REFER, and SUBSCRIBE requests. These messages may be implicitly created and sent as a result of a Global Call function call (for example, **gc_MakeCall( )** sends INVITE, **gc_InvokeXfer( )** sends REFER, and **gc_ReqService( )** sends REGISTER), or they may be messages that are explicitly constructed and then sent via **gc_Extension( )**, such as INFO or NOTIFY requests. On the receiving side, the messages are passed to the application as GCEV_OFFERED, GCEV_REQ_XFER, GCEV_CALLINFO, or GEEV_EXTENSION events, depending on the SIP request type, with the message information contained in the metaevent. The SIP header access feature is described in the following topics:

- SIP Header Access Overview
- Enabling Access to SIP Header Information
- Enabling Long Header Values
- Registering SIP Header Fields to be Retrieved
- Setting SIP Header Fields for Outbound Messages
- Retrieving SIP Message Header Fields
- SIP Header for BYE and CANCEL Messages
- Configuring Persistent SIP Headers
- Retrieving SIP "To tag" for Inbound Calls

4.9.1 SIP Header Access Overview

The Dialogic® Global Call API library provides a uniform mechanism for setting SIP header fields in SIP messages using a single Global Call parameter definition (namely IPSET_SIP_MSGINFO / IPPARM_SIP_HDR). This new mechanism is intended to replace the previous header access mechanism that relied on header-specific parameter definitions. Among the advantages of the new mechanism are:

- supports all SIP header fields, including optional and proprietary fields
- directly extensible to support new header fields
- field content length can exceed 255 bytes
- uniform programming approach
- application can register to receive only the header fields it needs to access from incoming messages

Header Fields in Outgoing SIP Messages

After access to SIP message information has been enabled (see Section 4.9.2, “Enabling Access to SIP Header Information”, on page 181), an application sets SIP message header fields for outgoing messages by inserting the set ID / parm ID pair and the parameter value (header contents) for each field into a GC_PARM_BLK using **gc_util_insert_parm_ref_ex( )** or **gc_util_insert_parm_val( )**. The application uses the IPSET_SIP_MSGINFO parameter set ID and IPPARM_SIP_HDR parameter ID to set any SIP header field. The parameter value must start with the header name and must conform to the SIP specifications for content, syntax, and punctuation.

Once the GC_PARM_BLK is composed, the application can pass that parm block as a parameter in a Global Call function that directly sends a message (such as **gc_Extension( )**, which is used to send messages like INFO or OPTIONS, or **gc_ReqService( )**, which is used to send REGISTER requests) or can preset the header fields for the next message to be sent by calling the **gc_SetUserInfo( )** function. The use of **gc_SetUserInfo( )** to preset SIP message header fields for the next message is only recommended when using **gc_MakeCall( )**. For messages that are sent directly (using **gc_Extension( )**, for example) the preferred method is to pass the parameter block directly to the function, because a preset header is always used for the very next message sent, which might not be the intended message. When using **gc_SetUserInfo( )** to preset SIP message header fields, the **duration** parameter must be set to GC_SINGLECALL, and the information is not transmitted until the next Global Call function that sends a SIP message is issued.

Table 5 shows the relationship between some of the most common SIP header fields, the SIP messages that commonly use them, and the Global Call functions that are used to set the headers and send the message.

Note: The Dialogic® Global Call API library handles the SIP Request-URI exactly like a standard SIP header field even though it is technically distinct from the header fields in a SIP message.

Table 5. Common Header Fields in Outbound SIP Messages

SIP header field	SIP message	Global Call function to set / send message
Accept	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
Accept-Encoding	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
Accept-Language	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
Allow	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
Call-ID	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
‡ From and To header fields are not set in INVITE messages using SIP message information parameters.		

Table 5. Common Header Fields in Outbound SIP Messages (Continued)

SIP header field	SIP message	Global Call function to set / send message
Contact (display string and URI separately accessible separately using field-specific parameters)	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REGISTER	gc_ReqService( )
Content-Disposition	INFO	gc_Extension( )
Content-Encoding	INFO	gc_Extension( )
Content-Length	INFO	gc_Extension( )
Content-Type	INFO	gc_Extension( )
Diversion (URI separately accessible via field-specific parameter)	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
Event	NOTIFY, SUBSCRIBE	gc_Extension( )
Expires	SUBSCRIBE	gc_Extension( )
From (display string separately accessible via field-specific parameter)	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
	REGISTER	gc_ReqService( )
Reason	BYE, CANCEL	gc_SetConfigData( ) to register Reason generic header gc_SetUserInfo( ) to send Reason generic header
Refer-To	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
Referred-By	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
Replaces	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled

‡ From and To header fields are not set in INVITE messages using SIP message information parameters.

Table 5. Common Header Fields in Outbound SIP Messages (Continued)

SIP header field	SIP message	Global Call function to set / send message
Request-URI	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
	REGISTER	gc_ReqService( )
Require	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REGISTER	gc_ReqService( )
Supported	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REGISTER	gc_ReqService( )
To (display string separately accessible via field-specific parameter)	INVITE	gc_SetUserInfo( ) / gc_MakeCall( )
	INFO, NOTIFY, SUBSCRIBE	gc_Extension( )
	OPTIONS	gc_Extension( ) if E_SIP_OPTIONS_Access is enabled
	REFER	gc_SetUserInfo( ) / gc_InvokeXfer( ) if call transfer is enabled
	REGISTER	gc_ReqService( )

‡ From and To header fields are not set in INVITE messages using SIP message information parameters.

Header Fields in Incoming SIP Messages

For incoming SIP messages, the Dialogic® Global Call API library packages the header fields that the application has registered to receive as parameters in the GC_PARM_BLK that is associated with the Global Call event that notifies the application of the message. The application retrieves the parameter block by calling **gc_GetMetaEvent( )**, and can then extract the contents of the various header fields from the GC_PARM_BLK. The application must complete the retrieval of the necessary SIP message header information (for example, by copying it into its own buffer) before the next call to **gc_GetMetaEvent( )**, since the parameter block is no longer available from the metaevent buffer once the next **gc_GetMetaEvent( )** call is issued.

In addition to the header fields that the application specifically registers to receive, the GC_PARM_BLK for a message-related Global Call event may contain one or more of the header-specific parameters that were used in the previous header access methodology. It is important to note that these parameters are limited to a 255 byte data length and may potentially contain a truncation of the a header field's contents.

Table 6 lists some common SIP header fields along with the SIP message that commonly contains them and the Global Call event that is used to convey the message information to the application.

Note: The From URI and To URI in incoming INVITE messages are accessible using the **gc_GetCallInfo( )** function; see Section 8.3.10, "gc_GetCallInfo( ) Variances for IP", on page 551, for more information. In all other cases, applications must access the complete From and To header fields in order to access the URIs.

Table 6. Common Header Fields in Inbound SIP Messages

SIP header	SIP message	Global Call event
Accept	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
Accept-Encoding	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
Accept-Language	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
Allow	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
Call-ID †	INVITE	GCEV_OFFERED
	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
Contact (display string and URI separately returned in field-specific parameters)	INVITE	GCEV_OFFERED
	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
	REFER	GCEV_REQ_XFER if call transfer is enabled
	3xx to 6xx responses	GCEV_DISCONNECTED
Content-Disposition	INFO	GC_CALLINFO
Content-Encoding	INFO	GC_CALLINFO
Content-Length	INFO	GC_CALLINFO
Content-Type	INFO	GC_CALLINFO
Diversion (URI separately returned in field-specific parameter)	INVITE	GCEV_OFFERED
	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
Event †	NOTIFY, SUBSCRIBE	GCEV_EXTENSION
Expires †	SUBSCRIBE	GCEV_EXTENSION
From ‡ (display string and full header also returned in header-specific parameters)	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
	REFER	GCEV_REQ_XFER if call transfer is enabled

† Header field also accessible via field-specific parameter define.
‡ From and To header fields are not retrieved from INVITE messages using SIP message information parameters.

Table 6. Common Header Fields in Inbound SIP Messages (Continued)

SIP header	SIP message	Global Call event
Reason	BYE, CANCEL	GCEV_DISCONNECTED
Referred-By †	INVITE	GCEV_OFFERED
	REFER	GCEV_REQ_XFER if call transfer is enabled
Replaces †	INVITE	GCEV_OFFERED
	REFER	GCEV_REQ_XFER if call transfer is enabled
Request-URI †	INVITE	GCEV_OFFERED
	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
	REFER	GCEV_REQ_XFER if call transfer is enabled
Require	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
Supported	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
To ‡ (display string and full header also returned in header-specific parameters)	INFO, NOTIFY, SUBSCRIBE	GCEV_EXTENSION
	OPTIONS	GCEV_EXTENSION if E_SIP_OPTIONS_Access is enabled
	REFER	GCEV_REQ_XFER if call transfer is enabled

† Header field also accessible via field-specific parameter define.
‡ From and To header fields are not retrieved from INVITE messages using SIP message information parameters.

API Functions for Long Header Values

Because some SIP header fields (particularly those that allow multiple values to be contained in a single header field in a comma-delimited list) can be arbitrarily long, the Global Call IP library has been extended to remove the inherent 255 byte data length limitation for parameters that are contained in a GC_PARM_BLK data structure.

When using the IPSET_SIP_MSGINFO/IPPARM_SIP_HDR parameter, and the new, extended **gc_util_... _ex( )** utility functions (see Section 8.2, "IP-Specific Dialogic® Global Call API Functions", on page 484, for complete information on these functions), the maximum length of the parameter value can be configured by the application using IPCCLIB_START_DATA.max_parm_data_size before the library is started.When an application has configured an extended maximum parameter length it *must not* make any attempt to access parameter block data directly; instead, the new, extended **gc_util_..._ex( )** utility functions, which handle the extended-length data properly, should *always* be used.

The new, extended **gc_util_..._ex( )** utility functions are backwards compatible and can be used with any GC_PARM_BLOCK regardless of whether it contains parameters that may exceed 255 bytes. For this reason, it is recommended that the extended functions should always be used in application code that accesses SIP header fields.

Field-Specific Parameters for SIP Header Access

Certain standard SIP header fields can be accessed using header-specific Global Call parameter IDs instead of the generic IPSET_SIP_MSGINFO / IPPARM_SIP_HDR parameter that is described in above.

The use of the header-specific parameter IDs has the following limitations:

- This mechanism is being deprecated. The defines will remain in the IP Call Control library for backward compatibility, but no further development will be done on these parameters and no issues or problems will be fixed.
- The parameter data associated with header-specific parameter IDs (that is, the header field contents) is limited to 255 bytes. You **must** use the generic IPPARM_SIP_HDR parameter ID rather than a header-specific parameter ID to handle any header field that is longer than 255 bytes.

Table 7 lists the SIP header fields that have field-specific parameter IDs, all of which are deprecated. The table also indicates the size defines that correspond to each parameter ID, each of which is equated to 255. Note that some of these parameters provide access to specific portions of the corresponding header field, such as only the URI or only the display string.

Note that there is no advantage to using the field-specific parameters that identify complete fields when setting SIP headers. Parameters that access only a part of the corresponding header field (i.e., just the URI or just the display string) may provide some convenience but should be used with caution because all of these parameter IDs are being deprecated.

When a SIP message is received, the associated parm block contained in the event metadata contains an element that uses the header-specific parameter ID for each corresponding header field that is present in the message, regardless of whether the same field is registered to be received using the generic IPSET_SIP_MSGINFO / IPPARM_SIP_HDR parameter

Table 7. Field-Specific Parameters (Deprecated) for SIP Header Access

Header Field Name	Set ID and Parameter ID	Maximum Data Length Define †
Call-ID ††	IPSET_SIP_MSGINFO • IPPARM_CALLID_HDR	IP_CALLID_HDR_MAXLEN
Contact display string	IPSET_SIP_MSGINFO • IPPARM_CONTACT_DISPLAY	IP_CONTACT_DISPLAY_MAXLEN
Contact URI	IPSET_SIP_MSGINFO • IPPARM_CONTACT_URI	IP_CONTACT_URI_MAXLEN
Diversion URI	IPSET_SIP_MSGINFO • IPPARM_DIVERSION_URI	IP_DIVERSION_MAXLEN
Event	IPSET_SIP_MSGINFO • IPPARM_EVENT_HDR	IP_EVENT_HDR_MAXLN

† The value for each listed parameter ID is a character array with the maximum size of the array (including the NULL) equal to the corresponding maximum length define.
†† Directly setting the Call-ID header field using this parameter overrides any Call-ID value that is set using the IPSET_CALLINFO / IPPARM_CALLID parameter.

Table 7. Field-Specific Parameters (Deprecated) for SIP Header Access (Continued)

Header Field Name	Set ID and Parameter ID	Maximum Data Length Define †
Expires	IPSET_SIP_MSGINFO • IPPARM_EXPIRES_HDR	IP_EXPIRES_HDR_MAXLEN
From display string	IPSET_SIP_MSGINFO • IPPARM_FROM_DISPLAY	IP_FROM_DISPLAY_MAXLEN
From (complete header field, with display string, URI, and parameters)	IPSET_SIP_MSGINFO • IPPARM_FROM	IP_FROM_MAXLEN
Referred-By	IPSET_SIP_MSGINFO • IPPARM_REFERRED_BY	IP_REFERRED_BY_MAXLEN
Replaces (parameter in Refer-To header field for attended call transfers)	IPSET_SIP_MSGINFO • IPPARM_REPLACES	IP_REPLACES_MAXLEN
Request-URI	IPSET_SIP_MSGINFO • IPPARM_REQUEST_URI	IP_REQURI_MAXLEN
To display string	IPSET_SIP_MSGINFO • IPPARM_TO_DISPLAY	IP_TO_DISPLAY_MAXLEN
To (complete header field, with display string, URI, and parameters)	IPSET_SIP_MSGINFO • IPPARM_TO	IP_TO_MAXLEN

† The value for each listed parameter ID is a character array with the maximum size of the array (including the NULL) equal to the corresponding maximum length define.
†† Directly setting the Call-ID header field using this parameter overrides any Call-ID value that is set using the IPSET_CALLINFO / IPPARM_CALLID parameter.

4.9.2 Enabling Access to SIP Header Information

The ability to set and retrieve information from SIP message header fields is an optional feature that can be enabled or disabled at the time the **gc_Start( )** function is called.

The mandatory **INIT_IP_VIRTBOARD( )** utility function populates the IP_VIRTBOARD structure with default values. The default value of the sip_msginfo_mask field in the IP_VIRTBOARD structure disables application access to all SIP message header fields. The value IP_SIP_MSGINFO_ENABLE (possibly OR'ed with other defined mask values) must be set into the sip_msginfo_mask field for each IPT board device on which the feature is to be enabled. The following code snippet provides an example for two virtual boards:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE; /* override SIP message default */
ip_virtboard[1].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE; /* override SIP message default */
```

Setting the value IP_SIP_MSGINFO_ENABLE (possibly OR'ed with other bitmask values) in the sip_msginfo_mask field enables overall set/retrieve access to SIP header fields for the virtual board. Enabling and disabling access to individual SIP header fields is **not** supported.

Note: Features that are enabled or configured via the IP_VIRTBOARD structure cannot be disabled or reconfigured once the library has been started. All items set in this data structure take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

4.9.3 Enabling Long Header Values

The ability to set and retrieve SIP message header fields that exceeds 255 bytes in length is an optional feature that can be enabled at the time the **gc_Start( )** function is called. The maximum length for SIP header fields is configured in the IPCCLIB_START_DATA data structure and applies to all virtual boards in the system.

The mandatory **INIT_IPCCLIB_START_DATA( )** utility function populates the IPCCLIB_START_DATA structure with default values. The default value of the max_parm_data_size field in the IPCCLIB_START_DATA structure sets the maximum data length for parameter data in a GC_PARM_BLK structure at 255 for backwards compatibility. If the application requires the ability to send and receive SIP header fields that are longer than this default maximum length (up to a maximum of 4096 bytes), it can overwrite the default value after initializing the IPCCLIB_START_DATA but before calling **gc_Start( )**. The following code snippet provides an example of setting a maximum length of 1024 bytes for SIP header fields (and other parameter types that specifically support extended-length data) for each of two virtual boards:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ipcclibstart.max_parm_data_size = 1024;  /* set maximum SIP header length to 1k */
ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE; /* override SIP message default */
ip_virtboard[1].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE; /* override SIP message default */
```

Note: Features that are enabled or configured via the IPCCLIB_START_DATA structure cannot be disabled or reconfigured once the library has been started. All items set in this data structure take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

4.9.4 Registering SIP Header Fields to be Retrieved

In order to receive specific SIP header fields, the application must register the field names. The registration is accomplished by constructing a GC_PARM_BLK where each element contains registration information for an individual header field to be retrieved, then calling **gc_SetConfigData( )** to set the registration list in the library. Each element in the parm block uses the IPSET_CONFIG set ID and the parameter ID IPPARM_REGISTER_SIP_HEADER, plus the header field name as the parameter value. The registration of header fields only needs to be performed once for a board device, but the application is free to set a different registration list at some other time, if desired.

When registering standard SIP header fields (that is, header fields which are defined in the IETF RFC documents), the field names must be spelled consistently so that the SIP stack can recognize the header fields properly. Be certain that the spelling matches the following list (noting that case does not matter). Note that Request-URI is handled just like a standard header field, even though it is technically distinct from true header fields.

Note: In this list, header fields that are assumed to be accessible to applications to support functionality documented in this guide are marked with a †, and fields that are accessible in part or in whole via deprecated header-specific parameter defines are marked with an *.

- Accept †
- Accept-Encoding †
- Accept-Language †
- Allow †
- Allow-Events
- Authentication
- Authentication-Info
- Authorization
- Call-ID † *
- Contact † *
- Content-Disposition †
- Content-Encoding †
- Content-Language †
- Content-Length
- CSeq
- Date
- Diversion † *
- Event † *
- Expires † *
- From † *
- Max-Forwards
- Min-Expires
- Min-SE
- Proxy-Authenticate
- Proxy-Authorization
- RAck
- Reason
- Referred-By † *
- Refer-To
- Replaces † *
- Request-URI † *

- Require †
- Retry-After
- Route
- RSeq
- Session-Expires
- Subscription-State
- Supported †
- To † *
- Unsupported
- Via
- Warning
- WWW-Authenticate †

The following code snippet illustrates how an application would register to receive the six SIP header fields required for use of OPTIONS messages that are not accessible via header-specific parameter defines.

Note: This example uses **gc_util_insert_parm_ref( )** rather than **gc_util_insert_parm_ref_ex( )** because it is known that header field name strings are short and never come close to the 255 byte data length limit.

```
// all devices are open
// register SIP headers to monitor

GC_PARM_BLKP parmblkp = NULL;

char *pAccept = "Accept";
char *pAcceptEnc = "Accept-Encoding";
char *pAcceptLang = "Accept-Language";
char *pAllow = "Allow";
char *pRequire = "Require";
char *pSupported = "Supported";

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pAccept) + 1,
                        pAccept);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pAcceptEnc) + 1,
                        pAcceptEnc);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pAcceptLang) + 1,
                        pAcceptLang);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pAllow) + 1,
                        pAllow);
```

```
gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pRequire) + 1,
                        pRequire);

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_REGISTER_SIP_HEADER,
                        strlen(pSupported) + 1,
                        pSupported);

long request_id = 0;

// SetConfigData
// NOTE: device handle is a handle to the board device
if (gc_SetConfigData(GCTGT_CCLIB_NETIF, boarddevh, parmblkp, 0,
                     GCUPDATE_IMMEDIATE, &request_id, EV_ASYNC) != GC_SUCCESS)
{
   sprintf(str, "gc_SetConfigData(boarddevh=%ld) Failed registering SIP headers", boarddevh);
   printf ("%s"str);
}

gc_util_delete_parm_blk(parmblkp);
```

4.9.5 Setting SIP Header Fields for Outbound Messages

Note that it is not necessary for applications to register in advance the header field types that it will be setting (as described in Section 4.9.4, “Registering SIP Header Fields to be Retrieved”, on page 182). Registration of header field names is only required when the application needs to *retrieve* those header fields from received messages.

Assuming that SIP message information access was enabled when the virtual board was started, applications set SIP message header fields by inserting the set ID/parm ID and value string for each field being set into a GC_PARM_BLK using **gc_util_insert_parm_ref_ex( )** or **gc_util_insert_parm_val( )**, and then either setting the header fields for the next message to be sent by calling the **gc_SetUserInfo( )** function or immediately sending the message by calling **gc_Extension( )** or another Global Call function that causes a SIP message to be sent.

When calling **gc_SetUserInfo( )** to preset SIP message header fields (which is only recommended when using the **gc_MakeCall( )** function), the **duration** parameter must be set to GC_SINGLECALL, and the information is not transmitted until the next Global Call function that sends a SIP message is issued. Note that the preset header fields will be sent in the next SIP message, so that the application must ensure that no other Global Call function is called before **gc_MakeCall( )**.

Calling the **gc_SetUserInfo( )** function results in the following behavior:

- SIP message header fields that are set do not take effect until **gc_MakeCall( )** or another function that transmits a SIP message is issued.
- Using the **gc_SetUserInfo( )** does not affect incoming SIP messages on the same channel.
- Any SIP message header fields that are set only affect the next Dialogic® Global Call API function call.

- The **gc_SetUserInfo( )** function fails with GC_ERROR if the sip_msginfo_mask field in the IP_VIRTBOARD structure is not set to IP_SIP_MSGINFO_ENABLE. When **gc_ErrorInfo( )** is called in this case, the error code is IPERR_BAD_PARAM.

The **gc_Extension( )** function is typically used when sending supplementary SIP messages, such as INFO or OPTIONS. It is possible to use the **gc_SetUserInfo( )** function to set the header field before sending the message with the **gc_Extension( )** function call or other function that directly produces a SIP request (such as **gc_ReqService( )** for SIP REGISTER requests), but that approach is not recommended. This is the case because the preset header fields will be used in the very next SIP message that is sent, so the application must ensure that no other Global Call function is called before the intended function.

Refer to Table 5, "Common Header Fields in Outbound SIP Messages", on page 175, to see the correspondence between the most common SIP header fields, the supported SIP messages in which these header fields are commonly set, and the Global Call functions that are called to transmit these messages.

Applications should use the IPSET_SIP_MSGINFO set ID and the IPPARM_SIP_HDR parameter ID when setting SIP header fields in the GC_PARM_BLK. This same set ID/parm ID pair can be used to set any settable SIP header field, whether it is a required field, an optional one, or a proprietary one. In each case, the parameter value that is inserted into the parameter block is a string that is the complete header field to be sent, starting with the header field name and including all required syntax elements and punctuation.

As permitted in RFC 3261 and other IETF standards, applications can insert multiple header fields of the same type with different values, or can insert a single header field with multiple values in a comma-delimited string.

When an optional or proprietary header field is being set, the IP call control library and SIP stack simply pass through the header contents as specified by the application. The library and stack check for the presence of all header fields that are required for a specific SIP request or reply, and if such a required field is being set by the application, there may be some level of validation performed, as well. Further details regarding validation and error checking will be provided in future revisions of this document.

Note: Setting SIP message header information requires a detailed knowledge of the SIP protocol and its relationship to Dialogic® Global Call API. The application has the responsibility to ensure that the correct SIP message information is set before calling the appropriate Dialogic® Global Call API function to send the message.

Note that header-specific Global Call parameter IDs exist for some standard SIP header fields, but that there is no advantage to using the those parameters when setting SIP headers if the parameter accesses a complete header field. Parameters that access only a part of the corresponding header field (i.e., just the URI or just the display string) may provide some convenience, but this approach is not recommended because all of the header-specific parameter defines are being deprecated. Table 8 identifies the parameter IDs that provide access to partial header fields.

Table 8. Parameter IDs for Partial Header Field Access (Deprecated)

Header Field Name	Set ID and Parameter ID	Maximum Data Length Define †
Contact display string	IPSET_SIP_MSGINFO • IPPARM_CONTACT_DISPLAY	IP_CONTACT_DISPLAY_MAXLEN
Contact URI	IPSET_SIP_MSGINFO • IPPARM_CONTACT_URI	IP_CONTACT_URI_MAXLEN
Diversion URI	IPSET_SIP_MSGINFO • IPPARM_DIVERSION_URI	IP_DIVERSION_MAXLEN
From display string	IPSET_SIP_MSGINFO • IPPARM_FROM_DISPLAY	IP_FROM_DISPLAY_MAXLEN
Replaces (parameter in Refer-To header field for attended call transfers)	IPSET_SIP_MSGINFO • IPPARM_REPLACES	IP_REPLACES_MAXLEN
To display string	IPSET_SIP_MSGINFO • IPPARM_TO_DISPLAY	IP_TO_DISPLAY_MAXLEN
† The value for each listed parameter ID is a character array with the maximum size of the array (including the NULL) equal to the corresponding maximum length define, all of which are equated to 255.		

The following code snippet shows how to set the Request-URI header information before issuing **gc_MakeCall( )**. This translates to a SIP INVITE message with the specified Request-URI.

```
#include "gclib.h"
..
..
GC_PARM_BLK  *pParmBlock = NULL;
char         *pDestAddrBlk = "1111@127.0.0.1\0";
char         *pReqURI = "sip:2222@127.0.0.1\0";


/* Insert SIP Request-URI */
/* Add 1 to strlen for the NULL termination character */
gc_util_insert_parm_ref_ex(&pParmBlock,
                           IPSET_SIP_MSGINFO,
                           IPPARM_REQUEST_URI,
                           (unsigned long) (strlen(pReqURI) + 1),
                           pReqURI);

/* Set Call Information */
gc_SetUserInfo(GCTGT_GCLIB_CHAN, ldev, pParmBlock, GC_SINGLECALL);

gc_util_delete_parm_blk(pParmBlock);

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_TRANSPARENT;

/* calling the function with the MAKECALL_BLK,
the INVITE "To" field will be: 1111@127.0.0.1
the INVITE RequestURI will be: sip:2222@127.0.0.1
*/
gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout, EV_ASYNC);
```

The following code snippet illustrates how an application can set a proprietary header called Remote-Party-ID. This header is a CableLabs (DCS Group) sponsored extension to transmit trusted Caller Identity and Privacy ISUP indications which have not been standardized for translation across SIP networks.

```
GC_PARM_BLKP parmblkp = NULL;
char *pRemotePartyIdHeader = "Remote-Party-ID:Alice";

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long) (strlen(pRemotePartyIdHeader) + 1),
                           pRemotePartyIdHeader);

gc_SetUserInfo(GCTGT_GCLIB_CRN, crn, parmblkp, GC_SINGLECALL);

gc_util_delete_parm_blk(parmblkp);

// transmit SIP message to network
...
...
```

4.9.6 Retrieving SIP Message Header Fields

The reception of most SIP requests and replies is reported to the application by means of a Global Call event, with information about the type of message contained in the metaevent data. If SIP message information access was enabled when the virtual board was started (see Section 4.9.2, "Enabling Access to SIP Header Information", on page 181), the metaevent will also contain information from SIP header fields. The application processes the Dialogic® Global Call API event using the **gc_GetMetaEvent( )** function, and then processes the GC_PARM_BLK using Global Call utility functions.to retrieve the message type information and individual SIP header fields of interest.

Note: The application must retrieve the necessary SIP message header field information by copying the GC_PARM_BLK into its own buffer with **gc_util_copy_parm_blk( )** before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the header information no longer available from the metaevent buffer.

Refer to Table 6, "Common Header Fields in Inbound SIP Messages", on page 178, to see the correspondence between SIP message type and Global Call event type for common SIP header fields.

If the application has registered one or more SIP header fields to be received (as described in Section 4.9.4, "Registering SIP Header Fields to be Retrieved", on page 182), the GC_PARM_BLK contains a separate parameter element for each registered field that was present in the received message. Each of these elements contains the IPSET_SIP_MSGINFO set ID and the IPPARM_SIP_HDR parameter ID. The associated data buffer contains the entire header field, complete with name, value, and any optional parameters. It is the application's responsibility to parse the data to determine the type of the header field.

Note: If a header field that the application has registered to receive is longer than the maximum parameter length (as configured via IPCCLIB_STARTDATA.max_parm_data_size at library start-up), the header field will be truncated in the IPSET_SIP_MSGINFO / IPPARM_SIP_HDR parameter element. Applications can check for this situation by calling **gc_ResultInfo( )** upon receiving any Global Call event that corresponds to a SIP message. A result value of IPEC_SipHeaderTruncation indicates that one or more of the SIP header values in the GC_PARM_BLK associated with the event were truncated.

If the received message contains multiple header field rows with the same field name, there will be a single parameter element in the GC_PARM_BLK data structure with header values appearing as a comma-separated list, in the same order as that of the incoming message.

The following code snippet illustrates how an application retrieves registered SIP header fields when a Global Call event has been received. The example assumes that the header field name has been registered and that the event has already been received.

```
char               siphdr[IP_SIP_HDR_MAXLEN];
GC_PARM_DATA_EXT   parm_data;
INIT_GC_PARM_DATA_EXT(&parm_data);

while ((ret = gc_util_next_parm_ex(pParmBlock, &parm_data)) == GC_SUCCESS)
{
   switch (parm_data.parm_ID)
   {
      case IPPARM_SIP_HDR:
         strncpy(siphdr, (char*)parm_data.pData, parm_data.data_size);
         siphdr[parm_data.data_size]='\0';
         sprintf(m_DisplayString, "\t\tGeneric Sip Header = %s", siphdr);
         printf("%s", m_DisplayString);
         break;
   }
}
```

In addition to the IPPARM_SIP_HDR elements that correspond to the registered header fields, the parm block will also contain elements that use the deprecated field-specific parameter IDs listed in Table 7, "Field-Specific Parameters (Deprecated) for SIP Header Access", on page 180. Some of these field-specific parameters provide access to a specific part of the corresponding header field (specifically just the display string or just the URI) rather than the complete header field.

The following code demonstrates how to copy the Request-URI from a GCEV_OFFERED event using the (deprecated) field-specific parameter ID IPPARM_REQUEST_URI. The GC_PARM_BLK structure containing the data is referenced via the extevtdatap pointer in the METAEVENT structure. In this particular scenario, the GCEV_OFFERED event is generated as a result of receiving an INVITE message.

```
#include "gclib.h"
..
..
METAEVENT          metaevt;
GC_PARM_DATA_EXT   parm_data;
GC_PARM_BLK        *pParmBlock = NULL;
char               reqestURI[IP_REQUEST_URI_MAXLEN];

/* Get Meta Event */
gc_GetMetaEvent(&metaevt);

switch(metaevt->evttype)
   {
    .
    .
    .
   case GCEV_OFFERED:
      currentCRN = metaevt->crn;
      pParmBlock = (GC_PARM_BLK*)(metaevt->extevtdatap);
      INIT_GC_PARM_DATA_EXT(&parm_data);
      /* going thru each parameter block data*/
      while ((ret = gc_util_next_parm_ex(pParmBlock,&parm_data)) == GC_SUCCESS)
      {
         switch (parm_data.set_ID)
         {
         /* Handle SIP message information */
            case IPSET_SIP_MSGINFO:
               switch (parm_data.parm_ID)
               {
               /* Copy Request URI from parameter block */
                  case IPPARM_REQUEST_URI:
                     strncpy(requestURI, (char*) parm_data.pData, parm_data.data_size);
                     break;
               }
         }
      break;
      }
   .
   .
   .
}
```

4.9.7 SIP Header for BYE and CANCEL Messages

The application can send and receive a Reason header for SIP BYE and CANCEL messages via the generic Global Call SIP header access functionality.

With this feature, the application can add/append the Reason headers on outgoing BYE and CANCEL requests, which indicate why the call was terminated. In the event of a BYE or CANCEL request, the application can determine how to treat failures depending on the reason, and to transmit the reason code to the other leg of the call.

The existing Global Call methods for retrieving generic SIP headers remain the same for BYE and CANCEL messages. The application uses the **gc_SetConfigData( )** function on the incoming side to register the Reason generic header and uses the **gc_SetUserInfo( )** function to send the generic header. The GCEV_DISCONNECT parameter block contains the IPSET_SIP_MSGINFO and IPPARM_SIP_HDR parameter pair and data containing the generic header.

Examples

To allow for the manipulation of generic SIP headers in Global Call, add the following code before calling the **gc_Start( )** function:

```
IPCCLIB_START_DATA ipcclibstart;
IP_VIRTBOARD ip_virtboard[1];

long request_id = 0;
int index = 1;
GC_PARM_DATAP t_gcParmDatap = NULL;

CCLIB_START_STRUCT cc_Lib_Start[2] ;
GC_START_STRUCT gc_Lib_Start;

memset(&ipcclibstart, 0, sizeof(IPCCLIB_START_DATA));
memset(ip_virtboard, 0,sizeof(IP_VIRTBOARD)*1);

INIT_IPCCLIB_START_DATA(&ipcclibstart, 1, ip_virtboard);
ipcclibstart.max_parm_data_size = 1000;
```

```
     INIT_IP_VIRTBOARD(&ip_virtboard[0]);
     ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE; /* override SIP
                                                               *  message
                                                               */

     cc_Lib_Start[0].cclib_name = "GC_IPM_LIB";
     cc_Lib_Start[0].cclib_data = NULL;
     cc_Lib_Start[1].cclib_name = "GC_H3R_LIB";
     cc_Lib_Start[1].cclib_data = &ipcclibstart;
     gc_Lib_Start.cclib_list = cc_Lib_Start;
     gc_Lib_Start.num_cclibs = 2;

     if (gc_Start(&gc_Lib_Start) != GC_SUCCESS) {
}
```

Add the following code before Drop Call on the outbound side. It demonstrates how a Reason header is added to an outgoing BYE or CANCEL SIP request resulting from the **gc_DropCall( )** function.

```
     char *pReasonHdr = "reason: SIP cause=290 text=Call completed elsewhere";

     gc_util_insert_parm_ref(&parmblkp, IPSET_SIP_MSGINFO, IPPARM_SIP_HDR,
     (unsigned char)strlen(pReasonHdr ) + 1, pReasonHdr );
     gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->call[0].crn, parmblkp,
                    GC_SINGLECALL);
     gc_util_delete_parm_blk(parmblkp);
```

Add the following code upon receiving a GCEV_OPENEX event on the incoming side board device. This registers a Reason header that can be retrieved upon receiving a GCEV_DISCONNECT event.

```
     char *pReasonHdr = "reason";
     long request_id = 0;
     GC_PARM_BLKP  parmblkp = NULL;
     GC_PARM_DATA_EXT parmdata;
     GC_PARM_DATAP t_gcParmDatap = NULL;
     char siphdr[IP_SIP_HDR_MAXLEN];
     gc_util_insert_parm_ref(&parmblkp, IPSET_CONFIG,
     IPPARM_REGISTER_SIP_HEADER,
     strlen(pReasonHdr ) + 1,   pReasonHdr);

     if (gc_SetConfigData(GCTGT_CCLIB_NETIF, port[index].ldev, parmblkp, 0,
         GCUPDATE_IMMEDIATE,  &request_id, EV_ASYNC) != GC_SUCCESS)
```

Add the following code upon receiving a GCEV_DISCONNECT event on the incoming side:

```
     char *pReasonHdr = "reason";
       long request_id = 0;
       GC_PARM_BLKP  parmblkp = NULL;
       GC_PARM_DATA_EXT parmdata;
       GC_PARM_DATAP t_gcParmDatap = NULL;
       char siphdr[IP_SIP_HDR_MAXLEN];

     case GCEV_DISCONNECT:
           parmblkp = (GC_PARM_BLK *) metaevent.extevtdatap;
           while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap)){

           if(t_gcParmDatap->set_ID == IPSET_SIP_MSGINFO && t_gcParmDatap->parm_ID ==
              IPPARM_SIP_HDR){
           memcpy(&siphdr, (char*)t_gcParmDatap->value_buf, t_gcParmDatap->value_size);
           sprintf(str, "Generic Sip Header = %s", siphdr);
           printandlog(pline->index, MISC, NULL, str, 0);
        }
     }
```

4.9.8 Configuring Persistent SIP Headers

The application can configure a SIP header, such as a Contact header, for all outgoing SIP messages for all calls or a single SIP session in 1PCC or 3PCC mode. The value of the persistent SIP header remains in effect until the call is terminated or the application changes the value during the call.

Persistent headers are configured on a line device or CRN. The duration parameter in **gc_SetUserInfo( )** is used to determine the scope of the configuration. When setting the value of a SIP header, it is assumed that the assigned value's syntax complies with the requirements specified in the *IETF RFC 3261: SIP: Session Initiation Protocol* and the *IETF RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*. It is the responsibility of the application to ensure compliance.

The application can remove existing persistent headers by header name on the line device or CRN. The duration parameter in **gc_SetUserInfo( )** is used to determine the scope of the removal, for example, for all calls or a single SIP session. The application may want to set multiple persistent Contact headers for all outgoing messages. If the application wishes to overwrite the previous persistent header configuration, it has to remove that configuration before setting the new one. For example, the application may want to change the Contact header only for the current call (CRN), but wish to preserve the persistent Contact header for all other calls on the same line device.

Note: The application is not allowed to set SIP persistent headers across all line devices on a virtual board. Setting SIP headers on a virtual board only applies to standalone SIP transactions.

The Global Call library also supports the retrieval of SIP headers for incoming messages. See Section 4.9, "Setting and Retrieving SIP Message Header Fields", on page 174 for more information.

4.9.8.1 Enabling Persistent SIP Headers

The Global Call library allows the application to set SIP generic headers using the **gc_SetUserInfo( )** function on a CRN with GC_SINGLECALL duration. Using this method, only the next outgoing SIP message will consume the configured header.

To set persistent SIP headers, specify one of the following duration values in **gc_SetUserInfo( )** for the appropriate line device or CRN:

GC_ALLCALLS duration
: Use the same SIP generic header, such as Contact header, for all outgoing SIP messages on all SIP calls from a particular line device.

GC_SINGLE_SIP_SESSION duration
: Use the same SIP generic header for all outgoing SIP messages within one call only. This one call can be either on a CRN (for new or existing calls) or on a line device (for next established call). Configured SIP headers remain persistent until the call is terminated. The next call on the same line device uses the default SIP headers.

To remove a persistent SIP header, use this parameter ID in the IPSET_SIP_MSGINFO set ID:

IPPARM_SIP_HDR_REMOVE
: Remove persistent SIP header configuration by header name.

4.9.8.2 Examples

This section provides example code for configuring persistent SIP headers.

Set Persistent Headers on a Line Device for All Calls

```
#include "gclib.h"
..
..
GC_PARM_BLK *pParmBlock = NULL;
char ContactHeader[] = "Contact: <sip:user@example.com>";
char ContactHeaderName[] = "Contact";

//First remove existing header configuration
//This is optional

gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR_REMOVE,
                        (char)(strlen(ContactHeaderName)+1),
                        ContactHeaderName);

gc_SetUserInfo(GCTGT_GCLIB_CHAN, ldev, pParmBlock, GC_ALLCALLS);

gc_util_delete_parm_blk(pParmBlock);

pParmBlock=NULL;

//Set persistent Contact header
gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        (char)(strlen(ContactHeader)+1),
                        ContactHeader);

gc_SetUserInfo(GCTGT_GCLIB_CHAN, ldev, pParmBlock, GC_ALLCALLS);

gc_util_delete_parm_blk(pParmBlock);
```

Set Persistent Headers on a Line Device for a Single SIP Session

```
#include "gclib.h"
..
..
GC_PARM_BLK *pParmBlock = NULL;
char ContactHeader[] = "Contact: <sip:user@example.com>";
char ContactHeaderName[] = "Contact";

//First remove existing header configuration
//This is optional

gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR_REMOVE,
                        (char)(strlen(ContactHeaderName)+1),
                        ContactHeaderName);
```

```
gc_SetUserInfo(GCTGT_GCLIB_CHAN, ldev, pParmBlock, GC_SINGLE_SIP_SESSION);

gc_util_delete_parm_blk(pParmBlock);

pParmBlock=NULL;

//Set persistent Contact header
gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        (char)(strlen(ContactHeader)+1),
                        ContactHeader);

gc_SetUserInfo(GCTGT_GCLIB_CHAN, ldev, pParmBlock, GC_SINGLE_SIP_SESSION);

gc_util_delete_parm_blk(pParmBlock);
```

Set Persistent Headers on a Call (CRN) for a Single SIP Session

```
#include "gclib.h"
..
..
GC_PARM_BLK *pParmBlock = NULL;
char ContactHeader[] = "Contact: <sip:user@example.com>";
char ContactHeaderName[] = "Contact";

//First remove existing header configuration
//This is optional

gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR_REMOVE,
                        (char)(strlen(ContactHeaderName)+1),
                        ContactHeaderName);

gc_SetUserInfo(GCTGT_GCLIB_CRN, crn, pParmBlock, GC_SINGLE_SIP_SESSION);

gc_util_delete_parm_blk(pParmBlock);

pParmBlock=NULL;

//Set persistent Contact header
gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_SIP_MSGINFO,
                        IPPARM_SIP_HDR,
                        (char)(strlen(ContactHeader)+1),
                        ContactHeader);

gc_SetUserInfo(GCTGT_GCLIB_CRN, crn, pParmBlock, GC_SINGLE_SIP_SESSION);

gc_util_delete_parm_blk(pParmBlock);
```

4.9.9 Retrieving SIP "To tag" for Inbound Calls

The application can retrieve a SIP "To tag" for inbound calls for both first party call control (1PCC) and third party call control (3PCC) operating modes.

Note: For outbound call support, all SIP header information can be obtained from the event data when the application receives the GCEV_ALERTING event.

With this feature, the application can retrieve a SIP "To tag" using the **gc_GetCallInfo( )** function shortly after the GCEV_ACCEPT event is received, and prior to the call being connected. If the application answers a call immediately after receiving the GCEV_OFFERED, the SIP “To tag” will be available after the application calls the **gc_AnswerCall( )** function and sends a SIP 200 OK message to the remote end.

The IPPARM_GETCALLINFOUPDATE parameter ID in IPSET_CONFIG supports this feature at the board level. The feature is enabled using **gc_SetConfigData( )**.

Because a small time window exists between when the GCEV_ACCEPT event is returned and when the updated call information is available, a separate event is provided. When the Global Call API library is ready to provide the “To tag” information, a GCEV_EXTENSION of type IPEXTID_GETCALLINFOUPDATE is sent to the application. This extension ID notifies the application that call information has been updated and is retrievable using **gc_GetCallInfo( )**.

Note: Enabling “To tag” retrieval is only supported at the board level. Once enabled, all channels are enabled.

The application can use **gc_GetCallInfo( )** to retrieve the DESTINATION_ADDRESS_SIP, the ORIGINATION_ADDRESS_SIP, and the IP_CALLID (if desired). The DESTINATION_ADDRESS_SIP at this time will also include the “To tag”.

For example, when the application accepts the call using **gc_AcceptCall( )**, the following information is obtained by calling **gc_GetCallInfo( )** after receipt of the GCEV_EXTENSION event:

- DESTINATION_ADDRESS_SIP (sip:ms-ivr@64.52.111.192;tag=969b6d0-0-13c4-50022-2826d-2f0e8b34-2826d)
- ORIGINATION_ADDRESS_SIP (sip:60000027@64.52.111.182;tag=a0016671)
- IP_CALLID

The same information is obtained using **gc_GetCallInfo( )** after receipt of the GCEV_EXTENSION event in the case the application answers the call immediately using **gc_AnswerCall( )** after it receives the GCEV_OFFERED event.

Examples

The following code example shows how to enable “To tag” retrieval:

```
parmblkp = NULL;

 gc_util_insert_parm_val(&parmblkp,
                            IPSET_CONFIG,
                            IPPARM_GETCALLINFOUPDATE,
                            sizeof(int),
                            GCPV_ENABLE);

if (gc_SetConfigData(   GCTGT_CCLIB_NETIF,
                         bdev,
                         parmblkp,
                         0,
                         GCUPDATE_IMMEDIATE,
                         &request_id,
                         EV_ASYNC) != GC_SUCCESS)
{
          sprintf(str, "Enabling send_totag has failed\n");
```

```
           printandlog(index, GC_APICALL, NULL, str, 0);
}
else
{
           sprintf(str, "gc_SetConfigData(linedev=%ld) Success ", bdev);
           printandlog(index, GC_APICALL, NULL, str, 0);
}
```

The following code example shows how to retrieve "To tag" information using **gc_GetCallInfo( )** once an extension event is received:

```
// process GCEV_EXTENSION event to check if extID is IPEXTID_GETCALLINFOUPDATE
// get SIP Msg and SIP Msg Info
int OnExtensionEvent(METAEVENT *metaeventp)
{
   EXTENSIONEVTBLK  *pExtensionBlock = NULL;
   unsigned char    extID;
   pExtensionBlock = (EXTENSIONEVTBLK*)(metaeventp->extevtdatap);
   extID = pExtensionBlock->ext_id;

   printf("extID : %x\n",extID);

   if(extID == IPEXTID_GETCALLINFOUPDATE)
       return 1;
   else
       return 0;
}

if(evttype == GCEV_EXTENSION)
 {
     ToTagQueryable = OnExtensionEvent(metaeventp);
     if(ToTagQueryable)
     {
         /* Call gc_getcallinfo here */

         sprintf(str, "!!!!! Received GCEV_extension in offered state");
         printandlog(index, GC_APICALL, NULL, str, 0);
         /* get to party number + to tag */
         if (gc_GetCallInfo(pline->call[callindex].crn, DESTINATION_ADDRESS_SIP, ToAddr) ==
             GC_SUCCESS) {
             sprintf(str, "!!!!!! gc_GetCallInfo(crn=0x%lx) Success in OFFERED state  = %s",
             pline->call[callindex].crn, ToAddr);
         }  else {
             sprintf(str, "!!!!!! gc_GetCallInfo(crn=0x%lx) Failure - Ram  ", pline
             ->call[callindex].crn);
         }
         printandlog(pline->index, GC_APICALL, NULL, str, 0);
}
```

4.10 Sending and Receiving MIME Bodies in SIP Messages (SIP-T)

When using SIP, the Dialogic® Global Call API library supports the sending and receiving of messages that include a single-part or multipart MIME body.

This feature was implemented primarily to allow applications to send and receive SIP Telephony (SIP-T) information, which is encoded in a MIME message body as defined in RFC 3372, a document which describes a framework for SIP-PSTN interworking gateways. This capability allows the encapsulation of ISUP in the SIP body during or after call setup, and the use of the INFO method for mid-call signaling. With the use of a separate SS7 signaling stack to translate the ISUP information, applications can route SIP messages with dependencies on ISUP to provide ISUP transparency across SS7-ISUP internetworking.

The Global Call implementation of SIP MIME messages is very general, so that it should support MIME for a variety of other purposes besides SIP-T, such as text messaging. The call control library only copies data to and from a SIP MIME body. With the exception of SDP (Session Description Protocol), the Global Call library treats MIME body information as raw data and does not parse or translate information that is encapsulated in SIP MIME messages. (SDP is not exposed to the application like other MIME-encoded data because the call control library controls media negotiations internally.)

4.10.1 SIP MIME Overview

The Dialogic® Global Call API library handles single-part MIME and multipart MIME in the same way to simplify application coding. The library uses two levels of GC_PARM_BLK data structures to contain information being embedded into or extracted from MIME messages. The top-level GC_PARM_BLK structure contains a list of one or more lower-level GC_PARM_BLK structures that contain the header and body information for each MIME part. When an application sends a single MIME part in a SIP message that already includes a MIME part for SDP (which is not exposed to applications in 1PCC mode and is not exposed using the mechanism described in this section in 3PCC mode), the library transparently creates a multipart MIME message with the appropriate multipart headers. In the case where an incoming message has multipart MIME embedded in a multipart MIME part (nested parts), the Global Call library parses through all the parts in order and extracts them to a flat list of data structures.

For incoming SIP messages with MIME information, the call control library creates a Global Call event corresponding to the message type with GC_PARM_BLK structures attached. Standard Global Call practices are used to retrieve the GC_PARM_BLK structures, and all information in each MIME part is accessed through parameters in the corresponding GC_PARM_BLK structure. It is important to note that the specific parameters that contain the MIME part header fields have been defined as parameters that may exceed the 255 byte length limit of most Global Call parameters. (The actual maximum size is configured via the max_parm_data_size field in the IPCCLIB_START_DATA structure when initializing the library.) For this reason, applications should always use the extended **gc_util_..._ex( )** functions when retrieving MIME information from incoming messages.

For outgoing SIP messages, the application must populate GC_PARM_BLK structures with parameters that specify the content of all the MIME parts to be sent, and then set the MIME information before or at the time of calling the relevant Global Call function that sends the SIP message. If any of the MIME part header fields are longer than 255 bytes (up to the maximum size configured by the application in the max_parm_data_size field in IPCCLIB_START_DATA), the application **must** use the extended **gc_util_insert_parm_ref_ex( )** function rather than the standard **gc_util_insert_parm_ref( )** utility function.

Figure 50 shows the relationships between Global Call function calls, SIP messages, and Global Call events for outgoing and incoming SIP messages with MIME content in a normal call setup/teardown scenario. Figure 51 shows the same relationships in a reject scenario.

Figure 50. SIP MIME Scenario for Normal Call Setup and Teardown

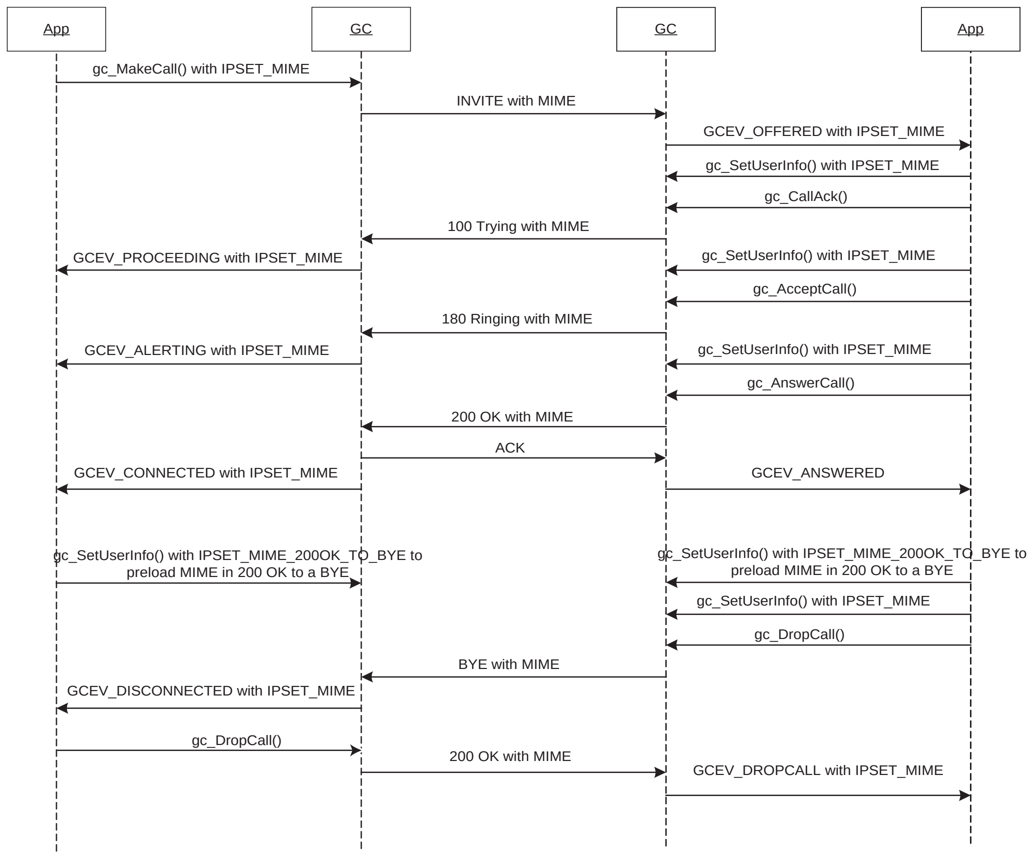
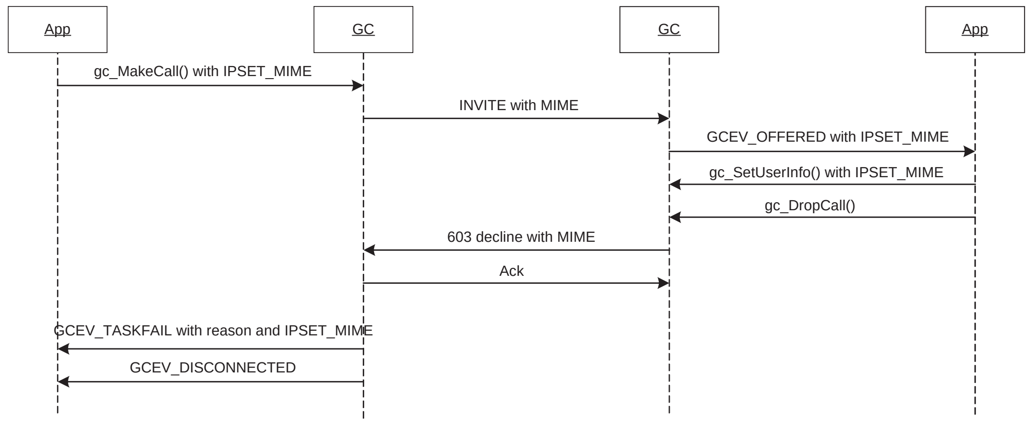
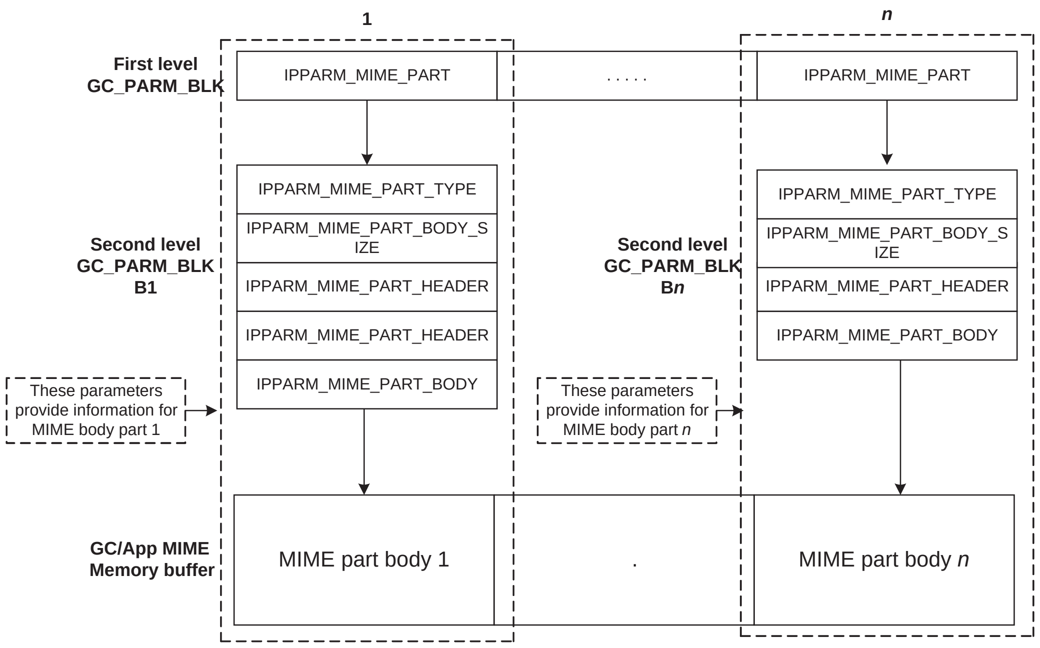

App
GC
GC
App
gc_MakeCall() with IPSET_MIME
INVITE with MIME
GCEV_OFFERED with IPSET_MIME
gc_SetUserInfo() with IPSET_MIME
gc_CallAck()
100 Trying with MIME
GCEV_PROCEEDING with IPSET_MIME
gc_SetUserInfo() with IPSET_MIME
gc_AcceptCall()
180 Ringing with MIME
GCEV_ALERTING with IPSET_MIME
gc_SetUserInfo() with IPSET_MIME
gc_AnswerCall()
200 OK with MIME
ACK
GCEV_CONNECTED with IPSET_MIME
GCEV_ANSWERED
gc_SetUserInfo() with IPSET_MIME_200OK_TO_BYE to preload MIME in 200 OK to a BYE
gc_SetUserInfo() with IPSET_MIME_200OK_TO_BYE to preload MIME in 200 OK to a BYE
gc_SetUserInfo() with IPSET_MIME
gc_DropCall()
BYE with MIME
GCEV_DISCONNECTED with IPSET_MIME
gc_DropCall()
200 OK with MIME
GCEV_DROPCALL with IPSET_MIME

Figure 51. SIP MIME Scenario for Rejected Call


App
GC
GC
App
gc_MakeCall() with IPSET_MIME
INVITE with MIME
GCEV_OFFERED with IPSET_MIME
gc_SetUserInfo() with IPSET_MIME
gc_DropCall()
603 decline with MIME
Ack
GCEV_TASKFAIL with reason and IPSET_MIME
GCEV_DISCONNECTED

Global Call uses two levels of GC_PARM_BLK data structures to handle MIME parts. The top-level GC_PARM_BLK contains the parameter set ID IPSET_MIME and one or more IPPARM_MIME_PART parameters, each of which points to a second-level GC_PARM_BLK structure that contains parameters for a specific MIME part. Within the second-level structure are three mandatory parameters that identify the type, size, and body data buffer location for the MIME part, plus an optional, possibly multiple parameter for MIME part header lines.

Figure 52. SIP MIME GC_PARM_BLK Structure



4.10.2 Enabling and Configuring the SIP MIME Feature

SIP MIME is a feature that can be disabled or enabled at the time the **gc_Start( )** function is called.

The mandatory **INIT_IP_VIRTBOARD( )** function populates the IP_VIRTBOARD structure with default values. The default value of the sip_msginfo_mask field in the IP_VIRTBOARD structure disables access to SIP message information fields (headers) and the SIP MIME feature. The default sip_msginfo_mask field value must be overridden with the value IP_SIP_MIME_ENABLE for each ipt board device on which SIP MIME capabilities are to be enabled. The following code snippet provides an example of enabling SIP header access and MIME capability for two virtual boards:

```
.
.
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE
```

```
                                 /* override default to enable SIP header and MIME access*/
ip_virtboard[1].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE;
                                 /* override default to enable AIP header and MIME access */
.
.
```

Note: Features that are enabled or configured via the IP_VIRTBOARD structure cannot be disabled or reconfigured once the library has been started. All items set in this data structure take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

When the SIP MIME feature is enabled, a dedicated MIME memory pool is allocated by the Dialogic® Global Call API library at initialization time, according to data that is contained in the MIME_MEM data structure that is in IP_VIRTBOARD. Because the size of a MIME body is potentially unlimited, the application is in the best position to set the size and number of memory buffers in the pool by overriding the default values in the MIME_MEM structure.

The buffer size should be big enough for each anticipated MIME part, including the MIME part body and all MIME part headers, but should not be larger than the maximum size permitted by the transport protocol. The default transport protocol, UDP over Ethernet, can handle up to 1500 bytes, so the MIME buffer size should be no more than 1500 if using UDP. The default buffer size value that is set by the **INIT_IP_VIRTBOARD( )** function is 1500.

The number of buffers should be large enough to handle SIP-T on all channels in both incoming and outgoing directions. To allow two buffers per direction plus one additional buffer for preloading the MIME information for the 200OK to BYE message that is sent when **gc_DropCall( )** is issued, the default number of buffers is 5 times the value of sip_max_calls.

Note that the MIME memory pool is completely separate from the application memory pool, and that it is only allocated if SIP MIME is enabled when the virtual board is initialized.

4.10.3 Getting MIME Information

In this section, we will consider the following SIP message as an example:

```
INVITE sip:user2@127.0.0.1 SIP/2.0
From: <sip:user1@127.0.0.1>;tag=0-13c4-3f9fecfb-1a356266-56c9
To: <sip:user2@127.0.0.1>
Call-ID: 93d5f4-0-13c4-3f9fecfb-1a356266-2693@127.0.0.1
CSeq: 1 INVITE
Via: SIP/2.0/UDP 146.152.84.141:5060;received=127.0.0.1;branch=z9hG4bK-3f9fecfb-
1a356270-61ce
Max-Forwards: 70
Supported: 100rel
Mime-Version: 1.0
Contact: <sip:user1@127.0.0.1>
Content-Type: multipart/mixed ;boundary=unique-boundary-1
Content-Length: 886

--unique-boundary-1
Content-Type: application/SDP ;charset=ISO-10646
```

```
v=0
o=jpeterson 2890844526 2890842807 IN IP4 126.16.64.4
s=SDP seminar
c=IN IP4 MG122.level3.com
t=2873397496 2873404696
m=audio 9092 RTP/AVP 0 3 4

--unique-boundary-1
Content-Type: application/ISUP ;version=nxv3 ;base=etsi121
Content-Disposition: signal ;handling=optional
Content-User: Dialogic ;type=demo1

01 00 49 00 00 03 02 00 07 04 10 00 33 63 21
43 00 00 03 06 0d 03 80 90 a2 07 03 10 03 63
53 00 10 0a 07 03 10 27 80 88 03 00 00 89 8b
0e 95 1e 1e 1e 06 26 05 0d f5 01 06 10 04 00

--unique-boundary-1—
Content-Type: image/jpeg
Content-Transfer-Encoding: base64

iQCVAwUBMJrRF2N9oWBghPDJAQE9UQQAtl7LuRVndBjrk4EqYBIb3h5QXIX/LC//
jJV5bNvkZIGPIcEmI5iFd9boEgvpirHtIREEqLQRkYNoBActFBZmh9GC3C041WGq
uMbrbxc+nIs1TIKlA08rVi9ig/2Yh7LFrK5Ein57U/W72vgSxLhe/zhdfolT9Brn
HOxEa44b+EI=
--unique-boundary-1—
```

Note that this example of a SIP MIME message includes three MIME parts, and that one of these MIME parts contains SDP, which is handled internally by the Dialogic® Global Call API library when the library is in the 1PCC operating mode (except for the special case of responses to OPTIONS requests). When handling this message, the application sees only two MIME parts because SDP is not exposed to applications.

Also note that this example illustrates a SIP INVITE message, which is only one of many different SIP message types that can contain MIME parts in their bodies.

Table 9. Global Call Events for Incoming SIP Messages that can Contain MIME Bodies

Incoming SIP Message	Global Call Event
BYE	GCEV_DISCONNECTED
INFO	GCEV_CALLINFO
INVITE	GCEV_OFFERED
NOTIFY	GCEV_EXTENSION
OPTIONS	GCEV_EXTENSION
REFER	GCEV_REQ_XFER
SUBSCRIBE	GCEV_EXTENSION
100 Trying	GCEV_PROCEEDING
180 Ringing	GCEV_ALERTING
200 OK to BYE	GCEV_DROPCALL
200 OK to INVITE	GCEV_CONNECTED
3xx to 6xx Request Failure	GCEV_DISCONNECTED

When receiving a Global Call event with an attached GC_PARM_BLK that contains the parameter IPPARM_MIME_PART, the application needs to retrieve the pointer to the second-level GC_PARM_BLK from the value of IPPARM_MIME_PART. In this example, there are three MIME parts in the message, but only two IPPARM_MIME_PART parameters in the GC_PARM_BLK because the SDP MIME part is not exposed. The order of the IPPARM_MIME_PART parameters is the same as the order of the MIME parts in the SIP message.

The first-level GC_PARM_BLK contains the following parameters and values for the example shown above:

`IPPARM_MIME_PART`
`0x78339ff0`
[address of first second-level GC_PARM_BLK (B1)]

`IPPARM_MIME_PART` (required)
`0x78356144`
[address of second second-level GC_PARM_BLK (B2)]

The first second-level GC_PARM_BLK (B1), at address 0x78339ff0 in this example, contains the following parameters and values, which represent the information for the first non-SDP MIME part in the example shown above:

`IPPARM_MIME_PART_TYPE`
`Content-Type: application/ISUP ;version=nxv3 ;base=etsi121`
[data from MIME part header in received MIME message]

`IPPARM_MIME_PART_BODY_SIZE`
`182`
[size of received data in buffer]

`IPPARM_MIME_PART_BODY`
`0x329823e8`
[address of buffer]

`IPPARM_MIME_BODY_HEADER` [optional parameter]
`Content-Disposition: signal ;handling=optional`
[data from MIME part header in received MIME message]

`IPPARM_MIME_BODY_HEADER` [optional parameter]
`Content-User: Dialogic ;type=demo1`
[data from MIME part header in received MIME message]

The buffer at the address given in the value of IPPARM_MIME_PART_BODY (0x329823e8 in this example) contains the data that was received in the MIME part body:

```
01 00 49 00 00 03 02 00 07 04 10 00 33 63 21
43 00 00 03 06 0d 03 80 90 a2 07 03 10 03 63
53 00 10 0a 07 03 10 27 80 88 03 00 00 89 8b
0e 95 1e 1e 1e 06 26 05 0d f5 01 06 10 04 00
```

The second, second-level GC_PARM_BLK (B2), at address 0x78356144 in this example, contains the following parameters and values, which represent the information for the second non-SDP MIME part in the example shown above:

`IPPARM_MIME_PART_TYPE`
: `Content-Type: image/jpeg`
: [data from MIME part header in received MIME message]

`IPPARM_MIME_PART_BODY_SIZE`
: `208`
: [size of received data in buffer]

`IPPARM_MIME_PART_BODY`
: `0x3298a224`
: [address of buffer]

`IPPARM_MIME_BODY_HEADER` [optional parameter]
: `Content-Transfer-Encoding: base64`
: [data from MIME part header in received MIME message]

The buffer at the address given in the value of IPPARM_MIME_PART_BODY (0x3298a224 in this example) contains the data that was received in the MIME part body:

```
    iQCVAwUBMJrRF2N9oWBghPDJAQE9UQQAtl7LuRVndBjrk4EqYBIb3h5QXIX/LC//
    jJV5bNvkZIGPIcEmI5iFd9boEgvpirHtIREEqLQRkYNoBActFBZmh9GC3C041WGq
    uMbrbxc+nIs1TIKlA08rVi9ig/2Yh7LFrK5Ein57U/W72vgSxLhe/zhdfolT9Brn
    HOxEa44b+EI=
```

Note that the data that is retrieved from each MIME part body is copied into the buffer as a continuous block of binary data whose length (in bytes) is indicated in IPPARM_MIME_PART_BODY_SIZE. No type checking or data formatting is performed by the Dialogic® Global Call API library. Note that a MIME part body does not necessarily end with '\0', and that a MIME body might contain '\0' as part of the body itself.

All GC_PARM_BLK structures (on both levels) and MIME part body buffers will be freed when the next Global Call event is accessed. The application must therefore copy the necessary parameters and the data buffers before processing the next Global Call event. When copying a complete GC_PARM_BLK structure, the application should use the **gc_util_copy_parm_blk( )** function rather than **memcopy( )** or some similar function because the parameters for MIME part headers are among the Global Call parameters that support data length greater than 255 bytes.

Code Example

The following code example illustrates the retrieval of MIME information from a GCEV_OFFERED event. It prints out every MIME part header and MIME part body (except for any SDP) that exists in the SIP INVITE message. Note that the example uses the extended utility functions because the parameters for MIME part header fields may be longer than 255 bytes.

```
INT32 processEvtHandler()
{
   METAEVENT       metaEvent;
   GC_PARM_BLK     *parmblkp = NULL;
   GC_PARM_DATAP   t_gcParmDatap = NULL;
   GC_PARM_BLK     *parmblkp2 = NULL;
   .
   .
   .
   switch (evtType)
   {

   case GCEV_OFFERED:
      /* received GC event, parse PARM_BLK, examine extension data */
      parmblkp = (GC_PARM_BLK *) metaEvent.extevtdatap;
      while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
      {
         switch(t_gcParmDatap->set_ID)
         {
         case IPSET_MIME:
            switch(t_gcParmDatap->parm_ID)
            {
            case IPPARM_MIME_PART:
               /* Get MIME part pointer */
               parmblkp2= (GC_PARM_BLK*)(*(UINT32*)( t_gcParmDatap ->value_buf));

               if(NULL == parmblkp2 || 0 != getMIMEPart(parmblkp2))
               {
                  printf("\n!!!error getting MIME part!!!\n");
                  return -1;
               }
               break;
            }
            break;
         }
      }
   }
   :
}

INT32 getMIMEPart(GC_PARM_BLK* parmblkp)
{
   GC_PARM_DATA_EXT   ParmDataExt;
   //Initialize the structure to start from the 1st parm in the GC_PARM_BLK
   INIT_GC_PARM_DATA_EXT(&ParmDataExt);

   UINT32             bodySize = 0;
   char               *appBuff = NULL;
   char               *bodyBuff = NULL;

   /* get the first param data*/
   if(GC_SUCCESS != gc_util_next_parm_ex(parmblkp, &ParmDataExt))
   {
      /* error condition */
      printf("\n !!! unable to get parm data ext !!!\n");
      return -1;
   }

   /* Get MIME type info, this has to be the first parameter */
   if(IPSET_MIME == ParmDataExt.set_ID && IPPARM_MIME_PART_TYPE == ParmDataExt.parm_ID)
   {
       printf("\t Content-Type = %s\n", (char*)ParmDataExt.pData);
   }
   else
   {
      /* error condition */
      printf("\n !!! first parameter in MIME part is not MIME type!!!\n");
      return -1;
   }
```

```
    /* Get the rest of MIME info*/
    while (GC_SUCCESS == gc_util_next_parm_ex(parmblkp, &ParmDataExt))
    {
       switch(ParmDataExt.set_ID)
       {
          case IPSET_MIME:
             switch(ParmDataExt.parm_ID)
             {
                case IPPARM_MIME_PART_TYPE:
                   /* duplicate MIME part, error out */
                   printf("\n!!!Duplicate MIME part error!!!\n");
                   return -1;
                   break;

                case IPPARM_MIME_PART_BODY_SIZE:
                   /* Get MIME part body size */
                   bodySize = *(UINT32*)(ParmDataExt.pData);
                   printf("\t MIME part body Size = %d\n", bodySize);
                   break;

                case IPPARM_MIME_PART_HEADER:
                   /* Get MIME part header */
                   printf("\t MIME part header = %s\n", (char*)ParmDataExt.pData);
                   break;

                case IPPARM_MIME_PART_BODY:
                   /* get body buffer pointer */
                   bodyBuff = (char*)(*(UINT32*)(ParmDataExt.pData));

                   /* copy MIME part body */
                   if(bodySize>0)
                   {
                      /* allocate memory */
                      appBuff = (char*)malloc(bodySize+1);
                      memcpy(appBuff, bodyBuff, bodySize);
                   }
                   else
                   {
                      /*error body size must be available*/
                      printf("\n!!! Body Size not available error !!!\n");
                      return -1;
                   }
                   /* Null terminated */
                   appBuff[bodySize] = '\0';

                   /* Only print the buffer content as string */
                   /* For binary data the buffer is not printable*/
                   printf("\t MIME part Body:\n%s\n",appBuff);

                   /* Free allocated memory*/
                   free(appBuff);
                   break;

             }
             break;

       }
    }
    .
    .
    .
    return 0;
 }
```

4.10.4 Sending MIME Information

Table 10 lists the Global Call functions that can be used to send SIP messages with MIME information using the IPSET_MIME parameter set ID in the attached GC_PARM_BLK. Except in the cases of **gc_MakeCall( )** and **gc_Extension( )**, sending a SIP message with MIME requires two function calls, **gc_SetUserInfo( )** to set the information, and a second function to cause the library to send the message.

Table 10. Global Call Functions for SIP MIME Messages Using IPSET_MIME

Global Call Function to Set MIME Parameter Block	Global Call Function to Send MIME Message	Device Type	Outgoing SIP Message with MIME
---	**gc_MakeCall( )**	LD	INVITE
---	**gc_Extension( )**	CRN or LD	INFO, OPTIONS, SUBSCRIBE, NOTIFY
gc_SetUserInfo( )	**gc_CallAck( )**	CRN	100 Trying
gc_SetUserInfo( )	**gc_AcceptCall( )**	CRN	180 Ringing
gc_SetUserInfo( )	**gc_AnswerCall( )**	CRN	200OK to INVITE
gc_SetUserInfo( )	**gc_DropCall( )**	CRN	603 Decline if before call setup BYE if after call setup

If the application only needs to send a single MIME part but the call control library also needs to send SDP information, the firmware automatically and transparently constructs the required multipart MIME message.

If the application needs to send multipart MIME, all the MIME information is set collectively within one function call on the given device by inserting multiple IPPARM_MIME_PART parameters in the desired order to construct a multipart MIME body. The MIME information set by the current function always overwrites any MIME information set by previous functions, so that an application **cannot** set multiple MIME parts by calling **gc_SetUserInfo( )** multiple times.

The application can construct a MIME message body for inclusion in the SIP 200 OK response to an incoming remote user BYE request (200 OK to BYE). You can load the MIME information on a channel using **gc_SetUserInfo( )** with set ID IPSET_MIME_200OK_TO_BYE after the GCEV_DISCONNECTED event (SIP BYE message) is received. The 200 OK to BYE message is sent after **gc_DropCall( )** is issued. Note that the parameter set ID must be set to IPSET_MIME_200OK_TO_BYE in **every** GC_PARM_BLK associated with the message, not just the top-level block. MIME information set with IPSET_MIME_200OK_TO_BYE and MIME information set with IPSET_MIME are kept independent of each other on a given channel.

The data that is to be sent in the MIME part body is copied into the message MIME part from an application buffer. The data in the buffer must match the data type that is specified by the IPPARM_MIME_PART_TYPE parameter. The Dialogic® Global Call API library treats the buffer as a continuous block of binary data of the length (in bytes) specified in IPPARM_MIME_PART_BODY_SIZE; no type checking or formatting is performed. Note that a MIME body part does not necessarily end with ‘\0’, and that a MIME body might contain ‘\0’ as part of the body itself.

Constructing and setting a MIME message is a multi-part process that can be broken down into several sub-processes:

1. Create and populate a separate GC_PARM_BLK structure for each MIME part to be sent in the SIP message.
2. Create a top-level GC_PARM_BLK structure and populate it with IPPARM_MIME_PART parameters that point to the GC_PARM_BLK structures created in the first step.
3. Set or send the message by calling the appropriate Global Call function.
4. Clean up the data structures after the function returns.

Create MIME part structures

The process of constructing an outgoing SIP MIME message begins by constructing a separate GC_PARM_BLK structure for each MIME part to be sent in the message:

1. Create a GC_PARM_BLK structure.
2. Insert the required IPPARM_MIME_PART_TYPE parameter to identify the MIME part type using the extended **gc_util_insert_parm_ref_ex( )** function because the type string may exceed 255 bytes in length.
3. Insert any MIME part headers via one or more optional IPPARM_MIME_PART_HEADER parameters, using the extended **gc_util_insert_parm_ref_ex( )** function because the headers may exceed 255 bytes in length.
4. Insert the required IPPARM_MIME_PART_BODY_SIZE parameter to identify the actual number of bytes to be copied from the application buffer to the MIME part body using the **gc_util_insert_parm_val( )** function.
5. Insert the required IPPARM_MIME_PART_BODY parameter with a pointer to the application buffer that contains the data for the MIME part body using the **gc_util_insert_parm_val( )** function. Note that the Dialogic® Global Call API library treats the buffer as a continuous block of binary data, and that the data must have the appropriate format for the MIME part type specified in the IPPARM_MIME_PART_TYPE parameter.

Create top-level GC_PARM_BLK

After repeating the preceding procedure for each MIME part to be sent in the SIP message, construct the top-level data structure that lists the MIME part structures:

1. Create a GC_PARM_BLK structure.
2. Insert a required IPPARM_MIME_PART parameter to point to the GC_PARM_BLK structure for the first MIME part in the message using the **gc_util_insert_parm_val( )** function.
3. Repeat Step 2 for each additional MIME part, inserting the parameters in order of how the MIME parts should be organized in the message.

Set/send message data and clean upsending

After creating and populating the top-level GC_PARM_BLK structure that lists all the MIME parts to be sent in the SIP message, set or send the message and clean up the set-up structures:

1. Call **gc_SetUserInfo( )** or **gc_MakeCall( )** with a pointer to the top-level GC_PARM_BLK to set or send the MIME message data.
2. Delete all GC_PARM_BLK structures created during the set-up process after the Global Call function returns.
3. Optionally, free the application buffer holding the MIME part body data, since that data has been copied into the dedicated MIME buffer when the function was called. Or you can choose to not free the application buffer and instead reuse it for the next MIME part body.

Code Example

The following code example constructs a single part MIME message and uses the **gc_MakeCall( )** function to send it in an INVITE message. Note that the example uses the extended utility function **gc_util_insert_parm_ref_ex( )** because the Content-Type and Content-Disposition header strings exceed 255 bytes.

```
#include "gclib.h"
.
.
.
GC_PARM_BLK *pParmBlockA = NULL;
GC_PARM_BLK *pParmBlockB = NULL;

char *pBodyType = "Content-Type: application/ISUP ;version=nxv3 ;base=etsi121;
userInput=123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
3456789012345678901234567890123456789012345678901234567890123456789012345678901234567890";

char *pBody = "01 00 49 00 00 03 02 00 07 04 10 00 33 63 21\r\n43 00 00 03 06 0d 03 80 90 a2 07
03 10 03 63\r\n53 00 10 0a 07 03 10 27 80 88 03 00 00 89 8b\r\n0e 95 1e 1e 1e 06 26 05 0d f5 01
06 10 04 00";

char *pPartHeader1 = "Content-Disposition: signal ;handling=optional;
userInput=123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
3456789012345678901234567890123456789012345678901234567890123456789012345678901234567890";

char *pPartHeader2 = "Content-User: Dialogic ;type=demo1";

/* Insert Content-Type field */
/* Add 1 to strlen for the NULL termination character */
gc_util_insert_parm_ref_ex(&pParmBlockB,
                           IPSET_MIME,
                           IPPARM_MIME_PART_TYPE,
                           (unsigned long)(strlen(pBodyType) + 1),
                           pBodyType);

/* Insert Body Size  */
gc_util_insert_parm_val(&pParmBlockB,
                        IPSET_MIME,
                        IPPARM_MIME_PART_BODY_SIZE,
                        sizeof(unsigned long),
                        strlen(pBody));
```

```
/* Insert MIME part Body Pointer  */
gc_util_insert_parm_val(&pParmBlockB,
                        IPSET_MIME,
                        IPPARM_MIME_PART_BODY,
                        sizeof(unsigned long),
                        (unsigned long)pBody);

/* Insert other header fields */
gc_util_insert_parm_ref_ex(&pParmBlockB,
                           IPSET_MIME,
                           IPPARM_MIME_PART_HEADER,
                           (unsigned long)(strlen(pPartHeader1) + ),
                           pPartHeader1);

/* Insert other header fields */
gc_util_insert_parm_ref_ex(&pParmBlockB,
                           IPSET_MIME,
                           IPPARM_MIME_PART_HEADER,
                           (unsigned long)(strlen(pPartHeader2) + 1),
                           pPartHeader2);

/* Insert parm block B pointer to parm block A */
gc_util_insert_parm_val(&pParmBlockA,
                        IPSET_MIME,
                        IPPARM_MIME_PART,
                        sizeof(unsigned long),
                        (unsigned long)pParmBlockB;

/* Set Call Information */
gc_SetUserInfo(GCTGT_GCLIB_CHAN, ldev, pParmBlockA, GC_SINGLECALL);

gc_util_delete_parm_blk(pParmBlockB);
gc_util_delete_parm_blk(pParmBlockA);

.
.
.

/* Make a call */
gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout, EV_ASYNC);
```

4.10.5 MIME Error Conditions

When using the SIP MIME feature, any of the following conditions causes the Global Call function to return an error with the last error set to IPERR_BAD_PARAM:

- A Global Call function attempts to set MIME information when the SIP MIME feature was not enabled by setting IP_SIP_MIME_ENABLE in the IP_VIRTBOARD structure at initialization time.
- The application attempts to set MIME information with the MIME body part size larger than the MIME memory buffer size that was configured during initialization.
- The total size of MIME parts is greater than 1500 bytes when using UDP.

If the MIME memory pool is empty, or if the configured MIME buffer size is smaller than the MIME body of an incoming SIP-T message, a GCEV_TASKFAIL event is sent to the application with the reason set to IPEC_MIME_POOL_EMPTY or IPEC_MIME_BUFF_TOO_SMALL, respectively. In addition, these error conditions also cause a response message with response code 486(Busy Here) to be sent to the remote UA. The current transaction will be terminated without causing the state of the current call to change.

4.10.6 Inserting MIME Bodies in Outgoing SIP ACK

The Dialogic® HMP Software supports a method for embedding a MIME body in an outgoing SIP ACK request message for 4xx/5xx/6xx response messages that terminate certain transactions.

Some interworking SIP configurations use embedded MIME bodies in messages to pass call state information to a non-SIP destination network. Although HMP Software supports the encapsulation of MIME bodies in many request messages out of HMP, this is not the case for ACK requests.

In particular, when an outbound call out of HMP (UAC) cannot be completed by the remote UAS, it will send a rejection message using the appropriate 4xx, 5xx, or 6xx SIP final response message.

Currently, once a 4xx/5xx/6xx response is received, the stack automatically generates the ACK request, terminates the transaction, and informs the application of the failure with a GCEV_DISCONNECTED event. Since the ACK is automatically generated without application intervention, MIME body encapsulation is not possible.

This feature allows the application to create a MIME body ahead of time so that it is then available for the Global Call API library to add it to outgoing ACK requests that are a result of certain outbound call rejections.

Once the feature is enabled at the IPT network device (channel), and the application builds the desired MIME body ahead of time, the MIME body is added in ACK messages out of HMP that are generated as a response to any 4xx/5xx/6xx SIP final response messages in most transaction scenarios.

An exception is the 487 Request Terminated message, generated by the server (UAS) as a final response to a CANCEL to a previous INVITE out of HMP. In this case, the ACK message for this 487 will not contain any MIME body irrespective of feature enablement. The **gc_DropCall( )** that generated the ACK will internally disable MIME body encapsulation to the subsequent ACK to this 487.

Note: A 487 message generated by the server (UAC) resulting from a transaction timer expiration, as per an Expires header in the outgoing INVITE out of HMP, will be handled like most 4xx/5xx/6xx final response scenarios. The ACK message sent in this case will contain a MIME body pre-built by the application as part of this feature.

Disclaimers

- The feature is intended for use with proxy or general UAS that are able to handle encapsulated bodies in ACK messages. In particular, IETF RFC 3261 SIP stipulates that the placement of bodies in ACK for non-2xx is NOT RECOMMENDED since they cannot be rejected if the body is not understood.
- If bodies are to be inserted, IETF RFC 3261 SIP RECOMMENDS that they be the same as they appeared in the original INVITE. This feature does not restrict the building of any type of valid MIME bodies, and the application is responsible for MIME content generation and applicability.

- Although this feature is exemplified in a SIP <-> ISDN User Part (ISUP) interworking, known as SIP-I, it does not imply SIP-I compliance. The feature is very specific to limited outbound scenarios.

4.10.6.1 Enabling MIME Insertion

To enable this feature, use the IPSET_MIME_ACK_TO_REJECTION set ID with the **gc_SetUserInfo( )** function, along with target_type GCTGT_GCLIB_CHAN, and duration set to GC_SINGLECALL.

The method of building the MIME body using IPSET_MIME_ACK_TO_REJECTION is the same as the method used for building the two-level GC_PARM_BLK data structures as described for IPSET_MIME or IPSET_MIME_200OKTOBYE in Section 9.2.12, “IPSET_MIME and IPSET_MIME_200OK_TO_BYE”, on page 626.

Note: To create MIME bodies requires the manipulation of MIME bodies to be enabled at the IP virtual board level, prior to calling the **gc_Start( )** function, and using the mask sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE.

4.10.6.2 Setting the MIME Body for the ACK Message

After the feature is enabled, and prior to making an outbound call, the application creates the MIME body containing the desired data to be sent along with ACK responses to final 4xx/5xx/6xx responses for those transactions.

The following parameter IDs are in the IPSET_MIME_ACK_TO_REJECTION set ID:

IPPARM_MIME_PART
: Required parameter. Used to set or get SIP message MIME part(s). Parameter value is a pointer to a GC_PARM_BLK structure that contains a list of pointers to one or more GC_PARM_BLK structures that contain MIME message parts.

IPPARM_MIME_PART_BODY
: Required parameter. Used to copy MIME part body between application and Global Call space. Parameter value is a pointer to a MIME part body.

IPPARM_MIME_PART_BODY_SIZE
: Required parameter. Used to indicate the actual size of the MIME part body, not including MIME part headers.

IPPARM_MIME_PART_HEADER
: Optional parameter. Used to contain MIME part header field in format of “field-name: field-value”. Field-name can be any string other than “Content-type”. Content is not checked by Global Call before insertion into SIP message.

IPPARM_MIME_PART_TYPE
: Required parameter. Used to contain name and value of the MIME part content type field. String must begin with the field name “Content-Type:”.

Notes: ***1.*** The application is responsible for building the appropriate MIME bodies. IP cclib does not make any attempt to parse the raw data in the MIME, and encapsulates it in the outgoing ACK as built.

2. IPSET_MIME_ACK_TO_REJECTION is supported only on a channel basis specifically for the GCTGT_GCLIB_CHAN target type. The MIME message set is not valid after the call is cleared. For the next outbound call, the application must call the **gc_SetUserInfo( )** function to reset the MIME body.

For more information about creating MIME messages, refer to Section 4.10, "Sending and Receiving MIME Bodies in SIP Messages (SIP-T)", on page 196.

4.10.6.3 Examples

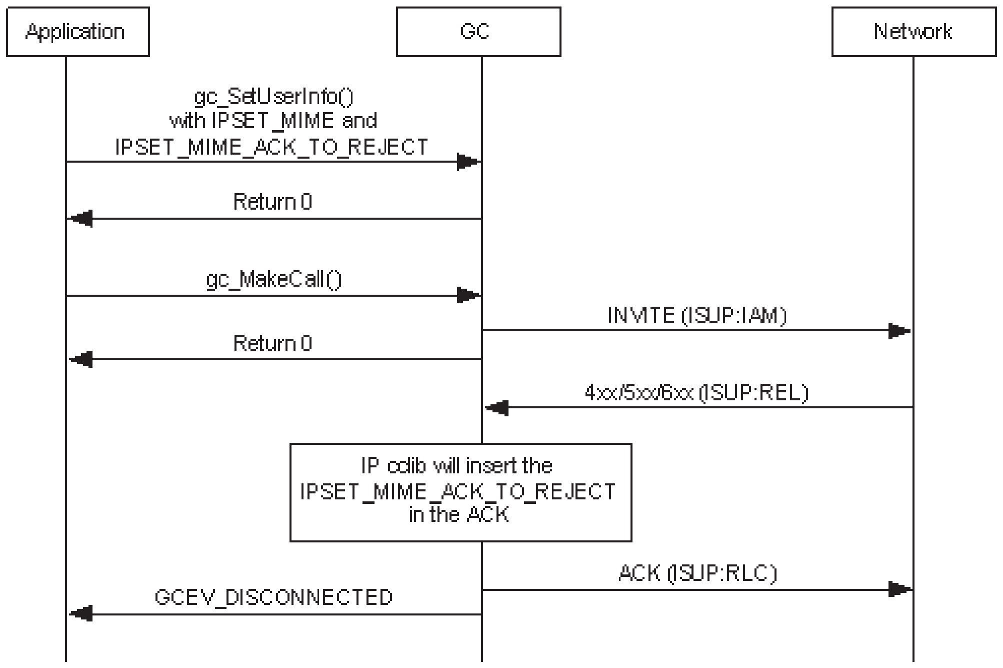
The following examples illustrate how to enable this feature and send embedded MIME bodies in outgoing ACK response messages. The examples assume that the MIME body manipulation mask was enabled using the INIT_IP_VIRTBOARD data structure.

The following diagram shows a very specific example of encapsulating MIME bodies in two different request messages out of HMP, one of them is specific to this feature. This scenario represents a possible SIP <-> ISUP interworking scenario.

Notes: ***1.*** This is a specific usage scenario. It is not intended to imply that this is the only SIP interworking usage scenario where this feature may be used.

2. The following diagram and code shows how to send a possible ISUP:IAM body as part of an outgoing INVITE out of HMP. This functionality is already available in HMP as part of the IPSET_MIME set ID.

3. An example ISUP:RLC body is sent automatically (via this feature) as part of an outgoing ACK out of HMP from this 4xx/5xx/6xx final rejection response from UAS.


Application
GC
Network
gc_SetUserInfo() with IPSET_MIME and IPSET_MIME_ACK_TO_REJECT
Return 0
gc_MakeCall()
INVITE (ISUP:IAM)
Return 0
4xx/5xx/6xx (ISUP:REL)
IP cclib will insert the IPSET_MIME_ACK_TO_REJECT in the ACK
ACK (ISUP:RLC)
GCEV_DISCONNECTED

Two different MIME bodies are to be created, one for the INVITE request and the other one for the ACK request (this feature) out of HMP. The examples show two possible ways of building the two MIME bodies:

- Single call to **gc_SetUserInfo( )**
- Two separate **gc_SetUserInfo( )** calls

Note: The MIME bodies in this example are purely for the sake of a complete example, and should not be relied on to contain valid ISUP messages.

Example 1: Single Call to gc_SetUserInfo( )

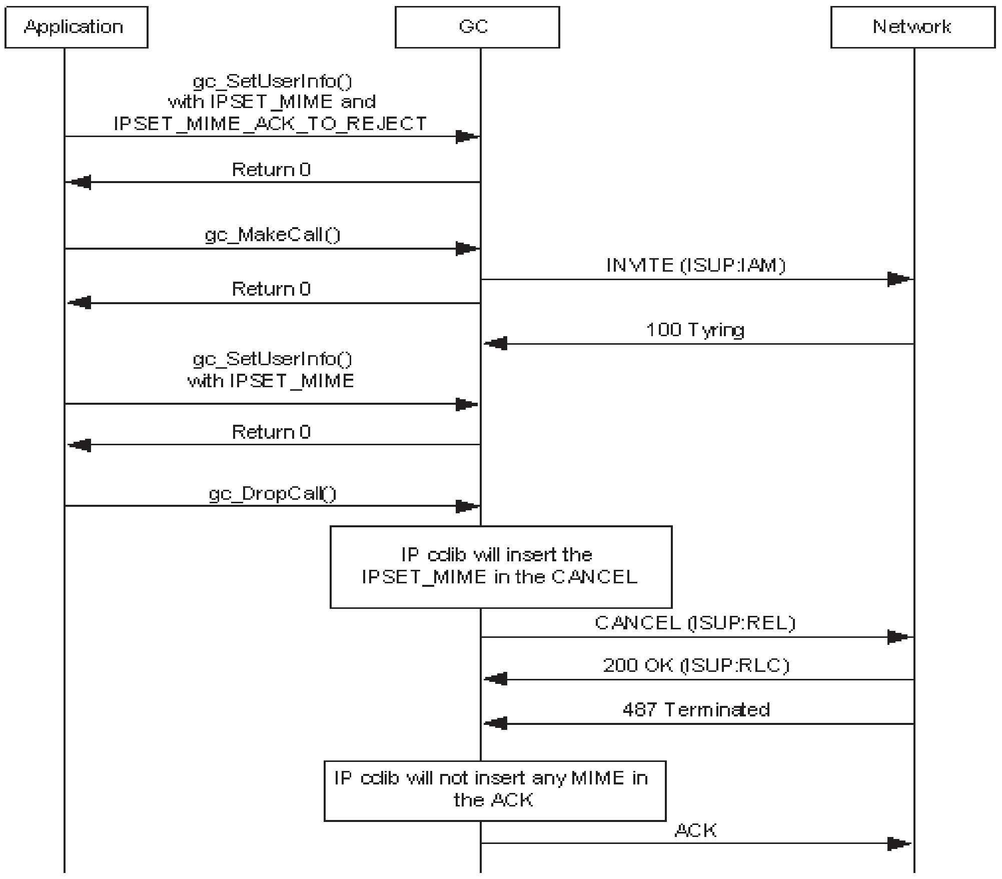
```
void  setMIMEInfo(int index)
{
    char str[MAX_STRING_SIZE];
    int frc;

    /* The following variable is used for the MAIN GC_PARM_BLK that
     * will contain the different MIME blocks */
    GC_PARM_BLK *pParmBlockMain = NULL;

    /* The following variables are used for the IAM MIME */
    GC_PARM_BLK *pParmBlockIAM_B = NULL;

    char *pBodyTypeIAM = "Content-Type: application/ISUP; version = itu-t92+";
    char *pBodyIAM = "01 00 49 00 00 03 02 00 07 04 10 00 33 63 21\r\n43 00 00 03 06 0d 03 80 90
a2 07 03 10 03 63\r\n53 00 10 0a 07 03 10 27 80 88 03 00 00 89 8b\r\n0e 95 1e 1e 1e 06 26 05 0d
f5 01 06 10 04 00";
    char *pPartHeaderIAM1 = "Content-Disposition: signal ;handling=optional";
    char *pPartHeaderIAM2 = "Content-User: Dialogic ;type=demo";

    /* The following variables are used for the RLC MIME */
    GC_PARM_BLK *pParmBlockRLC_B = NULL;

    char *pBodyTypeRLC = "Content-Type: application/ISUP; version = itu-t92+";
    char *pBodyRLC = "01 00 49 00 00 03 02 00 07 04 10 00 33 63 21\r\n43 00 00 03 06 0d 03 80 90 
a2 07 03 10 03 63";
    char *pPartHeaderRLC1 = "Content-Disposition: signal ;handling=optional";
    char *pPartHeaderRLC2 = "Content-User: Dialogic ;type=demo";


    /* Set the IAM MIME */
    /* Add 1 to strlen for the NULL termination character */
    gc_util_insert_parm_ref_ex(&pParmBlockIAM_B,
                               IPSET_MIME,
                               IPPARM_MIME_PART_TYPE,
                               (unsigned long)(strlen(pBodyTypeIAM) + 1),
                               pBodyTypeIAM);

    /* Insert Body Size */
    gc_util_insert_parm_val(&pParmBlockIAM_B,
                            IPSET_MIME,
                            IPPARM_MIME_PART_BODY_SIZE,
                            sizeof(unsigned long),
                            strlen(pBodyIAM));

    /* Insert MIME part Body Pointer */
    gc_util_insert_parm_val(&pParmBlockIAM_B,
                            IPSET_MIME,
                            IPPARM_MIME_PART_BODY,
                            sizeof(unsigned long),
                            (unsigned long)pBodyIAM);

    /* Insert other header fields */
    gc_util_insert_parm_ref_ex(&pParmBlockIAM_B,
                               IPSET_MIME,
                               IPPARM_MIME_PART_HEADER,
                               (unsigned long)(strlen(pPartHeaderIAM1) + 1),
                               pPartHeaderIAM1);

    /* Insert other header fields */
    gc_util_insert_parm_ref_ex(&pParmBlockIAM_B,
                               IPSET_MIME,
                               IPPARM_MIME_PART_HEADER,
                               (unsigned long)(strlen(pPartHeaderIAM2) + 1),
                               pPartHeaderIAM2);

    /* Insert parm block B pointer to Main parm block */
    gc_util_insert_parm_val(&pParmBlockMain,
                            IPSET_MIME,
                            IPPARM_MIME_PART,
                            sizeof(unsigned long),
                            (unsigned long)pParmBlockIAM_B);

    /* Set the RLC MIME */
    /* Add 1 to strlen for the NULL termination character */
    gc_util_insert_parm_ref_ex(&pParmBlockRLC_B,
                               IPSET_MIME_ACK_TO_REJECTION,
                               IPPARM_MIME_PART_TYPE,
                               (unsigned long)(strlen(pBodyTypeRLC) + 1),
                               pBodyTypeRLC);

    /* Insert Body Size */
    gc_util_insert_parm_val(&pParmBlockRLC_B,
                            IPSET_MIME_ACK_TO_REJECTION,
                            IPPARM_MIME_PART_BODY_SIZE,
                            sizeof(unsigned long),
                            strlen(pBodyRLC));
    /* Insert MIME part Body Pointer */
    gc_util_insert_parm_val(&pParmBlockRLC_B,
                             IPSET_MIME_ACK_TO_REJECTION,
                             IPPARM_MIME_PART_BODY,
                             sizeof(unsigned long),
                             (unsigned long)pBodyRLC);

    /* Insert other header fields */
    gc_util_insert_parm_ref_ex(&pParmBlockRLC_B,
                                 IPSET_MIME_ACK_TO_REJECTION,
                                 IPPARM_MIME_PART_HEADER,
                                 (unsigned long)(strlen(pPartHeaderRLC1) + 1),
                                 pPartHeaderRLC1);

    /* Insert other header fields */
    gc_util_insert_parm_ref_ex(&pParmBlockRLC_B,
                                 IPSET_MIME_ACK_TO_REJECTION,
                                 IPPARM_MIME_PART_HEADER,
                                 (unsigned long)(strlen(pPartHeaderRLC2) + 1),
                                 pPartHeaderRLC2);

    /* Insert parm block B pointer to MAIN parm block */
    gc_util_insert_parm_val(&pParmBlockMain,
                             IPSET_MIME_ACK_TO_REJECTION,
                             IPPARM_MIME_PART,
                             sizeof(unsigned long),
                             (unsigned long)pParmBlockRLC_B);

    frc = gc_SetUserInfo(GCTGT_GCLIB_CHAN, port[index].ldev,pParmBlockMain,GC_SINGLECALL);
    if(GC_SUCCESS != frc)
    {
        sprintf(str, "gc_SetUserInfo failed");
        printandlog(index, GC_APIERR, NULL, str, 0);
    }

    gc_util_delete_parm_blk(pParmBlockIAM_B);
    gc_util_delete_parm_blk(pParmBlockRLC_B);
    gc_util_delete_parm_blk(pParmBlockMain);

}
```

Example 2: Two Separate Calls to gc_SetUserInfo( )

```
void  setMIMEInfo(int index)
{
    char str[MAX_STRING_SIZE];
    int frc;

    /* The following variable is used for the MAIN GC_PARM_BLK that
     * will contain the different MIME blocks */
    GC_PARM_BLK *pParmBlockMain = NULL;

    /* The following variables are used for the IAM MIME */
    GC_PARM_BLK *pParmBlockIAM_B = NULL;

    char *pBodyTypeIAM = "Content-Type: application/ISUP; version = itu-t92+";
    char *pBodyIAM = "01 00 49 00 00 03 02 00 07 04 10 00 33 63 21\r\n43 00 00 03 06 0d 03 80 90 
a2 07 03 10 03 63\r\n53 00 10 0a 07 03 10 27 80 88 03 00 00 89 8b\r\n0e 95 1e 1e 1e 06 26 05 0d 
f5 01 06 10 04 00";
    char *pPartHeaderIAM1 = "Content-Disposition: signal ;handling=optional";
    char *pPartHeaderIAM2 = "Content-User: Dialogic ;type=demo";

    /* The following variables are used for the RLC MIME */
    GC_PARM_BLK *pParmBlockRLC_B = NULL;

    char *pBodyTypeRLC = "Content-Type: application/ISUP; version = itu-t92+";
    char *pBodyRLC = "01 00 49 00 00 03 02 00 07 04 10 00 33 63 21\r\n43 00 00 03 06 0d 03 80 90
a2 07 03 10 03 63";
    char *pPartHeaderRLC1 = "Content-Disposition: signal ;handling=optional";
    char *pPartHeaderRLC2 = "Content-User: Dialogic ;type=demo";

    /* Set the IAM MIME */
    /* Add 1 to strlen for the NULL termination character */
    gc_util_insert_parm_ref_ex(&pParmBlockIAM_B,
                               IPSET_MIME,
                               IPPARM_MIME_PART_TYPE,
                               (unsigned long)(strlen(pBodyTypeIAM) + 1),
                               pBodyTypeIAM);

    /* Insert Body Size */
    gc_util_insert_parm_val(&pParmBlockIAM_B,
                           IPSET_MIME,
                           IPPARM_MIME_PART_BODY_SIZE,
                           sizeof(unsigned long),
                           strlen(pBodyIAM));

    /* Insert MIME part Body Pointer */
    gc_util_insert_parm_val(&pParmBlockIAM_B,
                           IPSET_MIME,
                           IPPARM_MIME_PART_BODY,
                           sizeof(unsigned long),
                           (unsigned long)pBodyIAM);

    /* Insert other header fields */
    gc_util_insert_parm_ref_ex(&pParmBlockIAM_B,
                               IPSET_MIME,
                               IPPARM_MIME_PART_HEADER,
                               (unsigned long)(strlen(pPartHeaderIAM1) + 1),
                               pPartHeaderIAM1);

    /* Insert other header fields */
    gc_util_insert_parm_ref_ex(&pParmBlockIAM_B,
                               IPSET_MIME,
                               IPPARM_MIME_PART_HEADER,
                               (unsigned long)(strlen(pPartHeaderIAM2) + 1),
                               pPartHeaderIAM2);

    /* Insert parm block B pointer to Main parm block */
    gc_util_insert_parm_val(&pParmBlockMain,
                           IPSET_MIME,
                           IPPARM_MIME_PART,
                           sizeof(unsigned long),
                           (unsigned long)pParmBlockIAM_B);

    frc = gc_SetUserInfo(GCTGT_GCLIB_CHAN, port[index].ldev,pParmBlockMain,GC_SINGLECALL);
    if(GC_SUCCESS != frc)
    {
        sprintf(str, "gc_SetUserInfo failed for IAM MIME");
        printandlog(index, GC_APIERR, NULL, str, 0);
    }

    gc_util_delete_parm_blk(pParmBlockIAM_B);
    gc_util_delete_parm_blk(pParmBlockMain);

    pParmBlockMain = NULL;

    /* Set the RLC MIME */
    /* Add 1 to strlen for the NULL termination character */
    gc_util_insert_parm_ref_ex(&pParmBlockRLC_B,
                               IPSET_MIME_ACK_TO_REJECTION,
                               IPPARM_MIME_PART_TYPE,
                               (unsigned long)(strlen(pBodyTypeRLC) + 1),
                               pBodyTypeRLC);
    /* Insert Body Size */
    gc_util_insert_parm_val(&pParmBlockRLC_B,
                            IPSET_MIME_ACK_TO_REJECTION,
                            IPPARM_MIME_PART_BODY_SIZE,
                            sizeof(unsigned long),
                            strlen(pBodyRLC));

    /* Insert MIME part Body Pointer */
    gc_util_insert_parm_val(&pParmBlockRLC_B,
                            IPSET_MIME_ACK_TO_REJECTION,
                            IPPARM_MIME_PART_BODY,
                            sizeof(unsigned long),
                            (unsigned long)pBodyRLC);

    /* Insert other header fields */
    gc_util_insert_parm_ref_ex(&pParmBlockRLC_B,
                                IPSET_MIME_ACK_TO_REJECTION,
                                IPPARM_MIME_PART_HEADER,
                                (unsigned long)(strlen(pPartHeaderRLC1) + 1),
                                pPartHeaderRLC1);

    /* Insert other header fields */
    gc_util_insert_parm_ref_ex(&pParmBlockRLC_B,
                                IPSET_MIME_ACK_TO_REJECTION,
                                IPPARM_MIME_PART_HEADER,
                                (unsigned long)(strlen(pPartHeaderRLC2) + 1),
                                pPartHeaderRLC2);

    /* Insert parm block B pointer to MAIN parm block */
    gc_util_insert_parm_val(&pParmBlockMain,
                            IPSET_MIME_ACK_TO_REJECTION,
                            IPPARM_MIME_PART,
                            sizeof(unsigned long),
                            (unsigned long)pParmBlockRLC_B);
        frc = gc_SetUserInfo(GCTGT_GCLIB_CHAN, port[index].ldev,pParmBlockMain,GC_SINGLECALL);
    if(GC_SUCCESS != frc)

    {
        sprintf(str, "gc_SetUserInfo failed for ACK MIME");
        printandlog(index, GC_APIERR, NULL, str, 0);
    }

    gc_util_delete_parm_blk(pParmBlockRLC_B);
    gc_util_delete_parm_blk(pParmBlockMain);
}
```

4.10.6.4 Exception INVITE/CANCEL/487 Usage Scenario

As described in Section 4.10.6, "Inserting MIME Bodies in Outgoing SIP ACK", on page 210, there is a particular usage scenario where the ACK will not contain an embedded MIME body even if the feature was enabled, and the IPSET_MIME_ACK_TO_REJECTION was set ahead of time. Since the call cancellation was already processed by the network (ISUP:RLC in 200OK example below), there is no further need to embed a MIME body in the ACK that completes the transaction. The diagram below shows this specific scenario for a possible SIP <-> ISUP interworking configuration.


Application
GC
Network
gc_SetUserInfo() with IPSET_MIME and IPSET_MIME_ACK_TO_REJECT
Return 0
gc_MakeCall()
INVITE (ISUP:IAM)
Return 0
100 Trying
gc_SetUserInfo() with IPSET_MIME
Return 0
gc_DropCall()
IP cclib will insert the IPSET_MIME in the CANCEL
CANCEL (ISUP:REL)
200 OK (ISUP:RLC)
487 Terminated
IP cclib will not insert any MIME in the ACK
ACK

4.11 MIME-Based Overlap Receive Support for Limited SIP-I Interworking Scenarios

The Dialogic® HMP Software supports SIP-I interworking capability by providing a method for handling overlap receive SIP calls, where called-party addressing is supplied in multiple INVITEs but needs to be propagated to the application as standard en bloc signaling calls.

This specific SIP-I interworking scenario is for an overlap signaling call originated at a remote ISUP exchange containing partial DNIS addressing in the called-party number field of a IAM and SAM(s). The IAM and SAM(s) are propagated to HMP as embedded MIME bodies in multiple SIP INVITE requests as part of a SIP transaction. In this scenario, each INVITE within the transaction contains an ISUP IAM and zero or more SAM messages embedded in a MIME body. IAM/SAM messages would carry partial called-party addressing until a final INVITE contains the complete called-party addressing within the ISUP IAM and optional SAM messages.

With this feature, HMP in 1PCC mode handles a SIP-I overlap receive scenario by parsing the ISUP IAM and optional SAM MIME bodies in each INVITE and determining if the called-party address is incomplete. If it is incomplete, HMP automatically rejects it with a 484 Address

Incomplete response. The processing continues until the called-party address is complete, at which point the application is presented with the incoming call as if it were a standard en bloc signaling INVITE.

4.11.1 Feature Description

This feature is concerned with ITU-T Q.763 IAM and SAM messages, in particular with the called-party number field in them, containing the DNIS. HMP, acting as a SIP UAS, can support specific SIP-I overlap receive scenarios where an interworking unit (gateway) at the far end embeds the IAM/SAM message in a multipart MIME body as part of the INVITE requests, in addition to the SDP MIME part. Each INVITE carries embedded ISUP IAM and optional SAM MIME bodies with partial called-party addressing (DNIS). In these scenarios, the first INVITE normally carries an ISUP IAM from the originating exchange, and subsequent INVITEs carry ISUP SAMs belonging to the same call transaction. The ISUP IAM contains an initial called-party addressing and a number of ISUP SAMs will contain additional called-party addressing until the complete called-party address signaling is provided.

Multiple INVITEs containing ISUP IAM and zero or more SAM messages, each one containing the original addressing plus additional addressing, are received by HMP until the called-party addressing is complete, at which point an end of signaling (stop digit) appears at the end of the called-party number field, indicating a complete address. This digit uniquely identifies the last INVITE in the sequence.

When this feature is enabled, HMP parses the MIME body in each incoming INVITE request to determine if ISUP IAM or SAM messages in ITU-T Q.763 format are present. If so, the called-party number field is parsed looking for an end of signaling (stop digit “F”) presence. If the IAM or SAM message does not contain the stop digit, then the INVITE is automatically rejected by HMP with a 484 Address Incomplete response with no application interaction. No further processing by HMP is done on this INVITE, nor is the application made aware of the automatic 484 Address Incomplete response. If the stop digit (“F”) is present, then the INVITE is presented to the application as a standard GCEV_OFFERED event with the complete en bloc DNIS address, which will be contained in the INVITE header fields.

4.11.2 Enabling MIME-based Overlap Receive

To enable MIME-based overlap receive on a board basis, use **gc_SetConfigData( )** with IPSET_CONFIG set ID and the following parameter ID:

IPPARM_MIME_OVERLAP_RECEIVE
: Enables the overlap receive feature of SIP-I based on embedded MIME ISUP messages. Possible values are:
  - GCPV_ENABLE – ISUP IAM and optional SAM messages embedded in a MIME body in the incoming INVITE message are parsed to check if the “F” stop digit is present. If not present, the call will be rejected with a 484 Address Incomplete response.
  - GCPV_DISABLE – ISUP MIME bodies will not be parsed. This is the default behavior.

Once the feature is enabled at the board level, all IPT network devices (channels) are able to parse MIME bodies carrying ISUP IAM and SAM messages in ITU-T Q.763 format containing overlap called-party number fields and automatically reject those INVITEs within the transaction that do not contain complete addressing. Once the complete address is received, HMP presents the incoming call and its complete called-party addressing as a standard GCEV_OFFERED event to a channel. The channel can handle the incoming event as it would normally do with any other incoming call. The called-party addressing signaling is expected to be available in the relevant header fields within the INVITE which will be made available to the channel as a standard DNIS number.

Notes: ***1.*** HMP requires that ISUP messages embedded as MIME bodies in the incoming INVITEs are of IAM or SAM type and adhere to the ITU-T Q.763 specification in order for it to parse overlap receive signaling.

2. ISUP SAM messages are not always required to carry additional called-party signaling, and a single IAM message may be sufficient to carry complete en bloc called-party addressing.

3. It is possible for one INVITE to carry an ISUP IAM and one or more SAMs, each containing partial called-party number fields.

Example

The following example shows how to enable this feature:

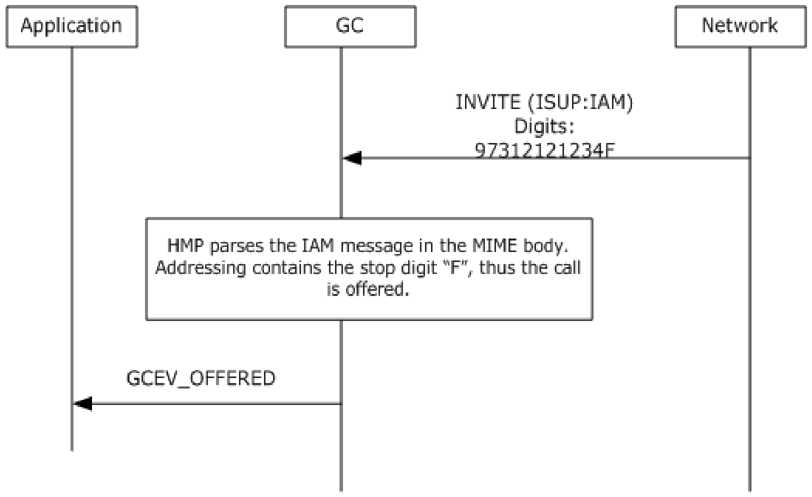
```
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>
#include <gcip_defs.h>

/*
* Assume the following has been done:
* 1. Open board device iptB1 and save handle as boardDev which will be
* used in gc_SetConfigData()
*/

int enable_overlap()
{
        GC_PARM_BLKP parmblkp = NULL;
        long request_id = 0;
        gc_util_insert_parm_val(&parmblkp,
               IPSET_CONFIG,
               IPPARM_MIME_OVERLAP_RECEIVE,
               sizeof(int),
               GCPV_ENABLE);

         if(gc_SetConfigData(GCTGT_CCLIB_NETIF, boardDev, parmblkp, 0, /*timeout*/,
             GCUPDATE_IMMEDIATE, &request_id, EV_ASYNC) != GC_SUCCESS)
         {
              // handle error…
         }

         return (0);
}
```

4.11.3 Usage Scenarios

The MIME-based overlap receive feature involves one or more INVITE messages sent by the remote UAC. Each INVITE may contain initial and additional called-party addressing in embedded ISUP IAM and optional SAM MIME bodies within the request. The ISUP IAM and optional SAM bodies are parsed seeking for the called-party number field. Each INVITE is treated independently of any previous INVITE in the transaction until the last INVITE with complete addressing is received. The following figures demonstrate typical usage scenarios for this feature.

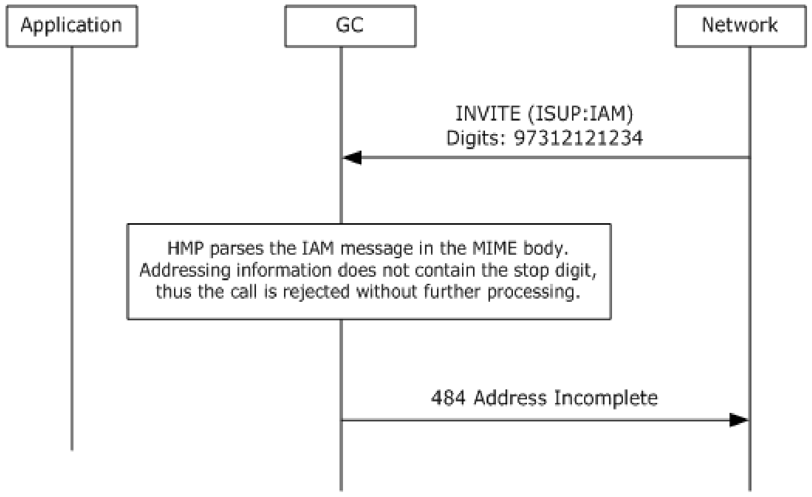
INVITE with Complete Address in IAM

The incoming INVITE contains an IAM message with the complete address (contains the stop digit "F").


Application
GC
Network
INVITE (ISUP:IAM)
Digits:
97312121234F
HMP parses the IAM message in the MIME body. Addressing contains the stop digit "F", thus the call is offered.
GCEV_OFFERED

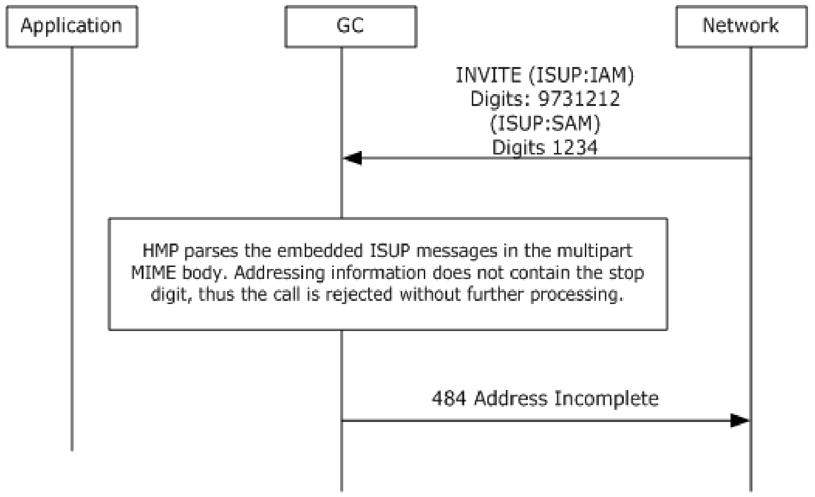
INVITE with Incomplete Address in IAM

The incoming INVITE contains an IAM message with incomplete address (does not contain the stop digit).


Application
GC
Network
INVITE (ISUP:IAM)
Digits: 97312121234
HMP parses the IAM message in the MIME body. Addressing information does not contain the stop digit, thus the call is rejected without further processing.
484 Address Incomplete

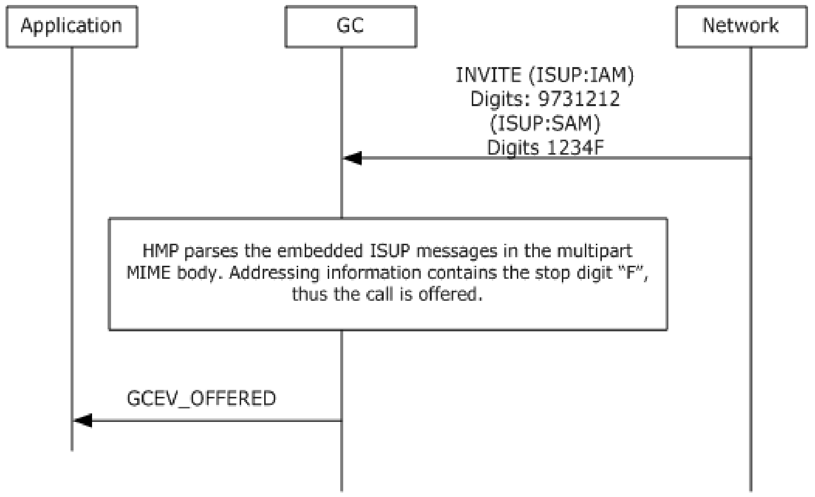
INVITE with Incomplete Address in IAM and SAM

The incoming INVITE contains an IAM and a SAM, and neither message contains the complete address (neither one contains the stop digit).


Application
GC
Network
INVITE (ISUP:IAM)
Digits: 9731212
(ISUP:SAM)
Digits 1234
HMP parses the embedded ISUP messages in the multipart MIME body. Addressing information does not contain the stop digit, thus the call is rejected without further processing.
484 Address Incomplete

INVITE with Complete Address in IAM and SAM

The incoming INVITE contains both IAM and SAM, and the messages contain the complete address (contains the stop digit "F").


Application
GC
Network
INVITE (ISUP:IAM)
Digits: 9731212
(ISUP:SAM)
Digits 1234F
HMP parses the embedded ISUP messages in the multipart MIME body. Addressing information contains the stop digit "F", thus the call is offered.
GCEV_OFFERED

For additional information about MIME bodies, refer to Section 4.10, "Sending and Receiving MIME Bodies in SIP Messages (SIP-T)", on page 196.

4.12 Overlap Send and Receive Support

Dialogic® HMP Software provides support for SIP overlap send and receive signaling per RFC 3578. Given the extent of this RFC, this implementation is focused on interoperability with the Dialogic® Integrated Media Gateway (IMG) 1010 product. For the details on the gateway, see: *http://www.dialogic.com/webhelp/IMG1010/10.5.3/WebHelp/IMG.htm*.

Overlap signaling is most prominent in networks with delays, and is used so that network equipment in the path to the end party can make early decisions about the call routing to the end party. Overlap signaling is not native to SIP; however, the intent of the RFC3 578 is to define the interworking behavior so that SIP can cope with this type of signaling from PSTN networks.

The following sections describe the specific differences between the overlap signaling model as defined in Scenario 3, in the "Basic Call Control in Asynchronous Mode" section, Call State Models chapter of the *Dialogic® Global Call API Programming Guide* and the actual implementation and behavior of this feature.

4.12.1 Overlap Receiving (Ingress)

Overlap receiving is implemented using the GCEV_SENDMOREINFO and GCEV_REQMOREINFO events and the **gc_ReqMoreInfo( )** function.

Note: The text in italics is extracted from the Global Call call state model. If any differences exist with the implementation of this feature, it is indicated below the italic text.

When an incoming call is detected, the technology call control layer immediately sends an acknowledgement. If the minimum amount of information required is specified, then the call is offered to the application only after the minimum amount of information required is received. Otherwise the call is immediately offered to the application.

Even though the Global Call model expects that the application is configured to send a call proceeding indication after sufficient information has been received, with SIP there is no minimum amount of information required so the call is always immediately offered.

```
HMP                         SIP far end
|                           |<--------INVITE CSeq=1----|  (incomplete)
|<------GCEV_OFFERED-----|
```

When the call is in the Offered state (after generation of the unsolicited GCEV_OFFERED event), the application may selectively retrieve call information, such as the Destination and Origination address (caller ID) by issuing the **gc_GetCallInfo( )** *function. If more information is required, the application may also request more address information using the* **gc_CallAck(GCACK_SERVICE_INFO)** *function.*

With this feature, this is handled as follows:

```
|---gc_GetCallInfo()------>|
|--gc_CallAck(MORE_INFO)-->|---------100 CSeq=1------>|
```

No acknowledgement is sent to the remote side at this time since an acknowledgement was already sent out. When the additional information is received, the GCEV_MOREINFO event is generated.

With this feature, this is handled as follows:

```
|                          |<--------INVITE CSeq=2----| (incomplete)
|<-------GCEV_MOREINFO-----|                          |
|----gc_GetCallInfo()----->|                          |
|-----gc_ReqMoreInfo()---->|---------100 CSeq=2------>|
|                          |---------484 CSeq=1------>|
|                          |<--------ACK CSeq=1-------|
```

The **gc_ReqMoreInfo( )** *function is issued to request more information if required. When the additional information is received, the GCEV_MOREINFO event is again generated.*

With this feature, this is handled as follows:

```
|                          |<--------INVITE CSeq=3----| (incomplete)
|<-------GCEV_MOREINFO-----|                          |
|---gc_GetCallInfo()------>|                          |
|-------gc_ReqMoreInfo()-->|---------100 CSeq=3------>|
|                          |--------484 CSeq=2------->|
```

When all the required information is received, the application may send a call proceeding indication to the remote side by issuing the **gc_CallAck(GCACK_SERVICE_PROC)** *function. Otherwise, the application can choose to accept or answer the call.*

Even though the Global Call model expects that the application issues the **gc_CallAck(GCACK_SERVICE_PROC)** function to "send a call proceeding indication to the other side", this is taken care of by the **gc_AcceptCall( )** function by sending out a 183 Progress provisional response to the far end point:

```
|                            |<--------INVITE CSeq=4------| (complete)
|<-------GCEV_MOREINFO-------|                            |
|                            |                            |
|-  --gc_GetCallInfo()------>|                            |
|-----gc_AcceptCall()------->|---------183 CSeq=4-------->|
|                            |---------200 CSeq=4-------->|
|                            |<--------ACK CSeq=4---------|
|<-------GCEV_ACCEPT---------|                            |
```

4.12.2 Overlap Sending (Egress)

Overlap sending is implemented using the GCEV_MOREINFO event and the **gc_SendMoreInfo( )** function.

Overlap sending closely follows the Global Call call state model described in the *Dialogic® Global Call API Programming Guide*; however, Dialogic® HMP Software does not provide any indication of address completion (in R-URI), such as a termination digit.

4.12.3 Enabling Overlap Signaling

Both overlap send and receive are disabled by default, and each can be independently enabled on a board basis using the **gc_SetConfigData( )** function. The IPSET_CONFIG set ID is used in conjunction with the following parameter ID:

IPPARM_OVERLAP_SIGNALING
    Enables the overlap feature. Possible values are:

- IP_SEND_RECEIVE_TYPE_RFC_3578 – Both send and receive modes are enabled.
- IP_RECEIVE_TYPE_RFC_3578 – Only receive mode is enabled.
- IP_DISABLE – Both send and receive modes are disabled (default).

4.12.4 Guidelines

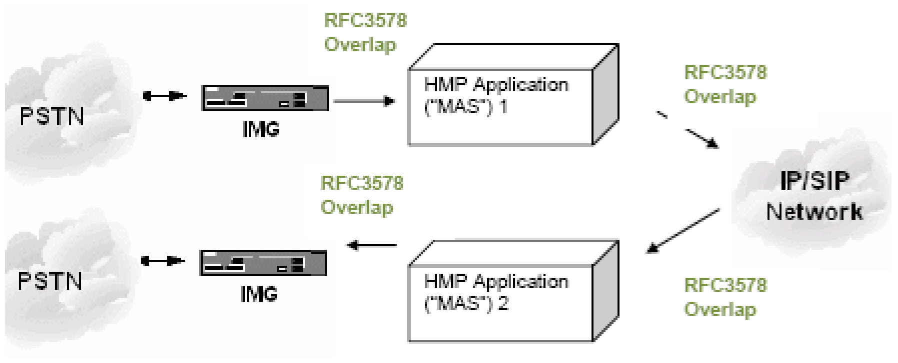
Follow these guidelines when using overlap send and receive:

- The address_info element of the GCLIB_CALL_BLK structure must be set to GCADDRINFO_OVERLAP before calling the **gc_MakeCall( )** function.
- The **gc_MakeCall( )** function does not go in PROCEEDING state but stays in the DIALING state when GCLIB_CALL_BLK structure is set to GCADDRINFO_OVERLAP.
- The user can call **gc_SendMoreInfo( )** with **info_ptr**=NULL to indicate that all dialing digits are complete. When this is done, the application should receive the GCEV_PROCEEDING event.

Refer to the Call State Models chapter of the *Dialogic® Global Call API Programming Guide* for more information about overlap sending and receiving.

4.12.5 Overlap Signaling Scenarios

The diagram below shows a configuration to transit overlap signaling from one SS7 network, through the IMG, routed through Dialogic® HMP Software to an overlap-compatible third-party SIP endpoint. This endpoint may be a similar set-up in reverse.



This requires the ability of Dialogic® HMP Software to interface with a media gateway controller that converts ISUP Overlap Receive signaling into outgoing SIP messages. The application, acting as a back-to-back user agent, must be able to receive multiple (nested) incoming INVITEs on the same call leg, and to relay these multiple INVITEs to a second call leg.

The term “nested INVITEs” refers to each request sent to Dialogic® HMP Software belonging to the same call leg, as in the same Call-ID and the same From header field including the tag as the first INVITE sent. The CSeq number (incremental) will be different for each INVITE.

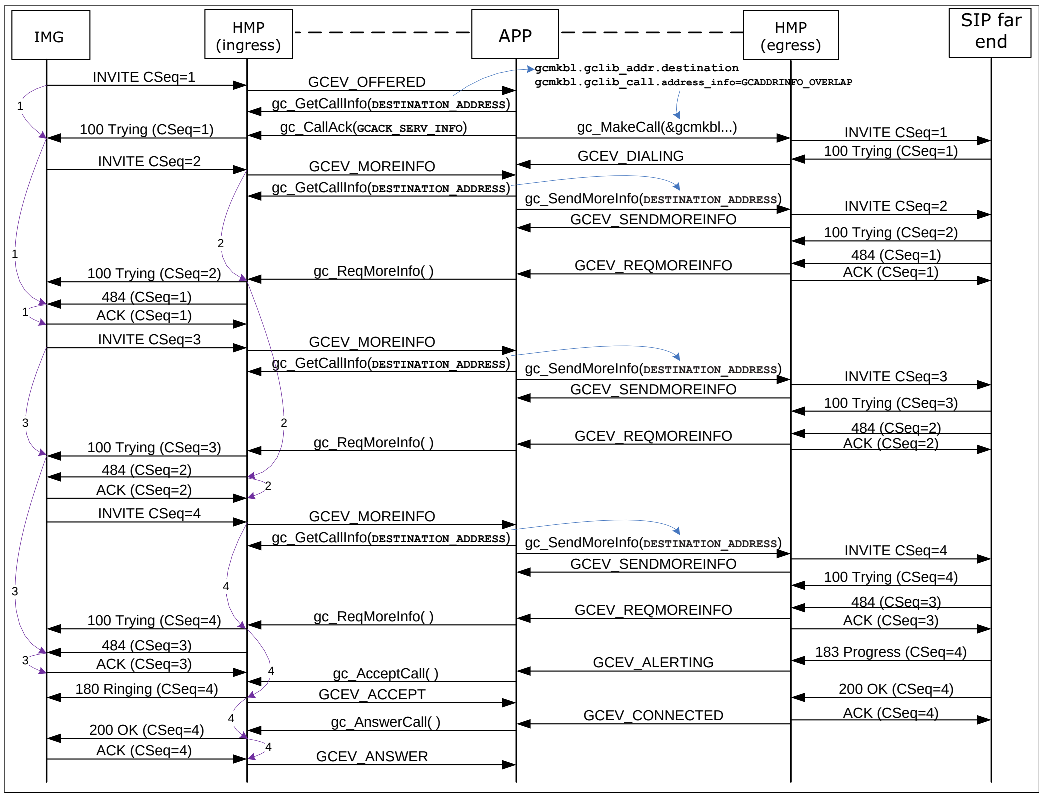
The assumption is that in the inbound side the HMP application will not know in advance what the addressing number completeness criteria is; however, when working as a back-to-back user agent, the outbound end point knows these criteria and hence is able to provide final response to HMP when the appropriate number of addressing information is available.

The following scenarios are based on the above diagram.

4.12.5.1 Overlap Send and Receive Scenario 1

The scenario shown in the diagram below shows a typical successful overlap send and receive implementation.

Note: The numbered curved lines show the CSeq correlation of each SIP message in the dialog.



Sequence for Scenario 1

The application configures HMP for Global Call IP Overlap Send and Receive using the **gc_SetConfigdata( )** function.

Inbound side actions:

- Receive a GCEV_OFFERED event.

- Call the **gc_CallAck( )** function to send a 100 Trying back to the IMG as is usually done on a. normal call.
- Retrieve the DESTINATION_ADDRESS.

Note: The IMG does not support getting a 484 response to the first (nested) INVITE prior to the second INVITE on same call leg, despite the fact RFC 3578 has a provision for this case. When in Overlap Receive mode, HMP does not automatically respond to a first INVITE until the stack receives a subsequent INVITE, at which point it can trigger the sending of a 484 (Address Incomplete) response to the immediately previous nested INVITE. HMP is designed with this delayed response behavior in the sequence of (nested) INVITEs so that responses are always behind one.

HMP relays the incoming call to the outbound side.

- The **gc_MakeCall( )** function passes the addressing information received from the inbound side in Overlap mode:
  - GCLIB_ADDRESS_BLK.destination is retrieved from the inbound side.
  - GCLIB_CALL_BLK.address_info is set to GCADDRINFO_OVERLAP.
- The call moves to the DIALING state; and a GCEV_DIALING event may be received.

Inbound side actions:

- The far end (ISUP) equipment sends a SAM with additional addressing; the gateway translates it as a (nested) INVITE(2) with the same Call-ID and the same From header field including the tag as the first INVITE sent (but has an updated Request-URI), and increased CSeq value. This triggers Overlap Receive mode in the Call Control library stack.
- The GCEV_MOREINFO event is received as part of the same call object.
- The application retrieves the additional addressing using the **gc_GetCallInfo( )** function.

Outbound side actions:

- Additional addressing digits are now sent to the far end IMG by invoking the **gc_SendMoreInfo( )** function.
- This triggers the Call Control library to enter in Overlap Send mode for the channel.
- The IP stack is instructed to send a new INVITE(2) with the same Call-ID and the same From header field including the tag as the first INVITE sent (but has an updated Request-URI), and increased CSeq value.
- The IMG may respond with a 1xx message.
- The far end (ISUP) equipment determines address-completion criteria. In this particular scenario, the far end knows there are not enough digits and terminates the previous (1) request with 484 Address Incomplete.
- This triggers HMP to send a GCEV_REQMOREINFO event to the application.

Inbound side actions:

- The application uses the GCEV_REQMOREINFO event to invoke the **gc_ReqMoreInfo( )** function. This function terminates the previous (1) request from IMG also with a 484 message.
- The far end (ISUP) equipment sends a SAM with additional addressing, which is translated by the IMG as a new INVITE(3).
- The GCEV_MOREINFO event is received as part of the same call object.

The above sequence continues for inbound side and outbound side with the expectation that the far end ISUP equipment knows address-completion criteria for the call. Once the outbound side far end determines that one of the nested INVITEs is complete, it acts as follows:

Outbound side action:

- Typically, a 183 Progress is received and triggers a GCEV_ALERTING event to the application.

  Note: A GCEV_PROCEEDING event is not received, even upon receipt of a 100 Trying message.

Inbound side action:

- The application accepts the previous request as a new INVITE(4) using the **gc_AcceptCall( )** function.

Outbound side action:

- A GCEV_CONNECTED event is received.

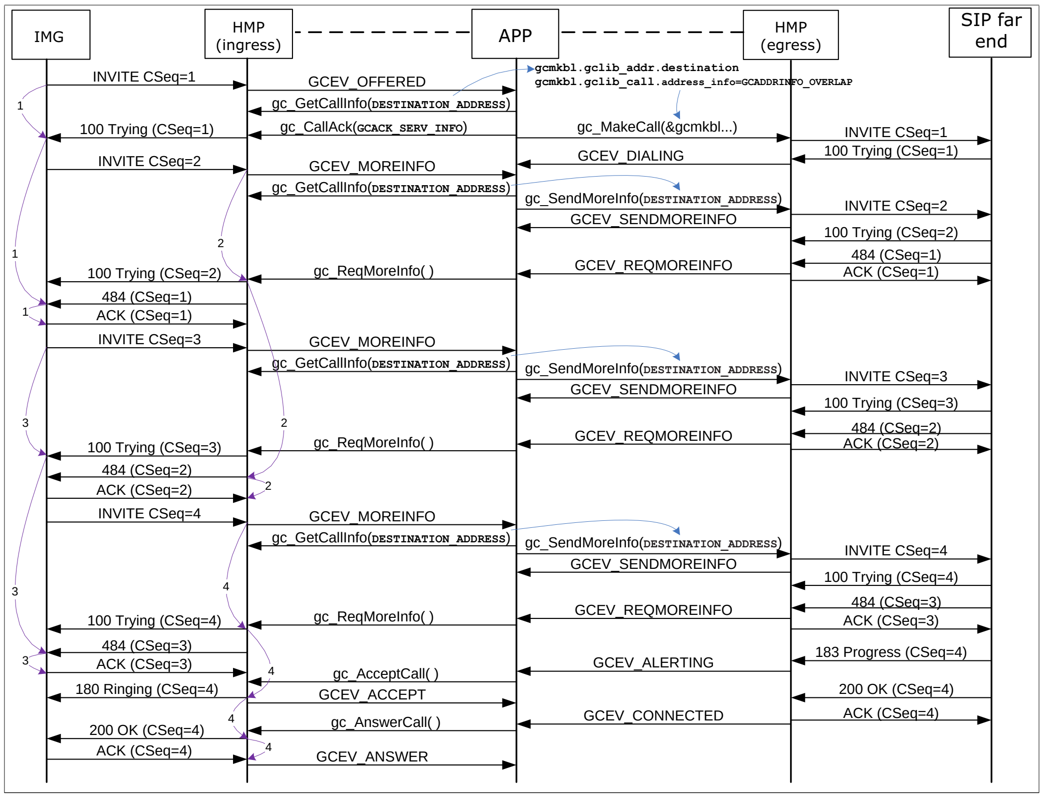
Inbound side action:

- The application answers the call (dialog) using the **gc_AnswerCall( )** function.

4.12.5.2 Overlap Send and Receive Scenario 2

The scenario shown in the diagram below shows a typical successful overlap send and receive with GCEV_PROCEEDING implementation. The sequence that follows explains the process.

Note: The numbered curved lines show the CSeq correlation of each SIP message in the dialog.



Sequence for Scenario 2

Refer to the sequence in Scenario 1 up to the receipt of the last nested INVITE(4). The following actions take place:

Inbound side action:

- The application has knowledge of overlap receive address completion.

  Note: The application may choose to call the **gc_ReqMoreInfo( )** function again in order to terminate the previous nested INVITE request (CSeq=3). The application immediately calls the **gc_AcceptCall( )** function to proceed with the acceptance of the final INVITE request received (CSeq=4).

Outbound side action:

- Upon receipt of the GCEV_SENDMOREINFO event, a **gc_SendMoreInfo( )** function is called with its **info_ptr** parameter set to NULL.

- The "100 Trying" (4) provisional from last nested INVITE(4) out of HMP triggers a GCEV_PROCEEDING event to the application.
- The 484 message from the previous nested INVITE(3) out of HMP is ignored.

  Note: The GCEV_REQMOREINFO event is not provided to the application.

4.13 Specifying Transport for SIP Messages

When a virtual board is configured with default values in the IP_VIRTBOARD data structure, the supported transport protocol for all SIP messages is UDP. Applications do not have the ability to send messages using TCP, and incoming TCP messages are refused.

By setting non-default parameter values in the IP_VIRTBOARD before calling **gc_Start( )**, applications can enable support of TCP as well as UDP. In addition to enabling overall TCP support, the application can configure the board to use TCP as the default transport protocol, and can set the persistence of TCP connections. See Section 4.1.2, "Configuring SIP Transport Protocol", on page 102, for details about the configuration process.

When TCP is enabled, incoming TCP messages are accepted, and if the application needs to determine the transport protocol it can access the Request-URI in the Global Call event as described in Section 4.9.6, "Retrieving SIP Message Header Fields", on page 188. When responding to a SIP request, the application does not need to specify TCP because the transport parameter is already present in the Request-URI.

SIP requests that are sent by the application outside of a SIP dialog (for example, INVITE, SUBSCRIBE, or NOTIFY) normally use the default transport protocol, but the application can override the default to send a specific request using the non-default protocol by setting a "transport=" parameter in the Request-URI header field before the message is sent. If the default transport is UDP, the relevant parameter string to override the default is ";transport=tcp"; if the default transport is TCP, the relevant parameter string to override the default is ";transport=udp". Setting the transport for a specific SIP request requires that the SIP message information access feature be enabled and uses the process described in Section 4.9.5, "Setting SIP Header Fields for Outbound Messages", on page 185. The following code lines illustrate how a Request-URI with transport parameter would be inserted into the parameter block for the message to be sent.

```
sprintf(strReqURI, "sip:%s:%d;transport=tcp", strIPaddr, intPort);
gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_REQUEST_URI,
                        strlen(strReqURI)+1,
                        strReqURI);
```

For SIP requests within a dialog (for example, INFO, NOTIFY, or REFER), there is no need to set the transport protocol if the persistence configuration item in IP_VIRTBOARD is set to ENUM_PERSISTENCE_TRANSACT_USER (the default value), because the existing TCP connection will be used.

BYE requests are exceptions to the general TCP behavior in several respects. First, BYE requests always make a new connection; an existing TCP connection is not used even if TCP is configured for user persistence. Second, a default transport protocol setting of TCP or a ";transport=tcp" parameter in the Request-URI header field is not sufficient to force TCP for a BYE request. Instead, it is necessary to also set ";transport=tcp" in the Contact URI header field.

Due to network conditions, in certain instances a 1xx Informational Response or an ACK response may be lost and the SIP standards specify that these messages are not re-transmitted. Only in instances where the SIP protocol provides for retries of the encompassing transaction will the call control library be able to generate proper termination events to the application when a response is lost. Applications should be written to handle cases of missing completion events that may be caused by missing response messages.

4.14 Handling SIP Transport Failures

The Global Call SIP implementation provides facilities to retry a SIP request when a transport failure occurs as well as notifying the application of the failure. The retry logic used by the SIP stack is determined by the value that is set for the E_SIP_RequestRetry field in the IP_VIRTBOARD configuration structure that is used when the virtual board is started. The default configuration enables all allowable retries.

The following code snippet illustrates the general procedure for setting up the IP_VIRTBOARD structure to specify non-default request retry behavior. This specific example disables request retries following transport failure. Note that all data structure fields that are not explicitly set are assumed to contain their default values as configured by the **INIT_IP_VIRTBOARD( )** function.

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..
..
/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data */
INIT_IPCCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);

/* initialize virtual board */
INIT_IP_VIRTBOARD(&virtBoards[0]);

// Enable SIP Message Info to allow transport selection for individual requests
virtBoards[0].ip_sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE;

//enable TCP for individual requests
virtBoards[0].E_SIP_tcpenabled = ENUM_Enabled;
virtBoards[0].E_Persistence = ENUM_PERSISTENCE_TRANSACT_USER;

//disable SIP request retry
virtboard[0].E_SIP_RequestRetry = ENUM_REQUEST_RETRY_NONE
```

Note: Features that are enabled or configured via the IP_VIRTBOARD structure cannot be disabled or reconfigured once the library has been started. All items set in this data structure take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

When UDP is used as the transport protocol, the SIP stack automatically retries the request on the same address until a timeout occurs or a response is received. When such a timeout occurs there is generally no point in retrying further on the same address, but having the stack automatically retry on any additional addresses that are contained in the DNS server may be useful. All request retry configuration options that enable retry include this type of retry using DNS entries.

When TCP is used as the transport protocol, a request may fail because the destination is not able to accept TCP in addition to other failure causes. When a TCP request fails, it is generally desirable to have the stack retry the request using UDP, but because a UDP request is retried automatically until a response is received or the request times out, the time interval before the application receives a final fatal transport error may be significantly extended. Because of this, the options for enabling request retry allow retry using UDP on the same address for a TCP failure to be enabled separately in addition to retrying using addresses from the DNS server. Additionally, the SIP stack only retries TCP requests on the same address using UDP if the failure reason indicates that there is a reasonable possibility that the UDP request will succeed. In particular, there is little point in retrying if the failure was a 503 Service Unavailable because sending a UDP request to a busy server is no more likely to succeed than the failed TCP request. Another case where retrying a failed TCP request is not appropriate is if the failed connection was a connection to a proxy, since a failed connection to a proxy indicates that the proxy is not able to accept TCP or that the proxy is down—a fatal error in either case.

An important third case occurs when an application attempts a request using UDP, but the request is forced to TCP because of its size. In this case, the application supplies an address that is valid for UDP transport because that is the protocol it assumes will be used. If the connection fails because the destination cannot accept TCP, it is appropriate for the SIP stack to retry the same address over UDP without the application's intervention, because a UDP request is what the application expected to be sent in the first place. A separate configuration option is provided to accommodate this specific circumstance while disabling retry on the same address for requests explicitly sent over TCP.

When a request retry occurs, the Global Call IP library generates a GCEV_EXTENSION event that contains the following parameter element:

IPSET_SIP_REQUEST_ERROR
  IPPARM_SIP_DNS_CONTINUE
  - value = REQUEST_ERROR data structure

If retry is not enabled in a particular circumstance, or if the retry attempt failed, the Dialogic® Global Call API library generates a GCEV_EXTENSION event containing the following parameter element:

IPSET_SIP_REQUEST_ERROR
  IPPARM_SIP_SVC_UNAVAIL
  - value = REQUEST_ERROR data structure

In both the "retry continuing" and "no retry" cases, REQUEST_ERROR.error is an enumerated error code value, and REQUEST_ERROR.method is an array that contains up to IP_SIP_METHODSIZE characters of the method name. The defined values for the error field are:

IP_SIP_REQUEST_503_RCVD
  Connection failed due to 503 Service Unavailable or other fatal error cause.

IP_SIP_REQUEST_FAILED
: Connection failed due to general or unclassified error.

IP_SIP_REQUEST_NETWORK_ERROR
: Connection failed due to network error or local failure.

IP_SIP_REQUEST_RETRY_FAILED
: Failure in request retry logic; retry not attempted.

IP_SIP_REQUEST_TIMEOUT
: Connection failed due to connection timeout.

The following code illustrates how an application can extract the failure cause information from the Extension events associated with SIP transport failures. The example assumes that the event has already been received.

```
   switch(pextensionBlk->ext_id)
   {
   .
   .
   .
      case IPSET_SIP_REQUEST_ERROR:
         ProcessRequestError(l_pParmData);
         break;
      .
      .
      .

void ProcessRequestError(GC_PARM_DATA  *parmp)
{
   REQUEST_ERROR RE;
   memcpy(&RE,parmp->value_buf,parmp->value_size);
   switch (parmp->parm_ID)
   {
      case IPPARM_SIP_DNS_CONTINUE:
         printf(" Received IPPARM_SIP_DNS_CONTINUE on %s ", RE.Method);
         break;

      case IPPARM_SIP_SVC_UNAVAIL:
         printf(" Received IPPARM_SIP_SVC_UNAVAIL on %s ",RE.Method);
         break;

      default:
         printf(" Received Unknown Request error");
         break;
   }

   switch(RE.Error)
   {
      case IP_SIP_REQUEST_NETWORK_ERROR:
         printf("IP_SIP_REQUEST_NETWORK_ERROR\n");
         break;

      case IP_SIP_REQUEST_TIMEOUT:
         printf("IP_SIP_REQUEST_TIMEOUT\n");
         break;

      case IP_SIP_REQUEST_503_RCVD:
         printf("IP_SIP_REQUEST_503_RCVD\n");
         break;

      case IP_SIP_REQUEST_FAILED:
         printf("IP_SIP_REQUEST_FAILED\n");
         break;
      default:
         printf(" Received Unknown Error cause\n");
         break;
   }
}
```

4.15 Sending and Receiving SIP INFO Messages

The SIP INFO message (as specified in IETF RFC 2976) provides a means for transporting application-level, session-related control information along the SIP signaling path after the setup of a SIP-controlled session has begun. INFO messages can be sent on an early INVITE-initiated SIP dialog (after a 101-199 provisional response) or on a confirmed dialog. The information of interest to the application can be contained in standard message header fields, proprietary header fields, or one or more MIME-encoded body parts. The Dialogic® Global Call API library provides facilities for sending and receiving INFO requests and responses on a "pass-through" basis, meaning that there are no Global Call state changes associated with such messages. The library generates Call Info events to notify applications of incoming INFO messages, and Extension events for incoming INFO response messages. The **gc_Extension( )** Send Message API is used for outgoing INFO requests and responses.

Only one INFO request can be pending on a dialog. Once an INVITE request has been sent, another one cannot be sent until a response has been received.

The following topics discuss how applications can send, receive, and respond to INFO requests.

- Sending an INFO Message
- Receiving a Response to an INFO Message
- Receiving an INFO Message
- Responding to an INFO Message

Note: Application access to the header fields in INFO messages requires that the mask value IP_SIP_MSGINFO_ENABLE must be set into the sip_msginfo_mask field of the IP_VIRTBOARD configuration data structure before **gc_Start( )** is called. Additionally, INFO messages frequently utilize MIME message bodies, and the ability to access MIME data must be enabled by setting the IP_SIP_MIME_ENABLE mask value in the same sip_msginfo_mask.

4.15.1 Sending an INFO Message

To send an INFO message, the application begins by creating a GC_PARM_BLK that contains an element with the IPSET_MSG_SIP parameter set ID, the IPPARM_MSGTYPE parameter ID and the IP_MSGTYPE_SIP_INFO parameter value. The application adds elements for the desired header fields (any combination of standard and proprietary header fields) and one or more MIME body parts, if appropriate, to the parameter block. (The technique for setting the header fields to be sent is described in Section 4.9.5, "Setting SIP Header Fields for Outbound Messages", on page 185, and the technique for constructing MIME-encoded body parts is described in Section 4.10, "Sending and Receiving MIME Bodies in SIP Messages (SIP-T)", on page 196.) After constructing the complete parameter block, the application uses the **gc_Extension( )** function to send the message. Because INFO messages relate to dialogs that have been initiated or confirmed, the **target_type** in the function call must be GCTGT_GCLIB_CRN, and the t**arget_id** must be the CRN handle for the current call.

The following standard header fields are generally required for INFO messages:

- To
- From
- Contact
- Request-URI
- Diversion
- Call-ID

Note: If the application does not explicitly set the Request-URI, the library populates it with the URI from the To header field by default.

The following standard header fields are also commonly used in INFO requests:

- Content-Disposition
- Content-Encoding

Note: The Content-Length and Content-Type header fields are normally filled in by the library and should not be set by the application.

The following code snippet illustrates the essential steps for constructing and sending an INFO request. The example assumes that a GC_PARM_BLK has already been declared.

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_INFO);

// Insert SIP Call ID field
gc_util_insert_parm_ref(&parmblkp,
                        IPSET_SIP_MSGINFO,
                        IPPARM_CALLID_HDR,
                        strlen(m_CurrentCallID),
                        m_CurrentCallID);

// Insert other SIP header information here
   .
   .
   .

// transmit INFO message to network
retval = gc_Extension(GCTGT_GCLIB_CRN, crn, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC);
   .
   .
   .
// outbound INFO has been sent.
// expect to receive a GCEV_EXTENSION containing a response
```

4.15.2 Receiving a Response to an INFO Message

After an INFO message is sent, the SIP stack will receive a response message and the library will generate a GCEV_EXTENSION event of type IPEXTID_RECEIVEMSG to notify the application.

The GC_PARM_BLK associated with Extension event will contain a parameter element as follows:

ID IPSET_MSG_SIP
  ID IPPARM_MSGTYPE
  and one of the following values:
  - IP_MSGTYPE_SIP_INFO_OK
  - IP_MSGTYPE_SIP_INFO_FAILED

The application can also retrieve the specific SIP response code from the Extension event's parameter block using the IPSET_MSG_SIP parameter set ID and the parameter ID IPPARM_MSG_SIP_RESPONSE_CODE.

Note: The application must retrieve the necessary SIP message header information by copying it into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the header information is no longer available from the metaevent buffer.

The following code snippet illustrates the procedure for extracting the INFO response information from an Extension event.

```
//  An outbound SIP INFO request has been sent previously
//  expect an inbound SIP INFO response

switch(metaeventp->evttype)
{
   case GCEV_EXTENSION:
      while ((parmp = gc_util_next_parm(pParmBlock,parmp)) != 0)
      {
         switch (parmp->set_ID)
         {
            // Handle SIP message information
            case IPSET_MSG_SIP:
               switch (parmp->parm_ID)
               {
                  // determine message type
                  case IPPARM_MSGTYPE:
                     MessType = (int)(*(parmp->value_buf));
                     switch (MessType)
                     {
                        case IP_MSGTYPE_SIP_INFO_OK:
                           // process INFO response
                           break;

                        case IP_MSGTYPE_SIP_INFO_FAILED:
                           // process INFO response
                           break;
                     }
                     break;

                   //  get the SIP response code
                  case IPPARM_MSG_SIP_RESPONSE_CODE:
                     ResponseCode = (int)(*(parmp->value_buf));
                     break;
               }
               break;
         }
      }
      break;
}
```

4.15.3 Receiving an INFO Message

When the SIP stack receives an incoming SIP INFO message, it generates a GCEV_CALLINFO event to the application.

The application can extract standard message header fields from the parameter block associated with the GCEV_CALLINFO event using the technique described in Section 4.9.6, "Retrieving SIP Message Header Fields", on page 188. If the message contains MIME-encoded information in its body (as many INFO messages do), the application can use the technique described in Section 4.10.3, "Getting MIME Information", on page 200 to extract the information.

Note: The application must retrieve the necessary SIP message header and body information by copying it into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the message information is no longer available from the metaevent buffer.

The following code snippet illustrates the essential process for extracting INFO message header information from a Call Info event.

```
switch(metaeventp->evttype)
{
   case GCEV_CALLINFO:
      pParmBlock = (GC_PARM_BLK*)(metaeventp->extevtdatap);
      parmp = NULL;

      /* going thru each parameter block data*/
      while ((parmp = gc_util_next_parm(pParmBlock,parmp)) != 0)
      {
         switch (parmp->set_ID)
         {
            /* Handle SIP message information */
            case IPSET_SIP_MSGINFO:
               switch (parmp->parm_ID)
               {
                  case IPPARM_REQUEST_URI:
                     strncpy(requestURI,(char*)parmp->value_buf,parmp->value_size);
                     sprintf(str, "gc_util_next_parm() Success, Request URI = %s",requestURI);
                     break;
                  case IPPARM_CONTACT_URI:
                     .
                     .
                     break;
                  case IPPARM_DIVERSION_URI:
                     .
                     .
                     break;
                   .
                   .
               }
               break;
            .
            .
         // etc.
            .
            .
         }
         break;
      }
      break;
}
```

4.15.4 Responding to an INFO Message

Once an application has received a GCEV_CALLINFO event for a SIP INFO message and extracted the information from the event, it must send a response message.

The response is sent by passing a GC_PARM_BLK containing the following parameter element to the **gc_Extension( )** function:

IPSET_MSG_SIP
  IPPARM_MSGTYPE
  and one of the following parameter values:
  - IP_MSGTYPE_SIP_INFO_OK
  - IP_MSGTYPE_SIP_FAILED

In addition, the application can set a specific SIP response code in the response message using the following parameter element:

IPSET_MSG_SIP
  IPPARM_MSG_SIP_RESPONSE_CODE
  and one of the following values:
  - For an “OK” response, the value should be in the range 200 to 299; if the application does not set this parameter, the Dialogic® Global Call API library fills in the default value 200.
  - For a “Failed” response, the value should be 300 or higher; if the application does not set this parameter, the Dialogic® Global Call API library fills in the default value 501.

The following two code snippets illustrate how an application would send “OK” and “Failed” responses to INFO messages.

"OK" Response to INFO Message

```
// Inbound SIP INFO request has been received
// Reply to INFO with an OK

gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_INFO_OK);

// Insert SIP response code
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSG_SIP_RESPONSE_CODE,
                        sizeof(int),
                        200);

retval = gc_SetUserInfo(GCTGT_GCLIB_CRN, crn, parmblkp, GC_SINGLECALL);

// Transmit INFO response message to network
retval = gc_Extension(GCTGT_GCLIB_CRN, crn, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC);
```

"Failed" Response to INFO Message

```
// Application has just received an inbound SIP INFO request.
// In this case, we are sending a "Not Implemented" failure response
```

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_INFO_FAILED);

// Insert SIP response code
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSG_SIP_RESPONSE_CODE,
                        sizeof(int),
                        501);

retval = gc_SetUserInfo(GCTGT_GCLIB_CRN, crn, parmblkp, GC_SINGLECALL);

// Transmit INFO response message to network
retval = gc_Extension(GCTGT_GCLIB_CRN, crn, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC);
```

4.16 Sending and Receiving SIP OPTIONS Messages

The SIP OPTIONS method provides a means for a SIP User Agent to query the capabilities of another UA or proxy, either within or outside of a SIP dialog. As an example, a client can use the OPTIONS method to discover the content types, extensions, methods, codecs, etc. that are supported by another party without having to "ring" the party by sending an INVITE.

RFC 3261 requires all user agents to support the OPTIONS method. The default behavior of the Dialogic® Global Call API library is to send automatic responses to incoming OPTIONS requests and not provide facilities for applications to send OPTIONS requests. Optionally, an IPT virtual board can be configured to enable application access to OPTIONS messages. When access is enabled, applications can send OPTIONS requests to remote parties and are responsible for responding to incoming OPTIONS requests.

The following topics describe the Dialogic® Global Call API library's implementation of support for the OPTIONS method.

- Default OPTIONS Behavior
- Enabling Application Access to OPTIONS Messages
- Sending OPTIONS Requests
- Receiving Responses to OPTIONS Requests
- Receiving OPTIONS Requests
- Responding to OPTIONS Requests

4.16.1 Default OPTIONS Behavior

If the SIP OPTIONS access feature is not enabled when the ipt virtual board device is started, the SIP stack in the Dialogic® Global Call API library responds to incoming OPTIONS requests automatically, using default information, because all SIP User Agents are required to support the OPTIONS method. The application has no control over the content of these automatic response messages, nor can it send OPTIONS requests.

When Global Call automatically responds to an incoming OPTIONS request, there are two possibilities:

- If a channel is available to handle the incoming request, Global Call sends a 200 OK message that includes an SDP message body (Content-Type: application/sdp) which indicates the same capabilities that the library would report in an outgoing INVITE request.
- If there is no channel available to handle an incoming connection request (for example, all channels in use or **gc_WaitCall( )** not having been called), Global Call sends a "busy" response. The specific code that is sent can be configured by means of the IPSET_SIP_RESPONSE_CODE/ IPPARM_BUSY_REASON parameter, but the default busy response is 486 Busy Here. This behavior allows a remote UA to use an OPTIONS request to determine whether it can initiate a new call on the target system.

The default Allow header will be the following if supplementary services (call transfer) is not enabled:
    Allow: INVITE, CANCEL, ACK, BYE
or the following if supplementary services is enabled:
    Allow: INVITE, CANCEL, ACK, BYE, REFER, NOTIFY
Note that in either case, OPTIONS is not included in the list.

4.16.2 Enabling Application Access to OPTIONS Messages

The ability to send and respond to SIP OPTIONS requests under application control is an optional feature that can be enabled or disabled at the time that the **gc_Start( )** function is called.

The mandatory **INIT_IP_VIRTBOARD( )** utility functions populates the IP_VIRTBOARD structure with default values. The default values of two fields in the IP_VIRTBOARD structure must be overridden to enable application access to OPTIONS messages:

- The E_SIP_OPTIONS_Access field must be set to ENUM_Enabled. The default value is ENUM_Disabled, which disables access to OPTIONS messages.
- The sip_msginfo-mask field must be set to the OR of IP_SIP_MSGINFO_ENABLE and IP_SIP_MIME_ENABLE (and any other appropriate mask values). The default mask value disables access to the header fields and MIME bodies of SIP messages, which would prevent the application from doing anything useful with OPTIONS messages.

See the reference page for IP_VIRTBOARD on page 665 for more information on these fields.

The following code snippet provides an example of enabling OPTIONS access for two virtual boards:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE;
ip_virtboard[1].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE;
ip_virtboard[0].E_SIP_OPTIONS_Access = ENUM_Enabled;
ip_virtboard[1].E_SIP_OPTIONS_Access = ENUM_Enabled;
```

Note: Features that are enabled or configured via the IP_VIRTBOARD structure cannot be disabled or reconfigured once the library has been started. All items set in this data structure take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

Note that in addition to enabling OPTIONS access, SIP message information access, and SIP MIME access before the virtual board is started, the application must also register the six additional SIP headers that it will need to access in OPTIONS-related messages it receives (Accept, Accept-Encoding, Accept-Language, Allow, Require, and Supported). This registration is performed on a one-time basis after the virtual board has been started, as described in Section 4.9.4, "Registering SIP Header Fields to be Retrieved", on page 182, but the header field registration list can be updated at any time.

4.16.3 Sending OPTIONS Requests

When SIP OPTIONS access is enabled, applications use **gc_Extension( )** to send the message after assembling the appropriate header fields and any MIME body parts in a GC_PARM_BLK. To build an OPTIONS request, the application uses the parameter set ID IPSET_MSG_SIP, the parameter ID IPPARM_MSGTYPE, and the parameter value IP_MSGTYPE_SIP_OPTIONS.

The application can send an OPTIONS message outside of a SIP dialog by using a board device handle in the **gc_Extension( )** call:

```
gc_Extension(GCTGT_GCLIB_CHAN, boarddevhandle, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC)
```

Alternatively, the application can send an OPTIONS request within a dialog by using the line device handle in the **gc_Extension( )** call:

```
gc_Extension(GCTGT_GCLIB_CHAN, linedevhandle, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC)
```

When SIP OPTIONS access is enabled, the Allow header field will be the following if supplementary services (call transfer) is not enabled:

    Allow: INVITE, CANCEL, ACK, BYE, OPTIONS

or the following if supplementary services is enabled:

    Allow: INVITE, CANCEL, ACK, BYE, REFER, NOTIFY, OPTIONS

The application can add additional methods to the Allow header, but the Dialogic® Global Call API library will ensure that all of the methods supported by the library are included.

The following parameters IDs are used with the IPSET_SIP_MSGINFO parameter set ID to set the header fields in the OPTIONS message, using the general techniques described in Section 4.9.5, "Setting SIP Header Fields for Outbound Messages":

parm_ID	value_buf	Default value
IPPARM_TO	To header field	Based on destination
IPPARM-REQUEST_URI	Request header URI	Derived from To header
IPPARM_FROM	From header field	Based on source
IPPARM_CONTACT_URI	Contact header URI	-none-
IPPARM_SIP_HDR	Accept header field	"Accept: application/sdp"
IPPARM_SIP_HDR	Accept-encoding header field	"Accept-encoding: " †
IPPARM_SIP_HDR	Accept-language header field	"Accept-language: en"
IPPARM_SIP_HDR	Supported header field	List of extensions supported by Global Call
IPPARM_SIP_HDR	Allow header field	List of methods supported by Global Call
IPPARM_SIP_HDR	Require header field	-none-
IPPARM_CALLID_HDR	Call-ID header field	Generated by Global Call

An empty Accept-encoding field value is permissible and equivalent to "Accept-encoding: identity", meaning no encoding

Note: If the IP Call Control library was started in the first party call control (1PCC) operating mode, the library automatically inserts a MIME body part containing SDP data that reflects the current capability set (that is, the same SDP information that would be sent in an INVITE request).This is the case even though the SDP information is not required and may not be meaningful to the User Agent that will receive the OPTIONS request (since an OPTIONS request is not part of a negotiation).

If the library was started in the third party call control (3PCC) operating mode, SDP information is **not** automatically inserted in OPTIONS requests or responses. If the application needs to include SDP information, it must explicitly insert it using **gc_SetUserInfo( )** and the IPSET_SDP / IPPARM_SDP_OPTION_OFFER or IPSET_SDP / IPPARM_SDP_OPTION_ANSWER parameter.

Once the header fields are set up, the application can send the message within a call using:

```
gc_Extension(GCTGT_GCLIB_CRN, crn, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC)
```

where `crn` is the CRN returned on a **gc_MakeCall( )** or in a GCEV_OFFERED event.

Or the application can send the message outside a dialog using:

```
gc_Extension(GCTGT_GCLIB_CHAN, boardh, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC)
```

where `boardh` is the handle obtained by opening the board device.

The following pseudo-code shows a more complete example of constructing and sending an OPTIONS request.

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_OPTIONS);
```

```
gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_TO,
                           (unsigned long)(strlen(szTo)+1),
                           szTo);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_REQUEST_URI,
                           (unsigned long)(strlen(szRURI)+1),
                           szRURI);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_FROM,
                           (unsigned long)(strlen(szFrom)+1),
                           szFrom);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_CONTACT_URI,
                           (unsigned long)(strlen(szCntct)+1),
                           szCntct);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szAccept)+1),
                           szAccept);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szAcceptE)+1),
                           szAcceptE);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szAcceptL)+1),
                           szAcceptL);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szSupp)+1),
                           szSupp);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szAllow)+1),
                           szAllow);

gc_Extension(GCTGT_GCLIB_CHAN,
             devhandle,
             IPEXTID_SENDMSG,
             parmblkp,
             &retblkp,
             EV_ASYNC);
```

4.16.4 Receiving Responses to OPTIONS Requests

When the Dialogic® Global Call API library's SIP stack receives a response to a SIP OPTIONS request, it generates a GCEV_EXTENSION event of type IPEXTID_RECEIVEMSG.

The GC_PARM_BLK associated with the Extension event will contain a parameter element as follows:

IPSET_MSG_SIP
  IPPARM_MSGTYPE parameter ID
  and one of the following values:
  - IP_MSGTYPE_SIP_OPTIONS_OK
  - IP_MSGTYPE_SIP_OPTIONS_FAILED

The application can also retrieve the specific SIP response code from the event's parameter block using the IPSET_MSG_SIP set ID and the IPPARM_MSG_SIP_RESPONSE_CODE parameter ID.

In the case of an IP_MSGTYPE_SIP_OPTIONS_OK response, the application can use the techniques described in Section 4.9.6, "Retrieving SIP Message Header Fields" to retrieve message header fields of interest, including:

- Request-URI
- To header field
- From header field
- Contact URI
- Accept header field
- Accept-encoding header field
- Accept-language header field
- Supported header field
- Allow header field
- Require header field
- Call-ID header field

The application can also extract any MIME information from the message body using the techniques described in Section 4.10.3, "Getting MIME Information", on page 200. Note that responses to OPTIONS requests are the single case where the MIME part containing SDP information is exposed to the application rather than handled internally by the Dialogic® Global Call API library. The SDP information is identified by the string "Content-Type: application/sdp".

In the case of an IP_MSGTYPE_SIP_OPTIONS_FAILED response, the application can use the techniques described in Section 4.9.6, "Retrieving SIP Message Header Fields" to retrieve the following message header fields:

- Request-URI
- To header field
- From header field

- Contact URI

Note: The application must retrieve the necessary SIP message header and body information by copying it into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the message information is no longer available from the metaevent buffer.

The following pseudo-code illustrates how to extract “OK” and “Failed” responses to OPTIONS requests from a GCEV_EXTENSION event.

```
char siphdr[IP_SIP_HDR_MAXLEN];
char AcceptHeader[IP_SIP_HDR_MAXLEN];
char Accept_encodingHeader[IP_SIP_HDR_MAXLEN];
char Accept_languageHeader[IP_SIP_HDR_MAXLEN];


case   GCEV_EXTENSION:
   if( pextensionBlk->ext_id== IPEXTID_RECEIVEMSG )
   {
      while ((l_pParm  = gc_util_next_parm(pParmBlock, l_pParm )) != 0)
      {
         int l_mtype=  (int)(*( l_pParm ->value_buf));
         switch (l_pParm ->set_ID)
         {
            case IPSET_MSG_SIP:
               if(l_pParm ->parm_ID == IPPARM_MSGTYPE)
               {
                  if(l_mtype== IP_MSGTYPE_SIP_OPTIONS_OK)
                  {
                     printf("OPTIONS request successful\n");
                  }
                  else if (l_mtype== IP_MSGTYPE_SIP_ OPTIONS_FAILED)
                  {
                     printf("OPTIONS request failedl\n");
                  }
               }
               else if(l_pParm ->parm_ID == PARM_MSG_SIP_RESPONSE_CODE)
               {
                  int *l_RC= (int *) l_pParm ->value_buf;
                  printf ("Response Code %d \n",*l_RC);
               }
            case IPSET_SIP_MSGINFO:
               switch(l_pParm ->parm_ID)
               {
                  case IPPARM_SIP_HDR:
                     strncpy(siphdr,(char*)parmp->value_buf,parmp->value_size);
                     siphdr[parmp->value_size]='\0';
                     if(!strnicmp(siphdr,"Accept-encoding",strlen("Accept-encoding" )))
                     {
                        strcpy(Accept_encodingHeader,siphdr);
                     }
                     else if (! strnicmp(siphdr,"Accept-language",strlen("Accept-language")))
                     {
                        strcpy(Accept_languageHeader,siphdr);
                     }
                     else if (! strnicmp(siphdr,"Accept",strlen("Accept")))
                     {
                        strcpy(AcceptHeader,siphdr);
                     }
                     …
                  //(process other headers)
            default :
               break;
         }
```

4.16.5 Receiving OPTIONS Requests

When the Dialogic® Global Call API library is started with the IP_VIRTBOARD.E_SIP_OPTIONS_Access field set to ENUM_Enabled (to allow application access to OPTIONS requests), the library will act in one of two ways when the SIP stack receives a SIP OPTIONS request:

- If there is no channel available to handle an incoming connection request (for example, all channels in use or **gc_WaitCall( )** not having been called), Global Call automatically sends a "busy" response. The application can set the specific code that is sent by means of the IPSET_SIP_RESPONSE_CODE/ IPPARM_BUSY_REASON parameter, but the default busy response is 486 Busy Here. The behavior of sending a busy response allows a remote UA to use an OPTIONS request to determine whether it can initiate a new call on the target system.
- If there is a channel available to handle incoming calls, the library generates an Extension event (GCEV_EXTENSION) of type IPEXTID_RECEIVEMSG to notify the application of the incoming OPTIONS request. The GC_PARM_BLK associated with the Extension event will contain a parameter element with the IPSET_MSG_SIP set ID, the IPPARM_MSGTYPE parameter ID, and the value IP_MSGTYPE_SIP_OPTIONS.

The application can use the techniques described in Section 4.9.6, "Retrieving SIP Message Header Fields" to retrieve header fields of interest, including:

- To header field
- Request URI
- From header field
- Contact URI
- Accept header field
- Accept-encoding header field
- Accept-language header field
- Supported header field
- Allow header field
- Require header field
- Call-ID header field

The application can also extract MIME information from the message body using the techniques described in Section 4.10.3, "Getting MIME Information", on page 200. Note that the MIME part that contains SDP information is **not** exposed to the application.

Note: The application must retrieve the necessary SIP message header and body information by copying the data into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the message information is no longer available from the metaevent buffer.

The following pseudo-code illustrates how to extract an OPTIONS request from a received GCEV_EXTENSION event,

```
case   GCEV_EXTENSION:
   if( pextensionBlk->ext_id== IPEXTID_RECEIVEMSG)
   {
      while ((l_pParm = gc_util_next_parm(pParmBlock, l_pParm )) != 0)
      {
         int l_mtype=  (int)(*( l_pParm->value_buf));
         switch (l_pParm->set_ID)
         {
            case IPSET_MSG_SIP:
               if(l_pParm ->parm_ID == IPPARM_MSGTYPE)
               {
                  if(l_mtype== IP_MSGTYPE_SIP_OPTIONS )
                  {
                     printf("OPTIONS request received\n");
                  }
                  …
               }
               break
            case IPSET_SIP_MSGINFO:
               switch(l_pParm ->parm_ID)
               {
                  case  IPPARM_CALLID_HDR:
                     strncpy(g_CurrentCallID,(char*)parmp->value_buf,parmp->value_size);
                     g_CurrentCallID[parmp->value_size]='\0';
                     break;
                  …
                  //(process other headers)
               default :
         break;
      }
   }
```

4.16.6 Responding to OPTIONS Requests

If SIP OPTIONS access is enabled, it is the application’s responsibility to respond to incoming OPTIONS requests, assuming that there is a channel available to handle the incoming request. (If there is no channel available, Global Call automatically sends a “busy” response.)

OPTIONS responses are sent as Global Call Extension messages using **gc_Extension( )**. There are separate message types for “OK and “Failed” response messages, but both types **must** use the Call-ID header obtained from the received request.

“Success” Response Message

“OK” responses to OPTIONS requests use the IPSET_MSG_SIP / IPPARM_MSGTYPE parameter set and ID with a value of IP_MSGTYPE_SIP_OPTIONS_OK.

The following parameters in the parameter set IPSET_SIP_MSGINFO are used to set the header fields in the OPTIONS response message, using the general techniques described in Section 4.9.5, “Setting SIP Header Fields for Outbound Messages”:

parm_ID	value_buf	Default value
IPPARM_CONTACT_URI	Contact header URI	-none-
IPPARM_SIP_HDR	Accept header field	“application/sdp”
IPPARM_SIP_HDR	Accept-encoding header field	“ “
IPPARM_SIP_HDR	Accept-language header field	“en”

parm_ID	value_buf	Default value
IPPARM_SIP_HDR	Supported header field	List of extensions supported by Global Call
IPPARM_SIP_HDR	Allow header field	List of methods supported by Global Call
IPPARM_SIP_HDR	Require header field	-none-
IPPARM_CALLID_HDR	Call-ID header field	Generated by Global Call

The Dialogic® Global Call API library ensures that the Allow header field contains all SIP methods supported by the library, which includes the following methods if supplementary services (call transfer) is not enabled:

INVITE, CANCEL, ACK, BYE, OPTIONS

or the following if supplementary services is enabled:

INVITE, CANCEL, ACK, BYE, REFER, NOTIFY, OPTIONS

When sending an "OK" response, the IP Call Control library automatically inserts a MIME body part that contains SDP data which reflects the current capability set (that is, the same SDP information that would be sent in an INVITE request). This may be the standard capability set, or the application may explicitly configure the capabilities to send in the "OK" by inserting a parameter element of the following type into the GC_PARM_BLK:

GCSET_CHAN_CAPABILITY
- IPPARM_LOCAL_CAPABILITY
  - value = IP_CAPABILITY data structure

```
gc_util_insert_parm_ref_ex(&target_datap,
                           GCSET_CHAN_CAPABILITY,
                           IPPARM_LOCAL_CAPABILITY,
                           (unsigned long)(sizeof(IP_CAPABILITY)),
                           &a_DefaultCapability);
```

The application can also send generic, non-SDP MIME information using the techniques described in Section 4.10.4, "Sending MIME Information", on page 206.

The following pseudo-code illustrates the general procedure for constructing a successful response to an OPTIONS request.

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_OPTIONS_OK);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szAccept)+1),
                           szAccept);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_CALLID_HDR,
                           (unsigned long)(strlen(g_CurrentCallID)+1,
                           g_CurrentCallID);
```

```
gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szAcceptE)+1),
                           szAcceptE);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szAcceptL)+1),
                           szAcceptL);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szSupp)+1),
                           szSupp);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szAllow)+1),
                           szAllow);

//insert a message body

gc_Extension(GCTGT_GCLIB_CHAN,
             devhandle,
             IPEXTID_SENDMSG,
             parmblkp,
             &retblkp,
             EV_ASYNC);
```

“Failed” Response Message

“Failed” responses to OPTIONS requests use the IPSET_MSG_SIP set ID and IPPARM_MSGTYPE parameter ID with a value of IP_MSGTYPE_SIP_OPTIONS_FAILED.

When sending the response message, the application **must** include the Call-ID header field value that was retrieved from the incoming OPTIONS request. The response is on the board device (that is, the **gc_Extension( )** call uses the board handle that was obtained when opening the board device), and the Call-ID is used to identify the specific request to which the response applies.

The application can also set a specific SIP response code in a “Failed” OPTIONS response message using IPSET_MSG_SIP / IPPARM_MSG_SIP_RESPONSE_CODE. If the application does not set a specific response code, Global Call uses the default value 486 (Busy Here).

The following pseudo-code illustrates sending a “Failed” response with the response code 486.

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_OPTIONS_FAILED);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_CALLID_HDR,
                           (unsigned long)(strlen(g_CurrentCallID)+1),
                           g_CurrentCallID);
```

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSG_SIP_RESPONSE_CODE,
                        sizeof(int),
                        486);

gc_Extension(GCTGT_GCLIB_CHAN,
             boardh,
             IPEXTID_SENDMSG,
             parmblkp,
             &retblkp,
             EV_ASYNC);
```

The following pseudo-code illustrates sending a “Failed” response with the response code 415, which requires Accept, Accept-Encoding, and Accept-Language header fields.

```
gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_SIP_OPTIONS_FAILED);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szAccept)+1),
                           szAccept);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_CALLID_HDR,
                           (unsigned long)(strlen(g_CurrentCallID)+1),
                           g_CurrentCallID);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szAcceptE)+1),
                           szAcceptE);

gc_util_insert_parm_ref_ex(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned long)(strlen(szAcceptL)+1),
                           szAcceptL);

gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSG_SIP_RESPONSE_CODE,
                        sizeof(int),
                        415);

gc_Extension(GCTGT_GCLIB_CHAN,
             boardh,
             IPEXTID_SENDMSG,
             parmblkp,
             &retblkp,
             EV_ASYNC);
```

4.17 Sending and Receiving SIP SUBSCRIBE and NOTIFY Messages

The SIP SUBSCRIBE and NOTIFY methods (as documented in IETF RFC 3265) provide a basic mechanism for event notification between nodes. In the most basic implementation, an entity on a network can use the SUBSCRIBE request to communicate its interest in certain state changes for resources or calls on the network, and those entities (or other entities acting on their behalf) can send NOTIFY messages as notifications when those state changes occur. This SUBSCRIBE / NOTIFY mechanism is used outside of a dialog or call.

In addition, there may be unsubscribed NOTIFY messages that are not preceded by a corresponding SUBSCRIBE request. One common use of unsubscribed NOTIFY messages is to enable and disable the Message Waiting Indicator (MWI) on a PIMG.

The Dialogic® Global Call API library for SIP fully supports both the SUBSCRIBE and NOTIFY methods, including both subscribed and unsubscribed NOTIFY. These messages are all handled on a "pass-through" basis (in other words, there are no Global Call state changes associated with the events). The Global Call Extension API mechanism is used in all cases. Outgoing requests and responses are sent by building an appropriate GC_PARM_BLK and then calling **gc_Extension( )**, while incoming requests and responses are passed to the application as GCEV_EXTENSION events.

Note that the NOTIFY messages which are used in the Dialogic® Global Call API library implementation of SIP call transfer are not handled explicitly by applications using the techniques described in this section. The Dialogic® Global Call API library handles these messages implicitly, automatically generating the outgoing NOTIFY messages that are required in a call transfer operation, and passing incoming NOTIFY messages associated with a call transfer to the application as GCEV_INVOKE_XFER or GCEV_INVOKE_XFER_FAIL events. The exception to this generalization is a NOTIFY message that is sent to the Transferor after the primary call has been dropped; in this case, the message is interpreted as a "normal" NOTIFY outside of a dialog and is passed as a GCEV_EXTENSION event that the application must explicitly accept or reject as described in Section 4.17.8, "Responding to NOTIFY Requests", on page 264. These post-termination NOTIFY messages may occur under various circumstances, including the following:

- In the normal course of events in the scenario where the Transferor is notified upon ringing of the transferred call (see Figure 26, "Successful SIP Unattended Call Transfer, Party A Notified with Ringing", on page 72)
- If a 200 OK to NOTIFY is lost in the network and the primary call is terminated by party A before party B sends another NOTIFY as a retry
- If a non-Global Call UA sends a NOTIFY for some reason after the primary call is terminated

Note that an application that will be sending and receiving SUBSCRIBE and NOTIFY messages must enable both the SIP message information (header) and SIP MIME (body) access features before starting the IPT virtual board with the **gc_Start( )** function. The **INIT_IP_VIRTBOARD( )** utility function populates the IP_VIRTBOARD structure with default values. The default values of the sip_msginfo_mask field in this structure must be overridden to enable application access to SUBSCRIBE and NOTIFY messages. Specifically, the sip_msginfo_mask field must be set to the OR of IP_SIP_MSGINFO_ENABLE and IP_SIP_MIME_ENABLE. See the reference page for IP_VIRTBOARD on page 665 for more information on this field and these mask values.

The following code snippet provides an example of enabling message header and body access for two virtual boards:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE;
ip_virtboard[1].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE;
```

The following topics describe how applications send, receive, and respond to SUBSCRIBE and NOTIFY requests:

- Sending SUBSCRIBE Requests
- Receiving Responses to SUBSCRIBE Requests
- Receiving SUBSCRIBE Requests
- Responding to SUBSCRIBE Requests
- Sending NOTIFY Requests
- Receiving Responses to NOTIFY Requests
- Receiving NOTIFY Requests
- Responding to NOTIFY Requests

4.17.1 Sending SUBSCRIBE Requests

To send a SUBSCRIBE request message, the application begins by creating a GC_PARM_BLK that contains an element with the IPSET_MSG_SIP set ID, the IPPARM_MSGTYPE parameter ID and the IP_MSGTYPE_SIP_SUBSCRIBE parameter value. The application adds elements for the desired header fields and one or more MIME body parts, if appropriate, to the parameter block, then uses the **gc_Extension( )** function to send the message. The header may include any combination of standard header fields and proprietary header fields. General techniques for setting header fields are described in Section 4.9.5, “Setting SIP Header Fields for Outbound Messages”. The technique for constructing MIME body parts is described in Section 4.10.4, “Sending MIME Information”.

The header fields that normally must be set in a SUBSCRIBE request include the following:

- To display string (IPPARM_TO_DISPLAY)
- From display string (IPPARM_FROM_DISPLAY)
- Expires header field (IPPARM_EXPIRES_HDR)
- Event header field (IPPARM_EVENT_HDR)
- Call-ID header field (IPPARM_CALLID_HDR)

SUBSCRIBE requests normally contain an Expires header field, which indicates the duration of the subscription. When the application does not explicitly set an Expires header field, the default duration that is defined in the SIP “event package” for the particular type of event will apply. To keep a subscription effective beyond the accepted duration, the subscriber needs to send a new SUBSCRIBE message on the same dialog when it receives an expiration message. To terminate or unsubscribe an existing subscription, the application can send a SUBSCRIBE request with the value 0 in the Expires header field to specify immediate expiration.

The following code snippet illustrates how an application constructs and sends a SUBSCRIBE request.

```
void CSubNotMgr::SendSIPSubscribe (char* pRequestURI,
                                                    char* pTo,
                                                    char* pFrom,
                                                    char* pExpire,
                                                    char* pEvent,
                                                    char* pCallID)
{
      char str[MAX_STRING_SIZE];
      sprintf(str, "<--- Sending SIP SUBSCRIBE\n");
      printandLog(ALL_DEVICES, MISC, NULL, str, 0);

      GC_PARM_BLKP parmblkp = NULL; // input parameter block pointer
      GC_PARM_BLKP retblkp = NULL; // return parameter block
      GC_INFO gc_error_info; // GlobalCall error information data
      int retval = GC_SUCCESS;

      gc_util_insert_parm_val(&parmblkp,
             IPSET_MSG_SIP,
             IPPARM_MSGTYPE,
             sizeof(int),
             IP_MSGTYPE_SIP_SUBSCRIBE);

      // Insert SIP request URI field
      if (pRequestURI)
      {
             gc_util_insert_parm_ref_ex(&parmblkp,
             IPSET_SIP_MSGINFO,
             IPPARM_REQUEST_URI,
             (unsigned long)(strlen(pRequestURI)+1),
             pRequestURI);
      }

      // Insert SIP To field
      if (pTo)
     {
           gc_util_insert_parm_ref_ex(&parmblkp,
           IPSET_SIP_MSGINFO,
           IPPARM_TO_DISPLAY,
          (unsigned long)(strlen(pTo)+1),
           pTo);
      }

     // Insert SIP From field
     if (pFrom)
     {
            gc_util_insert_parm_ref_ex(&parmblkp,
            IPSET_SIP_MSGINFO,
            IPPARM_FROM_DISPLAY,
            (unsigned long)(strlen(pFrom)+1),
            pFrom);
      }

     // Insert SIP Expire field
     if (pExpire)
     {
            gc_util_insert_parm_ref_ex(&parmblkp,
            IPSET_SIP_MSGINFO,
            IPPARM_EXPIRES_HDR,
            (unsigned long)(strlen(pExpire)+1),
            pExpire);
      }

      // Insert SIP Event field
```

```
            if (pEvent)
            {
                   gc_util_insert_parm_ref_ex(&parmblkp,
                   IPSET_SIP_MSGINFO,
                   IPPARM_EVENT_HDR,
                   (unsigned long)(strlen(pEvent)+1),
                   pEvent);
             }

             // Insert SIP Call ID field
             if (pCallID)
             {
                    gc_util_insert_parm_ref_ex(&parmblkp,
                    IPSET_SIP_MSGINFO,
                    IPPARM_CALLID_HDR,
                    (unsigned long)(strlen(pCallID)+1),
                    pCallID);
              }

              if (parmblkp == NULL)
              {
                    // memory allocation error
                    return;
              }

              // transmit SUBSCRIBE message to network
              retval = gc_Extension(GCTGT_GCLIB_CHAN, CIPChannel::s_gcBrdDev,
                       IPEXTID_SENDMSG, parmblkp,
                       &retblkp, EV_ASYNC);

              if (retval != GC_SUCCESS)
              {
                       gc_ErrorInfo( &gc_error_info );
                       printf ("Error : gc_Extension() on HANDLE: 0x%lx, GC ErrorValue: 0x%hx -
                                %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                                CIPChannel::s_gcBrdDev, gc_error_info.gcValue,
                                gc_error_info.gcMsg,gc_error_info.ccLibId,
                                gc_error_info.ccLibName, gc_error_info.ccValue,
                                gc_error_info.ccMsg);

                       return;
                }
                // clean up
                gc_util_delete_parm_blk(parmblkp);
       }
```

4.17.2 Receiving Responses to SUBSCRIBE Requests

After a SUBSCRIBE request is sent, the remote entity responds with an accept or reject reply, which the call control library passes to the application as a GCEV_EXTENSION event of type IPEXTID_RECEIVEMSG.

The data associated with the Extension event will contain the following parameter element:

IPSET_MSG_SIP
    IPPARM_MSGTYPE
    and one of the following two values:
- IP_MSGTYPE_SIP_SUBSCRIBE_ACCEPT
- IP_MSGTYPE_SIP_SUBSCRIBE_REJECT

Additionally, the subscriber application may periodically receive an event that indicates the expiration of the subscription duration. Note that the application does not have to respond to an expiration message because the message indicates that the transaction is no longer active. The data associated with the expiration message event is:

IPSET_MSG_SIP
  IPPARM_MSGTYPE
  - value = IP_MSGTYPE_SIP_SUBSCRIBE_EXPIRE

Note: The application must retrieve the necessary SIP message header information by copying it into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the header information is no longer available from the metaevent buffer.

The following example illustrates the general procedure for extracting information from the Extension event for any of the incoming messages associated with the SUBSCRIBE and NOTIFY methods.

```
// main event loop
// get a GCEV_EXTENSION event and process it
void process_event(void)
{
   METAEVENT  metaevent;
   int        evttype;

   gc_GetMetaEvent(&metaevent);
   evttype = metaevent.evttype;

   GC_PARM_BLK  *pParmBlock = NULL;
   GC_PARM_DATA *parmp = NULL;

   switch (evttype)
   {
      case GCEV_EXTENSION:
         OnExtensionEvent(&metaevent);
         break;
   }
}

// process GCEV_EXTENSION event
// get SIP Msg and SIP Msg Info
void OnExtensionEvent(METAEVENT *metaeventp)
{
   GC_PARM_BLK      *pParmBlock = NULL;
   EXTENSIONEVTBLK  *pExtensionBlock = NULL;
   GC_PARM_DATA     *parmp = NULL;

   pExtensionBlock = (EXTENSIONEVTBLK*)(metaeventp->extevtdatap);
   pParmBlock = &pExtensionBlock->parmblk;

   parmp = NULL;
   int CurrentMessage = 0;

   // going thru each parameter block data
   while ((parmp = gc_util_next_parm(pParmBlock,parmp)) != 0)
   {
      switch (parmp->set_ID)
      {
         // Handle SIP message information
         case IPSET_MSG_SIP:
            CurrentMessage = ProcessSIPMsg(parmp);
            break;
         /* Handle SIP message information */
         case IPSET_SIP_MSGINFO:
            ProcessSIPMsgInfo(parmp);
            break;

         default:
            break;
      }
   }

   pParmBlock = (GC_PARM_BLK*)(metaeventp->extevtdatap);
   parmp = NULL;
}

// determine type of SIP Message and process accordingly
int CSubNotMgr::ProcessSIPMsg(GC_PARM_DATA  *parmp)
{
   int MessType=0;
   switch (parmp->parm_ID)
   {
      case IPPARM_MSGTYPE:
      {
         MessType = (int)(*(parmp->value_buf));
         switch (MessType)
         {
            case IP_MSGTYPE_SIP_SUBSCRIBE:
               // process here
               break;
            case IP_MSGTYPE_SIP_SUBSCRIBE_ACCEPT:
               // process here
               break;
            case IP_MSGTYPE_SIP_SUBSCRIBE_REJECT:
               // process here
               break;
            case IP_MSGTYPE_SIP_SUBSCRIBE_EXPIRE:
               // process here
               break;
            case IP_MSGTYPE_SIP_NOTIFY:
               // process here
               break;
            case IP_MSGTYPE_SIP_NOTIFY_ACCEPT:
               // process here
               break;
            case IP_MSGTYPE_SIP_NOTIFY_REJECT:
               // process here
               break;
            default:
               break;
         }
         break;
      }
      default:
         break;
   }
   return MessType;
}

// process SIP Msg Info
void CSubNotMgr::ProcessSIPMsgInfo(GC_PARM_DATA  *parmp)
{
   char   requestURI[IP_REQUEST_URI_MAXLEN];
   char   contactURI[IP_CONTACT_URI_MAXLEN];
   char   diversionURI[IP_DIVERSION_URI_MAXLEN];
   char   event[IP_EVENT_HDR_MAXLEN];
   char   expires[IP_EXPIRES_HDR_MAXLEN];
   switch (parmp->parm_ID)
   {
      case IPPARM_REQUEST_URI:
         strncpy(requestURI,(char*)parmp->value_buf,parmp->value_size);
         requestURI[parmp->value_size]='\0';
         break;
      case IPPARM_CONTACT_URI:
         strncpy(contactURI,(char*)parmp->value_buf,parmp->value_size);
         contactURI[parmp->value_size]='\0';
         break;
      case IPPARM_DIVERSION_URI:
         strncpy(diversionURI,(char*)parmp->value_buf,parmp->value_size);
         diversionURI[parmp->value_size]='\0';
         break;
      case IPPARM_EVENT_HDR:
         strncpy(event,(char*)parmp->value_buf,parmp->value_size);
         event[parmp->value_size]='\0';
         break;
      case IPPARM_EXPIRES_HDR:
         strncpy(expires,(char*)parmp->value_buf,parmp->value_size);
         expires[parmp->value_size]='\0';
         break;
      case IPPARM_CALLID_HDR:
         strncpy(m_CurrentCallID,(char*)parmp->value_buf,parmp->value_size);
         m_CurrentCallID[parmp->value_size]='\0';
         break;
      default:
         break;
   }
}
```

4.17.3 Receiving SUBSCRIBE Requests

When the SIP stack receives a SIP SUBSCRIBE request, the Dialogic® Global Call API library generates an Extension event of type IPEXTID_RECEIVEMSG. The data associated with this Extension event contains the following parameter element:

IPSET_MSG_SIP
    IPPARM_MSGTYPE
    - value = IP_MSGTYPE_SIP_SUBSCRIBE

The application can use the techniques described in Section 4.9.6, “Retrieving SIP Message Header Fields” to retrieve message header fields of interest, including:

- To display string
- From display string
- Expires header field
- Event header field
- Call-ID header field

If the message has a body, the application can extract the MIME-encoded information using the techniques described in Section 4.10.3, “Getting MIME Information”.

Note: The application must retrieve the necessary SIP message header and body information by copying the data into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the message information is no longer available from the metaevent buffer.

A code example that illustrates the general procedure for retrieving information from all incoming messages associated with the SUBSCRIBE and NOTIFY methods is included in Section 4.17.2, "Receiving Responses to SUBSCRIBE Requests", on page 254.

4.17.4 Responding to SUBSCRIBE Requests

Once an application has received a GCEV_EXTENSION event for a SIP SUBSCRIBE request and extracted the information from the event, it must send a response message.

The response is sent as an Extension message, passing a parameter block that contains the following element:

IPSET_MSG_SIP
    IPPARM_MSGTYPE
    and one of the following two parameter values:
- IP_MSGTYPE_SIP_SUBSCRIBE_ACCEPT
- IP_MSGTYPE_SIP_SUBSCRIBE_REJECT

The "Accept" message is a 200 OK, while the "Reject" message is a 501 response. In either case, the response message **must** include the Call-ID header field value that was received in the SUBSCRIBE request so that the subscriber can match the response to the request.

The following two code snippets illustrate how an application would send "Accept" and "Reject" responses to SUBSCRIBE requests.

"Accept" response to SUBSCRIBE request

When accepting a SUBSCRIBE request, a SIP entity normally includes an Expires header field, which may contain the same value that was received in the Expires header field of the SUBSCRIBE request or any smaller value.

```
void CSubNotMgr::SendSIPSubscribeAccept (char* pExpire)
{
   char        str[MAX_STRING_SIZE];
   sprintf(str, "<--- Sending SIP SUBSCRIBE Accept\n");
   printandlog(ALL_DEVICES, MISC, NULL, str, 0);

   GC_PARM_BLKP   parmblkp = NULL;   // input parameter block pointer
   GC_PARM_BLKP   retblkp = NULL;    // return parameter block
   GC_INFO        gc_error_info;     // GlobalCall error information data
   int            retval = GC_SUCCESS;

   gc_util_insert_parm_val(&parmblkp,
                           IPSET_MSG_SIP,
                           IPPARM_MSGTYPE,
                           sizeof(int),
                           IP_MSGTYPE_SIP_SUBSCRIBE_ACCEPT);

   // Insert SIP Expire field
   gc_util_insert_parm_ref_ex(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_EXPIRES_HDR,
                              (unsigned long)(strlen(pExpire)+1),
                              pExpire);
   // Insert SIP Call ID field
   gc_util_insert_parm_ref_ex(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_CALLID_HDR,
                              (unsigned long)(strlen(m_CurrentCallID)+1),
                              m_CurrentCallID);

   if (parmblkp == NULL)
   {
      // memory allocation error
      return;
   }

   // transmit NOTIFY message to network
   retval = gc_Extension(GCTGT_GCLIB_CHAN, boardh,
                         IPEXTID_SENDMSG, parmblkp,
                         &retblkp, EV_ASYNC);

   if (retval != GC_SUCCESS)
   {
      gc_ErrorInfo( &gc_error_info );
      printf ("Error : gc_Extension() on HANDLE: 0x%lx,
              GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
              CC ErrorValue: 0x%lx - %s\n", boardh,
              gc_error_info.gcValue, gc_error_info.gcMsg,
              gc_error_info.ccLibId, gc_error_info.ccLibName,
              gc_error_info.ccValue, gc_error_info.ccMsg);
      return;
   }

   // clean up
   gc_util_delete_parm_blk(parmblkp);

   m_bSubscribeAcceptSent = true;
}
```

“Reject” response to SUBSCRIBE request

```
void CSubNotMgr::SendSIPSubscribeReject (void)
{
   char        str[MAX_STRING_SIZE];
   sprintf(str, "<--- Sending SIP SUBSCRIBE Reject\n");
   printandlog(ALL_DEVICES, MISC, NULL, str, 0);

   GC_PARM_BLKP   parmblkp = NULL;   // input parameter block pointer
   GC_PARM_BLKP   retblkp = NULL;    // return parameter block
   GC_INFO        gc_error_info;     // GlobalCall error information data
   int            retval = GC_SUCCESS;

   gc_util_insert_parm_val(&parmblkp,
                           IPSET_MSG_SIP,
                           IPPARM_MSGTYPE,
                           sizeof(int),
                           IP_MSGTYPE_SIP_SUBSCRIBE_REJECT);

   // Insert SIP Call ID field
   gc_util_insert_parm_ref_ex(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_CALLID_HDR,
                              (unsigned long)(strlen(m_CurrentCallID)+1),
                              m_CurrentCallID);
   if (parmblkp == NULL)
   {
      // memory allocation error
      return;
   }
   // transmit NOTIFY message to network
   retval = gc_Extension(GCTGT_GCLIB_CHAN, boardh,
                         IPEXTID_SENDMSG, parmblkp,
                         &retblkp, EV_ASYNC);

   if (retval != GC_SUCCESS)
   {
      gc_ErrorInfo( &gc_error_info );
      printf ("Error : gc_Extension() on HANDLE: 0x%lx,
              GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
              CC ErrorValue: 0x%lx - %s\n", boardh,
              gc_error_info.gcValue, gc_error_info.gcMsg,
              gc_error_info.ccLibId, gc_error_info.ccLibName,
              gc_error_info.ccValue, gc_error_info.ccMsg);
      return;
   }

   // clean up
   gc_util_delete_parm_blk(parmblkp);

   m_bSubscribeRejectSent = true;
}
```

4.17.5 Sending NOTIFY Requests

To send a NOTIFY message, the application begins by creating a GC_PARM_BLK that contains an element of the following type:

IPSET_MSG_SIP
    IPPARM_MSGTYPE
    - value = IP_MSGTYPE_SIP_NOTIFY

The application adds elements for the desired header fields and one or more MIME body parts, if appropriate, to the parameter block, then uses the **gc_Extension( )** function to send the message. The header fields that can be set and the general technique for setting them are described in Section 4.9.5, "Setting SIP Header Fields for Outbound Messages". The technique for constructing MIME bodies is described in Section 4.10.4, "Sending MIME Information".

The header fields that normally must be set in a NOTIFY request include the following:

- Request-URI
- To display string
- From display string
- Event header field
- Call-ID header field

If the NOTIFY being sent is a subscribed NOTIFY, the Call-ID header field must contain the same Call-ID value as the SUBSCRIBE request that the NOTIFY relates to. For an unsubscribed NOTIFY, the Call-ID header field must be NULL.

The following code snippet illustrates how an application constructs and sends a NOTIFY request.

Note: For an unsubscribed NOTIFY, the "To" and "From" display strings should be populated with both IP address and Display information. For a subscribed NOTIFY, only Display information is required since the IP address is obtained from the SUBSCRIBE.

```
void CSubNotMgr::SendSIPNotify ( char* pRequestURI,
                                 char* pTo,
                                 char* pFrom,
                                 char* pEvent,
                                 char* pBody,
                                 char* pCallID)

{
   char str[MAX_STRING_SIZE];
   char *pBlankBody = " ";
   sprintf(str, "<--- Sending SIP NOTIFY on device %d\n", hsendboard);
   printandlog(ALL_DEVICES, MISC, NULL, str, 0);

   GC_PARM_BLKP parmblkp = NULL;    // input parameter block pointer
   GC_PARM_BLKP parmblkbody = NULL; // body parms
   GC_PARM_BLKP retblkp = NULL;     // return parameter block
   GC_INFO      gc_error_info;      // GlobalCall error information data
   int          retval = GC_SUCCESS;

   // Insert SIP message type
   gc_util_insert_parm_val(&parmblkp,
                           IPSET_MSG_SIP,
                           IPPARM_MSGTYPE,
                           sizeof(int),
                           IP_MSGTYPE_SIP_NOTIFY);

   // Insert SIP Request-URI
   if (pRequestURI)
   {
      gc_util_insert_parm_ref_ex(&parmblkp,
                                 IPSET_SIP_MSGINFO,
                                 IPPARM_REQUEST_URI,
                                 (unsigned long)(strlen(pRequestURI)+1),
                                 pRequestURI);
   }

   // Insert SIP To field
   if (pTo)
   {
      gc_util_insert_parm_ref_ex(&parmblkp,
                                 IPSET_SIP_MSGINFO,
                                 IPPARM_TO_DISPLAY,
                                 (unsigned long)(strlen(pTo)+1),
                                 pTo);
   }

   // Insert SIP From field
   if (pFrom)
   {
      gc_util_insert_parm_ref_ex(&parmblkp,
                                 IPSET_SIP_MSGINFO,
                                 IPPARM_FROM_DISPLAY,
                                 (unsigned long)(strlen(pFrom)+1),
                                 pFrom);

      //Insert SIP Contact header field
      gc_util_insert_parm_ref_ex(&parmblkp,
                                 IPSET_SIP_MSGINFO,
                                 IPPARM_CONTACT_URI,
                                 (unsigned long)(strlen(pFrom)+1),
                                 pFrom);
   }
```

```
// Insert SIP Event field
if (pEvent)
{
   gc_util_insert_parm_ref_ex(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_EVENT_HDR,
                              (unsigned long)(strlen(pEvent)+1),
                              pEvent);
}

// Insert SIP CallID field
if (pCallID)
{
   gc_util_insert_parm_ref_ex(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_CALLID_HDR,
                              (unsigned long)(strlen(pCallID)+1),
                              pCallID);
}

// Insert the message Body
if (pBody)
{

   // Insert Content-Type field
   // Add 1 to strlen for the NULL termination character
   gc_util_insert_parm_ref_ex(&parmblkbody,
                              IPSET_MIME,
                              IPPARM_MIME_PART_TYPE,
                              (unsigned long)(strlen(pBody)+1),
                              pBody);

   // Insert Body Size
   gc_util_insert_parm_val(&parmblkbody,
                           IPSET_MIME,
                           IPPARM_MIME_PART_BODY_SIZE,
                           sizeof(unsigned long),
                           strlen(pBlankBody));

   // Insert MIME part Body Pointer
   gc_util_insert_parm_val(&parmblkbody,
                           IPSET_MIME,
                           IPPARM_MIME_PART_BODY,
                           sizeof(unsigned long),
                           (unsigned long)pBlankBody);

   // Insert parm block B pointer to parm block A
   gc_util_insert_parm_val(&parmblkp, //pParmBlockA,
                           IPSET_MIME,
                           IPPARM_MIME_PART,
                           sizeof(unsigned long),
                           (unsigned long)parmblkbody);

   if (parmblkbody == NULL)
   {
      // memory allocation error
      return;
   }
}

if (parmblkp == NULL)
{
   // memory allocation error
   return;
}
```

```
    // transmit NOTIFY message to network
    retval = gc_Extension(GCTGT_GCLIB_CHAN,
                          hsendboard,
                          IPEXTID_SENDMSG,
                          parmblkp,
                          &retblkp,
                          EV_ASYNC);

    if (retval != GC_SUCCESS)
    {
       gc_ErrorInfo( &gc_error_info );
       printf ("Error : gc_Extension() on HANDLE: 0x%lx,
               GC ErrorValue: 0x%hx - %s,
               CCLibID: %i - %s,
               CC ErrorValue: 0x%lx - %s\n",
               boardh,
               gc_error_info.gcValue,
               gc_error_info.gcMsg,
               gc_error_info.ccLibId,
               gc_error_info.ccLibName,
               gc_error_info.ccValue,
               gc_error_info.ccMsg);
       return;
    }

    // clean up
    gc_util_delete_parm_blk(parmblkp);
    if (pBody) gc_util_delete_parm_blk(parmblkbody);

    m_bNotifySent=true;
}
```

4.17.6 Receiving Responses to NOTIFY Requests

After a NOTIFY request is sent, the remote entity responds with an accept or reject reply, which the call control library sends to the application as a GCEV_EXTENSION event of type IPEXTID_RECEIVEMSG.

The GC_PARM_BLK associated with the Extension event for a NOTIFY response contains the following parameter element:

IPSET_MSG_SIP
    IPPARM_MSGTYPE
    and one of the following two values:
- IP_MSGTYPE_SIP_NOTIFY_ACCEPT
- IP_MSGTYPE_SIP_NOTIFY_REJECT

Note: The application must retrieve the necessary SIP message header information by copying it into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the header information is no longer available from the metaevent buffer.

A code example that illustrates the general technique for retrieving information from all incoming messages associated with the SUBSCRIBE and NOTIFY methods is included in Section 4.17.2, “Receiving Responses to SUBSCRIBE Requests”, on page 254.

4.17.7 Receiving NOTIFY Requests

When the SIP stack receives a SIP NOTIFY request, the Dialogic® Global Call API library generates an Extension event (GCEV_EXTENSION) of type IPEXTID_RECEIVEMSG.

The data associated with this Extension event contains a parameter element as follows:

IPSET_MSG_SIP
  IPPARM_MSGTYPE
  - value = IP_MSGTYPE_SIP_NOTIFY

Both subscribed and unsubscribed NOTIFY requests can be received; in the case of a subscribed NOTIFY, the value of the Call-ID header field will match the Call-ID of a previously sent SUBSCRIBE request.

The application can use the techniques described in Section 4.9.6, "Retrieving SIP Message Header Fields" to retrieve message header fields of interest, including:

- To display string
- From display string
- Event header field
- Call-ID header field

If the message has a body, the application can extract the MIME-encoded information using the techniques described in Section 4.10.3, "Getting MIME Information".

Note: The application must retrieve the necessary SIP message header and body information by copying the data into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the message information is no longer available from the metaevent buffer.

A code example that illustrates the general procedure for retrieving information from all incoming messages associated with the SUBSCRIBE and NOTIFY methods is included in Section 4.17.2, "Receiving Responses to SUBSCRIBE Requests", on page 254.

4.17.8 Responding to NOTIFY Requests

Once an application has received a GCEV_EXTENSION event for a SIP NOTIFY message (either subscribed or unsubscribed) and extracted the information from the event, it must send a response message.

The response is sent as an Extension message using the following parameter element in the parameter block:

IPSET_MSG_SIP
  IPPARM_MSGTYPE
  and one of the following two parameter values:
  - IP_MSGTYPE_SIP_NOTIFY_ACCEPT
  - IP_MSGTYPE_SIP_NOTIFY_REJECT

For an “Accept” response the message sent is a 200 OK, while “Reject” sends a 501 response. In either case, the response message must include the Call-ID header that was received in the NOTIFY request.

The following two code snippets illustrate how an application would send “Accept” and “Reject” responses to NOTIFY requests.

“Accept” Response to NOTIFY Request

```
void CSubNotMgr::SendSIPNotifyAccept ()
{
   char        str[MAX_STRING_SIZE];
   sprintf(str, "<--- Sending SIP NOTIFY Accept\n");
   printandlog(ALL_DEVICES, MISC, NULL, str, 0);

   GC_PARM_BLKP   parmblkp = NULL;  // input parameter block pointer
   GC_PARM_BLKP   retblkp = NULL;   // return parameter block
   GC_INFO        gc_error_info;    // GlobalCall error information data
   int            retval = GC_SUCCESS;

   gc_util_insert_parm_val(&parmblkp,
                           IPSET_MSG_SIP,
                           IPPARM_MSGTYPE,
                           sizeof(int),
                           IP_MSGTYPE_SIP_NOTIFY_ACCEPT);

   // Insert SIP Call ID field
   gc_util_insert_parm_ref_ex(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_CALLID_HDR,
                              (unsigned long)(strlen(m_CurrentCallID)+1),
                              m_CurrentCallID);

   if (parmblkp == NULL)
   {
      // memory allocation error
      return;
   }

   // transmit NOTIFY message to network
   retval = gc_Extension(GCTGT_GCLIB_CHAN, boardh,
                         IPEXTID_SENDMSG, parmblkp,
                         &retblkp, EV_ASYNC);

   if (retval != GC_SUCCESS)
   {
      gc_ErrorInfo( &gc_error_info );
      printf ("Error : gc_Extension() on HANDLE: 0x%lx,
              GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
              CC ErrorValue: 0x%lx - %s\n", boardh,
              gc_error_info.gcValue, gc_error_info.gcMsg,
              gc_error_info.ccLibId, gc_error_info.ccLibName,
              gc_error_info.ccValue, gc_error_info.ccMsg);
      return;
   }

   // clean up
   gc_util_delete_parm_blk(parmblkp);

   m_bNotifyAcceptSent = true;
}
```

"Reject" Response to NOTIFY Request

```
void CSubNotMgr::SendSIPNotifyReject (void)
{
   char        str[MAX_STRING_SIZE];
   sprintf(str, "<--- Sending SIP NOTIFY Reject\n");
   printandlog(ALL_DEVICES, MISC, NULL, str, 0);

   GC_PARM_BLKP  parmblkp = NULL; // input parameter block pointer
   GC_PARM_BLKP  retblkp = NULL;  // return parameter block
   GC_INFO       gc_error_info;   // GlobalCall error information data
   int           retval = GC_SUCCESS;

   gc_util_insert_parm_val(&parmblkp,
                           IPSET_MSG_SIP,
                           IPPARM_MSGTYPE,
                           sizeof(int),
                           IP_MSGTYPE_SIP_NOTIFY_REJECT);

   // Insert SIP Call ID field
   gc_util_insert_parm_ref_ex(&parmblkp,
                              IPSET_SIP_MSGINFO,
                              IPPARM_CALLID_HDR,
                              (unsigned long)(strlen(m_CurrentCallID)+1),
                              m_CurrentCallID);
   if (parmblkp == NULL)
   {
      // memory allocation error
   return;
   }

   // transmit NOTIFY message to network
   retval = gc_Extension(GCTGT_GCLIB_CHAN, boardh,
                         IPEXTID_SENDMSG, parmblkp,
                         &retblkp, EV_ASYNC);

   if (retval != GC_SUCCESS)
   {
      gc_ErrorInfo( &gc_error_info );
      printf ("Error : gc_Extension() on HANDLE: 0x%lx,
              GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
              CC ErrorValue: 0x%lx - %s\n", boardh,
              gc_error_info.gcValue, gc_error_info.gcMsg,
              gc_error_info.ccLibId, gc_error_info.ccLibName,
              gc_error_info.ccValue, gc_error_info.ccMsg);
      return;
   }

   // clean up
   gc_util_delete_parm_blk(parmblkp);

   m_bNotifyRejectSent = true;
}
```

4.18 Sending and Receiving SIP UPDATE Messages

The application can send and receive RFC 3311 SIP UPDATE requests and responses in third-party call control (3PCC) mode.

The application can send and receive SIP UPDATE requests and responses to modify the state of a pending or pre-established session. Prior to the availability of this feature, the UPDATE message was only supported in conjunction with SIP Session Timer refreshing and did not address the updating of session parameters. This feature uses the Global Call Extension API mechanism for application notification and message construction.

The following topics provide more information about SIP UPDATE:

- Enabling the Feature
- Usage Scenarios
- Handling SIP UPDATE Post-Connection Messages

4.18.1 Enabling the Feature

The E_SIP_UPDATE_Access field in the IP_VIRTBOARD data structure allows the application to specify whether to send and receive UPDATE requests and responses. The default value of this field is ENUM_Disabled. To enable the feature, set the field to ENUM_Enabled.

The IPSET_MSG_SIP set ID and IPPARM_MSGTYPE parameter ID with the following values support this feature:

IPPARM_MSG_SIP_RESPONSE_CODE
: Inserted or extracted from the parameter block. Used by the application to set or get the SIP-specific response code.

  Associated SIP Response Codes: any

IP_MSGTYPE_SIP_UPDATE
: Contained in the parameter block when calling the **gc_Extension( )** function or when receiving a GCEV_EXTENSION event. Used by the application to set or determine the event type.

  Associated SIP Response Codes: not applicable.

IP_MSGTYPE_SIP_UPDATE_OK
: Contained in the parameter block when calling the **gc_Extension( )** function or when receiving a GCEV_EXTENSION event. Used by the application to set or determine the event type. The SIP-specific response code can be inserted or extracted from the IP_MSGTYPE_SIP_UPDATE event.

  Associated SIP Response Codes: 200-299.

IP_MSGTYPE_SIP_UPDATE_FAILED
: Contained in the parameter block when calling the **gc_Extension( )** function or when receiving a GCEV_EXTENSION event. Used by the application to set or determine the event type. The SIP-specific response code can be inserted or extracted from the IP_MSGTYPE_SIP_UPDATE_FAILED event.

  Associated SIP Response Codes: 300+

The following code snippet demonstrates how to enable this feature at the board level to support sending and receiving a SIP UPDATE.

```
.
.
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
```

```
Ip_virtboard[0].E_SIP_UPDATE_Access = ENUM_Enabled;
Ip_virtboard[1].E_SIP_UPDATE_Access = ENUM_Enabled;
Ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE;
Ip_virtboard[1].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE;
.
.
```

4.18.2 Usage Scenarios

The following sections present usage scenarios for SIP UPDATE. Sample code is provided after each scenario.

Note: All examples shown in this section are pre-connection only.

4.18.2.1 Sending an UPDATE Request

When SIP UPDATE access is enabled, applications use the **gc_Extension( )** function to send the message after assembling the appropriate header fields and any SDP parts. To build an UPDATE request, the application uses the parameter set ID IPSET_MSG_SIP, the parameter ID IPPARM_MSGTYPE, and the parameter value IP_MSGTYPE_SIP_UPDATE.

The application can send an UPDATE request within a dialog by using the line device handle in the **gc_Extension( )** function call: **gc_Extension**(GCTGT_GCLIB_CHAN, linedevhandle, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC).

If the application requires SDP information, the information must be explicitly inserted using the **gc_SetUserInfo( )** function and the IPSET_SDP, IPPARM_SDP_OFFER, IPPARM_SDP_ANSWER parameter.

Once the header fields are set up, the application can send the message within a dialog using: **gc_Extension**(GCTGT_GCLIB_CRN, crn, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC).

Example

```
// Set the SDP for 3PCC only
printf("\n Setting SDP \n");
GC_PARM_BLKP libBlock = NULL;
libBlock = set3PCCSDPInfo1(libBlock, 0, true);

if (gc_SetUserInfo(GCTGT_GCLIB_CRN,
                   pline->crn,
                   libBlock,
                   GC_SINGLECALL) != GC_SUCCESS)
{
   printf("\n gc_SetUserInfo failed for setting SDP \n");
   exit(1);
}

if (libBlock) gc_util_delete_parm_blk(libBlock);

// Send the UPDATE message
pline->crn = callindex;
printf("Sending UPDATE ....... \n");
sendUpdate(pline->crn);
..
```

```
..
..

GC_PARM_BLKP  set3PCCSDPInfo1(GC_PARM_BLKP libBlock,int index, bool isAnswer)
{
   if (isAnswer) {} // For compilation

   char sdp_buf[1024];
   int ip_parm = IPPARM_SDP_ANSWER;

   sprintf(sdp_buf,
           "v=0%c%c" \
           "o=- 25678 753849 IN IP4 192.168.0.12%c%c" \
     "s=xxxSession %d  %sStartxxx%c%c" \
     "c=IN IP4 192.168.0.12%c%c" \
     "t=0 0%c%c" \
     "m=audio 3456 RTP/AVP 0%c%c" \
     "a=ptime:20%c%c",
           0xd, 0xa,
           0xd, 0xa,
           index, "Slow", 0xd, 0xa,
           0xd, 0xa,
           0xd, 0xa,
           0xd, 0xa,
           0xd, 0xa);

   if ((gc_util_insert_parm_ref_ex(&libBlock, IPSET_SDP, ip_parm, strlen(sdp_buf)+1, sdp_buf))
        != GC_SUCCESS) {
      printf("gc_util_insert_parm_ref_ex(IPSET_SDP, ip_parm=0x%x) failed: ", ip_parm);
   } else {
      printf("GC_APP : [%d] 3PCC SDP: body inserted (IPPARM=0x%x):\n%s\n\n",index,ip_parm,
              sdp_buf);
   }
   return libBlock;
}

static void sendUpdate(int callindex)
{
   printf("sendUpdate:: callindex = %d \n", port[0].crn);

   GC_PARM_BLKP   parmblkp=NULL;
   GC_PARM_BLKP   retblkp=NULL;

   gc_util_insert_parm_val(&parmblkp,
                           IPSET_MSG_SIP,
                           IPPARM_MSGTYPE,
                            sizeof(int),

   IP_MSGTYPE_SIP_UPDATE);
   if (gc_Extension(GCTGT_GCLIB_CRN, port[0].crn, IPEXTID_SENDMSG, parmblkp, &retblkp, EV_ASYNC)
       !=      GC_SUCCESS) {
      printf("\n <---- gc_GetCallInfo(crn=0x%lx) Failure - calling party not available \n",
              callindex);
      exit(1);
   } else {
       printf("Sent UPDATE successfully \n");
   }
}
```

4.18.2.2 Receiving an UPDATE Response

When the Global Call API library’s SIP stack receives a response to a SIP UPDATE request, it generates a GCEV_EXTENSION event of type IPEXTID_RECEIVEMSG.

The GC_PARM_BLK associated with the GCEV_EXTENSION event will contain a parameter element as follows:

IPSET_MSG_SIP
  IPPARM_MSGTYPE parameter ID

  And one of the following values:
  - IP_MSGTYPE_SIP_UPDATE_OK
  - IP_MSGTYPE_SIP_UPDATE_FAILED

The application may also retrieve the specific SIP response code from the event's parameter block using the IPSET_MSG_SIP set ID and the IPPARM_MSG_SIP_RESPONSE_CODE parameter ID.

The SDP information from the responses is passed to the application using the IPSET_SDP set ID and the IPPARM_SDP_UPDATE_OFFER/ IPPARM_SDP_UPDATE_ANSWER parameter ID.

If an IP_MSGTYPE_SIP_UPDATE_FAILED response is received, the application can retrieve the SIP header fields from the meta event. The application processes this using the **gc_GetMetaEvent( )** function, and then processes GC_PARM_BLK using the Global Call utility functions to retrieve the message type information and individual SIP header fields.

Example

```
case GCEV_EXTENSION:
{
   printf("\n ----> GCEV_EXTENSION \n");
   getExtensionInfo(metaeventp, 0);
}
break;

int getExtensionInfo(METAEVENT * metaeventp, int index)
{
   int retCode = 0;  // for success case

   GC_PARM_BLKP gcParmBlkp = NULL;
   GC_PARM_DATAP t_gcParmDatap = NULL;
   EXTENSIONEVTBLK *ext_evtblkp = NULL;
   GC_IE_BLK * t_gcIEBlk = NULL;
   char str[500];

   ext_evtblkp = (EXTENSIONEVTBLK *)metaeventp->extevtdatap;
   gcParmBlkp = &ext_evtblkp->parmblk;

   sprintf(str, "Received GCEV_EXTENSION event with ExtID = 0x%x", ext_evtblkp->ext_id);
   printf("%s \n", str);
   while(t_gcParmDatap = gc_util_next_parm(gcParmBlkp, t_gcParmDatap))
   {
      switch (t_gcParmDatap->set_ID)
      {
      case IPSET_SDP:
          printf("IPSET_SDP \n");

          switch(t_gcParmDatap->parm_ID)
          {
          case IPPARM_SDP_OFFER:
          case IPPARM_SDP_ANSWER:
                 printf("GC_APP : [%d] SDP received (IPPARM=0x%x):\n  \n\n", index,
                         t_gcParmDatap->parm_ID);
                 break;
```

```
                default:
                  printf("setID is IPSET_SDP \n");
                  break;
           }
           break;

           case IPSET_MSG_SIP:
              if (t_gcParmDatap->parm_ID == IPPARM_MSGTYPE)
              {
                 int l_mtype = (int)(*(t_gcParmDatap->value_buf));
                    if (l_mtype == IP_MSGTYPE_SIP_UPDATE_OK)
                    {
                       printf("GC_APP : Received 200OK for UPDATE \n");
                    } else if (l_mtype == IP_MSGTYPE_SIP_UPDATE_FAILED)
                    {
                        printf("GC_APP : Received 3XX ~ 5XX response for UPDATE \n");
                    } else {
                        printf("GC_APP : This is the default for IPPARM_MSGTYPE \n");
                    }
              }
           break;

           default:
              printf("This is default \n");
           break;
       }
    }
}
```

4.18.2.3 Receiving an UPDATE Request

When the Global Call API library is started with the E_SIP_UPATE_Access field set to ENUM_Enabled (IP_VIRTBOARD structure), the library generates a GCEV_CALLUPDATE event to the application when the SIP stack receives an incoming SIP UPDATE message. The application can extract standard message header fields from the parameter block associated with the GCEV_CALLUPDATE event.

The SDP information from the responses is passed to the application using the IPSET_SDP set ID and the IPPARM_SDP_UPDATE_OFFER/ IPPARM_SDP_UPDATE_ANSWER parameter ID.

Example

```
case GCEV_CALLUPDATE:
   printf("\n ---> GCEV_CALLUPDATE \n");
   printf("\n *** GCST_ACCEPTED State *** \n");
   pline->call_state = GCST_ACCEPTED;

   printf("\n Extract the SDP from GCEV_CALLUPDATE \n");
   if (get3PCCSDPInfo((GC_PARM_BLK *)metaeventp->extevtdatap, 0))
   {
      printf("ModifyMedia Already done ......\n");
   }
   // Send 200 OK for UPDATE
   printf("Responding to the UPDATE message with 200 OK.... \n");
   sendUpdate(pline->crn);

   break;
   int get3PCCSDPInfo(GC_PARM_BLK* pParmBlk, int index)
   {
      int retCode = 0;  // for success case
      printf("GC_APP : [%d] Looking for SDP...\n",index);
```

```
        if (pParmBlk)
        {
            GC_PARM_DATA_EXT ParmDataExt;

            //Initialize the structure to start from the first parm in the GC_PARM_BLK
            INIT_GC_PARM_DATA_EXT(&ParmDataExt);
            int UtilRet = gc_util_next_parm_ex(pParmBlk, &ParmDataExt);
            while (GC_SUCCESS == UtilRet) {

               if (ParmDataExt.set_ID == IPSET_SDP) {
                  switch (ParmDataExt.parm_ID)
                  {
                     case IPPARM_SDP_OFFER:
                     case IPPARM_SDP_ANSWER:

                     // What type of SDP do we expect?
                     retCode = 1;
                     printf("GC_APP : [%d] SDP received (IPPARM=0x%x):\n%s\n\n", index,
                         ParmDataExt.parm_ID, (char*)(ParmDataExt.pData));
                     return retCode;
                     break;

                  default:
                     printf("GC_APP : [%d] ERROR!!!  3PCC SDP received, invalid IPPARM!
                         (IPPARM=0x%x):\n%s\n\n", index, ParmDataExt.parm_ID, (char
                            *)(ParmDataExt.pData));
                     return retCode;
                     break;
               }
        }

UtilRet = gc_util_next_parm_ex(pParmBlk, &ParmDataExt);
      }
      printf("No ParmBlk for IPSET_SDP found \n");
   }
   else
   {
   printf("No ParmBlk for SDP found \n");
   }
   return retCode;
}
```

4.18.2.4 Sending an UPDATE Response

Once an application has received a GCEV_CALLUPDATE event for a SIP UPDATE message and extracted the information from the event, it sends a response message.

The response is sent by passing a GC_PARM_BLK structure containing the following parameter to the **gc_Extension( )** function:

IPSET_MSG_SIP
    IPPARM_MSGTYPE
    And one of the following parameter values:
    - IP_MSGTYPE_SIP_UPDATE_OK
    - IP_MSGTYPE_SIP_UPDATE_FAILED

The application may also set a specific SIP response code in the response message using the following parameter:

IPSET_MSG_SIP
    IPPARM_MSG_SIP_RESPONSE_CODE

And one of the following values:

- For an OK response, the values should be in the range 200 to 299. Default value is 200.
- For a Failed response, the value should be 300 or higher. Default value is 501.

Example

```
case GCEV_CALLUPDATE:
   printf("\n ---> GCEV_CALLUPDATE \n");
   printf("\n *** GCST_ACCEPTED State *** \n");
   pline->call_state = GCST_ACCEPTED;

   printf("\n Extract the SDP from GCEV_CALLUPDATE \n");
   if (get3PCCSDPInfo((GC_PARM_BLK *)metaeventp->extevtdatap, 0))
   {
      printf("ModifyMedia Already done ......\n");
   }
   // Send 200 OK for UPDATE
   printf("Responding to the UPDATE message with 200 OK.... \n");
   sendUpdateResponse(pline->crn);

   break;

static void sendUpdateResponse(int callindex)
{
    printf("sendUpdateResponse:: callindex = %d \n", port[0].crn);
    GC_PARM_BLKPparmblkp=NULL;
    GC_PARM_BLKPretblkp=NULL;

    gc_util_insert_parm_val(&parmblkp,
                        IPSET_MSG_SIP,
                        IPPARM_MSGTYPE,
    sizeof(int),
    IP_MSGTYPE_SIP_UPDATE_OK);

    gc_util_insert_parm_val(&parmblkp,
                  IPSET_MSG_SIP,
    IPPARM_MSG_SIP_RESPONSE_CODE,
    sizeof(int),
    200);

    if (gc_Extension(GCTGT_GCLIB_CRN, port[0].crn, IPEXTID_SENDMSG, parmblkp, &retblkp,
        EV_ASYNC) != GC_SUCCESS) {
        printf("\n <---- gc_GetCallInfo(crn=0x%lx) Failure - calling party not available \n",
               callindex);
      exit(1);
    } else {
      printf("Sent UPDATE successfully \n");
    }
}
```

4.18.3 Handling SIP UPDATE Post-Connection Messages

Dialogic® HMP Software supports handling SIP UPDATE post-connection requests with limited response capabilities in third-party call control (3PCC) mode.

Note: Receiving post-connection SIP UPDATE messages has limited functionality in that the application may only respond with OK or failure, but cannot change any session parameters. Use Re-INVITE if a change to session parameters is required.

This feature uses the Global Call Extension API mechanism for application notification and message construction as described in Section 4.18.1, “Enabling the Feature”, on page 267.

To enable handling UPDATE messages post connection, use the IPSET_CONFIG set ID and IPPARM_SEND_SIP_UPDATE_POSTCONNECTION parameter ID with the **gc_SetConfigData( )** function and Target Type GCTGT_CCLIB_NETIF.

This parameter ID can only be issued on a board device. It is defined as follows:

IPPARM_SEND_SIP_UPDATE_POSTCONNECTION
: Specifies whether to send the GCEV_CALLUPDATE event to the application. This event is set via **gc_util_insert_parm_val( )**. Disabled by default.
  - GCPV_ENABLE – Send GCEV_CALLUPDATE event to the application.

The application sends an UPDATE response post connection in the same manner as outlined in Section 4.18.2.4, “Sending an UPDATE Response”, on page 272; however, responses must follow the guidelines outlined in the SIP UPDATE RFC.

Examples

After opening the board device and receiving a GCEV_OPENEX event, the application calls the following on the board handle:

```
GC_PARM_BLKP parmblkp = NULL;
long request_id = 0;
gc_util_insert_parm_val(&parmblkp,IPSET_CONFIG,IPPARM_SEND_SIP_UPDATE_POSTCONNECTION,
                        sizeof(int),GCPV_ENABLE);
gc_SetConfigData(GCTGT_CCLIB_NETIF, bdev, parmblkp, 0,GCUPDATE_IMMEDIATE,
                 &request_id, EV_ASYNC);
gc_util_delete_parm_blk(parmblkp);
```

After the application receives a GCEV_CALLUPDATE event, it responds with a 200OK as follows:

```
gc_util_insert_parm_val(&parmblkp,IPSET_MSG_SIP,IPPARM_MSGTYPE,
                          sizeof(int) ,IP_MSGTYPE_SIP_UPDATE_OK);

gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->call[callindex].crn, parmblkp,
                 GC_NEXT_OUTBOUND_MSG);

// transmit the 200 OK message to network

gc_Extension(GCTGT_GCLIB_CRN, pline->call[callindex].crn, IPEXTID_SENDMSG,
               parmblkp, &retblkp, EV_ASYNC);

gc_util_delete_parm_blk(parmblkp);
```

4.19 Sending and Receiving SIP PRACK

The Provisional Response ACKnowledgement or PRACK method (RFC 3262) is supported in 3PCC mode. The application can manage reliable provisional response-related SIP messages.

This feature provides the reliable transmit of a provisional response in the public SIP network. As request/response protocol for initiating and managing communication sessions, SIP uses provisional and final responses. Final responses are sent reliably and convey the result of request processing. Provisional responses provide information on the progress of request processing, but are not sent reliably in RFC.

Notes: *1.* This feature is supported in 3PCC mode only.

2. Refer to RFC 3262 for the Offer/Answer Model with PRACK messages.

4.19.1 API Library Support

The following sections describe the Global Call API library support for SIP PRACK.

API Functions

The following Global Call API functions support the PRACK handling feature:

- **gc_SipPrack( )** sends a PRACK message.
- **gc_SipPrackResponse( )** sends a PRACK response message.

Events

The following events support PRACK handling:

GCEV_SIP_PRACK
: Indicates arrival of a SIP PRACK message.

GCEV_SIP_PRACK_RESPONSE
: Indicates arrival of a SIP PRACK response message.

Data Structure

The following field in the IP_VIRTBOARD data structure can be adjusted for each virtual board:

E_SIP_PrackEnabled
: Enables PRACK method. By default, PRACK is disabled. A value of ENUM_Enabled means that the application wants to enable the PRACK handling.

Parameter ID

The following parameter ID supports this feature:

IPPARM_SIP_PRACK_MANDATORY
: This parameter is used with IPSET_CALLINFO set ID, when PRACK handling is enabled. It specifies whether the application wants to make PRACK handling mandatory or optional for the outbound SIP call. It also indicates if the remote endpoint has made the PRACK handling mandatory or optional for the SIP inbound call.
  - IP_SIP_PRACK_MANDATORY_ON – When PRACK handling is enabled, this value indicates that PRACK handling is mandatory.
  - IP_SIP_PRACK_MANDATORY_OFF – When PRACK handling is enabled, this value indicates that PRACK handling is optional.

4.19.2 Enabling PRACK Handling

To enable a PRACK handling for SIP in 3PCC mode, the application sets the E_SIP_PrackEnabled field in IP_VIRTBOARD to ENUM_Enabled. The IP_VIRTBOARD structure is then passed to the **gc_Start( )** function. A setid and parmid of IPSET_CALLINFO and IPPARM_SIP_PRACK_MANDATORY are specified in the GCEV_OFFERED and GCEV_ALERTING events parm block.

A setid/parmid/value of IPSET_CALLINFO/IPPARM_SIP_PRACK_MANDATORY/IP_ SIP_PRACK_MANDATORY_ON indicates that the remote endpoint has made the PRACK handling mandatory.

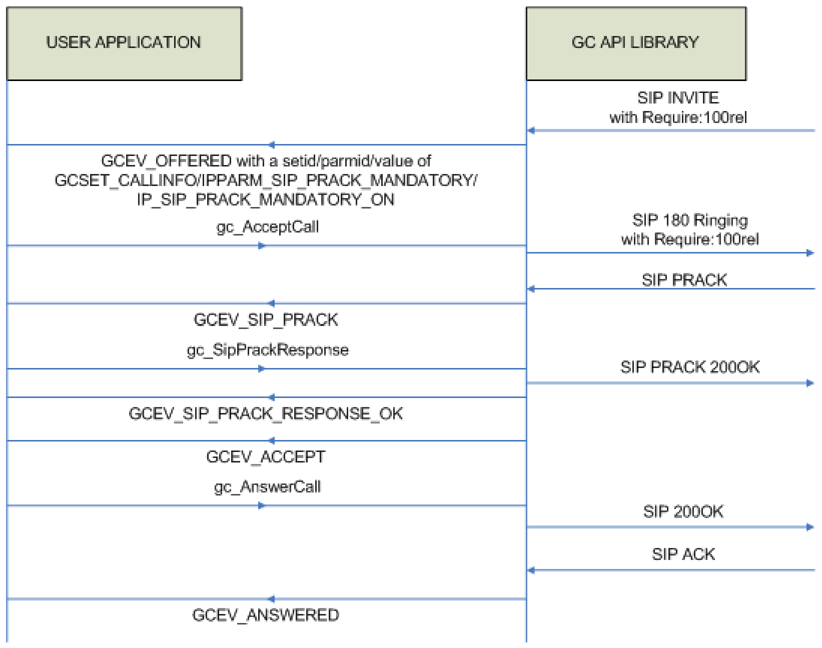
A setid/parmid/value of IPSET_CALLINFO/IPPARM_SIP_PRACK_MANDATORY/IP_ SIP_PRACK_MANDATORY_OFF indicates that the remote endpoint has made the PRACK handling optional.

When the setid/parmid combination is not present, then the remote endpoint is unable to handle PRACK messages. The call proceeds without any PRACK message exchanges.

4.19.2.1 PRACK Enabled and Remote Endpoint is Capable

If the application enables PRACK handling, and the remote endpoint is capable, the application automatically includes “Require:100rel” header in the outgoing provisional response, but excludes “100 Trying”. To send out the provisional response other than “100 Trying,” the application calls **gc_AcceptCall( )**. This causes the remote endpoint to respond with a PRACK message.

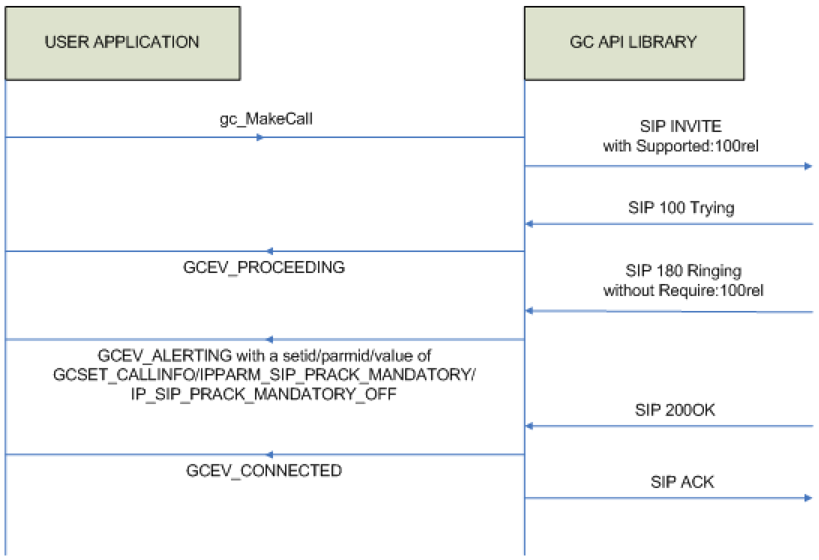
Upon receipt of a PRACK request, the application is notified with a GCEV_SIP_PRACK event. The application calls **gc_SipPrackResponse( )** to send out a PRACK response to the remote endpoint. If the PRACK response is successfully sent, a GCEV_SIP_PRACK_RESPONSE_OK is returned. A return of GCEV_SIP_PRACK_RESPONSE_FAILED indicates a failure in sending out the PRACK response.


USER APPLICATION
GC API LIBRARY
SIP INVITE with Require:100rel
GCEV_OFFERED with a setid/parmid/value of GCSET_CALLINFO/IPPARM_SIP_PRACK_MANDATORY/ IP_SIP_PRACK_MANDATORY_ON
gc_AcceptCall
SIP 180 Ringing with Require:100rel
SIP PRACK
GCEV_SIP_PRACK
gc_SipPrackResponse
SIP PRACK 200OK
GCEV_SIP_PRACK_RESPONSE_OK
GCEV_ACCEPT
gc_AnswerCall
SIP 200OK
SIP ACK
GCEV_ANSWERED

4.19.2.2 PRACK Enabled and Remote Endpoint is Optional

If PRACK handling is enabled for outbound calls, the API will automatically include "Supported:100rel" header in the outgoing INVITE message by default making PRACK handling optional for the remote end point.

The application can overwrite this default behavior on a per board basis using **gc_SetConfigData( )** or on a per call/channel basis using **gc_SetUserInfo( )** by changing the value of setid/parmid of IPSET_CALLINFO/IPPARM_SIP_PRACK_MANDATORY. A value of IP_SIP_PRACK_MANDATORY_ON will make the PRACK handling mandatory for the remote endpoint by putting "Require:100rel" header in an outgoing INVITE message and a value of IP_SIP_PRACK_MANDATORY_OFF will make the PRACK handling optional for the remote endpoint by putting "Supported:100rel" header in an outgoing INVITE message.

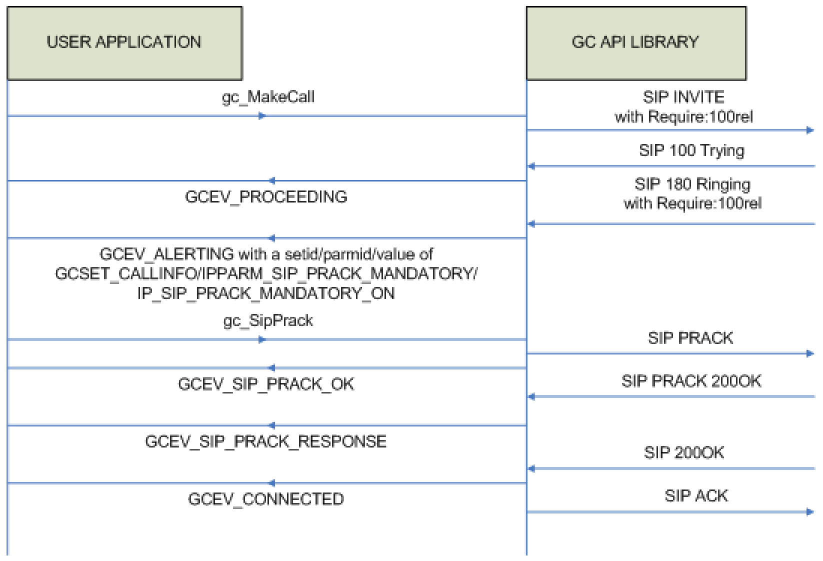

USER APPLICATION
GC API LIBRARY
gc_MakeCall
SIP INVITE with Supported:100rel
SIP 100 Trying
GCEV_PROCEEDING
SIP 180 Ringing without Require:100rel
GCEV_ALERTING with a setid/parmid/value of GCSET_CALLINFO/IPPARM_SIP_PRACK_MANDATORY/ IP_SIP_PRACK_MANDATORY_OFF
SIP 200OK
GCEV_CONNECTED
SIP ACK

4.19.2.3 PRACK Enabled and Remote Endpoint is Mandatory

Upon receiving a provisional response other than “100 Trying”, the API will generate a GCEV_ALERTING event to the application with a setid/parmid/value of IPSET_CALLINFO/IPPARM_SIP_PRACK_MANDATORY /IP_SIP_PRACK_MANDATORY_ON if “Require:100” header is present in the incoming provisional response (excluding “100 Trying”).

The application calls **gc_SipPrack( )** to send out PRACK for this provisional response. The application is notified with either a GCEV_SIP_PRACK_OK for successfully sending out a PRACK or with a GCEV_SIP_PRACK_FAILED in case of failure in sending out a PRACK.

When the remote endpoint responds with an ACK response, the API generates a GCEV_SIP_PRACK_RESPONSE to the application.


USER APPLICATION
GC API LIBRARY
gc_MakeCall
SIP INVITE with Require:100rel
SIP 100 Trying
GCEV_PROCEEDING
SIP 180 Ringing with Require:100rel
GCEV_ALERTING with a setid/parmid/value of GCSET_CALLINFO/IPPARM_SIP_PRACK_MANDATORY/ IP_SIP_PRACK_MANDATORY_ON
gc_SipPrack
SIP PRACK
GCEV_SIP_PRACK_OK
SIP PRACK 200OK
GCEV_SIP_PRACK_RESPONSE
SIP 200OK
GCEV_CONNECTED
SIP ACK

4.19.2.4 Remote Endpoint Does Not Support PRACK

When the application sends an INVITE with “Require:100rel” to a remote site that does not support PRACK, the **gc_MakeCall( )** function fails and a GCEV_DISCONNECTED event is returned. The reason stated in the event is a bad extension.

4.19.3 Examples

This section provides example code for PRACK handling.

Example 1

The following code snippet demonstrates how to enable PRACK handling.

```
CCLIB_START_STRUCT cclibStartStruct[] = {
{"GC_IPM_LIB", NULL},
"GC_H3R_LIB", &cclibStartData}
};
GC_START_STRUCT gcStartStruct;
IPCCLIB_START_DATA cclibStartData;
IP_VIRTBOARD virtBoards[1];

memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
memset(virtBoards,0,sizeof(IP_VIRTBOARD));

INIT_IPCCLIB_START_DATA(&ipcclibstart, 1, ip_virtboard);
INIT_IP_VIRTBOARD(&virtBoards[0]);

cclibStartData.num_boards = 1;
cclibStartData.board_list = virtBoards;
cclibStartData.version = 0x201;

//Set the mode as 3PCC. Prack handling can only be possible for 3PCC mode.
cclibStartData.media_operational_mode = MEDIA_OPERATIONAL_MODE_3PCC;

//Set the flag to enable the PRACK handling
virtBoards[0].E_SIP_PrackEnabled = ENUM_Enabled;

gcStartStruct.cclib_list = cclibStartStruct;
gcStartStruct.num_cclibs = GC_ALL_LIB;
gc_Start(&gcStartStruct);
```

Example 2

This code snippet shows whether the remote endpoint is interested in handling the PRACK.

```
void findoutIfRemoteIsInterestedInPRACK(METAEVENT metaevent,
GC_PARM_BLKP parm_blk)
{
  int remoteNotInterestedinPrackHandling = 1;

  switch (metaevent.evttype)
  {
       case GCEV_ALERTING:
       case GCEV_OFFERED:
         {
             GC_PARM_DATA_EXT parm;
             int rc;

             INIT_GC_PARM_DATA_EXT(&parm);

             rc = gc_util_next_parm_ex(parm_blk,&parm);
             if (rc != 0)
             {
                 printf("Remote endpoint is not interested in PRACK handling.\n");
                 return;
             }

             while (rc != EGC_NO_MORE_PARMS)
             {
               switch (parm.set_ID)
               {
                   case IPSET_CALLINFO:
                     if(parm.parm_ID == IPPARM_SIP_PRACK_MANDATORY)
                     {
                          if(*parm.pData == SIP_PRACK_MANDATORY_ON)
                          {
                             printf("Remote endpoint has made the PRACK handling mandatory.\n");
                             remoteNotInterestedinPrackHandling = 0;
```

```
                        }
                        else
                        if(*parm.pData==SIP_PRACK_MANDATORY_OFF)
                        {
                           printf("Remote endpoint has made the PRACK handling optional.\n");
                           remoteNotInterestedinPrackHandling = 0;
                        }
                   }
                   break;

                 default:
                   break;
           }
           rc = gc_util_next_parm_ex(parm_blk,&parm);
          }
     }
     break;

   default:
     return;
     break;
 }

 if(remoteNotInterestedinPrackHandling == 1)
 {
      printf("Remote endpoint is not interested in PRACK handling.\n");
 }
 return;
```

Example 3

This code snippet shows how to make PRACK handling mandatory or optional to the remote endpoint for outbound SIP.

```
int setPrackForOutboundCall(int mode /* 1 = mandatory, 0 = optional */)
{
   GC_PARM_BLK *parmblkp = NULL;

   if(mode == 1)
   {

        /* Mandatory PRACK mode */
        gc_util_insert_parm_val(&parmblkp, IPSET_CALLINFO,IPPARM_SIP_PRACK_MANDATORY,
                                sizeof(char), IP_SIP_PRACK_MANDATORY_ON);
    }
    else
    {

        /* Optional PRACK mode */
        gc_util_insert_parm_val(&parmblkp, IPSET_CALLINFO,IPPARM_SIP_PRACK_MANDATORY,
                                sizeof(char), IP_SIP_PRACK_MANDATORY_OFF);
    }

    if(gc_SetUserInfo(GCTGT_GCLIB_CHAN, port[index].ldev, parmblkp, GC_ALLCALLS) != GC_SUCCESS)
    {
       printf("gc_SetUserInfo(linedev=%ld) Failed configuring PRACK mode for outbound call",
               port[index].ldev);
            return(-1)
    }

    gc_util_delete_parm_blk(parmblkp);

    return(0);
}
```

4.19.4 Specifying Reliable or Non-reliable SIP Provisional Responses

The SIP PRACK feature described in Section 4.19, “Sending and Receiving SIP PRACK”, on page 274 is extended. The application can specify either Reliable or Non-Reliable SIP provisional response handling when receiving incoming INVITE messages that contain the Supported header with the optional tag 100rel. Previously, reliable response handling was the only option available to the application, regardless of the optional Supported header designation.

If the application has enabled PRACK handling and the remote is capable of PRACK handling, then Global Call automatically includes "Require:100rel" header in the outgoing provisional response except “100 Trying” for SIP inbound calls with Require:100rel header. For SIP inbound calls with Supported:100rel header, Global Call does not automatically include Require:100rel unless the application explicitly overrides it on a per call basis.

For SIP inbound calls with Supported:100rel header, the application will receive a GCEV_OFFERED with a GC PARM block with a setid, parmid, and value of IPSET_CALLINFO, IPPARM_SIP_PRACK_MANDATORY, IP_SIP_PRACK_MANDATORY_OFF. The application can overwrite the IPPARM_SIP_PRACK_MANDATORY value on a per call basis to IP_SIP_PRACK_MANDATORY_ON using the **gc_SetUserInfo( )** with SINGLECALL parameter. This will result in Global Call including a “Require:100rel” header in the outgoing provisional response except “100 Trying”.

If the application sets the IPPARM_SIP_PRACK_MANDATORY value to IP_SIP_PRACK_MANDATORY_OFF on a per call basis using the **gc_SetUserInfo( )** function, then the Global Call IP does not include “Supported:100rel” nor “Require:100rel” header in the outgoing provisional response.

Note: When PRACK handling is enabled, all response messages for incoming INVITE without a “100rel” header contain the “Supported:100rel”.

The following Global Call API functions support the PRACK handling feature:

- **gc_SipPrack( )** sends a PRACK message.
- **gc_SipPrackResponse( )** sends a PRACK response message.

Example

The following example demonstrates how to make PRACK handling mandatory or optional for inbound SIP:

```
Static void process_event(struct channel *pline, METAEVENT metaevent)
{
int eventtype;
int callindex;
eventtype = metaeventp->evttype;
callindex = metaeventp->crn;


switch (pline->call_state)
{
   case GCST_NULL:
```

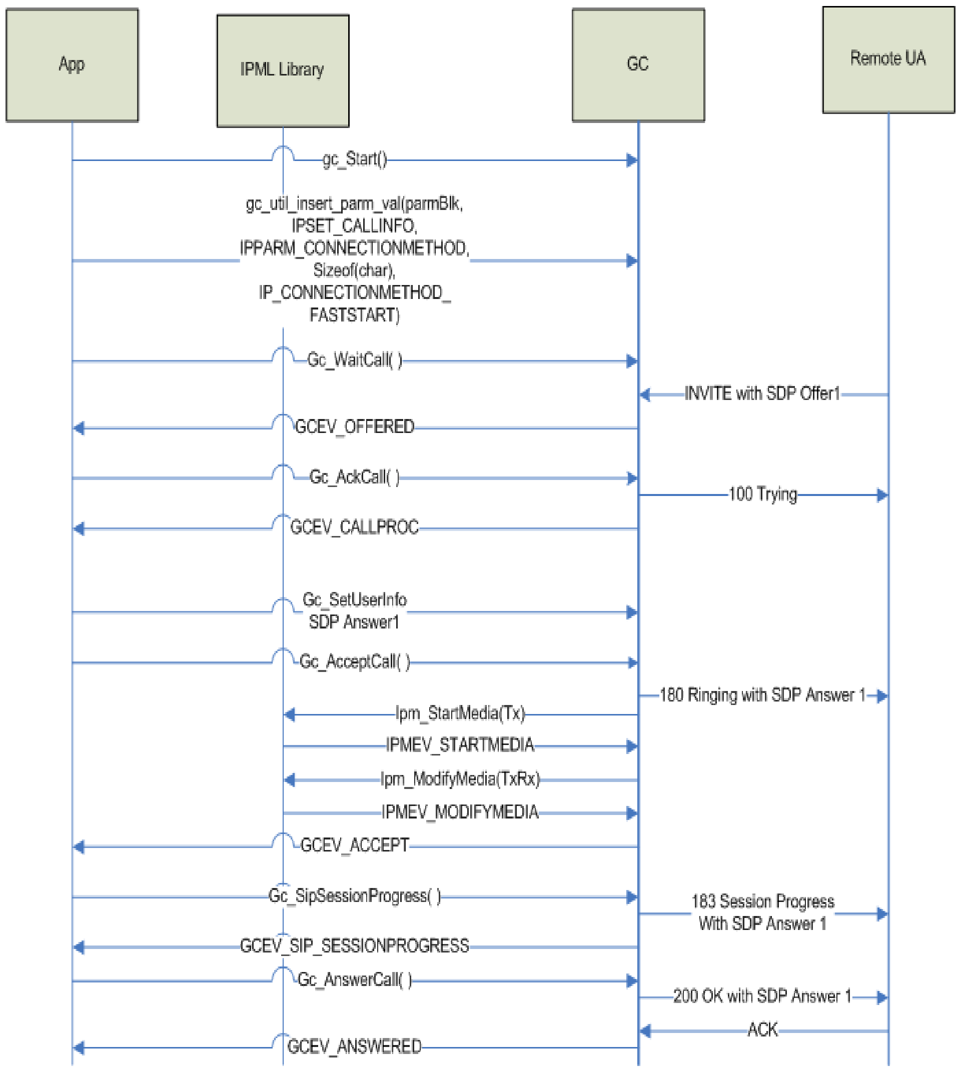
```
   {
     switch (eventtype)
    {
       case GCEV_OFFERED:
      {
         setPrackForInboundCall(1); // Mode=1 for mandatory PRACK
         ..
         ..
         gc_AcceptCall(   ); // or gc_CallAck()

       }
         break;
     }
     ..
     ..
     ..
     break;
    }
    ..
}


int setPrackForInboundCall(int mode /* 1 = mandatory, 0 = optional */)
{
    GC_PARM_BLK *parmblkp = NULL;
    if(mode == 1)
    {
       /* Mandatory PRACK mode */
       gc_util_insert_parm_val(&parmblkp, IPSET_CALLINFO,
          IPPARM_SIP_PRACK_MANDATORY, sizeof(char), IP_SIP_PRACK_MANDATORY_ON);
    }
    else
    {
        /* Optional PRACK mode */
        gc_util_insert_parm_val(&parmblkp, IPSET_CALLINFO,
           IPPARM_SIP_PRACK_MANDATORY, sizeof(char),IP_SIP_PRACK_MANDATORY_OFF);
    }

    if(gc_SetUserInfo(GCTGT_GCLIB_CHAN, port[index].ldev, parmblkp, GC_SINGLECALL) !=
                      GC_SUCCESS)
    {
        printf("gc_SetUserInfo(linedev=%ld) Failed configuring PRACK mode for inbound call",
                port[index].ldev);
        return(-1)
    }

    gc_util_delete_parm_blk(parmblkp);

    return(0);
}
```

4.20 Sending and Receiving SIP 183 Session Progress Messages

The application can transmit SIP 183 Session Progress responses using the **gc_SipSessionProgress( )** function. The application can also transmit both 180 Ringing and 183 Session Progress responses using a combination of **gc_AcceptCall( )** and **gc_SipSessionProgress( ).**

This feature is applicable when operating in 1PCC and 3PCC mode for SIP only.

The scenario outlined in the following figure shows an incoming INVITE with an SDP offer (fast start), followed by a subsequent outgoing 180 Ringing with SDP and 183 Session Progress response. This scenario is in 3PCC mode.



4.20.1 Receiving SIP 183 Session Progress Responses Containing SDP

The application can enable/disable the sending of an unsolicited event to the application upon receipt of a SIP 183 Session Progress informational response message. This feature can be enabled at the board level and on a per-call level.

This feature allows user applications to register for an unsolicited call status event upon receiving an incoming 183 Session Progress response. Both first party call control (1PCC) and third party call control (3PCC) modes are supported.

For 3PCC, access to the associated Session Description Protocol (SDP) answer information contained in the informational response message is also provided.

When operating in 3PCC mode, the application initiates early media by establishing a bidirectional path (transfer/receive) based on the media attributes offered by the remote endpoint as specified in the SDP answer.

Note: This feature applies to Fast Start only.

4.20.1.1 Enabling the Feature

Use the following parameter ID (and GCACT_SETMSK, GCACT_ADDMSK, GCACT_SUBMSK) with the GCSET_CALLEVENT_MSK set ID for this feature:

GCMSK_PROGRESS
: Enable or disable sending of GCEV_PROGRESSING event to the application upon receipt of 183 session response message.

Example: Enable the Feature at the Board Level

The following example illustrates how to enable the feature at the board level to support sending an unsolicited event (GCEV_PROGRESSING) to the application.

```
#include "gclib.h"
..
..

// To set on a board level
GC_PARM_BLK *pParmBlock = NULL;
long request_id = 0;

gc_util_insert_parm_val(&pParmBlock,
                        GCSET_CALLEVENT_MSK,
                        GCACT_SETMSK,
                        sizeof(long),
                        GCMSK_PROGRESS);

// Set config data
gc_SetConfigData(GCTGT_CCLIB_NETIF,
                 boarddev,
                 pParmBlock,
                 0,
                 GCUPDATE_IMMEDIATE,
                 &request_id,
                 EV_ASYNC);

gc_util_delete_parm_blk(pParmBlock);
```

Example: Enable the Feature at the Call Level

The following example illustrates enabling this feature at the call level.

```
#include "gclib.h"
..
..

GC_PARM_BLK *ParmBlock = NULL;

gc_util_insert_parm_val(&pParmBlock,
                        GCSET_CALLEVENT_MSK,
                        GCACT_SETMSK,
                        sizeof(long),
                        GCMSK_PROGRESS);

// Set config data
gc_SetUserInfo(GCTGT_GCLIB_CHAN,
               ldev,
               pParmBlock,
               GC_ALLCALLS);

gc_util_delete_parm_blk(pParmBlock);
```

4.20.1.2 Retrieving SDP Data from GCEV_PROGRESSING Event in 3PCC Mode

The GCEV_PROGRESSING event has an associated GC_PARM_BLK that contains extra data about the event. Depending on the list of header fields that the application has registered to receive, the GC_PARM_BLK associated with the GCEV_PROGRESSING event may contain multiple parameter elements that use the IPSET_SIP_MSG_INFO and IPPARM_SIP_HDR ID pair. The SDP data in the GCEV_PROGRESSING event can be retrieved using the IPSET_SDP and IPPARM_SDP_ANSWER ID pair.

The following example illustrates how the user application can retrieve the information from the unsolicited event in the 3PCC mode.

```
switch(metaeventp->evttype)
{
   case GCEV_PROGRESSING:
     printf("Received GCEV_PROGRESSING. Retrieve the SDP data \n");
     get3PCCSDPInfo(metaeventp->extevtdatap);
     break;

   default:
     printf("Unexpected event ……… %d", metaeventp->evttype);
     break;
}

int get3PCCSDPInfo(GC_PARM_BLK* pParmBlk)
{
   int retCode = 0;  // for success case
   printf(" Looking for 3PCC SDP...\n");

   if (pParmBlk) {
   GC_PARM_DATA_EXT ParmDataExt;

   //Initialize the structure to start from the 1st parm in the  GC_PARM_BLK

   INIT_GC_PARM_DATA_EXT(&ParmDataExt);
   int UtilRet = gc_util_next_parm_ex(pParmBlk, &ParmDataExt);

   while (GC_SUCCESS == UtilRet) {

     if (ParmDataExt.set_ID == IPSET_SDP) {
        switch (ParmDataExt.parm_ID)
        {
           case IPPARM_SDP_OFFER:
           case IPPARM_SDP_ANSWER:

          if (3PCCmode == 1) {
             printf(" 3PCC SDP received (IPPARM=0x%x):\n%s\n\n",
                            ParmDataExt.parm_ID,
                            (char *)(ParmDataExt.pData));
          } else if (3PCCmode > 1) {
              printf(" ERROR!!!  3PCC SDP received, but not expecting any!
                          (IPPARM=0x%x):\n%s\n\n",
                           ParmDataExt.parm_ID,
                           (char *)(ParmDataExt.pData));
              retCode = -1;
          } else {
               printf(" ERROR!!!  3PCC SDP received, but not in 3PCC mode!
                           (IPPARM=0x%x):\n%s\n\n",
                            ParmDataExt.parm_ID,
                            (char *)(ParmDataExt.pData));
               retCode = -1;
          }
          break;

          default:

             if (3PCCmode == 1) {
                 printf(" ERROR!!!  3PCC SDP received, invalid IPPARM!
                       (IPPARM=0x%x):\n%s\n\n",
                             ParmDataExt.parm_ID,
                             (char *)(ParmDataExt.pData));
             } else if (3PCCmode > 1) {
                 printf(" ERROR!!!  3PCC SDP received with invalid
                       IPPARM, but not expecting any!
                       (IPPARM=0x%x):\n%s\n\n",
                              ParmDataExt.parm_ID,
                              (char *)(ParmDataExt.pData));
                 retCode = -1;
             } else {
                 printf(" ERROR!!!  3PCC SDP received with invalid
                       IPPARM, but not even in 3PCC mode!
                       (IPPARM=0x%x):\n%s\n\n",
                               ParmDataExt.parm_ID,
                               (char *)(ParmDataExt.pData));
                 retCode = -1;
                 }
                 break;
        }
    }

    UtilRet = gc_util_next_parm_ex(pParmBlk, &ParmDataExt);

    }
  }
  return retCode;
}
```

4.21 Receiving Multiple SIP 18x Provisional Responses

Dialogic® HMP Software provides a method for obtaining subsequent provisional 18x SIP responses using the GCEV_EXTENSION event. When this feature is enabled, the first incoming 18x response generates a GCEV_ALERTING event as expected; however, all subsequent 18x responses are sent to the application by the GCEV_EXTENSION event. The block parameter associated with the event will contain SIP response headers in the same block format as in the GCEV_ALERTING event for the first provisional response received on the channel for the same transaction.

The following 18x provisional responses are supported:

- 180 Ringing
- 181 Call is Being Forwarded
- 182 Queued
- 183 Session Progress

A bitmask value, EXTENSIONEVT_SIP_18x_RESPONSE, is available in the IPSET_EXTENSIONEVT_MSK parameter set ID for feature enablement on a board-level basis using the **gc_SetConfigData( )** function with the GCTGT_CCLIB_NETIF target_type and the board-level device handle as the target ID. Once enabled, all IPT network devices (channels) have the ability to be notified of multiple18x provisional responses from the UAS.

An extension ID, IPEXTID_RECEIVED_18x_RESPONSE, is introduced so a GCEV_EXTENSION event for this feature can be differentiated from another feature. Once the GCEV_EXTENSION event is received, the application can easily find the reason by looking at the ext_id which is part of the EXTENSIONEVTBLK embedded in the event data pointer.

Furthermore, if the ext_id matches IPEXTID_RECEIVED_18x_RESPONSE, the 18x provisional response headers can be found in the same format as in the GCEV_ALERTING event for the first provisional response received on the channel in the same transaction. That is, the parmblkp associated with the GCEV_EXTENSION event contains the following mandatory set_ID and parm_ID:

- set_ID = IPSET_SIP_RESPONSE_CODE
- parm_ID = IPPARM_RECEIVED_RESPONSE_STATUS_CODE
- value = 3-digit integer representing the Status-Code from the response's Status-Line

If Reason-Phrase retrieval from 182 and 183 Provisional Responses has been enabled, then the parmblkp associated with the GCEV_EXTENSION event contains set_ID and parm_ID:

- set_ID = IPSET_SIP_MSGINFO
- parm_ID = IPPARM_SIP_HDR
- value = NULL-terminated string which begins with the string "Reason-Phrase"

Processing Responses

The following behavior occurs within the application when the feature is enabled.

The first incoming 18x response generates a GCEV_ALERTING as expected, or if a 183 response was received and the application enabled support for incoming 183 Session Progress Informational Response Containing SDP, a GCEV_PROGRESSING is received instead.

The GCEV_EXTENSION event indicates the receipt of any subsequent 18x responses within a transaction. The METAEVENT structure associated with it contains:

- EXTENSIONEVTBLK.ext_id = IPEXTID_RECEIVED_18x_RESPONSE
- EXTENSIONEVTBLK.parmblk has at least one (mandatory) entry with the following (reused from GCEV_ALERTING):
  set_ID = IPSET_SIP_RESPONSE_CODE
  parm_ID = IPPARM_RECEIVED_RESPONSE_STATUS_CODE
  value = 3-digit integer representing the Status-Code from Status-Line of the received provisional response

If Reason-Phrase retrieval from 182 and 183 Provisional Responses is enabled, the EXTENSIONEVTBLK.parmblk contains the following additional entry:

- set_ID = IPSET_SIP_MSGINFO
- parm_ID = IPPARM_SIP_HDR
- value = NULL-terminated string which begins with the string “Reason-Phrase” and contains the equivalent header from the response’s Status-Line

Note: This feature is activated on the board level; however, the GCEV_EXTENSION event is IPT network device (channel) specific.

4.21.1 Enabling and Disabling GCEV_EXTENSION

Enabling and disabling unsolicited GCEV_EXTENSION notification events is done by manipulating the event mask using the **gc_SetConfigData( )** function as described in Section 4.7.1, “Enabling and Disabling Unsolicited Notification Events”, on page 158.

The following example shows how to set the EXTENSIONEVT_SIP_18X_RESPONSE mask:

```
char *boardDeviceName = "N_iptB0:P_IP"; // example of board name for gc_openex()
LINEDEV boardDevice = 0; // IP channel board-level line device filled by gc_openex()

static void evt_hdlr()
{
...
        case GCEV_OPENEX:
        ...
        // board case
        {

                GC_PARM_BLK * pparm = NULL;
                long req_id;

                gc_util_insert_parm_val(pparm,
                        IPSET_EXTENSIONEVT_MSK,
                        GCACT_ADDMSK,
                        GC_VALUE_LONG,
                        EXTENSIONEVT_SIP_18X_RESPONSE);

                gc_setConfigData(GCTGT_CCLIB_NETIF,
                         boardDevice,
                         pparm,
                         0,
                         GCUPDATE_IMMEDIATE,
                            &req_id,
                            EV_ASYNC);
        }
...
}
```

4.21.2 Retrieving 18x Code from GCEV_EXTENSION

The following example demonstrates how to retrieve the Status-Code and Reason-Phrase headers (if available) from an 18x response mapped to a GCEV_EXTENSION unsolicited event once the feature is enabled. Refer to Section 4.4.3.2, "Retrieving Reason-Phrase from 182 and 183 Provisional Responses", on page 129 for information about setting up the application to receive Reason-Phrase header from these SIP responses.

```
case GCEV_EXTENSION:
GC_PARM_BLKP pParmBlock;
EXTENSIONEVTBLK *pextensionBlk;
GC_PARM_DATAP l_pParmData;
pextensionBlk = (EXTENSIONEVTBLK *)(m_pMetaEvent->extevtdatap);
pParmBlock = (&(pextensionBlk->parmblk));

if(pextensionBlk->ext_id== IPEXTID_RECEIVED_18X_RESPONSE)
{
      while ((l_pParm = gc_util_next_parm(pParmBlock, l_pParm)) != 0)
      {
             int l_mtype= (int)(*( l_pParm->value_buf));
             switch (l_pParm->set_ID)
             {
             case IPSET_SIP_RESPONSE_CODE:
                    if (l_pParm->parm_ID == IPPARM_RECEIVED_RESPONSE_STATUS_CODE)
                    {

                          if (l_pParmData->value_size != 0)
                          {
                                 int code_18x = *(int *) l_pParm ->value_buf;
                                 //...
                          }
                     }
                     break;
              case IPSET_SIP_MSGINFO:
                     if (l_pParm->parm_ID == IPPARM_SIP_HDR)
                     {
                           if (l_pParmData->value_size != 0)
                           {
                                  char siphdr_18x[IP_SIP_HDR_MAXLEN];

                                  strncpy(siphdr_18x,(char*)parmp->value_buf,parmp->value_size);

                                  siphdr_18x[parmp->value_size]='\0';
                                  //...
                            }
                     }
                     break;
              default:
                     //... warning no other type allowed
                     break;
              }
         }
}
else
{
        // other ext_id
}
break;

... other GCEV_ case
```

4.22 Defer the Sending of SIP Messages

The application can delay sending the appropriate response to an incoming BYE request (such as 200OK), as well as delay the sending of a BYE request until the **gc_ReleaseCallEx( )** function is issued on the channel.

Call termination is accomplished using **gc_DropCall( )** and **gc_ReleaseCallEx( )**. With the SIP protocol, the **gc_DropCall( )** function handles signaling messages by default, while the **gc_ReleaseCallEx( )** function is used to free up the resource. For instance, if call termination was initiated by the remote side (the remote side sent the BYE request), then the **gc_DropCall( )** function will send out a 200OK response. If the call termination was initiated locally, the **gc_DropCall( )** function will send out the BYE request.

When a call is terminated using **gc_DropCall( )**, an incoming call could be received before the application has a chance to invoke **gc_ReleaseCallEx( )** to release resources. In situations where there is a lot of call traffic, the incoming call is rejected because even though the last call was dropped, channels are not yet available since the resource has not been released.

By enabling this feature, the **gc_ReleaseCallEx( )** function is responsible for the sending of SIP messages in addition to the freeing of resources. This allows the application to defer sending the appropriate response to an incoming BYE or CANCEL request or to the BYE and CANCEL requests themselves, or to DECLINE the incoming INVITE, depending on the call state.

Notes: ***1.*** Once enabled, this feature applies to the entire virtual board.

2. The **gc_ReleaseCallEx( )** function is the preferred equivalent to the deprecated **gc_ReleaseCall( )** function.

4.22.1 Deferring SIP Signaling

When **gc_DropCall( )** or **gc_ReleaseCallEx( )** is issued, the Global Call library sends a SIP message (BYE-Decline/CANCEL/200OK) depending on the state of the call. The following table describes these call states and the behavior of the Global Call library when the SIP signaling deferral feature is enabled or disabled. To enable this feature, see Section 4.22.2, “Enabling the Deferral of SIP Signaling”, on page 292.

Table 11. Deferral of SIP Signaling Messages - Call States and Behavior

Call State	Feature is disabled	Feature is enabled
Connected	When **gc_Dropcall( )** is issued to initiate the call termination, the BYE message is sent. The GCEV_DROPCALL event is generated once the 200OK response is received.	When **gc_DropCall( )** is issued, the media session is terminated and the GCEV_DROPCALL event is generated. When **gc_ReleaseCallEx( )** is issued, the BYE message is sent. The GCEV_RELEASECALL event is generated when the 200OK response is received.
Disconnected	After the BYE request is received in the connected state, GCEV_DISCONNECTED is sent to the application and the state transitions to the Disconnected state. The **gc_DropCall( )** function sends the 200OK response and the GCEV_DROPCALL event is generated immediately.	After the BYE is received in the connected state, **gc_ReleaseCallEx( )** sends the 200OK final response. The GCEV_RELEASECALL event is generated immediately
Offered or Accepted	When **gc_DropCall( )** is issued in one of these states, the "603-Decline" message is sent to reject the incoming call. The GCEV_DROPCALL event is generated once the ACK is received.	When **gc_ReleaseCallEx( )** is issued in the one of these states, the "603-Decline" message is sent to reject the incoming call. The GCEV_RELEASECALL event is generated when the ACK is received.
Dialing	When **gc_DropCall( )** is issued to initiate the call termination after **gc_Makecall( )** is invoked, the CANCEL message is sent to cancel the request to start a dialog.	When **gc_ReleaseCallEx( )** is issued to initiate the call termination after **gc_Makecall( )** is invoked, the CANCEL message is sent to cancel the request to start a dialog.

4.22.2 Enabling the Deferral of SIP Signaling

To defer sending SIP messages until **gc_ReleaseCallEx**( ) is issued, enable this feature for all channels using the **gc_SetConfigData**( ) function.

Use IPSET_CONFIG set ID and IPPARM_SIGNALING_DEFERRED parameter ID to dynamically enable or disable this feature on a virtual board basis. By default, this feature is disabled. This parameter ID is inserted in the GC_PARM_BLKP data structure using **gc_insert_parm_ref**( ).

The supported values for IPPARM_SIGNALING_DEFERRED are as follows:

- GCPV_ENABLE – The signaling messages (BYE/ Decline/CANCEL/200OK, etc) are not sent when **gc_DropCall**( ) is issued. They are deferred to **gc_ReleaseCallEx**( ).
- GCPV_DISABLE – Default. The signaling messages (BYE/Decline/CANCEL/200OK) are sent when the **gc_DropCall**( ) is issued.

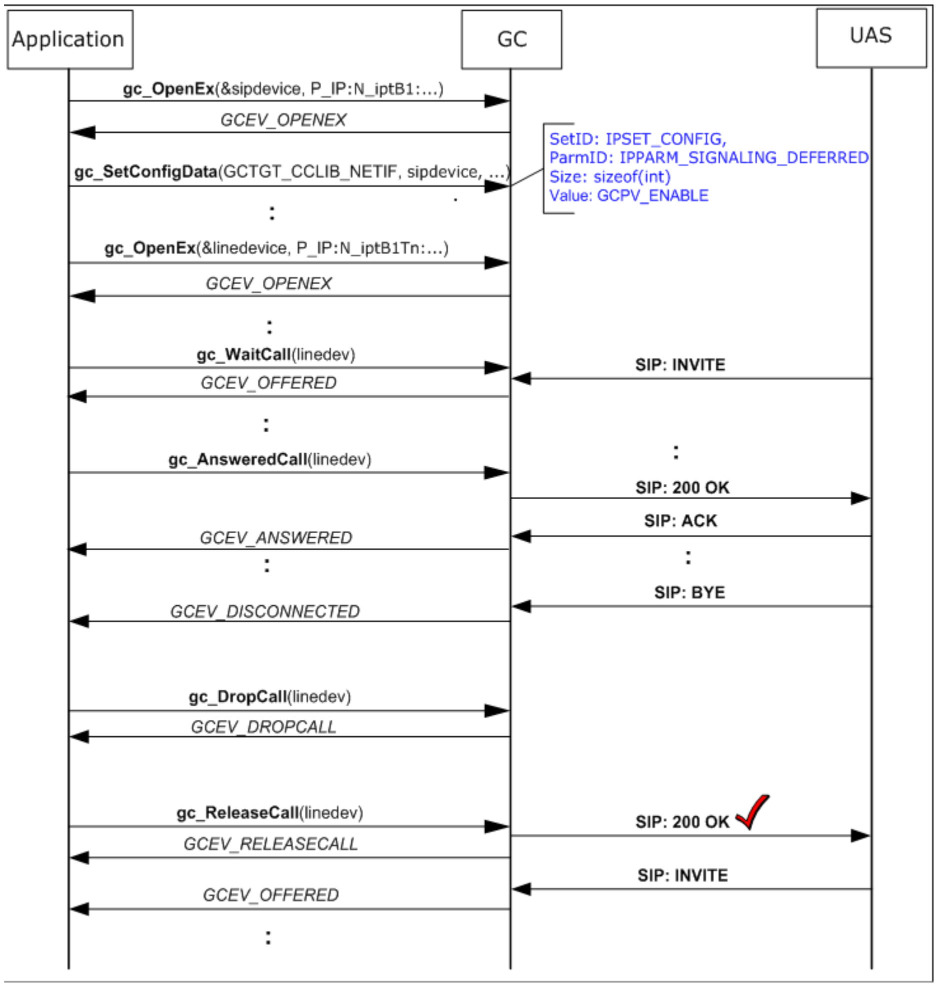
Example

This example demonstrates how to enable the deferral of sending SIP messages during call termination.

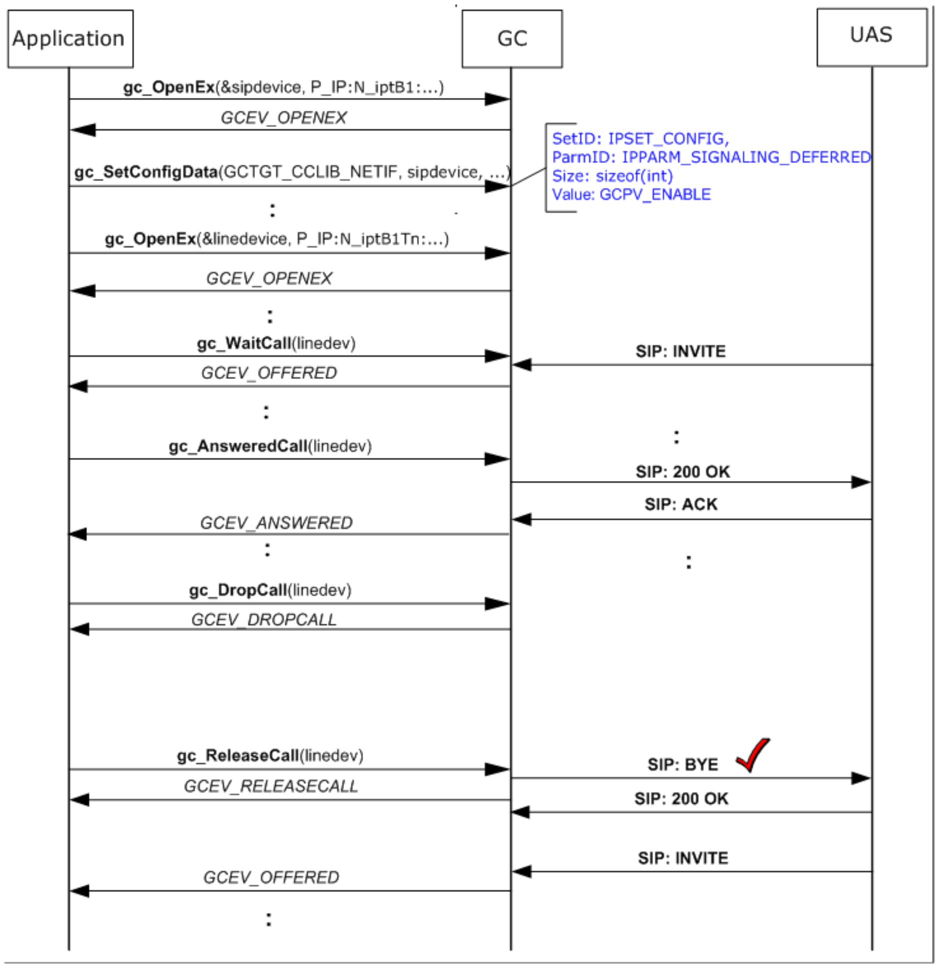
```
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>
#include <gcip_defs.h>

/*
 * Assume the board device iptB1 has been opened
 */

int defer_signaling()
{
        GC_PARM_BLKP parmblkp = NULL;   /* input parameter block pointer */
        long request_id = 0;
        gc_util_insert_parm_val(&parmblkp,
                IPSET_CONFIG,
                IPPARM_SIGNALING_DEFERRED,
                sizeof(int),
                GCPV_ENABLE);

        if (gc_SetConfigData(GCTGT_CCLIB_NETIF, boardDev, parmblkp, 0 /*timeout*/,
            GCUPDATE_IMMEDIATE, &request_id, EV_ASYNC) != GC_SUCCESS)
        {
            // handle error…
        }

        return (0);
}
```

4.22.3 High-level Scenarios

The following diagrams illustrate deferrals of a 200OK response and a BYE request, respectively. Other possible scenarios under different call states are not shown.



The following scenario demonstrates a BYE request.



4.23 Service Uniform Resource Names (URNs)

SIP Uniform Resource Names (URNs) can be used when contacting a Service Namespace Identifier.

Note: Service in this context refers to URNs for Emergency and other well-known services.

According to IETF RFC 3406, SIP URNs are “resource namespace identifiers with the specific requirements for enabling location independent identification of a resource, as well as longevity of a reference. URNs are part of the larger Uniform Resource Identifier (URI) family [RFC 3305], with the specific goal of providing persistent naming of resources.”

A service URN identifies a logical service, returning a particular instance of the service, and this instance may be different even for two users making the same request in the same place at the same time. The logical service identified by the URN, however, is persistent and unique. For example:

```
urn:service:sos (specifies an emergency service URN)
```

URN support is limited to HMP acting as the UAC initiating a SIP call with that resource identifier, when in conjunction with an outbound SIP proxy server. Since Service URNs are identifiers that carry no information about the final destination of the actual call, call routing decisions must be delegated to third party entities.

With this feature, the application has the ability to set the Request-URI header in an INVITE out of HMP to a Service URN identifier, using the standard **gc_MakeCall( )** API set. If the specific Service URN identifier has meaning in the local area of the call, then the outbound proxy server should appropriately route the call to the organization that manages the Service in that local area.

Notes: ***1.*** The prefix strings "urn:" or "URN:" are valid identifiers for a destination; HMP will route to the proxy server, which must be configured in HMP for the feature to take effect.

2. No provisions are made as to whether the URN namespace is registered with the Internet Assigned Numbers Authority (IANA), or whether it is of “service” or any other namespace identifier.

3. A Service URN may contain a hierarchy of subservices that further describe it with a sequence of labels separated by periods (e.g., urn:service:sos.ambulance).

For additional information about the SIP Proxy Server and the SIP Transport Protocol, refer to Section 4.33, “Dynamic Selection of Outbound SIP Proxy”, on page 366.

4.23.1 Usage Scenario

Following is a usage scenario for generating an Emergency Service URN outgoing call.

- The application specifies the SIP proxy addressing information using the **gc_Start( )** function and the IP_VIRTBOARD structure.

  Note: Alternatively, the application can select an outbound SIP proxy server on the virtual board device dynamically; see Section 4.33, “Dynamic Selection of Outbound SIP Proxy”, on page 366

- The application calls the **gc_MakeCall( )** function to a service URN, e.g., `urn:service:sos.animal-control` as the destination in the GCLIB_MAKECALL_BLK.
- Global Call will use the service URN string as entry in the Request-URI and To Header URI; however, the INVITE will go to the setup proxy server using the Via Header.
- The proxy server will route the call to the appropriate SIP URL.

Code Example

```
void MakeCall(struct channel *pline)
{
       char *pDestAddrBlk = "urn:service:sos\0";
       char *pSrcAddrBlk = "joe@dialogic.com\0";

       pline->gcmkblk.cclib = NULL; /* CCLIB pointer unused */
       pline->gcmkblk.gclib = &pline->gclib_mkbl; //
       memset(&pline->gclib_mkbl,0,sizeof(GCLIB_MAKECALL_BLK));

       /* set GCLIB_ADDRESS_BLK with destination string & type*/
       strcpy(pline->gcmkblk.gclib->destination.address,pDestAddrBlk);
       pline->gcmkblk.gclib->destination.address_type = GCADDRTYPE_TRANSPARENT;

       /* set GCLIB_ADDRESS_BLK with origination string & type*/
       strcpy(pline->gcmkblk.gclib-> origination.address, pSrcAddrBlk);
       pline->gcmkblk.gclib-> origination.address_type = GCADDRTYPE_TRANSPARENT;

       gc_MakeCall(pline->ldev, &crn, NULL, &pline->gcmkblk, MakeCallTimeout,EV_ASYNC);
}
```

4.24 Sending and Receiving DTMF

DTMF handling is described in the following topics:

- Specifying DTMF Support
- Getting Notification of DTMF Detection
- Generating DTMF
- Generating or Detecting DTMF Tones Using a Voice Resource
- Setting Receive-only RFC 2833 Mode
- Retrieving SIP Inbound DTMF
- Exposing Raw RFC 2833 Data

4.24.1 Specifying DTMF Support

The Dialogic® Global Call API can be used to configure which DTMF transmission modes are supported by the application. The DTMF mode can be specified in one of three ways:

- for all line devices simultaneously by using **gc_SetConfigData( )**
- on a per-line device basis by using **gc_SetUserInfo( )** with a **duration** parameter value of GC_ALLCALLS
- on a per-call basis by using **gc_SetUserInfo( )** with a **duration** parameter value of GC_SINGLECALL

The GC_PARM_BLK associated with the **gc_SetConfigData( )** or **gc_SetUserInfo( )** function is used to indicate which DTMF modes are supported. The GC_PARM_BLK should include the following parameter element

IPSET_DTMF
  IPPARM_SUPPORT_DTMF_BITMASK
  - value = a single bitmask value or the OR of more than one value to specify multiple supported DTMF transmission modes

Note: The IPPARM_SUPPORT_DTMF_BITMASK parameter can only be replaced rather than modified. For each **gc_SetConfigData( )** or **gc_SetUserInfo( )** call, the previous value of the IPPARM_SUPPORT_DTMF_BITMASK parameter is overwritten.

Bitmask values for SIP

SIP applications **must** set the DTMF signaling mode before calling **gc_MakeCall( )**, **gc_AnswerCall( )**, **gc_AcceptCall( )**, or **gc_CallAck( )**. If a SIP application does not do this, the function call fails with an IPERR_NO_DTMF_CAPABILITY indication. Supported bitmask values are:

IP_DTMF_TYPE_INBAND_RTP
  DTMF digits are sent and received inband via standard RTP transcoding.
  Note: Inband mode cannot be used when using low bit-rate (LBR) coders.

IP_DTMF_TYPE_RFC_2833
  DTMF digits are sent and received in the RTP stream as defined in RFC 2833.

Bitmask values for H.323

An H.323 application that supports only the default H.245 User Input Indication (UII) Alphanumeric mode does not need to explicitly set the DTMF signaling mode. All other applications must set the DTMF mode using the following bitmask values:

IP_DTMF_TYPE_ALPHANUMERIC (default)
  DTMF digits are sent and received in H.245 UII Alphanumeric messages.
  Note: HMP only supports the H.245 UII Alphanumeric mode; H.245 UII Signal mode is **not** supported.

IP_DTMF_TYPE_INBAND_RTP
  DTMF digits are sent and received inband via standard RTP transcoding.
  Note: Inband mode cannot be used when using low bit-rate (LBR) coders.

IP_DTMF_TYPE_RFC_2833
  DTMF digits are sent and received in the RTP stream as defined in RFC 2833.

As an example, the following code snippet shows how to specify the out-of-band signaling mode for all calls on a line device:

```
{
   GC_PARM_BLKP parmblkp = NULL;
   gc_util_insert_parm_val(&parmblkp,
                           IPSET_DTMF,
                           IPPARM_SUPPORT_DTMF_BITMASK,
                           sizeof(char),
                           IP_DTMF_TYPE_INBAND_RTP);

   if (gc_SetUserInfo(GCTGT_GCLIB_CHAN, port[callindex].ldev,
                      parmblkp, GC_ALLCALLS) != GC_SUCCESS) {

          // gc_SetUserInfo returned an error
}
gc_util_delete_parm_blk(parmblkp);
```

The mode in which DTMF is transmitted (Tx) is determined by the intersection of the mode values specified by the IPPARM_SUPPORT_DTMF_BITMASK and the receive capabilities of the remote endpoint. When this intersection includes multiple modes, the selected mode is based on the following priority:

1. RFC 2833
2. H.245 UII Alphanumeric (H.323 only)
3. Inband

The mode in which DTMF is received (Rx) is based on the selection of transmission mode from the remote endpoint; however, RFC 2833 can only be received if RFC 2833 is specified by the IPPARM_SUPPORT_DTMF_BITMASK parameter ID.

Table 12 summarizes the DTMF mode settings and associated behavior.

Table 12. Summary of DTMF Mode Settings and Behavior

IP_DTMF_TYPE_ RFC_2833	IP_DTMF_TYPE_ ALPHANUMERIC†	IP_DTMF_TYPE_ INBAND	Transmit (Tx) DTMF Mode	Receive (Rx) DTMF Mode
1 (enabled)	0 (disabled)	0 (disabled)	RFC 2833 if supported by remote endpoint, otherwise UII Alphanumeric†	RFC 2833, UII Alphanumeric† or Inband as chosen by the remote endpoint
0 (disabled)	1 (enabled)	0 (disabled)	UII Alphanumeric†	UII Alphanumeric† or Inband as chosen by the remote endpoint
0 (disabled)	0 (disabled)	1 (enabled)	Inband	UII Alphanumeric† or Inband as chosen by the remote endpoint
0 (disabled)	1 (enabled)	1 (enabled)	UII Alphanumeric†	UII Alphanumeric† or Inband as chosen by the remote endpoint
† Applies to H.323 only.				

Table 12. Summary of DTMF Mode Settings and Behavior (Continued)

IP_DTMF_TYPE_ RFC_2833	IP_DTMF_TYPE_ ALPHANUMERIC†	IP_DTMF_TYPE_ INBAND	Transmit (Tx) DTMF Mode	Receive (Rx) DTMF Mode
1 (enabled)	0 (disabled)	0 (disabled)	RFC 2833 if supported by remote endpoint, otherwise UII Alphanumeric†	RFC 2833, UII Alphanumeric† or Inband as chosen by the remote endpoint
1 (enabled)	1 (enabled)	0 (disabled)	RFC 2833 if supported by remote endpoint, otherwise UII Alphanumeric†	RFC 2833, UII Alphanumeric† or Inband as chosen by the remote endpoint
1 (enabled)	0 (disabled)	1 (enabled)	RFC 2833 if supported by the remote endpoint, otherwise Inband	RFC 2833, UII Alphanumeric† or Inband as chosen by the remote endpoint
1 (enabled)	1 (enabled)	1 (enabled)	RFC 2833 if supported by the remote endpoint, otherwise UII Alphanumeric†	RFC 2833, UII Alphanumeric† or Inband as chosen by the remote endpoint
† Applies to H.323 only.				

When using RFC 2833, the payload type is specified using the following parameter element:

IPSET_DTMF
  IPPARM_DTMF_RFC2833_PAYLOAD_TYP
  and one of the following values:
  - IP_USE_STANDARD_PAYLOADTYPE – default payload type (101)
  - any value in the range 96 to 127 – dynamic payload type

4.24.2 Getting Notification of DTMF Detection

Once DTMF support has been configured (see Section 4.24.1, “Specifying DTMF Support”, on page 297), the application can specify which DTMF modes will provide notification when DTMF digits are detected. The events for this notification must be enabled; see Section 4.7.1, “Enabling and Disabling Unsolicited Notification Events”, on page 158.

4.24.2.1 Processing H.245 UII Alphanumeric Events

Once the events are enabled, when an incoming DTMF digit is detected, the application receives a GCEV_EXTENSION event, with an extID of IPEXTID_RECEIVE_DTMF. The GCEV_EXTENSION event contains the digit and the method. The GC_PARM_BLK associated with the event contains the IPSET_DTMF parameter set ID and the following parameter ID:

IPPARM_DTMF_ALPHANUMERIC
  For H.323, DTMF digits are received in H.245 User Input Indication (UII) alphanumeric messages. The parameter value is a data structure of type IP_DTMF_DIGITS (it is **not** a

string). See the reference page for IP_DTMF_DIGITS on page 657 for more information. For SIP, this parameter is **not** supported.

4.24.2.2 Processing RFC 2833 Events

Once DTMF support has been configured as RFC 2833 (see Section 4.24.1, “Specifying DTMF Support”, on page 297), and GCEV_TELEPHONY_EVENT notification is enabled (see Section 4.7.1, “Enabling and Disabling Unsolicited Notification Events”, on page 158), a GCEV_TELEPHONY_EVENT is generated for each RFC 2833 DTMF digit received.

The GCEV_TELEPHONY_EVENT delivers raw RFC 2833 information in the form of IPM_TELEPHONY_INFO structure for each DTMF digit received (see the *Dialogic® IP Media Library API Programming Guide and Library Reference* for the structure definition). The IPM_TELEPHONY_INFO structure contains the volume, duration and event ID of the digit.

Raw RFC 2833 DTMF digit information can be retrieved from GCEV_TELEPHONY_EVENT using the Set ID and Parameter ID:

IPSET_DTMF
    IPPARM_TELEPHONY_EVENT_INFO
    - value = IPM_TELEPHONY_INFO

Example of retreiving raw RFC 2833 information about the received DTMF digit from the GCEV_TELEPHONY_EVENT event:

```
#include "gcip.h"
#include "gclib.h"
#include "gcip_defs.h"
#include "ipmlib.h"

void process_event(void)
{
      METAEVENT metaevent;
      GC_PARM_BLKP my_blkp = NULL;
      GC_PARM_DATAP my_datap;
      IPM_TELEPHONY_INFO my_telInfo;
      IPM_TELEPHONY_EVENT_INFOmy_telEvtInfo;

      if(gc_GetMetaEvent(&metaevent) != GC_SUCCESS)
      {
      /* Process error */
      }

switch(metaevent.evttype)
{
       .
       .
       .
       case GCEV_TELEPHONY_EVENT:
            /* Make a copy of the parm blk */
            if(metaevent.extevtdatap)
            {
                if ( gc_util_copy_parm_blk( &my_blkp,
               (GC_PARM_BLKP)(metaevent.extevtdatap))!= GC_SUCCESS)
                  {
                         /* Process error */
                  }

            my_datap = gc_util_find_parm(my_blkp, IPSET_DTMF,
                                    IPPARM_TELEPHONY_EVENT_INFO);

                                    if (my_datap != NULL)
                                    {
                                        memcpy(&my_telInfo, my_datap->value_buf, my_datap->value_size);
                                    }

                                    switch(my_telInfo.eTelInfoType)
                                      {
                                          case TEL_INFOTYPE_EVENT:

                                                my_telEvtInfo = my_telinfo.TelephonyInfo.TelEvtInfo;
                                                printf("GCEV_TELEPHONY_EVENT received dmtf Information:
                                                TelephonyEventID=%d, Volume=%d, Duration=%d",
                                                my_telEvtInfo.eTelephonyEventID,
                                                my_telEvtInfo.sVolume,
                                                my_telEvtInfo.usDuration);
                                          break;
                                      default:
                                          /* Print Error */
                                      }

                                    }
                                    else
                                    {
                                         /* Process error */
                                    }
                                    break;
                                .
                                .
                                .
                            }
                      }
```

4.24.3 Generating DTMF

Once DTMF support has been configured (see Section 4.24.1, “Specifying DTMF Support”, on page 297), the application can use the **gc_Extension( )** function to generate DTMF digits. The relevant **gc_Extension( )** function parameter values in this context are:

- **target_type** should be GCTGT_GCLIB_CRN
- **target_id** should be the actual CRN
- **ext_ID** should be IPEXTID_SEND_DTMF

The GC_PARM_BLK pointed to by the **parmblkp** parameter must contain the IPSET_DTMF parameter set ID and the following parameter ID:

IPPARM_DTMF_ALPHANUMERIC
: For H.323, specifies that DTMF digits are to be sent in H.245 User Input Indication (UII) Alphanumeric messages. For SIP, this parameter is **not** supported.

4.24.4 Generating or Detecting DTMF Tones Using a Voice Resource

Using a voice resource to generate or detect DTMF tones in Inband or RFC2833 DTMF transfer mode requires that the voice resource (for example, dxxxB1C1) be attached to the IPT network device (for example, iptB1T1) that also has an IP Media device (ipmB1C1) attached. This can be achieved using the **gc_OpenEx( )** function as follows:

```
gc_OpenEx(lindevice, ":P_IP:N_iptB1T1:M_ipmB1C1:V_dxxxB1C1", EV_ASYNC, userattr)
```

where:

- linedevice is a Global Call device
- P_IP indicates that the device supports both the H.323 and SIP protocols
- N_iptB1T1 identifies the IPT network device
- M_ipmB1C1 identifies the IPT Media device
- V_dxxxB1C1 specifies the voice resource that will be used to generate or detect the DTMF tones
- EV_ASYNC indicates the function operates in asynchronous mode
- userattr points to a buffer where user information can be stored

Note: Alternatively, the IPT network device and IP Media device can be opened without the voice resource, and the IP line device can be routed to the voice device when needed.

Once the voice resource is attached to the IPT network and IPT Media devices, the following voice library functions can be used:

- **dx_dial( )** to generate DTMF tones
- **dx_getdig( )** to detect DTMF tones

4.24.5 Setting Receive-only RFC 2833 Mode

In full-duplex RCF 2833 mode, the mechanism used to detect and remove in-band DTMF digits from the audio stream prior to the transmission of audio RTP packets contributes to audio latency. In receive-only RFC 2833 mode, this additional transmit audio latency is reduced.

For the third-party call control model (3PCC), select this mode of operation at runtime using Dialogic® IP Media Library functions. See the *Dialogic® IP Media Library API Programming Guide and Library Reference* for more information.

For the first-party call control model (1PCC), select this mode using the parameter element IPSET_DTMF in GC_PARM_BLK, which is associated with **gc_SetUserInfo( )** and **gc_SetConfigData( )**.

The following example demonstrates how to set PARMCH_DISABLE_TX_TELEPHONY_EVENT using the Dialogic® Global Call API.

```
int  value  = 1;    // Disable transmit RFC2833 digits
IPM_PARM_INFO ipmParmInfo;
GC_PARM_BLKP parmblkp;

ipmParmInfo.eParm = PARMCH_DISABLE_TX_TELEPHONY_EVENT;
ipmParmInfo.pvParmValue = (void *)&value;
```

```
gc_util_insert_parm_ref(&parmblkp, IPSET_CONFIG, IPPARM_IPMPARM, (unsigned
long)sizeof(IPM_PARM_INFO), &ipmParmInfo);
gc_SetUserInfo(GCTGT_GCLIB_CHAN, lineDev, parmblkp, GC_ALLCALLS);
gc_util_delete_parm_blk(parmblkp);
```

4.24.6 Retrieving SIP Inbound DTMF

The application can retrieve the incoming RFC 2833 payload type on a call (CRN) basis using the **gc_GetUserInfo( )** function with IPSET_DTMF set ID and IPPARM_DTMF_RFC2833_PAYLOAD_TYPE parameter ID.

If available, the RFC 2833 payload type of the remote User Agent Client (UAC) can be retrieved on the inbound side upon receipt of a GCEV_OFFERED event. Alternatively, the outbound side can retrieve the remote User Agent Server (UAS) RFC 2833 payload type upon receipt of a GCEV_CONNECTED event.

This feature is applicable in 1PCC mode only.

The following limitations apply:

- To retrieve the RFC 2833 payload type, the **gc_GetUserInfo( )** function can only be called on a CRN. If passed an IPT board device or network device handle, the function will return an error.
- An application can call the **gc_GetUserInfo( )** function any time after receiving a GCEV_OFFERED event (inbound) or GCEV_CONNECTED (outbound) event. Calling the function prior to receipt of the events will return the error IPERR_INVALID_STATE.

Enabling Retrieval of SIP Inbound DTMF

To enable the feature, the application follows the same procedure for manipulating local RFC 2833 payload type by calling the **gc_SetUserInfo( )** function with IPSET_DTMF set ID and IPPARM_SUPPORT_DTMF_BITMASK parameter ID (value=IP_DTMF_TYPE_RFC_2833) after an IPT network device is opened; that is, after the GCEV_OPENEX event is received on the channel device (logical board number and logical channel number). Once this is accomplished, the application can set its own RFC 2833 payload type and also retrieve the remote RFC 2833 payload type.

Note: If an incoming SDP does not contain RFC 2833 payload type mapping, the **gc_GetUserInfo( )** function will return the IPERR_INVALID_STATE error code. You can retrieve this error code by calling the **gc_ErrorInfo( )** function.

Example 1

The following example shows how to enable RFC 2833 payload type manipulation:

```
gc_util_insert_parm_val(&parmblkp, IPSET_DTMF, IPPARM_SUPPORT_DTMF_BITMASK,
                        sizeof(char), IP_DTMF_TYPE_RFC_2833);

if (gc_SetUserInfo(GCTGT_GCLIB_CHAN, port[index].ldev, parmblkp, GC_ALLCALLS) != GC_SUCCESS) {
          // process error
}
    gc_util_delete_parm_blk(parmblkp);
```

Example 2

The following example shows how to retrieve the remote payload type from an incoming SIP message SDP carrying the appropriate media attribute with the RFC 2833 payload type mapping in an INVITE.

```
int payloadType = 0;
      case GCEV_OFFERED:
           INIT_GC_PARM_DATA_EXT(&parmdata);
           gc_util_insert_parm_ref(&infoparmblkp,
                                   IPSET_DTMF,
                                   IPPARM_DTMF_RFC2833_PAYLOAD_TYPE,
                                   sizeof(int),
                                   &payloadType );

           if(gc_GetUserInfo(GCTGT_GCLIB_CRN, pline->call[pline->index].crn, &infoparmblkp) !=
              GC_SUCCESS) {
                 // Process error
           }

           while (t_gcParmDatap = gc_util_next_parm(infoparmblkp, t_gcParmDatap)){
                  if(t_gcParmDatap->set_ID == IPSET_DTMF && t_gcParmDatap->parm_ID ==
                     IPPARM_DTMF_RFC2833_PAYLOAD_TYPE){
                     memcpy(&payloadType, (char*)t_gcParmDatap->value_buf, t_gcParmDatap-
                            >value_size);
                    printf("Payload type retrieved for the CRN= %ld payload = %d\n ", port
                            [index].crn, payloadType);
                  }

                  if(payloadType < 96 && payloadType > 127){
                      // Invalid payload type range; process error
                 }
           }
           gc_util_delete_parm_blk(infoparmblkp);
            break;
```

4.24.7 Exposing Raw RFC 2833 Data

The application can expose received raw RFC 2833 data for each DTMF tone received using the Global Call API library. This feature is applicable in 1PCC mode only.

Currently, received raw RFC 2833 data is exposed at the IP Media Library level. Here, the data exists for the duration of each RFC 2833 event in milliseconds, as well as the amplitude (returned with IPMEV_TELEPHONY_EVENT). This feature extends the Global Call API **gc_SetUserInfo( )** function, which allows the application to set technology-specific user information. This function is used to enable notification when a DTMF tone is received. Once enabled, the application receives the Global Call event, GCEV_TELEPHONY_EVENT, for every tone received.

This functionality is enabled/disabled on a per-channel basis using the **gc_SetUserInfo( )** function. By default, this functionality is disabled.

For more about the IP Media Library, see the *Dialogic® IP Media Library API Programming Guide and Library Reference.*

Parameter ID and Values

Use **gc_SetUserInfo( )** with IPSET_DTMF set ID to configure DTMF-related parameters for notification, suppression, or sending DTMF digits. The following parameter IDs support this feature:

IPPARM_TELEPHONY_EVENT_DTMF
: Used for DTMF digit notification when received within an RTP stream in RFC2833 format. Notification is via the GCEV_TELEPHONY_EVENT. Values are: IP_ENABLE, IP_DISABLE (default).

IPPARM_TELEPHONY_EVENT_INFO
: Used for fetching raw RFC2833 DTMF digit information from the GCEV_TELEPHONY_EVENT event. A GCEV_TELEPHONY_EVENT notification is received for every incoming RFC2833 DTMF digit. The information is received in the form of an IPM_TELEPHONY_INFO data structure.

Event and Data Structure

THE GCEV_TELEPHONY_EVENT event delivers raw RFC2833 data in an IPM_TELEPHONY_INFO structure (of the IP Media Library). Use the **gc_SetUserInfo( )** function to enable event notification.

The IPM_TELEPHONY_INFO data structure contains information about the volume, duration and digit ID of the received RFC2833 DTMF digit. For example:

```
typedef enum
{
   TEL_INFOTYPE_EVENT,
   TEL_INFOTYPE_TONE
} eIPM_TELEPHONY_INFO_TYPE;

typedef struct ipm_telephony_event_info_tag
{
     unsigned int   unVersion;                   /* Structure version for library use only */
     eIPM_TELEPHONY_EVENT_ID eTelephonyEventID;  /* The named event usually DTMF named event */
     short           sVolume;                    /* The power level for the DTMF event tone*/
     unsigned short   usDuration;                /* Duration for the DTMF digit in ms*/
} IPM_TELEPHONY_EVENT_INFO, *PIPM_TELEPHONY_EVENT_INFO;

typedef struct ipm_telephony_info_tag
{
    unsigned int  unVersion;                  /* Structure version for library use only */
    eIPM_TELEPHONY_INFO_TYPE eTelInfoType;    /* RFC2833 Info type - named event or tone */
    union
    {
      IPM_TELEPHONY_EVENT_INFO TelEvtInfo;      /* DTMF named event info eg. DTMF digit */
      IPM_TELEPHONY_TONE_INFO   TelToneInfo;      /* Not used as part of this FR */
   }TelephonyInfo;
} IPM_TELEPHONY_INFO, *PIPM_TELEPHONY_INFO;
```

Information about the received DTMF tone is contained in the IPM_TELEPHONY_EVENT_INFO data structure (of the IP Media Library). The following fields are applicable to this feature:

unVersion
: Specifies the version of the IPM_TELEPHONY_EVENT_INFO structure. This field is used by the IP Media library for checking the backward binary compatibility of future versions of the data structure.

eTelephonyEventID
: Specifies a named event, typically a DTMF named event. The data type of the telephony_event field is an eIPM_TELEPHONY_EVENT_ID enumeration that lists all possible tone signal identifiers as described in RFC 2833. The eIPM_TELEPHONY_EVENT_ID is an enumeration with the following values:
  - SIGNAL_ID_EVENT_DTMF_0
  - SIGNAL_ID_EVENT_DTMF_1
  - SIGNAL_ID_EVENT_DTMF_2
  - SIGNAL_ID_EVENT_DTMF_3
  - SIGNAL_ID_EVENT_DTMF_4
  - SIGNAL_ID_EVENT_DTMF_5
  - SIGNAL_ID_EVENT_DTMF_6
  - SIGNAL_ID_EVENT_DTMF_7
  - SIGNAL_ID_EVENT_DTMF_8
  - SIGNAL_ID_EVENT_DTMF_9
  - SIGNAL_ID_EVENT_DTMF_*
  - SIGNAL_ID_EVENT_DTMF_#

sVolume
: Specifies the power level associated with the DTMF event tone, expressed in dBm0 after dropping the sign (i.e., with a range from 0 to -63 dBm0 resulting in values from 0 to 63).

usDuration
: Specifies the duration of the DTMF digit in milliseconds.

Enabling Event Notification Example

The following example demonstrates how to enable event notification when a DTMF tone is received.

```
#include "gcip.h"
#include "gclib.h"
#include "gcip_defs.h"

int enable_dtmf_event(LINEDEV linedev)
{
    GC_PARM_BLKP parmblkp = NULL;

    /* Set the parameter block */
    gc_util_insert_parm_val(&parmblkp, IPSET_DTMF,
    IPPARM_TELEPHONY_EVENT_DTMF, sizeof(int), IP_ENABLE);

    /* Enable DTMF events functionality in the IPCCLIB */
    if (gc_SetUserInfo(GCTGT_GCLIB_CHAN, linedev, parmblkp, GC_ALLCALLS) !=
    GC_SUCCESS)
    {
           /* Process error */
    }

    /* Free the parameter block */
    gc_util_delete_parm_blk(parmblkp);

    /* More processing */
    return;
}
```

GCEV_TELEPHONY_EVENT Processing Example

The following example illustrates GCEV_TELEPHONY_EVENT processing.

```
#include "gcip.h"
#include "gclib.h"
#include "gcip_defs.h"
#include "ipmlib.h"

void process_event(void)
{
    METAEVENT metaevent;
    GC_PARM_BLKP my_blkp = NULL;
    GC_PARM_DATAP my_datap;
    IPM_TELEPHONY_INFO my_telInfo;

    if(gc_GetMetaEvent(&metaevent) != GC_SUCCESS)
    {
           /* Process error */
    }

    switch(metaevent.evttype)
    {
           .
           .
           .
           case GCEV_TELEPHONY_EVENT:
                  /* Make a copy of the parm blk */
                  if(metaevent.extevtdatap)
                 {
                          if ( gc_util_copy_parm_blk( &my_blkp,
                          (GC_PARM_BLKP)(metaevent.extevtdatap))!= GC_SUCCESS)
                          {
                                 /* Process error */
                          }

                           my_datap = gc_util_find_parm(my_blkp, IPSET_DTMF,
                           IPPARM_TELEPHONY_EVENT_INFO);

                           if (my_datap != NULL)
                           {
                                  memcpy(&my_telInfo, my_datap->value_buf,
                                  my_datap->value_size);
                           }

                           switch(my_telInfo.eTelInfoType)
                           {
                                  case TEL_INFOTYPE_EVENT:
                                         /* Fetch DTMF tone information */
                                         break;
                                   default:
                                         /* Print Error */
                           }

                }
                else
                {
                          /* Process error */
                }
                break;
     .
     .
     .
   }
}
```

4.25 Specifying RTP Stream Establishment

Note: The information in this section only applies when the Dialogic® Global Call API IP Call Control library is started in the first party call control (1PCC) operating mode. The capability described in this section is not supported when the library is started in the third party call control (3PCC) operating mode.

When using the Dialogic® Global Call API, RTP streaming can be established before the call is connected (that is, before the calling party receives the GCEV_CONNECTED event). This feature enables a voice message to be played to the calling party (for example, a message stating that the called party is unavailable for some reason) without the calling party being billed for the call.

The **gc_SetUserInfo( )** function can be used to specify call-related information such as coder information and display information before issuing **gc_CallAck( )**, **gc_AcceptCall( )** or **gc_AnswerCall( )**. See Section 8.3.26, "gc_SetUserInfo( ) Variances for IP", on page 587 for more information.

On the called party side, RTP streaming can be established before any of the following functions are issued to process the call:

- **gc_AcceptCall( )** – SIP Ringing (180) message returned to the calling party
- **gc_AnswerCall( )** – SIP OK (200) message returned to the calling party

4.26 Managing Quality of Service Alarms

Note: The information in this section only applies when the Dialogic® Global Call API IP Call Control library is started in the first party call control (1PCC) operating mode. When the library is started in the third party call control (3PCC) operating mode, QoS alarms are configured and handled via the IP Media (IPML) API library.

The Dialogic® Global Call API supports the setting and retrieving of Quality of Service (QoS) thresholds and the handling of a QoS alarm when it occurs. The QoS thresholds supported by Dialogic® Global Call API are:

- jitter
- lost packets

- RTCP inactivity
- RTP inactivity

When using Dialogic® Global Call API with other technologies (such as E1 CAS or T1 Robbed Bit), alarms are managed and reported on the network device. For example, when **gc_OpenEx( )** is issued, specifying both a network device (dtiB1T1) and a voice device (dxxxB1C1) in the **devicename** parameter, the function retrieves a Dialogic® Global Call API line device. This Dialogic® Global Call API line device can be used directly in Dialogic® Global Call API Alarm Management System (GCAMS) functions to manage alarms on the network device.

When using the Dialogic® Global Call API with IP technology, alarms such as QoS alarms are more directly related to the media processing and are therefore reported on the media device rather than on the network device. When **gc_OpenEx( )** is issued, specifying both a network device (iptB1T1) and a media device (ipmB1C1) in the **devicename** parameter, two Dialogic® Global Call API line devices are created:

- The first Dialogic® Global Call API line device corresponds to the network device and is retrieved in the **gc_OpenEx( )** function.
- The second Dialogic® Global Call API line device corresponds to the media device and is retrieved using the **gc_GetResourceH( )** function. This is the line device that must be used with GCAMS functions to manage QoS alarms. See the *Global Call API Programming Guide* for more information about GCAMS.

Note: Applications **must** include the *gcipmlib.h* header file before Dialogic® Global Call API can be used to set or retrieve QoS threshold values.

4.26.1 Alarm Source Object Name

In Dialogic® Global Call API, alarms are managed using the Dialogic® Global Call API Alarm Management System (GCAMS). Each alarm source is represented by an Alarm Source Object (ASO) that has an associated name. When using Dialogic® Global Call API with IP, the ASO name is **IPM QoS ASO**. The ASO name is useful in many contexts, for example, when configuring a device for alarm notification.

4.26.2 Retrieving the Media Device Handle

To retrieve the Dialogic® Global Call API line device corresponding to the media device, use the **gc_GetResourceH( )** function. See Section 8.3.12, "gc_GetResourceH( ) Variances for IP", on page 554 for more information.

The Dialogic® Global Call API line device corresponding to the media device is the device that must be used with GCAMS functions to manage QoS alarms.

4.26.3 Setting QoS Threshold Values

To set QoS threshold values, use the **gc_SetAlarmParm( )** function. See Section 8.3.24, "gc_SetAlarmParm( ) Variances for IP", on page 582 for more information.

The following code demonstrates how to set QoS threshold values.

Note: The following code uses the IPM_QOS_THRESHOLD_INFO structure from the IP Media Library (IPML). See the *Dialogic® IP Media Library API Library Reference* and the *Dialogic® IP Media Library API Programming Guide* for more information.

```
/*****************************************************************************
Routine: SetAlarmParm
Assumptions/Warnings: None.
Description: calls gc_SetAlarmParm()
Parameters: handle of the Media device
Returns: None
******************************************************************************/

void SetAlarmParm(int hMediaDevice)
{
   ALARM_PARM_LIST alarm_parm_list;
   IPM_QOS_THRESHOLD_INFO QoS_info;
   alarm_parm_list.n_parms = 1;
   QoS_info.unCount=1;
   QoS_info.QoSThresholdData[0].eQoSType = QOSTYPE_JITTER;
   QoS_info.QoSThresholdData[0].unTimeInterval = 1000;
   QoS_info.QoSThresholdData[0].unDebounceOn = 5000;
   QoS_info.QoSThresholdData[0].unDebounceOff = 15000;
   QoS_info.QoSThresholdData[0].unFaultThreshold = 50;
   QoS_info.QoSThresholdData[0].unPercentSuccessThreshold = 90;
   QoS_info.QoSThresholdData[0].unPercentFailThreshold = 10;

   alarm_parm_list.alarm_parm_fields[0].alarm_parm_data.pstruct =
   (void *) &QoS_info;

   if (gc_SetAlarmParm(hMediaDevice, ALARM_SOURCE_ID_NETWORK_ID,
        ParmSetID_qosthreshold_alarm, &alarm_parm_list, EV_SYNC)!= GC_SUCCESS)
   {
      /* handle gc_SetAlarmParm() failure */
      printf("SetAlarmParm(hMediaDevice=%d, mode=EV_SYNC) Failed", hMediaDevice);
      return;
   }
   printf("SetAlarmParm(hMediaDevice=%d, mode=EV_SYNC) Succeeded", hMediaDevice);
}
```

4.26.4 Retrieving QoS Threshold Values

To retrieve QoS threshold values, use the **gc_GetAlarmParm( )** function. See Section 8.3.9, "gc_GetAlarmParm( ) Variances for IP", on page 550 for more information.

The following code demonstrates how to retrieve QoS threshold values.

Note: The following code uses the IPM_QOS_THRESHOLD_INFO structure from the IP Media Library (IPML). See the *Dialogic® IP Media Library API Library Reference* and the *Dialogic® IP Media Library API Programming Guide* for more information.

```
/*****************************************************************************
Routine: GetAlarmParm
Assumptions/Warnings: None
Description: calls gc_GetAlarmParm()
Parameters: handle of Media device
Returns: None
******************************************************************************/

void GetAlarmParm(int hMediaDevice)
```

```
{
   ALARM_PARM_LIST alarm_parm_list;
   unsigned int n;
   IPM_QOS_THRESHOLD_INFO QoS_info;
   IPM_QOS_THRESHOLD_INFO *QoS_infop;

   QoS_info.unCount=2;
   QoS_info.QoSThresholdData[0].eQoSType = QOSTYPE_LOSTPACKETS;
   QoS_info.QoSThresholdData[1].eQoSType = QOSTYPE_JITTER;

   /* get QoS thresholds for LOSTPACKETS and JITTER */
   alarm_parm_list.alarm_parm_fields[0].alarm_parm_data.pstruct = (void *) &QoS_info;
   alarm_parm_list.n_parms = 1;

   if (gc_GetAlarmParm(hMediaDevice, ALARM_SOURCE_ID_NETWORK_ID,
        ParmSetID_qosthreshold_alarm, &alarm_parm_list, EV_SYNC) != GC_SUCCESS)
   {
      /* handle gc_GetAlarmParm() failure */
      printf("gc_GetAlarmParm(hMediaDevice=%d, mode=EV_SYNC) Failed", hMediaDevice);
      return;
   }

   /* display threshold values retrieved */
   printf("n_parms = %d\n", alarm_parm_list.n_parms);
   QoS_infop = alarm_parm_list.alarm_parm_fields[0].alarm_parm_data.pstruct;
   for (n=0; n < QoS_info.unCount; n++)
   {
      printf("QoS type = %d\n", QoS_infop->QoSThresholdData[n].eQoSType);
      printf("\tTime Interval = %u\n", QoS_infop->QoSThresholdData[n].unTimeInterval);
      printf("\tDebounce On = %u\n", QoS_infop->QoSThresholdData[n].unDebounceOn);
      printf("\tDebounce Off = %u\n", QoS_infop->QoSThresholdData[n].unDebounceOff);
      printf("\tFault Threshold = %u\n", QoS_infop->QoSThresholdData[n].unFaultThreshold);
      printf("\tPercent Success Threshold = %u\n",
             QoS_infop->QoSThresholdData[n].unPercentSuccessThreshold);
      printf("\tPercent Fail Threshold = %u\n",
             QoS_infop->QoSThresholdData[n].unPercentFailThreshold);
      printf("\n\n");
   }
}
```

4.26.5 Handling QoS Alarms

The application must first be enabled to receive notification of alarms on the specified line device. The following code demonstrates how this is achieved.

```
/*****************************************************************
*        NAME: enable_alarm_notification(struct channel *pline)
* DESCRIPTION: Enables all alarms notification for pline
*              Also fills in pline->mediah
*       INPUT: pline - pointer to channel data structure
*     RETURNS: None - exits if error
*    CAUTIONS: Does no sanity checking as to whether or not the technology
*              supports alarms - assumes caller has done that already
*****************************************************************/

static void enable_alarm_notification(struct channel *pline)
{
   char     str[MAX_STRING_SIZE];
   int      alarm_ldev;                /* linedevice that alarms come on */

   alarm_ldev = pline->ldev;           /* until proven otherwise */
   if (pline->techtype == H323)
   {
      /* Recall that the alarms for IP come on the media device, not the network device */
      if (gc_GetResourceH(pline->ldev, &alarm_ldev, GC_MEDIADEVICE) != GC_SUCCESS)
      {
         sprintf(str, "gc_GetResourceH(linedev=%ld, &alarm_ldev,
                 GC_MEDIADEVICE) Failed", pline->ldev);
         printandlog(pline->index, GC_APIERR, NULL, str);
         exitdemo(1);
      }
      sprintf(str, "gc_GetResourceH(linedev=%ld, &alarm_ldev,
              GC_MEDIADEVICE) passed, mediah = %d", pline->ldev, alarm_ldev);
      printandlog(pline->index, MISC, NULL, str);
      pline->mediah = alarm_ldev;         /* save for later use */
   }
   else
   {
      printandlog(pline->index, MISC, NULL, "Not setting pline->mediah
                  since techtype != H323");
   }
   sprintf(str, "enable_alarm_notification - pline->mediah = %d\n", (int) pline->mediah);

   if (gc_SetAlarmNotifyAll(alarm_ldev, ALARM_SOURCE_ID_NETWORK_ID,
       ALARM_NOTIFY) != GC_SUCCESS)
   {
      sprintf(str, "gc_SetAlarmNotifyAll(linedev=%ld,
              ALARM_SOURCE_ID_NETWORK_ID, ALARM_NOTIFY) Failed", pline->ldev);
      printandlog(pline->index, GC_APIERR, NULL, str);
       exitdemo(1);
   }
   sprintf(str, "gc_SetAlarmNotifyAll(linedev=%ld, ALARM_SOURCE_ID_NETWORK_ID,
           ALARM_NOTIFY) PASSED", pline->ldev);
   printandlog(pline->index, MISC, NULL, str);
}
```

When a GCEV_ALARM event occurs, use the Dialogic® Global Call API Alarm Management System (GCAMS) functions such as, **gc_AlarmNumber( )** to retrieve information about the alarm. The following code demonstrates how to process a QoS alarm when it occurs. In this case the application simply logs information about the alarm.

```
/*****************************************************************
*        NAME: void print_alarm_info(METAEVENTP metaeventp,
*                                 struct channel *pline)
* DESCRIPTION: Prints alarm information
*      INPUTS: metaeventp - pointer to the alarm event
*              pline - pointer to the channel data structure
*     RETURNS: NA
*    CAUTIONS: Assumes already known to be an alarm event
*****************************************************************/

static void print_alarm_info(METAEVENTP metaeventp, struct channel *pline)
{
   long              alarm_number;
   char              *alarm_name;
   unsigned long     alarm_source_objectID;
   char              *alarm_source_object_name;
   char              str[MAX_STRING_SIZE];

   if (gc_AlarmNumber(metaeventp, &alarm_number) != GC_SUCCESS)
   {
      sprintf(str, "gc_AlarmNumber(...) FAILED");
      printandlog(pline->index, GC_APIERR, NULL, str);
      printandlog(pline->index, STATE, NULL, " ");
      exitdemo(1);
   }
   if (gc_AlarmName(metaeventp, &alarm_name) != GC_SUCCESS)
   {
      sprintf(str, "gc_AlarmName(...) FAILED");
      printandlog(pline->index, GC_APIERR, NULL, str);
      printandlog(pline->index, STATE, NULL, " ");
      exitdemo(1);
   }

   if (gc_AlarmSourceObjectID(metaeventp, &alarm_source_objectID) != GC_SUCCESS)
   {
      sprintf(str, "gc_AlarmSourceObjectID(...) FAILED");
      printandlog(pline->index, GC_APIERR, NULL, str);
      printandlog(pline->index, STATE, NULL, " ");
      exitdemo(1);
   }

   if (gc_AlarmSourceObjectName(metaeventp, &alarm_source_object_name) != GC_SUCCESS)
   {
      sprintf(str, "gc_AlarmSourceObjectName(...) FAILED");
      printandlog(pline->index, GC_APIERR, NULL, str);
      printandlog(pline->index, STATE, NULL, " ");
      exitdemo(1);
   }

   sprintf(str, "Alarm %s (%d) occurred on ASO %s (%d)",
           alarm_name, (int) alarm_number, alarm_source_object_name,
           (int) alarm_source_objectID);

   printandlog(pline->index, MISC, NULL, str);
}
```

See the *Dialogic® Global Call API Programming Guide* for more information about the operation of GCAMS and the *Dialogic® Global Call API Library Reference* for more information about GCAMS functions.

4.27 Registration

In an H.323 network, a Gatekeeper manages the entities in a specific zone and an endpoint must register with the Gatekeeper to become part of that zone. In a SIP network, a Registrar manages a set of associations or bindings between Addresses-of-Record and actual endpoint addresses for a domain. The Dialogic® Global Call API provides applications with the ability to perform endpoint registration. These capabilities are described in the following topics:

- Registration Overview
- Registration Operations
- Sending and Receiving Nonstandard Registration Messages (H.323)
- Registration Code Examples
- Gatekeeper Registration Failure (H.323)

4.27.1 Registration Overview

The Dialogic® Global Call API provides a number of options for registration and manipulation of registration information. The Dialogic® Global Call API simplifies and abstracts the network RAS messages in H.323 and REGISTER messages in SIP.

When using the Dialogic® Global Call API to perform endpoint registration, the following general conditions and restrictions apply:

- An application must use an IPT board device handle to perform registration. A board device handle can be obtained by using **gc_OpenEx( )** with a **devicename** parameter of "N_iptBx".
- When using the **gc_ReqService( )** function, two mandatory parameter elements, GCSET_SERVREQ / PARM_REQTYPE and GCSET_SERVREQ / PARM_ACK, are required in the GC_PARM_BLK parameter block. These parameters are required by the generic service request mechanism provided by Dialogic® Global Call API and are not sent in any registration message.
- When setting H.323 alias or SIP Transport Address information, the **gc_ReqService( )** function can include more than one address in the GC_PARM_BLK associated with the function. Prefixes are ignored for SIP.
- Registration operations cannot be included in the preset registration information using **gc_SetConfigData( )**.

H.323 Gatekeeper Registration

In H.323, the following operations (and the corresponding RAS messages) are supported:

- locating a gatekeeper via unicast or multicast (RAS messages: GRQ/GCF/GRJ)
- registration (RAS message: RRQ)
- specifying one-time or periodical registration (RAS message: RRQ)
- changing registered information (RAS message: RRQ)
- removing registered information by value (RAS message: RRQ)
- sending non-standard registration message (RAS message: NonStandardMessage)
- deregistering (RAS messages: URQ/UCF/URJ)
- handling calls according to the gatekeeper policy for directing and routing calls (RAS messages: ARQ/ACF/ARJ, DRQ/DCF/DRJ)

Note: For detailed information on RAS negotiation, see *ITU-T Recommendation H.225.0*.

When using the Dialogic® Global Call API to perform H.323 Gatekeeper registration, the following conditions and restrictions apply in addition to the general conditions noted above:

- An H.323 application must perform registration only when there are no active calls.
- Once an H.323 application chooses to be registered with a Gatekeeper, it can change its Gatekeeper by deregistering and reregistering with another Gatekeeper.
- Once an H.323 application is registered and has active calls, deregistration or switching to a different Gatekeeper will disconnect all active calls and cause GCEV_DISCONNECTED events to be sent to the application. The **gc_ResetLineDev( )** function can be used to put channels in the Idle state before deregistering.
- Once an H.323 application chooses to be registered with a Gatekeeper, it cannot handle calls without being registered with some Gatekeeper or explicitly deregistering. If the Gatekeeper connection is lost, for example, the application cannot handle calls until it either reregisters or deregisters.

- Once an application is registered, if it wishes to handle calls without the registration protocol (that is, return to the same mode as before registration), it can simply deregister. When the application deregisters, all existing calls are dropped and GCEV_DISCONNECTED events are sent to the application, and new calls may be blocked for a short time while the H.323 stack restarts in manual RAS mode.

SIP Registration

The SIP REGISTER method is used to register associations between a media endpoint alias and its real (transport) address. These associations are commonly referred to as *bindings*, each of which represents a unique tuple of several items, including:

- the Registrar's address, which is specified as the Request-URI
- the Address of Record (a "name" that will be used to easily locate the SIP endpoint), which is specified as the To header field
- the Transport address (the actual URI of the SIP endpoint), which is specified as the Contact header field
- the Sender's Address of Record (only used in third-party call control environments), which is specified as the From header field

An application can register as many bindings as it wants, so that a given SIP endpoint may have multiple AORs or aliases. When a Proxy receives an INVITE request addressed to a registered AOR, it routes the request to the endpoint address identified in the binding. For example, if a binding exists between the AOR
tom@somewhere.com
and the transport address
454554-tom-sdih53@py1.somewhere.com:5063
an INVITE addressed to tom@somewhere.com would be routed by a Proxy to the address 454554-tom-sdih53@py1.somewhere.com:5063. When the application receives the GCEV_OFFERED event for this INVITE, it can extract the "454554-tom-sdih53" portion of the address from the Phone List and use that information to route the call to the appropriate logical SIP endpoint. Note that calls are **not** automatically routed to a specific IPT device by the registration mechanism.

Global Call supports registering and de-registering with a Registrar, and querying the Registrar for existing bindings; it does not support receiving SIP REGISTER requests. Table 13 associates abstract Registrar registration concepts with SIP REGISTER message elements and Global Call programming interface elements.

Table 13. SIP REGISTER Method

Concept	SIP REGISTER Element	Global Call Interface Element
Initiate registration	REGISTER method	gc_ReqService( )
Registrar's address	Request-URI	IPSET_REG_INFO IPPARM_REG_ADDRESS IP_REGISTER_ADDRESS.reg_server
† If not supplied by application, library automatically uses the value provided for Alias		

Table 13. SIP REGISTER Method (Continued)

Concept	SIP REGISTER Element	Global Call Interface Element
Alias (Address-of-record)	To header field	IPSET_REG_INFO IPPARM_REG_ADDRESS IP_REGISTER_ADDRESS.reg_client
Sender's address-of-record (only used in 3rd party call control environments)	From header field	IPSET_SIP_MSGINFO IPPARM_SIP_HDR header string starting with "From:" †
Transport address (actual endpoint address)	Contact header field	IPSET_LOCAL_ALIAS IPPARM_ADDRESS_TRANSPARENT address string
Auto-refresh interval	Expires header field	IPSET_REG_INFO IPPARM_REG_ADDRESS IP_REGISTER_ADDRESS.time_to_live
† If not supplied by application, library automatically uses the value provided for Alias		

Note: Because the Transport Address is sent to the Registrar in the Contact header field, which can use any valid URI scheme according to RFC 3261, the header field must include a valid URI scheme prefix, such as "sip:" or "sips:". If the application does not supply a scheme prefix, the call control library automatically inserts "sip:", but only after the SIP stack has generated a parser error. These stack parser errors are written to the RTFLog file unless the user turns off logging of this type of error. To turn off the logging of these parser errors, find the line

```
<MClient name="PARSER" state = "1"/>
```

in the *RtfConfigWin.xml* file and replace it with

```
<MClient name="PARSER" state = "1">
   <MClientLabel name="Error" state = "0"/>
</MClient>
```

When using SIP, it is important to note that RFC3261 specifies that the "host" portion of a URI that is given as a numeric IPv4 address (for example, 123.211.40.90) and one given as a domain name (for example, example.com) are treated as unique even if they actually resolve to the same entity. Applications should be careful to ensure that the "host" portions of any URIs in all subsequent operations on that binding are consistent with way they were specified during the initial registration.

4.27.2 Registration Operations

Applications perform all types of registration operations (registering, deregistering, querying, and modifying or deleting registration information) using the **gc_ReqService( )** function. The specific operation to perform and the information necessary for that operation are specified in parameter elements in a GC_PARM_BLK that is passed to the **gc_ReqService( )** function. The specific parameters to use for each type of operation are described in the following subsections.

In addition to the parameter elements that are required for H.323 or SIP registrations, there are two mandatory parameter elements that are required by the generic service request mechanism even though they have no meaning in the context of H.323/SIP endpoint registration. These two parameters, GCSET_SERVREQ / PARM_REQTYPE and GCSET_SERVREQ / PARM_ACK, must always be present in the GC_PARM_BLK.

The **gc_ReqService( )** function operates in the asynchronous mode, and the application receives a GCEV_SERVICERESP termination event if the call control library succeeds in communicating with the registration server. It is important to note that a GCEV_SERVICERESP event indicates that the requested registration operation was completed successfully only if the event's result code (the ccValue field in the GC_INFO structure from a **gc_ResultInfo( )** function call) is IPERR_OK. If the result code is any other value, there was some sort of error during the registration.

4.27.2.1 Configuring the Maximum Number of Registrations (SIP)

Because internal stack resources are required to monitor each unique binding that is set to auto-refresh, and because auto-refresh is the default mode for SIP registration, the Global Call call control library allows the application to configure the maximum number of registrations for each virtual board when the system is started. If an application requests a registration that exceeds the configured maximum number of registrations for the virtual board, the application's request is rejected by the call control library, which generates a GCEV_SERVICERESP event with the response code IPEC_REG_FAIL_insufficientInternalResources.

The configuration of the maximum number of registrations is accomplished on a virtual board basis by setting the sip_registrar_registrations field in the IP_VIRTBOARD structure for each virtual board before **gc_Start( )** is called. The default value for this field sets the maximum number of registrations to be the same as the maximum number of SIP calls (the sip_max_calls field in IP_VIRTBOARD), which is appropriate in most situations. If the application needs to register all or most users with more than one Registrar, or to register multiple transport addresses for all or most users, it needs to increase this configuration parameter from the default value.

The mandatory **INIT_IP_VIRTBOARD( )** function populates the IP_VIRTBOARD structure with the default value for the sip_registrar_registrations field. The following code snippet illustrates how an application might increase the maximum number of registrations on the second of two virtual boards to allow two registrations per user:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[1].sip_registrar_registrations = 240; /* override defaults no. of registrations*/
```

Note: Features that are enabled or configured via the IP_VIRTBOARD structure cannot be disabled or reconfigured once the library has been started. All items set in this data structure take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

4.27.2.2 Locating a Registration Server

A Dialogic® Global Call API application can choose to use a known address for the registration server (H.323 Gatekeeper or SIP Registrar) or to discover a registration server by multicasting to a well-known address on which registration servers listen. This choice is determined by the IP address specified as the registration address during registration.

The registration address is specified in the IPPARM_REG_ADDRESS parameter in the IPSET_REG_INFO parameter set ID. The value of the IPPARM_REG_ADDRESS is an IP_REGISTER_ADDRESS structure, which includes a reg_server field that contains the address value. A specific range of IP addresses is reserved for multicast transmission:

- If the application specifies an address in the range of multicast addresses or specifies the default multicast address (IP_REG_MULTICAST_DEFAULT_ADDR), then registration server discovery is selected.
- If the application specifies an address outside the range of multicast addresses, then registration with a specific server is selected.

Note: In SIP, if the reg_server field contains NULL or an invalid address, the default multicast address is automatically used by the library.

When using the default multicast registration address, the application can specify the maximum number of hops (connections between routers) in the max_hops field of the IP_REGISTER_ADDRESS structure.

H.323

For H.323 registration, the port number used for RAS is one less than the port number used for signaling. To avoid a port conflict when configuring multiple ipt board devices, do not assign consecutive H.323 signaling port numbers to ipt board devices in the IPCCLIB_START_DATA structure. See Section 8.3.27, "gc_Start( ) Variances for IP", on page 590 for more information.

4.27.2.3 Registration Requests

An application uses the **gc_ReqService**( ) function to register with a Gatekeeper/Registrar. The registration information in this case is included in the GC_PARM_BLK associated with the **gc_ReqService**( ) function. See Section 4.27.4, "Registration Code Examples", on page 326 for more information.

H.323

If registration is initiated by a Dialogic® Global Call API application via **gc_ReqService( )** and the Gatekeeper rejects the registration, a GCEV_SERVICERESP event containing the result code IPEC_RASReasonInvalidIPEC_RASAddress.

If an application's registration attempt fails for any reason, it is the application's responsibility to re-register.

If the stack receives an unsolicited URQ, it silently responds with a UCF, and immediately tries to re-register with the same Gatekeeper. If three successive attempts at re-registration fail, the library generates GCEV_TASKFAIL. If the application attempts to use the **gc_ReqService( )** function during this time, those function calls will fail.

SIP

In SIP, an application can make multiple simultaneous registration requests to different Registrars or to the same Registrar on behalf of different User Agents. To allow the application to distinguish among multiple completion events from these simultaneous requests, the data associated with the completion event contains a Service ID parameter that is the number that was handed back to the application when the initiating **gc_ReqService( )** was made.

According to RFC3261, applications may not make more than one registration attempt at the same time for a particular User Agent on a particular Registrar. If the application attempts to send a second REGISTER request to a given Registrar for the same UA before the initial REGISTER transaction completes, the call control library rejects the request and generates a GCEV_SERVICERESP event containing the result code IPEC_REG_FAIL_registrationTransactionInProgress to notify the application of the rejection.

4.27.2.4 Auto-Refreshing Registrations

The Dialogic® Global Call API enables an application to specify a one-time registration or periodic registration where bindings are automatically re-registered with the Gatekeeper/Registrar at the interval (in seconds) specified by the application. Applications that are using automatic re-registration are not notified of successful registration refresh transactions.

H.323

In H.323 registration, periodic registration is achieved by setting the time_to_live field in the IP_REGISTER_ADDRESS structure. If the parameter is set to zero (the default value), then the stack uses one-time registration functionality. If the parameter is set to a value greater than zero, then each registration with the server is valid for the specified number of seconds and the stack automatically refreshes its request before timeout.

If the Gatekeeper rejects the registration (sends RRJ) during periodic registration, the application will receive an unsolicited GCEV_TASKFAIL event that contains a reason provided by the Gatekeeper. If the Gatekeeper does not set the reason, the default reason is IPEC_RASReasonInvalidIPEC_RASAddress.

SIP

When using SIP, auto-refresh is used by default. If the application does not explicitly set the time_to_live value in the IP_REGISTER_ADDRESS structure (that is, doesn't change the value from its default value of 0), the call control library automatically sets the Expires header field in the REGISTER request to a a value of 3600 seconds. If the application wishes to request a longer or shorter auto-refresh interval, it simply sets the time_to_live field to the appropriate value, and that value is set in the Expires header field.

The actual expiration time for registration is determined by the Registrar, which may or may not accept the Expires value suggested in the REGISTER request. The expiration time received from the Registrar is recorded and used by the Dialogic® Global Call API library only if the application has not disabled the auto-refresh mechanism. If the expiration time returned by the Registrar is greater than 40 seconds, re-registration is attempted 30 seconds before the registration is set to expire. If the expiration time returned by the Registrar is 40 seconds or less, re-registration is attempted within 5 seconds of receiving that response. When auto-refresh is enabled, the call control library rejects registration refresh times of 5 seconds or less and generates a GCEV_SERVICERESP event with the response code IPEC_REG_FAIL_invalidExpires. If a refresh time of 5 seconds or less is actually desired, the application must disable the auto-refresh mechanism for each binding and will then be responsible for explicitly renewing those bindings with the Registrar.

If the automatic re-registration fails because the Registrar rejects the request, the Registrar's response code is forwarded to the application in a GCEV_SERVICERESP event. Automatic re-registration can also fail if constant application activity on a particular binding causes re-registration to be postponed beyond the binding's actual expiration time. (A 500ms postponement occurs when an auto re-registration attempt collides with a current application transaction on the same binding.) In this case the GCEV_SERVICERESP event sent to the application contains the result code IPEC_REG_FAIL_reRegistrationRequired. In either case, the application is then responsible for re-registering the binding, if appropriate.

The extra data associated with a re-registration failure event includes:

- Request-URI (as IPSET_SIP_MSGINFO / IPPARM_REQUEST_URI)
- To header field value (as IPSET_SIP_MSGINFO / IPPARM_TO)
- From header field value, if one had been provided (as IPSET_SIP_MSGINFO / IPPARM_TO)
- Contact header field value that failed to auto refresh (as IPSET_LOCAL_ALIAS / IPPARM_ADDRESS_TRANSPARENT)

A SIP application can explicitly disable or re-enable auto-refresh on a per registration basis, by using the following parameter element:

IPSET_REG_INFO
  IPPARM_REG_AUTOREFRESH
  and one of the following values:
  - IP_REG_AUTOREFRESH_DISABLE – disable auto-refresh for a specific registration
  - IP_REG_AUTOREFRESH_ENABLE – enable auto-refresh for a specific registration, using the non-zero value specified in IP_REGISTER_ADDRESS.time_to_live or the default value of 3600 in the Expires header field

  Note: If this parameter is not present in the GC_PARM_BLK when registration is requested, auto-refresh is enabled by default.

4.27.2.5 Receiving Notification of Registration

An application that sends a registration request to a Gatekeeper/Registrar receive notification of whether the registration is successful or not. When using the Dialogic® Global Call API, the application receives a GCEV_SERVICERESP termination event with an associated GC_PARM_BLK that contains the following elements:

IPSET_PROTOCOL
  IPPARM_PROTOCOL_BITMASK
  with one of the following values:
  - IP_PROTOCOL_H323
  - IP_PROTOCOL_SIP

IPSET_REG_INFO
  IPPARM_REG_STATUS
  with one of the following values:
- IP_REG_CONFIRMED – registration operation completed properly
- IP_REG_REJECTED – registration operation did not complete properly; the **gc_ResultInfo( )** function can be used to retrieve the reason for the failure

SIP

For registrations with a SIP Registrar, the GC_PARM_BLK associated with the GCEV_SERVICERESP termination event also contains the following element:

IPSET_REG_INFO
  IPPARM_REG_SERVICEID
- value = the Service ID that was handed back to the application when the initiating **gc_ReqService( )** was made

This Service ID can be used by the application to distinguish among multiple events returned on a given handle, since the application can send multiple simultaneous REGISTER requests to different Registrars or to the same Registrar on behalf of different User Agents.

4.27.2.6 Querying Registration Information (SIP)

Global Call provides a mechanism for a SIP application to query a Registrar to determine what bindings currently exist. To do this, the application calls **gc_ReqService( )** with the following parameter element included in the GC_PARM_BLK that is passed to the function:

IPSET_REG_INFO
  IPPARM_OPERATION_REGISTER
- value = IP_REG_QUERY_INFO

The application specifies the Registrar and Alias to query by including the following parameter element in the GC_PARM_BLK that is passed to **gc_ReqService( )**:

IPSET_REG_INFO
  IPPARM_REG_ADDRESS
- value = IP_REGISTER_ADDRESS structure with reg_client and reg_server fields filled in to indicate the desired Registrar address and Alias to query

Note: This parameter is optional. If it is not included in the GC_PARM_BLK, or if either of the addresses in the IP_REGISTER_ADDRESS structure is not supplied, the most recently used Registrar address and Alias are used by default.

By default, the registration query operation returns all Transport Addresses that are currently registered for the specified Alias by the application. If the application wishes to query *all* Transport Addresses that have been registered in the Registrar for the specified Alias (that is, all registrations by all applications), the GC_PARM_BLK that it supplies to the **gc_ReqService( )** function must include the following element:

IPSET_LOCAL_ALIAS
  IPPARM_ADDRESS_TRANSPARENT
- value = "*"

The GCEV_SERVICERESP completion event for this function call contains all current bindings for the specified Address of Record in a series of IPSET_LOCAL_ALIAS / IPPARM_ADDRESS_TRANSPARENT parameter elements. The value of each of these elements is a null-terminated string that contains a current binding created by this application along with any header field parameters that were appended by the Registrar.

4.27.2.7 Changing Registration Information

The Dialogic® Global Call API provides the ability to modify or add to the registration information after it has been registered with the Gatekeeper/Registrar. To change registration information, the application uses the **gc_ReqService( )** function and passes a GC_PARM_BLK that contains the following element:

IPSET_REG_INFO
    IPPARM_OPERATION_REGISTER
    and one of the following values:
    - IP_REG_SET_INFO – override existing registration
    - IP_REG_ADD_INFO – add to existing registration information

A SIP application can specify the Registrar and Alias to modify information for by including the following parameter in the GC_PARM_BLK that is passed to **gc_ReqService( )**:

IPSET_REG_INFO
    IPPARM_REG_ADDRESS
    - value = IP_REGISTER_ADDRESS structure with reg_client and reg_server fields filled in to indicate the desired Registrar address and Alias

Note: This parameter is optional. If it is not included in the GC_PARM_BLOCK, or if either of the addresses in the IP_REGISTER_ADDRESS structure is not supplied, the most recently used Registrar address and Alias are used by default.

The overriding or additional information is contained in other elements in the GC_PARM_BLK. The elements that can be included are given in Table 39, “Registration Information When Using H.323”, on page 580 and Table 40, “Registration Information When Using SIP”, on page 581.

Note: For SIP, the Sender’s Address of Record that was used to initially register a binding never changes. Any attempt to update this value is ignored.

4.27.2.8 Removing Registered Information by Value

Global Call allows applications to delete one or more registration values from an existing registration. This applies to aliases and supported prefixes in H.323, and to Transport Addresses in SIP. When an application needs to delete one or more specific values, it uses the **gc_ReqService( )** function and passes a GC_PARM_BLK that contain the following parameter element:

IPSET_REG_INFO
    IPPARM_OPERATION_REGISTER
    - value = IP_REG_DELETE_BY_VALUE

Each H.323 alias or SIP Transport Address to be deleted is contained in an additional element in the GC_PARM_BLK that uses the IPSET_LOCAL_ALIAS set ID and the appropriate parameter ID for the address type.

H.323

Supported prefixes to be deleted from the registration are specified via GC_PARM_BLK elements that use the IPSET_SUPPORTED_PREFIXES set ID.

If the string that is contained in the value of the GC_PARM_BLK element matches a registered alias or supported prefix, it is deleted from the local database and an updated list is sent to the Gatekeeper.

SIP

A SIP application can specify the Registrar and Alias to modify information for by including the following parameter in the GC_PARM_BLK that is passed to **gc_ReqService( )**:

IPSET_REG_INFO
    IPPARM_REG_ADDRESS
- value = IP_REGISTER_ADDRESS structure with reg_client and reg_server fields filled in to indicate the desired Registrar address and Alias

Note: This parameter is optional. If it is not included in the GC_PARM_BLOCK, or if either of the addresses in the IP_REGISTER_ADDRESS structure is not supplied, the most recently used Registrar address and Alias are used by default.

If the GC_PARM_BLK does not contain any IPSET_LOCAL_ALIAS elements specifying Transport Addresses to be deleted, no bindings will be deleted and the function call has the same result as the query operation described in Section 4.27.2.6, “Querying Registration Information (SIP)”, on page 322.

If the GC_PARM_BLK contains an IPSET_LOCAL_ALIAS / IPPARM_ADDRESS_TRANSPARENT parameter element with the value `"*"`, all bindings that exist in the specified Registrar for the specified Alias are deleted, regardless of what application created them.

4.27.2.9 Deregistering

The Dialogic® Global Call API provides the ability to deregister from a Gatekeeper/Registrar. When deregistering, the application can decide whether to keep the registration information locally or delete it. To deregister, an application uses the **gc_ReqService( )** function and passes it a GC_PARM_BLK that contains the following element:

IPSET_REG_INFO
    IPPARM_OPERATION_DEREGISTER
    and one of the following values:
- IP_REG_MAINTAIN_LOCAL_INFO – keep the registration information locally
- IP_REG_DELETE_ALL – delete the local registration information

See Section 4.27.4.2, “Deregistration Example”, on page 329 for more information.

SIP

A SIP application can specify the Registrar and Alias to deregister by including the following parameter in the GC_PARM_BLK that is passed to **gc_ReqService**( ):

IPSET_REG_INFO
  IPPARM_REG_ADDRESS
  - value = IP_REGISTER_ADDRESS structure with reg_client and reg_server fields filled in to indicate the desired Registrar address and Alias

Note: This parameter is optional. If it is not included in the GC_PARM_BLOCK, or if either of the addresses in the IP_REGISTER_ADDRESS structure is not supplied, the most recently used Registrar address and Alias are used by default.

If the GC_PARM_BLK does not contain any IPSET_LOCAL_ALIAS elements specifying Transport Addresses to be deleted, all bindings previously created by this application for the specified Alias will be removed from the Registrar.

If the GC_PARM_BLK contains an IPSET_LOCAL_ALIAS / IPPARM_ADDRESS_TRANSPARENT parameter element with the value `"*"`, all bindings that exist in the specified Registrar for the specified Alias are deleted, regardless of what application created them.

4.27.3 Sending and Receiving Nonstandard Registration Messages (H.323)

The Dialogic® Global Call API provides the ability to send nonstandard messages to and receive nonstandard messages from the gatekeeper or registrar. To send nonstandard messages, the application uses the **gc_Extension( )** function. The first element must be set as described in Section 9.2.16, "IPSET_MSG_REGISTRATION", on page 628. Other elements are set as in conventional nonstandard messages; see Section 9.2.19, "IPSET_NONSTANDARDDATA", on page 631.

An unsolicited GCEV_EXTENSION event with an extension ID (ext_id) of IPEXTID_RECEIVEMSG can be received that contains a nonstandard registration message. The associated GC_PARM_BLK contains the message details in parameter elements as follows:

The parameter element that identifies the message type is:

IPSET_MSG_REGISTRATION
  IPPARM_MSGTYPE
  - value = IP_MSGTYPE_REG_NONSTD

The parameter element for the Nonstandard Data data is:

IPSET_NONSTANDARDDATA
  IPPARM_NONSTANDARDDATA_DATA
  - value = Nonstandard Data string, max length = max_parm_data_size (configurable at library start-up)

The parameter element for the Nonstandard Data identifier is one (and only one) of the following:

IPSET_NONSTANDARDDATA
  IPPARM_NONSTANDARDDATA_OBJID
  - value = array of unsigned integers, max length = MAX_NS_PARM_OBJID_LENGTH

IPSET_NONSTANDARDDATA
  IPPARM_H221NONSTANDARD
  - value = IP_H221NONSTANDARD structure

The maximum length of the Global Call parameter used for the Nonstandard Data information is configured at start-up via the max_parm_data_size field in the IPCCLIB_START_DATA structure. The default size is 255 (for backwards compatibility), but applications may configure it to be as large as 4096 bytes. Applications must use the extended **gc_util_..._ex( )** functions to insert or extract any GC_PARM_BLK parameter elements whose data length is defined to be greater than 255.

Note: In practice, applications may not be able to utilize the full maximum length of the nonstandard data parameter element as configured in max_parm_data_size. The H.323 stack limits the overall size of messages to be max_parm_data_size + 512 bytes, and any messages that exceed this limit are truncated without any notification to the application.

4.27.4 Registration Code Examples

This section contains code examples illustrating SIP registration and deregistration.

4.27.4.1 Registration Example

The following code example shows how to populate a GC_PARM_BLK structure that can be used to register an endpoint with a gatekeeper (H.323) or registrar (SIP). The GC_PARM_BLK structure contains the following registration information:

- two mandatory parameters required by the generic **gc_ReqService( )** function
- the protocol type (H.323 or SIP)
- the type of operation (register/deregister) and sub-operation (set information, add information, delete by value, delete all)
- the IP address to be registered
- the endpoint type to register as
- a number of local aliases
- a number of supported prefixes

```
int boardRegistration(IN LINEDEV boarddev, IN char protocol)
{

   GC_PARM_BLKP pParmBlock = NULL;
   int frc = GC_SUCCESS;

   if (protocol != IP_PROTOCOL_H323 && protocol != IP_PROTOCOL_SIP )
   {
      printf("failed bad protocol identifier.\n");
      return GC_ERROR;
   }
```

```
/****** Two (mandatory) elements that are not related directly to
 the server-client negotiation ********/

frc = gc_util_insert_parm_val(&pParmBlock,
                              GCSET_SERVREQ,
                              PARM_REQTYPE,
                              sizeof(char),
                              IP_REQTYPE_REGISTRATION);

frc = gc_util_insert_parm_val(&pParmBlock,
                              GCSET_SERVREQ,
                              PARM_ACK,
                              sizeof(char),
                              1);

/******Setting the protocol target***********/
frc = gc_util_insert_parm_val(&pParmBlock,
                              IPSET_PROTOCOL,
                              IPPARM_PROTOCOL_BITMASK,
                              sizeof(char),
                              protocol);  /*can be H323 or SIP*/

/****** Setting the operation to perform ***********/
frc = gc_util_insert_parm_val(&pParmBlock,
                              IPSET_REG_INFO,
                              IPPARM_OPERATION_REGISTER, /* can be Register or Deregister */
                              sizeof(char),
                              IP_REG_SET_INFO);  /* can be other relevant "sub" operations */

/****** Setting address information ***********/
IP_REGISTER_ADDRESS registerAddress;
memset(registerAddress, 0, sizeof(IP_REGISTER_ADDRESS));
strcpy(registerAddress.reg_server,"101.102.103.104"); /* set server address*/
if (protocol == IP_PROTOCOL_SIP)
{
   strcpy(registerAddress.reg_client,"user@10.20.30.40"); /* set alias for SIP*/
}

registerAddress.max_hops = regMulticastHops;
registerAddress.time_to_live = regTimeToLive;
frc = gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_REG_INFO,
                              IPPARM_REG_ADDRESS,
                              (UINT8)sizeof(IP_REGISTER_ADDRESS),
                              &registerAddress);

if (protocol == IP_PROTOCOL_H323)
{
   /**** SIP does not allow setting of these parm elements ****/

   /****** Setting endpoint type to GATEWAY ***********/
   gc_util_insert_parm_val(&pParmBlock,
                           IPSET_REG_INFO,
                           IPPARM_REG_TYPE,
                           (unsigned char)sizeof(EPType),
                           IP_REG_GATEWAY);

   /****** Setting supportedPrefixes information ***********/
   /****   This parm block may be repeated with different ****
    ****   supported prefixes and supported prefix types  ****/
   frc = gc_util_insert_parm_ref(&pParmBlock,
                                 IPSET_SUPPORTED_PREFIXES,
                                 (unsigned short)IPPARM_ADDRESS_PHONE,
                                 (UINT8)(strlen("011972")+1),
                                 "011972");
}
```

```
   /**** Setting terminalAlias information ****/
   /**** May repeat this line with different addresses and address types ****/
   frc = gc_util_insert_parm_ref (&pParmBlock,
                                 IPSET_LOCAL_ALIAS,
                                 (unsigned short)IPPARM_ADDRESS_EMAIL,
                                 (UINT8)(strlen("someone@someplace.com")+1),
                                 "someone@someplace.com");

   /****** Send the request ***********/
   unsigned long serviceID ;
   int rc = gc_ReqService(GCTGT_CCLIB_NETIF,
                         boarddev,
                         &serviceID,
                         pParmBlock,
                         NULL,
                         EV_ASYNC);

   if (rc != GC_SUCCESS)
   {
      printf("failed in gc_ReqService\n");
      return GC_ERROR;
   }

   gc_util_delete_parm_blk(pParmBlock);
   return GC_SUCCESS;
}



int boardUnregisterH323(IN char protocol)
{
   GC_PARM_BLKP pParmBlock = NULL;
   unsigned long serviceID = 1;
   int rc,frc;
   int gc_error;     // GC error code
   int cclibid;      // Call Control library ID for gc_ErrorValue
   long cc_error;    // Call Controll library error code
   char *resultmsg;  // String associated with cause code
   char *lib_name;   // Library name for cclibid

   if (protocol != IP_PROTOCOL_H323 && protocol != IP_PROTOCOL_SIP)
   {
      printf("failed bad protocol identifier.\n");
      return GC_ERROR;
   }

   gc_util_insert_parm_val (&pParmBlock,
                           IPSET_REG_INFO,
                           IPPARM_OPERATION_DEREGISTER,
                           sizeof(unsigned char),
                           IP_REG_DELETE_ALL);

   frc = gc_util_insert_parm_val(&pParmBlock,
                                GCSET_SERVREQ,
                                PARM_REQTYPE,
                                sizeof(unsigned char),
                                IP_REQTYPE_REGISTRATION);
   if (frc != GC_SUCCESS)
   {
      printf("failed in PARM_REQTYPE\n");
      return GC_ERROR;
   }
   frc = gc_util_insert_parm_val(&pParmBlock,
                                 GCSET_SERVREQ,
                                 PARM_ACK,
                                 sizeof(unsigned char),
                                 1);
   if (frc != GC_SUCCESS)
   {
      printf("failed in PARM_ACK\n");
      return GC_ERROR;
   }

   frc = gc_util_insert_parm_val(&pParmBlock,
                                 IPSET_PROTOCOL,
                                 IPPARM_PROTOCOL_BITMASK,
                                 sizeof(char),
                                 protocol); /* can be H323 or SIP */
   if (frc != GC_SUCCESS)
   {
      printf("failed in IPSET_PROTOCOL\n");
      return GC_ERROR;
   }

   rc = gc_ReqService(GCTGT_CCLIB_NETIF,
                      brddev,
                      &serviceID,
                      pParmBlock,
                      NULL,
                      EV_ASYNC);
   if ( GC_SUCCESS != rc)
   {
      printf("gc_ReqService failed while unregestering\n");
      if (gc_ErrorValue(&gc_error, &cclibid, &cc_error) != GC_SUCCESS)
      {
         printf("gc_Start() failed: Unable to retrieve error value\n");
      }
      else
      {
         gc_ResultMsg(LIBID_GC, (long) gc_error, &resultmsg);
         printf("gc_ReqService() failed: gc_error=0x%X: %s\n", gc_error, resultmsg);
         gc_ResultMsg(cclibid, cc_error, &resultmsg);
         gc_CCLibIDToName(cclibid, &lib_name);
         printf("%s library had error 0x%lx - %s\n", lib_name, cc_error, resultmsg);
      }
      gc_util_delete_parm_blk(pParmBlock);
      return GC_ERROR;
   }

   printf ("Unregister request to the GK was sent ...\n");
   gc_util_delete_parm_blk(pParmBlock);
   return GC_SUCCESS;
}
```

4.27.4.2 Deregistration Example

The following code example shows how to populate a GC_PARM_BLK structure that can be used to deregister an endpoint with a gatekeeper (H.323). The GC_PARM_BLK structure contains the following deregistration information:

- the type of operation (in this case, deregister) and sub-operation (do not retain the registration information locally)
- two mandatory parameters required by the generic **gc_ReqService( )** function
- the protocol type (in this case, H.323)

```
void unregister()
{
   GC_PARM_BLKP        pParmBlock = NULL;
   unsigned long       serviceID = 1;
   int                 rc,frc;
   int gc_error;           // GC error code
   int cclibid;            // Call Control library ID for gc_ErrorValue
   long cc_error;          // Call Controll library error code
   char *resultmsg;        // String associated with cause code
   char *lib_name;         // Library name for cclibid

   gc_util_insert_parm_val(&pParmBlock,
                           IPSET_REG_INFO,
                           IPPARM_OPERATION_DEREGISTER,
                           sizeof(unsigned char),
                           IP_REG_DELETE_ALL);

   frc = gc_util_insert_parm_val(&pParmBlock,
                                 GCSET_SERVREQ,
                                 PARM_REQTYPE,
                                 sizeof(unsigned char),
                                 IP_REQTYPE_REGISTRATION);

   if (frc != GC_SUCCESS)
   {
      printf("failed in PARM_REQTYPE\n");
      termapp();
   }

   frc = gc_util_insert_parm_val(&pParmBlock,
                                 GCSET_SERVREQ,
                                 PARM_ACK,
                                 sizeof(unsigned char),
                                 1);

   if (frc != GC_SUCCESS)
   {
      printf("failed in PARM_ACK\n");
      termapp();
   }

   frc = gc_util_insert_parm_val(&pParmBlock,
                                 IPSET_PROTOCOL,
                                 IPPARM_PROTOCOL_BITMASK,
                                 sizeof(char),
                                 IP_PROTOCOL_H323); /*can be H323, SIP or Both*/

   if (frc != GC_SUCCESS)
   {
      printf("failed in IPSET_PROTOCOL\n");
      termapp();
   }

   rc = gc_ReqService(GCTGT_CCLIB_NETIF,
                      brddev,
                      &serviceID,
                      pParmBlock,
                      NULL,
                      EV_ASYNC);

   if ( GC_SUCCESS != rc)
   {
      printf("gc_ReqService failed while unregestering\n");
      if (gc_ErrorValue(&gc_error, &cclibid, &cc_error) != GC_SUCCESS)
      {
         printf("gc_Start() failed:  Unable to retrieve error value\n");
      }
      else
      {
         gc_ResultMsg(LIBID_GC, (long) gc_error, &resultmsg);
         printf("gc_ReqService() failed:  gc_error=0x%X:  %s\n", gc_error, resultmsg);
         gc_ResultMsg(cclibid, cc_error, &resultmsg);
         gc_CCLibIDToName(cclibid, &lib_name);
         printf("%s library had error 0x%lx - %s\n", lib_name, cc_error, resultmsg);
      }
      gc_util_delete_parm_blk(pParmBlock);
      exit(0);
   }

   printf("Unregister request to the GK was sent ...\n");
   printf("the application will not be able to make calls !!! so it will EXIT\n");
   gc_util_delete_parm_blk(pParmBlock);
   return;
}
```

4.27.5 Gatekeeper Registration Failure (H.323)

Gatekeeper registration can fail for any one of several reasons, such as disconnecting the network cable, a network topology change that result in the loss of all paths to the Gatekeeper, a Gatekeeper failure, or a Gatekeeper shutdown. Terminals may not be immediately aware of the registration failure unless a RAS registration is attempted when the cable is disconnected, in which case the transaction fails immediately because of a socket bind failure. More typically, a RAS registration failure is only detected when either the Time To Live interval (programmable, with a default of 20 seconds) or the Response timeout (2 seconds) expires. RAS failure detection times can be improved by setting the Time To Live value in the RAS registration request to a value smaller than the default value, to 10 seconds, for example.

When RAS loses the Gatekeeper registration, all existing calls are automatically disconnected by Global Call, and GCEV_DISCONNECTED events are sent to the application. Calls in progress that are disconnected during RAS recovery are identified by a call control library result value of IPEC_RASReasonNotRegistered in the GCEV_DISCONNECTED event. All new calls are gracefully rejected and will continue to be rejected until RAS successfully registers with another Gatekeeper or explicitly unregisters and allows the H.323 stack to restart in manual RAS mode. The application can use the **gc_ReqService( )** function to perform the re-register or unregister operation.

All **gc_ReqService( )** function calls result in the return of either a GCEV_SERVICERESP (success) or GCEV_TASKFAIL (fail) completion event. If RAS registration fails (for example, as a result of an immediate socket bind failure or failure notification following a Time To Live timeout), the application receives a GCEV_TASKFAIL event. The range of applicable cause values for RAS-related GCEV_TASKFAIL events is IPEC_RASReasonMin to IPEC_RASReasonMax. The application must use the **gc_ReqService( )** function to reconfigure or register RAS in response to that event. If the RAS registration is rejected, the call control library is still cleaning up after the RAS registration failure and the application will receive another GCEV_TASKFAIL event, in which case it must issue **gc_ReqService( )** yet again.

It is recommended (but not required) that after receiving a GCEV_TASKFAIL event which identifies loss of Gatekeeper registration, the application should:

- stop attempting to make new calls, because this uses resources unnecessarily and slows down the cleanup time

- immediately issue a new RAS register or RAS unregister request

RAS registration requests should be made immediately on receipt of a RAS GCEV_TASKFAIL. Recovery from the loss of registration with the Gatekeeper is not completed until the call control library re-registers or attempts to unregister. Re-registration or unregistration is not attempted by the call control library until commanded by the application using the **gc_ReqService( )** function to issue a RAS REGISTER REQUEST or a RAS UNREGISTER SERVICE REQUEST respectively.

Note: The RAS GCEV_TASKFAIL event automatically repeats at intervals of 30 seconds if the application does not re-register with a Gatekeeper. This is done to remind the application that it must deal with the registration failure before it can successfully make or receive any new calls.

4.28 SIP Digest Authentication

Authentication is a process which allows a remote endpoint (a User Agent Server, or UAS) to verify the identity of a User Agent Client (UAC) that has sent a request to the UAS. If the UAS rejects a request with a 401 or 407 response, the UAC can re-send the request in a form that includes the sender's username and password to authenticate its identity. Once the UAC has authenticated its identity to the UAS, the UAS may require further verification that the UAC is authorized to make the original request, but that is a separate process from authentication. The standard type of SIP authentication is called "digest authentication", which refers to the encryption method used for secure transmission of the user's secret password in the message, and is documented in IETF RFC 2617.

To be able to respond automatically respond to authentication challenges, a UAC typically registers one or more triplets containing {realm, username, password}, where realm identifies the protected domain and the username and password identify the specific user. When a UAC receives a 401 or 407 response, it searches the triplets for a realm string that matches the one contained in the WWW-Authenticate or Proxy-Authenticate header field in the response. If it finds a matching realm string, it calculates a digest of the corresponding username and password strings and includes that result in the Authorization header field of the request it re-sends to the UAS.

The Global Call implementation of digest authorization extends this model to use quadruplets of {realm, identity, username, password}, where the identity represents the user's URI in the realm. This extension allows applications to either register a single username and password for a given realm, or multiple username/password pairs that are each associated with a different identity URI. For quadruplets that have an empty string as the identity element, the Dialogic® Global Call API library matching process uses the realm element only, exactly as if it were using a conventional authentication triplet instead of a quadruplet. If the identity element is a non-empty string, the library compares the identity string against the URI in the From header field of the 401/407 response. When the identity is non-empty, the library re-sends the request with the username/password digest only if both the realm and identity match the appropriate fields in the response message.

As an example, if the following header fields are received in a 401 Unauthorized response:

```
From: <sip:bob@example.com>;tag=0-13c4-4129f5f4-3bf3065a-7fc2
...
WWW-Authenticate: Digest realm="atlanta.com", domain="sip:ss1.carrier.com", qop="auth",
nonce="f84f1cec41e6cbe5aea9c8e88d359", opaque="", stale=FALSE, algorithm=MD5
```

both of the following quadruplets would be considered to be matches:

> {"atlanta.com", "sip:bob@example.com", "bob", "password1"}
>
> {"atlanta.com", "", "anonymous", ""}

Applications that require multiple identities per realm set multiple quadruplets with different, non-empty identity strings. Such applications may also set a default username and password by setting a quadruplet with an empty identity string. This default username/password is only used when a 401/407 response does not match the identity in any of the triplets for the given realm and may be an anonymous authentication as shown in the preceding example.

Applications that require only a single username/password pair per realm set only a single quadruplet with an empty identity string. In this case the application would not set any quadruplets that include non-empty identity strings.

Applications that wish to use the authentication mechanism should configure the desired authentication quadruplets before calling any function that may send a SIP request. Any 401 or 407 response that is received for a request that was sent before authentication quadruplets were configured causes the call/request to be terminated and not re-sent by Global Call even if an appropriate authentication quadruplet was configured in the interim. The reason code for such a termination is IPEC_SIPReasonStatus401Unauthorized or IPEC_SIPReasonStatus407ProxyAuthenticationRequired.

Digest authentication is supported for the following SIP message types:

- BYE
- INFO within a dialog
- INVITE and re-INVITE (subsequent INVITE within a dialog)
- NOTIFY within a dialog
- OPTIONS within a dialog
- REFER within a dialog
- REGISTER
- SUBSCRIBE

Authentication is specifically not supported for the following SIP message types:

- INFO outside of a dialog
- NOTIFY outside of a dialog
- OPTIONS outside of a dialog

Applications configure authentication quadruplets for virtual board by constructing a GC_PARM_BLK that contains a separate parameter element for each quadruplet, then calling the **gc_SetAuthenticationInfo( )** function with that parameter block. Authentication quadruplets are removed in the same way but using a different parameter ID in the parameter element. The same function call can configure or remove any number of quadruplets for a given virtual board by including the appropriate combination of parameter elements in the GC_PARM_BLK. For a given function call, each parameter in the GC_PARM_BLK should have a unique realm/identity pair; if multiple parameter elements have the same realm/identity pair, only the last of these elements in the parameter block becomes effective.

To add or modify an authentication quadruplet, the relevant set ID and parameter ID are:

IPSET_CONFIG
    IPPARM_AUTHENTICATION_CONFIGURE
    - value = IP_AUTHENTICATION data structure containing the desired quadruplet values. If the realm/identity pair is unique for the virtual board, a new quadruplet is added to the library's authentication database. If the realm/identity pair matches an existing quadruplet, the existing username/password pair is replaced by the new username/password pair.

To remove an existing authentication quadruplet, the relevant set ID and parameter ID are:

IPSET_CONFIG
    IPPARM_AUTHENTICATION_REMOVE
    - value = IP_AUTHENTICATION data structure that identifies the realm and identity of the quadruplet to be removed. The username and password elements of this structure are ignored. If the specified realm and identity do not match those of an existing quadruplet, the function call produces an IPERR_UNAVAILABLE error.

The elements of the authentication quadruplets are contained in an IP_AUTHENTICATION data structure, with each element having the following characteristics:

realm
    a case-insensitive string that defines the protected domain name. This element must always contain a non-empty string.

identity
    for a single-user realm, an empty string

    for a multi-user realm, either a case-insensitive string that identifies the user in the given realm, or else an empty string to allow specification of a default username/password pair. Non-empty strings must conform to the conventions for a SIP URI, and must begin with a "sip:" or "sips:" scheme

username
    a case-sensitive, null-terminated string that is the user's name. This element must always contain a non-empty string when configuring an authentication quadruplet. This value of this structure element is ignored when removing an authentication quadruplet.

password
    a case-sensitive, null-terminated string that is the user's secret password in clear text. This element can optionally be an empty string, for example, if the quadruplet contains an anonymous username. This value of this structure element is ignored when removing an authentication quadruplet.

When preparing to configure a quadruplet, the application should begin by initializing the IP_AUTHORIZATION structure with the **INIT_IP_AUTHORIZATION**( ) function, which configures the structure with the correct version number and with NULL string pointers for each element. The application should then populate each element with the desired string, including any empty strings. If any of the elements is left with a NULL pointer when passed to the function, the function call fails with IPERR_BAD_PARM.

Note that the **gc_SetConfigData( )** and **gc_SetUserInfo( )** functions **cannot** be used to configure authentication quadruplets. If a GC_PARM_BLK containing either of the authentication parameter IDs is passed to either of those functions, the function call fails with IPERR_BAD_PARM.

4.28.1 Registering Authentication Data Without Realm String

HMP software provides a method for registering authentication data without using realm string.

Previously, the realm element of the authentication quadruplet had to contain a non-empty string. This feature supports realm and identity as an empty string, so that identity authentication is verified using the same identity or username as the registrar server.

If the realm value is empty, the application uses the identity element to match with the identity provided by the server.

```
["",identity, username, password]
```

If realm and identity are empty, the application can use the username. The username must be the same username in the string identity provided by the registrar server.

```
["","", username, password]
```

For example, if the registrar server provides [bob@10.10.20.25] as identity, then "bob" must be configured as the username for the registrar server and the authentication quadruplet.

For more information about SIP Digest Authentication, refer to Section 4.28, “SIP Digest Authentication”, on page 332.

4.29 Using SIP Transport Layer Security (TLS)

The Dialogic® Global Call API library supports SIP Transport Layer Security (TLS), which is a security mechanism that operates on the Transport layer, on top of TCP transport. By using TLS as the connection transport, a SIP entity can send and receive SIP messages in a secure, authenticated manner.

The Global Call implementation of TLS is described in the following topics:

- Overview of TLS
- Configuring and Enabling TLS
- Making Calls Using TLS
- TLS Transport Failures

4.29.1 Overview of TLS

Internet Protocol security in general and Transport Layer Security in specific are very complex subjects, and a comprehensive discussion is well beyond the scope of this document. But anyone attempting to use TLS must have at least a basic understanding of the concepts and entities involved, and this brief introduction is intended to provide that foundation.

First we present definitions of a few key concepts:

Certificate
: A digital certificate is an electronic document which links a public key to a person or company in a public key infrastructure, enabling the user to send encrypted and digitally signed electronic messages. The certificate identifies the user and is required to verify his digital signature. The certificate contains information about the identity and public key of the person/company as well as the certificate's expiration date. Furthermore, the certificate may contain information about the usage of the certificate.

Certificate Authority (CA)
: A certificate Authority authorizes certificates by signing the contents using its private key. Certificate authorities are well known authorities, whose signatures are known and trusted. By signing other certificates, they act as a digital notary. Examples of CAs are VeriSign and DigiCert.

Diffie-Hellman (D-H) key exchange
: A cryptographic protocol that allows two parties who have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

Digital Signature Algorithm (DSA)
: DSA is used for creating and verifying digital signatures. It provides authentication, but cannot be used for encryption or secrecy.

Digital Signature Standard (DSS)
: DSS specifies Digital Signature Algorithm (DSA) appropriate for applications requiring digital signature.

PEM
: PEM specifies a base64-encoded certificate format.

Public Key Infrastructure
: The Public Key Infrastructure is the network security architecture of an organization. It includes software, encryption technologies, and services that enable secure transactions on the Internet, intranets, and extranets.

RSA
: RSA is public-key encryption technology developed by RSA Data Security, Inc. The acronym stands for Rivest, Shamir, and Adelman, the inventors of the technology.

Secure Socket Layer (SSL))
: SSL is a sophisticated encryption scheme that does not require the client and the server to arrange for a secret key to be exchanged between the client and server BEFORE the transaction is started. SSL uses public/private keys to provide a flexible encryption scheme that can be set up at the time of the secure transaction. A short tutorial on SSL is available at http://www.eventhelix.com/RealtimeMantra/Networking/SSL.pdf.

By using TLS as a connection transport, a SIP entity can send and receive data in a secure authenticated manner. TLS, together with the commonly used Public Key Infrastructure certification distribution mechanism achieves the following goals:

- Guarantees the identity of a remote computer
- Transmits messages to that remote computer in a secure encrypted manner

TLS uses pairs of asymmetrical encryptions keys to guarantee the identity of a remote computer. The public key of each remote computer is published in a certificate, which is a document digitally signed by a certificate authority. Both sides of the connection agree to trust the certificate (either directly or through a chain of intermediate trusted certificates) before the TLS connection establishment has started. In the TLS connection establishment process, the certificate of the remote computer is retrieved and verified and a new key and encryption algorithm is negotiated for the specific connection.

Establishment of a TLS connection is a three-phase process:

Phase 1: TCP connection establishment
: TLS uses TCP as its underlying transport protocol, so, a TLS handshake can start only after a TCP connection has reached the CONNECTED state.

Phase 2: TLS handshake
: The basic TLS handshake process consists of several TCP message exchanges between the client and the server, in which the client retrieves the server's certificate, verifies it, and negotiates an encryption key and algorithm for the session. Both parties also make sure that the security of the handshake has not been compromised. For more information on the TLS handshake see RFC 2246 and RFC 3546.

Phase 3: Post connection assertion
: In this phase, the client makes sure that the certificate handed to it by the server does indeed belong to server. This step is taken to prevent the situation in which a server named malise.com will present a valid certificate of someonelse.com.

After these three phases have been completed, encrypted messages can be transmitted on the connection in a secure manner.

RFC 3261 defines the use of TLS as a transport mechanism by using the "sips:" scheme. When using the "sips:" scheme in a URI (or any other header that indicates the next hop of a message, such as Route or Via) RFC 3261 mandates the transport to be TLS. For this reason, TLS cannot guarantee a secure delivery end-to-end, but only to the next hop.

The SIP stack used by Global Call uses an open source library called OpenSSL that provides TLS and encryption services. For more information about OpenSSL, refer to the OpenSSL project website at http://www.openssl.org. The list of ciphers supported by OpenSSL is available at http://www.openssl.org/docs/apps/ciphers.html.

A *digital certificate* is an electronic document which links a public key to a person or company in a public key infrastructure, enabling the user to send encrypted and digitally signed electronic messages. The certificate identifies the user and is required to verify his digital signature. The certificate contains information about the identity and public key of the person/company as well as the certificate's expiration date. Additionally, the certificate may contain information about the usage of the certificate.

Global Call only supports certificates that use the Privacy Enhanced Mail (PEM) format. Applications will need to convert other formats to PEM format. Similarly, Global Call only supports certificates that use the RSA or DSA key formats.

An example of an RSA certificate in PEM format is shown below.

```
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 8 (0x8)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: O=RVTEST, L=Metropolis, ST=New York, C=US, CN=udiw@radvision.com
        Validity
            Not Before: Apr 13 04:54:37 2003 GMT
            Not After : Apr 10 04:54:37 2013 GMT
        Subject: C=US, ST=New York, O=RVTEST, CN=127.0.0.1
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:dc:81:8f:86:b7:ff:cc:63:ff:6b:2b:bb:f2:d0:
                    21:71:bf:4f:ea:11:ac:b9:ce:6e:87:ef:ae:00:4f:
                    85:03:84:76:c9:25:1c:9f:33:43:a3:1a:96:a6:e8:
                    8d:35:f9:1a:e1:b9:90:b8:ee:15:2d:cc:47:6d:a9:
                    32:fa:75:fc:6c:ea:fd:c6:4b:cf:22:26:49:f6:46:
                    bb:99:e5:92:b7:d2:2f:22:f6:26:b2:5b:52:37:13:
                    70:78:df:09:e6:2f:d0:db:ee:aa:9e:a9:14:27:cb:
                    5a:38:5a:2a:de:4e:fa:63:7c:9a:67:6c:ac:8e:f1:
                    8a:63:d8:e9:82:0a:2d:71:7b
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
            CA:FALSE
            X509v3 Subject Key Identifier:
            82:2F:CE:D0:15:ED:C0:01:73:C1:2D:54:65:C0:6E:04:C2:28:FB:5D
            X509v3 Subject Alternative Name:
            DNS:localhost
    Signature Algorithm: md5WithRSAEncryption
        aa:df:5c:35:2d:d0:71:32:b8:9f:be:71:50:5d:e3:d0:90:68:
        ec:f7:9a:35:b3:19:61:fc:5d:c2:3a:4c:83:aa:67:de:50:a9:
        f5:60:ee:1a:16:10:26:2f:8a:e4:98:71:5a:06:8c:cc:59:02:
        b5:f4:88:12:e9:28:27:41:1e:de:07:61:56:2c:2a:7b:4c:6a:
        39:b0:10:d8:78:8f:e8:6d:7d:56:1f:48:5b:b5:79:9e:aa:be:
        a9:b4:1d:84:f9:4d:10:5c:fe:e1:6d:81:47:35:96:95:79:bb:
        76:33:88:a0:8f:22:1d:e0:c1:42:9e:9a:bd:83:0f:a0:ee:9c:
        d9:e0
```

```
-----BEGIN CERTIFICATE-----
MIICWzCCAcQCAQgwDQYJKoZIhvcNAQEEBQAwYzEPMA0GA1UEChMGUlZURVNUMRMw
EQYDVQQHEwpNZXRyb3BvbGlzMREwDwYDVQQIEwhOZXcgWW9yazELMAkGA1UEBhMC
VVMxGzAZBgNVBAMUEnVkaXdAcmFkdmlzaW9uLmNvbTAeFw0wMzA0MTMwNDU0Mzda
Fw0xMzA0MTAwNDU0MzdaMEUxCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhOZXcgWW9y
azEPMA0GA1UEChMGUlZURVNUMRIwEAYDVQQDEwkxMjcuMC4wLjEwgZ8wDQYJKoZI
hvcNAQEBBQADgY0AMIGJAoGBANyBj4a3/8xj/2sru/LQIXG/T+oRrLnObofvrgBP
hQOEdsklHJ8zQ6MalqbojTX5GuG5kLjuFS3MR22pMvp1/Gzq/cZLzyImSfZGu5nl
krfSLyL2JrJbUjcTcHjfCeYv0Nvuqp6pFCfLWjhaKt5O+mN8mmdsrI7ximPY6YIK
LXF7AgMBAAGjQjBAMAkGA1UdEwQCMAAwHQYDVR0OBBYEFIIvztAV7cABc8EtVGXA
bgTCKPtdMBQGA1UdEQQNMAuCCWxvY2FsaG9zdDANBgkqhkiG9w0BAQQFAAOBgQCq
31w1LdBxMrifvnFQXePQkGjs95o1sxlh/F3COkyDqmfeUKn1YO4aFhAmL4rkmHFa
BozMWQK19IgS6SgnQR7eB2FWLCp7TGo5sBDYeI/obX1WH0hbtXmeqr6ptB2E+U0Q
XP7hbYFHNZaVebt2M4igjyId4MFCnpq9gw+g7pzZ4A==
-----END CERTIFICATE-----
```

A private key is used to decipher the information encrypted by the public key in the certificate. An example of private key in PEM format is shown below.

```
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQDcgY+Gt//MY/9rK7vy0CFxv0/qEay5zm6H764AT4UDhHbJJRyf
M0OjGpam6I01+RrhuZC47hUtzEdtqTL6dfxs6v3GS88iJkn2RruZ5ZK30i8i9iay
W1I3E3B43wnmL9Db7qqeqRQny1o4WireTvpjfJpnbKyO8Ypj2OmCCi1xewIDAQAB
AoGAc4HCx1U3P7/aGi+sooL4IfePSxO7IRHLwJWC1lLRYIhRGQjBt3tJIPVamVAU
OIOm2zszXkwI+BacDAun0p9ffE1NQaIyihpoTxMThYhQgmpVUdtsz0UqRhMFB+o+
Glf236M2fQr4nTdHvW8OVAhUzGQf7yfR48Ntx6ekjf2B6BECQQD6vSTUApa9UPGD
cPUzEaoCNergUPdM6G72+Gls9NSI73AHYBGA97ba23gah/hBjmjdziF0UxmPAP4Q
KgP1haCXAkEA4SIFJIwg4v8fUIIp9KmhM60RAT+diqLJ90AJPy4x3aLM38YJuRfk
F30ALePuR7MkvuP994GnsfUg9cWjzENuvQJBAO+QN9e4gX1wENCc5Cle/ygNi9O2
iBGbIiolPdU0Nrx+yHLDfvXRt4tzlVUEBFXeUqNZhu01WH4hXJzlB9NVURECQCaT
y8nNcT00dks3YrUX9BWEzGsoWXiOGImToYIACm9uHCkkKDpdS6pysvsqGYSTv/It
4zDsOK4X0QQMT9sKmwkCQAZN7GNJ8QSgwMgwDw4hkmu+GHUXLC6cF68xdUurA2Gc
olCWLrdlpqJSUzp1XHXff/oqEygwjNmPbVujES09c4w=
-----END RSA PRIVATE KEY-----
```

A *Certificate Authority* (CA) authorizes certificates by signing the contents using its private key. Certificate Authorities are well known authorities, whose signatures are known and trusted. By signing other certificates, they act as a digital notary. A number of commercial CAs are available, such as VeriSign and Thawte, and there are also some free CAs, such as www.cacert.org. For test purposes, or for a case where the links to be secured will be local calls that use the local CA, it is also possible for a system to install its own CA, using OpenSSL for example.

It often occurs that a client will not accept a certificate supplied by a server because the certificate is signed by an intermediate CA which is not known to the client. The client typically states that the validity of the certificate cannot be verified. In such cases, a chained SSL certificate or certificate group may allow the client to accept the server's certificate by connecting it back to a CA that is known and trusted by the client.

A *certificate chain* is a sequence of certificates, where each certificate in the chain is signed by the subsequent certificate. The purpose of certificate chain is to establish a chain of trust from a peer certificate to a trusted Certification Authority (CA) certificate. The CA vouches for the identity in the peer certificate by signing it. If the CA is one that you trust (indicated by the presence of a copy of the CA certificate in your root certificate directory), this implies you can trust the signed peer certificate as well.

To illustrate a certificate chain, we show three fictional example certificates: root.pem, serverCA.pem and server.pem.

First, the root.pem certificate. Note that the certificate is self-signed and X509v3 Basic Constrains shows CA:TRUE.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 10 (0xa)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, ST=New Jersey, O=dialgic.com, CN=hmfu-
rootCA.dialogic.com/emailAddress=h.fu@dialogic.com
        Validity
            Not Before: Nov 21 17:36:28 2005 GMT
            Not After : Nov 21 17:36:28 2006 GMT
        Subject: C=US, ST=New Jersey, O=dialogic.com, CN=hmfu-
rootCA.dialogic.com/emailAddress=h.fu@dialogic.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:bc:42:8d:5b:d5:7c:9b:ad:bd:46:3a:63:04:8a:
                    6a:7b:5e:c3:79:15:cd:4e:83:13:64:ac:3c:dd:ea:
                    7a:34:51:7f:ce:b1:3b:3d:42:a9:d1:98:9a:88:76:
                    c4:4e:7b:d6:33:db:81:95:4a:01:92:49:5e:f1:bb:
                    45:47:f9:be:18:f9:af:5d:7b:61:39:78:56:28:31:
                    bd:e8:ef:06:09:44:f8:33:bb:4d:f3:43:fe:7d:18:
                    88:80:0c:38:fb:be:36:ac:00:1f:74:84:da:fd:3d:
                    d4:48:05:21:aa:e8:db:1c:0d:86:33:ed:c7:bd:55:
                    b8:08:e7:53:7c:ad:67:7f:ed
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: sha1WithRSAEncryption
        37:58:76:91:f3:cf:66:5b:01:43:d6:d4:76:dc:ae:a3:71:37:
        47:ee:f3:a9:db:10:27:da:9f:7e:69:b4:79:d1:36:6e:ab:16:
        a8:4a:70:b2:a3:f2:f9:38:a7:90:c4:1c:65:bc:9e:e9:7d:5d:
        38:50:6b:9f:f3:05:82:f2:20:cb:74:45:ca:53:ce:fb:0a:7f:
        60:b8:c0:be:1a:52:fb:70:88:a8:99:6b:a8:d5:c7:56:d6:a9:
        59:3d:fb:4b:cf:0f:3e:08:64:7e:ee:40:76:24:3e:61:8a:00:
        af:a3:fa:a5:67:b0:23:c2:40:4b:03:bc:ff:1b:ce:46:94:55:
        e5:a7
```

```
-----BEGIN CERTIFICATE-----
MIICcDCCAdmgAwIBAgIBCjANBgkqhkiG9w0BAQUFADB1MQswCQYDVQQGEwJVUzET
MBEGA1UECBMKTmV3IEplcnNleTESMBAGA1UEChMJaW50ZWwuY29tMR4wHAYDVQQD
ExVobWZ1LXJvb3RDQS5pbnRlbC5jb20xHTAbBgkqhkiG9w0BCQEWDmguZnVAaW50
ZWwuY29tMB4XDTA1MTEyMTE3MzYyOFoXDTA2MTEyMTE3MzYyOFowdTELMAkGA1UE
BhMCVVMxEzARBgNVBAgTCk5ldyBKZXJzZXkxEjAQBgNVBAoTCWludGVsLmNvbTEe
MBwGA1UEAxMVaG1mdS1yb290Q0EuaW50ZWwuY29tMR0wGwYJKoZIhvcNAQkBFg5o
LmZ1QGludGVsLmNvbTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAvEKNW9V8
m629RjpjBIpqe17DeRXNToMTZKw83ep6NFF/zrE7PUKp0ZiaiHbETnvWM9uBlUoB
kkle8btFR/m+GPmvXXthOXhWKDG96O8GCUT4M7tN80P+fRiIgAw4+742rAAfdITa
/T3USAUhqujbHA2GM+3HvVW4COdTfK1nf+0CAwEAAaMQMA4wDAYDVR0TBAUwAwEB
/zANBgkqhkiG9w0BAQUFAAOBgQA3WHaR889mWwFD1tR23K6jcTdH7vOp2xAn2p9+
abR50TZuqxaoSnCyo/L5OKeQxBxlvJ7pfV04UGuf8wWC8iDLdEXKU877Cn9guMC+
GlL7cIiomWuo1cdW1qlZPftLzw8+CGR+7kB2JD5higCvo/qlZ7AjwkBLA7z/G85G
lFXlpw==
-----END CERTIFICATE-----
```

Next we show serverCA.pem. Note that this certificate is signed by the root certificate and X509v3 Basic Constrains shows CA:TRUE.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 11 (0xb)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, ST=New Jersey, O=dialogic.com, CN=hmfu-
rootCA.dialogic.com/emailAddress=h.fu@dialogic.com
        Validity
            Not Before: Nov 21 17:40:29 2005 GMT
            Not After : Nov 21 17:40:29 2006 GMT
        Subject: C=US, ST=New Jersey, O=dialogic.com, CN=hmfu-
serverCA.dialogic.com/emailAddress=h.fu@dialogic.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:f8:8a:34:ea:95:34:66:23:aa:31:4d:62:47:52:
                    8b:e5:8e:f2:0b:87:36:db:d6:d6:5c:49:3f:d6:93:
                    4d:9c:06:26:df:cb:e1:11:64:ac:10:35:71:91:79:
                    22:e1:75:c9:f0:33:c5:1b:67:8b:5f:3e:bc:21:7c:
```

```
                    0c:1a:f7:e5:bc:00:44:dc:1b:36:17:5c:49:04:a0:
                    a5:6a:d2:99:31:d6:24:a4:76:93:94:69:e2:80:a9:
                    d2:fa:e9:fd:b6:dc:80:17:f2:12:62:1e:46:e8:83:
                    4a:82:d8:b0:60:a3:6c:5e:60:c0:73:f4:dd:50:db:
                    9d:16:a0:92:51:67:5d:a5:31
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: sha1WithRSAEncryption
        58:f1:e4:37:45:96:e5:fd:9e:96:58:57:79:08:69:35:6f:da:
        af:16:21:0b:2f:87:d3:37:85:e2:93:6c:0d:fc:7f:25:17:4e:
        af:93:1a:53:57:69:bb:58:e6:0e:a4:05:ef:a9:3a:16:27:e4:
        e5:a8:01:54:cb:c6:46:17:47:3d:98:7f:af:19:10:1e:6a:15:
        b0:93:c2:4a:12:c1:30:fb:46:ba:31:76:6f:51:71:4b:67:2e:
        d3:31:64:58:d4:0a:b7:14:a6:95:9a:2c:b8:f9:a5:f3:8d:56:
        13:11:bf:76:5e:16:d9:be:91:de:12:3f:e4:e5:62:96:4d:d7:
        6c:f4

-----BEGIN CERTIFICATE-----
MIICcjCCAdugAwIBAgIBCzANBgkqhkiG9w0BAQUFADB1MQswCQYDVQQGEwJVUzET
MBEGA1UECBMKTmV3IEplcnNleTESMBAGA1UEChMJaW50ZWwuY29tMR4wHAYDVQQD
ExVobWZ1LXJvb3RDQS5pbnRlbC5jb20xHTAbBgkqhkiG9w0BCQEWDmguZnVAaW50
ZWwuY29tMB4XDTA1MTEyMTE3NDAyOVoXDTA2MTEyMTE3NDAyOVowdzELMAkGA1UE
BhMCVVMxEzARBgNVBAgTCk5ldyBKZXJzZXkxEjAQBgNVBAoTCWludGVsLmNvbTEg
MB4GA1UEAxMXaG1mdS1zZXJ2ZXJDQS5pbnRlbC5jb20xHTAbBgkqhkiG9w0BCQEW
DmguZnVAaW50ZWwuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQD4ijTq
lTRmI6oxTWJHUovljvILhzbb1tZcST/Wk02cBibfy+ERZKwQNXGReSLhdcnwM8Ub
Z4tfPrwhfAwa9+W8AETcGzYXXEkEoKVq0pkx1iSkdpOUaeKAqdL66f223IAX8hJi
Hkbog0qC2LBgo2xeYMBz9N1Q250WoJJRZ12lMQIDAQABoxAwDjAMBgNVHRMEBTAD
AQH/MA0GCSqGSIb3DQEBBQUAA4GBAFjx5DdFluX9npZYV3kIaTVv2q8WIQsvh9M3
heKTbA38fyUXTq+TGlNXabtY5g6kBe+pOhYn5OWoAVTLxkYXRz2Yf68ZEB5qFbCT
wkoSwTD7Rroxdm9RcUtnLtMxZFjUCrcUppWaLLj5pfONVhMRv3ZeFtm+kd4SP+Tl
YpZN12z0
-----END CERTIFICATE-----
```

Finally, the server.pem certificate file. Note that this certificate is signed by the serverCA certificate and X509v3 Basic Constraints shows CA:FALSE.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 12 (0xc)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, ST=New Jersey, O=dialogic.com, CN=hmfu-
serverCA.dialogic.com/emailAddress=h.fu@dialogic.com
        Validity
            Not Before: Nov 21 17:42:38 2005 GMT
            Not After : Nov 21 17:42:38 2006 GMT
        Subject: C=US, ST=New Jersey, O=dialogic.com, CN=hmfu-
mobile.dialogic.com/emailAddress=h.fu@dialogic.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:ad:71:3b:df:99:78:5d:8c:f2:e5:22:5d:6d:90:
                    38:37:d8:7e:25:6f:b7:2b:18:c7:6b:dd:c4:f1:af:
                    06:55:04:d6:e3:fc:a1:ed:35:59:a9:c1:73:b6:c4:
                    71:2f:75:3d:5c:ce:61:80:a6:f1:53:3f:35:f3:4d:
                    58:07:ee:ae:9d:ce:b5:30:13:2e:7a:6a:24:75:be:
                    95:6a:8b:25:33:e6:4a:f4:4a:19:9f:03:d7:09:d3:
                    83:b2:de:8c:2d:8c:8e:79:a3:f3:d8:07:12:80:0e:
                    68:5a:35:dd:53:f1:0b:02:32:fa:1f:93:fe:64:61:
                    d4:7b:9e:f7:6a:eb:98:19:99
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Subject Key Identifier:
                E6:56:D3:2E:8F:7D:5B:04:99:D6:B0:C9:4C:54:A2:0B:33:31:67:FD
            X509v3 Authority Key Identifier:
                DirName:/C=US/ST=New Jersey/O=dialogic.com/CN=hmfu-
rootCA.dialogic.com/emailAddress=h.fu@dialogic.com
                serial:0B

            Netscape CA Revocation Url:
                https://www.sial.org/ca-crl.pem
    Signature Algorithm: sha1WithRSAEncryption
        47:52:fa:c6:77:7f:9c:7e:f2:8c:df:4c:21:2e:57:2a:a8:14:
        06:72:aa:fb:68:8d:90:f8:c3:5c:4b:07:b4:60:c9:21:26:a1:
        f9:b4:de:0e:09:4c:93:14:1b:4c:e8:af:49:1c:48:c7:6d:33:
        06:5d:b6:a3:fd:c3:f5:09:41:2b:0c:20:71:3c:2d:92:2e:32:
        7a:a0:d1:00:ea:49:ee:7a:14:8e:06:f5:e3:16:92:b4:85:ab:
        3a:04:65:7a:d4:65:9d:6d:f4:65:d7:d4:49:b1:4f:a8:8e:0a:
        49:ec:fc:7e:0a:ca:31:62:f7:7d:72:64:fb:6c:de:0c:c1:d7:
        f2:a8

-----BEGIN CERTIFICATE-----
MIIDSzCCArSgAwIBAgIBDDANBgkqhkiG9w0BAQUFADB3MQswCQYDVQQGEwJVUzET
MBEGA1UECBMKTmV3IEplcnNleTESMBAGA1UEChMJaW50ZWwuY29tMSAwHgYDVQQD
ExdobWZ1LXNlcnZlckNBLmludGVsLmNvbTEdMBsGCSqGSIb3DQEJARYOaC5mdUBp
bnRlbC5jb20wHhcNMDUxMTIxMTc0MjM4WhcNMDYxMTIxMTc0MjM4WjB1MQswCQYD
VQQGEwJVUzETMBEGA1UECBMKTmV3IEplcnNleTESMBAGA1UEChMJaW50ZWwuY29t
MR4wHAYDVQQDExVobWZ1LW1vYmlsZS5pbnRlbC5jb20xHTAbBgkqhkiG9w0BCQEW
DmguZnVAaW50ZWwuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCtcTvf
mXhdjPLlIl1tkDg32H4lb7crGMdr3cTxrwZVBNbj/KHtNVmpwXO2xHEvdT1czmGA
pvFTPzXzTVgH7q6dzrUwEy56aiR1vpVqiyUz5kr0ShmfA9cJ04Oy3owtjI55o/PY
BxKADmhaNd1T8QsCMvofk/5kYdR7nvdq65gZmQIDAQABo4HoMIHlMAkGA1UdEwQC
MAAwHQYDVR0OBBYEFOZW0y6PfVsEmdawyUxUogszMWf9MIGIBgNVHSMEgYAwfqF5
pHcwdTELMAkGA1UEBhMCVVMxEzARBgNVBAgTCk5ldyBKZXJzZXkxEjAQBgNVBAoT
CWludGVsLmNvbTEeMBwGA1UEAxMVaG1mdS1yb290Q0EuaW50ZWwuY29tMR0wGwYJ
KoZIhvcNAQkBFg5oLmZ1QGludGVsLmNvbYIBCzAuBglghkgBhvhCAQQEIRYfaHR0
cHM6Ly93d3cuc2lhbC5vcmcvY2EtY3JsLnBlbTANBgkqhkiG9w0BAQUFAAOBgQBH
UvrGd3+cfvKM30whLlcqqBQGcqr7aI2Q+MNcSwe0YMkhJqH5tN4OCUyTFBtM6K9J
HEjHbTMGXbaj/cP1CUErDCBxPC2SLjJ6oNEA6knuehSOBvXjFpK0has6BGV61GWd
bfRl19RJsU+ojgpJ7Px+CsoxYvd9cmT7bN4MwdfyqA==
-----END CERTIFICATE-----
```

A *Certificate Revocation List* (CRL) contains a list of all the revoked certificates a CA has issued that have yet to expire. When a certificate is revoked, the CA declares that the certificate should no longer be trusted. OpenSSL support both Version 1 and Version 2 CRLs.

An example of a CRL file in PEM format is shown as following.

```
Certificate Revocation List (CRL):
        Version 1 (0x0)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: /CN=hmfu-serverCA.dialogic.com/C=US/ST=New
Jersey/L=Parsippany/O=dialogic.com/emailAddress=h.fu@dialogic.com
        Last Update: Nov 16 16:17:08 2005 GMT
        Next Update: Dec 16 16:17:08 2005 GMT
Revoked Certificates:
    Serial Number: DD862A284475A685
        Revocation Date: Nov 16 16:15:44 2005 GMT
    Signature Algorithm: sha1WithRSAEncryption
        c7:de:1f:5c:0a:cc:ae:90:45:89:6d:35:3d:2c:ad:8b:cb:10:
        06:8b:ce:49:6a:4a:65:9f:c8:fd:16:6a:6e:5c:e4:d5:d4:7b:
        fd:3f:bd:88:24:bd:5d:f0:98:47:40:8f:50:87:53:50:9d:8e:
        1b:42:7c:87:d7:23:96:2d:7f:f4:fa:50:6d:a3:88:3f:e4:57:
        0a:e3:f3:40:3c:f7:82:5d:14:62:5d:86:0f:ce:72:80:56:b1:
        a6:af:7e:be:70:3c:7a:5a:18:c3:de:79:cf:b1:38:46:a7:f4:
```

```
        9b:5e:b3:85:92:7c:bb:c8:c9:93:fd:98:fa:e6:54:39:5b:58:
        37:1c
-----BEGIN X509 CRL-----
MIIBcDCB2jANBgkqhkiG9w0BAQUFADCBjDEgMB4GA1UEAxMXaG1mdS1zZXJ2ZXJD
QS5pbnRlbC5jb20xCzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpOZXcgSmVyc2V5MRMw
EQYDVQQHEwpQYXJzaXBwYW55MRIwEAYDVQQKEwlpbnRlbC5jb20xHTAbBgkqhkiG
9w0BCQEWDmguZnVAaW50ZWwuY29tFw0wNTExMTYxNjE3MDhaFw0wNTEyMTYxNjE3
MDhaMBwwGgIJAN2GKihEdaaFFw0wNTExMTYxNjE1NDRaMA0GCSqGSIb3DQEBBQUA
A4GBAMfeH1wKzK6QRYltNT0srYvLEAaLzklqSmWfyP0Wam5c5NXUe/0/vYgkvV3w
mEdAj1CHU1CdjhtCfIfXI5Ytf/T6UG2jiD/kVwrj80A894JdFGJdhg/OcoBWsaav
fr5wPHpaGMPeec+xOEan9Jtes4WSfLvIyZP9mPrmVDlbWDcc
-----END X509 CRL-----
```

Global Call applications can act as either a TLS server or a TLS client.

TCP or TLS connections that are opened to Global Call are referred to as server connections. Generally, server connections should be closed by the party that initiated the connection. Server connections are not reusable by other calls or standalone transactions outside of calls. Server connections should be terminated by the initiator when no transaction is using it.

TCP or TLS connections that are opened by Global Call are referred to as client connections. The persistence of TLS client connections is configurable using the same mechanism that sets the persistence of TCP connections.

The Dialogic® Global Call API library implements a *TLS engine*, which binds together a complete set of parameters related to TLS operation. Each virtual board in a system is configured with its own TLS engine, which identifies the TLS port number, the certificate, private key and optional certificate chains that will be used when the library is acting as a TLS server, and one or more trusted root certificate authorities (CAs) that will be used when the library will be acting as a TLS client.

4.29.2 Configuring and Enabling TLS

TLS is configured and enabled separately for each virtual board in the system through the IP_VIRTBOARD data structures that configure each virtual board. As with other IP features that are configured and enabled via IP_VIRTBOARD, the configuration of this feature cannot be changed at run-time; the values that are contained in IP_VIRTBOARD when **gc_Start( )** is called remain in effect until the system is stopped and the application restarted.

There are several specific steps required to configure and enable TLS, in addition to the initial step of allocating and initializing the IP_VIRTBOARD structure and the final step of including the IP_VIRTBOARD structures in the IPCCLIB_START_DATA structure that is passed to **gc_Start( )**, which are common to all features that are configured via IP_VIRTBOARD. The feature-specific steps are discussed in the following sections:

- Allocating, Initializing, and Configuring a SIP_TLS_ENGINE Data Structure
- Enabling TCP in IP_VIRTBOARD
- Configuring TCP/TLS Persistence in IP_VIRTBOARD
- Enabling TLS in IP_VIRTBOARD

4.29.2.1 Allocating, Initializing, and Configuring a SIP_TLS_ENGINE Data Structure

The process of configuring the TLS feature for a virtual board begins by allocating a SIP_TLS_ENGINE data structure and initializing it to default values using the **INIT_SIP_TLS_ENGINE( )** function.

After the SIP_TLS_ENGINE structure is initialized, it must be configured for TLS client operation, TLS server operation, or both. The default values in the structure do not set the minimum configuration for either server or client operation. If an initialized but unconfigured SIP_TLS_ENGINE structure is referenced in an IP_VIRTBOARD structure that is passed to **gc_Start( )**, the library start operation will fail.

Changing the Default TLS Port Number

The default values set in SIP_TLS_ENGINE by the initialization function specify port number 5061 as the TLS port (the default UDP and TCP ports are 5060). The default value is valid and only needs to be changed if the application specifically requires a different port number. The port number is specified in the sip_tls_port field of the structure.

Configuring for Local Certificates for TLS Server Operation

To configure a virtual board to operate as a TLS server, the application must configure an RSA certificate and/or a DSS certificate in the SIP_TLS_ENGINE structure. In either case, the certificate and its associated key should be issued by a CA and should identify the local host name. The TLS engine can hold one of each type certificate, and Global Call will report the appropriate one to a remote UA depending on the cipher selected during the TLS handshake.

One or both of the local certificate/key pairs must be configured if Global Call will be operating as a TLS server. If Global Call will be operating as a TLS client, it will need to configure one or both local certificates (and optionally a certificate chain) to support mutual authentication.

For either type of certificate, the application must configure three items:

- private key filename—the name of the file that contains the private key, either an RSA key for the RSA certificate or a DSS certificate for a DSA certificate. In either case, the file may be in plain text format or may be encrypted.
- private key password—the password string that is required to use the private key if the private key file is encrypted. If the private key for either certificate is not encrypted, the corresponding password field in SIP_TLS_ENGINE should be left at its default NULL value.
- certificate filename—the name of the file that contains the certificate that identifies the local host name

Configuring a Certificate Chain

In addition to the local certificates, applications can optionally configure the a certificate chain using the chain_cert_number and chain_cert_filename fields. A certificate chain configuration is typically necessary if the local certificate is issued by an intermediate CA rather than a root CA. Note that the TLS engine contains only a single certificate chain, which is appended to both the

RSA and DSS certificates. Application cannot use different certificate chains for RSA and DSS certificates at the same time.

Each member of the chain_cert_filename array identifies a single certificate in the chain that links the local certificate to the root CA. The order of the chain certificates must start with the intermediate certificate that issues the local certificate. The next certificate in the chain is the one that issued the previous certificate and so on until the root CA certificate is reached. For example, if root.pem signs serverCA1.pem, and serverCA1.pem signs serverCA2.pem, and serverCA2.pem signs server.pem, then chain_cert_number should be set to 2, chain_cert_filename[0] should point to serverCA2.pem, and chain_cert_filename[1] should point to serverCA1.pem.

Configuring CA Certificates for TLS Client Operation

To configure a virtual board to operate as a TLS server, the application must configure an array of one or more CA certificates in the SIP_TLS_ENGINE structure using the ca_cert_number and ca_cert_filename fields.

The ca_cert_filename field identifies as an array of one or more root CA certificates which it trusts. The ca_cert_number field identifies the number of certificates in the array.

If a TLS client application needs to support mutual authentication, it will also need to configure the one or both local certificate/private key pairs, and optionally a certificate chain. During mutual authentication, the client needs to identify itself to the server in the same way that a server identifies itself to a client.

Configuring Certificate Revocation Lists (CRLs)

An application may optionally configure the library to use one or more Certificate Revocation List (CRL) files via the crl_number and crl_filename fields. In this configuration crl_filename is an array that contains one or more files in PEM format; the size of the array is crl_number. When one or more CRLs have been configured, Global Call consults these CRLs to decide whether the certificate has been revoked when it examines incoming certificates.

Configuring the Cipher Suite

An application may optionally configure the local cipher suite that is used to negotiate encryption algorithms with the remote UA. The local_cipher_suite field is a list of ciphers that is specified as a specially formatted string defined by OpenSSL. OpenSSL allows for several keywords in the elist, which are shortcuts for sets of ciphers. Details of the cipher list and keywords can be found in openSSL manual page at http://www.openssl.org/docs/apps/ciphers.html.

Note: The local_cipher_suite field is a pointer to the formatted string itself rather than the name of the file that contains the string.

The default value of local_cipher_suite is NULL which uses OpenSSL's default string "ALL:!ADH:+RC4:@STRENGTH".

Configuring Diffie-Hellman (D-H) Key Exchange Parameters

In order to perform a Diffie-Hellman (D-H) key exchange the server must use a D-H group (D-H parameters) and generate a D-H key. As TLS server, Global Call always generates a new D-H key during the negotiation. dh_param_512_filename should points to a PEM-format file that contains D-H parameters with 512-bit key, and dh_param_1024_filename should points to a PEM-format file that contains D-H parameters with 1024-bit key. If the application does not provide D-H parameters, Global Call uses the pre-built default D-H parameters for D-H key exchange ciphers.

Note that the non-ephemeral D-H modes are currently unimplemented in OpenSSL because there is no support for D-H certificates.

Configuring Server Session Caching

An application may optionally enable server session caching by setting session_id string. If the string is set, Global Call enables session caching on server side and supplies a session identifier to the client during handshake. During a new handshake, if session_id in ClientHello is non-empty, Global Call looks up the session cache for a match and resumes a session if possible. Server session cache terminates when Global Call closes. The session timeout is not configurable and is set at 300 seconds.

The default value of session_id is NULL, and in this case the server returns an empty session_id to indicate that the session will not be cached and therefore cannot be resumed.

Note that *client* session caching is not supported in Global Call because Global Call already supports client connection persistency (see Section 4.29.2.3, "Configuring TCP/TLS Persistence in IP_VIRTBOARD", on page 349) so that multiple calls can share the same TLS connection whenever possible. Server session caching may provide a benefit to remote UA which does not support client connection persistency and wishes to re-establish TLS connection every time and resume TLS session if possible.

Setting the Mutual Authentication Option

The E_client_cert_required field determines whether or not the Dialogic® Global Call API library will require the client to present its certificate for mutual authentication during a TLS handshake when the library is acting as TLS server. If the client fails to present its certificate or if the certificate verification fails, the TLS handshake will fail. Mutual authentication is only required if the application sets this field to the value ENUM_Enabled; the default value is ENUM_Disable.

Setting the Insecure Port Blocking Options

To prevent downgrade attack, Global Call allows applications to optionally block the local UDP and/or TCP ports by configuring the block_udp_port and block_tcp_port fields in SIP_TLS_ENGINE. When either port is blocked, both send and receive on that port are disabled and the application may not make calls or receive calls on that port. If both the UDP and TCP ports are blocked, only the TLS port (the default TLS port is 5061) can be used as the secure port for sending and receiving SIP messages.

In both cases, the default value set by INIT_SIP_TLS_ENGINE( ) is ENUM_Disabled, which leaves both the UDP and TCP ports open. If the application wishes to block either or both of the ports, it must set the value ENUM_Enabled in the appropriate field or fields.

Simple SIP_TLS_ENGINE Configuration Example

The following code sample illustrates how an application might set up a simple TLS configuration:

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..
..

/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data */
INIT_IPCCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);

/* initialize virtual board */
INIT_IP_VIRTBOARD(&virtBoards[0]);

/* initialize TLS Engine */

SIP_TLS_ENGINE sip_tls_engine;
INIT_SIP_TLS_ENGINE(&sip_tls_engine);

sip_tls_engine.local_rsa_private_key_filename = "localhost.rsa-key-cert.pem";
sip_tls_engine.local_rsa_cert_filename = "localhost.rsa-key-cert.pem";
sip_tls_engine.ca_cert_number = 1;
sip_tls_engine.ca_cert_filename[0] = "cacert.pem";

/* configure virtual board TLS engine pointer */
virtBoard[0].sip_tls_engine = &sip_tls_engine
```

Advanced SIP_TLS_ENGINE Configuration Example

The following code sample illustrates a more sophisticated TLS configuration:

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..
..

/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data */
INIT_IPCCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);

/* initialize virtual board */
INIT_IP_VIRTBOARD(&virtBoards[0]);
```

```
/* initialize TLS Engine */

SIP_TLS_ENGINE sip_tls_engine;
INIT_SIP_TLS_ENGINE(&sip_tls_engine);

/* change default port number */
sip_tls_engine.sip_tls_port = 5062;

/* configure local RSA certificate and key */
sip_tls_engine.local_rsa_private_key_filename = "localhost.rsa-key-cert.pem";
sip_tls_engine.local_rsa_private_key_password = "RSAKeyPassword";
sip_tls_engine.local_rsa_cert_filename = "localhost.rsa-key-cert.pem";

/* configure local DSS certificate and key */
sip_tls_engine.local_dss_private_key_filename = "localhost.dss-key-cert.pem";
sip_tls_engine.local_dss_private_key_password = "DSSKeyPassword";
sip_tls_engine.local_dss_cert_filename = "localhost.dss-key-cert.pem";

/* configure two root certficates */
sip_tls_engine.ca_cert_number = 2;
sip_tls_engine.ca_cert_filename = (char**)calloc(sip_tls_engine.ca_cert_number,sizeof(char*));
sip_tls_engine.ca_cert_filename[0] = "cacert1.pem";
sip_tls_engine.ca_cert_filename[1] = "cacert2.pem";

/* configure two chain certificates */
sip_tls_engine.chain_cert_number = 2;
sip_tls_engine.chain_cert_filename =
      (char**)calloc(sip_tls_engine.chain_cert_number,sizeof(char*));
sip_tls_engine.chain_cert_filename[0] = "chaincert1.pem";
sip_tls_engine.chain_cert_filename[0] = "chaincert2.pem";

/* configure one CRL */
sip_tls_engine.crl_number = 1;
sip_tls_engine.crl_filename = (char**)calloc(sip_tls_engine.crl_number,sizeof(char*));
sip_tls_engine.crl_filename[0] = "crl.pem";

/* configure local cipher list to be exportable, sorted with key strength */
sip_tls_engine.local_cipher_suite = "EXP:@STRENGTH";

/* configure DH parameters */
sip_tls_engine.dh_param_512_filename = "dh512_param.pem";
sip_tls_engine.dh_param_1024_filename = "dh1024_param.pem";

/* enable server session cache by setting session id string */
sip_tls_engine.session_id = "HMP Media Server";

/* enable mutual authentication, disable UDP and TCP ports */
sip_tls_engine.E_client_cert_required = ENUM_Enabled;
sip_tls_engine.E_block_udp_port = ENUM_Disabled;
sip_tls_engine.E_block_tcp_port = ENUM_Disabled;

/* configure virtual board TLS engine pointer */
virtBoard[0].sip_tls_engine = &sip_tls_engine
```

4.29.2.2 Enabling TCP in IP_VIRTBOARD

The TLS security mechanism operates on top of the TCP protocol, support for which is optional in Global Call. It is therefore necessary to enable the TCP protocol in IP_VIRTBOARD by setting the E_SIP_tcpenabled field to ENUM_Enabled. If an IP_VIRTBOARD structure which contains a SIP_TLS_ENGINE structure but which does not enable TCP is passed to **gc_Start( )**, the library initialization will fail.

4.29.2.3 Configuring TCP/TLS Persistence in IP_VIRTBOARD

Because TLS operates on top of TCP, the Global Call mechanism for configuring the persistence of TCP connections also affects TLS connections. This configuration is accomplished via the E_SIP_Persistence field in IP_VIRTBOARD as described in Section 4.1.2, "Configuring SIP Transport Protocol", on page 102 and Section 4.1.2.1, "Configuring TCP Transport", on page 103.

The default persistency is ENUM_PERSISTENCE_TRANSACT_USER, which means that the TLS client connection will be reused among calls, registrations, and other standalone transactions if possible. Reusing the TLS client connection will save TLS connection time between the same source and destination addresses and port numbers. When no one uses a TLS client connection, it will be terminated by Global Call, and the TLS client connection is therefore kept alive only if someone is using it.

If the application sets ENUM_PERSISTENCE_TRANSACT as the persistence, a TLS client connection is terminated as soon as the SIP transaction is terminated. This means that multiple TLS client connections may be required within the same SIP call. This persistence setting is therefore not recommended for performance reasons.

In the case where an outbound proxy is configured with valid IP address, Global Call will try to establish a persistent TCP or TLS client connection to the outbound proxy IP address during library start up. Note that an outbound proxy name can not be used to resolve to an IP address in either TCP or TLS during Global Call start up. (This is a limitation only during start up time; during run time, an outbound proxy name *can* be used to resolve to IP address.) If TLS is configured as outbound proxy transport, the outbound proxy name must be configured to verify certificate identify during Global Call start up, otherwise the persistent client connection can not be established.

If established, this persistent TCP or TLS client connection could then be reused by all outgoing/incoming SIP messages to/from the proxy. This persistent TCP or TLS client connection will be kept alive until Global Call closes, regardless of the E_SIP_Persistence setting in the IP_VIRTBOARD structure.

4.29.2.4 Enabling TLS in IP_VIRTBOARD

The final step in the process of configuring and enabling TLS is to include the configured SIP_TLS_ENGINE data structure in the sip_tls_engine field of IP_VIRTBOARD.

If this sip_tls_engine field references a SIP_TLS_ENGINE structure that is not properly configured for either TLS server or TLS client operation, the library will fail to load when **gc_Start( )** is called. In this case the error will be reported as IPERR_INVALID_TLS_PARAM.

The library will also fail to load when **gc_Start( )** is called if TLS is enabled but the TCP protocol is not enabled via E_SIP_tcpenabled because TLS operates on top of TCP. In this case, the reported error will be IPERR_INVALID_TLS_WITHOUT_TCP.

4.29.3 Making Calls Using TLS

RFC 3261 defines the use of TLS as a transport mechanism by using the “sips:” scheme. When using the “sips:” scheme in a URI (or in any other header that indicates the next hop of a message, such as Route, Via, and others), RFC 3261 mandates the transport to be TLS. This is the reason why TLS will not guarantee a secure delivery end-to-end, but only to the next hop.

There are several different scenarios of how a Global Call application can originate a call using TLS. These include:

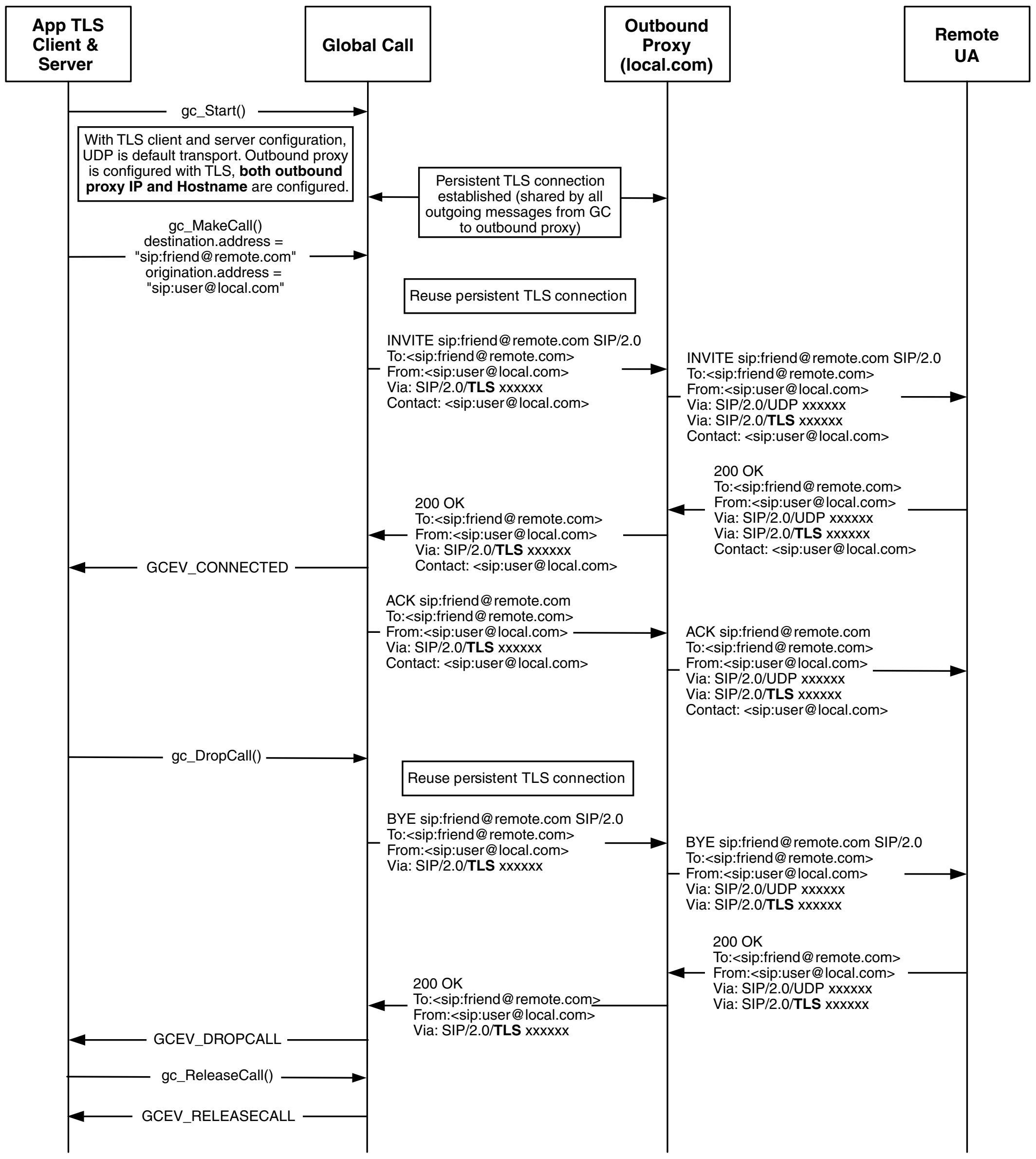
- Outbound proxy transport configured to be TLS
- Source address is “sip:” URI, destination address is “sip:” URI
- Source address is “sip:” URI, destination address is “sips:” URI
- Source address is “sips:” URI, destination address is “sip:” URI
- Source address is “sips:” URI, destination address is “sips:” URI

Outbound proxy transport configured to be TLS

When an outbound proxy is enabled, the transport protocol is determined by the E_SIP_OutboundProxyTransport field in the IP_VIRTBOARD structure. If the application wishes to use TLS transport, it must set the outbound proxy transport in IP_VIRTBOARD to the value ENUM_TLS. The transport method is independent of the URI scheme of the destination address.

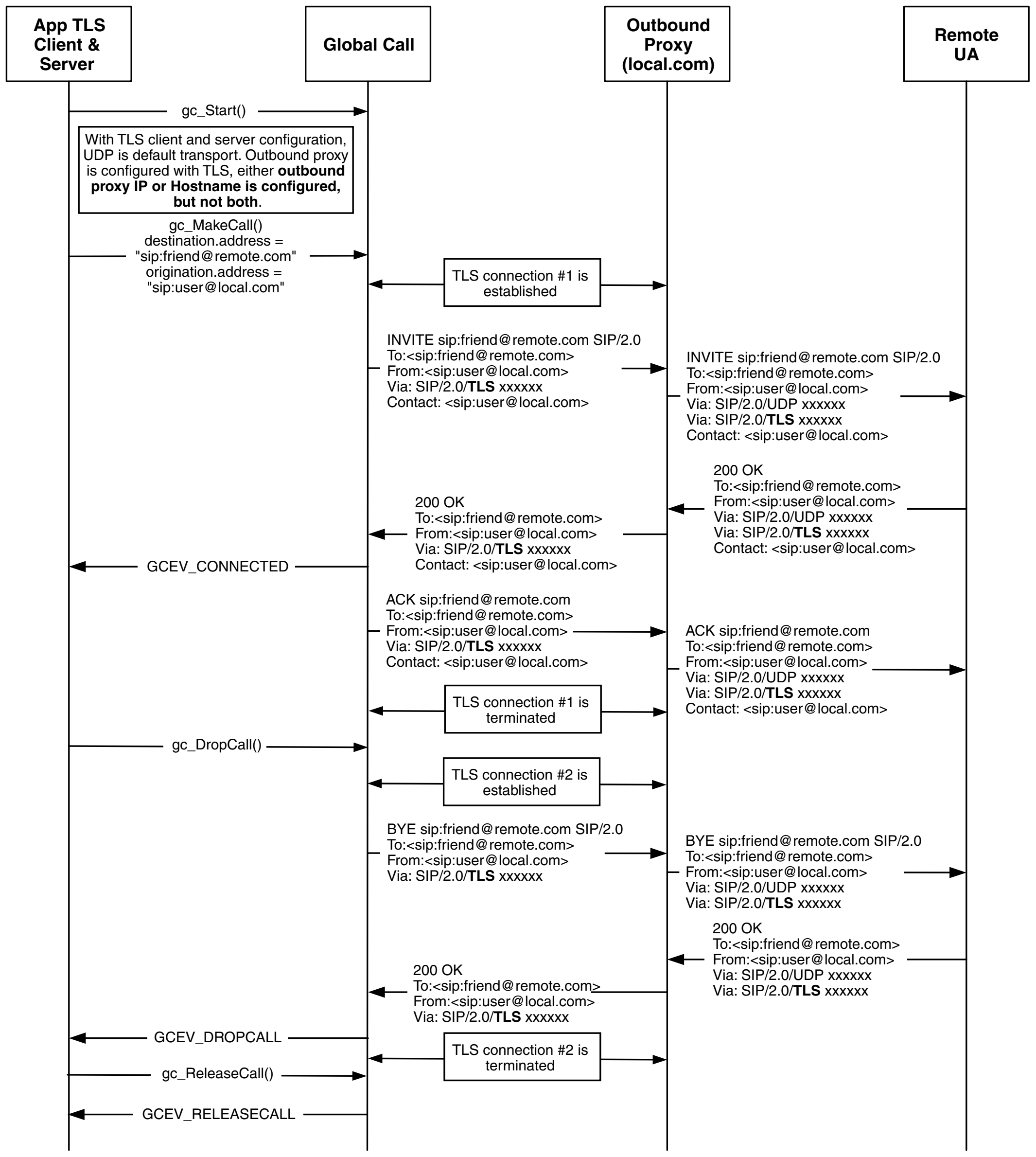
In the following scenario the outbound proxy transport is configured for TLS transport for all outgoing requests. With both the outbound proxy IP address and hostname configured, a persistent TLS connection will be established and reused for all subsequent outgoing messages.

Figure 53. Outbound Proxy Configured for TLS Transport with Both IP and Hostname


App TLS Client & Server
Global Call
Outbound Proxy (local.com)
Remote UA
gc_Start()
With TLS client and server configuration, UDP is default transport. Outbound proxy is configured with TLS, both outbound proxy IP and Hostname are configured.
Persistent TLS connection established (shared by all outgoing messages from GC to outbound proxy)
gc_MakeCall() destination.address = "sip:friend@remote.com" origination.address = "sip:user@local.com"
Reuse persistent TLS connection
INVITE sip:friend@remote.com SIP/2.0 To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
INVITE sip:friend@remote.com SIP/2.0 To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/UDP xxxxxx Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
200 OK To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/UDP xxxxxx Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
200 OK To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
GCEV_CONNECTED
ACK sip:friend@remote.com To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
ACK sip:friend@remote.com To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/UDP xxxxxx Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
gc_DropCall()
Reuse persistent TLS connection
BYE sip:friend@remote.com SIP/2.0 To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/TLS xxxxxx
BYE sip:friend@remote.com SIP/2.0 To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/UDP xxxxxx Via: SIP/2.0/TLS xxxxxx
200 OK To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/UDP xxxxxx Via: SIP/2.0/TLS xxxxxx
200 OK To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/TLS xxxxxx
GCEV_DROPCALL
gc_ReleaseCall()
GCEV_RELEASECALL

The following scenario also illustrates a case where the outbound proxy is configured for TLS transport, but here the proxy is only identified by one of the two means (that it, only by IP address or by hostname, but not both). In this case, there will be no persistent TLS connection established. Instead, a TLS connection will be established for the SIP transaction. This TLS connection will be reused only as long as some transaction is using it and will be terminated when no transaction is using it.

Figure 54. Outbound Proxy Configured for TLS Transport with Only IP Address Or Hostname


App TLS Client & Server
Global Call
Outbound Proxy (local.com)
Remote UA
gc_Start()
With TLS client and server configuration, UDP is default transport. Outbound proxy is configured with TLS, either outbound proxy IP or Hostname is configured, but not both.
gc_MakeCall() destination.address = "sip:friend@remote.com" origination.address = "sip:user@local.com"
TLS connection #1 is established
INVITE sip:friend@remote.com SIP/2.0 To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
INVITE sip:friend@remote.com SIP/2.0 To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/UDP xxxxxx Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
200 OK To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/UDP xxxxxx Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
200 OK To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
GCEV_CONNECTED
ACK sip:friend@remote.com To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
ACK sip:friend@remote.com To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/UDP xxxxxx Via: SIP/2.0/TLS xxxxxx Contact: <sip:user@local.com>
TLS connection #1 is terminated
gc_DropCall()
TLS connection #2 is established
BYE sip:friend@remote.com SIP/2.0 To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/TLS xxxxxx
BYE sip:friend@remote.com SIP/2.0 To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/UDP xxxxxx Via: SIP/2.0/TLS xxxxxx
200 OK To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/UDP xxxxxx Via: SIP/2.0/TLS xxxxxx
200 OK To:<sip:friend@remote.com> From:<sip:user@local.com> Via: SIP/2.0/TLS xxxxxx
GCEV_DROPCALL
TLS connection #2 is terminated
gc_ReleaseCall()
GCEV_RELEASECALL

Source address is “sip:” URI, destination address is “sip:” URI

In this scenario, the transport protocol of an initial INVITE is decided as follows:

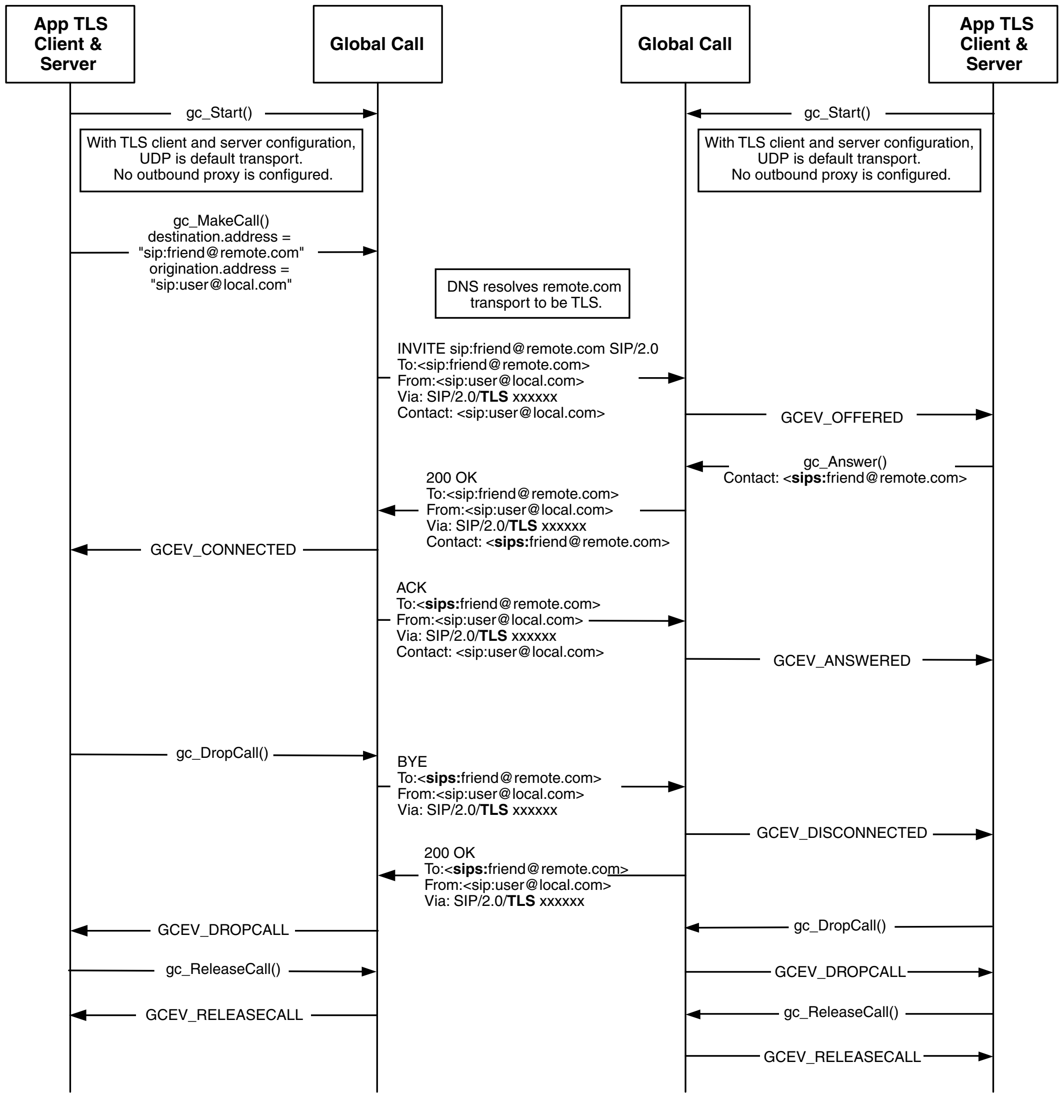
1. If the E_SIP_DefaultTransport field in the IP_VIRTBOARD structure is ENUM_UDP, the actual transport protocol depends on DNS lookup, as defined by RFC3263. Global Call automatically matches the remote UA’s supported protocols with local supported protocols. The final transport for the initial INVITE may be TLS, TCP, or UDP.

2. If the E_SIP_DefaultTransport field in the IP_VIRTBOARD structure is ENUM_TCP, only TCP will be used as transport protocol. No TLS will be used in the initial INVITE.

Global Call will always use a “sip:” URI as the local contact URI unless this is specifically changed by the application.

The following figure illustrates an initial INVITE with TLS where both source and destination addresses use the “sip:” scheme. In this case, the DNS resolves the required transport to be TLS. Note that in the 200 OK to the INVITE, the Contact URI has changed to the “sips:” scheme, which causes the subsequent ACK and BYE transactions to use TLS transport.

Figure 55. TLS with “sip:” Source Address and “sip:” Destination Address

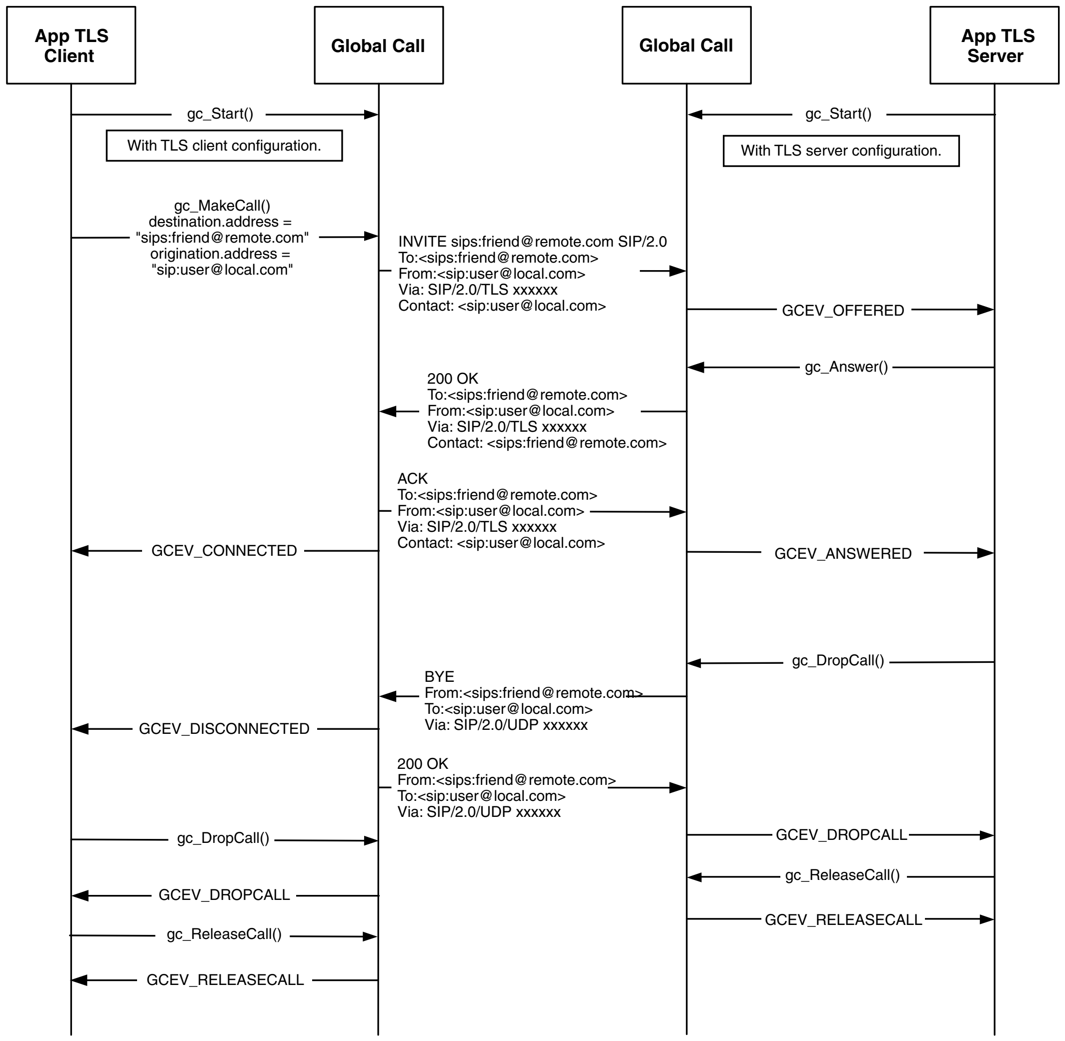

App TLS Client & Server
Global Call
Global Call
App TLS Client & Server
gc_Start()
With TLS client and server configuration, UDP is default transport. No outbound proxy is configured.
gc_Start()
With TLS client and server configuration, UDP is default transport. No outbound proxy is configured.
gc_MakeCall() destination.address = "sip:friend@remote.com" origination.address = "sip:user@local.com"
DNS resolves remote.com transport to be TLS.
INVITE sip:friend@remote.com SIP/2.0
To:<sip:friend@remote.com>
From:<sip:user@local.com>
Via: SIP/2.0/TLS xxxxxx
Contact: <sip:user@local.com>
GCEV_OFFERED
gc_Answer() Contact: <sips:friend@remote.com>
200 OK
To:<sip:friend@remote.com>
From:<sip:user@local.com>
Via: SIP/2.0/TLS xxxxxx
Contact: <sips:friend@remote.com>
GCEV_CONNECTED
ACK
To:<sips:friend@remote.com>
From:<sip:user@local.com>
Via: SIP/2.0/TLS xxxxxx
Contact: <sip:user@local.com>
GCEV_ANSWERED
gc_DropCall()
BYE
To:<sips:friend@remote.com>
From:<sip:user@local.com>
Via: SIP/2.0/TLS xxxxxx
GCEV_DISCONNECTED
200 OK
To:<sips:friend@remote.com>
From:<sip:user@local.com>
Via: SIP/2.0/TLS xxxxxx
GCEV_DROPCALL
gc_DropCall()
gc_ReleaseCall()
GCEV_DROPCALL
GCEV_RELEASECALL
gc_ReleaseCall()
GCEV_RELEASECALL

Source address is “sip:” URI, destination address is “sips:” URI

In this scenario, the transport protocol of initial INVITE is TLS and Global Call acts as TLS client. Global Call will always use a “sip:” URI as local contact URI unless the application specifically changes it, which means that the subsequent incoming request message should use UDP because local URI is “sip:”.

The following figure illustrates an initial INVITE transaction where TLS is specified via a “sips:” URI as the destination address. Note that the BYE transaction is UDP because the source address is given as a “sip:” URI.

Figure 56. TLS with “sip:” Source Address and “sips:” Destination Address

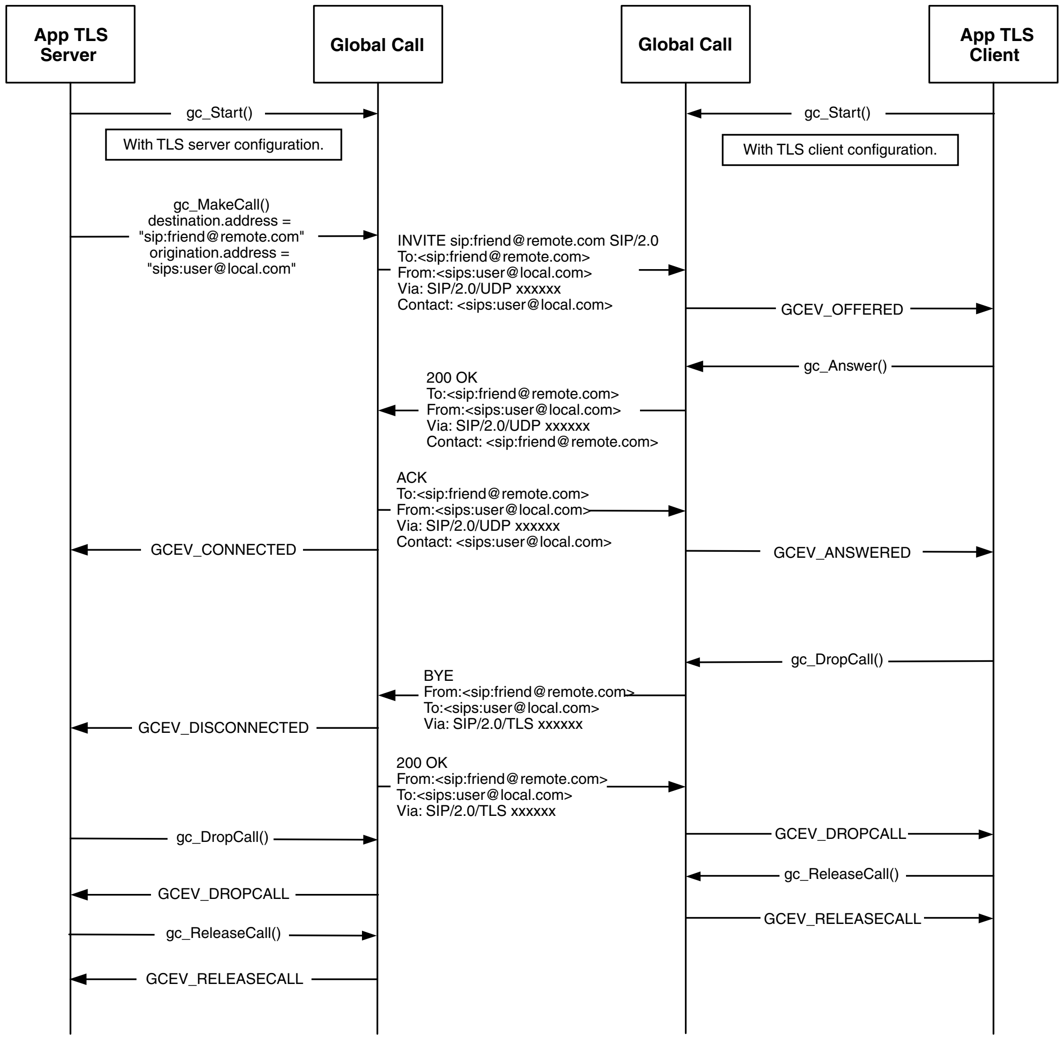


Source address is “sips:” URI, destination address is “sip:” URI

In this scenario, the transport protocol of an initial INVITE is determined using the same process as in the “sip:” source/“sip:” destination case. Global Call will always use a “sips:” URI as the local contact URI unless the application specifically changes it, which means that the subsequent incoming request message should use TLS because the local URI is “sips:”

The following figure illustrates the scenario for an initial INVITE where the source address is a “sips:” URI but the destination is a “sip:” URI. In this case, the initial INVITE is UDP because of the “sip:” destination address but the BYE is TLS because of the “sips:” URI in the source address.

Figure 57. TLS with “sips:” Source Address and “sip:” Destination Address

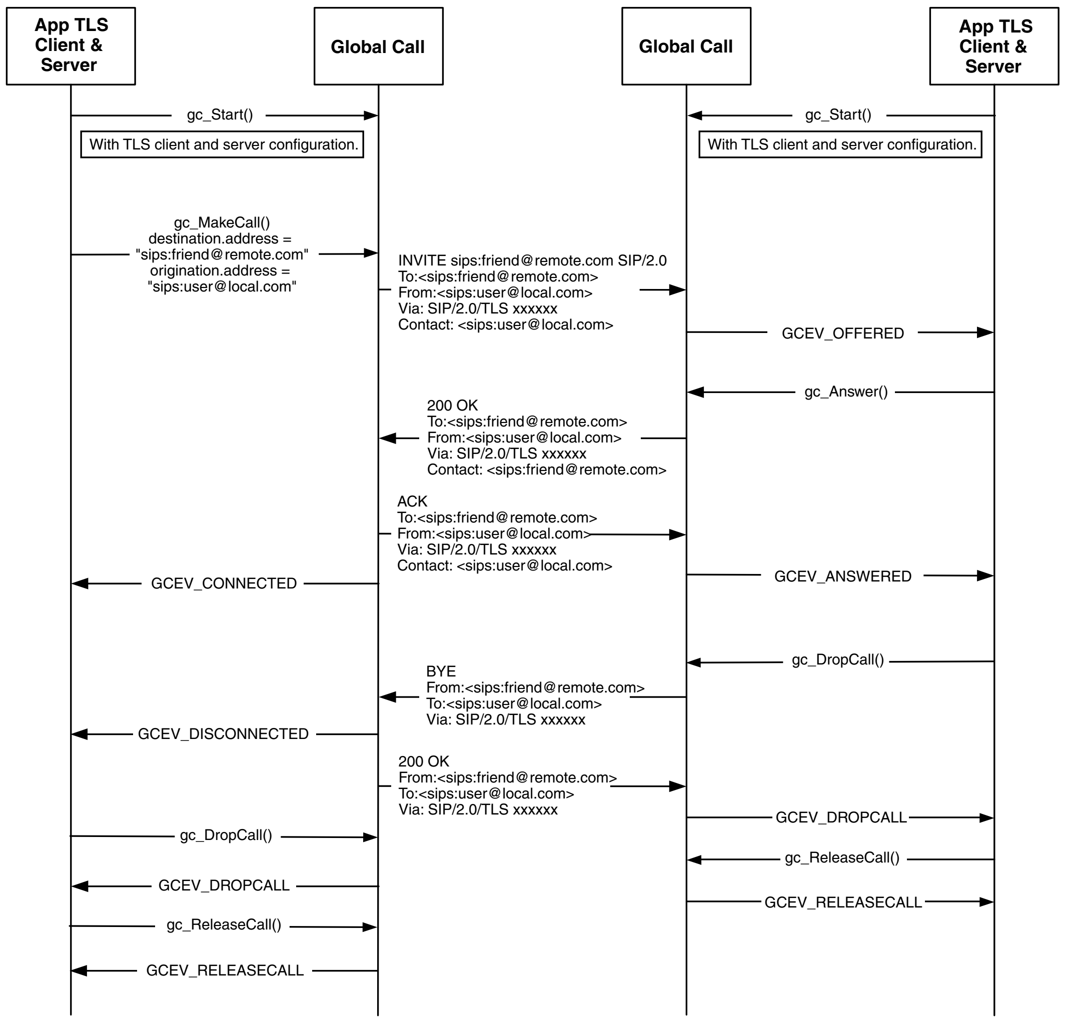

App TLS Server
Global Call
Global Call
App TLS Client
gc_Start()
gc_Start()
With TLS server configuration.
With TLS client configuration.
gc_MakeCall() destination.address = "sip:friend@remote.com" origination.address = "sips:user@local.com"
INVITE sip:friend@remote.com SIP/2.0
To:<sip:friend@remote.com>
From:<sips:user@local.com>
Via: SIP/2.0/UDP xxxxxx
Contact: <sips:user@local.com>
GCEV_OFFERED
gc_Answer()
200 OK
To:<sip:friend@remote.com>
From:<sips:user@local.com>
Via: SIP/2.0/UDP xxxxxx
Contact: <sip:friend@remote.com>
ACK
To:<sip:friend@remote.com>
From:<sips:user@local.com>
Via: SIP/2.0/UDP xxxxxx
Contact: <sips:user@local.com>
GCEV_CONNECTED
GCEV_ANSWERED
gc_DropCall()
BYE
From:<sip:friend@remote.com>
To:<sips:user@local.com>
Via: SIP/2.0/TLS xxxxxx
GCEV_DISCONNECTED
200 OK
From:<sip:friend@remote.com>
To:<sips:user@local.com>
Via: SIP/2.0/TLS xxxxxx
gc_DropCall()
GCEV_DROPCALL
GCEV_DROPCALL
gc_ReleaseCall()
gc_ReleaseCall()
GCEV_RELEASECALL
GCEV_RELEASECALL

Source address is “sips:” URI, destination address is “sips:” URI

In this scenario, the transport protocol of an initial INVITE is TLS and Global Call acts as TLS client and server. Global Call will always use a “sips:” URI as the local contact URI unless the application specifically changes it, which means that the subsequent incoming request message should use TLS because the local URI is “sips:”.

The following figure illustrates a scenario where all transactions are TLS because both the source address and destination address are “sips:” URIs.

Figure 58. TLS with “sips” Source Address and “sips:” Destination Address



After the initial INVITE transaction, the transport of subsequent request message to the remote UA depends on the Contact URI scheme from the remote response or request messages. This means that the remote UA has the freedom to change the Contact URI to be either a “sip:” or “sips:” URI independent of initial INVITE request. Global Call will try to use the required transport depending on the remote Contact URI scheme in conjunction with default transport protocol and any outbound proxy configuration. Global Call also allows the application to change the contact URI scheme to be different than the initial INVITE by setting the complete contact header as described in Section 4.9.5, "Setting SIP Header Fields for Outbound Messages", on page 185.

For out-of-call request messages, such as REGISTER, OPTIONS, INFO, and SUBSCRIBE/NOTIFY, the transport method depends on the destination address URI scheme as well as the default transport protocol and any outbound proxy configuration.

4.29.4 TLS Transport Failures

If a TLS connection fails to establish due to a timeout, a network error, or some other reason, Global Call notifies the application with a GCEV_EXTENSION event with extension ID IPEXTID_RECEIVEMSG. The metadata for this extension event will contain a parameter of type IPSET_SIP_REQUEST_ERROR / IPPARM_SIP_SVC_UNAVAIL whose data is a REQUEST_ERROR data structure with the error code IP_SIP_REQUEST_NETWORK_ERROR.

Note: Unlike TCP connection failure, Global Call will not retry the transaction using UDP if the TLS connection fails, regardless of the value of the E_SIP_RequestRetry parameter in the IP_VIRTBOARD structure.

Common causes of connection failure include:

- remote UA does not support TLS
- TLS negotiation fails
- post-connection assertion fails (TLS negotiation succeeds but remote TLS certificate hostname does not match request URI hostname)
- certificate has been revoked or is outdated

4.30 Accessing the Transport Layer IP Address of an Inbound SIP Call

The application has the ability to examine the SIP transport layer IP address of an inbound SIP call. This feature provides a method to determine the authenticity of the originator, and to immediately detect unauthorized requests.

The IPPARM_SIP_TRANSPORT_ADDR parameter ID in the IPSET_CALLINFO set ID supports this feature, and allows receipt of the transport layer IP address of an inbound SIP call in a decimal-dotted notation. The parameter ID, when used with the GCEV_OFFERED and GCEV_REQ_MODIFY_CALL events, extracts the remote SIP transport layer address upon receipt of incoming INVITE and re-INVITE requests.

The IP_SIP_TRANSPORT_ADDR structure contains the remote SIP transport layer address.

4.30.1 Enabling Receipt of SIP Transport Address Information

By default, the underlying SIP stack is not enabled to retrieve incoming SIP transport address message information elements. You must set the sip_msginfo_mask value to

IP_SIP_TRANSADDR_ENABLE in IP_VIRTBOARD for each IPT virtual board device prior to calling **gc_Start( )**.

Note: Currently, the only way to turn off the receipt of the SIP transport address information received by the underlying SIP stack is by removing the IP_SIP_TRANSADDR_ENABLE from the sip_msginfo_mask; then stop and restart the application.

Example

```
IP_VIRTBOARD virtBoard[MAX_BOARDS];
memset(virtBoard,0,sizeof(IP_VIRTBOARD) * MAX_BOARDS);
bid = 1;
INIT_IP_VIRTBOARD(&virtBoard[bid]);
 // fill up other board parameters
 ….
 virtBoard[bid].localIP.ip_ver = IPVER4;
 virtBoard[bid].localIP.u_ipaddr.ipv4 = (unsigned int) IP_CFG_DEFAULT;
 …
 virtBoard[1].sip_msginfo_mask |= IP_SIP_TRANSADDR_ENABLE;
 //Then use the virtBoard structure in the gc_Start() function appropriately
```

4.30.2 Retrieving SIP Remote Transport Address Layer Information

The application uses the data received with the GCEV_OFFERED event (in the case of an incoming SIP INVITE message) and GCEV_REQ_MODIFY_CALL event (in the case of an incoming SIP re-INVITE message) to retrieve the SIP transport address embedded within the received data.

The following code example demonstrates how to retrieve the incoming SIP message originator's transport address from a GCEV_OFFERED event using the field-specific parameter ID IPPARM_SIP_TRANSPORT_ADDR. The GC_PARM_BLK structure containing the data is referenced via the extevtdatap pointer in the METAEVENT structure. In this particular scenario, the GCEV_OFFERED event is generated upon receiving a SIP INVITE message.

Example

```
#include "gclib.h"
..
..
METAEVENT metaevt;
GC_PARM_DATA_EXT parm_data;
GC_PARM_BLK *pParmBlock = NULL;

/* Get Meta Event */
gc_GetMetaEvent(&metaevt);
switch(metaevt->evttype)
{
…
case GCEV_OFFERED:
  currentCRN = metaevt->crn;
  pParmBlock = (GC_PARM_BLK*)(metaevt->extevtdatap);
  INIT_GC_PARM_DATA_EXT(&parm_data);

  /* going thru each parameter block data*/

   while ((ret = gc_util_next_parm_ex(pParmBlock,&parm_data)) == GC_SUCCESS)
```

```
    {
     switch (parm_data.set_ID)
     {
        case IPSET_CALLINFO:
         switch (parm_data.parm_ID)
         {
             case IPPARM_SIP_TRANSPORT_ADDR:
             {
              IP_SIP_TRANSPORT_ADDR *pSipTA;

                 pSipTA = (IP_SIP_TRANSPORT_ADDR *)parm_data.pData;
                 printf("\tGot Sip transport address IP=%s Port=%d, Transport Type=%d,
                          Address Type=%d, version = d\n",
                      pSipTA->remTA,
                      pSipTA->remTAport,
                      pSipTA->eremTransportType,
                      pSipTA->eremAddrType,
                      pSipTA->version);
              }
              break;
            }
          break;
        }
     }
.
.
}
```

4.31 SIP Session Timer

The SIP session timer feature allows the application to set the session duration to keep Session Initiation Protocol (SIP) sessions active. The feature includes a keep alive mechanism to set the length and the minimum time of the session and to refresh the duration of the call. It allows User Agents and proxies to determine if the SIP session is still active.

4.31.1 Overview

This feature provides a keep alive mechanism for the SIP to determine whether a session is still active using an extension defined in RFC 4028. The session timer uses three fields in the IP_VIRTBOARD data structure to enable the timer, and to set the values for session expiration and minimum time. For negotiation between the user agent server (UAS) and user agent client (UAC), the session timer uses SIP IP parameters available via the Global Call API.

A Session-Expires header included in the 2xx response of an initial INVITE determines the session duration. Periodic refresh enables extending the duration of the call and allows both User Agents and proxies to determine if the SIP session is still active. The refresh of a SIP session is done through a re-INVITE or UPDATE. The Session-Expires header in the 2xx response also indicates which side will be the refresher; that is, the side responsible for generating the refresh request. The refresher can be the UAC or UAS. The refresher should generate a refresh request (using re-INVITE or UPDATE) before the session expires. If no refresh request is sent or received before the session expires, or if the refresh request is rejected, both parties should send BYE.

The session timer feature has two headers, Session-Expires and Min-SE, and a response code of 422. The Session-Expires header conveys the lifetime of the session, the Min-SE header conveys the minimum value for the session timer, and the 422 response code indicates that the session timer duration is too small. If the Min-SE header is missing, the minimum value for the session timer is 90 seconds by default, in compliance with RFC 4028.

The session timer is supported for 1PCC and 3PCC modes.

4.31.2 Enabling SIP Session Timer in IP_VIRTBOARD

The following fields in the IP_VIRTBOARD data structure can be adjusted for each virtual board:

E_SIP_SessionTimer_Enabled
: Enables session timer for a virtual board. Default is disabled.

SIP_SessionTimer_SessionExpires
: Sets the session expires value. Used in timer negotiation for outgoing and incoming calls. Default timer value is 1800 seconds for all calls enabled.

SIP_SessionTimer_MinSE
: Sets the minimum session timer value. Used in timer negotiation for outgoing and incoming calls. Default timer value is 90 seconds for all calls enabled. A 422 response code indicates that the session timer duration is too small. That response contains a Min-SE header field identifying the minimum session interval it is willing to support.

Note: If the session timer is not enabled on the virtual board, the UPDATE method is not supported. Incoming UPDATE message is responded with a 501 Not Implemented message.

Example

This code snippet shows configuration of default Session-Expires and Min-SE timer values on the virtual board using **gc_Start( )**.

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..
..
/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data */
INIT_IPCCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);

/* initialize virtual board */
INIT_IP_VIRTBOARD(&virtBoards[0]);

/* Enable session timer and configure default parameters */
virtBoard[0].E_SIP_SessionTimer_Enabled= ENUM_Enabled;
virtBoard[0].SIP_SessionTimer_SessionExpires= 3600;
virtBoard[0].SIP_SessionTimer_MinSE= 1800;
```

4.31.3 Configuring SIP Session Timer

Session timer negotiation between UAS and UAC follows RFC 4028.

The following parameter IDs in the IPSET_SIP_SESSION_TIMER parameter set ID are used for the SIP session timer feature. All parameters can be set on the board device, line device, or call reference number (CRN). Use the appropriate **gc_SetConfigData( )** or **gc_SetUserInfo( )** function.

IPPARM_SESSION_EXPIRES

The parameter value overwrites the configured SIP_SessionTimer_SessionExpires value on the virtual board (IP_VIRTBOARD structure). When the application makes a new call or answers an incoming call, Global Call uses this value to negotiate the session interval when functioning as either a UAC or UAS. Parameter value is in seconds.

IPPARM_MIN_SE

The parameter value overwrites the configured SIP_SessionTimer_MinSE value on the virtual board (IP_VIRTBOARD structure). When the application makes a new call or answers an incoming call, Global Call uses this value to negotiate session interval when functioning as either a UAC or UAS. Parameter value is in seconds.

Note: SIP_SessionTimer_MinSE on the virtual board is always used automatically to reject INVITE whose Session-Expires value is too small. The automatic rejection with 422 is not affected by this parameter because once the value is set in **gc_Start( )**, you can not use IPPARM_MIN_SE to change it for incoming calls. If rejected by 422 response, the application receives GCEV_DISCONNECTED with cause IPEC_SIPReasonStatus422SessionIntervalTooSmall.

IPPARM_REFRESHER_PREFERENCE

The possible values are:

- IP_REFRESHER_LOCAL
- IP_REFRESHER_REMOTE
- IP_REFRESHER_DONT_CARE (default)

If set to local or remote refresher preference, the refresher parameter is added in the Session-Expires header. If set to don't care, the refresher parameter is not added in the Session-Expires header in outgoing INVITE, and default refresher is selected, according to RFC 4028.

The actual refresher is decided by UAS and appears on 200 OK. The following table lists the refresher results if both UAS and UAC are Global Call applications using this refresher preference parameter.

UAC preference	UAS preference	Refresher
IP_REFRESHER_DONT_CARE	IP_REFRESHER_DONT_CARE	UAS
IP_REFRESHER_REMOTE	IP_REFRESHER_REMOTE	UAS
IP_REFRESHER_LOCAL	IP_REFRESHER_LOCAL	UAC
IP_REFRESHER_LOCAL	IP_REFRESHER_REMOTE	UAC
IP_REFRESHER_REMOTE	IP_REFRESHER_LOCAL	UAS

IPPARM_REFRESH_METHOD

The possible values are:

- IP_REFRESH_REINVITE (default)
- IP_REFRESH_UPDATE

If selected as refresher, Global Call sends either a re-INVITE or UPDATE message to periodically refresh the session.

IPPARM_REFRESH_WITHOUT_REMOTE_SUPPORT

The possible values are:

- IP_REFRESH_WITHOUT_REMOTE_SUPPORT_DISABLE
- IP_REFRESH_WITHOUT_REMOTE_SUPPORT_ENABLE (default)

The session timer mechanism can operate as long as one of the two User Agents in the call leg supports the timer extension. As call originator or terminator, the application may choose to execute the session timer if the remote side does not support the session timer. If enabled, Global Call operates the session timer if the remote side does not support session timer and sends the refresh. The other side sees the refreshes as repetitive re-INVITEs. If disabled, Global Call does not operate the session timer if the remote side does not support the session timer.

When session timer is supported on only one side, re-INVITE is the only method supported in order to refresh the session. If UA uses UPDATE to refresh the session, it gets a 501 Not Implemented message back. It is up to the application to decide if it wants to terminate the session or not once the session expires through the use of IPPARM_TERMINATE_CALL_WHEN_EXPIRES.

IPPARM_REFRESH_WITHOUT_PREFERENCE

The possible values are:

- IP_REFRESH_WITHOUT_PREFERENCE_DISABLE
- IP_REFRESH_WITHOUT_PREFERENCE_ENABLE (default)

When the 2xx final response is received, but the refresher preference does not match the call refresher, the application may choose to execute the session timer. If enabled, Global Call operates the session timer mechanism and the refresher is different from the application preference. If disabled, Global Call does not operate the session timer.

IPPARM_TERMINATE_CALL_WHEN_EXPIRES

The possible values are:

- IP_TERMINATE_CALL_WHEN_EXPIRES_DISABLE
- IP_TERMINATE_CALL_WHEN_EXPIRES_ENABLE (default)

When the session timer is about to expire, Global Call sends GCEV_SIP_SESSION_EXPIRES event to application. If enabled, Global Call sends BYE to terminate the call. If disabled, Global Call does not send BYE and the call stays connected.

4.31.4 422 Response Code

The 422 response code indicates that the session timer duration is too small. This response contains a Min-SE header field identifying the minimum session interval supported.

The application can send the acceptable minimum session timer value in the 422 response code. Use **gc_SetConfigData( )** with IPSET_CONFIG / IPPARM_REGISTER_SIP_HEADER parameter pair, and include the header field name **min-SE** as the parameter value, to set the min-SE value on a per call basis. If a call is dropped with reason 422, the min-SE value is included in the response.

4.31.5 SIP Session Timer Supports SDP in Re-INVITE

When a session timer is configured using Dialogic® HMP software in 3PCC mode, an SDP body that was previously negotiated during the offer/answer exchange is added to the session timer re-INVITE.

An application, acting as the UAS and configured in 3PCC mode, may choose to send an SDP body in an 18x Provisional Response requiring a PRACK, in which case an SDP body should not be contained in the final 200OK response. Previously, with HMP using session timers under this scenario, the automatic session timer re-INVITE (out of HMP) would not contain an SDP body. Now, the automatic HMP session timer will contain the same SDP body previously negotiated as part of the offer/answer, even if the SDP was not part of the final 200OK response out of HMP.

4.32 Unspecified G.723.1 Bit Rate in Outgoing SIP Requests with SDP

An application in 1PCC mode can choose not to specify the G.723.1 codec bit rate, namely 5.3 kbps or 6.3 kbps, in an outgoing SIP message with SDP body. Instead, the application can let the far end UA request the bit rate. Feature enablement and disablement can be controlled either at the IPT board-level device or the IPT network device (channel).

This feature is only applicable in 1PCC mode.

G.723.1 is a dual-rate codec supporting 5.3 kbps and 6.3 kbps bit rates. With SIP, the choice of one or the other rate for RTP streaming is specified in an SDP body in outgoing SIP messages using an optional bit rate parameter in the format transport (a=fmtp) attribute of the media announcement. Currently, when in 1PCC mode, Global Call always includes this optional bit rate parameter in SDP bodies irrespective of whether the application’s IP_CAPABILITY data structure setting specifies a single G.723.1 bit rate (5.3 or 6.3 kbps), both G.723.1 bit rates (5.3 and 6.3 kbps), or “don’t care” for local transmit rates.

When both G.723.1 rates are specified, only the one in the first IP_CAPABILITY structure is included. The following example shows an SDP message extract from an HMP SIP request where the application set both rates in the IP_CAPABILITY data structure. The outgoing SDP contains only the bit rate that was specified first. In this particular case, the capability field in the first IP_CAPABILITY element with IP_CAP_DIR_LCLTRANSMIT direction was set to GCCAP_AUDIO_g7231_5_3k for the IPT network device:

```
v=0
o=Intel_IPCCLib 50083120 50083121 IN IP4 192.168.185.64
s=Dialogic_SIP_CCLLIB
i=session information
c=IN IP4 192.168.185.64
t=0 0
```

```
m=audio 49152 RTP/AVP 0 8 4 101
a=rtpmap:4 G723/8000
a=fmtp:4 bitrate=5.3; annexa=no
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

With this feature, when the application specifies both G.723.1 bit rates as transmit codec selection, the HMP software no longer sends the optional bit rate parameter in the SDP body of a SIP request or response. Instead, the remote UA has the responsibility to select one of the G.723.1 bit rates for bit rate negotiation.

Note: If only one of the G.723.1 bit rates is specified by the application transmit codec selection, the HMP software maintains its existing behavior and the specified selection applies when negotiating the bit rate to use irrespective of this feature enablement. Likewise, there is no change in behavior if the application selects "don't care" (GCCAP_dontcare) as transmit codec selection, meaning that the complete list of coders supported by a product is used when negotiating the coder type to be used, also irrespective of this feature enablement.

4.32.1 Enabling the Feature

To control the behavior of the G.723.1 bit rate for the entire IPT board, use **gc_SetConfigData( )** with a Target Type GCTGT_CCLIB_NETIF and the board-level device handle for Target ID.

To control the behavior of the G.723.1 bit rate for all calls on a given channel, use **gc_SetUserInfo( )** with the IPT network device (channel).

In both cases, use IPSET_CONFIG set ID with the following parameter ID to disable or enable this feature.

IPPARM_SEND_G723_UNSPECIFIED_BITRATE
: Specifies not to send any bit rate for G.723.1 on outgoing SIP requests or responses. Valid values:
- IP_ENABLE (Default)
- IP_DISABLE

Note: This parameter ID is only applicable if both 5.3 kbps and 6.3 kbps G.723.1 bit rates are specified in the capability field in IP_CAPABILITY elements with IP_CAP_DIR_LCLTRANSMIT direction. Otherwise, the IPPARM_SEND_G723_UNSPECIFIED_BITRATE parameter has no effect.

When controlling the feature behavior at the board level, use **gc_SetConfigData( )** with the following arguments:

target_type
: The target object type. Use GCTGT_CCLIB_NETIF.

cclib_target_id
: Indicates the board-level device.

GC_PARM_BLKP target_datap
: Pointer to the data from target object. Use set_ID = IPSET_CONFIG and parm_ID = IPPARM_SEND_G723_UNSPECIFIED_BITRATE.

This feature is enabled by default, which eliminates the bit rate parameter in SDP bodies in SIP messages from HMP when both bit rates are specified in IP_CAPABILITY structures with

IP_CAP_DIR_LCLTRANSMIT as direction. The application has the option to disable this feature using the IP_DISABLE value for IPPARM_SEND_G723_UNSPECIFIED_BITRATE if desired to maintain backward compatibility.

It is recommended that whenever the application desires to specify one G.723.1 bit rate as part of the SIP request, that only one G.723.1 bit rate is passed to the HMP software in a capability field of an IP_CAPABILITY element with IP_CAP_DIR_LCLTRANSMIT as direction. In that particular case, the IPPARM_SEND_G723_UNSPECIFIED_BITRATE parameter is not applicable.

4.32.2 Example

Since this feature is enabled by default, this example is intended to help customers revert to the previous behavior by disabling the feature if so desired.

The example is divided in two parts. The first part demonstrates how to disable this feature on a board-level basis; that is, make the HMP software behave prior to this feature implementation.

```
char *boardDeviceName = "N_iptB0:P_IP";  // example of board name for gc_openex()
LINEDEV boarddevh = 0;                  // IP board-level line device filled by gc_openex()

INT32 processEvtHandler()
{
GC_PARM_BLK *parmblkp = NULL;
long request_id = 0;

      switch (evtType)
      {
      ...
      case GCEV_OPENEX:
                 // board case

                 gc_util_insert_parm_val(&parmblkp, IPSET_CONFIG,
                           IPPARM_SEND_G723_UNSPECIFIED_BITRATE, IP_DISABLE);

                 if (gc_SetConfigData(GCTGT_CCLIB_NETIF, boarddevh, parmblkp, 0,
                           GCUPDATE_IMMEDIATE, & request_id, EV_ASYNC) != GC_SUCCESS)
                 {
                            // Process error
                 }
                 gc_util_delete_parm_blk(parmblkp);
                 break;
      }
      ...
```

This second part completes the example by showing how to include both G.723.1 audio bit rates in any outgoing SIP messages with SDP body on a given channel.

```
   if (ipcap.capability != GCCAP_DATA_t38UDPFax) {
            ipcap.type = GCCAPTYPE_AUDIO;
            ipcap.direction = IP_CAP_DIR_LCLTRANSMIT;
            ipcap.extra.audio.frames_per_pkt = GCCAP_AUDIO_g7231_6_3k;
            ipcap.extra.audio.VAD = vad;
            /* append the GC_PARM_BLK with the respective TX codec */
            gc_util_insert_parm_ref(&parmblkp, GCSET_CHAN_CAPABILITY,
            IPPARM_LOCAL_CAPABILITY, sizeof(IP_CAPABILITY), &ipcap);
            ipcap.type = GCCAPTYPE_AUDIO;
            ipcap.direction = IP_CAP_DIR_LCLTRANSMIT;
            ipcap.extra.audio.frames_per_pkt = GCCAP_AUDIO_g7231_5_3k;
            ipcap.extra.audio.VAD = vad;
            /* append the GC_PARM_BLK with the respective TX codec */
            gc_util_insert_parm_ref(&parmblkp, GCSET_CHAN_CAPABILITY,
            IPPARM_LOCAL_CAPABILITY, sizeof(IP_CAPABILITY), &ipcap);

            if (gc_SetUserInfo(GCTGT_GCLIB_CHAN, port[index].ldev, parmblkp,
                       scope) != GC_SUCCESS)
            {
            //Process error
            }
    }
```

4.33 Dynamic Selection of Outbound SIP Proxy

The application can select an outbound SIP proxy server on the Dialogic® HMP virtual board device dynamically. If an outbound SIP proxy server was selected at board initialization, it is overridden; otherwise it is selected for the first time.

This feature is supported in first party call control (1PCC) and third party call control (3PCC) operating modes.

By default, the outbound proxy address setting is established statically as the Dialogic® HMP board is initialized by setting a specific outbound proxy IP address or a host name in the IP_VIRTBOARD structure that is used by **gc_Start( )**. Afterwards, all outgoing SIP messages will pass through the configured outbound proxy. While this method remains in effect, the ability to temporarily and dynamically select the outbound proxy IP address is available with this feature.

The application can now configure (or reconfigure) the proxy address to use temporarily for routing all future SIP messages on a virtual HMP board basis. The temporary proxy can be deselected dynamically to revert the outbound proxy setting to the prior setting. If a prior setting was made at virtual board initialization time, then it takes effect immediately; otherwise outbound proxy is disabled.

When the set ID and parameter ID pairs are passed to the Dialogic® HMP virtual board via the **gc_SetConfigData( )** function, outbound messages on all IP channels are routed via the newly set outbound proxy server. The same method is used to deselect the alternate outbound proxy. Once the outbound proxy selection is in place, all supported SIP requests and all outbound SIP responses, within a dialog or not, are affected.

Note: A Contact header field in a 200OK response from the called party (User Agent Server, or UAS) resulting from a Dialogic® HMP INVITE will cause the ACK message and all future requests from Dialogic® HMP Software to bypass the proxy and go directly to the UAS for the remaining of the dialog.

This feature is disabled by default. Once enabled, this feature applies to the following outgoing SIP requests, in addition to SIP responses out of Dialogic® HMP Software.

- 1PCC mode
  ACK, INFO, INVITE, OPTIONS, REFER, REGISTER, BYE, NOTIFY, SUBSCRIBE and CANCEL.

- 3PCC mode
  ACK, INFO, INVITE, OPTIONS, REFER, REGISTER, BYE, NOTIFY, SUBSCRIBE, UPDATE, PRACK and CANCEL.

Note: Explicitly manipulating Route and Record-Route headers in the application will override the proxy settings.

See also Section 4.34, "Proxy Bypass in SIP Register Requests", on page 369 for related information.

4.33.1 Enabling Dynamic Proxy Selection

This feature is disabled by default. To set an alternate proxy dynamically, you must:

- Enable this feature at virtual board initialization time, using the IP_VIRTBOARD structure in the **gc_Start( )** function. Set the E_SIP_dynamic_outbound_proxy_enabled field in IP_VIRTBOARD to ENUM_Enabled.
- Configure the IPSET_PROXY_INFO set ID and its parameter IDs using the **gc_SetConfigData( )** function. This function uses the IP_PROXY_INFO data structure.

The IPSET_PROXY_INFO set ID and parameter ID pair are inserted into the GC_PARM_BLKP data structure. The parameter IDs are:

IPPARM_PROXY_ACTION
: Determines whether to use the proxy for the specific request.
  - IP_PROXY_SELECT – instructs the library to dynamically choose the proxy
  - IP_PROXY_DESELECT – instructs the library to deselect the proxy

IPPARM_PROXY_INFO
: Denotes the proxy information structure, IP_PROXY_INFO.

4.33.2 Dynamic Proxy Selection Examples

Several examples are provided to illustrate dynamic proxy selection.

Enabling the Feature

This example shows how to enable the feature using the IP_VIRTBOARD structure. Here, the E_SIP_dynamic_outbound_proxy_enabled field is set to ENUM_Enabled and the version is set to VIRTBOARD_VERSION_DYNAMIC_ OUTBOUND_PROXY_SUPPORT.

```
GC_START_STRUCT gclib_start;
IPCCLIB_START_DATA cclibStartData;
IP_VIRTBOARD virtBoards[1]; // only 1 NIC supported in current release
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
memset(virtBoards,0,sizeof(IP_VIRTBOARD));

INIT_IP_VIRTBOARD(virtBoards);
virtBoards[0].total_max_calls = g_h323count + g_sipcount;
virtBoards[0].h323_max_calls = g_h323count;
virtBoards[0].sip_max_calls = g_sipcount;
virtBoards[0].sip_signaling_port = g_sipport;
virtBoards[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE | IP_SIP_MIME_ENABLE;
```

```
virtBoards[0].E_SIP_OPTIONS_Access = ENUM_Enabled;
virtBoards[0].E_SIP_dynamic_outbound_proxy_enabled = ENUM_Enabled;
virtBoards[0].E_SIP_DefaultTransport = ENUM_UDP;
virtBoards[0].sip_signaling_port =5060;
virtBoards[0].version = VIRTBOARD_VERSION_DYNAMIC_OUTBOUND_PROXY_SUPPORT;

INIT_IPCCLIB_START_DATA(&cclibStartData, 1, virtBoards);
cclibStartData.max_parm_data_size = 512;

CCLIB_START_STRUCT cclib_start[]={{"GC_H3R_LIB", &cclibStartData},
                                  {"GC_IPM_LIB", NULL},
                                  {"GC_DM3CC_LIB", NULL}};

gclib_start.num_cclibs = 2;
gclib_start.cclib_list = cclib_start;

if (gc_Start(&gclib_start) != GC_SUCCESS)
{
     // Handle error
}
```

Setting Dynamic Proxy Selection

The following example shows how to set the dynamic proxy information to be selected:

```
// Declare a new structure
IP_PROXY_INFO proxyinfo;

// Open the board device and get the handle
if(gc_OpenEx( &g_sipbd, ":N_iptB1:P_IP", EV_SYNC, NULL) == -1)
{
    // Handle Error
}

// Select the proxy action to select, which will cause the  proxy information to be used in
   setting the outbound details
// Insert this information into the GC Parameter block
rc = gc_util_insert_parm_val(&l_pParmBlk,
                       IPSET_PROXY_INFO,
                       IPPARM_PROXY_ACTION,
                       sizeof(char),
                       IP_PROXY_SELECT);
if (rc < 0)
{
    // Handle error
}

// Initialize the proxy structure using the macro INIT_IP_PROXY_INFO(&proxyinfo);

// Fill in the appropriate values in the structure
strcpy(&(proxyinfo.arrAddr[0]), "146.152.97.100");
proxyinfo.eAddrType = ePROXY_ADDRESS_TYPE_IP;
proxyinfo.eProtocol = ePROXY_PROTOCOL_UDP;
proxyinfo.usPort = 5060;

// Insert the proxy info structure into the GC Parameter Block
rc = gc_util_insert_parm_ref_ex (&l_pParmBlk,
                       IPSET_PROXY_INFO,
                       IPPARM_PROXY_INFO,
                       sizeof(IP_PROXY_INFO),
                       (void*) &(proxyinfo)
                       );
if (rc < 0)
{
     // Handle Error
}

// Use the parameter block from above and send it down to the IPCCLIB
rc = gc_SetConfigData(GCTGT_CCLIB_NETIF,
                g_sipbd,
                l_pParmBlk,
                0,
                GCUPDATE_IMMEDIATE,
                &l_requestID,
                EV_ASYNC);
if(rc < 0)
{
     // Handle error
}
```

Disabling Dynamic Proxy Selection

The following example shows how to disable the currently active proxy selection:

```
// Set the appropriate value for the proxy action
rc = gc_util_insert_parm_val(&l_pParmBlk,
                      IPSET_PROXY_INFO,
                      IPPARM_PROXY_ACTION,
                      sizeof(char),
                      IP_PROXY_DESELECT);
if (rc < 0)
{
     // Handle error
}

// Use the parameter block from above and send it down to the IPCCLIB
rc = gc_SetConfigData(GCTGT_CCLIB_NETIF,
                g_sipbd,
                l_pParmBlk,
                0,
                GCUPDATE_IMMEDIATE,
                &l_requestID,
                EV_ASYNC);
if(rc < 0)
{
     // Handle error
}
```

4.34 Proxy Bypass in SIP Register Requests

The application can override the automatic route to a configured proxy previously set while sending outgoing REGISTER request messages. This feature is applicable in 1PCC and 3PCC modes, and applies only to SIP REGISTER. It does not apply to H.323 protocol or any other SIP Message.

The SIP REGISTER method is used to register associations between a media endpoint alias and its real (transport) address. These associations are commonly referred to as bindings, each of which represents a unique combination of several items, including:

- Request-URI represented in the request line.
- Address of Record (a “name” that will be used to easily locate the SIP endpoint) represented by the ‘To’ header field in the SIP REGISTER request.

- Transport address (the actual URI of the SIP endpoint), which is specified as the 'Contact' header field.
- Sender's Address of Record (only used in 3PCC environments) represented by the 'From' header field.

Secure IP environments, apart from having a Registrar, also consist of a Proxy. The purpose of the Proxy is to filter communications and mask the communication exiting the secure environment. In a proxy-enabled environment all outbound communication, including Register requests, exit only after passing through the Proxy.

With this feature, user applications can bypass sending SIP REGISTER request messages to a configured outbound proxy server. Instead, requests can be sent directly to the Registrar server.

Notes:

1. If AutoRefresh is enabled for the binding, the AutoRefresh requests also bypass the proxy when this feature is enabled.
2. The override functionality is applied on an association basis. Every association is a combination of To, From, and Request URI header fields. Depending on the sub-operation requested, if the contents of the request target is the same association, if that association uses proxy override, then the outgoing REGISTER request bypasses the proxy.
3. The Request-URI field or the Registrar address in the REGISTER request is used to bypass the configured proxy and direct it to the Registrar.

See also Section 4.33, "Dynamic Selection of Outbound SIP Proxy", on page 366 for related information.

4.34.1 Enabling the Feature

To bypass the proxy, use the IPSET_PROXY_INFO set ID and IPPARM_PROXY_ACTION parameter ID. These parameters are inserted into the GC_PARM_BLKP data structure using **gc_insert_parm_ref**( ). When the **gc_ReqService**( ) function is called with the parameter set ID, the REGISTER request messages are sent directly to the Registrar instead of the Proxy.

Note: The IPSET_PROXY_INFO set ID is supported only in registration requests.

Example

The following example demonstrates the override procedure. To enable override, the parameter IPPARM_PROXY_ACTION must be set to the value IP_PROXY_BYPASS. By default, proxy override is disabled. Enable only when the Register request type is IP_REG_ADD_INFO or IP_REG_SET_INFO.

```
GC_PARM_BLKP        pParmBlock= NULL;

if (gc_util_insert_parm_val(&pParmBlock,
IPSET_PROXY_INFO,
IPPARM_PROXY_ACTION,
sizeof(char),
IP_PROXY_BYPASS) < 0)
{
// Error Handling
}
```

```
// Setting the operation to perform
    if (gc_util_insert_parm_val(&pParmBlock,
                                IPSET_REG_INFO,
                                IPPARM_OPERATION_REGISTER,
                                sizeof(char),
                                IP_REG_SET_INFO) < 0)  // Or IP_REG_ADD_INFO
{
// Error Handling
}
if (gc_ReqService(GCTGT_CCLIB_NETIF,
                      bdev,
                      &reg[index].ServiceId,
                      pParmBlock,
                      NULL,
                      EV_ASYNC) < 0)
{
// Error Handling
}
```

4.35 SIP WaitCall Cancellation

A SIP application is able to dynamically block the ability of an IPT network device to receive a call. This action prevents possible glare conditions during an attempt to make an outbound call while an incoming call is being processed on the same channel. With this feature, the application can block the channel from accepting calls before making an outbound call. If an incoming call is already in progress, the application is notified and the call in progress is not affected.

The Global Call and SIP call control libraries provide IPT network devices (channels) with the ability to receive incoming calls using the **gc_WaitCall( )** function. With IP call control, incoming calls are presented to the application on channels that issued **gc_WaitCall( )** based on an internal algorithm.

Prior to this feature, **gc_MakeCall( )** would fail on a channel that was processing an incoming call. To avoid this condition, the application had to call the **gc_Close( )** or **gc_ResetLineDev( )** functions to disable a previously issued **gc_WaitCall( )** function on a channel-by-channel basis. As a result, any incoming call that existed on the line device would be also terminated without notification.

This feature provides an API function specific to SIP call control. The **gc_CancelWaitCall( )** function lets the application prevent incoming SIP calls on a particular IPT network device without losing any incoming calls that may have just arrived. The application can use this functionality to cancel any previously issued **gc_WaitCall( )** on a channel-by-channel basis.

If the application intends to make an outbound call, it issues **gc_CancelWaitCall( )** prior to the **gc_MakeCall( )** function. The **gc_CancelWaitCall( )** failure indicates that a call is already in progress so it will not be cancelled. When the application receives the error, it does not attempt an outbound call, thus avoiding a possible glare condition.

Use the **gc_CancelWaitCall( )** function to cancel a **gc_WaitCall( )** previously issued on an IPT network device. The GCEV_CANCELWAITCALL event indicates that notification of incoming calls was successfully disabled.

4.36 Configuring SIP Stack Parameters

You can configure selected SIP stack parameters such as timers using the Dialogic® Global Call API.

Use the SIP_STACK_CFG data structure to configure SIP stack parameters. To support SIP stack configuration, IP_VIRTBOARD is updated with a pointer to this structure.

Note: SIP stack parameters can only be configured once per virtual board (at **gc_Start( )**) and remain in effect throughout the Global Call application (per process).

Example

The following example sets the SIP T1 timer to 64 ms.

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..
..

/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data */
INIT_IPCCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);

/* initialize virtual board */
INIT_IP_VIRTBOARD(&virtBoards[0]);

/* sip stack cfg support */
SIP_STACK_CFG sip_stack_cfg;
INIT_SIP_STACK_CFG(&sip_stack_cfg);
   virtBoard[bid].sip_stack_cfg = &sip_stack_cfg;

   sip_stack_cfg.retransmissionT1 = 64;
```

4.37 Setting Dialogic® IP Media Library Parameters

As a convenience to Global Call application developers, most Dialogic® IP Media Library API parameters that are set via the IPM_PARM_INFO data structure can be set using a Dialogic® Global Call API call.

The IP Media Library settings that can be configured for a line device from Global Call include the following:

- enabling/disabling echo cancellation
- specifying adaptive coefficients for echo cancellation
- specifying the echo tail length for echo cancellation
- adjusting audio volume level to or from the IP network

- specifying the type of service in IPv4 headers, either as a 7-bit TOS field or as a 6-bit DSCP field for Differentiated Services (per RFC2474)

For more information on the IP Media Library parameters that can be set and the supported values for those parameters, see the reference pages for the IPM_PARM_INFO data structure in the *Dialogic® IP Media Library API Library Reference*.

To set an IP Media Library parameter for a line device from Global Call, the application first constructs an IPM_PARM_INFO data structure that contains the desired parameter ID and value. Then a parameter element containing the structure is inserted into a GC_PARM_BLK via the **gc_util_insert_parm_ref**( ) function using the following IDs:

IPSET_CONFIG

  IPPARM_IPMPARM

  - Value = IPM_PARM_INFO data structure

The application then calls the **gc_SetUserInfo( )** function to send the parameter block to the **ipm_SetParm( )** function on a pass-through basis (that is, without any validity checking on the Global Call side).

The **ipm_SetParm( )** function is called asynchronously even though **gc_SetUserInfo( )** is a synchronous function. The return value of the **ipm_SetParm( )** function call is passed through as the return value for the **gc_SetUserInfo( )** call and must be interpreted as it is for the asynchronous **ipm_SetParm( )** call; specifically, the success return value only indicates that the **ipm_SetParm( )** function began execution successfully. If the set parameter operation completes successfully, an IPMEV_SETPARM event will be generated by IP Media Library, but there will be no corresponding Global Call event because there is no completion event for the synchronous **gc_SetUserInfo( )** function. If an error occurs while setting the parameter, there an IPMEV_ERROR event is generated by IP Media Library and a GCEV_TASKFAIL event is generated by Global Call.

The following example illustrates how a Global Call application might enable echo cancellation:

```
IPM_PARM_INFO ipmParmInfo;
Int echoCancellation = ECACTIVE_ON;

ipmParmInfo.eParm = PARMCH_ECACTIVE;
ipmParmInfo.pvParmValue = (void *)&echoCancellation;

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_IPMPARM,
                        (unsigned long)sizeof(IPM_PARM_INFO),
                        &ipmParmInfo);

gc_SetUserInfo(GCTGT_GCLIB_CHAN, lineDev, parmblkp, GC_ALLCALLS);

gc_util_delete_parm_blk(parmblkp);
```

The following code example illustrates how the TOS field might be set from a Global Call application:

```
IPM_PARM_INFO ipmParmInfo;
char tos=5;
```

```
ipmParmInfo.eParm = PARMCH_TOS;
ipmParmInfo.pvParmValue = (void *)&tos;

gc_util_insert_parm_ref(&parmblkp,
                        IPSET_CONFIG,
                        IPPARM_IPMPARM,
                        (unsigned long)sizeof(IPM_PARM_INFO),
                        &ipmParmInfo);

gc_SetUserInfo(GCTGT_GCLIB_CHAN, port[index].ldev, parmblkp, GC_ALLCALLS);

gc_util_delete_parm_blk(parmblkp);
```

4.38 T.38 Fax Server

Dialogic® Global Call API support for T.38 Fax Server is described under the following topics:

- T.38 Fax Server Support Overview
- Specifying Manual Operating Mode
- Initiating a Switch from Audio to T.38 Fax
- Associating a T.38 Fax Device with a Media Device When a Fax Request is Received
- Accepting/Rejecting a Request to Switch Between Audio and T.38 Fax
- Sending a T.38 Fax in a Session Without Audio Established
- Receiving a T.38 Fax in a Session Without Audio Established
- Sending a Request to Switch from T.38 Fax to Audio
- Receiving a Request to Switch from T.38 Fax to Audio
- Terminating a Call After a T.38 Fax Session
- Handling non-2xx Responses to T.38 Fax Switch

4.38.1 T.38 Fax Server Support Overview

Dialogic® Global Call API provides T.38 fax server functionality to support fax-on-demand and other applications. The functionality includes the ability of an application to:

- initiate and complete a T.38 session without an audio connection being first established
- switch coders from audio to T.38 fax and back again during a pre-established audio connection

To support T.38 fax functionality, Dialogic® Global Call API uses two types of media devices:

- a traditional Media device
- a new T.38 Fax device

By default, ipmBxCy represents the media device on board x and channel y, which has no fax capability on HMP. By associating the corresponding voice handle with a fax handle, the ipmBxCy device represents the fax channel defined by the fax handle, with no voice capability. Disassociating the voice and fax devices restores the ipmBxCy device voice capability.

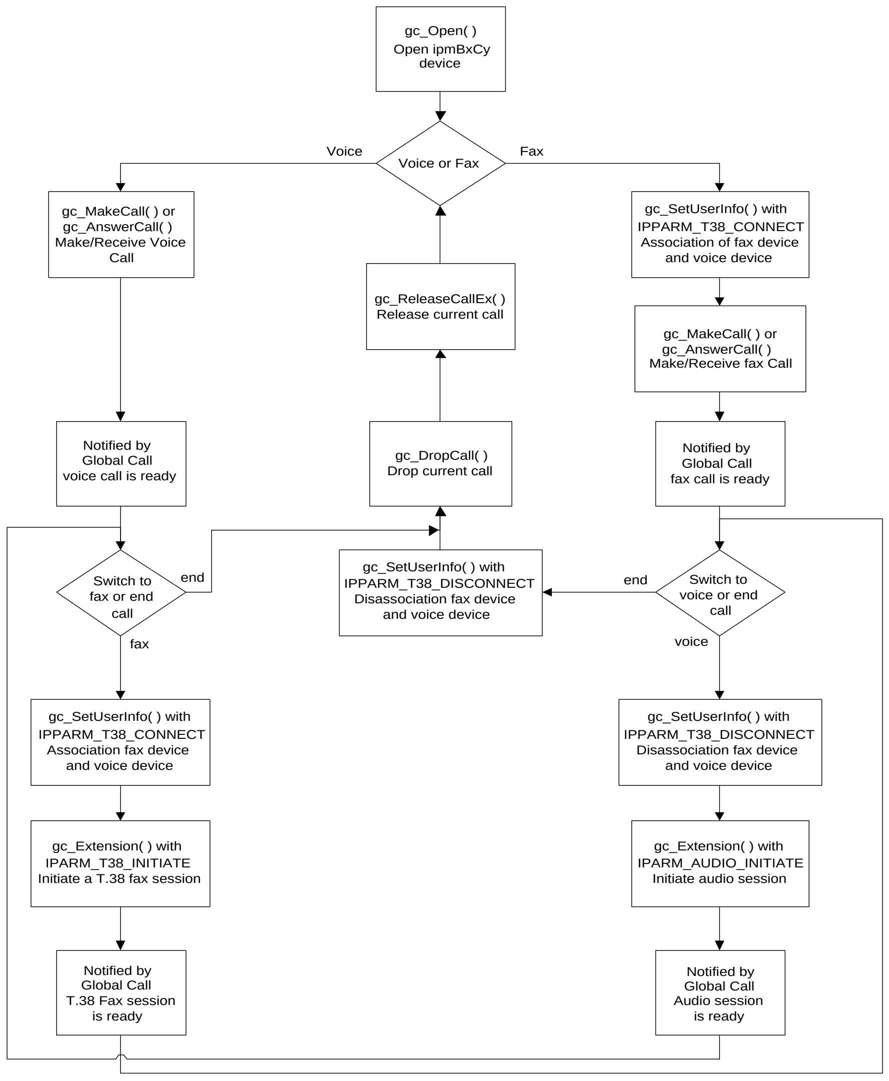
The Dialogic® Global Call API uses the **gc_SetUserInfo( )** function to associate and disassociate a traditional Media device with a T.38 Fax device when establishing or concluding a T.38 fax connection. Manual device association must be done before the next Dialogic® Global Call API function that requires fax information:

- For H.323, the association must be made before **gc_MakeCall( )** on the outbound call side, and **gc_CallAck( )**, **gc_AcceptCall( )** and **gc_AnswerCall( )** on the inbound call side, whichever occurs first since the underlying "open logical channel" can happen at any of these times if coder capabilities and fax port information is available.
- For SIP, the association must be made before **gc_MakeCall( )** on the outbound call side and **gc_AnswerCall( )** on the inbound call side, since media can only be opened after either of these functions.

Note: When a Media device is associated with a T.38 Fax device to establish a fax session over an existing audio connection, then when the fax session concludes, the Media device must be disassociated with the T.38 Fax device, **optionally** reestablishing the audio connection, **before** the call is dropped.

Figure 59 provides a flowchart that summarizes the T.38 fax server functionality and indicates the Dialogic® Global Call API functions and events used at different stages in the call control process. The initial voice or fax capability decision before call connection is determined as described in Section 4.3.2.1, "Specifying Media Capabilities Before Connection", on page 120.

Figure 59. T.38 Fax Server Support in Manual Mode


gc_Open( )
Open ipmBxCy device
Voice or Fax
Voice
Fax
gc_MakeCall( ) or gc_AnswerCall( ) Make/Receive Voice Call
gc_ReleaseCallEx( ) Release current call
gc_SetUserInfo( ) with IPPARM_T38_CONNECT Association of fax device and voice device
gc_MakeCall( ) or gc_AnswerCall( ) Make/Receive fax Call
Notified by Global Call voice call is ready
gc_DropCall( ) Drop current call
Notified by Global Call fax call is ready
Switch to fax or end call
end
fax
gc_SetUserInfo( ) with IPPARM_T38_DISCONNECT Disassociation fax device and voice device
end
Switch to voice or end call
voice
gc_SetUserInfo( ) with IPPARM_T38_CONNECT Association fax device and voice device
gc_SetUserInfo( ) with IPPARM_T38_DISCONNECT Disassociation fax device and voice device
gc_Extension( ) with IPARM_T38_INITIATE Initiate a T.38 fax session
gc_Extension( ) with IPARM_AUDIO_INITIATE Initiate audio session
Notified by Global Call T.38 Fax session is ready
Notified by Global Call Audio session is ready

4.38.2 Specifying Manual Operating Mode

An application must be configured in "Manual" mode to control the association and disassociation of Media and T.38 Fax devices during each call. The mode of operation is set on a board device basis. Once the GCEV_OPENEX event is received to confirm that the board device is open, the **gc_SetConfigData( )** function can be used to configure "Manual" mode as indicated in the code example below:

```
INT32 processEvtHandler()
{
   GC_PARM_BLK  *parmblkp = NULL;
   long         t = 0;
   :
   :
```

```
   switch (evtType)
   {
      :
      :
      case GCEV_OPENEX:
          gc_util_insert_parm_val(&parmblkp, IPSET_CONFIG, IPPARM_OPERATING_MODE,
                                  sizeof(int), IP_MANUAL_MODE);

          gc_SetConfigData(GCTGT_CCLIB_NETIF, pline->ldev, parmblkp, 0,
                           GCUPDATE_IMMEDIATE, &t, EV_ASYNC);

          gc_util_delete_parm_blk(parmblkp);
       break;
      :
      :
   }
   :
   :
}
```

4.38.3 Initiating a Switch from Audio to T.38 Fax

After an audio session has been established, the application can use the **gc_Extension( )** function to initiate a RequestMode (H.323) or re-INVITE (SIP) to change the coder. Prior to initiating a coder change, the T.38 Fax device must be associated with the Media device. This can be achieved using the **gc_SetUserInfo( )** function. The application receives a GCEV_EXTENSION event to indicate that T.38 media is ready to send and receive fax information. The following code provides an example:

```
INT32 processEvtHandler()
{
   METAEVENT      metaEvent;
   GC_PARM_BLK    *parmblkp = NULL;
   IP_CONNECT     ipConnect;
     :
   switch (evtType)
   {
     :
      case GCEV_CONNECTED:
         /* received Connect event */
         /* in conversation */
         ipConnect.version = 0x100;
         ipConnect.mediaHandle = pline->mediaH;
         ipConnect.faxHandle = pline->faxH;
         ipConnect.connectType = IP_FULLDUP;

         gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_CONNECT,
                                 (sizeof(IP_CONNECT)), (void *)(&ipConnect));
         gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);
         gc_util_delete_parm_blk(parmblkp);

         /* Initiate T.38 codec switch */
         gc_util_insert_parm_ref(&parmblkp, IPSET_SWITCH_CODEC, IPPARM_T38_INITIATE,
                                 sizeof(int), NULL);
         gc_Extension(GCTGT_GCLIB_CRN,pline->crn, IPEXTID_CHANGEMODE,
                      parmblkp, NULL, EV_ASYNC);
         gc_util_delete_parm_blk(parmblkp);
      break;

      case GCEV_EXTENSIONCMPLT:
             /* received extension complete event for T.38 initiation*/
             /* do nothing */
      break;
      case GCEV_EXTENSION:
           /* received extension event for media readiness */
         ext_evtblkp = (EXTENSIONEVTBLK *) metaEvent.extevtdatap;
         parmblkp = &ext_evtblkp->parmblk;

         while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
         {
            switch(t_gcParmDatap->set_ID)
            {
               case IPSET_SWITCH_CODEC:
                  switch(t_gcParmDatap->parm_ID)
                  {
                     case IPPARM_READY:
                        /* Ready to send and receive fax */
                        fx_sendfax();
                     break;
                     :
                     :
                  }
               break;
            }
         }

      break;
      :
   }
   :
}
```

4.38.4 Associating a T.38 Fax Device with a Media Device When a Fax Request is Received

During a voice call, a T.38 Fax request can be received by a RequestMode (H.323) or re-INVITE (SIP) message. The application receives notification of the request as a GCEV_EXTENSION event. A T.38 Fax device must then be associated with the Media device by filling in an IP_CONNECT structure with the appropriate T.38 Fax and Media device handles and using the **gc_SetUserInfo( )** function. To continue to accept the request, the **gc_Extension( )** function is used as described in Section 4.38.5, "Accepting/Rejecting a Request to Switch Between Audio and T.38 Fax", on page 379. The following code provides an example:

```
INT32 processEvtHandler()
{
   METAEVENT        metaEvent;
   GC_PARM_BLK      *parmblkp = NULL;
   GC_PARM_DATAP    t_gcParmDatap = NULL;
   GC_PARM_BLK      *parmblkp2 = NULL;
   EXTENSIONEVTBLK  *ext_evtblkp = NULL;
   IP_CONNECT       ipConnect;
   :
   switch (evtType)
   {
      case GCEV_EXTENSION:
         /* received extension event, parse PARM_BLK examine
          * extension data
          */
         ext_evtblkp = (EXTENSIONEVTBLK *) metaEvent.extevtdatap;
         parmblkp = &ext_evtblkp->parmblk;
         while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
         {
            switch(t_gcParmDatap->set_ID)
            {
               case IPSET_SWITCH_CODEC:
                  switch(t_gcParmDatap->parm_ID)
                  {
                     case IPPARM_T38_REQUESTED:
                         /* connect the media and fax devices */
                         ipConnect.version = 0x100;
                         ipConnect.mediaHandle = pline->mediaH;
                         ipConnect.faxHandle = pline->faxH;
                         ipConnect.connectType = IP_FULLDUP;

                         gc_util_insert_parm_ref(&parmblkp2, IPSET_FOIP, IPPARM_T38_CONNECT,
                                                 sizeof(IP_CONNECT), (void *)(&ipConnect));
                         gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn,
                                        parmblkp, GC_SINGLECALL);
                         gc_util_delete_parm_blk(parmblkp2);

                         /* accept T.38 request by example 4.17.5 */
                         acceptCodecSwitchRequest();
                      break;

                     case IPPARM_READY:
                        /* Ready to send and receive fax */
                        fx_sendfax();
                     break;
                  }
               break;
            }
         }
      break;
   }
   :
}
```

4.38.5 Accepting/Rejecting a Request to Switch Between Audio and T.38 Fax

After T.38 coder change request has been received, followed by association of T.38 Fax device with Media device as described in Section 4.38.4, "Associating a T.38 Fax Device with a Media Device When a Fax Request is Received", on page 378, the application can use the **gc_Extension( )** function to accept or reject the request as follows:

- To accept the request, the GC_PARM_BLK associated with the **gc_Extension( )** function includes components that indicate acceptance, specifically IPSET_SWITCH_CODEC and IPPARM_ACCEPT. A RequestModeAck (H.323) or 200 OK (SIP) message is not sent until the request is accepted. The following code provides an example:

```
/* Reject the incoming request */

INT32 acceptCodecSwitchRequest()
{
   GC_PARM_BLK *parmblkp = NULL;
   :
   gc_util_insert_parm_val(&parmblkp, IPSET_SWITCH_CODEC, IPPARM_ACCEPT,
                                  sizeof(int), NULL);
   gc_Extension(GCTGT_GCLIB_CRN,pline->crn, IPEXTID_CHANGEMODE,
                  parmblkp, NULL, EV_ASYNC);
   gc_util_delete_parm_blk(parmblkp);

}
```

- To reject the request, the GC_PARM_BLK associated with the **gc_Extension( )** function includes components that indicate rejection, specifically IPSET_SWITCH_CODEC and

IPPARM_REJECT. The reason for rejecting the request is also included in the GC_PARM_BLK. Chapter 11, “IP-Specific Event Cause Codes” describes the supported reject reasons that can be used in this context. For H.323, reasons prefixed by “IPEC_Q931Cause” can be used. For SIP, reasons prefixed by “IPEC_SIPReason” can be used. The reason parameter corresponds to a RequestModeReject cause (H.323) or a negative response code (SIP). The following code provides an example:

```
/* Reject the incoming request */

INT32 rejectCodecSwitchRequest()
{
   GC_PARM_BLK *parmblkp = NULL;
   :
   :
   /* Reject with reason being busy, SIP */
   gc_util_insert_parm_val(&parmblkp, IPSET_SWITCH_CODEC, IPPARM_REJECT,
                           sizeof(int), IPEC_SIPReasonStatus486BusyHere);
   gc_Extension(GCTGT_GCLIB_CRN, pline->crn, IPEXTID_CHANGEMODE,
                parmblkp, NULL, EV_ASYNC);
   gc_util_delete_parm_blk(parmblkp);
}
```

4.38.6 Sending a T.38 Fax in a Session Without Audio Established

The Dialogic® Global Call API supports the transmission of fax information in a session that does not already have an audio connection established. To send T.38 Fax in such a session, the application must use the **gc_SetConfigData( )** function to specify “Manual” mode, then associate a T.38 Fax device with a media device before calling the **gc_MakeCall( )** function to actually send the fax information. The association only applies to a single call and can be accomplished by calling the **gc_SetUserInfo( )** function on a line device for a single call, or in the GC_MAKECALL_BLK structure when calling **gc_MakeCall( )**.

Note: If using **gc_SetUserInfo( )** to make the association on a line device, the duration must be set to GC_SINGLECALL rather than GC_ALLCALLS.

The following code provides an example:

```
INT32 processEvtHandler()
{
   GC_PARM_BLK *parmblkp = NULL;
   :
   :
   switch (evtType)
   {
      case GCEV_OPENEX:
         /* Set manual mode */
         gc_util_insert_parm_val(&parmblkp, IPSET_CONFIG, IPPARM_OPERATING_MODE,
               sizeof(int), IP_MANUAL_MODE);
         gc_SetConfigData(GCTGT_GCLIB_NETIF, boarddev, parmblkp, 0,
               GCUPDATE_IMMEDIATE, &t, EV_ASYNC);
         gc_util_delete_parm_blk(parmblkp);
```

```
            /* Associate T.38 device with media device */
            ipConnect.version = 0x100;
            ipConnect.mediaHandle = pline->mediaH;
            ipConnect.faxHandle = pline->faxH;
            ipConnect.connectType = IP_FULLDUP;
            gc_util_insert_parm_ref(&(libBlock.ext_datap), IPSET_FOIP, IPPARM_T38_CONNECT,
                 sizeof(IP_CONNECT), (void *)(&ipConnect));
            gc_SetUserInfo(GCTGT_GCLIB_CHAN, pline->LDEV, parmblkp, GC_SINGLECALL);
            gc_util_delete_parm_blk(parmblkp);

            /* Make call now */
            gc_MakeCall();
         break;

         case GCEV_CONNECTED:
            fx_sendfax();
         break;
      }
      :
      :
   }
```

4.38.7 Receiving a T.38 Fax in a Session Without Audio Established

The Dialogic® Global Call API supports the reception of fax information in a session that does not already have an audio connection established. The application can receive a GCEV_OFFERED event with a T.38 Fax request even if the session has no audio connection.

Note: The parameter block associated with the GCEV_OFFERED event indicates an incoming T.38 Fax request if T.38 Fax is the **only** media offered in the incoming request. If more than T.38 media is offered, no specific T.38 information will be associated with offered event.

To answer the T.38 offer, the application must associate a Fax device with the Media device and set local T.38 media capability before calling the **gc_AnswerCall( )** function. The following code provides an example:

```
INT32 processEvtHandler()
{
   METAEVENT         metaEvent;
   GC_PARM_BLK       *parmblkp = NULL;
   GC_PARM_DATAP     t_gcParmDatap = NULL;
   GC_PARM_BLK       *parmblkp2 = NULL;
   EXTENSIONEVTBLK   *ext_evtblkp = NULL;
   IP_CONNECT        ipConnect;
   IP_CAPABILITY     ipcap;
   :
   switch (evtType)
   {
      case GCEV_OFFERED:
         /* parse PARM_BLK examine data */
         parmblkp = (GC_PARM_BLK *)metaEvent.extevtdatap;

         while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
         {
            switch(t_gcParmDatap->set_ID)
            {
               case IPSET_FOIP:
                  switch(t_gcParmDatap->parm_ID)
                  {
                     case IPPARM_T38_OFFERED:
```

```
                                /* connect media with fax devices */
                                ipConnect.version = 0x100;
                                ipConnect.mediaHandle = pline->mediaH;
                                ipConnect.faxHandle = pline->faxH;
                                ipConnect.connectType = IP_FULLDUP;

                                gc_util_insert_parm_ref(&parmblkp2, IPSET_FOIP, IPPARM_T38_CONNECT,
                                                        (sizeof(IP_CONNECT)), (void *)(&ipConnect));
                                gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp2, GC_SINGLECALL);
                                gc_util_delete_parm_blk(parmblkp2);

                                /* set T.38 media capability*/
                                ipcap.capability = GCCAP_DATA_t38UDPFax;
                                ipcap.type = GCCAPTYPE_RDATA;
                                ipcap.direction = IP_CAP_DIR_LCLTXRX;
                                ipcap.extra.data.max_bit_rate = 144;

                                gc_util_insert_parm_ref(&parmblkp2, GCSET_CHAN_CAPABILITY,
                                                        IPPARM_LOCAL_CAPABILITY,
                                                        sizeof(IP_CAPABILITY), &ipcap);
                                gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, pParmBlock2,
                                               GC_SINGLECALL);
                                gc_util_delete_parm_blk(pParmBlock2);

                                /* received completion event for gc_Extension() */
                                gc_AnswerCall(pline->crn, 0, EV_ASYNC);
                             break;
                         }
                     break
                  }
               }
            break
         }
      }
```

4.38.8 Sending a Request to Switch from T.38 Fax to Audio

To request a switch from a T.38 Fax session back to an audio session, the application uses the **gc_Extension( )** function, which initiates a RequestMode (H.323) or re-INVITE (SIP) message to actually perform the action. Before initiating the change of coder, the Fax device must be disassociated from the Media device using the **gc_SetUserInfo( )** function. The application receives a GCEV_EXTENSION event to indicate that audio can now be sent and received. The following code provides and example:

```
INT32 switchFromFaxToAudio()
{
   GC_PARM_BLK      *parmblkp = NULL;
   IP_CONNECT       ipConnect;

   ipConnect.version = 0x100;
   ipConnect.mediaHandle = pline->mediaH;

   gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_DISCONNECT,
                           (sizeof(IP_CONNECT)), (void *)(&ipConnect));
   gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);
   gc_util_delete_parm_blk(parmblkp);

   /* Initiate audio codec switch */
   gc_util_insert_parm_ref(&parmblkp, IPSET_SWITCH_CODEC,
                            IPPARM_AUDIO_INITIATE, sizeof(int), NULL);
   gc_Extension(GCTGT_GCLIB_CRN,pline->crn, IPEXTID_CHANGEMODE, parmblkp, NULL, EV_ASYNC);
   gc_util_delete_parm_blk(parmblkp);
}
```

```
INT32 processEvtHandler()
{
   METAEVENT       metaEvent;
   GC_PARM_BLK     *parmblkp = NULL;
   :
   switch (evtType)
   {
      case GCEV_EXTENSIONCMPLT:
         /* received extension complete event for audio initiation*/
         /* do nothing */
      break;

      case GCEV_EXTENSION:
         /* received extension event for media readiness */
         ext_evtblkp = (EXTENSIONEVTBLK *) metaEvent.extevtdatap;
         parmblkp = &ext_evtblkp->parmblk;

         while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
         {
            switch(t_gcParmDatap->set_ID)
            {
               case IPSET_SWITCH_CODEC:
                  switch(t_gcParmDatap->parm_ID)
                  {
                     case IPPARM_READY:
                        /* Ready to send and receive audio */
                        gc_Listen();
                     break;
                     :
                     :
                  }
               break;
            }
         }
      break;
     :
   }
 :
}
```

4.38.9 Receiving a Request to Switch from T.38 Fax to Audio

An application may receive a request to switch from a T.38 Fax session back to an audio session. The request is received as a GCEV_EXTENSION event that is triggered by a RequestMode (H.323) or re-INVITE (SIP) message. Before accepting the incoming request, the application must disassociate the T.38 Fax device from the Media device using the **gc_SetUserInfo( )** function, then continue accepting the request as described in Section 4.38.5, “Accepting/Rejecting a Request to Switch Between Audio and T.38 Fax”, on page 379.

```
INT32 processEvtHandler()
{
   METAEVENT        metaEvent;
   GC_PARM_BLK      *parmblkp = NULL;
   GC_PARM_DATAP    t_gcParmDatap = NULL;
   GC_PARM_BLK      *parmblkp2 = NULL;
   EXTENSIONEVTBLK  *ext_evtblkp = NULL;
   IP_CONNECT       ipConnect;
   :
   switch (evtType)
   {
      case GCEV_EXTENSION:
         /* received extension event, parse PARM_BLK examine
          * extension data
```

```
                 */
                ext_evtblkp = (EXTENSIONEVTBLK *) metaEvent.extevtdatap;
                parmblkp = &ext_evtblkp->parmblk;
                while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
                {
                   switch(t_gcParmDatap->set_ID)
                   {
                      case IPSET_SWITCH_CODEC:
                         switch(t_gcParmDatap->parm_ID)
                         {
                            case IPPARM_AUDIO_REQUESTED:
                               /* disconnect the media and fax devices */
                               ipConnect.version = 0x100;
                               ipConnect.mediaHandle = pline->mediaH;

                               gc_util_insert_parm_ref(&parmblkp2, IPSET_FOIP, IPPARM_T38_DISCONNECT,
                                                       sizeof(IP_CONNECT), (void *)(&ipConnect));
                               gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);
                               gc_util_delete_parm_blk(parmblkp2);


                               /* accept audio request by example 4.3.3 */
                               acceptCodecSwitchRequest();
                            break;

                            case IPPARM_READY:
                               /* Ready to send and receive audio */
                               gc_Listen();
                            break;
                         }
                      break;
                      :
                   }
                }
             break;
            :
          }
        :
       }
```

4.38.10 Terminating a Call After a T.38 Fax Session

After a T.38 fax session is finished, and prior to issuing **gc_DropCall( )**, the T.38 Fax device needs to be disassociated from the Media device using the **gc_SetUserInfo( )** function. The following code provides and example.

```
INT32 processEvtHandler()
{
METAEVENT      metaEvent;
GC_PARM_BLK    *parmblkp = NULL;
IP_CONNECT     ipConnect;
   :
   switch (evtType)
      {
         case GCEV_DISCONNECTED:
            /* received extension event, parse PARM_BLK examine extension data */

            /* disconnect the media and fax devices */
            ipConnect.version = 0x100;
            ipConnect.mediaHandle = pline->mediaH;

            gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_DISCONNECT,
                                    sizeof(IP_CONNECT), (void *)(&ipConnect));
            gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);
            gc_util_delete_parm_blk(parmblkp);

            /* dropcall */
            gc_DropCall(pline->crn, GC_NORMAL_CLEARING, EV_ASYNC);
          break;
        }
      :
    }
```

4.38.11 Handling non-2xx Responses to T.38 Fax Switch

HMP software provides 1PCC Global Call support for RFC 3261 compliance for non-2xx responses to re-INVITE requests to switch to or from audio to T.38 fax and back.

Without this feature, when a Global Call SIP application initiates a media type switch from/to audio or to/from fax within a dialog with a re-INVITE request, the existing media session and dialog are terminated if the request is rejected by the UAS with any non-2xx response. RFC 3261 clearly requires that the UAC keep the existing session in a dialog alive as though the re-INVITE never occurred.

With this feature, the existing media session within the dialog remains active upon a switching request from one media type to another (fax to audio or audio to fax).

This feature is enabled by default when the application calls the **gc_ReqModifyCall( )** function or the **gc_Extension( )** function with the codec switch value. On failure to switch, the application will receive the failure events, GCEV_REQ_MODIFY_REJ/GCEV_REQ_MODIFY_FAIL and the GCEV_EXTENSION event with parm ID set to IPPARM_REJECT for set ID IPSET_SWITCH_CODEC respectively. The existing media session will be reestablished underneath, and the requested local media information will contain the stored (existing) media information.

Because this feature is limited to Manual operating mode, an application must be configured in Manual mode to control the association and disassociation of media and T.38 fax devices during each call. The mode of operation is set on a board-level basis. The operating mode for set ID/parm ID pair IPSET_CONFIG/IPPARM_OPERATING_MODE must be set to either of the following:

- IP_T38_MANUAL_MODE
- IP_T38_MANUAL_MODIFY_MODE

For additional information, see Section 3.4, “T.38 Fax Server Call Scenarios”, on page 89.

4.38.11.1 Manual Mode Examples

Switch from T.38 Fax to Audio is Unsuccessful

This example demonstrates Manual mode when the switch from T.38 fax to audio is unsuccessful.

```
INT32 switchFromFaxToAudio( )
{
      GC_PARM_BLK *parmblkp = NULL;

      IP_CONNECT ipConnect;
      ipConnect.version = 0x100;
```

```
        ipConnect.mediaHandle = pline->mediaH;

        gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_DISCONNECT,
                               (sizeof(IP_CONNECT)), (void *)(&ipConnect));

        gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);

        gc_util_delete_parm_blk(parmblkp);
        /* Initiate audio codec switch */
        gc_util_insert_parm_ref(&parmblkp, IPSET_SWITCH_CODEC, IPPARM_AUDIO_INITIATE, sizeof(int),
                                NULL);

        gc_Extension(GCTGT_GCLIB_CRN,pline->crn, IPEXTID_CHANGEMODE, parmblkp, NULL, EV_ASYNC);
        gc_util_delete_parm_blk(parmblkp);
}

INT32 processEvtHandler()
{
        METAEVENT metaEvent;
        GC_PARM_BLK *parmblkp = NULL;
        GC_INFO    t_info;

        switch (evtType)
        {
            case GCEV_EXTENSIONCMPLT:
            /* received extension complete event for audio initiation*/
            /* do nothing */
            break;

            case GCEV_EXTENSION:
            /* received extension event for media readiness */
            ext_evtblkp = (EXTENSIONEVTBLK *) metaEvent.extevtdatap;
            parmblkp = &ext_evtblkp->parmblk;
            while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
            {
            switch(t_gcParmDatap->set_ID)
            {
                 case IPSET_SWITCH_CODEC:
                 switch(t_gcParmDatap->parm_ID)
            {
                 case IPPARM_REJECT:
                  gc_ResultInfo(&metaEvent,&t_info);
                  gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_CONNECT,
                                         (sizeof(IP_CONNECT)), (void *)(&ipConnect));

                  gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);
                  break;
                  case IPPARM_READY:
                  /* Ready to send and receive audio */
                    gc_Listen();
                  break;
             }
}
```

Switch from Audio to T.38 Fax is Unsuccessful

This example demonstrates Manual mode when the switch from audio to fax is unsuccessful.

```
INT32 processEvtHandler( )
{
METAEVENT metaEvent;
GC_PARM_BLK *parmblkp = NULL;
IP_CONNECT ipConnect;
GC_INFO    t_info;
```

```
switch (evtType)
{

      case GCEV_CONNECTED:
      /* received Connect event */
      /* in conversation */
      ipConnect.version = 0x100;
      ipConnect.mediaHandle = pline->mediaH;
      ipConnect.faxHandle = pline->faxH;
      ipConnect.connectType = IP_FULLDUP;
      gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_CONNECT,
                             (sizeof(IP_CONNECT)), (void *)(&ipConnect));

      gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);

      gc_util_delete_parm_blk(parmblkp);
      /* Initiate T.38 codec switch */
      gc_util_insert_parm_ref(&parmblkp,IPSET_SWITCH_CODEC,IPPARM_T38_INITIATE,
                             sizeof(int), NULL);

      gc_Extension(GCTGT_GCLIB_CRN,pline->crn,IPEXTID_CHANGEMODE, parmblkp, NULL, EV_ASYNC);
      gc_util_delete_parm_blk(parmblkp);
      break;

      case GCEV_EXTENSIONCMPLT:
      /* received extension complete event for T.38 initiation*/
      /* do nothing */
      break;

      case GCEV_EXTENSION:
      /* received extension event for media readiness */
      ext_evtblkp = (EXTENSIONEVTBLK *) metaEvent.extevtdatap;
      parmblkp = &ext_evtblkp->parmblk;
      while (t_gcParmDatap = gc_util_next_parm(parmblkp, t_gcParmDatap))
      {
      switch(t_gcParmDatap->set_ID)
      {
             case IPSET_SWITCH_CODEC:
             switch(t_gcParmDatap->parm_ID);
             {
             case IPPARM_REJECT:

                gc_ResultInfo(&metaEvent,&t_info);

                gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP,
                                        IPPARM_T38_DISCONNECT,(sizeof(IP_CONNECT)), (void
                                        *)(&ipConnect));

                gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);

                gc_Listen();
                /* IPT to IPM*/
                break;
            case IPPARM_READY:
            /* Ready to send and receive fax */
            fx_sendfax();
            break;
        }
        break;
}
```

4.38.11.2 Manual Modify Mode Examples

Switch from T.38 Fax to Audio is Unsuccessful

This example demonstrates Manual Modify mode when the switch from fax to audio is unsuccessful.

```
INT32 switchFromFaxToAudio()
{
      GC_PARM_BLK *parmblkp = NULL;
      IP_CONNECT ipConnect;
      ipConnect.version = 0x100;
      ipConnect.mediaHandle = pline->mediaH;

      gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_DISCONNECT,
                              (sizeof(IP_CONNECT)), (void *)(&ipConnect));

      gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp,GC_SINGLECALL);

      gc_util_delete_parm_blk(parmblkp);
      /* Initiate audio codec switch */
      if( gc_util_insert_parm_ref(&parmblkp,GCSET_CHAN_CAPABILITY, IPPARM_LOCAL_CAPABILITY,
          sizeof(IP_CAPABILITY), &ipcap) != GC_SUCCESS )
      {
       //error
      }
      gc_ReqModifyCall (GCTGT_GCLIB_CRN,pline->crn, parmblkp, EV_ASYNC);
      gc_util_delete_parm_blk(parmblkp);
}
INT32 processEvtHandler()
{
     METAEVENT metaEvent;
     GC_PARM_BLK *parmblkp = NULL;
     switch (evtType)
     {
         case GCEV_EXTENSIONCMPLT:
         /* received extension complete event for audio initiation*/
         /* do nothing */
         break;

         case GCEV_MODIFY_CALL_ACK:
        // switch complete
          gc_Listen();
        break;

        case GCEV_MODIFY_CALL_REJ:
        case GCEV_MODIFY_CALL_FAIL:
        gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_CONNECT,
                                (sizeof(IP_CONNECT)), (void *)(&ipConnect));
        break;
      }
```

Switch from Audio to T.38 Fax is Unsuccessful

This example demonstrates Manual Modify mode when the switch from audio to fax is unsuccessful.

```
INT32 processEvtHandler()
{
METAEVENT metaEvent;
GC_PARM_BLK *parmblkp = NULL;
IP_CONNECT ipConnect;
```

```
switch (evtType)
{
case GCEV_CONNECTED:
     /* received Connect event */
     /* in conversation */
     ipConnect.version = 0x100;
     ipConnect.mediaHandle = pline->mediaH;
     ipConnect.faxHandle = pline->faxH;
     ipConnect.connectType = IP_FULLDUP;
     gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP, IPPARM_T38_CONNECT,
                             (sizeof(IP_CONNECT)), (void *)(&ipConnect));
     gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);

     if( gc_util_insert_parm_ref(&parmblkp, GCSET_CHAN_CAPABILITY,
         IPPARM_LOCAL_CAPABILITY, sizeof(IP_CAPABILITY), &ipcap) != GC_SUCCESS )
     {
     //error
     }
     gc_ReqModifyCall (GCTGT_GCLIB_CRN,pline->crn, parmblkp, EV_ASYNC);
     gc_util_delete_parm_blk(parmblkp);
     break;

     case GCEV_MODIFY_CALL_ACK:
         // Switch Complete
         fx_sendfax();
         break;

     case GCEV_MODIFY_CALL_REJ:
     case GCEV_MODIFY_CALL_FAIL:

          /* received extension event for media readiness */
          gc_util_insert_parm_ref(&parmblkp, IPSET_FOIP,
                    IPPARM_T38_DISCONNECT,(sizeof(IP_CONNECT)), (void *)(&ipConnect));

          gc_SetUserInfo(GCTGT_GCLIB_CRN, pline->crn, parmblkp, GC_SINGLECALL);

          gc_Listen();
          /* IPT to IPM*/
                break;
}
```

4.39 Sending and Receiving V.17 Faxes

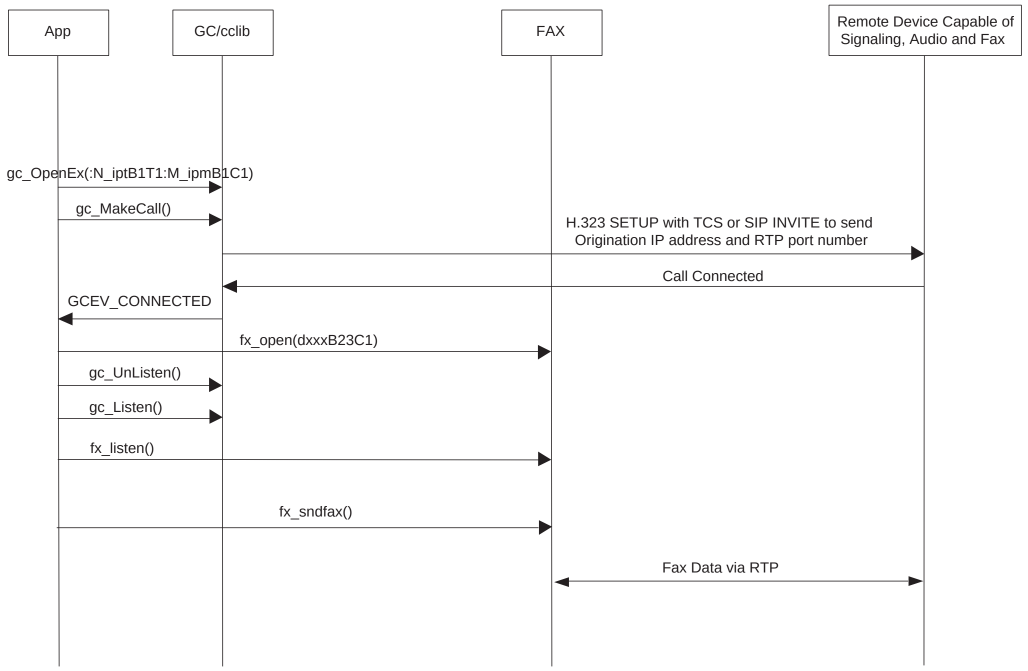
The Dialogic® HMP system is capable of originating and receiving a fax using a V.17 soft modem. This facility can be used with the Dialogic® Global Call API library routing the V.17 PCM data over a G.711 coder in a "fax pass-through" mode by connecting the fax (dxxx) device directly to the media (ipm) device via a PCM connection.

Alternatively, the V.17 PCM data can be routed over a PSTN connection by connecting the fax (dxxx) device directly to the DTI front end device (refer to the *Dialogic® Fax Software Reference*).

4.39.1 Sending G.711 Fax in an Established Audio Session

In the scenario shown in Figure 60, the user application uses the Dialogic® Global Call API to open a Media device and make a voice call. A Fax device is then opened and the application connects the Fax device to the voice session.

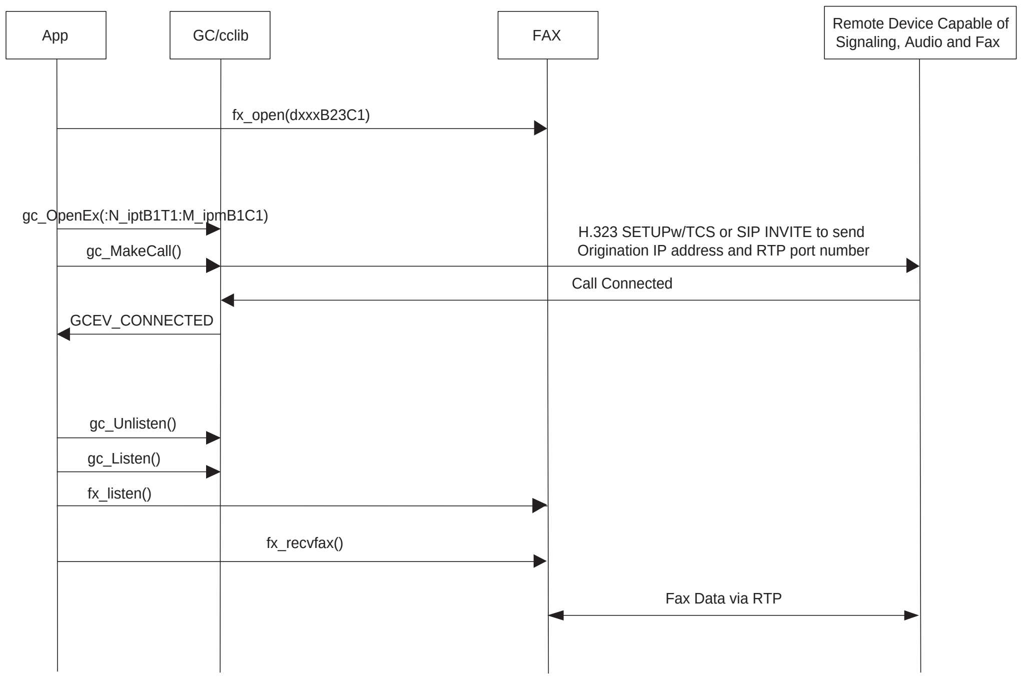
Figure 60. Sending G.711 Fax in an Established Audio Session


App
GC/cclib
FAX
Remote Device Capable of Signaling, Audio and Fax
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
gc_MakeCall()
H.323 SETUP with TCS or SIP INVITE to send Origination IP address and RTP port number
Call Connected
GCEV_CONNECTED
fx_open(dxxxB23C1)
gc_UnListen()
gc_Listen()
fx_listen()
fx_sndfax()
Fax Data via RTP

4.39.2 Receiving G.711 Fax in an Established Audio Session

In the scenario shown in Figure 61, the user application uses the Dialogic® Global Call API to open a Media device and establishes an audio session with the remote device. To prepare to receive a fax, the application must also open a Fax device.

Figure 61. Receiving G.711 Fax in an Established Audio Session


App
GC/cclib
FAX
Remote Device Capable of Signaling, Audio and Fax
fx_open(dxxxB23C1)
gc_OpenEx(:N_iptB1T1:M_ipmB1C1)
H.323 SETUPw/TCS or SIP INVITE to send Origination IP address and RTP port number
gc_MakeCall()
Call Connected
GCEV_CONNECTED
gc_Unlisten()
gc_Listen()
fx_listen()
fx_recvfax()
Fax Data via RTP

4.40 Using Object Identifiers

Object Identifiers (OIDs) are not free strings, they are standardized and assigned by various controlling authorities such as, the International Telecommunications Union (ITU), British Standards Institute (BSI), American National Standards Institute (ANSI), Internet Assigned Numbers Authority (IANA), International Standards Organization (ISO), and public corporations. Depending on the authority, OIDs use different encoding and decoding schemes. Vendors, companies, governments and others may purchase one or more OIDs to use while communicating with another entity on the network. For more information about OIDs, see *http://www.alvestrand.no/objectid/*.

An application may want to convey an OID to the remote side. This can be achieved by setting the OID string in any nonstandard parameter that can be sent in any Setup, Proceeding, Alerting, Connect, Facility, or User Input Indication (UII) message.

The Dialogic® Global Call API supports the use of any valid OID by allowing the OID string to be included in the GC_PARM_BLK associated with the specific message using the relevant parameter set ID and parameter IDs. Dialogic® Global Call API will not send an OID that is not in a valid format. For more information on the valid OID formats see *http://asn-1.com/x660.htm* which defines the general procedures for the operation of OSI (Open System Interconnection) registration authorities.

The application is responsible for the validity and legality of any OID used.

4.41 LAN Disconnection Alarms

The Dialogic® Global Call API IP Call Control library allows applications to receive notification of a disruption of traffic over the host network interface. The network disconnection notification uses the standard GCAMS alarm mechanism.

The Global Call IP Call Control library provides facilities to notify applications when there is a disruption of a host LAN connection that is handling call control signaling traffic, and when any such disruption is corrected. The most common cause of such a LAN disruption is cable disconnection, but any disruption of the LAN connection will cause the alarm to be sent to board devices that have registered for it. LAN status is monitored on a 4 second loop.

Signaling LAN disconnect (Alarm State ON) and recovery (Alarm State OFF) alarms are generated on a virtual board device level using the standard GCAMS mechanism. If multiple board devices are connected to different ports on the same NIC (rather than separate NICs), all of those devices that have registered for the alarm will receive alarm events when the NIC's LAN connection fails or when it is restored after a disconnection. There is a single disconnect alarm event and a single corresponding recovery event for each LAN disconnection on each virtual board.

The signaling LAN disconnect and recovery alarms are only reported via asynchronous GCAMS events. There is no mechanism for determining the LAN cable alarm status on demand. The signaling LAN disconnect alarm is not designated as a blocking or non-blocking GCAMS alarm because it is a board device level alarm rather than a line device level alarm. Refer to the *Dialogic® Global Call API Library Reference* and *Dialogic® Global Call API Programming Guide* for more information on GCAMS facilities.

The call control library does not take any action (for example, disconnecting an already set up call) in response to LAN disconnection alarm events. It is up to the application whether or not to take any action when alarm events are received. If the application does not take any action when a LAN disconnect alarm is received, the following behavior applies under the circumstances described:

- Already established calls will not be affected unless the LAN connection that has failed is carrying the media traffic as well as the signaling traffic. (Media LAN disconnection is not reported by the signaling LAN disconnect alarm.)
- A call that is in the process of being established will be disconnected by the Call Control library due to the signaling failure, and the application will be notified of the disconnection via existing Global Call disconnect events with appropriate disconnection reasons.
- If the application ignores the LAN disconnect error and tries to make a new call over the disconnected LAN connection, the call will fail and the application will be notified of the reason via existing Global Call events.

If a LAN disconnection failure occurs during application startup, no GCAMS alarm event will be generated, because there is no virtual board which is started up to receive the alarm. There will also be no alarm events generated for applications using the NIC address associated with the system loopback adapter (typically IP address 127.0.0.1) because the signaling never leaves the system in this case.

To enable the receipt of signaling LAN disconnect alarm events, the application must perform the following general steps:

- Explicitly open the board device.
- Register the device handle (from the open operation) with GCAMS using the Global Call function gc_SetAlarmNotifyAll( ). This registration uses the wildcard Alarm Source Object (ASO) ID, ALARM_SOURCE_ID_NETWORK_ID, because the IP Call Control library ASO ID is not known at this point.

When an alarm event is received, the alarm number, the alarm name, the ASO ID and the ASO name can be retrieved using standard Global Call alarm APIs. The retrieved alarm number is equal to TYPE_LAN_DISCONNECT for a disconnect alarm or TYPE_LAN_DISCONNECT + 0x10 for a reconnect alarm event. The retrieved alarm name will be “Lan Cable Disconnected” or “Lan cable connected”. The retrieved ASO ID will be “IPCCLIBAsoId”.

The following code illustrates how signaling LAN disconnect alarms are enabled and handled.

```
main()
{
   /* Initialize the SRL mode for the application */
   #ifdef _WIN32
      int   mode = SR_STASYNC;
      sr_setparm(SRL_DEVICE, SR_MODELTYPE, &mode)
   #else
      int   mode = SR_POLLMODE;
      sr_setparm(SRL_DEVICE, SR_MODEID, &mode)
   #endif

   /* Open the board device */
   sprintf(DevName,":N_iptB1:P_IP");
   rc = gc_OpenEx(&boarddev,DevName,EV_ASYNC,(void *)NULL);

   /* Enable Alarm notification on the board handle with generic ASO ID*/
   gc_SetAlarmNotifyAll (boarddev, ALARM_SOURCE_ID_NETWORK_ID, ALARM_NOTIFY);

   /* -- Forever loop where the main work is done - wait for an event or user requested exit */
   for (;;)
   {
      ret = sr_waitevt(500);              /* 1/2 second */
      if (ret != -1)
      {                    /* i.e. not timeout */
         process_event();
      }
   }
}

process_event()
{
   METAEVENT         metaevent;

   gc_GetMetaEvent(&metaevent)
   evttype = metaevent.evttype;

   switch (evttype)
   {
      case GCEV_ALARM:
         print_alarm_info(&metaevent);
         break;
   }
}
```

```
print_alarm_info(&metaevent);
{
   long              alarm_number;
   char              *alarm_name;
   unsigned long     alarm_source_objectID;
   char              *alarm_source_object_name;

   gc_AlarmNumber(metaeventp, &alarm_number);
                  // Will be of type TYPE_LAN_DISCONNECT = 0x01
                  // or TYPE_LAN_DISCONNECT + 0x10 (LAN connected).
   gc_AlarmName(metaeventp, &alarm_name);
                  // Will be  "Lan Cable Disconnected" or "Lan cable connected".
   gc_AlarmSourceObjectID(metaeventp, &alarm_source_objectID);
                  // Will usually be = 7.
   gc_AlarmSourceObjectName(metaeventp, &alarm_source_object_name)
                  // Will be "IPCCLIBAsoId"
   printf("Alarm %s (0x%lx) occurred on ASO %s (%d)", alarm_name, alarm_number,
           alarm_source_object_name, (int) alarm_source_objectID);
}
```

4.42 Setting and Retrieving Q.931 Message IEs

The Dialogic® Global Call API supports the setting and retrieving of Information Elements (IEs) in selected Q.931 messages. The level of support is described in the following topics:

- Enabling Access to Q.931 Message IEs
- Supported Q.931 Message IEs
- Setting Q.931 Message IEs
- Retrieving Q.931 Message IEs
- Common Usage Scenarios Involving Q.931 Message IEs
- Accessing Additional Q.931 Message IEs

4.42.1 Enabling Access to Q.931 Message IEs

The ability to set and retrieve Q.931 message IEs is an optional feature that can be enabled or disabled at the time the **gc_Start( )** function is called.

The mandatory **INIT_IP_VIRTBOARD( )** function populates the IP_VIRTBOARD structure with default values. The default value of the h323_msginfo_mask field in the IP_VIRTBOARD structure disables access to Q.931 message information elements. The default value of the h323_msginfo_mask field must therefore be overridden with the value IP_H323_MSGINFO_ENABLE for each ipt board device on which the feature is to be enabled. The following code snippet provides an example for two virtual boards:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE; /* override Q.931 message default */
ip_virtboard[1].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE; /* override Q.931 message default */
```

Setting the h323_msginfo_mask field to a value of IP_H323_MSGINFO_ENABLE enables the setting or retrieving of all supported Q.931 message information elements collectively. Enabling and disabling access to individual Q.931 message information elements is **not** supported.

Note: Features that are enabled or configured via the IP_VIRTBOARD structure cannot be disabled or reconfigured once the library has been started. All items set in these data structures take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

4.42.2 Supported Q.931 Message IEs

Table 14 shows the supported Q.931 message Information Elements (IEs), the parameter set ID and parameter ID that should be included in a GC_PARM_BLK when setting or retrieving the IEs, and the maximum allowed length of the IE value.

Table 14. Supported Q.931 Message Information Elements

IE Name	Set/Get	Set ID	Parameter ID	Maximum Length
Bearer Capability	Get and Set	IPSET_CALLINFO	IPPARM_BEARERCAP	255
Facility	Get and Set	IPSET_CALLINFO	IPPARM_FACILITY	255
Called Party	Get and Set	IPSET_CALLINFO	IPPARM_CDPN_ NUMBERING_PLAN_ID	
Called Party	Get and Set	IPSET_CALLINFO	IPPARM_CDPN_ TYPE_OF_NUMBER	
Calling Party	Get and Set	IPSET_CALLINFO	IPPARM_CGPN_ NUMBERING_PLAN_ID	
Calling Party	Get and Set	IPSET_CALLINFO	IPPARM_CGPN_ TYPE_OF_NUMBER	
Calling Party	Get and Set	IPSET_CALLINFO	IPPARM_CGPN_ PRESENTATION_ INDICATOR	
Calling Party	Get and Set	IPSET_CALLINFO	IPPARM_CGPN_ SCREENING_ INDICATOR	
Note: These parameters are character arrays with the maximum size of the array equal to the maximum length shown.				

4.42.3 Setting Q.931 Message IEs

The Dialogic® Global Call API library supports the setting of the following Information Elements (IEs) in the following *outgoing* Q.931 messages:

- Bearer Capability IE in a SETUP message
- Called party number and calling party number IEs in a SETUP message
- Facility IE in SETUP, CONNECT, and FACILITY messages

The **gc_SetUserInfo( )** function is used to set these IEs. The appropriate function parameters in this context are:

- **target_type** – GCTGT_GCLIB_CHAN
- **target_id** – line device
- **infoparmblkp** – a GC_PARM_BLK containing the IPSET_CALLINFO parameter set ID and one of the following parameter IDs:
  - IPPARM_BEARERCAP
  - IPPARM_FACILITY
  - IPPARM_CDPN_*xxxx*
  - IPPARM_CGPN_*xxxx*
- **duration** – GC_SINGLECALL (GC_ALLCALLS is not supported in this context)

4.42.4 Retrieving Q.931 Message IEs

The Dialogic® Global Call API library supports the retrieval of the following Information Elements (IEs) from the following *incoming* Q.931 messages:

- Bearer Capability IE in a SETUP message
- Called party number and calling party number IEs in a SETUP message
- Facility IE in SETUP, CONNECT, and FACILITY messages

Table 15 shows the Dialogic® Global Call API events generated for incoming Q.931 messages and the parameter set ID and parameter IDs contained in the GC_PARM_BLK associated with each event.

Table 15. Supported IEs in Incoming Q.931 Messages

Incoming Q.931 Message	Dialogic® Global Call API Event	Set ID	Parm ID
SETUP	GCEV_OFFERED	IPSET_CALLINFO	IPPARM_BEARERCAP
SETUP	GCEV_OFFERED	IPSET_CALLINFO	IPPARM_FACILITY
SETUP	GCEV_EXTENSION CMPLT	IPSET_CALLINFO	IPPARM_CDPN_*xxxx*
SETUP	GCEV_EXTENSION CMPLT	IPSET_CALLINFO	IPPARM_CGPN_*xxxx*
CONNECT	GCEV_CONNECTED	IPSET_CALLINFO	IPPARM_FACILITY
FACILITY	GCEV_EXTENSION with an ext_id of EXTID_RECEIVEMSG	IPSET_CALLINFO	IPPARM_FACILITY

Note: The application must retrieve the necessary IEs by copying them into its own buffer before the next call to **gc_GetMetaEvent( )**. Once the next **gc_GetMetaEvent( )** call is issued, the Q.931 information is no longer available.

4.42.5 Common Usage Scenarios Involving Q.931 Message IEs

Table 16 shows how the Dialogic® Global Call API handles common scenarios that involve the use of Q.931 message IEs.

Table 16. Common Usage Scenarios Involving Q.931 Message IEs

Scenario	Behavior
The application invokes **gc_SetUserInfo( )** to set the Bearer Capability IE, then invokes **gc_MakeCall( )**	The Bearer Capability IE is parsed and added to the new outgoing SETUP message.
The application invokes **gc_SetUserInfo( )** to set the Facility IE, then invokes **gc_MakeCall( )**	The Facility IE is parsed and added to the new outgoing SETUP message.
The application invokes **gc_SetUserInfo( )** to set the Bearer Capability IE and the Facility IE, then invokes **gc_MakeCall( )**	The Bearer Capability IE and the Facility IE are parsed and added to the new outgoing SETUP message.
The application invokes **gc_SetUserInfo( )** to set the Facility IE, then invokes **gc_AnswerCall( )**	The Facility IE is parsed and added to the new outgoing CONNECT message.
The application invokes **gc_SetUserInfo( )** to set the Facility IE, then invokes **gc_Extension( )**	The Facility IE is parsed and added to the new outgoing FACILITY message.
The application receives a GCEV_OFFERED event with a Bearer Capability IE	The application retrieves the Bearer Capability IE using **gc_GetMetaEvent( )** and **gc_util_next_parm( )**.
The application receives a GCEV_OFFERED event with a Facility IE	The application retrieves the Facility IE using **gc_GetMetaEvent( )** and **gc_util_next_parm( )**.
The application receives a GCEV_OFFERED event with Bearer Capability IE and Facility IE	The application retrieves the Bearer Capability IE and Facility IE using **gc_GetMetaEvent( )** and **gc_util_next_parm( )**.
The application receives a GCEV_CONNECTED event with a Facility IE	The application retrieves the Facility IE using **gc_GetMetaEvent( )** and **gc_util_next_parm( )**.
The application receives a GCEV_EXTENSION event with a Facility IE	The application retrieves the Facility IE using **gc_GetMetaEvent( )** and **gc_util_next_parm( )**.

4.42.6 Accessing Additional Q.931 Message IEs

You can access fields in the calling party number (CGPN) and called party number (CDPN) IEs within a H.225/Q.931 SETUP message.

The SETUP message is a standard call signaling message used by a calling H.323 entity to establish a connection with the called entity. Several CPN fields can be modified by the application via **gc_SetUserInfo**( ) or when invoking the **gc_MakeCall**( ) function. You can use the IPSET_CALLINFO set ID and its parameter IDs to send and receive these CPN fields via the Global Call API library over an IP network.

The parameter IDs for CGPN and CDPN are as follows:

Parameter ID	Description
IPPARM_CGPN_TYPE_OF_NUMBER IPPARM_CDPN_TYPE_OF_NUMBER	Contains the type of number in the CGPN or CDPN
IPPARM_CGPN_NUMBERING_PLAN_ID IPPARM_CDPN_NUMBERING_PLAN_ID	Contains the numbering plan identification in the CGPN or CDPN
IPPARM_CGPN_SCREENING_INDICATOR	Contains the screening indicator in the CGPN
IPPARM_CGPN_PRESENTATION_INDICATOR	Contains the presentation indicator in the CGPN

The following data variables are defined in the *gcip.h* file:

```
typedef unsigned char   CPN_TON;    /* Type of number */
typedef unsigned char   CPN_NPI;    /* Numbering plan identification */
typedef unsigned char   CPN_SI;     /* Screening Indicator       */
typedef unsigned char   CPN_PI;     /* Presentation Indicator */
```

4.42.6.1 Enabling Setting and Retrieval of Q.931 Message IEs

To enable setting and retrieval of all supported Q.931 message IEs, set the h323_msginfo_mask field in IP_VIRTBOARD structure to a value of IP_H323_MSGINFO_ENABLE for each IPT board device **before** calling **gc_Start( )**.

Note: By default the underlying H.323 stack is not enabled to receive incoming Q.931 message IEs.

The following code snippet shows how to enable this feature.

```
IP_VIRTBOARD virtBoard[MAX_BOARDS];
memset(virtBoard,0,sizeof(IP_VIRTBOARD) * MAX_BOARDS);
bid = 1;
INIT_IP_VIRTBOARD(&virtBoard[bid]);
// fill up other board parameters
….
 virtBoard[bid].localIP.ip_ver = IPVER4;
 virtBoard[bid].localIP.u_ipaddr.ipv4 = (unsigned int) IP_CFG_DEFAULT;
…
virtBoard[1].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE;
//Then use the virtBoard structure in the gc_Start() function appropriately
```

You can also enable reception of other H.323 fields simultaneously via the code:

```
virtBoard[1].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE | IP_H323_ANNEXMMSG_ENABLE;
```

For more information on enabling the setting and retrieving of all supported Q.931 message IEs, see Section 4.42.1, “Enabling Access to Q.931 Message IEs”, on page 394.

4.42.6.2 Stopping the Reception of CPN Information

Currently, there is no way for the user application to turn off the reception of Q.931 IEs received by the underlying H323 stack once they are enabled, without stopping the application or restarting the stack.

4.42.6.3 Setting up CPN fields in the GC_PARM_BLK data structure

Before calling the **gc_MakeCall( )** function, you must set up the CPN fields to be included in the GC_PARM_BLK data structure. The GC_PARM_BLK should include IPSET_CALLINFO and the parameter IDs described in Section 4.42.6, "Accessing Additional Q.931 Message IEs", on page 397, which specifies which CPN fields are to be set in the parameter block structure of a Make Call block.

To set up the CPN fields in the GC_PARM_BLK structure, call the **gc_util_insert_parm_ref( )** function.

The following is an example of how to specify the CPN fields for sending.

```
#include <stdio.h>
#include <string.h>
#include <gcip.h>
#include <.h>


void main()
{
   CPN_TON    cgpn_ton, cdpn_ton;
   CPN_NPIcgpn_npi, cdpn_npi;
   CPN_SIcgpn_si;
   CPN_PIcgpn_pi;
   GC_PARM_BLKPpParmBlock;

  /*. .   Main Processing….*/
  /*  Set CPN fields in the Make Call Block to be sent out via SETUP message */

  cgpn_ton = 0x1;   // Note that the field values must be valid.
  cdpn_ton = 0x4;
  cgpn_npi = 0x1;
  cdpn_npi = 0x1;
  cgpn_si = 0x0;
  cgpn_pi = 0x0;

    gc_util_insert_parm_ref(&pParmBlock,
                            IPSET_ CALLINFO,
                            IPPARM_CGPN_TYPE_OF_NUMBER,
                            sizeof(unsigned char),
                            &cgpn_ton);

    gc_util_insert_parm_ref(&pParmBlock,
                            IPSET_ CALLINFO,
                            IPPARM_CDPN_TYPE_OF_NUMBER,
                            sizeof(unsigned char),
                            &cdpn_ton);

    gc_util_insert_parm_ref(&pParmBlock,
                            IPSET_ CALLINFO,
                            IPPARM_CGPN_NUMBERING_PLAN_ID,
                            sizeof(unsigned char),
                            &cgpn_npi);

    gc_util_insert_parm_ref(&pParmBlock,
                            IPSET_ CALLINFO,
                            IPPARM_CDPN_NUMBERING_PLAN_ID,
                            sizeof(unsigned char),
                            &cdpn_npi);
    gc_util_insert_parm_ref(&pParmBlock,
                            IPSET_ CALLINFO,
                            IPPARM_CGPN_SCREENING_INDICATOR,
                            sizeof(unsigned char),
                            &cgpn_si);

    gc_util_insert_parm_ref(&pParmBlock,
                            IPSET_ CALLINFO,
                            IPPARM_CGPN_PRESENTATION_INDICATOR,
                            &cgpn_pi);

  /*. .. Continue Main processing. … call gc_MakeCall() */
}
```

4.42.6.4 Generating CPN Fields in a SETUP message

You can choose to insert the CPN data into the GC_MAKECALL_BLK using **gc_SetUserInfo( )**.

After setting the CPN signaling message fields as described in Setting up CPN fields in the GC_PARM_BLK data structure, the application calls the **gc_MakeCall( )** function passing it a pointer to the GC_MAKECALL_BLK.

4.42.6.5 Retrieving CPN Information

When the underlying H.323 stack is enabled to retrieve incoming call info fields, including CPN information, any CPN fields detected in an associated H.225 message will be retrieved and stored in Global Call.

Use the **gc_Extension( )** function to retrieve the CPN data within any valid incoming H.225 message containing CPN fields, while a call is in any state, after the OFFERED state. This is similar to receiving other call related information via the GCEV_EXTENSIONCMPLT event received as a termination event to the **gc_Extension( )** function.

Set the target_type to GCTGT_GCLIB_CRN. Set the target_id to the actual CRN. If incoming CPN data is available, the information is included with the corresponding GCEV_EXTENSIONCMPLT asynchronous termination event.

The extevtdatap field in the METAEVENT structure for the GCEV_EXTENSIONCMPLT event is a pointer to an EXTENSIONEVTBLK structure that contains a GC_PARM_BLK with the requested CPN information.

When trying to retrieve the CPN information, it is necessary to specify each of the CPN parameters in the extension request, for which information from the stack is needed.

4.42.6.6 Q.931 Message IE Examples

Specifying CPN Field for Receiving

An example of how to specify the Calling Number Type of Number field for receiving is shown below. The method to specify the other CPN fields would be similar.

```
int getCPNInfo(CRN crn)
{
   GC_PARM_BLKP gcParmBlk = NULL;
   GC_PARM_BLKP retParmBlk;
   int frc;

   frc = gc_util_insert_parm_val(&gcParmBlk,
                                 IPSET_CALLINFO,
                                 IPPARM_CGPN_TYPE_OF_NUMBER,
                                 sizeof(unsigned char),1);

   if (GC_SUCCESS != frc)
   {
       return GC_ERROR;
   }

   frc = gc_Extension (GCTGT_GCLIB_CRN,
                       crn,
                       IPEXTID_GETINFO,
                       gcParmBlk,
                       &retParmBlk,
                       EV_ASYNC);
   if (GC_SUCCESS != frc)
   {
       return GC_ERROR;
   }

   gc_util_delete_parm_blk(gcParmBlk);

    return GC_SUCCESS;

}
```

Retrieving CPN Information

An example of how to extract CPN information from an unsolicited GCEV_EXTENSIONCMPLT event received as a result of a request for call-related information is shown below:

```
int OnExtension(GC_PARM_BLKP parm_blk,CRN crn)
{
   GC_PARM_DATA *parmp = NULL;
   parmp = gc_util_next_parm(parm_blk,parmp);

   if (!parmp)
   {
      return GC_ERROR;
   }

   while (NULL != parmp)
   {
     switch (parmp->set_ID)
     {
       case IPSET_CALLINFO:
           switch (parmp->parm_ID)
           {
             case IPPARM_CGPN_TYPE_OF_NUMBER:
                   printf("\tReceived CPN data Calling Party Type of Number: %d\n", (*(unsigned
char*)(parmp->value_buf)));
    break;

    case IPPARM_CDPN_TYPE_OF_NUMBER:
                   printf("\tReceived CPN data Called Party Type of Number: %d\n", (*(unsigned
char*)(parmp->value_buf)));
    break;
```

```
     default:
                    printf("\tReceived unknown extension parmID %d\n",
                    parmp->parm_ID);

     break;
          }

    break;
    parmp = gc_util_next_parm(parm_blk,parmp);
  }
}
```

4.43 Accessing H.323 ALERTING Progress Indicator Information Element

The application has the ability to set and/or retrieve the Progress Indicator Information Element (PIIE) in an H.323 ALERTING message. The application can also disable an automatic PROGRESS message after an ALERTING message is received.

The application can populate and send a new PIIE and include it in the ALERTING response to an incoming call SETUP message. Currently, the Call Control Library automatically sends a PROGRESS message after every ALERTING message. With this feature, a parameter is defined to optionally disable an automatic PROGRESS message in the virtual board. Once disabled, PROGRESS is not sent after receipt of an ALERTING message. By default, this feature is disabled and current behavior is preserved.

The Global Call functions **gc_SetUserInfo( )** and **gc_GetMetaEvent( )** are used to set and get PIIE in an ALERTING message. Both functions use the GC_PARM_BLK and GC_PARM_DATA data structures.

The following parameter ID in the IPSET_CONFIG set ID is used for this feature:

IPPARM_H323_AUTO_PROGRESS_DISABLE
: Disables an automatic H.323 PROGRESS message after ALERTING on a virtual board.

4.43.1 Enabling Access to H.323 IE

In order to access H.323 IE, you must enable IP_H323_MSGINFO_ENABLE in the h323_msginfo_mask field (IP_VIRTBOARD structure) on a virtual board at **gc_Start( )**. The following code snippet illustrates how to enable access to H.323 IE.

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..

/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data */
```

```
INIT_IPCCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);

/* initialize virtual board */
INIT_IP_VIRTBOARD(&virtBoards[0]);

// IP_H323_MSGINFO_ENABLE must be set to activate this feature
virtBoard[bid].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE
```

4.43.2 Disabling an Automatic PROGRESS Message

The following code snippet illustrates how to disable an automatic PROGRESS message on a virtual board:

```
#include "gclib.h"
..
..
GC_PARM_BLK *pParmBlock = NULL;
long t = 0;

gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_CONFIG,
                        IPPARM_H323_AUTO_PROGRESS_DISABLE,
                        sizeof(int),
                         0);

// Set config data
gc_SetConfigData(GCTGT_CCLIB_NETIF,
                 boarddev,
                 pParmBlock,
                 0,
                 GCUPDATE_IMMEDIATE,
                 &t,
                 EV_ASYNC);

gc_util_delete_parm_blk(pParmBlock);
```

4.43.3 Setting the PIIE

Use the **gc_SetUserInfo( )** function to set the Progress Indicator Information Element. The information is not transmitted until calling **gc_AcceptCall( )**. The application overwrites the PIIE in any incoming ALERTING or PROGRESS messages for the same channel.

```
#include "gclib.h"
..
..
GC_PARM_BLK *pParmBlock = NULL;
unsigned char progress_ind[] = {0x1E,0x02,0x80,0x81};

//set up progress indicator
gc_util_insert_parm_ref(&pParmBlock,
                        IPSET_CALLINFO,
                        IPPARM_PROGRESS_IND,
                        sizeof(progress_ind),
                        progress_ind);


// Set Call Information
gc_SetUserInfo(GCTGT_GCLIB_CHAN, ldev, pParmBlock, GC_SINGLECALL);

gc_util_delete_parm_blk(pParmBlock);
gc_AcceptCall(crn, NULL, EV_ASYNC);
```

4.43.4 Retrieving the PIIE

The PIIE values are reported to the application through events processed using the **gc_GetMetaEvent( )**. The following example illustrates how to retrieve PIIE from the GCEV_ALERTING event after receiving an ALERTING message.

```
#include "gclib.h"
..
..
METAEVENT  metaevt;
GC_PARM_BLK *pParmBlock = NULL;
GC_PARM_DATA  *parmp = NULL;

/* Get Meta Event */
gc_GetMetaEvent(&metaevt);

switch(metaevt->evttype){
       .
       .
       .
       case GCEV_ALERTING:
               currentCRN = metaevt->crn;
               pParmBlock = (GC_PARM_BLK*)(metaevt->extevtdatap);
               parmp = NULL;

               /* going thru each parameter block data*/
               while ((parmp = gc_util_next_parm(pParmBlock,parmp)) != 0)
               {
                       switch (parmp->set_ID)
                       {
                       /* Handle the extended information */
                       case IPSET_CALLINFO:
                               switch (parmp->parm_ID)
                               {
                               case IPPARM_PROGRESS_IND:
                               if(parmp->data_size != 0)
                               {

                                      printf("\tGot PIIE: ");
                                      for(unsigned int pii= 0;pii<parmp->data_size;pii++)
                                      printf ("0x%x ",*(((unsigned char*)(parmp->pData))+pii));
                                      printf ("\n");
                                }
                                }
                                 break;
                        }
                         break;
                  }
          .
          .
          .
}
```

4.44 Sending Nonstandard Protocol Messages (H.323)

The Dialogic® Global Call API library allows applications that are using the H.323 protocol to send certain messages that contain Nonstandard Data. This capability is supported for the following message types:

- User Input Indication (UII) message (H.245)

- Facility messages (Q.931)
- Registration messages

Table 17 summarizes the set IDs and parameter IDs used to send the messages and describes the call states in which each message should be sent.

Table 17. Summary of Protocol Messages that Can be Sent with Nonstandard Data

Type	Set ID & Parameter ID	When Message Should be Sent
Nonstandard UII Message (H.245)	IPSET_MSG_H245 • IPPARM_MSGTYPE value = IP_MSGTYPE_H245_INDICATION	Only when call is in Connected state
Nonstandard Facility Message (Q.931)	IPSET_MSG_Q931 • IPPARM_MSGTYPE value = IP_MSGTYPE_Q931_FACILITY	In any call state
Nonstandard Registration Message	IPSET_MSG_RAS • IPPARM_MSGTYPE value = IP_MSGTYPE_REG_NONSTD	

The maximum length of the Global Call parameter used for the Nonstandard Data information is configured at start-up via the max_parm_data_size field in the IPCCLIB_START_DATA structure. The default size is 255 (for backwards compatibility), but applications may configure it to be as large as 4096 bytes. Applications must use the extended **gc_util_..._ex( )** functions to insert or extract any GC_PARM_BLK parameter elements whose data length is defined to be greater than 255.

Note: In practice, applications may not be able to utilize the full maximum length of the nonstandard data parameter element as configured in max_parm_data_size. The H.323 stack limits the overall size of messages to be max_parm_data_size + 512 bytes, and any messages that exceed this limit are truncated without any notification to the application.

4.44.1 Nonstandard UII Message (H.245)

To send nonstandard UII messages, use the **gc_Extension( )** function in asynchronous mode with an **ext_id** (extension ID) of IPEXTID_SENDMSG. The **target_type** should be GCTGT_GCLIB_CRN and the **target_id** should be the actual CRN. The GC_PARM_BLK must contain parameter elements that identify the message type, the nonstandard data, and the nonstandard data identifier. At the sending end, reception of a GCEV_EXTENSIONCMPLT event indicates that the message has been sent.

The parameter element that identifies the message type is:

IPSET_MSG_H245
    IPPARM_MSGTYPE
- value = IP_MSGTYPE_H245_INDICATION

The parameter element for the Nonstandard Data data is:

IPSET_NONSTANDARDDATA
    IPPARM_NONSTANDARDDATA_DATA
- value = Nonstandard Data string, max length = max_parm_data_size (configurable at library start-up)

The parameter element for the Nonstandard Data identifier is one (and only one) of the following:

IPSET_NONSTANDARDDATA
    IPPARM_NONSTANDARDDATA_OBJID
- value = array of unsigned integers, max length = MAX_NS_PARM_OBJID_LENGTH

IPSET_NONSTANDARDDATA
    IPPARM_H221NONSTANDARD
- value = IP_H221NONSTANDARD structure

When the Dialogic® Global Call API library receives a nonstandard UII message, it generates a GCEV_EXTENSION event with the ext_id value IPEXTID_RECEIVEMSG. The extevtdatap field in the METAEVENT structure for the GCEV_EXTENSION event is a pointer to an EXTENSIONEVTBLK structure which in turn contains a GC_PARM_BLK that includes all of the data in the message.

See Section 9.2.14, “IPSET_MSG_H245”, on page 628 and Section 9.2.19, “IPSET_NONSTANDARDDATA”, on page 631 for more information.

```
.
.
.
/* H245 UII with ObjId and data */

rc = gc_util_insert_parm_val(&t_PrmBlkp, IPSET_MSG_H245, IPPARM_MSGTYPE,
                             sizeof(int), IP_MSGTYPE_H245_INDICATION);

rc = gc_util_insert_parm_ref_ex(&t_PrmBlkp, IPSET_NONSTANDARDDATA,
                                IPPARM_NONSTANDARDDATA_OBJID, ObjLen+1, ObjId);

rc = gc_util_insert_parm_ref_ex(&t_PrmBlkp, IPSET_NONSTANDARDDATA,
                                IPPARM_NONSTANDARDDATA_DATA, DataLen+1, data);

if (rc == -1)
{
   printf("Fail to insert parm");
   return -1;
}
else
   printf("Sending IP H245 UII Message");

gc_Extension(GCTGT_GCLIB_CRN,
             crn,
             IPEXTID_SENDMSG,
             t_PrmBlkp,
             &t_RetBlkp,
             EV_ASYNC);

gc_util_delete_parm(t_PrmBlkp);
.
.
.
```

4.44.2 Nonstandard Facility Message (Q.931)

To send a nonstandard Facility message, use the **gc_Extension**( ) function in asynchronous mode with an **ext_id** (extension ID) of IPEXTID_SENDMSG. The **target_type** should be GCTGT_GCLIB_CRN and the **target_id** should be the actual CRN. The GC_PARM_BLK must contain parameter elements that identify the message type, the nonstandard data, and the nonstandard data identifier. At the sending end, reception of a GCEV_EXTENSIONCMPLT event indicates that the message has been sent.

The parameter element that identifies the message type is:

IPSET_MSG_Q931
  IPPARM_MSGTYPE
  - value = IP_MSGTYPE_Q931_FACILITY

The parameter element for the Nonstandard Data data is:

IPSET_NONSTANDARDDATA
  IPPARM_NONSTANDARDDATA_DATA
  - value = Nonstandard Data string, max length = max_parm_data_size (configurable at library start-up)

The parameter element for the Nonstandard Data identifier is one (and only one) of the following:

IPSET_NONSTANDARDDATA
  IPPARM_NONSTANDARDDATA_OBJID
  - value = array of unsigned integers, max length = MAX_NS_PARM_OBJID_LENGTH

IPSET_NONSTANDARDDATA
  IPPARM_H221NONSTANDARD
  - value = IP_H221NONSTANDARD structure

When the Dialogic® Global Call API library receives a nonstandard Facility message, it generates a GCEV_EXTENSION event with the ext_id value IPEXTID_RECEIVEMSG. The extevtdatap field in the METAEVENT structure for the GCEV_EXTENSION event is a pointer to an EXTENSIONEVTBLK structure which in turn contains a GC_PARM_BLK that includes all of the data in the message.

See Section 9.2.15, “IPSET_MSG_Q931”, on page 628 and Section 9.2.19, “IPSET_NONSTANDARDDATA”, on page 631 for more information.

The following code shows how to set up and send a Q.931 nonstandard facility message.

```
char ObjId[]= "1 22 333 4444";
char NSData[]= "DataField_Facility";

GC_PARM_BLKP    gcParmBlk = NULL;

gc_util_insert_parm_val(&gcParmBlk,
                        IPSET_MSG_Q931,
                        IPPARM_MSGTYPE,
                        sizeof(int),
                        IP_MSGTYPE_Q931_FACILITY);
```

```
gc_util_insert_parm_ref(&gcParmBlk,
                        IPSET_NONSTANDARDDATA,
                        IPPARM_NONSTANDARDDATA_OBJID,
                        sizeof(ObjId),
                        ObjId);

gc_util_insert_parm_ref_ex(&gcParmBlk,
                           IPSET_NONSTANDARDDATA,
                           IPPARM_NONSTANDARDDATA_DATA,
                           sizeof(NSData),
                           NSData);

gc_Extension( GCTGT_GCLIB_CRN,
              crn,
              IPEXTID_SENDMSG,
              gcParmBlk,
              NULL,
              EV_ASYNC);

gc_util_delete_parm_blk(gcParmBlk);
```

4.44.3 Nonstandard Registration Message

To send a nonstandard registration message, use the **gc_Extension( )** function in asynchronous mode with an **ext_id** (extension ID) of IPEXTID_SENDMSG. The **target_type** should be GCTGT_CCLIB_NETIF and the **target_id** should be the board device handle, since the message destination is the Gatekeeper. The GC_PARM_BLK must contain parameter elements that identify H.323 protocol, the message type, the nonstandard data, and the nonstandard data identifier. The application receives a GCEV_EXTENSIONCMPLT event to indicate that the message has been sent.

The following parameter element sets the protocol to be H.323:

IPSET_PROTOCOL
  IPPARM_PROTOCOL_BITMASK
  - value = IP_PROTOCOL_H323

The parameter element that identifies the message type is:

IPSET_MSG_REGISTRATION
  IPPARM_MSGTYPE
  - value = IP_MSGTYPE_REG_NONSTD

The parameter element for the Nonstandard Data data is:

IPSET_NONSTANDARDDATA
  IPPARM_NONSTANDARDDATA_DATA
  - value = Nonstandard Data string, max length = max_parm_data_size (configurable at library start-up)

The parameter element for the Nonstandard Data identifier is one (and only one) of the following:

IPSET_NONSTANDARDDATA
  IPPARM_NONSTANDARDDATA_OBJID
  - value = array of unsigned integers, max length = MAX_NS_PARM_OBJID_LENGTH

IPSET_NONSTANDARDDATA
    IPPARM_H221NONSTANDARD
- value = IP_H221NONSTANDARD structure

The following code snippet illustrates how to send an H.323 nonstandard registration message.

```
{
   GC_PARM_BLKP parmblkp = NULL;
   char h221nonstd_id[] = "My H.221 Nonstandard data identifier";
                                   /* must be <= MAX_NS_PARM_OBJID_LENGTH (40) */
   char nonstd_data[] = "My nonstandard_data";

   gc_util_insert_parm_val(&parmblkp, IPSET_PROTOCOL, IPPARM_PROTOCOL_BITMASK,
                        sizeof(char), IP_PROTOCOL_H323);
   gc_util_insert_parm_val(&parmblkp, IPSET_MSG_REGISTRATION, IPPARM_MSGTYPE,
                        sizeof(unsigned long), IP_MSGTYPE_REG_NONSTD);
   gc_util_insert_parm_ref_ex(&parmblkp, IPSET_NONSTANDARDDATA, IPPARM_NONSTANDARDDATA_DATA,
                           sizeof(nonstd_data), nonstd_data);
   gc_util_insert_parm_ref(&parmblkp, IPSET_NONSTANDARDDATA, IPPARM_H221NONSTANDARD,
                        sizeof(h221nonstd_id), h221nonstd_id);

   if (gc_Extension(GCTGT_CCLIB_NETIF, bdev, IPEXTID_SENDMSG, parmblkp, NULL,
                 EV_ASYNC) != GC_SUCCESS)
   {
      printandlog(ALL_DEVICES, GC_APIERR, NULL, "gc_Extension() Failed", 0);
      exitdemo(1);
   }
}
```

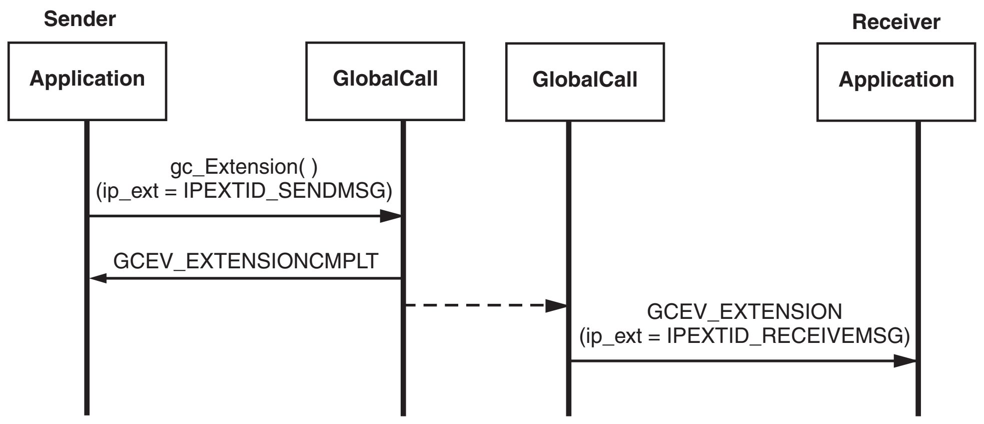
See Section 9.2.16, “IPSET_MSG_REGISTRATION”, on page 628 and Section 9.2.19, “IPSET_NONSTANDARDDATA”, on page 631 for more information.

4.44.4 Sending Facility, UII, or Registration Message Scenario

The **gc_Extension( )** function can be used to send H.245 UII messages or Q.931 nonstandard facility messages. Figure 62 shows this scenario.

An H.245 UII message can only be sent when a call is in the connected state. A Q.931 nonstandard facility message can be sent in any call state.

Figure 62. Sending Protocol Messages



4.45 Using H.323 Annex M Tunneled Signaling Messages

The Dialogic® Global Call API IP call control library supports the tunneled signaling message capability that is documented in Annex M of the ITU-T recommendations for H.323. This capability allows DSS/QSIG/ISUP messages to be encapsulated in common H.225 call signaling messages. Note that this tunneled message capability is separate and distinct from H.245 tunnelling.

The tunneled signaling message capabilities are described in the following topics:

- Tunneled Signaling Message Overview
- Enabling Tunneled Signaling Messages
- Composing Tunneled Signaling Messages
- Sending Tunneled Signaling Messages
- Receiving Tunneled Signaling Messages

4.45.1 Tunneled Signaling Message Overview

The ITU-T H.323 Annex M recommendation specifies that tunneled signaling message fields may be contained in any of nine different H.225 messages: Setup, Call Proceeding, Alerting, Connect, Release Complete, Facility, Progress, Information, and Notify. The Global Call implementation of tunneled signaling messages allows applications to send and receive tunneled messages in the first six of the listed H.225 messages. The Dialogic® Global Call API library does not support application access to the last three messages in the list of messages specified in Annex M (Progress, Information, and Notify) so these message types cannot be used for tunneled signaling messages.

The Dialogic® Global Call API library supports the ability to send and receive tunneled signaling messages in supported H.225 message types as an optional feature that is disabled by default for backwards compatibility. The ability to send and receive TSMs can only be enabled when starting the system; once enabled, the tunneled signaling message feature cannot be disabled without restarting the system. The feature can be enabled for any virtual board by setting a specific bitmask value in a field of the appropriate IP_VIRTBOARD data structure; see Section 4.45.2, "Enabling Tunneled Signaling Messages", for details. When the feature is enabled, applications can use the standard Global Call parameter mechanism to set up a TSM to be sent in the next outgoing H.225 message. To receive a TSM, the application requests the Dialogic® Global Call API library to forward the TSM after it has received a Global Call state change event that is associated with one of the supported message types. For most H.225 message types, the application uses **gc_Extension( )** to request the TSM contents which the library returns via a GCEV_EXTENSIONCMPLT asynchronous completion event. In the singular case of the Facility message, the tunneled signaling message content is provided via the unsolicited GCEV_EXTENSION event that notifies the application of the Facility message itself.

An application has no ability to specify which H.225 message types it wishes to receive tunneled signaling messages in, and should therefore be prepared to handle TSMs contained in any of the specified H.225 message types so that TSMs are not lost.

Applications construct a tunneled signaling message by constructing a GC_PARM_BLK composed of Global Call parameter elements that contain the TSM protocol identification and message content. The TSM protocol identification can use either a protocol object ID, specified in an IP_TUNNELPROTOCOL_OBJECTID data structure, or alternate identification data, specified in an IP_TUNNELPROTOCOL_ALTID structure. Only one TSM can be sent per H.225 message.

A tunneled signaling message can also include nonstandard data. The nonstandard data is handled as additional parameter elements in the same GC_PARM_BLK that contains the TSM. As in other Global Call implementations of nonstandard data, the protocol used for the nonstandard data in a TSM can be identified by either H.221 protocol or a protocol object ID. Only one nonstandard data element can be sent per tunneled signaling message.

The maximum data length for the Global Call parameters used for the tunneled signaling message content and the optional nonstandard data content is configured at system start-up. The maximum data length for these parameters is configured by setting the max_parm_data_size field in the IPCCLIB_START_DATA structure. The default size is 255 bytes (for backwards compatibility), but applications may configure it to be as large as 4096 bytes. Applications *must* use the extended **gc_util_..._ex( )** functions to insert or extract any GC_PARM_BLK parameter elements whose data length has been configured to be greater than 255 bytes.

Note: In practice, applications may not be able to utilize the full maximum length of the tunneled signaling message content parameter element as configured in max_parm_data_size, particularly if the tunneled signaling message contains optional nonstandard data. The H.323 stack limits the overall size of messages to be max_parm_data_size + 512 bytes, and any messages that exceed this limit are truncated without any notification to the application.

For all supported H.225 message types except Setup, the application presets the TSM contents to send by passing the configured GC_PARM_BLK in a call to the **gc_SetUserInfo( )** function. When the application subsequently calls one of the Global Call functions listed in Table 18, "H.225 Messages and Global Call Functions for Sending Tunneled Signaling Messages", on page 416, the library and stack use the preset data to construct and send a tunneled signaling message in the corresponding H.225 message. The duration parameter in the **gc_SetUserInfo( )** function must always be GC_SINGLECALL; TSM content cannot persist for more than one H.225 message.

The **gc_SetUserInfo( )** mechanism cannot be used to preset a tunneled signaling message to be sent in a Setup message because the function call requires a valid CRN, which does not yet exist at that point in the call setup process. When sending a TSM in Setup, the application must include the configured GC_PARM_BLK in the GC_MAKECALL_BLK data structure that is passes to the **gc_MakeCall( )** function.

Table 18, "H.225 Messages and Global Call Functions for Sending Tunneled Signaling Messages", on page 416 lists the H.225 message types that can be used to send tunneled signaling messages along with the corresponding Global Call mechanism that is used set the TSM information and the Global Call function that is used to send each message type.

When reception of tunneled signaling messages is enabled as described in the "Enabling Tunneled Signaling Messages" section, applications must specifically request the message by calling the **gc_Extension( )** function and providing a tag value that identifies the specific type of H.225 message that was received. When the corresponding H.225 message contains a tunneled signaling message, the library generates an asynchronous GCEV_EXTENSIONCMPLT completion event which includes the tunneled signaling message information in the metaevent data. Tunneled signaling messages can only be retrieved within a call (the application must use a valid CRN when registering to receive tunneled signaling messages), but the call can be in any state.

4.45.2 Enabling Tunneled Signaling Messages

The ability to send tunneled signaling messages in outgoing H.225 messages and to retrieve TSM content from inbound H.225 messages is an optional feature that is enabled or disabled on a virtual board basis at the time the **gc_Start( )** function is called.

The mandatory **INIT_IP_VIRTBOARD( )** function populates the IP_VIRTBOARD structure with default values. The default value of the h323_msginfo_mask field in the IP_VIRTBOARD structure does not enable either access to either Q.931 message information elements or to tunneled signaling messages. To enable either or both of these features for an ipt device, it is necessary to override the default value of the h323_msginfo_mask field with a value that represents the appropriate logical combination of the two defined mask values. To enable access to tunneled signaling messages in general, the value IP_H323_ANNEXMMSG_ENABLE must be set in the mask. The following code snippet enables Q.931 message IE access on two virtual boards and enables tunneled signaling messages on the second board only:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE;
                                    /* override Q.931 message default */
ip_virtboard[1].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE |  IP_H323_ANNEXMMSG_ENABLE;
                                    /* override Q.931 message and TSM defaults */
```

Note: Features that are enabled or configured via the IP_VIRTBOARD structure cannot be disabled or reconfigured once the library has been started. All items set in this data structure take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

4.45.3 Composing Tunneled Signaling Messages

The process of sending a tunneled signaling message begins by composing a GC_PARM_BLK that contains Global Call parameter elements for the message protocol, the message content, and any nonstandard data.

The first parameter element identifies the message protocol. It must be **one** of the following two forms:

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_PROTOCOL_OBJECTID
- value = protocol object ID information in an IP_TUNNELPROTOCOL_OBJECTID data structure and conforming to the appropriate ASN.1 format

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_ALTERNATEID
- value = alternate protocol ID information in an IP_TUNNELPROTOCOL_ALTID data structure

The second parameter element contains the actual message content. Applications should use the **gc_util_insert_parm_ref_ex( )** function to insert this parameter element because the parameter data may exceed 255 bytes.

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_CONTENT
- value = actual message content
  max. length = max_parm_data_size (configured at library start-up)

Note: In practice, applications may not be able to utilize the full maximum length of the TSM content parameter element as configured in max_parm_data_size, particularly if the TSM also contains non-standard data. The H.323 stack limits the overall size of messages to be max_parm_data_size + 512 bytes, and any messages that exceed this limit are truncated without any notification to the application.

If the tunneled signal message includes optional nonstandard data, the GC_PARM_BLK needs to contain two additional parameter elements. These parameters should only be inserted in the GC_PARM_BLK if nonstandard data is being sent in the message. The first parameter element for nonstandard data is:

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_NSDATA_DATA
- value = actual nonstandard data, max. length = max_parm_data_size (configured at library start-up)

Applications should use the **gc_util_insert_parm_ref_ex( )** function to insert this parameter element because the parameter data may exceed 255 bytes.

Note: In practice, applications may not be able to utilize full maximum parameter length configured in max_parm_data_size for nonstandard data content. The H.323 stack limits the overall size of messages to be max_parm_data_size + 512 bytes, which must contain the tunneled signaling message content as well as the nonstandard data.

The second parameter element for nonstandard data uses **one** of the following two forms:

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_NSDATA_OBJID
  - value = nonstandard data object ID string conforming to appropriate ASN.1 format

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_NSDATA_H221NS
  - value = H.221 nonstandard data protocol information in an IP_H221NONSTANDARD data structure

The following code example illustrates the process of composing the parameter block for a tunneled signaling message.

```
GC_PARM_BLKP               gcParmBlk = NULL;
IP_TUNNELPROTOCOL_ALTID    tsmTpAltId;
IP_TUNNELPROTOCOL_OBJECTID tsmTpObjId;

char *pMsgContent       = "00 11 22 33 44 55 44 33 33 66 66 55 77 22 11 11";
int asize = strlen(pMsgContent);
char *pTP_Oid           = "0 0 17 931";
                        // Note that the Object Id string must be in the correct ASN.1 format.
char TP_AltID_Type[]    = "Tunneled Protocol Alternate ID protocol type";
char TP_AltID_Variant[] = "Tunneled Protocol Alternate ID protocol variant";
char TP_AltID_SubId[]   = "Tunneled Protocol Alternate ID subidentifier - User";
char *ptsmNSData_Data   = "Tunneled Signaling Message Non Standard Data";
char *pTP_ObjID_Oid     = "0 0 17 931";
                        // Note that the Object Id string must be in the correct ASN.1 format.
char TP_ObjID_SubId[]   = "Tunneled Protocol Object ID subidentifier - User";
int bsize = strlen(TP_ObjID_SubId);

IP_H221NONSTANDARD tsmH221NS;
tsmH221NS.country_code = 91;
tsmH221NS.extension = 202;
tsmH221NS.manufacturer_code = 11;

INIT_IP_TUNNELPROTOCOL_ALTID(&tsmTpAltId);
strcpy(tsmTpAltId.protocolType, TP_AltID_Type);
tsmTpAltId.protocolTypeLength = strlen(TP_AltID_Type) + 1;
strcpy(tsmTpAltId.protocolVariant, TP_AltID_Variant);
tsmTpAltId.protocolVariantLength = strlen(TP_AltID_Variant) + 1;
strcpy(tsmTpAltId.subIdentifier, TP_AltID_SubId);
tsmTpAltId.subIdentifierLength = strlen(TP_AltID_SubId) + 1;

INIT_IP_TUNNELPROTOCOL_OBJECTID(&tsmTpObjId);
strcpy(tsmTpObjId.TunneledProtocol_Oid, pTP_ObjID_Oid);
tsmTpObjId.TunneledProtocol_OidLength = strlen(pTP_ObjID_Oid) + 1;
strcpy(tsmTpObjId.subIdentifier, TP_ObjID_SubId);
tsmTpObjId.subIdentifierLength = strlen(TP_ObjID_SubId) + 1;

choiceOfTSMProtocol = 1;
       /* App decides whether to use the tunneled signaling message Protocol Object ID/ AltID */
choiceOfNSData = 1;
       /* App decides which type of object identifier to use for TSM NS Data */

if (choiceOfTSMProtocol)
   /* App decides the choice of the tunneled signaling msg protocol object identifier */
   /* It cannot set both objid & alternate id */
{
   gc_util_insert_parm_ref(&gcParmBlk,
                           IPSET_TUNNELEDSIGNALMSG,
                           IPPARM_TUNNELEDSIGNALMSG_ALTERNATEID,
                           (unsigned char)sizeof(IP_TUNNELPROTOCOL_ALTID),
                           &tsmTpAltId);
```

```
}
else
{
   gc_util_insert_parm_ref(&gcParmBlk,
                           IPSET_TUNNELEDSIGNALMSG,
                           IPPARM_TUNNELEDSIGNALMSG_PROTOCOL_OBJECTID,
                           (unsigned char)sizeof(IP_TUNNELPROTOCOL_OBJECTID),
                           &tsmTpObjId);

/* Note the use of the extended gc_util function because TSM data may exceed 255 bytes */
gc_util_insert_parm_ref_ex(&gcParmBlk,
                           IPSET_TUNNELEDSIGNALMSG,
                           IPPARM_TUNNELEDSIGNALMSG_CONTENT,
                           (unsigned char)(strlen(pMsgContent)+1),
                           pMsgContent);

/* Now fill in the Tunneled Signaling message Non Standard data fields, if used */
/* Note the use of the extended gc_util function because NSD data may exceed 255 bytes */
gc_util_insert_parm_ref_ex(&gcParmBlk,
                           IPSET_TUNNELEDSIGNALMSG,
                           IPPARM_TUNNELEDSIGNALMSG_NSDATA_DATA,
                           (unsigned char)(strlen(ptsmNSData_Data)+1),
                           ptsmNSData_Data);

if (choiceOfNSData)
   /* App decides the CHOICE of Non Standard OBJECTIDENTIFIER. */
   /* It cannot set both objid & H221 */
{
   // Set the NS Object ID
   gc_util_insert_parm_ref(&gcParmBlk,
                           IPSET_TUNNELEDSIGNALMSG,
                           IPPARM_TUNNELEDSIGNALMSG_NSDATA_OBJID,
                           (unsigned char) (strlen(ptsmNSData_Oid)+1),
                           ptsmNSData_Oid
}
else
{
   // Set the H221
   gc_util_insert_parm_ref(&gcParmBlk,
                           IPSET_TUNNELEDSIGNALMSG,
                           IPPARM_TUNNELEDSIGNALMSG_NSDATA_H221NS,
                           (unsigned char)sizeof(IP_H221NONSTANDARD),
                           &tsmH221NS);
```

4.45.4 Sending Tunneled Signaling Messages

Once the GC_PARM_BLK containing the TSM information has been composed by the application, the application must pass the parameter block to the call control library to be transformed into a tunneled message that can be inserted into an H.225 message. The mechanism used to hand the TSM information to the library varies depending on what Global Call function and corresponding H.225 message will be used to send the TSM.

Table 18 lists the H.225 message types that can be used to send tunneled signaling messages along with the corresponding Global Call mechanism that is used set the TSM information and the Global Call function that is used to send each message type.

Table 18. H.225 Messages and Global Call Functions for Sending Tunneled Signaling Messages

H.225 message to be used to send TSM	Mechanism used to set TSM to send	Global Call Function used to send H.225 message containing TSM
Setup	GC_MAKECALL_BLK	**gc_MakeCall( )**
Proceeding	**gc_SetUserInfo( )** (GC_SINGLECALL)	**gc_CallAck( )**
Alerting	**gc_SetUserInfo( )** (GC_SINGLECALL)	**gc_AcceptCall( )**
Connected	**gc_SetUserInfo( )** (GC_SINGLECALL)	**gc_AnswerCall( )**
Release Complete	**gc_SetUserInfo( )** (GC_SINGLECALL)	**gc_DropCall( )**
Facility	**gc_SetUserInfo( )** (GC_SINGLECALL)	**gc_Extension( )** (IPEXTID_SENDMSG, IPSET_MSG_Q931, IPPARM_MSGTYPE, IP_MSGTYPE_Q931_FACILITY)

Sending a TSM in a Setup Message

Once the GC_PARM_BLK is composed, the block is included in a GC_MAKECALL_BLK structure via the intermediate GCLIB_MAKECALL_BLK structure, and that block is then passed as a parameter in a call to **gc_MakeCall( )**. The Setup message that is sent as a result of the call to **gc_MakeCall( )** will contain a TSM with elements as specified in the GC_PARM_BLK.

The **gc_SetUserInfo( )** function *cannot* be used to preset TSM information to be sent in a Setup message because that function requires a valid CRN when setting a tunneled signaling message and the CRN does not exist at this point in the call setup. The TSM can only be specified in the GC_MAKECALL_BLK for a Setup message.

Sending a TSM in an Alerting, Connected, Facility, Proceeding, or ReleaseComplete Message

To include a tunneled signaling message in any H.225 message other than a Setup message, the application uses the **gc_SetUserInfo( )** to preset the message data before calling the Global Call function that causes the H.225 message to be sent. Data set via **gc_SetUserInfo( )** applies to the next outgoing message, so applications should be careful to call this function immediately before the function that will send the intended H.225 message.

When calling **gc_SetUserInfo( )**, the parameters should be set as follows:

- **target_type** must be set to GCTGT_GCLIB_CRN
- **target_id** is the CRN
- **infoparmblkp** is a pointer to the GC_PARM_BLK that was configured with the parameter elements for the tunneled signaling message
- **duration** must be set to GC_SINGLECALL

Four of the supported H.225 message types are sent as a direct result of a specific Global Call function call for the CRN and require no other preparation after the **gc_SetUserInfo( )**:

- **gc_AcceptCall( )** sends Alerting
- **gc_AnswerCall( )** sends Connected
- **gc_CallAck( )** sends Proceeding
- **gc_DropCall( )** sends Release Complete

But because there is no call state change associated with the Facility message, there is no dedicated Global Call function to send this message (nor is there a dedicated Global Call event to receive a Facility message). Instead, Global Call uses the generic **gc_Extension( )** mechanism to send and receive Facility messages. Because of this, an application must construct a GC_PARM_BLK to pass to the **gc_Extension( )** function call to specify that it wishes to send a Facility message; note that this GC_PARM_BLK is completely separate from the structure that sets up the TSM itself via the **gc_SetUserInfo( )** function. The GC_PARM_BLK passed to **gc_Extension( )** must contain a parameter element of the following type:

IPSET_MSG_Q931
  IPPARM_MSGTYPE
  - value = IP_MSGTYPE_Q931_FACILITY

The parameters for the **gc_Extension( )** function call should be set as follows:

- **target_type** must be set to GCTGT_GCLIB_CRN
- **target_id** is the CRN
- **ext_id** must be set to IPEXTID_SENDMSG
- **parmblkp** is a pointer to the GC_PARM_BLK that was configured with the parameter element for the Q.931 Facility message
- **retblkp** is NULL
- **mode** must be set to EV_ASYNC

4.45.5 Receiving Tunneled Signaling Messages

Assuming that the TSM feature was enabled when the virtual board was started, an application can request the tunneled signaling message content whenever it receives a Global Call event that corresponds to one of the supported H.225 message types. For all supported message types except Facility, the application uses the **gc_Extension( )** function and the extension ID IPEXTID_GETINFO to request the TSM, and the TSM contents are transmitted in the external data associated with the asynchronous GCEV_EXTENSIONCMPLT completion event for the function call. In the case of the Facility message, Global Call notifies the application that it has received the message via an unsolicited GCEV_EXTENSION event, and this event itself conveys the TSM in its external data.

Table 19 relates the message types of the supported H.225 messages that can contain TSM fields to the Global Call event types that are used to notify the application of the message's arrival and the tag that is used by the application when retrieving the TSM content.

Table 19. H.225 Messages and Global Call Events for Receiving Tunneled Signaling Messages

H.225 message	Global Call event used to notify application	Tag used to retrieve message fields via GCEV_EXTENSIONCMPLT
Setup	GCEV_OFFERED	TSM_CONTENT_OFFERED
Proceeding	GCEV_PROCEEDING †	TSM_CONTENT_PROCEEDING
Alerting	GCEV_ALERTING	TSM_CONTENT_ALERTING
Connected	GCEV_CONNECTED	TSM_CONTENT_CONNECTED
Release Complete	GCEV_DISCONNECTED	TSM_CONTENT_DISCONNECTED
Facility	GCEV_EXTENSION	TSM_CONTENT_EXTENSION
† The GCEV_PROCEEDING event is maskable. When Tunneled Signalling Messages are enabled, the application must ensure that this event is not masked.		

Retrieving TSMs from Alerting, Connected, Proceeding, ReleaseComplete, and Setup Messages

To retrieve TSM after receiving a Global Call state change event corresponding to an Alerting, Connected, Proceeding, ReleaseComplete, or Setup message, the application first constructs a GC_PARM_BLK that specifies the type of information it wishes to retrieve, then calls the **gc_Extension( )** function to request the information.

The GC_PARM_BLK must contain the following two parameter elements:

IPSET_TUNNELEDSIGNALMSG
    IPPARM_TUNNELEDSIGNALMSG_CONTENT
    - value is unused; set to 1

IPSET_TUNNELEDSIGNALMSG
    IPPARM_TSM_CONTENT_EVENT
    - value = appropriate TSM_CONTENT_... tag value as listed in Table 19

When the application calls **gc_Extension( )**, the parameters should be set up as follows:

- **target_type** must be GCTGT_GCLIB_CRN
- **target_id** must be a valid CRN
- **ext_id** must be IPEXTID_GETINFO
- **parmblkp** must point to the GC_PARM_BLK that was configured to contain the required parameter element as just described.
- **retblkp** must be a valid pointer to a GC_PARM_BLK
- **mode** must be EV_ASYNC

If the received H.225 message contained a tunneled signaling message, the library generates an asynchronous GCEV_EXTENSIONCMPLT completion event. The extevtdatap field in the METAEVENT structure for this event is a pointer to an EXTENSIONEVTBLK structure, which in turn contains a GC_PARM_BLK that contains the fields of the received tunneled signaling message. Applications are then able to extract the data of interest using the **gc_util_..._ex( )** functions.

The GC_PARM_BLK will always contain the following three parameter elements:

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TSM_CONTENT_EVENT
  - value = the appropriate TSM_CONTENT_... tag value

one or the other of the following two elements:

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_PROTOCOL_OBJECTID
  - value = IP_TUNNELPROTOCOL_OBJECTID data structure

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_ALTERNATEID
  - value = IP_TUNNELPROTOCOL_ALTID data structure

and the following element:

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_CONTENT
  - value = tunneled signaling message content string

If the TSM includes optional nonstandard data, there will be two additional parameter elements:

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_NSDATA_DATA
  - value = nonstandard data string

and one of the following two elements:

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_NSDATA_OBJID
  - value = nonstandard data object ID string in ASN.1 format

IPSET_TUNNELEDSIGNALMSG
  IPPARM_TUNNELEDSIGNALMSG_NSDATA_H221NS
  - value = IP_H221NONSTANDARD data structure

Notes: ***1.*** The application must take care to retrieve the Annex M Message information from any incoming H.225 message before the next H.225 message arrives. If the new message also contains TSM information, that new TSM overwrites the prior information.

2. The overall message size that the Global Call H.323 stack can handle is defined as max_parm_data_size (which is configured at library startup) + 512 bytes. Any message that is received which exceeds this length is truncated.

3. Parameter values that are contained in a GC_PARM_BLK are subject to maximum length limits that are defined for each parameter type. Any data received in a TSM that exceeds these defined limits is truncated without notification to the application.

4. The application should use the extended **gc_util_..._ex( )** functions when extracting parameters from a GC_PARM_BLK that contains TSM contents because some of the Global Call parameters for TSMs support data length that may exceed 255 bytes.

TSM Retrieval Code Example

The following code example shows how an application might handle the process of requesting tunneled signaling message after it has received a Global Call event associated with one of the supported H.225 message types.

```
GC_PARM_BLKP gcParmBlk = NULL;
GC_PARM_BLKP retParmBlk;
GC_PARM_DATA_EXT parm_data_ext;
INIT_GC_PARM_DATA_EXT(&parm_data_ext);
int frc;

switch(event)
{
   case GCEV_ALERTING:
      frc = gc_util_insert_parm_val(&gcParmBlk,
                                    IPSET_TUNNELEDSIGNALMSG,
                                    IPPARM_TUNNELEDSIGNALMSG_CONTENT,
                                    sizeof(int),
                                    1);
      frc = gc_util_insert_parm_val(&gcParmBlk,
                                    IPSET_TUNNELEDSIGNALMSG,
                                    IPPARM_TSM_CONTENT_EVENT,
                                    sizeof(int),
                                    TSM_CONTENT_ALERTING);
      frc = gc_Extension(GCTGT_GCLIB_CRN,
                         crn,
                         IPEXTID_GETINFO,
                         gcParmBlk,
                         &retParmBlk,
                         EV_ASYNC);
      break;

   case GCEV_CONNECTED:
      frc = gc_util_insert_parm_val(&gcParmBlk,
                                    IPSET_TUNNELEDSIGNALMSG,
                                    IPPARM_TUNNELEDSIGNALMSG_CONTENT,
                                    sizeof(int),
                                    1);
      frc = gc_util_insert_parm_val(&gcParmBlk,
                                    IPSET_TUNNELEDSIGNALMSG,
                                    IPPARM_TSM_CONTENT_EVENT,
                                    sizeof(int),
                                    TSM_CONTENT_CONNECTED);
      frc = gc_Extension(GCTGT_GCLIB_CRN,
                         crn,
                         IPEXTID_GETINFO,
                         gcParmBlk,
                         &retParmBlk,
                         EV_ASYNC);
      break;

   ...
//Similar cases for other event types of interest
   ...

   case GCEV_EXTNCMPLT:
      GC_PARM_DATA *parmp = NULL;
      while (GC_SUCCESS == (gc_util_next_parm_ex(parm_blk, &parm_data_ext)))
      {
         switch (parmp->set_ID)
         {
            case IPSET_TUNNELEDSIGNALMSG:
            switch (parm_data_ext.parm_ID)
            {
               case IPPARM_TSM_CONTENT_EVENT:
                  printf("\tReceived TSM in message type: %s\n",
                          parm_data_ext.value_buf);
                  break;

               case IPPARM_TUNNELEDSIGNALMSG_CONTENT:
                  printf("\tReceived extension data (TSM) Msg Content: %s\n",
                          parm_data_ext.value_buf);
                  break;

               case IPPARM_TUNNELEDSIGNALMSG_PROTOCOLOBJID:
                  printf("\tReceived extension data (TSM) PROTOCOL_OBJID:
                         %s\n", parm_data_ext.value_buf);
                  break;

               case IPPARM_TUNNELEDSIGNALMSG_ALTERNATEID:
                  printf("\tReceived extension data (TSM) TUNNELPROTOCOL_ALTID:
                         %s\n", parm_data_ext.value_buf);
                  break;

               // Additional cases for optional NSD
               ...

            }

       }

  }
```

4.46 Retrieving User-to-User Information Elements from H.323 Messages

Various ITU-T recommendations specify User-to-User Information Elements (UU-IE), which may be contained in a number of different call control messages. The Global Call H.323 call control library allows applications to receive UU-IE in six types of H.225 messages: Setup, Call Proceeding, Alerting, Connect, Release Complete, and Facility. The library does not provide facilities to set and send a UU-IE in an outgoing call control message. UU-IE is retrieved and passed to the application in raw (ASN.1) format; it is the application's responsibility to convert the information from ASN.1 format as appropriate.

The ability to receive UU-IE from incoming H.225 messages is implemented as an optional feature that is disabled by default for backwards compatibility. The ability to receive UU-IE can only be enabled when starting the system; once enabled, the feature cannot be disabled without restarting the system. The feature can be enabled for any virtual board by setting a specific bitmask value in a field of the appropriate IP_VIRTBOARD data structure.

When the UU-IE retrieval feature is enabled, Global Call uses the metaevent mechanism to forward the UU-IE to an application along with the state change event associated with the supported H.225 message type (or an Extension event in the case of a Facility message). The UU-IE content is handled as a Global Call parameter element in the extension data that is associated with the metaevent.

An application has no ability to specify which H.225 message types it wishes to receive UU-IE in, and should therefore be capable of handling UU-IE contained in any of the specified H.225 message types.

The maximum data length for the Global Call parameter used for the UU-IE content is configured at system start-up. The maximum data length for this parameter is configured by setting the max_parm_data_size field in the IPCCLIB_START_DATA structure. The default size is 255 bytes (for backwards compatibility), but applications may configure it to be as large as 4096 bytes. Applications *must* use the extended **gc_util_..._ex( )** functions to extract any GC_PARM_BLK parameter elements whose data length has been configured to be greater than 255 bytes.

4.46.1 Enabling Reception of User-to-User Information

The ability to retrieve UU-IE from inbound H.225 messages is an optional feature that is enabled or disabled on a virtual board basis at the time the **gc_Start( )** function is called.

The mandatory **INIT_IP_VIRTBOARD( )** function populates the IP_VIRTBOARD structure with default values. The default value of the h323_msginfo_mask field in the IP_VIRTBOARD structure does not enable access to any of the supported additional information types (Q.931 message information elements, tunneled signaling messages, or User-to-User information elements). To enable any of these features for an ipt device, the default value of the h323_msginfo_mask field must be overridden with a value that represents the appropriate logical combination of the defined mask values. To enable access to User-to-User IEs, the value IP_H323_RETRIEVE_UUIE_ENABLE must be set in the mask. The following code snippet enables Q.931 message IE access on two virtual boards and enables UU-IE access on the second board only:

```
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE;
                                    /* override Q.931 message default */
ip_virtboard[1].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE |  IP_H323_RETRIEVE_UUIE_ENABLE;
                                    /* override Q.931 message and UU-IE defaults */
```

Note: Features that are enabled or configured via the IP_VIRTBOARD structure cannot be disabled or reconfigured once the library has been started. All items set in this data structure take effect when the **gc_Start( )** function is called and remain in effect until **gc_Stop( )** is called when the application exits.

Table 19 relates the supported H.225 messages that can contain UU-IE fields to the Global Call event types that are used to notify the application of the message's arrival.

Table 20. H.225 Messages and Global Call Events for Receiving UU-IE

H.225 message	Global Call event used to notify application
Setup	GCEV_OFFERED
Proceeding	GCEV_PROCEEDING †
Alerting	GCEV_ALERTING
† The GCEV_PROCEEDING event is maskable.	

Table 20. H.225 Messages and Global Call Events for Receiving UU-IE

H.225 message	Global Call event used to notify application
Connected	GCEV_CONNECTED
Release Complete	GCEV_DISCONNECTED
Facility	GCEV_EXTENSION (IPEXTID_RECEIVEMSG)
† The GCEV_PROCEEDING event is maskable.	

4.46.2 Retrieving UU-IEs

Once the UU-IE access feature is enabled, any User-to-User Information Element received in a supported message type is made available to the application as a parameter element in a GC_PARM_BLK that is associated with metaevent for one of the Global Call event types listed in Table 19. The metaevent is retrieved using the standard **gc_GetMetaEvent( )** method and the parameter element is retrieved from the parameter block using standard Global Call extended **gc_util_..._ex( )** functions.

Note: The application must take care to retrieve the UU-IE information from any incoming message or copy it to a new location before the next message arrives. The next call to **gc_GetMetaEvent( )** will wipe out the metaevent data from the prior event.

In all cases, User-to-User Information Elements are handled as Global Call parameter elements of the following type:

IPSET_CALLINFO
- IPPARM_UUIE_ASN1
  - value = octet string containing raw, ASN.1-encoded user-to-user information element

It is the application's responsibility to decode the ASN.1 data and retrieve any information that is of interest.

Note: The maximum length for the value of the UU-IE parameter element is max_parm_data_size, which is configured at library startup (the default value is 255). Any received UU-IE data that exceeds this maximum length is truncated without notification to the application. When max_parm_data_size is set to a value larger than 255, the application *must* use the extended **gc_util_..._ex( )** functions when extracting parameters from a GC_PARM_BLK.

Retrieving UU-IE from Alerting, Connected, Proceeding, ReleaseComplete, and Setup Messages

To retrieve UU-IE from the Global Call event that notifies the application of an Alerting, Connected, Proceeding, ReleaseComplete, or Setup message, the application uses **gc_GetMetaEvent( )** to retrieve the metaevent. If the received H.225 message contained a UU-IE, the GC_PARM_BLK structure pointed to by the extevtdatap field in the METAEVENT structure will contain a parameter element that has the UU-IE data as its value. The application is then able to use the **gc_util_..._ex( )** functions to extract the ASN.1-encoded data for processing.

Retrieving UU-IE from Facility Messages

Because there is no Global Call state change event associated with a Facility message, a slightly different retrieval mechanism applies to this message type. In the case of a Facility message, the UU-IE is sent to the application in an unsolicited GCEV_EXTENSION event which has an extension ID of IPEXTID_RECEIVEMSG rather than a Global Call state change event. The extevtdatap field of the metaevent for this event is a pointer to an EXTENSIONEVTBLK structure which contains a GC_PARM_BLK structure. This parameter block in turn contains a parameter element that has the ASN.1-encoded UU-IE data as its value.

UU-IE Retrieval Code Example

The following example illustrates retrieval of a UU-IE element received in an H.323 message:

```
int OnEventRetrieveUUIE(METAEVENT metaevent)
{
   // This function does the following:
   // 1) See if the event can have a UUIE data; if not then return
   // 2) Extract the GC parm block associated with the event
   // 3) Go through the GC parm block to see if there exists a setid/parmid combination
   //    of IPSET_CALLINFO/IPPARM_UUIE_ASN1
   // 4) For the GC parm data, that has the above combination, print the UUIE length and
   //    UUIE data bytes.

   int rc;
   GC_PARM_BLKP parm_blk
   GC_PARM_DATA_EXT parm;
   char *pChar;
   int evttype;
   EXTENSIONEVTBLK * pextensionBlk;
   Int i;

   evttype = metaevent.evttype;

   switch(evttype)
   {

      case GCEV_OFFERED:
      case GCEV_PROCEEDING:
      case GCEV_ALERTING:
      case GCEV_CONNECTED:
      case GCEV_DISCONNECTED:
         // For all these events, metaevent's extension event block will contain the actual
         // GC parm block.
         parm_blk  = metaevent.extevtdatap;
         break;

      case GCEV_EXTENSION:
         // For this event, metaevent's extension event block will contain the actual
         // extension event block.
         pextensionBlk = (EXTENSIONEVTBLK *)(metaevent.extevtdatap);
         parm_blk = (&(pextensionBlk->parmblk));
         if(pextensionBlk->ext_id != IPEXTID_RECEIVEMSG)
         {
            printf("UUIE extraction is possible only for ext id of IPEXTID_RECEIVEMSG.
                    Not for ext id of %d\n", pextensionBlk->ext_id);
            return(-1);
         }
         break;
      default:
         // Since UUIE data can be present only for the above mentioned events,
         // return from the fuction for all other events.
         printf("UUIE retrieval is not supported for this type of event.\n");
         return(-1);
         break;
   }

   // Initialize the GC parm data.
   INIT_GC_PARM_DATA_EXT(&parm);

   // Now go through all the GC parm datas of the GC parm block to find out if there is a
   // setid/parmid combination of IPSET_CALLINFO/IPPARM_UUIE_ASN1. Parm block with
   // this particular combination contains the ASN1 encoded UUIE.
   rc = gc_util_next_parm_ex(parm_blk,&parm);
   if (rc != 0)
   {
      return -1;
   }

   while (rc != EGC_NO_MORE_PARMS)
   {
      switch (parm.set_ID)
      {
         case IPSET_CALLINFO:
         switch (parm.parm_ID)
         {
            case IPPARM_UUIE_ASN1:
            // This GC parm data contains the ASN1 encoded UUIE.
            pChar = (unsigned char *)parm.pData;
            printf("The ASN1 encoded UU-IE data of size"
                   "%d is at %x.\n", parm.data_size, parm.pData);
            for(i=0; i< parm.data_size; i++)
            {
               printf("%d ", *(unsigned char *)(pChar+i));
            }
            // The user can pass this pChar to some ASN.1 decoder to convert it
            // to a textual format.
            break;
         }
         break;
      }

      // Get the next GC parm data.
      rc = gc_util_next_parm_ex(parm_blk,&parm);
   }

   return(0);
}
```

Third Party Call Control (3PCC) Operations and Multimedia Support 5

This chapter provides an overview of the libraries and protocols used for third party call control (3PCC) and describes how to use the Dialogic® Global Call API to perform certain third party call control operations in a SIP environment. Topics include:

5.1 Overview

This section provides an overview of third party call control, along with brief descriptions of the libraries and protocols required to support third party call control. The following topics are presented in this section:

- Third Party Call Control
- Global Call Library and IP Media Library for Third Party Call Control
- Session Description Protocol

5.1.1 Third Party Call Control

Third party call control enables one entity (for example, a third party call controller) to create, modify, or terminate a media session between two or more endpoints. Call control signaling and media exchange are separated and independently managed.

The key attributes of third party call control are:

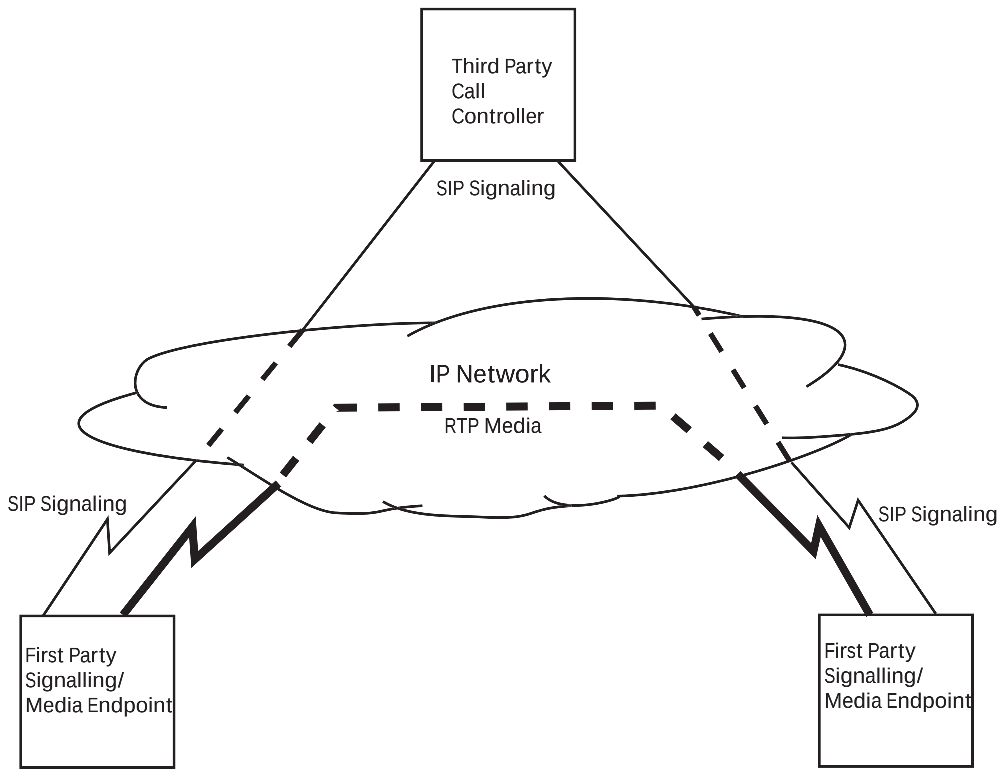
- A distinct third party call controller initiates the session.
- The third party call controller initiates communications via a SIP signaling interface to each of the endpoints involved in the session.
- The endpoints do not need to directly establish signaling interfaces between one another; instead, they have a signaling relationship with the third party call controller.
- The third party call controller does not serve as an endpoint for the media stream; the media stream flows between the two endpoints.

The Internet Engineering Task Force (IETF) has defined *RFC 3725 - Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP)*. RFC 3725 is available at *http://ietf.org/rfc/rfc3725.txt*.

Section 5.1.2, "Global Call Library and IP Media Library for Third Party Call Control", on page 431 describes how to configure the Dialogic® Global Call API library in third party call control (3PCC) mode or first party call control (1PCC) mode. Global Call behavior in 3PCC mode is similar to the behavior in 1PCC mode, except that media and session control are independently managed in 3PCC mode. This provides greater flexibility to 3PCC mode applications.

Figure 63 shows a basic third party call control connection:

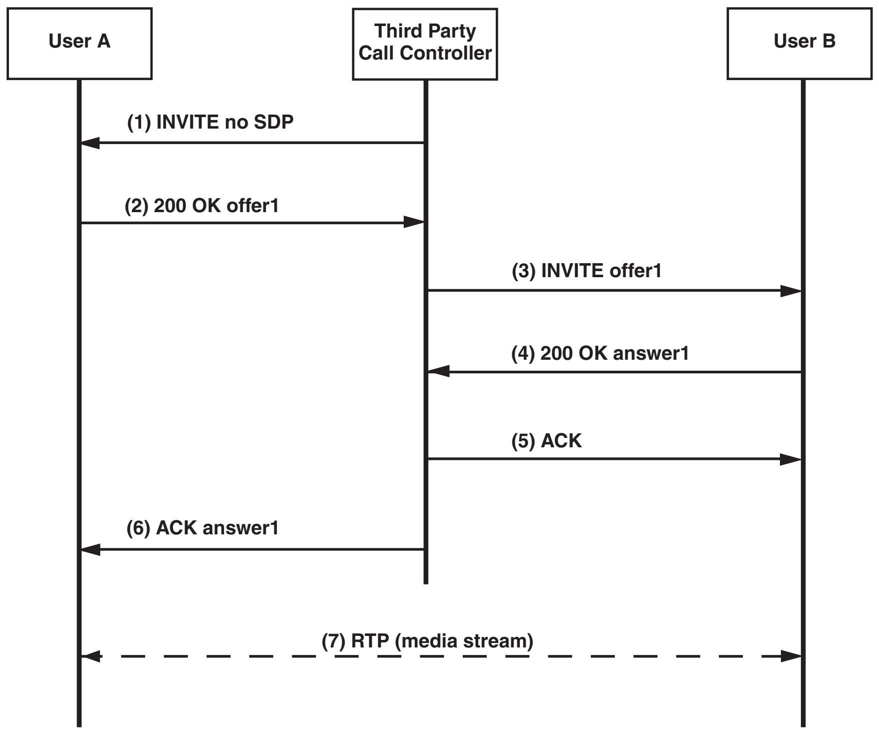
Figure 63. Third Party Call Controller


Third Party Call Controller
SIP Signaling
IP Network
RTP Media
SIP Signaling
SIP Signaling
First Party Signalling/ Media Endpoint
First Party Signalling/ Media Endpoint

SIP uses the Session Description Protocol (SDP) format for negotiating the media parameters of third party call control calls. Further information on SDP is available in the Internet Engineering Task Force (IETF) document *RFC 2327 - SDP: Session Description Protocol*. RFC 2327 is available at *http://ietf.org/rfc/rfc2327.txt*.

Figure 64 shows a basic call setup sequence for third party call control:

Figure 64. Basic Call Setup When Using Third Party Call Control



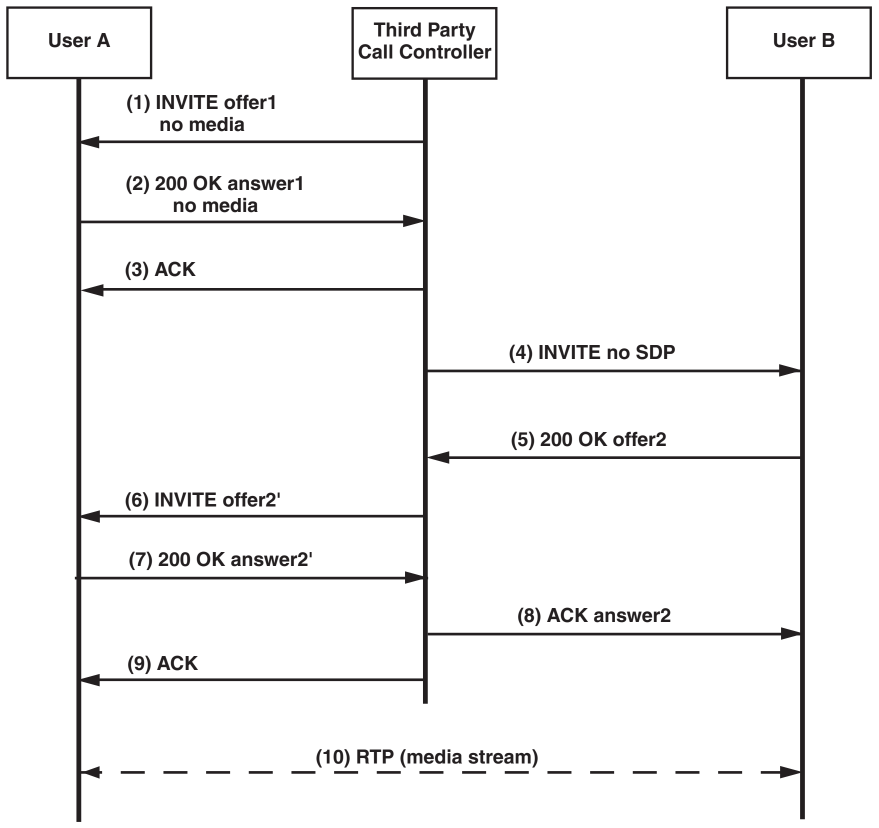
The call sequence description for Figure 64 is as follows:

1. The third party call controller sends a SIP INVITE method (1). This INVITE does not contain SDP information. The INVITE method causes User A’s phone to ring.
2. When User A’s phone rings, User A answers. A 200 OK (2) message is sent from User A to the third party call controller. The 200 OK message contains an SDP offer (offer1).
3. Per SIP rules (RFC 3261), the third party call controller must respond to User A’s 200 OK message with an ACK method. The ACK method should contain an answer to the offer (offer1) that was included in User A’s 200 OK message.
4. To obtain the answer, the third party call controller encapsulates the SDP offer (offer1) it received from User A in an INVITE method. This INVITE method is then sent to User B (3). This INVITE method causes User B’s phone to ring.
5. When User B’s phone rings, User B answers. A 200 OK (4) message is sent from User B to the third party call controller. The 200 OK message contains an SDP answer (answer1) to the offer (offer1).
6. The third party call controller sends an ACK method to User B (5).

7. The third party call controller then encapsulates the answer (answer1) in an ACK method. This ACK method is then sent to User A (6). User A has now received the ACK message that was required as part of step 3 above.
8. Because the SDP offer (offer1) was generated by User A and the SDP answer (answer1) was generated by User B, the RTP (media stream) flows between User A and User B (7).

While the call scenario described in Figure 64 is useful for explaining the fundamentals of third party call control call, it is not a realistic call scenario. Most significantly, it assumes that User B will answer the phone. If User B does not answer the phone, User A will never receive an ACK from the third party controller. This results in a time-out problem that will eventually cause the call to fail. Figure 65 describes the fundamentals of a more realistic example, using re-INVITE (a subsequent INVITE on an active dialog) to establish third party call control:

Figure 65. Third Party Call Control Setup using re-INVITE



The call sequence description for Figure 65 is as follows:

1. The third party call controller sends a SIP INVITE method (1) to User A. This INVITE contains an SDP offer (offer1), but the offer does not provide a media level description (no *m* lines in the SDP message body) for the session. This implies that the media session will eventually be coordinated via a re-INVITE at a later time. The initial INVITE method causes User A’s phone to ring.

Note: Per SDP rules (RFC 3264), if an SDP message body does not contain a media level description (at least one *m* line), the message body must contain connection information (*c* line must be present). To satisfy this requirement in IPv4 networks, a "black hole" IP address of 0.0.0.0 is provided as part of the *c* line in the SDP message body. IPv4 packets sent to this "black hole" address never leave the host that sent them; the packets are discarded. Keep in mind that this behavior is specific to IPv4 networks. IPv6 and other networks should use an IP address with similar properties.

2. When User A's phone rings, User A answers. A 200 OK (2) message is sent from User A to the third party call controller. The 200 OK message contains an SDP answer (answer1) to the third party call controller's offer (offer1). The 200 OK message does not provide a media level description, much like the INVITE method from (1).
3. The third party call controller responds to User A's 200 OK message with an ACK.
4. The third party call controller then sends an INVITE method to User B (4). This INVITE does not contain Session Description Protocol (SDP) information. This INVITE method causes User B's phone to ring.
5. When User B's phone rings, User B answers. A 200 OK (5) message is sent from User B to the third party call controller. The 200 OK message contains an SDP offer (offer2).
6. The third party call controller encapsulates the SDP offer from User B (offer2) into a re-INVITE that is sent to User A (6). The re-INVITE is based on the offer received from User B (offer2). The only difference between offer2' and offer2 is that the origin line (*o* line) in offer2' must be valid based on the *o* line in offer2.
7. User A responds with a 200 OK message (7). This message contains an answer (answer2') to the third party call controller's re-INVITE offer (offer2').
8. The third party call controller encapsulates the SDP answer from User A into an ACK method (answer2). The only difference between answer2 and answer2 is that the origin line (*o* line) in answer2' must be valid based on the *o* line in answer2. The ACK is sent to User B (8).
9. The third party call controller sends an ACK method to User A (9).
10. The RTP (media stream) flows between User A and User B (10).

5.1.2 Global Call Library and IP Media Library for Third Party Call Control

The Dialogic® Global Call API library has been extended to support third party call control (3PCC) mode. Dialogic® Global Call API can be configured to run in either the default first party call control mode (1PCC) or third party call control mode. The two modes are mutually exclusive.

The Dialogic® Global Call API library supports third party call control for SIP networks only. When the Dialogic® Global Call API library is initialized in 3PCC mode, H.323 operations will not be available.

Note: Multimedia (simultaneous audio and video) record/playback is only supported when the Dialogic® Global Call API library is initialized in 3PCC mode.

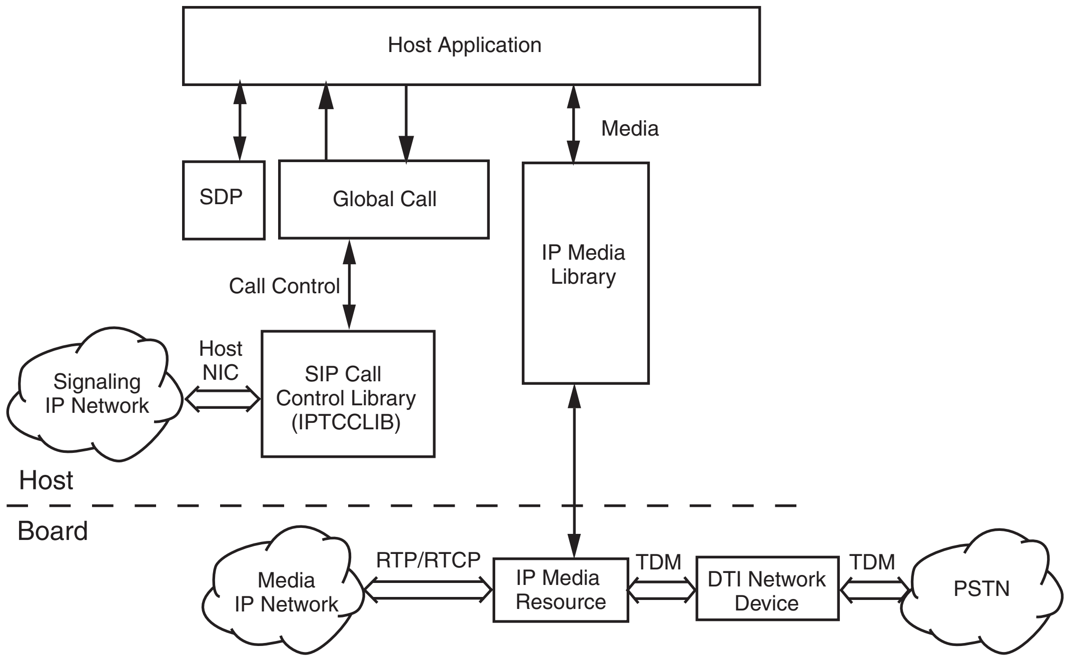
When the Dialogic® Global Call API library is initialized in 1PCC mode, the Dialogic® Global Call API library provides an abstraction layer for the Dialogic® IP Media Library API library. This allows the host application to open and close IP media channels for streaming via the Dialogic®

Global Call API library. The host application does not require direct access to the IP Media Library, as shown in Figure 5, “Dialogic® Global Call API Over IP Architecture”, on page 36.

When the Dialogic® Global Call API library is initialized in 3PCC mode, the host application is directly responsible for calling the Dialogic® IP Media Library API functions to manage RTP streams. This relationship is shown in Figure 63. Refer to the *Dialogic® IP Media Library API Programming Guide* and *Dialogic® IP Media Library API Library Reference* for more information about the IPML API.

Section 5.2, “Global Call in Third Party Call Control Mode”, on page 433 provides a complete overview of the Dialogic® Global Call API library modifications made to support 3PCC mode.

Figure 66. Global Call over IP Architecture for Third Party Call Control Mode



5.1.3 Session Description Protocol

The session description protocol (SDP) is the method of choice for communicating device capabilities between SIP endpoints. SDP is used to exchange endpoint capability information such as coder support, IP address, port information and media stream direction.

When the Dialogic® Global Call API is initialized in 1PCC mode, the SDP content in SIP messages is not exposed to the application. The call control library (IPCCLIB) controls media negotiations internally.

When the Dialogic® Global Call API is initialized in 3PCC mode, the call control library (IPCCLIB) does not internally negotiate media session parameters; the host application is responsible for negotiating the following:

- coordinating matching audio/video coders
- generating SDP data for outgoing SIP messages. These outgoing SIP messages are created when certain Dialogic® Global Call API functions are called (as shown in Table 22, “Summary of IPSET_SDP Parameters and Outbound SIP Messages”, on page 436)
- parsing the SDP data that is attached to incoming Dialogic® Global Call API events (as shown in Table 23, “Summary of Events That Support Global Call SDP Parameter Sets”, on page 437)

In third party call control call flows, the host application is entirely responsible for the SDP contents of SIP messages. The SDP data negotiates media session parameters and connects the RTP media streams. The SDP data is generated and attached to outbound SIP messages. Likewise, SDP data is parsed from incoming Global Call events. The SDP data is then passed to and from host applications in IPSET_SDP parameter set IDs.

There are many open source SDP generator/parser tools that can be used to build Dialogic® Global Call API-based multimedia applications. The Dialogic® Host Media Processing (HMP) Software includes an example SDP generator/parser. The example is described in Section 5.3, “Session Description Protocol Parser/Generator Example”, on page 443.

For complete information about SDP, refer to the Internet Engineering Task Force (IETF) document *RFC 2327 - SDP: Session Description Protocol*, which is available at *http://ietf.org/rfc/rfc2327.txt*.

5.2 Global Call in Third Party Call Control Mode

This section describes the Dialogic® Global Call API modifications and extensions made to support 3PCC mode. The topics are as follows:

- Initializing the Library in Third Party Call Control Mode
- Interface Changes
- Device Types and Usage
- Using Fast Start and Slow Start Setup in Third Party Call Control Mode
- Call Transfer Scenarios
- DTMF Transport
- T.38 Fax and Tone Detection

5.2.1 Initializing the Library in Third Party Call Control Mode

The Dialogic® Global Call API library supports two mutually exclusive modes of operation, first party call control (1PCC) mode and third party call control (3PCC) mode. The mode of operation is set when the Dialogic® Global Call API library is initialized by calling **gc_Start( )**. The IPCCLIB_START_DATA data structure, which is passed to **gc_Start( )** via the

CCLIB_START_STRUCT and GC_START_STRUCT structures, contains a media_operational_mode field that determines the Dialogic® Global Call API library mode of operation. The default value of this field that is set by the **INIT_IPCCLIB_START_DATA( )** initialization function specifies the first party call control (1PCC) mode; applications wishing to use the 3PCC mode must set the media_operational_mode field to the value MEDIA_OPERATIONAL_MODE_3PCC before calling **gc_Start( )**.

The following code snippet shows how an application initializes the CCLIB_START_STRUCT structure and sets the parameter for 3PCC operating mode.

```
#include "gclib.h"
..
..
#define BOARDS_NUM 1
..
..
/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

/* initialize start data structure */
INIT_IPCLIB_START_DATA(&cclibStartData, BOARDS_NUM, virtBoards);

/* initialize virtual board structure */
INIT_IP_VIRTBOARD(&virtBoards[0]);

// set 3PCC operating mode
cclibStartData.media_operational_mode = MEDIA_OPERATIONAL_MODE_3PCC;
```

Note: In order to change the operating mode when the Dialogic® Global Call API library is running, the library must first be stopped by calling the **gc_Stop( )** function. The IP CCLIB does not support the invocation of any library operations after performing a **gc_Stop( )**. This function drops all calls, stops the Dialogic® Global Call API library, and releases all resources so that the library can be restarted in a different mode. However, the application is responsible for terminating all processes after calling **gc_Stop( )**. The application must then be restarted and **gc_Start( )** invoked to change the library and virtual board startup parameters.

Refer to Section 4.1, "Call Control Library Initialization", on page 100 and Section 8.3.27, "gc_Start( ) Variances for IP", on page 590, for more information about initializing the Dialogic® Global Call API library.

5.2.2 Interface Changes

Several Dialogic® Global Call API changes have been made to support 3PCC mode. Third party call control-specific changes are as follows:

- IPCCLIB_START_DATA Data Structure
- IPSET_SDP Parameter Set Identifier
- gc_SipAck( )
- gc_Listen( ) and gc_UnListen( )
- gc_SetUserInfo( ) Duration Defines

- Events
- Error Codes
- Global Call Functions Invalid in Third Party Call Control Mode

5.2.2.1 IPCCLIB_START_DATA Data Structure

The **IPCCLIB_START_DATA** data structure contains a new media_operational_mode field that determines the Dialogic® Global Call API library mode of operation, first party call control mode (1PCC) or third party call control mode (3PCC).

5.2.2.2 IPSET_SDP Parameter Set Identifier

The IPSET_SDP parameter set ID and parameter IDs described in this section, along with those in Section 9, "IP-Specific Parameters", on page 599, are defined in the *gcip.h* header file. The IPSET_SDP parameter set ID is only applicable when the Dialogic® Global Call API library is initialized in 3PCC mode. Applications using this parameter ID **must** use the "extended" **gc_util_..._ex( )** utility functions, which are capable of handling parameter data longer than 255 bytes.

The Dialogic® Global Call API 3PCC model transports SDP buffers to and from the host application through IPSET_SDP parameter set ID parameter blocks that are attached to outbound SIP messages (generated by Dialogic® Global Call API function calls) and inbound Dialogic® Global Call API events.

Notes: ***1.*** The SDP offer/answer should be sent only one time during a transaction. Multiple attempts to send SDP content during a transaction will result in an error.

2. The SDP offer/answer protocol is strictly enforced. If the application receives an SDP offer within a Dialogic® Global Call API event, the application must respond with an SDP answer. The SDP answer is included as part of the Dialogic® Global Call API function call that completes the SIP transaction. For complete information about the offer/answer protocol with SDP, refer to the Internet Engineering Task Force (IETF) document *RFC 3264 - An Offer/Answer Model with Session Description Protocol (SDP)*, which is available at http://ietf.org/rfc/rfc3264.txt.

Table 21 shows the parameter IDs in the IPSET_SDP parameter set. The SDP content string must adhere to the following:

- consist of an array of variable length records of the form <type>=<value>
- each record must contain an array of ISO-10646 characters in UTF-8 encoding
- a record's content must not include 0x00, 0x0a or 0x0d characters
- each record must be terminated with a <CR><LF> (0x0D0A)
- maximum SDP content length is determined by IP_CFG_PARM_DATA_MAXLEN symbolic define

Sample SDP content is shown below:

```
char sdpMsgFormat[] =
          "v=0\r\n"
          "o=Dialogic_IPCCLib %d %d IN IP4 %s\r\n"
          "s=Dialogic_SIP_CCLIB\r\n"
          "i=session information\r\n"
          "c=IN IP4 %s\r\n"
          "t=0 0\r\n"
          "m=audio %d 2000 RTP/AVP %d\r\n";
```

Note: The SDP outbound content is generated by the application. The Dialogic® HMP Software includes an example SDP generator/parser. The example code is described in Section 5.3, "Session Description Protocol Parser/Generator Example", on page 443.

Table 21. IPSET_SDP Parameter Set

Parameter ID	Data Type and Size	Description
IPPARM_SDP_ANSWER	Type: string Size: GC_PARM_DATA_EXTP	Indicates the parameter value is an SDP answer.
IPPARM_SDP_OFFER	Type: string Size: GC_PARM_DATA_EXTP	Indicates the parameter value is an SDP offer.
IPPARM_SDP_OPTION_ANSWER	Type: string Size: GC_PARM_DATA_EXTP	Indicates the parameter data is associated with SIP OPTIONS response.
IPPARM_SDP_OPTION_OFFER	Type: string Size: GC_PARM_DATA_EXTP	Indicates the parameter data is associated with SIP OPTIONS request.
For Global Call parameters of type string, the data size is the length of the string plus 1 (for the null termination).		

Table 22 shows the Dialogic® Global Call API functions that can be used to add SDP content to SIP outbound messages:

Table 22. Summary of IPSET_SDP Parameters and Outbound SIP Messages

Parameter ID	Set	Send	Device Type	Outbound SIP Messages Containing SDP
IPPARM_SDP_ANSWER	**gc_SetUserInfo( )**†	**gc_AcceptCall( )**	CRN	180 Ringing or 1xx progress code
	---	**gc_AcceptModifyCall( )**	CRN	200 OK to re-INVITE
	gc_SetUserInfo( ) †	**gc_AnswerCall( )**	CRN	200 OK to INVITE
	gc_SetUserInfo( ) †	**gc_CallAck( )**	CRN	100 Trying
	gc_SetUserInfo( ) †	**gc_RejectModifyCall( )**	CRN	4xx - 6xx response to re-INVITE
	---	**gc_SipAck( )**	CRN	ACK to 200 OK
† The **duration** parameter must be set to GC_NEXT_OUTBOUND_MSG (to apply on next outbound message).				

Table 22. Summary of IPSET_SDP Parameters and Outbound SIP Messages (Continued)

Parameter ID	Set	Send	Device Type	Outbound SIP Messages Containing SDP
IPPARM_SDP_ OFFER	**gc_SetUserInfo( )**†	**gc_AcceptCall( )**	CRN	180 Ringing or 1xx progress code
	---	**gc_AcceptModifyCall( )**	CRN	200 OK to re-INVITE
	gc_SetUserInfo( ) †	**gc_AnswerCall( )**	CRN	200 OK to INVITE
	gc_SetUserInfo( ) †	**gc_CallAck( )**	CRN	100 Trying
	---	**gc_MakeCall( )**	LD	INVITE
	---	**gc_ReqModifyCall( )**	CRN	re-INVITE
IPPARM_SDP_ OPTION_ANSWER	---	**gc_Extension( )** for IPEXTID_SENDMSG, with GC_PARM_BLK containing IPSET_MSG_SIP / IPPARM_MSGTYPE / IP_MSGTYPE_SIP_ OPTIONS_OK	CRN	200 OK to OPTIONS
IPPARM_SDP_ OPTION_OFFER	--	**gc_Extension( )**, for IPEXTID_SENDMSG with GC_PARM_BLK containing IPSET_MSG_SIP / IPPARM_MSGTYPE / IP_MSGTYPE_SIP_ OPTIONS_OK	CRN	OPTIONS message

† The **duration** parameter must be set to GC_NEXT_OUTBOUND_MSG (to apply on next outbound message).

Table 23 shows the inbound Dialogic® Global Call API events that may have SDP content attached. The host application must register for all events that may contain SDP content. Host applications can retrieve the SDP content by parsing the attached parameter block for IPSET_SDP parameter IDs shown in Table 21, “IPSET_SDP Parameter Set”, on page 436.

Note: The SDP inbound content is parsed by the application. The Dialogic® HMP Software includes an example SDP generator/parser. The example code is described in Section 5.3, “Session Description Protocol Parser/Generator Example”, on page 443.

Table 23. Summary of Events That Support Global Call SDP Parameter Sets

Global Call Event with Possible SDP Parameter Set Attached†	Device Type	Inbound SIP Message with optional SDP
GCEV_ALERTING	CRN	180 Ringing or 1xx progress code
GCEV_ANSWERED	CRN	ACK
GCEV_CANCEL_MODIFY_CALL	CRN	remote party responded with a 200OK when the application sent a CANCEL request for a pending re-INVITE
GCEV_CONNECTED	CRN	200 OK to INVITE

† The Global Call event may contain a pointer to an EXTENSIONEVENTBLK which contains a pointer to a parameter block. This parameter block may contain SDP content as an IPSET_SDP parameter ID.

Table 23. Summary of Events That Support Global Call SDP Parameter Sets (Continued)

Global Call Event with Possible SDP Parameter Set Attached†	Device Type	Inbound SIP Message with optional SDP
GCEV_EXTENSION (IPEXTID_)	BRD CRN	SIP OPTIONS message (inbound request) SIP OPTIONS message's 200 OK reply (inbound response success) **Note:** This is for the IXPEXTID_RECEIVEMSG extension event type only, with a parameter block containing IPPARM_MSGTYPE parameter value of IP_MSGTYPE_SIP_OPTIONS.
GCEV_MODIFY_CALL_ACK	CRN	200 OK response to re-INVITE
GCEV_MODIFY_CALL_REJ	CRN	3xx-6xx response to re-INVITE
GCEV_OFFERED	LD	INVITE
GCEV_PROCEEDING	CRN	100 Trying
GCEV_REQ_MODIFY_CALL	CRN	re-INVITE
GCEV_SIP_ACK	CRN	ACK

† The Global Call event may contain a pointer to an EXTENSIONEVENTBLK which contains a pointer to a parameter block. This parameter block may contain SDP content as an IPSET_SDP parameter ID.

IPSET_SDP Code Example

```
/*                                                              */
/* Description: Send a SIP re-INVITE containing an SDP offer    */
/*              for IP Address, RTP port, coder type.           */
/*                                                              */
/* Assumes: 1) caller has verified call to be in connected state */
/*                                                              */

int sendOfferOnReinvite(CRN crn, long time, char *pIpAddr,
                          short rtpPort,  short coderType)

{
      int                  dataSize;
      int                  rc;
      GC_PARM_BLKP         gcParmBlk = NULL;
      char sdpMsg[512];

      char sdpMsgFormat[] =
                "v=0\r\n"
                "o=Dialogic_IPCCLib %d %d IN IP4 %s\r\n"
                "s=Dialogic_SIP_CCLIB\r\n"
                "i=session information\r\n"
                "c=IN IP4 %s\r\n"
                "t=0 0\r\n"
                "m=audio %d 2000 RTP/AVP %d\r\n";


        /* initialize the SDP content */
        sprintf(sdpMsg,  sdpMsgFormat, time, time + 1,
                  pIpAddr, pIpAddr, rtpPort, coderType);

        /* Add 1 to strlen for null termination */
        data_size = strlen(sdpMsg) + 1;
        /* put the SDP content into the parameter block */
        rc = gc_util_insert_parm_ref_ex(&gcParmBlk,
                                        IPSET_SDP,           /* set value */
                                        IPPARM_SDP_OFFER,    /* parm value */
                                        data_size
                                        (void *) sdpMsg);


        if (rc != 0) return FAILURE;


        /* send the re-INVITE message */
        if (gc_ReqModifyCall(crn, gcParmBlk, EV_ASYNC) != GC_SUCCESS)
              return FAILURE;


        /* cleanup and exit */
        gc_util_delete_parm_blk(gcParmBlk);
        return SUCCESS;
}
```

5.2.2.3 gc_SipAck( )

The gc_SipAck( ) function is used to send an ACK message to the remote party on an outbound INVITE and re-INVITE transaction. The Dialogic® Global Call API library cannot automatically send an ACK after it receives a 200 OK message, so the host application must call this function in response to the reception of a GCEV_SIP_200OK event. The ACK completes the dialog's transaction, avoiding time-out/call failure issues. SDP content may be attached to the ACK message by including a pointer to a parameter block that contains an element with the IPSET_SDP parameter set ID.

5.2.2.4 gc_Listen( ) and gc_UnListen( )

When the Dialogic® Global Call API library is initialized in 3PCC mode, **gc_Listen( )** and **gc_UnListen( )** requests are routed directly to the IPML. Valid IPML device handles are required for the **gc_Listen( )** and **gc_UnListen( )** functions, but Dialogic® Global Call API media control functions are not available when the Dialogic® Global Call API library is initialized in 3PCC mode.

Refer to Section 5.2.2.8, "Global Call Functions Invalid in Third Party Call Control Mode", on page 441 for information about media control functions that are not available when Dialogic® Global Call API is initialized in 3PCC mode.

5.2.2.5 gc_SetUserInfo( ) Duration Defines

A new duration define, GC_NEXT_OUTBOUND_MSG, is available for the **gc_SetUserInfo( )** function. This duration define is only valid when the Dialogic® Global Call API library is initialized in 3PCC mode. It has been included because the GC_SINGLECALL and GC_ALLCALLS are not sufficient for the life cycle of an IPSET_SDP parameter data set.

The GC_SINGLE_CALL and GC_ALL_CALLS duration defines are also inadequate for setting MIME data, and some generic header data. The GC_NEXT_OUTBOUND_MSG duration define implies that the data set is only valid until the next outbound SIP message is sent. The

GC_SINGLECALL and GC_ALLCALLS duration defines imply that the data set is valid for the entire length of the call.

Refer to Section 4.3, “Setting Call-Related Information”, on page 114 for an overview of the GC_SINGLECALL and GC_ALLCALLS duration defines.

Note: If the host application needs to set a CRN's or a device's parameters with different duration defines, then the application must call **gc_SetUserInfo( )** multiple times. All parameters set within a single **gc_SetUserInfo( )** function call have the same duration value.

5.2.2.6 Events

Table 24 provides information about the events that have been added to the Dialogic® Global Call API library to support third party call control mode. The application must use the **gc_SetConfigData( )** function to register for the events in Table 24. These events are only valid when the Dialogic® Global Call API library is initialized in 3PCC mode:

Table 24. Global Call Third Party Call Control Mode Events

Global Call Event	Description	Third Party Call Control Mode Notes
GCEV_SIP_200OK	Unsolicited event posted to application upon reception of a SIP 200 OK message on an active dialog (not in received responses to BYE or OPTIONS messages). The application must register for this event.	Application should call **gc_SipAck( )** in response to this event.
GCEV_SIP_ACK	Unsolicited event posted to application upon reception of a SIP ACK message on an active dialog (not posted during BYE or OPTIONS transactions). The application must register for this event.	This event may include IPSET_SDP in attached parameter block.
GCEV_SIP_ACK_FAILED	Failure completion termination event, associated with the **gc_SipAck( )** function.	This event may indicate a message transport failure.
GCEV_SIP_ACK_OK	Successful completion termination event, associated with the **gc_SipAck( )** function.	Indicates that the SIP ACK message was sent. It does not mean the SDP content, if any, was valid.

5.2.2.7 Error Codes

Table 25 provides information about the error codes that have been added to the library to support third party call control mode. These error codes are only valid when the Dialogic® Global Call API library is initialized in 3PCC mode:

Table 25. Global Call Third Party Call Control Mode Error Codes

Global Call Error	Description
EGC_INVALID_IN_1PCC	Generated when a 3PCC mode-specific function is called while the Dialogic® Global Call library is initialized in 1PCC mode.
EGC_NO_MEDIA_IN_3PCC	Generated when a media control function is called while the Dialogic® Global Call library is initialized in 3PCC mode. Section 5.2.2.8, "Global Call Functions Invalid in Third Party Call Control Mode", on page 441 describes the Global Call media control functions that are not valid when the Dialogic® Global Call library is initialized in 3PCC mode.
EGC_RESOURCE_NOT_LICENSED	Generated when the IPCCLIB library cannot obtain authorization from the product's license authority to use a resource (SIP signaling port or third party call control library instance).
EGC_SDP_ANSWER_MISSING	Generated when an SDP offer was received and no SDP answer was generated. This indicates a violation of the SDP offer/answer protocol.

5.2.2.8 Global Call Functions Invalid in Third Party Call Control Mode

When the Dialogic® Global Call API library is operating in 3PCC mode, the application is responsible for manipulating media behavior via the IPML. Therefore, the Dialogic® Global Call API media-specific functions are considered invalid in 3PCC mode. Table 26 lists these invalid functions:

Table 26. Global Call Functions Invalid in Third Party Call Control Mode

Global Call Function	Device Type	Parameter Set ID or Extension IDs (if applicable)	Result when Function is Called in Third Party Call Control Mode
gc_AttachResource( )	LD		Returns GC_ERROR
gc_Detach( )	LD		Returns GC_ERROR
gc_Extension( )	BRD CRN LD	IPEXTID_MEDIAINFO IPEXTID_SEND_DTMF IPEXTID_RECEIVE_DTMF IPEXTID_FOIP IPEXTID_CHANGEMODE	Returns GC_ERROR
gc_MakeCall( )	CRN	IPSET_MEDIA_STATE IPSET_FOIP GCSET_CHAN_CAPABILITY IPSET_DTMF IPSET_CALLINFO/IPPARM_ CONNECTIONMETHOD	Parameters using indicated set IDs are ignored
gc_ReqModifyCall( )	CRN	IPSET_MEDIA_STATE IPSET_FOIP GCSET_CHAN_CAPABILITY IPSET_DTMF	Parameters using indicated set IDs are ignored

Table 26. Global Call Functions Invalid in Third Party Call Control Mode (Continued)

Global Call Function	Device Type	Parameter Set ID or Extension IDs (if applicable)	Result when Function is Called in Third Party Call Control Mode
gc_SetConfigData( )	LD	IPSET_MEDIA_STATE IPSET_FOIP GCSET_CHAN_CAPABILITY IPSET_DTMF IPSET_CALLINFO/IPPARM_ CONNECTIONMETHOD	Parameters using indicated set IDs are ignored
gc_SetUserInfo( )	BRD CRN LD	IPSET_MEDIA_STATE IPSET_FOIP GCSET_CHAN_CAPABILITY IPSET_DTMF IPSET_CALLINFO/IPPARM_ CONNECTIONMETHOD	Parameters using indicated set IDs are ignored

5.2.3 Device Types and Usage

When using Dialogic® Global Call API in 3PCC mode, the **gc_OpenEx( )** and **gc_AttachResource( )** functions cannot be used to attach an IPT network device to an IPM media device. IPM media devices cannot be opened via Dialogic® Global Call API when the library is initialized in 3PCC mode. IPM media devices must be opened by the Dialogic® IP Media Library API library when the Dialogic® Global Call API is initialized in 3PCC mode.

When initialized in 3PCC mode, valid Dialogic® Global Call API devices are limited to IPT board devices and IPT network devices. Refer to Section 2.3, "Device Types and Usage", on page 38 for information about these device types.

5.2.4 Using Fast Start and Slow Start Setup in Third Party Call Control Mode

When the Dialogic® Global Call API library is initialized in 3PCC mode, the host application controls whether or not SDP information is included in the outbound INVITE message.

Note: When initialized in 3PCC mode, the Dialogic® Global Call API library ignores the IPSET_CALLINFO / IPPARM_CONNECTIONMETHOD parameter; this parameter is only valid when Dialogic® Global Call API is initialized in 1PCC mode. Refer to Section 4.2, "Fast and Slow Call Setup Modes", on page 108 for information.

5.2.5 Call Transfer Scenarios

Call transfer behavior when the Dialogic® Global Call API is initialized in 3PCC mode is identical to call transfer behavior when the Dialogic® Global Call API is initialized in 1PCC mode. In other words, no additional interface changes are required to implement call transfer in 3PCC mode. Refer to Section 3.3, "Call Transfer Scenarios When Using SIP", on page 66 for more information.

5.2.6 DTMF Transport

The media management interface required to support in-band DTMF transport is not available when the Dialogic® Global Call API is initialized in 3PCC mode. As a result, the application must use IP Medai Library functions to specify the type of DTMF transport to use (in-band or out-of-band).

5.2.7 T.38 Fax and Tone Detection

When the Dialogic® Global Call API is initialized in 3PCC mode, the Dialogic® Global Call API library does not provide support for T.38 fax or in-band tone detection. The host application is responsible for performing T.38 signaling through re-INVITE functionality and the SDP.

5.3 Session Description Protocol Parser/Generator Example

The session description protocol (SDP) is the method of choice for communicating device capabilities between SIP endpoints. SDP is transported as payload in SIP message bodies. SDP is used to exchange endpoint capability information such as coder support, IP address, port information and media stream direction. The SDP information is required to manage media resources, such as IPM devices and RTP media streams.

The host application must parse the SDP information incoming Dialogic® Global Call API events. Likewise, the host application must insert SDP content into the make call parameter block before making a call.

The Dialogic® HMP Software includes an example SDP generator/parser. The default installation path for the example SDP generator/parser is *%INTEL_DIALOGIC_DIR%\demos*, where *%INTEL_DIALOGIC_DIR%* is the environment variable for the directory in which the Dialogic® HMP Software is installed. The example includes:

- demo application
- source code
- binary code
- design diagrams

5.4 Message Sequence Diagrams

The following third party call control message sequence diagrams are included:

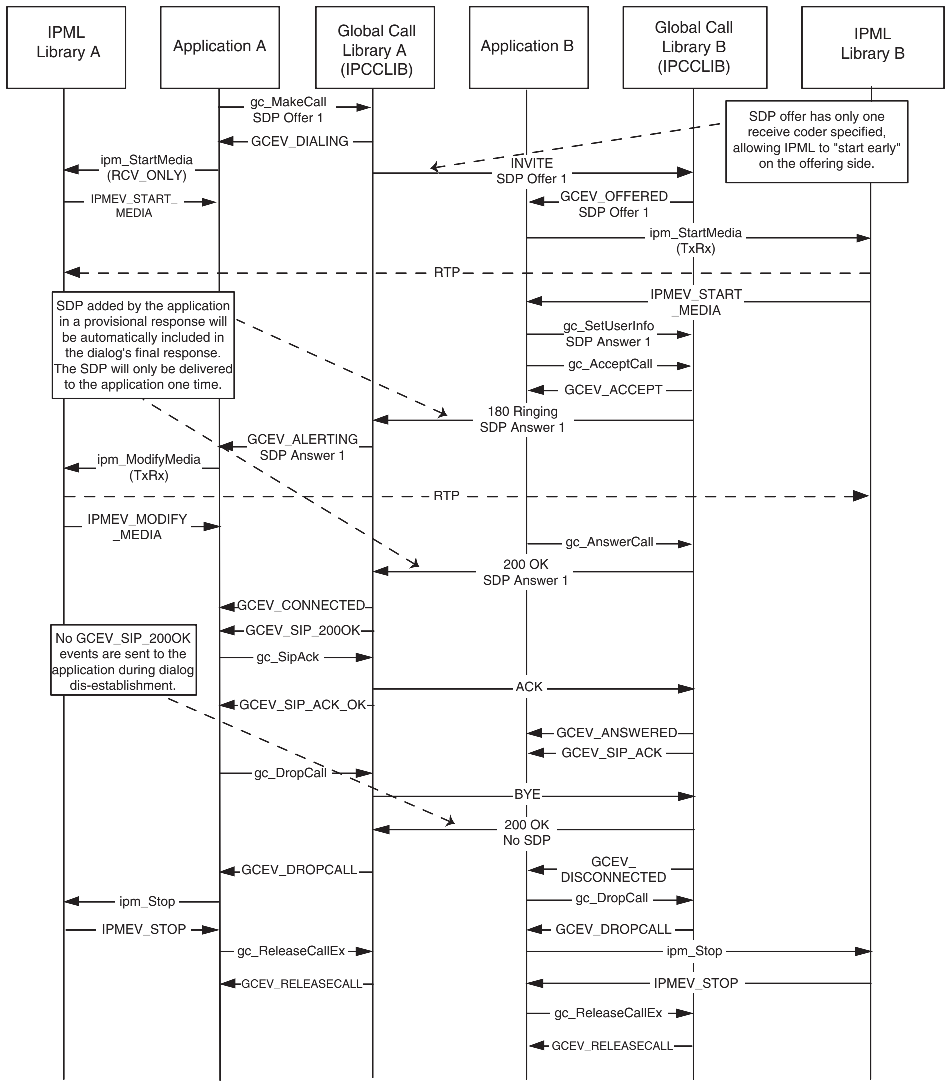
- First Party Call Establishment in Third Party Call Control Mode
- Basic Third Party Call Control Establishment
- Alternate Third Party Call Control Establishment
- Modifying the Coder

- Cancelling a re-INVITE Request
- Receiving an Invalid Answer SDP
- OPTIONS Request on an Active Dialog

5.4.1 First Party Call Establishment in Third Party Call Control Mode

Figure 67 shows how to implement a simple point-to-point SIP call between two endpoints (party A and party B). The Dialogic® Global Call API library is initialized in 3PCC mode. A third party call controller is not used:

Figure 67. First Party Call Control Establishment in Third Party Call Control Mode



5.4.2 Basic Third Party Call Control Establishment

Figure 68 and Figure 69 show how to establish a basic third party call control call. A third party call controller, party B, establishes signaling connections to two different endpoints, party A and party C. The third party call controller then establishes RTP media paths between the two endpoints:

Figure 68. Basic Third Party Call Control Establishment (part one)

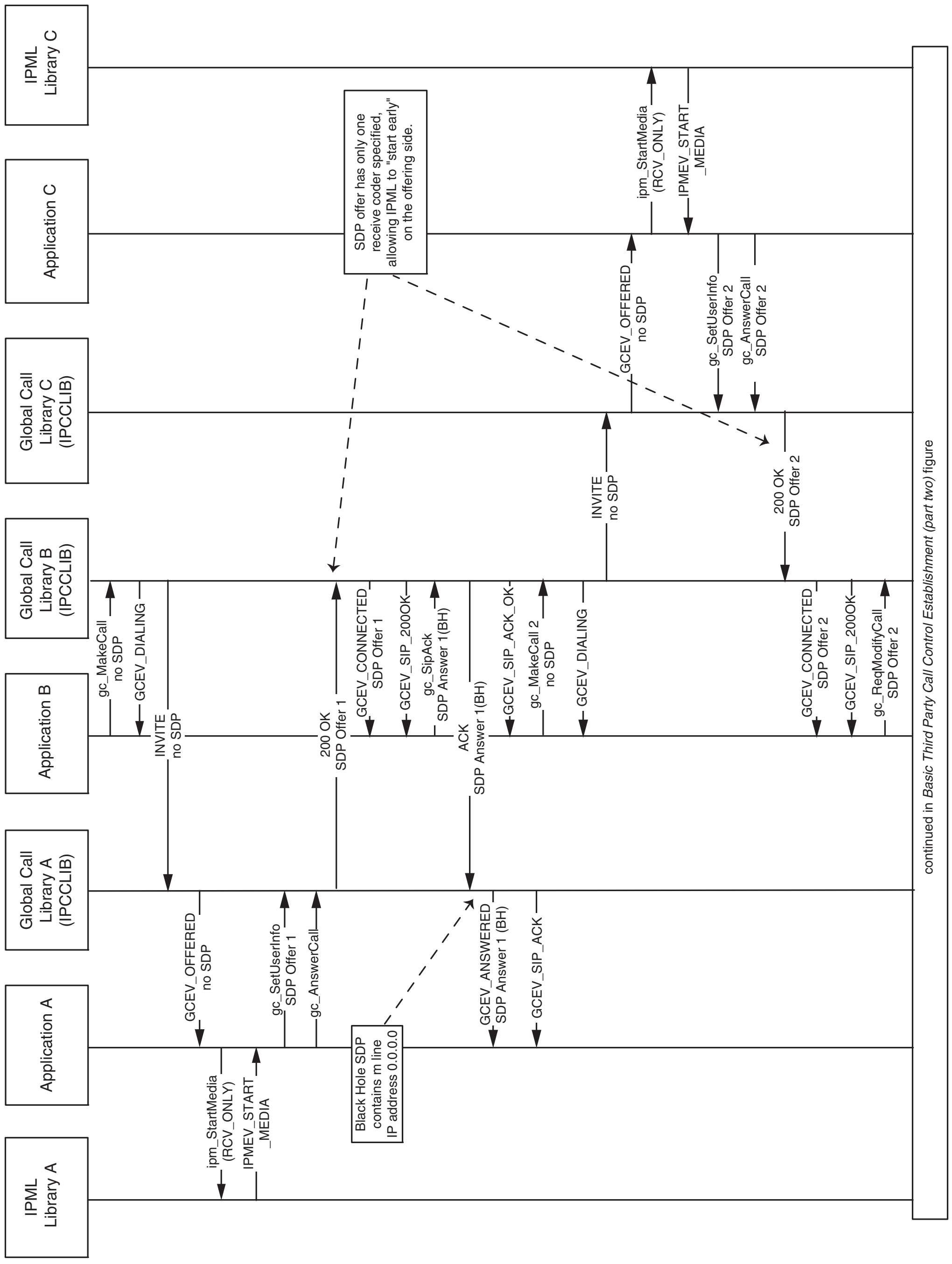
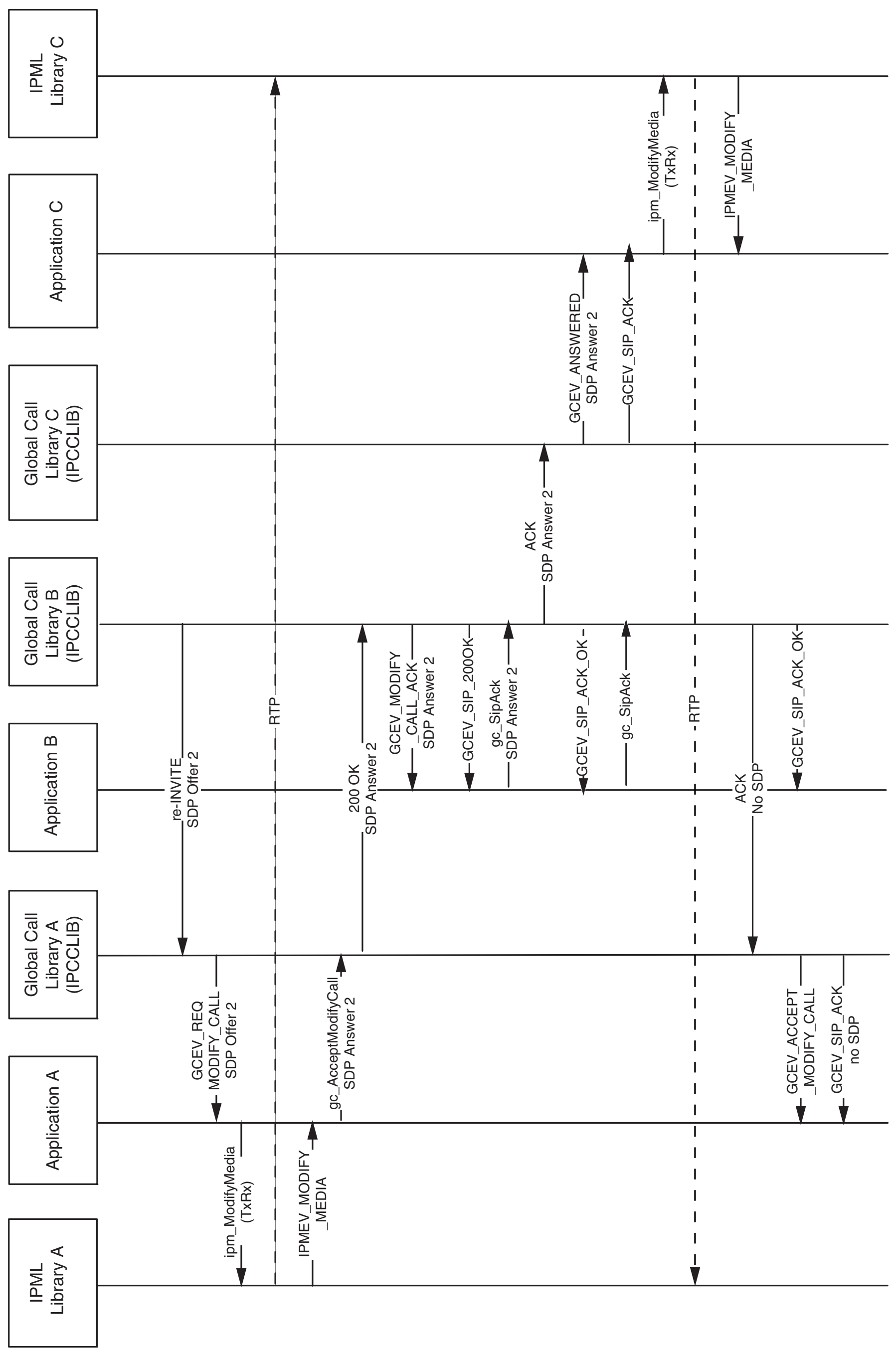

IPML Library C
Application C
Global Call Library C (IPCCLIB)
Global Call Library B (IPCCLIB)
Application B
Global Call Library A (IPCCLIB)
Application A
IPML Library A
gc_MakeCall no SDP
GCEV_DIALING
INVITE no SDP
GCEV_OFFERED no SDP
ipm_StartMedia (RCV_ONLY)
IPMEV_START _MEDIA
gc_SetUserInfo SDP Offer 1
gc_AnswerCall
Black Hole SDP contains m line IP address 0.0.0.0
200 OK SDP Offer 1
SDP offer has only one receive coder specified, allowing IPML to "start early" on the offering side.
GCEV_CONNECTED SDP Offer 1
GCEV_SIP_200OK
gc_SipAck SDP Answer 1(BH)
ACK SDP Answer 1(BH)
GCEV_ANSWERED SDP Answer 1 (BH)
GCEV_SIP_ACK
GCEV_SIP_ACK_OK
gc_MakeCall 2 no SDP
GCEV_DIALING
INVITE no SDP
GCEV_OFFERED no SDP
ipm_StartMedia (RCV_ONLY)
IPMEV_START _MEDIA
gc_SetUserInfo SDP Offer 2
gc_AnswerCall SDP Offer 2
200 OK SDP Offer 2
GCEV_CONNECTED SDP Offer 2
GCEV_SIP_200OK
gc_ReqModifyCall SDP Offer 2
continued in Basic Third Party Call Control Establishment (part two) figure

Figure 69. Basic Third Party Call Control Establishment (part two)



5.4.3 Alternate Third Party Call Control Establishment

Figure 70, Figure 71, and Figure 72 show an alternate third party call control flow. In this sequence, party A, establishes a dialog and media connection with another user agent, party B. Party B then establishes a dialog with a third user agent, party C. Party B then re-establishes the media path from A-B to A-C:

Figure 70. Alternate Third Party Call Control Establishment (part one)

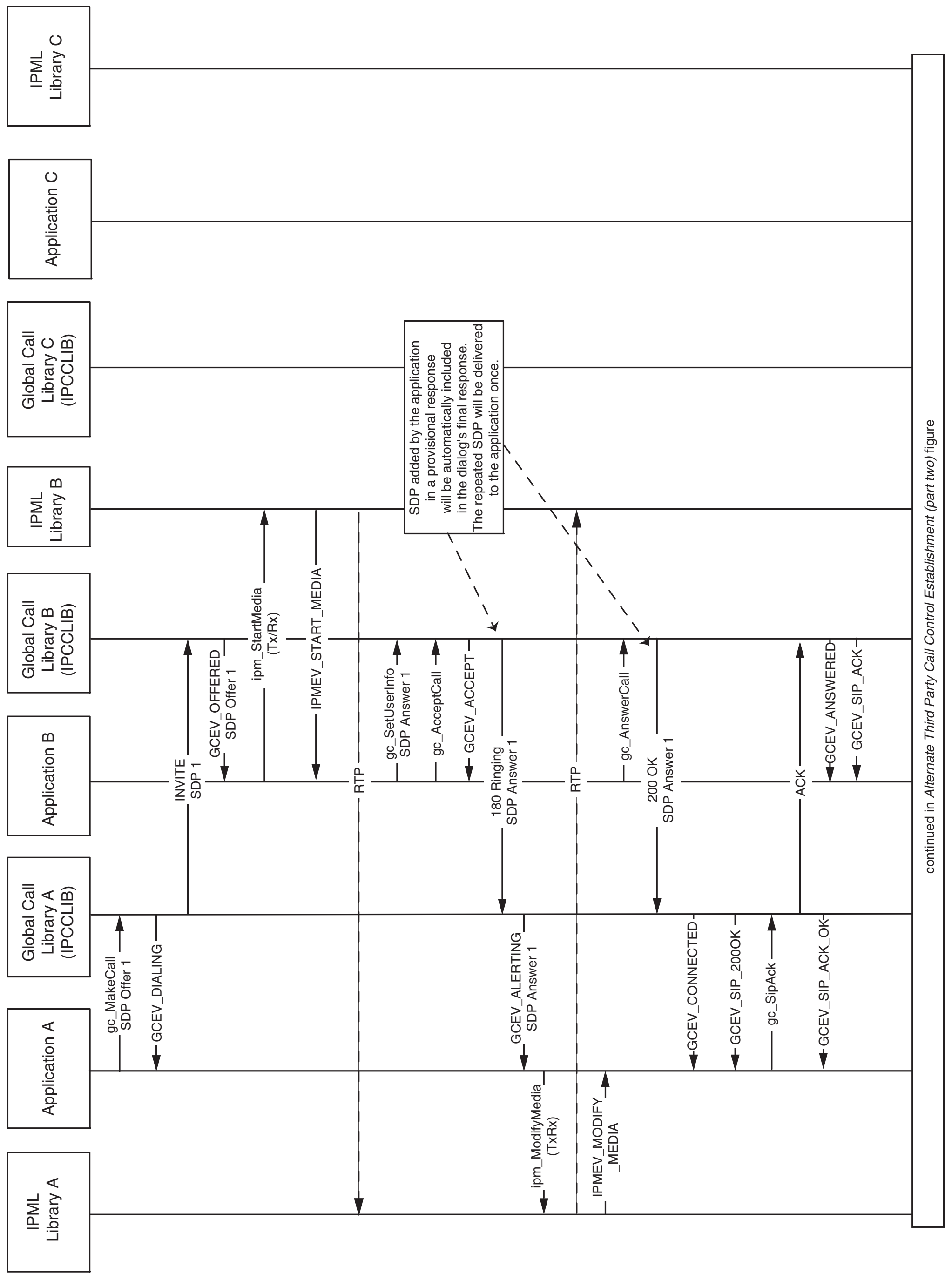

IPML Library C
Application C
Global Call Library C (IPCCLIB)
IPML Library B
Global Call Library B (IPCCLIB)
Application B
Global Call Library A (IPCCLIB)
Application A
IPML Library A
gc_MakeCall SDP Offer 1
GCEV_DIALING
INVITE SDP 1
GCEV_OFFERED SDP Offer 1
ipm_StartMedia (Tx/Rx)
IPMEV_START_MEDIA
RTP
gc_SetUserInfo SDP Answer 1
gc_AcceptCall
GCEV_ACCEPT
SDP added by the application in a provisional response will be automatically included in the dialog's final response. The repeated SDP will be delivered to the application once.
180 Ringing SDP Answer 1
GCEV_ALERTING SDP Answer 1
ipm_ModifyMedia (TxRx)
RTP
IPMEV_MODIFY_MEDIA
gc_AnswerCall
200 OK SDP Answer 1
GCEV_CONNECTED
GCEV_SIP_200OK
gc_SipAck
ACK
GCEV_ANSWERED
GCEV_SIP_ACK
GCEV_SIP_ACK_OK
continued in *Alternate Third Party Call Control Establishment (part two)* figure

Figure 71. Alternate Third Party Call Control Establishment (part two)

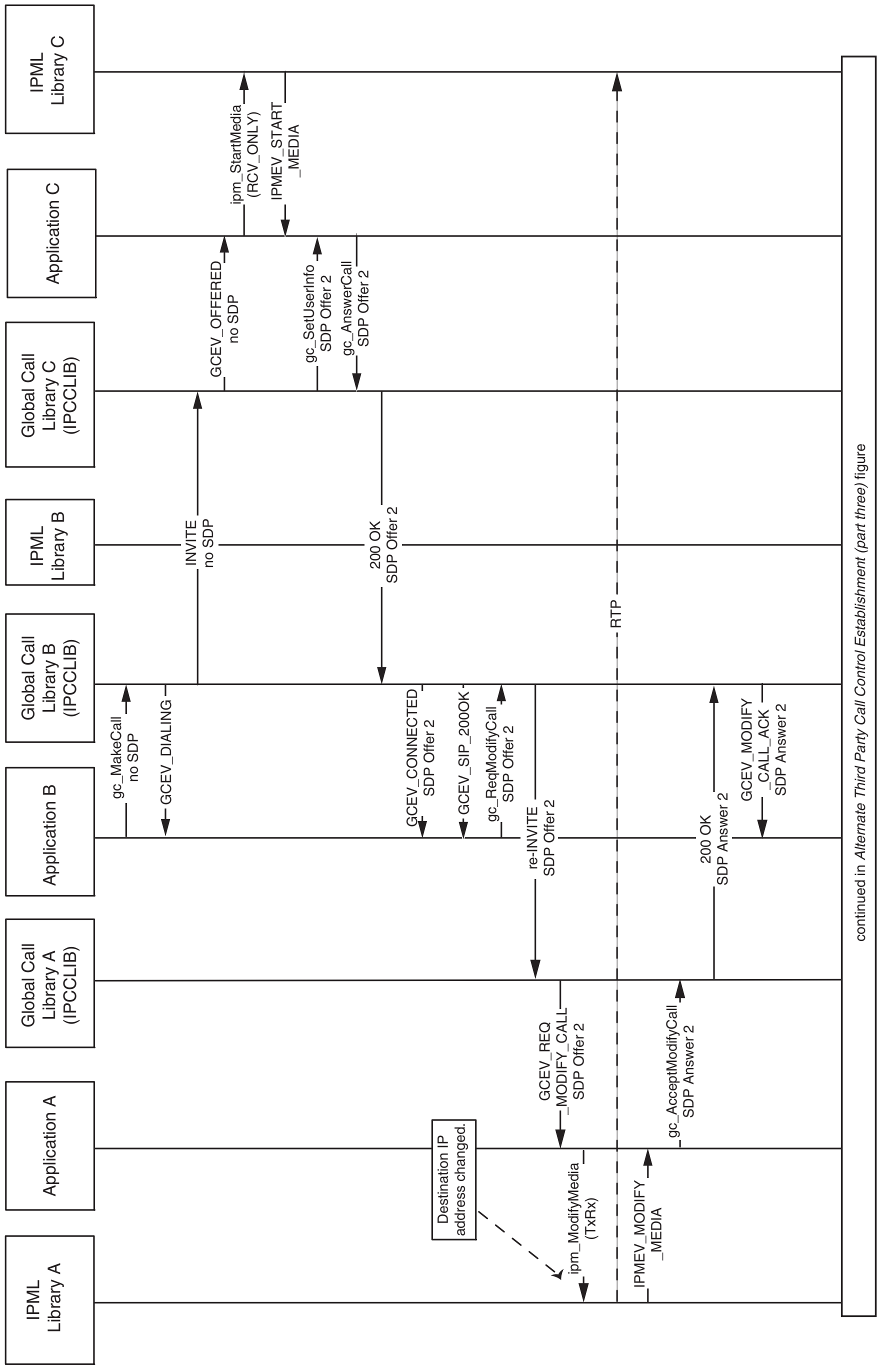
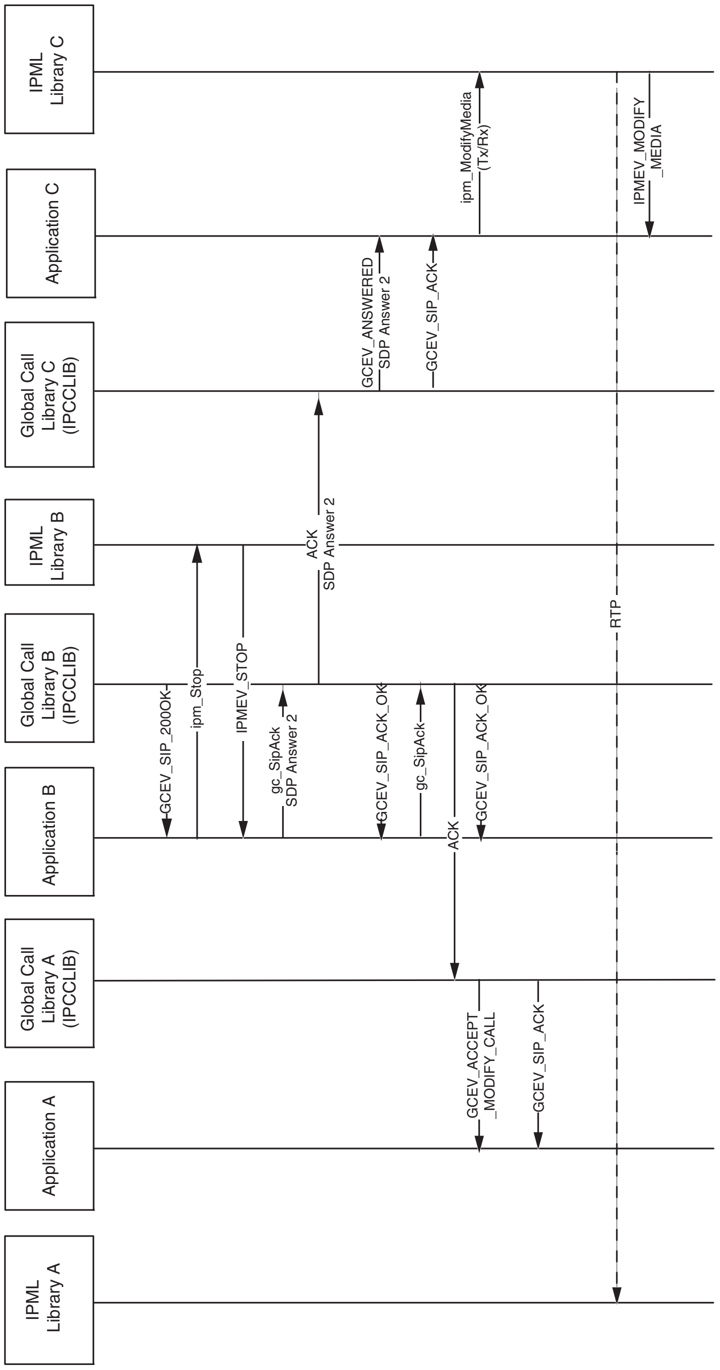

IPML Library C
Application C
Global Call Library C (IPCCLIB)
IPML Library B
Global Call Library B (IPCCLIB)
Application B
Global Call Library A (IPCCLIB)
Application A
IPML Library A
gc_MakeCall no SDP
GCEV_DIALING
INVITE no SDP
GCEV_OFFERED no SDP
ipm_StartMedia (RCV_ONLY)
IPMEV_START _MEDIA
gc_SetUserInfo SDP Offer 2
gc_AnswerCall SDP Offer 2
200 OK SDP Offer 2
GCEV_CONNECTED SDP Offer 2
GCEV_SIP_200OK
gc_ReqModifyCall SDP Offer 2
re-INVITE SDP Offer 2
GCEV_REQ _MODIFY_CALL SDP Offer 2
Destination IP address changed.
ipm_ModifyMedia (TxRx)
RTP
IPMEV_MODIFY _MEDIA
gc_AcceptModifyCall SDP Answer 2
200 OK SDP Answer 2
GCEV_MODIFY _CALL_ACK SDP Answer 2
continued in Alternate Third Party Call Control Establishment (part three) figure

Figure 72. Alternate Third Party Call Control Establishment (part three)



5.4.4 Modifying the Coder

When running in 3PCC mode, the host application can use the re-INVITE functionality to modify the coders of an active dialog. This section provides the following message sequence diagrams:

- Successfully Modifying the Coder
- Unsuccessfully Modifying the Coder

5.4.4.1 Successfully Modifying the Coder

Figure 73 and Figure 74 show the message sequence diagram for successfully modifying the coders of an active dialog:

Figure 73. Successfully Modifying the Coder (part one)

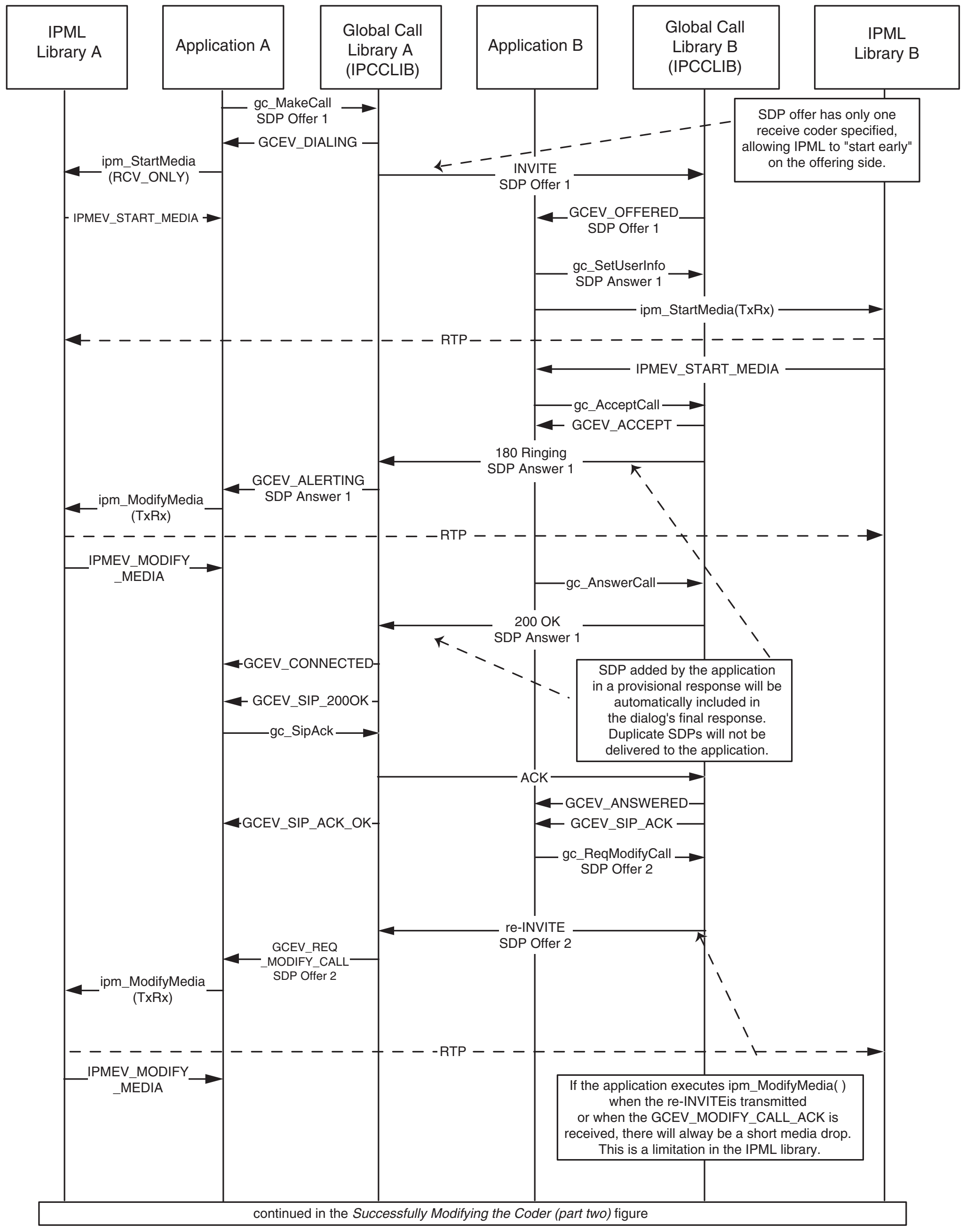
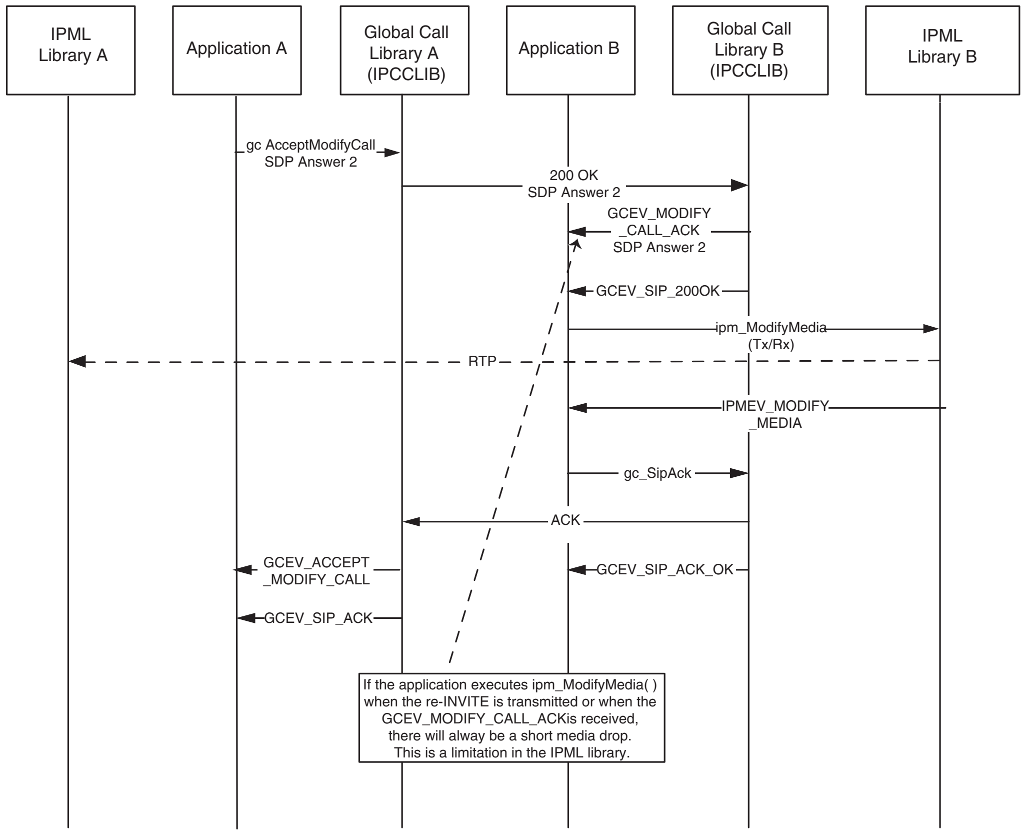

IPML Library A
Application A
Global Call Library A (IPCCLIB)
Application B
Global Call Library B (IPCCLIB)
IPML Library B
gc_MakeCall SDP Offer 1
SDP offer has only one receive coder specified, allowing IPML to "start early" on the offering side.
GCEV_DIALING
ipm_StartMedia (RCV_ONLY)
INVITE SDP Offer 1
IPMEV_START_MEDIA
GCEV_OFFERED SDP Offer 1
gc_SetUserInfo SDP Answer 1
ipm_StartMedia(TxRx)
RTP
IPMEV_START_MEDIA
gc_AcceptCall
GCEV_ACCEPT
180 Ringing SDP Answer 1
GCEV_ALERTING SDP Answer 1
ipm_ModifyMedia (TxRx)
RTP
IPMEV_MODIFY_MEDIA
gc_AnswerCall
200 OK SDP Answer 1
GCEV_CONNECTED
SDP added by the application in a provisional response will be automatically included in the dialog's final response. Duplicate SDPs will not be delivered to the application.
GCEV_SIP_200OK
gc_SipAck
ACK
GCEV_ANSWERED
GCEV_SIP_ACK_OK
GCEV_SIP_ACK
gc_ReqModifyCall SDP Offer 2
re-INVITE SDP Offer 2
GCEV_REQ_MODIFY_CALL SDP Offer 2
ipm_ModifyMedia (TxRx)
RTP
IPMEV_MODIFY_MEDIA
If the application executes ipm_ModifyMedia( ) when the re-INVITEis transmitted or when the GCEV_MODIFY_CALL_ACK is received, there will alway be a short media drop. This is a limitation in the IPML library.
continued in the *Successfully Modifying the Coder (part two)* figure

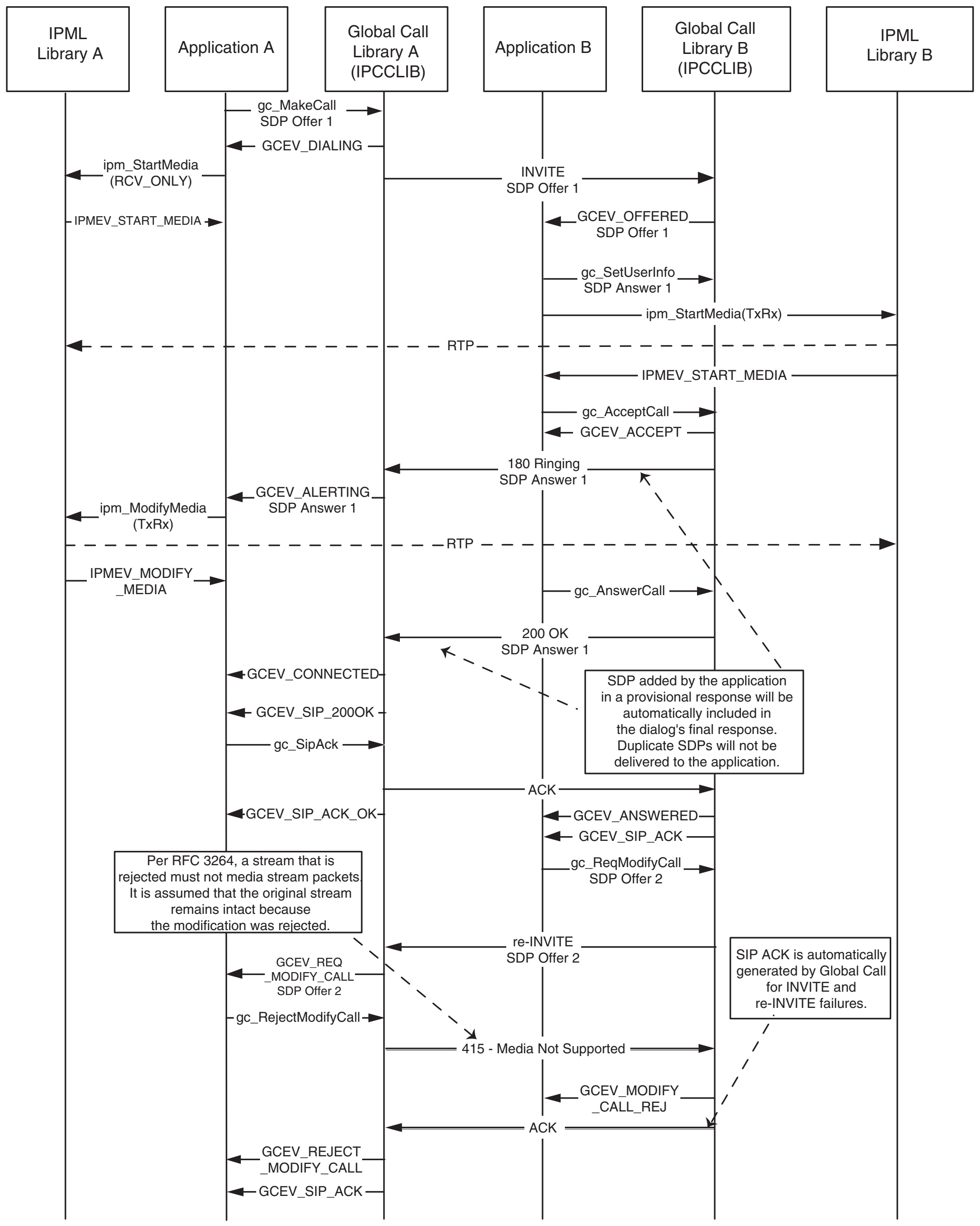
Figure 74. Successfully Modifying the Coder (part two)


IPML Library A
Application A
Global Call Library A (IPCCLIB)
Application B
Global Call Library B (IPCCLIB)
IPML Library B
gc AcceptModifyCall SDP Answer 2
200 OK SDP Answer 2
GCEV_MODIFY _CALL_ACK SDP Answer 2
GCEV_SIP_200OK
ipm_ModifyMedia (Tx/Rx)
RTP
IPMEV_MODIFY _MEDIA
gc_SipAck
ACK
GCEV_ACCEPT _MODIFY_CALL
GCEV_SIP_ACK_OK
GCEV_SIP_ACK
If the application executes ipm_ModifyMedia( ) when the re-INVITE is transmitted or when the GCEV_MODIFY_CALL_ACKis received, there will alway be a short media drop. This is a limitation in the IPML library.

5.4.4.2 Unsuccessfully Modifying the Coder

Figure 75 shows a message sequence diagram for an unsuccessful attempt at modifying the coder of an active dialog:

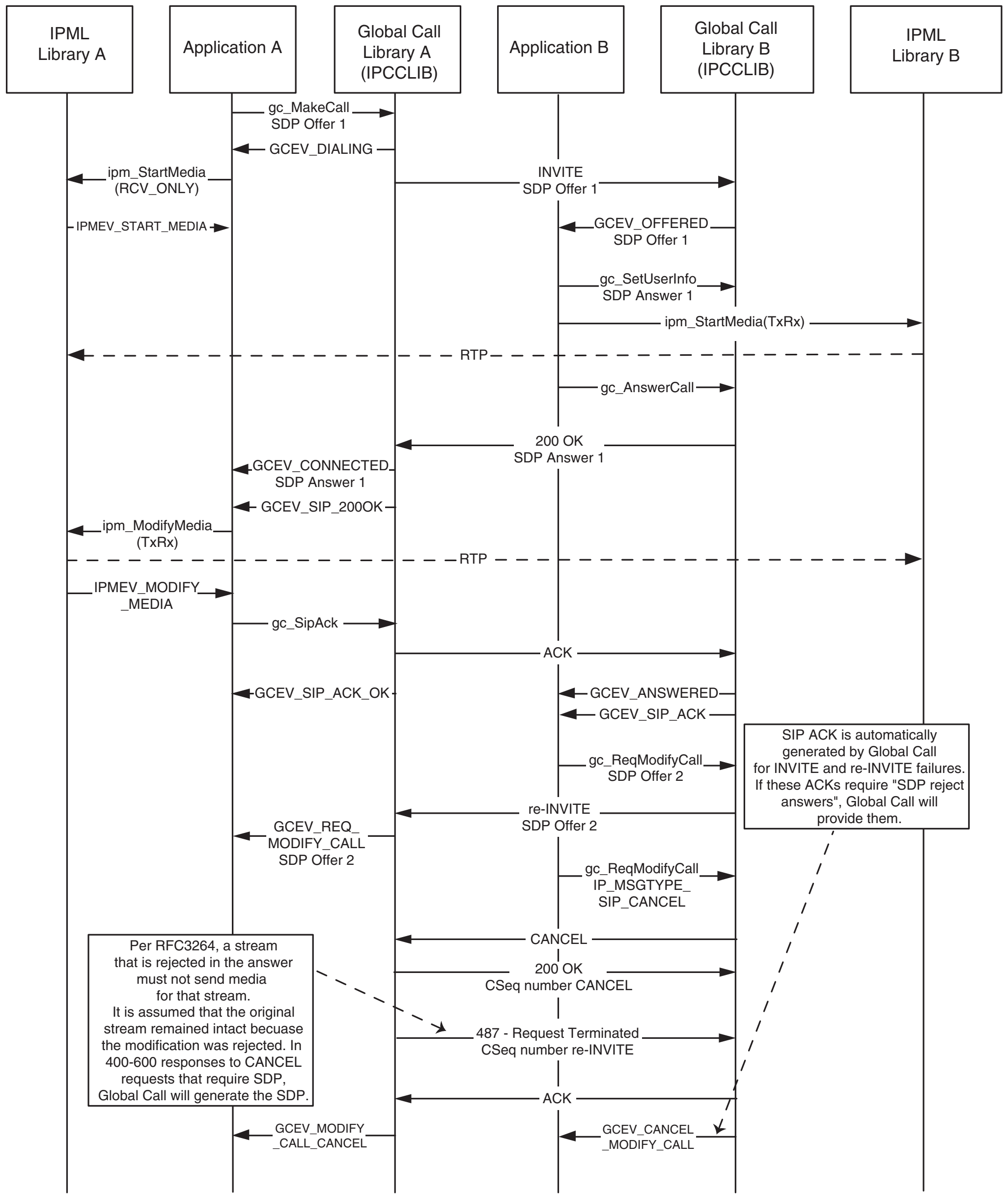
Figure 75. Unsuccessfully Modifying the Coder


IPML Library A
Application A
Global Call Library A (IPCCLIB)
Application B
Global Call Library B (IPCCLIB)
IPML Library B
gc_MakeCall SDP Offer 1
GCEV_DIALING
ipm_StartMedia (RCV_ONLY)
INVITE SDP Offer 1
IPMEV_START_MEDIA
GCEV_OFFERED SDP Offer 1
gc_SetUserInfo SDP Answer 1
ipm_StartMedia(TxRx)
RTP
IPMEV_START_MEDIA
gc_AcceptCall
GCEV_ACCEPT
180 Ringing SDP Answer 1
GCEV_ALERTING SDP Answer 1
ipm_ModifyMedia (TxRx)
RTP
IPMEV_MODIFY _MEDIA
gc_AnswerCall
200 OK SDP Answer 1
GCEV_CONNECTED
SDP added by the application in a provisional response will be automatically included in the dialog's final response. Duplicate SDPs will not be delivered to the application.
GCEV_SIP_200OK
gc_SipAck
ACK
GCEV_SIP_ACK_OK
GCEV_ANSWERED
GCEV_SIP_ACK
gc_ReqModifyCall SDP Offer 2
Per RFC 3264, a stream that is rejected must not media stream packets It is assumed that the original stream remains intact because the modification was rejected.
re-INVITE SDP Offer 2
SIP ACK is automatically generated by Global Call for INVITE and re-INVITE failures.
GCEV_REQ _MODIFY_CALL SDP Offer 2
gc_RejectModifyCall
415 - Media Not Supported
GCEV_MODIFY _CALL_REJ
ACK
GCEV_REJECT _MODIFY_CALL
GCEV_SIP_ACK

5.4.5 Cancelling a re-INVITE Request

Figure 76 shows the host application behavior when a re-INVITE request has been cancelled. The re-INVITE request is initially made to modify the coder used by the active dialog:

Figure 76. Cancelling a Coder Switch using re-INVITE

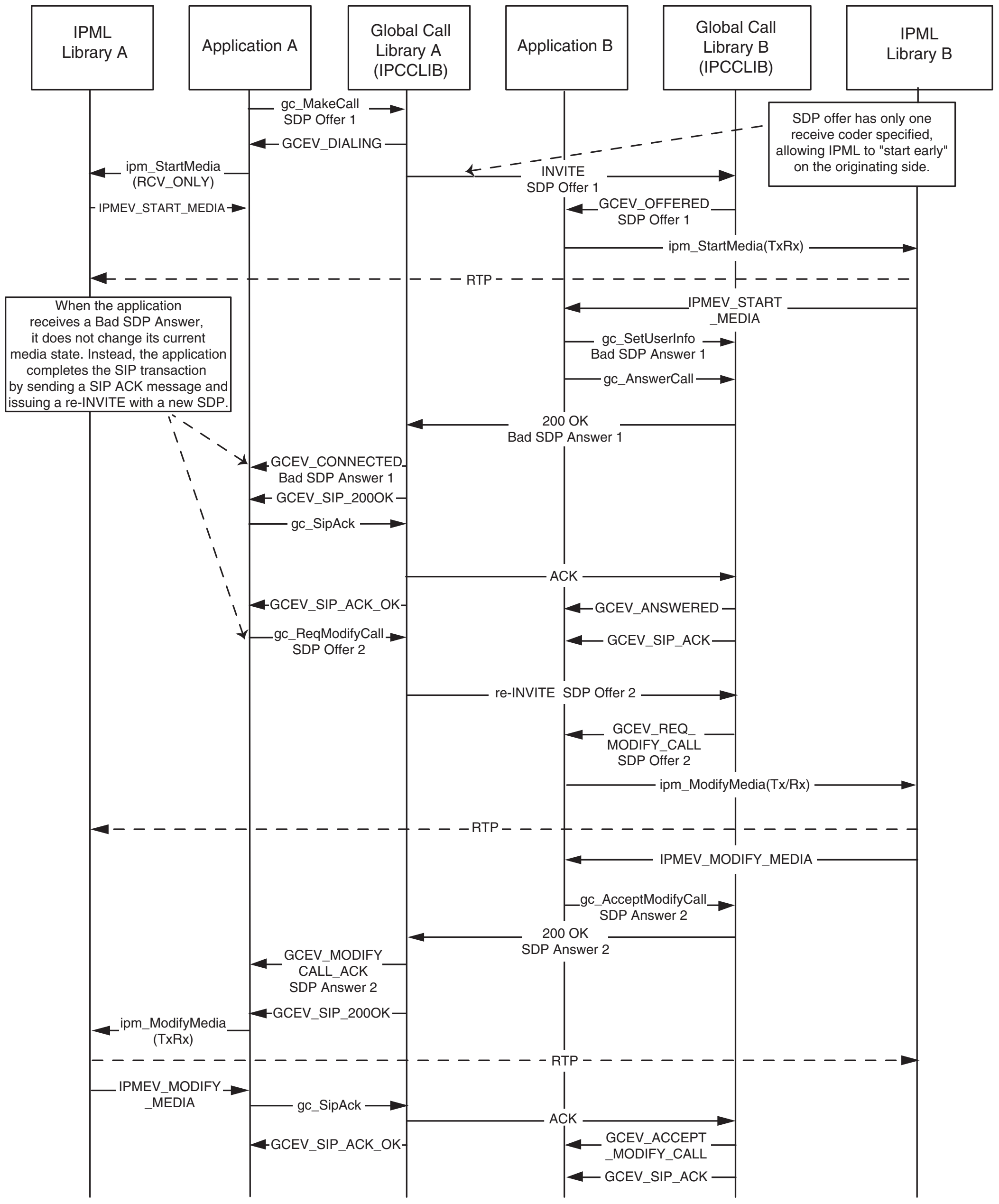

IPML Library A
Application A
Global Call Library A (IPCCLIB)
Application B
Global Call Library B (IPCCLIB)
IPML Library B
gc_MakeCall SDP Offer 1
GCEV_DIALING
ipm_StartMedia (RCV_ONLY)
INVITE SDP Offer 1
IPMEV_START_MEDIA
GCEV_OFFERED SDP Offer 1
gc_SetUserInfo SDP Answer 1
ipm_StartMedia(TxRx)
RTP
gc_AnswerCall
200 OK SDP Answer 1
GCEV_CONNECTED SDP Answer 1
GCEV_SIP_200OK
ipm_ModifyMedia (TxRx)
RTP
IPMEV_MODIFY _MEDIA
gc_SipAck
ACK
GCEV_SIP_ACK_OK
GCEV_ANSWERED
GCEV_SIP_ACK
SIP ACK is automatically generated by Global Call for INVITE and re-INVITE failures. If these ACKs require "SDP reject answers", Global Call will provide them.
gc_ReqModifyCall SDP Offer 2
re-INVITE SDP Offer 2
GCEV_REQ_ MODIFY_CALL SDP Offer 2
gc_ReqModifyCall IP_MSGTYPE_ SIP_CANCEL
CANCEL
Per RFC3264, a stream that is rejected in the answer must not send media for that stream. It is assumed that the original stream remained intact becuase the modification was rejected. In 400-600 responses to CANCEL requests that require SDP, Global Call will generate the SDP.
200 OK CSeq number CANCEL
487 - Request Terminated CSeq number re-INVITE
ACK
GCEV_MODIFY _CALL_CANCEL
GCEV_CANCEL _MODIFY_CALL

5.4.6 Receiving an Invalid Answer SDP

Figure 77 depicts host application behavior when an invalid answer SDP is received. In this message sequence diagram, the application is responsible for the following:

- determining that the received answer SDP is invalid
- providing a response to satisfy the offer/answer protocol requirements
- calling the **gc_ReqModifyCall( )** function to initiate a new offer/answer transaction

Figure 77. Receiving an Invalid Answer SDP


IPML Library A
Application A
Global Call Library A (IPCCLIB)
Application B
Global Call Library B (IPCCLIB)
IPML Library B
gc_MakeCall SDP Offer 1
GCEV_DIALING
SDP offer has only one receive coder specified, allowing IPML to "start early" on the originating side.
ipm_StartMedia (RCV_ONLY)
INVITE SDP Offer 1
IPMEV_START_MEDIA
GCEV_OFFERED SDP Offer 1
ipm_StartMedia(TxRx)
RTP
IPMEV_START _MEDIA
When the application receives a Bad SDP Answer, it does not change its current media state. Instead, the application completes the SIP transaction by sending a SIP ACK message and issuing a re-INVITE with a new SDP.
gc_SetUserInfo Bad SDP Answer 1
gc_AnswerCall
200 OK Bad SDP Answer 1
GCEV_CONNECTED Bad SDP Answer 1
GCEV_SIP_200OK
gc_SipAck
ACK
GCEV_SIP_ACK_OK
GCEV_ANSWERED
gc_ReqModifyCall SDP Offer 2
GCEV_SIP_ACK
re-INVITE SDP Offer 2
GCEV_REQ_ MODIFY_CALL SDP Offer 2
ipm_ModifyMedia(Tx/Rx)
RTP
IPMEV_MODIFY_MEDIA
gc_AcceptModifyCall SDP Answer 2
200 OK SDP Answer 2
GCEV_MODIFY CALL_ACK SDP Answer 2
GCEV_SIP_200OK
ipm_ModifyMedia (TxRx)
RTP
IPMEV_MODIFY _MEDIA
gc_SipAck
ACK
GCEV_ACCEPT _MODIFY_CALL
GCEV_SIP_ACK_OK
GCEV_SIP_ACK

5.4.7 OPTIONS Request on an Active Dialog

This section provides message sequence diagrams for the following two scenarios:

- OPTIONS Request Without a MIME Body
- With a MIME Body

5.4.7.1 OPTIONS Request Without a MIME Body

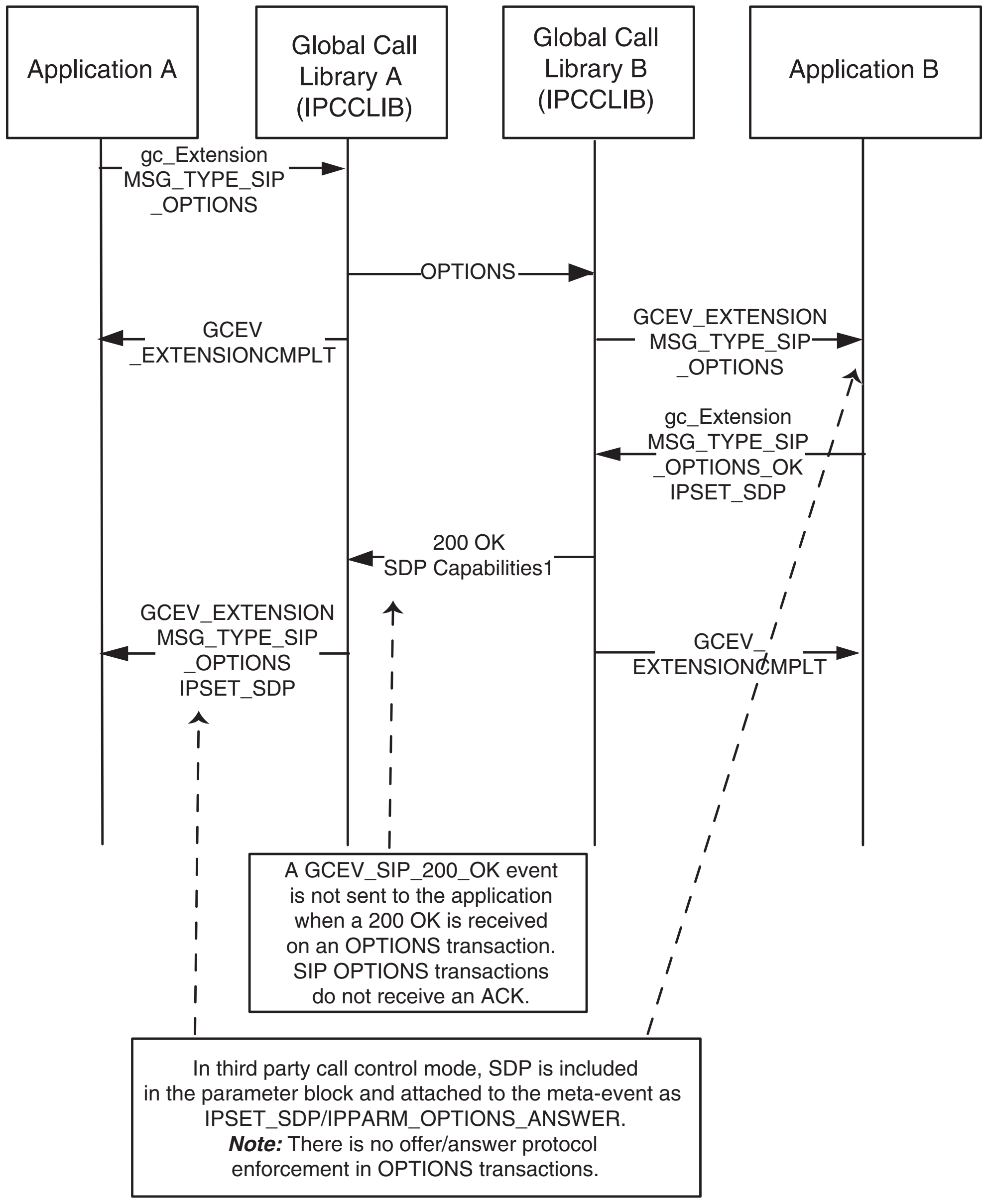
When performing SIP OPTIONS transactions in 3PCC mode, SDP content is always delivered in a parameter block attached to the Global Call event as an IPSET_SDP / IPPARAM_OPTION_ANSWER or IPSET_SDP / IPPARM_OPTION_OFFER parameter element regardless of MIME body inclusion. This is consistent with connection establishment SDP behavior in this mode, except that there is no offer/answer protocol enforcement in OPTIONS transactions. Offer/answer protocol enforcement applies to media connection establishment transactions.

When performing SIP OPTIONS transitions in 1PCC mode, the SDP information is delivered to the application in the parameter block attached to the Global Call event as an IPSET_MIME / IPPARM_MIME_PART_BODY parameter element regardless of MIME body inclusion.

Refer to Section 4.16, “Sending and Receiving SIP OPTIONS Messages”, on page 239 for more information about enabling OPTIONS access

Figure 78 depicts the host application behavior when performing an OPTIONS request on an active dialog. A MIME body is not attached in Figure 78:

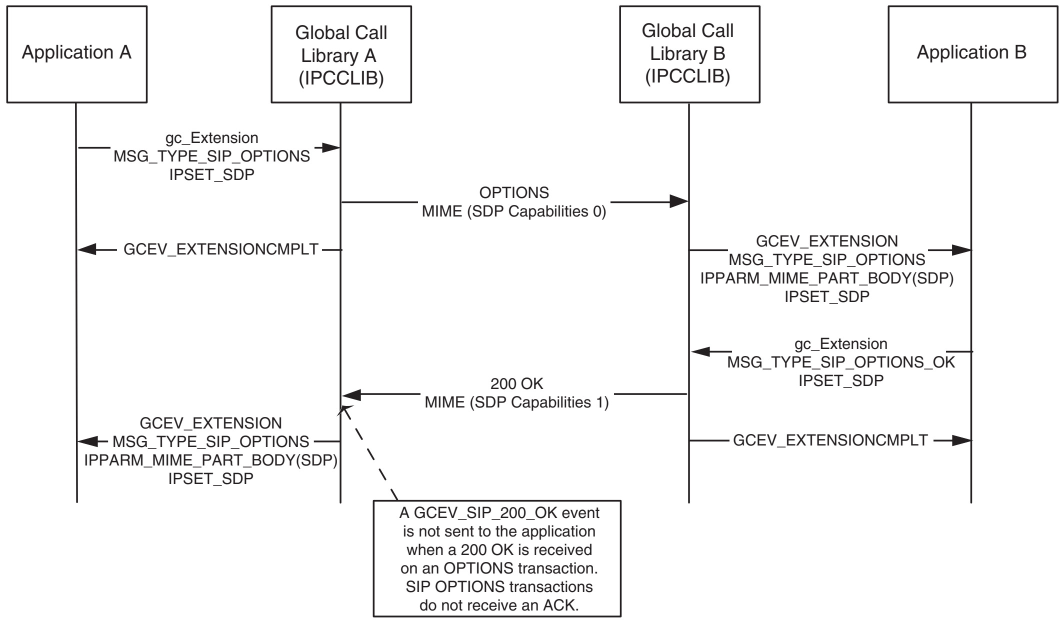
Figure 78. OPTIONS Request without a MIME Body


Application A
Global Call Library A (IPCCLIB)
Global Call Library B (IPCCLIB)
Application B
gc_Extension MSG_TYPE_SIP _OPTIONS
OPTIONS
GCEV _EXTENSIONCMPLT
GCEV_EXTENSION MSG_TYPE_SIP _OPTIONS
gc_Extension MSG_TYPE_SIP _OPTIONS_OK IPSET_SDP
200 OK SDP Capabilities1
GCEV_EXTENSION MSG_TYPE_SIP _OPTIONS IPSET_SDP
GCEV_ EXTENSIONCMPLT
A GCEV_SIP_200_OK event is not sent to the application when a 200 OK is received on an OPTIONS transaction. SIP OPTIONS transactions do not receive an ACK.
In third party call control mode, SDP is included in the parameter block and attached to the meta-event as IPSET_SDP/IPPARM_OPTIONS_ANSWER.
Note: There is no offer/answer protocol enforcement in OPTIONS transactions.

5.4.7.2 With a MIME Body

Figure 79 depicts the host application behavior when performing an OPTIONS request on an active dialog with the OPTIONS access enabled and a MIME body attached:

Figure 79. OPTIONS Request with a MIME Body



Application A
Global Call Library A (IPCCLIB)
Global Call Library B (IPCCLIB)
Application B
gc_Extension MSG_TYPE_SIP_OPTIONS IPSET_SDP
OPTIONS MIME (SDP Capabilities 0)
GCEV_EXTENSIONCMPLT
GCEV_EXTENSION MSG_TYPE_SIP_OPTIONS IPPARM_MIME_PART_BODY(SDP) IPSET_SDP
gc_Extension MSG_TYPE_SIP_OPTIONS_OK IPSET_SDP
200 OK MIME (SDP Capabilities 1)
GCEV_EXTENSION MSG_TYPE_SIP_OPTIONS IPPARM_MIME_PART_BODY(SDP) IPSET_SDP
GCEV_EXTENSIONCMPLT
A GCEV_SIP_200_OK event is not sent to the application when a 200 OK is received on an OPTIONS transaction. SIP OPTIONS transactions do not receive an ACK.

5.5 Processing Intraframe Requests for Video Streams

This section provides information about processing intraframe requests for video streams. Topics are as follows:

- Overview
- Requesting an I-Frame in SIP
- Global Call Example Code

5.5.1 Overview

When each frame of video is compressed separately, the type of compression is known as “intraframe” or “spatial” compression. Intraframes, also known as I-frames (or I-Pictures), are complete and independent. Video compression systems, however, typically utilize what is known as “inter-frame” or “temporal” compression as well. Inter-frame compression takes advantage of the fact that any given frame of video is most likely similar to the frames around it. So, instead of storing entire frames, the coder implementation can store just the differences between certain frames, reducing the amount of overall data that needs to be stored or transmitted. I-frames provide a reference point for dependent Inter-frames (P pictures and B pictures) and allow random access into the compressed video stream. When recording a video message, the first video frame stored should ideally be an I-frame. This will insure that the playback will be begin with a visually complete and recognizable frame.

5.5.2 Requesting an I-Frame in SIP

SIP provides a way to request the transmission of an Intraframe as defined by the “expired” IETF draft *draft-levin-mmusic-xml-schema-media-control-03*. The draft specification defines an XML schema (shown below) that is attached as a MIME body to a SIP INFO message:

```
<?xml version="1.0" encoding="utf-8" ?>
<media_control>
   <vc_primitive>
      <to_encoder>
   <picture_fast_update>
   </picture_fast_update>
      </to_encoder>
   </vc_primitive>
</media_control>
```

The receiving SIP entity re-transmits an Intraframe and acknowledges the INFO message with a 200 OK message. This process is referred to as video picture fast-update.

5.5.3 Global Call Example Code

Note: The Dialogic® Global Call API does not support sending of single-part MIME bodies in this release. The following code example shows sending of a multi-part MIME body.

```
////////////////////////////////////////////////////////////////////////////////
bool CMMStream::SendIFrameRequest()
{
   agwReport(INFO_MSG, s_eType, "SendIFrameRequest()");
   GC_PARM_BLKP    gcParmBlk_mime = 0;
   GC_PARM_BLKP    gcParmBlk_mime1 = 0;
   GC_PARM_BLKP    gcParmBlk_info = 0;
   bool bOk = true;

   // specify the body type
   char *pBodyType = "Content-Type:application/media_control+xml";
   if (gc_util_insert_parm_ref(&gcParmBlk_mime,
                            IPSET_MIME,
                            IPPARM_MIME_PART_TYPE,
                            (unsigned char)
                            (strlen(pBodyType) + 1),
                            pBodyType) < 0)
   {
      agwReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_util_insert_parm_ref()
               failed on %s for IPPARM_MIME_PART_TYPE ", m_devName);
      bOk = false;
   }

   // insert the body size
   if (gc_util_insert_parm_val(&gcParmBlk_mime,
                            IPSET_MIME,
                            IPPARM_MIME_PART_BODY_SIZE,
                            sizeof(unsigned long),
                            strlen(c_iFrameRequest)) < 0)
   {
      agwReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_util_insert_parm_val()
               failed on %s for IPPARM_MIME_PART_BODY_SIZE ", m_devName);
      bOk = false;
   }
   // insert the body
   if (gc_util_insert_parm_val(&gcParmBlk_mime,
                               IPSET_MIME,
                               IPPARM_MIME_PART_BODY,
                               sizeof(unsigned long),
                               (unsigned long)(c_iFrameRequest)) < 0)
   {
      agwReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_util_insert_parm_val()
                failed on %s for IPPARM_MIME_PART_BODY ", m_devName);
      bOk = false;
   }

   // insert the list of parmBlks into the top level parmBlk
   if (gc_util_insert_parm_val(&gcParmBlk_mime1,
                               IPSET_MIME,
                               IPPARM_MIME_PART,
                               sizeof(unsigned long),
                               (unsigned long)gcParmBlk_mime) < 0)
   {
      agwReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_util_insert_parm_val()
                failed on %s for IPPARM_MIME_PART", m_devName);
      bOk = false;
   }

   // now set it on the device
   if (gc_SetUserInfo(GCTGT_GCLIB_CRN,
                      m_gcCurrentCrn,
                      gcParmBlk_mime1,
                      GC_SINGLECALL) < 0) // for this call only
   {
      agwReport(ERROR_GCALL, s_eType, "gc_SetUserInfo() failed on %s for MIME body in INFO");
      bOk = false;
   }

   // insert the message type
   if (gc_util_insert_parm_val(&gcParmBlk_info,
                               IPSET_MSG_SIP,
                               IPPARM_MSGTYPE,
                               sizeof(int),
                               IP_MSGTYPE_SIP_INFO) < 0)
   {
      agwReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_util_insert_parm_val()
                failed on %s for SIP INFO", m_devName);
      bOk = false;
   }

   if (gc_Extension(GCTGT_GCLIB_CRN,
                    m_gcCurrentCrn,
                    IPEXTID_SENDMSG,
                    gcParmBlk_info,
                    NULL,
                    EV_ASYNC) < 0)
   {
      agwReport(ERROR_GCALL, s_eType, "SendIFrameRequest() -> gc_Extension failed");
      bOk = false;
   }

   gc_util_delete_parm_blk(gcParmBlk_info);
   gc_util_delete_parm_blk(gcParmBlk_mime);

   return bOk;
}
```

Building Dialogic® Global Call API IP Applications 6

This chapter describes the IP-specific header files and libraries required when building applications.

Note: For more information about building applications, see the *Dialogic® Global Call API Programming Guide*.

6.1 Header Files

When compiling Dialogic® Global Call API applications for the IP technology, it is necessary to include the following header files in addition to the standard Dialogic® Global Call API header files, which are listed in the *Dialogic® Global Call API Library Reference* and *Dialogic® Global Call API Programming Guide*:

gcip.h
: IP-specific data structures

gcip_defs.h
: IP-specific type definitions, error codes and IP-specific parameter set IDs and parameter IDs

gccfgparm.h
: Dialogic® Global Call API type definitions, configurable parameters in the Dialogic® Global Call API library and generic parameter set IDs and parameter IDs

gcipmlib.h
: for Quality of Service (QoS) features

6.2 Required Libraries

When building Dialogic® Global Call API applications for the IP technology, it is not necessary to link any libraries other than the standard Dialogic® Global Call API library, *libgc.lib*. Other libraries, including IP-specific libraries, are loaded automatically.

6.3 Required System Software

The Dialogic® Host Media Processing (HMP) Software must be installed on the development system. See the Dialogic® Software Installation Guide for your Dialogic® HMP Software release for further information.

Debugging Dialogic® Global Call API IP Applications 7

This chapter provides information about debugging Dialogic® Global Call API IP applications:

7.1 Debugging Overview

The Dialogic® Global Call API IP Call Control Library uses the RTF (Runtime Tracing Facility) system that is used by other Dialogic® Software libraries to write underlying call control library and stack information to a consolidated log file while an application is running. This information can help trace the sequence of events and identify the source of a problem. This information is also useful when reporting problems to technical support personnel.

All libraries and software modules that use RTF write their messages to a single, consolidated log file, with the default name *rftlog.txt*. The log file may optionally have a date and time stamp appended to the filename; for example, *rtflog01052005-13h24m19.923s*. When compared to the multiple independent log files used in previous implementations of the IP Call Control library, the consolidated log file has the advantage of clearly showing the time relationship of events associated with different software modules without requiring developers to correlate event time stamps.

Note: The SIP stack may also generate its own log file named *sdplog.txt* to capture any parsing errors that may occur.

The RTF facility allows developers to configure which events are written to the log file based on the importance of the event and the specific software module generating the event. All logging configuration for all libraries and modules that use RTF (not just the IP Call Control Library) is contained in a single, consolidated configuration file. This is in contrast to previous Global Call IP library implementations which used multiple configuration files for the library and the two IP protocol stacks.

The RTF facility uses the following entities to control which debug print statements are written to the log file:

module

An RTF module corresponds to a library or software module that has internal RTF APIs incorporated into its source code. Three separate RTF modules are used by the IP Call Control library:

- gc_h3r – call control, signal handler, and signal adaptation layer software modules
- sip_stack – SIP protocol stack
- h323_stack – H.323 protocol stack

client
: An entity for identifying a device, component, or function that is to be traced by the RTF. The RTF modules for the IP Call Control library include a large number of client entities to provide a high degree of control over what statements are written to the log file; these clients are listed in the following sections which describe how to configure the logging facility.

label
: An attribute associated with a trace statement to categorize the type or level of the information and to determine whether the statement is written to the log file. Labels are handled as independent entities and must be enabled or disabled individually; this is in contrast to the previous IP Call Control library logging implementation, where it was possible to enable log output for multiple statement levels collectively. Different RTF modules use different subsets of the overall RTF label set; the labels used for the IP Call Control library include only Error, Warning, and Debug.

7.2 Configuring the Logging Facility

The following topics provide information about how the user can customize the information written into the log file by the Global Call IP library:

- Configuration File Overview
- Configuring the gc_h3r Logging Module
- Configuring SIP Stack Logging
- Configuring H.323 Stack Logging

7.2.1 Configuration File Overview

This section describes how the common RTF configuration file is organized and what configuration is set up in the default configuration file that is supplied with the release software. The default configuration file may be named *RtfConfig.xml* or it may have an OS-specific name as appropriate to the specific release (i.e., *RtfConfigWin.xml* or *RtfConfigLinux.xml*); for simplicity, this document will only refer to the generic name. The entries in this configuration file conform to XML syntax rules.

Global Section

The global section of the *RtfConfig.xml* file contains one or more “GLabel” elements, which are used to globally enable logging of trace statements that are mapped to that RTF label. Globally enabling or disabling a label affects all RTF modules, but the global setting may be overridden locally.

The default *RtfConfig.xml* file globally enables the Error label, so that all error statements from all RTF modules will be logged unless disabled locally. The statement that globally enables the Error label is:

```
        <GLabel name="Error" state="1"/>
```

Module Sections

The *RtfConfig.xml* file contains a number of module sections, each of which controls the logging of trace statements for a specific RTF module. Three RTF modules apply to the IP Call Control library: gc_h3r, h323_stack, and sip_stack.

Each module section begins with a <Module> tag (with name and state attributes) and ends with a </Module> tag. Between these two tags, the configuration file contains one or more "MLabel" elements to locally enable or disable logging of the RTF labels that are used by the specific module. The behavior of the "MLabel" elements for each of the RTF modules for the IP Call Control library are described in the following sections of this chapter.

Client Entries

In addition to "MLabel" elements, a module section may also contain a number of "MClient" elements for any clients that are defined within the module. Each of the three of the RTF modules for the IP Call Control library include a number of MClient elements, as described in the following sections of this chapter.

7.2.2 Configuring the gc_h3r Logging Module

The gc_h3r module controls logging of error and debug statements that related to the call control, signal handling, and signal adaptation layer software modules of the IP Call Control library. These statements were logged to the *gc_h3r.log* file in previous implementations.

The RTF gc_h3r module supports three user-maskable RTF labels: Error, Warning, and Debug. This is in contrast to the previous non-RTF implementation of the GC_H3R module, which used six debug levels. The old levels are mapped to the new labels as follows:

RTF Label (and default state)	Old GC_H3R Debug Levels
Error (globally enabled)	LEVEL_ERROR
Warning (locally enabled)	LEVEL_WARNING
Debug (locally disabled)	LEVEL_INFO, LEVEL_INFO_EXT, LEVEL_ALL

In addition to the five GC_H3R debug levels that are mapped to RTF labels, there is an additional level, LEVEL_SPECIAL, which is not mapped to an RTF label and is therefore non-maskable. Statements marked with LEVEL_SPECIAL are always printed to the log file.

The Error label is normally enabled globally. The Warning label is normally enabled locally, on the module level. The Debug label is enabled and disabled on the module level, and if the label is enabled the logging of these statements is controllable on an individual client basis.

The cg_h3r module in the *RtfConfig.xml* file begins with the statement:

```
<Module name="gc_h3r" state="1">
```

Following this statement are "MLabel" statements to set the local state of the Warning and Debug labels. In the default *RtfConfig.xml* file, the Warning label is enabled (state="1") and the Debug label is disabled (state="0").

```
        <MLabel name="Warning" state="1"/>
        <MLabel name="Debug" state="0"/>
```

In the gc_h3r module, the "MLabel" statement for the Warning label enables or disables the logging of all statements from the gc_h3r module that have LEVEL_WARNING in them regardless of the state settings of the "MClient" elements. The "MLabel" statement for the Debug label, on the other hand, interacts with the state settings of the "MClient" elements. Setting the state of the Debug label to "0" disables all statements containing LEVEL_INFO, LEVEL_INFO_EXT, or LEVEL_ALL, regardless of the MClient states. But setting the state of the Debug label to "1" only enables these statements for software modules that have their client state to "1". By enabling only the client modules are of interest in a given debug process, users can avoid the very large output that would result if all low-level statements from all gc_h3r software modules are logged.

Note: Enabling the Debug label while all of the gc_h3r clients are set to the enabled state may produce a very large log file and may cause significant loading of the CPU.

The "MClient" statements for each software module in the gc_h3r module follow the "MLabel" statements in the *RtfConfig.xml* file. The "MClient" statements are divided into four groups which correspond to four functional groups covered by this logging module. The prefixes of the client names also reflect this four-part grouping. A typical "MClient" statement looks like the following:

```
        <MClient name="SH_CRN" state="1"/>
```

The following list gives the names and basic descriptions of the RTF clients in the GC_H3R module along with the corresponding module names that were used in the previous, non-RTF implementation of GC_H3R logging.

SH_CRN (formerly M_CRN)
: Sharon Call Reference Number

SH_MGR (formerly M_SH_MAN)
: Sharon Manager

SH_LD (formerly M_LD)
: Sharon Line Device

SH_MEDIA (formerly M_MEDIA)
: Sharon Media

SH_PDL (formerly M_PDL)
: Sharon Platform Dependent Layer

SH_PACKER (formerly M_PACKER)
: Sharon Packer

SH_DBASE (formerly M_SH_DB)
: Sharon Database

SH_DECODER (formerly M_SH_DEC)
: Sharon Decoder

SH_ENCODER (formerly M_SH_ENC)
: Sharon Encoder

SH_IPC (formerly M_SH_IPC)
: Sharon Inter-Process Communication

SH_UNPACK (formerly M_SH_UNPACK)
: Sharon Unpacker

SH_BOARD (formerly M_BOARD)
: Sharon Board Device.

SH_MONITOR (formerly M-MON)
: Sharon Manager (host LAN monitor)

H323_SIG_MGR (formerly M_SIG_MAN)
: H.323 Signal Adaptation Layer (Sigal) Manager

H323_CALL_MGR (formerly M_CALL_MAN)
: H.323 Call Manager

H323_SIGNAL (formerly M_SIGNAL)
: H.323 Signaling

H323_CONTROL (formerly M_CONTROL)
: H.323 Control

H323_CH_MGR (formerly M_CHAN_MAN)
: H.323 Channel Manager

H323_CHANNEL (formerly M_CHAN)
: H.323 Channel

H323_IE (formerly M_IE)
: H.323 Information Elements

H323_SIG_DEC (formerly M_SIG_DEC)
: H.323 Signal Adaptation Layer Decoder

H323_SIG_ENC (formerly M_SIG_ENC)
: H.323 Signal Adaptation Layer Encoder

H323_SIG_IPC (formerly M_SIG_IPC)
: H.323 Inter-Process Communication

H323_RAS (formerly M_RAS)
: H.323 Registration and Administration

H323_CAPS (formerly M_CAPS)
: H.323 Capabilities

SIP_SIGAL (formerly M_S_SIGAL)
: SIP Signal Adaptation Layer (Sigal)

SIP_SALL_MGR (formerly M_S_CALLM)
: SIP Call Manager

SIP_SIGNAL (formerly M_S_SIGNL)
: SIP Signaling

SIP_CH_MGR (formerly M_S_CHMGR)
: SIP Channel Manager

SIP_IE (formerly M_SIP_IE)
: SIP Information Elements

SIP_CAPS (formerly M_SIP_CAP)
: SIP Capabilities

SIP_SIG_DEC (formerly M_SIP_DEC)
: SIP Signal Adaptation Layer Decoder

SIP_SIG_ENC (formerly M_SIP_ENC)
: SIP Signal Adaptation Layer Encoder

SIP_IPC (formerly M_SIP_IPC)
: Inter-Process Communication

SIP_INFO (formerly M_INFO)
: SIP Information

SIP_REFER (formerly M_REFER)
: SIP Refer

SIP_PRACK (formerly M_PRACK)
: SIP Protocol Acknowledgement

SIP_AUTHENT (formerly M_AUTHENT)
: SIP Authenticator

SIP_SUBSYS (formerly M_S_SUBSM)
: SIP Subsystem

COM_MEMMGR (formerly M_MEMMGR)
: Common Memory Manager

COM_MIME (formerly M_MIME)
: Common Mime

COM_R_MGR (formerly M_R_MGR)
: Common "R" Manager

COM_MR_MGR (formerly M_MR_MGR)
: Common "MR" Manager

7.2.3 Configuring SIP Stack Logging

The sip_stack RTF module controls logging of debug statements that relate to the SIP protocol stack used by the IP Call control library. In previous implementations, this logging was configured via the *gc_h3r.cfg* file and the statements were logged to the file *gc_h3r.log*.

Note: The SIP stack may also generate its own log file named *sdplog.txt* to capture any parsing errors that occur.

The sip_stack module supports two user-maskable RTF labels: Error and Debug. This is in contrast to the previous non-RTF implementation of the GC_H3R module, which used five bit-encoded debug levels. The old levels are mapped to the new labels as follows:

RTF Label (and default state)	Old SIP Debug Levels in GC_H3R
Error (globally enabled)	EXCEP, ERROR, WARN
Debug (locally disabled)	INFO, DEBUG

The Error label is normally enabled globally. The Debug label is enabled and disabled on the module level, and if the label is enabled the logging of these statements is controllable on an individual client basis. The state of the Warning label has no effect on the sip_stack module.

The sip_stack module in the *RtfConfig.xml* file begins with the statement:

```
<Module name="sip_stack" state="1">
```

Following this statement is an "MLabel" statement to set the local state of the Debug label, which is disabled (state="0") in the default *RtfConfig.xml* file:

```
<MLabel name="Debug" state="0"/>
```

The "MLabel" statement for the Debug label interacts with the state settings of the "MClient" elements to enable or disable logging from the individual software modules of the SIP protocol stack. Setting the state of the Debug label to "0" disables all debug statements from the SIP stack, regardless of the states of the individual RTF clients. Setting the state of the Debug label to "1" enables logging of debug statements for any stack modules that have their RTF client state to "1".

Note: Enabling the Debug label while all of the sip_stack clients are set to the enabled state may produce a very large log file and may cause significant loading of the CPU.

The "MClient" statements for each software module in the sip_stack module follow the "MLabel" statement in the *RtfConfig.xml* file. A typical "MClient" statement in the *RtfConfig.xml* file looks like the following, which enables logging for the MESSAGE client if the Debug label is enabled:

```
<MClient name="MESSAGE" state="1"/>
```

The names of the RTF clients in the sip_stack module (along with the module names used in the previous GC_H3R logging implementation) include the following:

- MESSAGE (formerly RvSipStack_Message)
- TRANSPORT (formerly RvSipStack_Transport)
- TRANSACTION (formerly RvSipStack_Transaction)
- CALL (formerly RvSipStack_Call)
- PARSER (formerly RvSipStack_Parser)
- STACK (formerly RvSipStack_Stack)
- MSG BUILDER (formerly RvSipStack_MsgBuilder)
- AUTHENTICATOR (formerly RvSipStack_Authenticator)
- REG CLIENT (formerly RvSipStack_RegClient)
- SUBSCRIPTION

7.2.4 Configuring H.323 Stack Logging

The "h323_stack" RTF module controls logging of debug statements that relate to the H.323 protocol stack used by the IP Call control library. In previous implementations, this logging was configured via the *rvtele.ini* file and the statements were logged to the file *rvtsp1.log*.

The h323_stack RTF module uses a single label, namely Debug. The states of the Error and Warning labels have no effect on the h323_stack module.

The h323_stack module in the *RtfConfig.xml* file begins with the statement:

```
<Module name="h323_stack" state="1">
```

Following this statement is an "MLabel" statement to set the local state of the Debug label, which is disabled (state="0") in the default *RtfConfig.xml* file:

```
<MLabel name="Debug" state="0"/>
```

The "MLabel" statement for the Debug label interacts with the state settings of the "MClient" elements to enable or disable logging from the individual software modules of the H.323 protocol stack. Setting the state of the Debug label to "0" disables all debug statements from the H.323 stack, regardless of the states of the individual RTF clients. Setting the state of the Debug label to "1" enables logging of debug statements for any stack modules that have their RTF client state to "1".

Note: Enabling the Debug label while all of the h323_stack clients are set to the enabled state may produce a huge log file and may cause heavy loading of the CPU.

The "MClient" statements for each software module in the h323_stack module follow the "MLabel" statement in the *RtfConfig.xml* file. A typical "MClient" statement in the *RtfConfig.xml* file looks like the following, which enables logging for the EMA stack module if the Debug label is also enabled:

```
<MClient name="EMA" state="1"/>
```

The names of the RTF clients in the h323_stack module include the following (the † symbol marks the clients that are most commonly used in debugging):

- EMA
- MEMORY
- RA
- CAT
- CM †
- CMAPI †
- CMAPICB †
- CMERR †
- TPKTCHAN †
- CONFIG †
- APPL
- FASTSTART †
- VT
- UNREG
- RAS †
- UDPCHAN
- TCP
- TRANSPORT
- ETIMER

- PER †
- PERERR †
- Q931†
- Q931ERR
- LI
- TIMER
- ANNEXE
- SSEERR
- SSEAPI
- SSEAPICB
- SUPS
- SSCHAN

IP-Specific Function Information 8

Certain Dialogic® Global Call API functions have additional functionality or perform differently when used with IP technology. The generic function descriptions in the *Dialogic® Global Call API Library Reference* do not contain detailed information for any specific technology. Detailed information in terms of the additional functionality or the difference in performance of those functions when used with IP technology is contained in this chapter. The information provided in this guide therefore must be used in conjunction with the information presented in the *Dialogic® Global Call API Library Reference* to obtain the complete information when developing Dialogic® Global Call API applications that use IP technology. IP-specific variances are described in the following topics:

8.1 Dialogic® Global Call API Functions Supported by IP

Note: Except for **gc_Listen( )**, **gc_OpenEx( )**, **gc_ReleaseCallEx( )**, **gc_UnListen( )**, all Global Call functions that nominally support both the synchronous and asynchronous modes are supported in *asynchronous mode only* when using the IP technology.

The following is a full list of Dialogic® Global Call API functions that indicates the level of support when used with IP technology. The list indicates whether the function is supported, not supported, or supported with variances.

gc_AcceptCall( )
Supported in asynchronous mode only, with variances described in Section 8.3.1, “gc_AcceptCall( ) Variances for IP”, on page 542

gc_AcceptInitXfer( )
Supported with variances described in Section 8.3.2, “gc_AcceptInitXfer( ) Variances for IP”, on page 544

gc_AcceptModifyCall( )
IP-specific function. See page 485 for full details.

gc_AcceptXfer( )
Supported with variances described in Section 8.3.3, “gc_AcceptXfer( ) Variances for IP”, on page 545

gc_AlarmName( )
Supported

gc_AlarmNumber( )
Supported

gc_AlarmNumberToName( )
Supported

gc_AlarmSourceObjectID( )
Supported

gc_AlarmSourceObjectIDToName( )
Supported

gc_AlarmSourceObjectName( )
Supported

gc_AlarmSourceObjectNameToID( )
Supported

gc_AnswerCall( )
Supported in asynchronous mode only, with variances described in Section 8.3.4, "gc_AnswerCall( ) Variances for IP", on page 546

gc_Attach( )
Not supported

gc_AttachResource( )
Supported in asynchronous mode only in 1PCC operating mode; not supported in 3PCC operating mode

gc_BlindTransfer( )
Not supported

gc_CallAck( )
Supported in asynchronous mode only, with variances described in Section 8.3.5, "gc_CallAck( ) Variances for IP", on page 547

gc_CallProgress( )
Not supported

gc_CCLibIDToName( )
Supported

gc_CCLibNameToID( )
Supported

gc_CCLibStatus( )
Supported, but deprecated. Use **gc_CCLibStatusEx( )**.

gc_CCLibStatusAll( )
Supported, but deprecated. Use **gc_CCLibStatusEx( )**.

gc_CCLibStatusEx( )
Supported

gc_Close( )
Supported with variances described in Section 8.3.6, "gc_Close( ) Variances for IP", on page 547

gc_CompleteTransfer( )
Not supported

gc_CRN2LineDev( )
Supported

gc_Detach( )
Supported in asynchronous mode only in 1PCC operating mode; not supported in 3PCC operating mode

gc_DropCall( )
Supported in asynchronous mode only, with variances described in Section 8.3.7, "gc_DropCall( ) Variances for IP", on page 547

gc_ErrorInfo( )
Supported

gc_ErrorValue( )
Supported, but deprecated. Use **gc_ErrorInfo( )**.

gc_Extension( )
Supported in asynchronous mode only, with variances described in Section 8.3.8, "gc_Extension( ) Variances for IP", on page 548

gc_GetAlarmConfiguration( )
Supported

gc_GetAlarmFlow( )
Supported

gc_GetAlarmParm( )
Supported with variances described in Section 8.3.9, "gc_GetAlarmParm( ) Variances for IP", on page 550

gc_GetAlarmSourceObjectList( )
Supported

gc_GetAlarmSourceObjectNetworkID( )
Supported

gc_GetANI( )
Not supported

gc_GetBilling( )
Not supported

gc_GetCallInfo( )
Supported with variances described in Section 8.3.10, "gc_GetCallInfo( ) Variances for IP", on page 551

gc_GetCallProgressParm( )
Not supported

gc_GetCallState( )
Supported

gc_GetConfigData( )
Not supported

gc_GetCRN( )
Supported

gc_GetCTInfo( )
Supported with variances described in Section 8.3.11, "gc_GetCTInfo( ) Variances for IP", on page 554

gc_GetDNIS( )
Not supported

gc_GetFrame( )
Not supported

gc_GetInfoElem( )
Not supported

gc_GetLineDev( )
Supported

gc_GetLineDevState( )
Not supported

gc_GetMetaEvent( )
Supported

gc_GetMetaEventEx( )
Supported (Windows extended asynchronous programming model only)

gc_GetNetCRV( )
Not supported

gc_GetNetworkH( )
Not supported

gc_GetParm( )
Not supported

gc_GetResourceH( )
Supported with variances described in Section 8.3.12, "gc_GetResourceH( ) Variances for IP", on page 554

gc_GetSigInfo( )
Not supported

gc_GetUserInfo( )
Not supported

gc_GetUsrAttr( )
Supported

gc_GetVer( )
Supported

gc_GetVoiceH( )
Not supported

gc_GetXmitSlot( )
Supported with variances described in Section 8.3.13, "gc_GetXmitSlot( ) Variances for IP", on page 554

gc_HoldAck( )
Not supported

gc_HoldCall( )
Not supported

gc_HoldRej( )
Not supported

gc_InitXfer( )
Supported with variances described in Section 8.3.14, "gc_InitXfer( ) Variances for IP", on page 555

gc_InvokeXfer( )
Supported with variances described in Section 8.3.15, "gc_InvokeXfer( ) Variances for IP", on page 555

gc_LinedevToCCLIBID( )
Supported

gc_Listen( )
Supported with variances described in Section 8.3.16, "gc_Listen( ) Variances for IP", on page 559

gc_LoadDxParm( )
Not supported

gc_MakeCall( )
Supported in asynchronous mode only, with variances described in Section 8.3.17, "gc_MakeCall( ) Variances for IP", on page 560

gc_Open( )
Not supported

gc_OpenEx( )
Supported with variances described in Section 8.3.18, "gc_OpenEx( ) Variances for IP", on page 575

gc_QueryConfigData( )
Not supported

gc_RejectInitXfer( )
Supported with variances described in Section 8.3.19, "gc_RejectInitXfer( ) Variances for IP", on page 576

gc_RejectModifyCall( )
IP-specific function. See page 498 for full details.

gc_RejectXfer( )
Supported with variances described in Section 8.3.20, "gc_RejectXfer( ) Variances for IP", on page 577

gc_ReleaseCall( )
Not supported

gc_ReleaseCallEx( )
Supported with variances described in Section 8.3.21, "gc_ReleaseCallEx( ) Variances for IP", on page 578

gc_ReqANI( )
Not supported

gc_ReqModifyCall( )
IP-specific function. See page 506 for full details.

gc_ReqMoreInfo( )
Not supported

gc_ReqService( )
Supported in asynchronous mode only, with variances described in Section 8.3.22, “gc_ReqService( ) Variances for IP”, on page 578

gc_ResetLineDev( )
Supported in asynchronous mode only

gc_RespService( )
Supported in asynchronous mode only, with variances described in Section 8.3.23, “gc_RespService( ) Variances for IP”, on page 581

gc_ResultInfo( )
Supported

gc_ResultMsg( )
Not supported

gc_ResultValue( )
Not supported

gc_RetrieveAck( )
Not supported

gc_RetrieveCall( )
Not supported

gc_RetrieveRej( )
Not supported

gc_SendMoreInfo( )
Not supported

gc_SetAlarmConfiguration( )
Supported

gc_SetAlarmFlow( )
Supported

gc_SetAlarmNotifyAll( )
Supported

gc_SetAlarmParm( )
Supported with variances described in Section 8.3.24, “gc_SetAlarmParm( ) Variances for IP”, on page 582

gc_SetAuthenticationInfo( )
IP-specific function; see page 513 for complete information

gc_SetBilling( )
Not supported

gc_SetCallingNum( )
Not supported

gc_SetCallProgressParm( )
Not supported

gc_SetChanState( )
Not supported

gc_SetConfigData( )
Supported in asynchronous mode only, with variances described in Section 8.3.25, "gc_SetConfigData( ) Variances for IP", on page 583

gc_SetEvtMask( )
Not supported

gc_SetInfoElem( )
Not supported

gc_SetParm( )
Not supported

gc_SetupTransfer( )
Not supported

gc_SetUserInfo( )
Supported with variances described in Section 8.3.26, "gc_SetUserInfo( ) Variances for IP", on page 587

gc_SetUsrAttr( )
Supported

gc_SipAck( )
IP-specific function. Supported in 3PCC operating mode only. See page 516 for full details.

gc_SndFrame( )
Not supported

gc_SndMsg( )
Not supported

gc_Start( )
Supported with variances described in Section 8.3.27, "gc_Start( ) Variances for IP", on page 590

gc_StartTrace( )
Not supported

gc_Stop( )
Supported with variances described in Section 8.3.28, "gc_Stop( ) Variances for IP", on page 594

gc_StopTrace( )
Not supported

gc_StopTransmitAlarms( )
Not supported

gc_SwapHold( )
Not supported

gc_TransmitAlarms( )
Not supported

gc_UnListen( )
Supported with variances described in Section 8.3.29, "gc_UnListen( ) Variances for IP", on page 594

gc_util_copy_parm_blk( )
Supported function; see page 526 for full details

gc_util_delete_parm_blk( )
Supported

gc_util_find_parm( )
Supported

gc_util_find_parm_ex( )
Supported function; see page 528 for full details

gc_util_insert_parm_ref( )
Supported

gc_util_insert_parm_ref_ex( )
Supported function; see page 531 for full details

gc_util_insert_parm_val( )
Supported

gc_util_next_parm( )
Supported

gc_util_next_parm_ex( )
Supported function; see page 534 for full details

gc_WaitCall( )
Supported in asynchronous mode only

8.2 IP-Specific Dialogic® Global Call API Functions

The API reference pages in this section describe the following Dialogic® Global Call API functions that are specific to the use of IP technology:

Note: The **gc_util_..._ex( )** functions are backwards compatible with existing **gc_util_...( )** functions and may be used with any Dialogic® Global Call API technology, but IP call control is currently the only technology where these functions *must* be used to support parameter data longer than 255 bytes. The same information on the **gc_util_..._ex( )** functions is also presented in the *Dialogic® Global Call API Library Reference*.

gc_AcceptModifyCall( )

Name:	int gc_AcceptModifyCall (crn, parmblkp, mode)	
Inputs:	CRN crn	• call reference number of call targeted for modification
	GC_PARM_BLK *parmblkp	• pointer to GC_PARM_BLK which contains attributes of call which are being accepted (optional in 1PCC mode)
	unsigned long mode	• completion mode (EV_ASYNC)
Returns:	0 if successful <0 if unsuccessful	
Includes:	gclib.h	
Category:	Call Modification	
Mode:	Asynchronous only	

Description

The **gc_AcceptModifyCall( )** function is used to accept a request from the network or remote party to change one or more attributes of the current SIP dialog (call).

This function initiates a 200 OK response code to an incoming re-INVITE request (an INVITE request on an established call), which has been indicated to the application as an unsolicited GCEV_REQ_MODIFY_CALL event on the respective call object. The metaevent associated with this event references a GC_PARM_BLK that contains parameter elements which communicate the contents of the re-INVITE request to the application. The 200 OK response sent by this function indicates acceptance of the change(s) proposed in the re-INVITE request.

The changes which may be accepted by the application include:

- change in DTMF mode
- additional or changed dialog signaling attributes (SIP header fields)
- change in media session coder properties
- change in media session direction (half duplex vs. full duplex vs. suspended streaming)
- change in remote RTP address

Note: The Dialogic® Global Call API library does not provide a mechanism for requesting a change in RTP address, so requests to change the RTP address will only be received from remote endpoints that are not using Global Call.

Parameter	Description
crn	call reference number of call targeted for modification
parmblkp	pointer to GC_PARM_BLK which contains call attributes that are being accepted (optional in 1PCC operating mode)
mode	completion mode; must be EV_ASYNC

The function returns either <0 (to indicate failure) or 0 depending only upon the validity of the parameters. The function return does not indicate any status as to the success or failure of the sending of the response message. The final result of the attempt to send the response is provided in termination events.

First Party Call Control (1PCC) Mode

When an application receives a GCEV_REQ_MODIFY_CALL event, it must retrieve the parameter elements from the associated metaevent and analyze them to determine whether the proposed changes are acceptable before it calls **gc_AcceptModifyCall( )**.

In cases where one or more media sessions are present in an SDP offer within the re-INVITE, those session proposals that are supported by the given media platform are indicated as Global Call parameter elements associated with the GCEV_REQ_MODIFY_CALL event. Each proposed media type is indicated by a separate parameter within the parameter block using the following:

GCSET_CHAN_CAPABILITY
  IPPARM_LOCAL_CAPABILITY
  - value = IP_CAPABILITY structure

For a symmetrical, full-duplex media proposal, at least two such parameters—one for transmit and one for receive—are forwarded in the parameter block. For a half-duplex proposal or for an on-hold request, there may be only a single parameter element with the given set of session attributes.

In addition to being informed of the media session proposals, the application is also informed of the remote RTP transport addresses. Each RTP port that is proposed in an SDP offer received within a re-INVITE (one per "m=" line) is indicated as a separate parameter within the parameter block associated with the GCEV_REQ_MODIFY_CALL event. These remote RTP address parameters are identified as follows:

IPSET_RTP_ADDRESS
  IPPARM_REMOTE
  - value = RTP_ADDR structure

Because SDP does not communicate RTCP ports, only RTP ports are exchanged; the RTCP port will have the typical "plus one" offset from the RTP port.

To accept the changes to the dialog and media session exactly as proposed, the application calls **gc_AcceptModifyCall( )** with a NULL pointer as **parmblkp**.

An application can also formulate a specific SDP answer by inserting appropriate media session parameter elements (GCSET_CHAN_CAPABILITY / IPPARM_LOCAL_CAPABILITY) into the GC_PARM_BLK parameter block that it references in the **gc_AcceptModifyCall( )** function call. A full-duplex connection requires two such parameter elements, one for each direction. A half-duplex connection requires one parameter element with the direction field of the IP_CAPABILITY structure set appropriately. Accepting an on-hold request requires a single parameter with the proposed codec capability and one of the direction values that specifies inactive streaming.

If the capabilities to be used in the SDP answer—whether specified by the application or derived from the initial INVITE—do not match the capabilities that were contained in the SDP offer (both codec capability and direction), the library treats the situation as a rejection of the call modification request. In this case, the library sends a 488 Not Acceptable Here response to the remote party to terminate the re-INVITE, and generates a GCEV_REJECT_MODIFY_CALL event to notify the application.

Note: When accepting a codec change, the local endpoint's properties are updated immediately when the application calls this function; all outgoing packets use the new codec, and only incoming packets that use the new codec will be accepted. This may produce a perceptible artifact (for example, a click or a brief silence) until the remote endpoint receives the 200 OK response and changes its codec.

Third Party Call Control (3PCC) Mode

An incoming re-INVITE request generally contains an SDP offer that includes one or more session attributes that are different from those which were negotiated in the original INVITE dialog. A third party call control application must extract the SDP from the metaevent associated with the GCEV_REQ_MODIFY_CALL event as an IPSET_SDP/IPPARM_SDP_OFFER parameter (see Section 5.2.2.2, "IPSET_SDP Parameter Set Identifier", on page 435). The application must then parse and analyze the SDP offer to determine whether it is acceptable.

If the SDP offer is acceptable, the third party call control application must construct an appropriate SDP answer, then insert that answer into the GC_PARM_BLK referenced by **parmblkp** as an IPSET_SDP/IPPARM_SDP_ANSWER parameter element.

■ Termination Events

GCEV_ ACCEPT_MODIFY_CALL
: Successful termination event. Indicates that the 200OK response was successfully sent, and an ACK reply has been received. In 1PCC mode, this event also indicates that the requested call attribute change(s) has been performed locally.

GCEV_ACCEPT_MODIFY_CALL_FAIL
: Unsuccessful termination event. Indicates that the signaling of the modification acceptance response has failed. This could be caused by a failure in the message transport, a failure in the attempt to change the call attribute, or the expiration of a response timer for the request. The re-INVITE transaction is still in progress and the application may make another attempt to respond via a new call to **gc_AcceptModifyCall**( ) or **gc_RejectModifyCall**( ). In 1PCC mode, no modifications to the existing dialog or media session are performed and the current state remains as it was prior to the incoming modification request.

GCEV_REJECT_MODIFY_CALL
: Unsuccessful termination event. Indicates that the capabilities the application intended to signal in the SDP answer do not match any of the capabilities that were received in the SDP offer. This event implies that a 488 Not Acceptable Here final response was sent to the remote UA and that an ACK has been received, meaning that the re-INVITE dialog is terminated. In 1PCC mode, no modifications to the existing dialog or media session are performed and the current state remains as it was prior to the incoming modification request.

■ Cautions

- This function is only supported when the value IP_T38_MANUAL_MODIFY_MODE has been set for the IPSET_CONFIG / IPPARM_OPERATING_MODE parameter using the

gc_SetConfigData( ) function. If this parameter value has not been set, the function call will fail with an error value of IPERR_BAD_PARM.

- Only one modification transaction can be pending in a call at any given time. Until the pending re-INVITE has been accepted, rejected, or canceled, no additional re-INVITE can be sent by either party.
- Only one attempt to send a response to a re-INVITE request can be pending at a time. A response must fail (as indicated by a failure termination event) before a new response is attempted, otherwise the function call will fail.
- The GCEV_REQ_MODIFY_CALL event will only arrive when a call is connected. But if the call is dropped—either locally via **gc_DropCall( )** or remotely as indicated by a GCEV_DISCONNECTED event—before a response is initiated via **gc_AcceptModifyCall( )**, the request is invalid and the response can no longer be sent.
- The potential for glare situations exist with a CANCEL being received from the remote party as the local application intends to send 200OK. If the library receives the CANCEL before the **gc_AcceptModifyCall( )**, the function call fails because the re-INVITE dialog is terminated and the application receives an informational GCEV_MODIFY_CALL_CANCEL event.

■ Errors

- The function returns GC_ERROR if any of the parameters is invalid, if the call is not in the connected state, if there is no re-INVITE request pending, or if the value of the configuration parameter IPSET_CONFIG / IPPARM_OPERATING_MODE has not been set to IP_T38_MANUAL_MODIFY_MODE in 1PCC operating mode. Use the **gc_ErrorInfo( )** function to retrieve further information.
- Upon receiving a GCEV_ACCEPT_MODIFY_FAIL event, use the **gc_ResultInfo( )** function to retrieve information about the failure event. See the “Error Handling” section in the *Dialogic® Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file while IP-specific error codes are specified in *gcip_defs.h*. On failure, no modifications to the existing dialog or media session are performed and the current state remains as it was prior to the attempting the modification request.

■ Example

The following code example illustrates how the **gc_AcceptModifyCall( )** function is used in the first party call control (1PCC) operating mode.

```
.
.
.
/* Dialogic Header Files */
#include <gcip.h>
#include <gclib.h>
.
.
.

   /* SRL event handler: */
   for (;;)
   {
      if (-1 != sr_waitevt(500))   process_event();
   }
```

```
void process_event(void)
{
   METAEVENT    metaevent;
   GC_INFO      t_info;

   /* Populate the metaEvent structure */
   if(GC_SUCCESS != gc_GetMetaEvent(&metaevent))   return;

   /* process GlobalCall events */
   if ((metaevent.flags & GCME_GC_EVENT) == 0)   return;

   switch (metaevent.evttype)
   {
   .
   .
   .
      case GCEV_REQ_MODIFY_CALL:  /* request to modify call attribute */
      {
         GC_PARM_BLKP parm_blkp  = (GC_PARM_BLKP)  metaEvent.extevtdatap;
         GC_PARM_BLKP replyParmblkp = NULL;
         GC_PARM_DATAP curParm = NULL;
         IP_CAPABILITY cap;
         RTP_ADDR rtp;
         unsigned char proposal_accepted = FALSE;

         while ((curParm = gc_util_next_parm(parm_blkp, curParm)) != NULL)
         {
            if ((curParm->set_ID == GCSET_CHAN_CAPABILITY) &&
                (curParm->parm_ID == IPPARM_LOCAL_CAPABILITY))
            {
               memcpy(&cap, curParm->value_buf, curParm->value_size);
               /* determine if capability is acceptable (logic not shown) */
               if (isCapabilityAcceptable(cap) == TRUE)
               {
                  /* insert parameter with accepted capability in parameter block reply */
                  /* (logic not shown) */
                  insertCapIntoReply(cap,replyParmblkp);
                  proposal_accepted = TRUE;
               }
            }
            else if ((curParm->set_ID == IPSET_SIP_MSGINFO) &&
                     (curParm->parm_ID == IPPARM_SIP_HDR))
            {
               /* parse SIP header and make appropriate updates (logic not shown) */
               proposal_accepted = TRUE;
            }
            else if ((curParm->set_ID == IPSET_RTP_ADDRESS) &&
                     (curParm->parm_ID == IPPARM_REMOTE))
            {
               memcpy(&rtp, curParm->value_buf, curParm->value_size);
               if (isMediaReRouteAcceptable(rtp) == TRUE)
               {
                  /* update RTP transport addresses in GUI (logic not shown) */
                  updateRTPGUI(&rtp);
                  proposal_accepted = TRUE;
               }
            }
         }

         /* if proposal is acceptable accept the request                          */
         /* format accepted attributes (i.e. media types) in a parmblk (optional, */
         /* NULL if none) */
         if (proposal_accepted)
         {
            if (gc_AcceptModifyCall(crn, replyParmblkp, EV_ASYNC) < 0)
              /* failure logic here*/
         }
         else /* not acceptable so respond with SIP Client Error    */
              /* final response of 488 Not Acceptable Here          */
            if (gc_RejectModifyCall(crn,
                                    IPEC_SIPReasonStatus488NotAcceptableHere,
                                    EV_ASYNC) < 0)
              /* failure logic here */

         break;
      }

      case GCEV_ACCEPT_MODIFY_CALL:
         .
         .
         .
         /* notify user of changed attribute */
         .
         .
         .
         break;

      case GCEV_ACCEPT_MODIFY_CALL_FAIL:
         /* process failure to change attribute */
         if (gc_ResultInfo(&metaevent, &t_info) < 0)
         {
            /* failure logic here */
         }
         /* process information contained in t_info */
         /* can optionally call gc_RejectModifyCall( ) to retry */
         .
         .
         .
         break;

      case GCEV_REJECT_MODIFY_CALL:
         .
         .
         .
         /* notify user of rejected attribute */
         .
         .
         .
         break;

      case GCEV_REJECT_MODIFY_CALL_FAIL:
         /* process failure to reject request */
         if (gc_ResultInfo(&metaevent, &t_info) < 0)
         {
            /* failure logic here */
         }
         /* process information contained in t_info */
         /* can optionally call gc_RejectModifyCall( ) to retry */
         .
         .
         .
         break;
         .
         .
         .
   } /* endof switch */
} /* endof process_event function */
```

The following code example illustrates how the **gc_AcceptModifyCall( )** function is used in the third party call control (3PCC) operating mode.

```
// Assume application has enabled GCEV_200OK and GCEV_SIP_ACK eventing.
.
.
.
/* Dialogic Header Files */
#include <gcip.h>
#include <gclib.h>
.
.
.
/* SRL event handler: */
for (;;)
{
   if (-1 != sr_waitevt(500))    process_event();
}

void process_event(void)
{
   METAEVENT    metaevent;
   GC_INFO      t_info;
   /* Populate the metaEvent structure */
   if(GC_SUCCESS != gc_GetMetaEvent(&metaevent))   return;

   /* process GlobalCall events */
   if ((metaevent.flags & GCME_GC_EVENT) == 0)   return;

   switch (metaevent.evttype)
   {
   .
   .
   .
      case GCEV_REQ_MODIFY_CALL:  /* request to modify call attribute */
      {
         EXTENSIONEVTBLK *extblkp = metaevent.extevtdatap;
         GC_PARM_BLKP parm_blkp = &extblkp->parmblk;
         GC_PARM_DATA_EXT parm;
         GC_PARM_BLKP replyParmblkp = NULL;
         GC_PARM_DATAP curParm = NULL;
         IP_CAPABILITY cap;
         RTP_ADDR rtp;
         int      frc;
         bool     proposal_accepted;

         GC_PARM_BLKP parm_blkp = metaeventp->extevtdatap;
         INIT_GC_PARM_DATA_EXT(&parm);
         frc = gc_util_find_parm_ex(parm_blkp,
                                    (unsigned long)IPSET_SDP,
                                    (unsigned long)IPPARM_SDP_OFFER,
                                    &parm);

         if (frc == GC_SUCCESS)
         {
            // Raw SDP is in memory location parm.pData and is
            // of length parm.data_size.

            char sdpResponse[1000];
            int  sdpResponseSize = 1000;

            // applicationModifyMedia(...) is a user supplied function
            // that analyzes the raw SDP; it starts the media and provides
            // a raw sdp answer for the remote endpoint if the media offer
            // is acceptable. This function is not supplied in this sample.
            if (applicationModifyMedia(parm.pData, parm.data_size
                                       sdpResponse, &sdpResponseSize) == SUCCESS)
            {
```

```
            frc = gc_util_insert_parm_ref_ex(replyParmblkp,
                                             IPSET_SDP,
                                             IPPARM_SDP_ANSWER,
                                             sdpResponse,
                                             sdpResponseSize);

            if (frc != GC_SUCCESS)
            {
               // call application error handler to drop the call and log the error.
               applicationHandleError(...);
               break;
            }
            proposal_accepted = true;
         }
      }
      else
      {
         // No SDP was present in re-Invite. This is a re-Invite delayed offer.
         // This re-Invite will be rejected as this sample does not support
         // delayed offer call scenario.
         proposal_accepted = false;
      }

      /* If proposal is acceptable then accept the request.            */
      /* Format accepted attributes (i.e. raw sdp answer) in a parmblk  */
      /* (optional, NULL if none).                                     */
      if (proposal_accepted)
      {
         if (gc_AcceptModifyCall(crn, replyParmblkp, EV_ASYNC) < 0)
         {
            // Invoke the application error handler to drop the call
            applicationHandleError(...);
         }
         gc_util_delete_parm_blk(replyParmblkp);
      }
      else
      {
         /* not acceptable so respond with SIP Client Error    */
         /* final response of 488 Not Acceptable Here          */
         if (gc_RejectModifyCall(crn,
                                 IPEC_SIPReasonStatus488NotAcceptableHere,
                                 EV_ASYNC) < 0)
         {
            // Invoke the application error handler to drop the call
            applicationHandleError(...);
         }
      }
      break;
   }

   case GCEV_ACCEPT_MODIFY_CALL:
      .
      .
      .
      /* notify user of changed attribute.     */
      .
      .
      .
      break;

   case GCEV_ACCEPT_MODIFY_CALL_FAIL:
      /* process failure to change attribute */
      if (gc_ResultInfo(&metaevent, &t_info) < 0)
      {
         /* failure logic here */
      }
      /* process information contained in t_info */
      /* can optionally call gc_RejectModifyCall( ) to retry */
      .
      .
      .
      break;

   case GCEV_REJECT_MODIFY_CALL:
      .
      .
      .
      /* notify user of rejected attribute */
      .
      .
      .
      break;

   case GCEV_REJECT_MODIFY_CALL_FAIL:
      /* process failure to reject request */
      if (gc_ResultInfo(&metaevent, &t_info) < 0)
      {
         /* failure logic here */
      }
      /* process information contained in t_info */
      /* can optionally call gc_RejectModifyCall( ) to retry */
      .
      .
      .
      break;

   case GCEV_MODIFY_CALL_ACK:
      // indication that re-invite was accepted (200 ok received ) by the remote endpoint.
      // This metaevent may have an IPSET_SDP/IPPARM_SDP_OFFER or
      // IPSET_SDP/IPPARM_SDP_ANSWER attached.
      .
      .
      .
      break;

   case GCEV_SIP_200OK:
      // indication that the library needs to send a SIP ACK.
      // A parameter block containing a IPSET_SDP/IPPARM_SDP_ANSWER would be included
      // in the gc_SipAck for an outbound invite/re-invite delayed offer call scenario.
      if (gc_SipAck(crn, NULL, EV_ASYNC) != GC_SUCCESS)
      {
         // Invoke the application error handler to drop the call
         applicationHandleError(...);
      }
      break;

   case GCEV_SIP_ACK_FAIL:
      // gc_SipAck completion metaevent indicating the Sip Ack could not be sent.
      // Invoke the application error handler to drop the call.
      applicationHandleError(...);
      break;

   case GCEV_SIP_ACK_OK:
      // gc_SipAck completion metaevent indicating the Sip Ack was successfully sent.
      // All is OK.
      break;
```

```
         case GCEV_SIP_ACK:
            // Unsolicited event indicating SIP ACK was received on an invite/re-invite request.
            // This metaevent will contain an IPSET_SDP/IPPARM_SDP_ANSWER in an inbound
            // invite/re-invite delayed offer call scenario.
            .
            .
            .
            break;

            .
            .
            .
      } /* endof switch */
   } /* endof process_event function */
```

■ See Also

- **gc_RejectModifyCall( )**
- **gc_ReqModifyCall( )**

gc_CancelWaitCall( )

Name:	int gc_CancelWaitCall (linedev, mode)	
Inputs:	LINEDEV linedev	• Global Call line device handle
	unsigned long mode	• asynchronous mode
Returns:	0 if successful <0 if unsuccessful	
Includes:	gclib.h	
Category:	Basic	
Mode:	Asynchronous only	

■ Description

The **gc_CancelWaitCall( )** function cancels any previously issued **gc_WaitCall( )** and disables the ability to receive incoming calls on the given line device.

Notes: ***1.*** This function has no effect if the application did not call the **gc_WaitCall( )** function to enable notification of incoming calls. Notification is disabled by default when the channel is opened.

2. The **gc_CancelWaitCall( )** function is only supported in asynchronous mode, and only for SIP call control.

Parameter	Description
linedev	Global Call line device handle
mode	Set to EV_ASYNC for asynchronous execution.

■ Termination Events

GCEV_CANCELWAITCALL
: Indicates that the notification of incoming calls was successfully disabled.

GCEV_TASKFAIL
: Indicates that the function failed.

■ Errors

- If this function returns <0 to indicate failure, use the **gc_ErrorInfo( )** function to retrieve the reason for the error; however, if a GCEV_TASKFAIL event is generated, use the **gc_ResultInfo( )** function instead. See the "Error Handling" section in the *Dialogic® Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file, while IP-specific errors codes are specified in *gcip_defs.h*.
- If the line device is processing an incoming call, the function will either fail or generate a GCEV_TASKFAIL event. When the function fails, EGC_GLARE/ IPERR_ADDRESS_IN_USE are set as the GCcclib error codes. If a GCEV_TASKFAIL event

is generated, GCEV_CCLIBSPECIFIC /IPEC_InternalReasonIncomingCall are set as the GCcclib cause value.

Example

```
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>
#include <gcip_defs.h>

#define MAXCHAN 30   /* max. number of channels in system */
/*
 * Data structure which stores all information for each line
 */
struct linebag {
       LINEDEV ldev; /* line device handle */
       CRN crn; /* GlobalCall API call handle */
       int state; /* state of first layer state machine */
} port[MAXCHAN+1];

struct linebag *pline; /* pointer to access line device */

/*
 * Assume the following has been done:
 * 1. Open line devices for each time slot on iptB1.
 * 2. Each Line Device ID is stored in linebag structure, 'port'.
 * 3. The gc_WaitCall() has been issued on each Line Device in async mode
 * 4. No call exists on the Line Device
 */

int cancel_waitcall(int port_num)
{
       GC_INFO gc_error_info;
       /* GlobalCall error information data */
       /* Find info for this time slot, specified by 'port_num' */
       pline = port + port_num;
       /*
        * Cancel the wait call
        */
        if (gc_CancelWaitCall(pline->ldev, EV_ASYNC) != GC_SUCCESS) {
             /* process error return as shown */
             gc_ErrorInfo( &gc_error_info );
             printf ("Error: cancel_waitcall() - gc_CancelWaitCall() on device handle: 0x%lx,
                     GC ErrorValue: 0x%hx - %s,CCLibID: %i - %s, CC ErrorValue: 0x%lx -
                     %s\n", pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                     gc_error_info.ccLibId, gc_error_info.ccLibName, gc_error_info.ccValue,
                     gc_error_info.ccMsg);
             return (gc_error_info.gcValue);
       }

       return (0);
}
```

See Also

- **gc_WaitCall( )**

gc_RejectModifyCall( )

Name:	int gc_RejectModifyCall (crn, reason, mode)	
Inputs:	CRN crn	• call reference number of call targeted for modification
	unsigned long reason	• reason for rejecting request to change call attribute
	unsigned long mode	• completion mode (EV_ASYNC)
Returns:	0 if successful <0 if unsuccessful	
Includes:	gclib.h	
Category:	Call Modification	
Mode:	Asynchronous only	

■ Description

The **gc_RejectModifyCall( )** function is used to reject a request from the network or remote party to change an attribute of the current call.

This function initiates a 3xx thorough 6xx response code to an incoming re-INVITE request, as indicated by an incoming GCEV_REQ_MODIFY_CALL event on the respective call object. The actual response code that is sent is specified by the **reason** parameter.

Note: In a Global Call IP application, when the switch from a voice session to a fax session fails, the call will be disconnected and will not return back to the voice session.

Parameter	Description
crn	call reference number of the call targeted for modification; must match the CRN contained in the GCEV_REQ_MODIFY_CALL event
reason	the reason for rejecting the request to modify call attributes, specified using the IPEC_SIPReasonStatusXXX… symbolic defines for SIP reason codes from 300 through 699. These symbols are defined in *gcip_defs.h* and are listed in Section 11.5, "Failure Response Codes When Using SIP", on page 700.
mode	must be EV_ASYNC

The function returns either <0 (to indicate failure) or 0, depending only upon the validity of the parameters. The function return does not indicate any status as to the success or failure of the sending of the rejection response message. The final result of sending the response is provided to the application in termination events.

■ Termination Events

GCEV_REJECT_MODIFY_CALL
: Successful termination event. Indicates that rejection of the received re-INVITE request has completed successfully. This event implies that the specified 3xx through 6xx response was

sent and that an ACK was received from the remote party. In 1PCC mode, the requested call attribute modifications are not performed and the call state remains as it was prior to receiving the incoming re-INVITE request.

GCEV_REJECT_MODIFY_CALL_FAIL
: Unsuccessful termination event. Indicates that the signaling of the rejection response failed. The re-INVITE transaction is still in progress and the application may make another attempt to respond via a new call to **gc_AcceptModifyCall( )** or **gc_RejectModifyCall( )**. In 1PCC mode, no modifications to the existing dialog or media session are performed and the current state remains as it was prior to receiving the incoming re-INVITE request.

■ Cautions

- This function is only supported when the value of the parameter IPSET_CONFIG / IPPARM_OPERATING_MODE has been set to IP_T38_MANUAL_MODIFY_MODE using the **gc_SetConfigData( )** function. If this parameter value has not been set, the function call will fail with an error value of IPERR_BAD_PARM.
- Only one modification transaction can be pending in a call at any given time. Until the pending re-INVITE has been accepted, rejected, or canceled no additional re-INVITE can be sent by either party.
- A GCEV_REQ_MODIFY_CALL event can only arrive when a call is connected. But if the call is dropped—either locally via **gc_DropCall( )** or remotely as indicated by a GCEV_DISCONNECTED event—before a response is initiated via **gc_RejectModifyCall( )**, the request is invalid and the response can no longer be sent.
- Only one attempt to respond to a re-INVITE request can be pending at a time. A response attempt must fail (as indicated by a failure termination event) before a new response is attempted, otherwise the function call will fail.

■ Errors

- The function returns GC_ERROR if any of the parameters is invalid, if the call is not in the connected state, if there is no pending re-INVITE request, or if the value of the configuration parameter IPSET_CONFIG / IPPARM_OPERATING_MODE has not been set to IP_T38_MANUAL_MODIFY_MODE in 1PCC operating mode. Use the **gc_ErrorInfo( )** function to retrieve further information.
- Upon receiving a GCEV_REJECT_MODIFY_CALL_FAIL event, use the **gc_ResultInfo( )** function to retrieve information about the event. See the “Error Handling” section in the *Dialogic® Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file while IP-specific error codes are specified in *gcip_defs.h*. On failure, no modifications to the existing dialog or media session are performed and the current state remains as it was prior to the incoming modification request.

■ Example

The following code example illustrates how the **gc_RejectModifyCall( )** function is used in first party call control (1PCC) operating mode.

```
.
.
.
/* Dialogic Header Files */
#include <gcip.h>
#include <gclib.h>
.
.
.

   /* SRL event handler: */
   for (;;)
   {
      if (-1 != sr_waitevt(500))  process_event();
   }

void process_event(void)
{
   METAEVENT    metaevent;
   GC_INFO      t_info;

   /* Populate the metaEvent structure */
   if(GC_SUCCESS != gc_GetMetaEvent(&metaevent))  return;

   /* process GlobalCall events */
   if ((metaevent.flags & GCME_GC_EVENT) == 0)  return;
   switch (metaevent.evttype)
   {
   .
   .
   .
      case GCEV_REQ_MODIFY_CALL:  /* request to modify call attribute */
      {
         GC_PARM_BLKP parm_blkp  = (GC_PARM_BLKP)  metaEvent.extevtdatap;
         GC_PARM_BLKP replyParmblkp = NULL;
         GC_PARM_DATAP curParm = NULL;
         IP_CAPABILITY cap;
         RTP_ADDR rtp;
         unsigned char proposal_accepted = FALSE;

         while ((curParm = gc_util_next_parm(parm_blkp, curParm)) != NULL)
         {
            if ((curParm->set_ID == GCSET_CHAN_CAPABILITY) &&
                (curParm->parm_ID == IPPARM_LOCAL_CAPABILITY))
            {
               memcpy(&cap, curParm->value_buf, curParm->value_size);
               /* determine if capability is acceptable (logic not shown) */
               if (isCapabilityAcceptable(cap) == TRUE)
               {
                  /* insert parameter with accepted capability in parameter block reply */
                  /* (logic not shown) */
                  insertCapIntoReply(cap,replyParmblkp);
                  proposal_accepted = TRUE;
               }
            else if ((curParm->set_ID == IPSET_SIP_MSGINFO) &&
                     (curParm->parm_ID == IPPARM_SIP_HDR))
            {
               /* parse SIP header and make appropriate updates (logic not shown) */
               proposal_accepted = TRUE;
            }
            else if ((curParm->set_ID == IPSET_RTP_ADDRESS) &&
                     (curParm->parm_ID == IPPARM_REMOTE))
            {
               memcpy(&rtp, curParm->value_buf, curParm->value_size);
               if (isMediaReRouteAcceptable(rtp) == TRUE)
               {
                  /* update RTP transport addresses in application (logic not shown) */
                  updateRTPGUI(&rtp);
                  proposal_accepted = TRUE;
               }
            }
         }
         /* if proposal is acceptable accept the request */
         /* format accepted attributes (i.e. media types) in a parmblk (optional, */
         /* NULL if none) */
         if (proposal_accepted)
         {
            if (gc_AcceptModifyCall(crn, replyParmblkp, EV_ASYNC) < 0)
              /* failure logic here */
         }
         else /* not acceptable so respond with SIP Client Error */
              /* final response of 488 Not Acceptable Here          */
            if (gc_RejectModifyCall(crn,
                                    IPEC_SIPReasonStatus488NotAcceptableHere,
                                    EV_ASYNC) < 0)
              /* failure logic here */
         break;
      }

      case GCEV_ACCEPT_MODIFY_CALL:
        .
        .
        .
        /* notify user of changed attribute */
        .
        .
        .
        break;

      case GCEV_ACCEPT_MODIFY_CALL_FAIL:
         /* process failure to change attribute */
         if (gc_ResultInfo(&metaevent, &t_info) < 0)
            /* failure logic here */

         /* process information contained in t_info */
         /* can optionally call gc_RejectModifyCall( ) to retry */
         .
         .
         .
         break;

      case GCEV_REJECT_MODIFY_CALL:
         .
         .
         .
         /* notify user of rejected attribute */
         .
         .
         .
         break;

      case GCEV_REJECT_MODIFY_FAIL:
         /* process failure to reject request */
         if (gc_ResultInfo(&metaevent, &t_info) < 0)
            /* failure logic here */

         /* process information contained in t_info */
         /* can optionally call gc_RejectModifyCall( ) to retry */
         .
         .
         .
         break;
```

```
         .
         .
         .
   } /* endof switch */
} /* endof process_event function */
```

The following code example illustrates how the **gc_RejectModifyCall( )** function is used in third party call control (3PCC) operating mode.

```
// Assume application has enabled GCEV_200OK and GCEV_SIP_ACK eventing.
.
.
.
/* Dialogic Header Files */
#include <gcip.h>
#include <gclib.h>
.
.
.
   /* SRL event handler: */
   for (;;)
   {
      if (-1 != sr_waitevt(500))  process_event();
   }
void process_event(void)
{
   METAEVENT    metaevent;
   GC_INFO      t_info;
   /* Populate the metaEvent structure */
   if(GC_SUCCESS != gc_GetMetaEvent(&metaevent))  return;

   /* process GlobalCall events */
   if ((metaevent.flags & GCME_GC_EVENT) == 0)  return;

   switch (metaevent.evttype)
   {
   .
   .
   .
      case GCEV_REQ_MODIFY_CALL:  /* request to modify call attribute */
      {
         EXTENSIONEVTBLK *extblkp = metaevent.extevtdatap;
         GC_PARM_BLKP parm_blkp = &extblkp->parmblk;
         GC_PARM_BLKP replyParmblkp = NULL;
         GC_PARM_DATA_EXT parm;
         GC_PARM_DATAP curParm = NULL;
         IP_CAPABILITY cap;
         RTP_ADDR rtp;
         int      frc;
         bool     proposal_accepted;

         GC_PARM_BLKP parm_blkp = metaeventp->extevtdatap;
         INIT_GC_PARM_DATA_EXT(&parm);
         frc = gc_util_find_parm_ex(parm_blkp,
                                    (unsigned long)IPSET_SDP,
                                    (unsigned long)IPPARM_SDP_OFFER,
                                    &parm);

         if (frc == GC_SUCCESS)
         {
            // Raw SDP is in memory location parm.pData and is
            // of length parm.data_size.

            char sdpResponse[1000];
            int  sdpResponseSize = 1000;
            // applicationModifyMedia is a user supplied function
            // that analyzes the raw SDP; it starts the media and provides
            // a raw sdp answer for the remote endpoint if the media offer
            // is acceptable. This function is not supplied in this sample.
            if (applicationModifyMedia(parm.pData, parm.data_size
                                       sdpResponse, &sdpResponseSize) == SUCCESS)
            {
               frc = gc_util_insert_parm_ref_ex(&replyParmblkp,
                                                IPSET_SDP,
                                                IPPARM_SDP_ANSWER,
                                                sdpResponse
                                                sdpResponseSize);
               if (frc != GC_SUCCESS)
               {
                  // call application error handler to drop the call and log the error.
                  applicationHandleError(...);
                  break;
               }
               proposal_accepted = true;
            }
         }
         else
         {
            // No SDP was present in re-Invite. This is a re-Invite delayed offer.
            // This re-Invite will be rejected as this sample does not support
            // delayed offer call scenario.
            proposal_accepted = false;
         }

         /* if proposal is acceptable then accept the request.             */
         /* Format accepted attributes (i.e. raw sdp answer) in a parmblk  */
         /* (optional, NULL if none).                                      */
         if (proposal_accepted)
         {
            if (gc_AcceptModifyCall(crn, replyParmblkp, EV_ASYNC) < 0)
            {
               // Invoke the application error handler to drop the call
               applicationHandleError(...);
            }
            gc_util_delete_parm_blk(replyParmblkp);
         }
         else
         {
            /* not acceptable so respond with SIP Client Error */
            /* final response of 488 Not Acceptable Here       */
            if (gc_RejectModifyCall(crn,
                                    IPEC_SIPReasonStatus488NotAcceptableHere,
                                    EV_ASYNC) < 0)
            {
               // Invoke the application error handler to drop the call
               applicationHandleError(...);
            }
         }
         break;
      }

      case GCEV_ACCEPT_MODIFY_CALL:
        .
        .
        .
        /* notify user of changed attribute */
        .
        .
        .
        break;
```

```
case GCEV_ACCEPT_MODIFY_CALL_FAIL:
   /* process failure to change attribute */
   if (gc_ResultInfo(&metaevent, &t_info) < 0)
   {
      /* failure logic here */
   }
   /* process information contained in t_info */
   /* can optionally call gc_RejectModifyCall( ) to retry */
   .
   .
   .
   break;

case GCEV_REJECT_MODIFY_CALL:
   .
   .
   .
   /* notify user of rejected attribute */
   .
   .
   .
   break;

case GCEV_REJECT_MODIFY_FAIL:
   /* process failure to reject request */
   if (gc_ResultInfo(&metaevent, &t_info) < 0)
   {
      /* failure logic here */
   }
   /* process information contained in t_info */
   /* can optionally call gc_RejectModifyCall( ) to retry */
   .
   .
   .
   break;

case GCEV_MODIFY_CALL_ACK:
   // indication that re-invite was accepted (200 ok received ) by the remote endpoint.
   // This metaevent may have an IPSET_SDP/IPPARM_SDP_OFFER or
   // IPSET_SDP/IPPARM_SDP_ANSWER attached.
   .
   .
   .
   break;

case GCEV_SIP_200OK:
   // indication that the library needs to send a SIP ACK.
   // A parameter block containing a IPSET_SDP/IPPARM_SDP_ANSWER would be included
   // in the gc_SipAck for an outbound invite/re-invite delayed offer call scenario.
   if (gc_SipAck(crn, NULL, EV_ASYNC) != GC_SUCCESS)
   {
      // call application error handler to drop the call
      applicationHandleError(...);
   }
   break;

case GCEV_SIP_ACK_FAIL:
   // gc_SipAck completion metaevent indicating the Sip Ack could not be sent.
   // Invoke the application error handler to drop the call.
      applicationHandleError(...);
   break;

case GCEV_SIP_ACK_OK:
   // gc_SipAck completion metaevent indicating the Sip Ack was successfully sent
   break;
```

```
        case GCEV_SIP_ACK:
           // unsolicited event indicating SIP ACK was received on invite request.
           // This metaevent Will contain an IPSET_SDP/IPPARM_SDP_ANSWER in a inbound
           // invite/reinvite delayed offer call scenario.
           .
           .
           .
           break;
        .
        .
        .
     } /* endof switch */
  } /* endof process_event function */
```

■ See Also

- **gc_AcceptModifyCall( )**
- **gc_ReqModifyCall( )**

gc_ReqModifyCall( )

Name:	int gc_ReqModifyCall (crn, parmblkp, mode)	
Inputs:	CRN crn	• call reference number of call targeted for modification
	GC_PARM_BLK *parmblkp	• pointer to GC_PARM_BLK which contains attributes of call requested for modifying
	unsigned long mode	• completion mode (EV_ASYNC)
Returns:	0 if successful <0 if unsuccessful	
Includes:	gclib.h	
Category:	Call Modification	
Mode:	Asynchronous	

■ Description

The **gc_ReqModifyCall( )** function is used to initiate a request to the network or remote party to change an attribute of the current SIP call.

This function initiates a subsequent INVITE (also known as a re-INVITE) request in the context of a current dialog (connected call). The re-INVITE can be used to change signaling headers, one or more attributes of the media session, or the DTMF mode. This function is also used to cancel a pending re-INVITE that the application previously initiated.

Parameter	Description
crn	call reference number of call targeted for modification
parmblkp	pointer to GC_PARM_BLK which contains attributes of call requested for modifying. In 1PCC mode, this parameter block may contain a combination of SIP header fields and Global Call channel capabilities that will be inserted into the SDP offer that the library formulates. The parameter block may also contain a parameter element to change the DTMF mode of the call. In 3PCC mode, this parameter block may contain a combination of one or more SIP header fields and an SDP offer explicitly constructed by the third party call control application.
mode	must be EV_ASYNC

The function returns either <0 (to indicate failure) or 0, depending only upon the validity of the parameters. The function return does not indicate any status as to the success or failure of the sending of the re-INVITE request message. The final result of the attempt to send the request is provided in termination events.

The parameters elements contained in the GC_PARM_BLK that is passed to this function determine the contents of the re-INVITE request message. A special parameter element is also defined to cancel a pending re-INVITE request.

To set one or more message header fields in the re-INVITE request, the application inserts into the GC_PARM_BLK a parameter of the following form for each header field to be set:

IPSET_SIP_MSGINFO
IPPARM_SIP_HDR
- value = string representing the complete header field, including field name

Most SIP header fields that are valid in an INVITE request can be modified in a re-INVITE request without restriction. The most notable exceptions to this generalization are the Call-ID header and the URI and Tag in the To and From headers, which RFC 3261 specifies must match the headers in the original INVITE request. The following table specifies the header fields that are subject to restrictions or that are automatically populated by the SIP stack.

Header Field	Modifiable Parameters	Restricted Parameters	Automatically Populated Information
Call-ID	None	All	All
Contact	All	None	If not specified, copied from last INVITE or 2xx response transmitted in current dialog
CSeq	None	All	All
From	Display, URI parameters except: user, ttl, method, maddr	URI, Tag	URI, Tag
Max-Forwards	All	None	If not specified, 70
To	Display, URI parameters except: user, ttl, method, maddr	URI, Tag	URI, Tag
Via	All	None	If not specified, copied from last INVITE or 2xx response transmitted in current dialog

To request a change in the attributes of a media session, the application uses the same parameter mechanism that is used when offering a session invitation via **gc_MakeCall( )**. The application inserts into the GC_PARM_BLK one or more parameter of the following form:

GCSET_CHAN_CAPABILITY
IPPARM_LOCAL_CAPABILITY
- value = IP_CAPABILITY structure containing the details of the proposed media session, including capability (transcoder type) and direction

To modify the media attributes for a full-duplex connection, the application must insert at least two of these parameters, one for each direction, with the appropriate value set in the direction field of each IP_CAPABILITY structure. All fields in each IP_CAPABILITY structure must be fully specified even if only one characteristic is actually being changed (for example, if only the direction field is being modified to place a call on hold). If no media capability parameters are inserted into the GC_PARM_BLK, the library automatically includes the last SDP answer from the dialog when it constructs the re-INVITE request.

To request a change in the attributes of a media session **in 3PCC mode**, a call control application explicitly constructs an SDP offer containing the desired new attributes, and then inserts it into the GC_PARM_BLK as a parameter element of the following type (see Section 5.2.2.2, "IPSET_SDP Parameter Set Identifier", on page 435 for more details):

IPSET_SDP
  IPPARM_SDP_OFFER
  - value = properly constructed SDP offer

To request a change in the DTMF mode, the application inserts into the GC_PARM_BLK a parameter element of the following type:

IPSET_DTMF
  IPPARM_SUPPORT_DTMF_BITMASK
  - value = IP_DTMF_TYPE_INBAND_RTP or IP_DTMF_TYPE_RFC_2833

To cancel a pending re-INVITE request, the application inserts into the GC_PARM_BLK the following parameter:

IPSET_MSG_SIP
  IPPARM_SIP_METHOD
  - value = IP_MSGTYPE_SIP_CANCEL, size = sizeof(int)

  Note: When using this parameter value, this must be the only parameter element inserted into the GC_PARM_BLK.

■ Termination Events

GCEV_ MODIFY_CALL_ACK
  Successful termination event for call modification request. Indicates that the network or remote party accepted and acknowledged the request with a 200OK, and that the library has acknowledged the 200OK. In 1PCC mode, this event also indicates that any media changes that were proposed and accepted have been completed.

GCEV_MODIFY_CALL_REJ
  Unsuccessful termination event for call modification request, indicating that the request was rejected. The network or remote party declined and rejected the request by sending a 3xx, 4xx, 5xx, or 6xx response code in reply to the re-INVITE, and the library automatically sent an ACK. The specific response code can be retrieved from the Global Call METAEVENT by calling **gc_ResultInfo( )**. If the response code from the remote party was a 408 Request Timeout or 481 Dialog/Transaction Does Not Exist, the call that was being modified is disconnected automatically, and a GCEV_DISCONNECTED event is generated to the application. For all other response codes, no modifications to the existing dialog or media session are performed and the current state remains as it was prior to the attempting the modification request.

GCEV_MODIFY_CALL_FAIL
  Unsuccessful termination event for call modification request, indicating that the signaling of the request failed. Some possible reasons include a failure in the message transport, a timeout awaiting the response from the network or remote party, attempting to modify a call which is not currently connected, or attempting to initiate a request to modify a call while another modify request transaction is still pending. More specific information can be retrieved from the Global Call METAEVENT by calling **gc_ResultInfo( )**. On failure, no modifications to the

existing dialog or media session are performed and the current state remains as it was prior to the attempting the modification request.

GCEV_CANCEL_MODIFY_CALL
: Successful termination event for a request to cancel a pending call modification request. Indicates that the remote UA accepted the CANCEL method and sent a 200OK, and the library automatically sent an ensuing ACK. The previously sent re-INVITE dialog is terminated and no attribute changes are performed. In this case the application will not receive a termination event for the original **gc_ReqModifyCall( )** call (the one which initiated the re-INVITE dialog).

GCEV_CANCEL_MODIFY_CALL_FAIL
: Unsuccessful termination event for a request to cancel a pending call modification request. Indicates that the signaling of the CANCEL method failed, likely due to invalid state, such as having received a final 2xx-6xx response to the subject re-INVITE. In this case, the application *will* receive a termination event for the prior **gc_ReqModifyCall( )** call (either before or after this event) to indicate the successful or failed outcome of original re-INVITE transaction.

■ Cautions

- This function is only supported when the value of the parameter IPSET_CONFIG / IPPARM_OPERATING_MODE has been set to IP_T38_MANUAL_MODIFY_MODE using the **gc_SetConfigData( )** function. If this parameter value has not been set, the function call will fail with an error value of IPERR_BAD_PARM.
- Only asynchronous mode is supported. Calling the function in synchronous mode will fail and return an error value of GC_ERROR while setting CCLIB error to IPERR_BAD_PARAM.
- This function can only be called in the connected call state. If the CRN is not valid, the function fails and returns GC_ERROR while setting CCLIB error to IPERR_BAD_PARAM.
- Only one re-INVITE transaction can be pending in a call at any given time. Any re-INVITE transaction previously issued on the call must terminate (as indicated by a termination event) before a new one is initiated, otherwise the function will fail.

■ Errors

- The function returns GC_ERROR (with CCLIB error set to IPERR_BAD_PARM) if the CRN is not valid, if the mode is not set to EV_ASYNC, or if the value of the configuration parameter IPSET_CONFIG / IPPARM_OPERATING_MODE has not been set to IP_T38_MANUAL_MODIFY_MODE in 1PCC operating mode.
- Upon receiving a termination event that indicates a failure, use the **gc_ResultInfo( )** function to retrieve information about the event. See the "Error Handling" section in the *Dialogic® Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file while IP-specific error codes are specified in *gcip_defs.h*.

■ Example

The first code example illustrates an application requesting the current media session be changed to G.729 to limit bandwidth consumption.

The following code illustrates how this coder change is initiated in the first party call control (1PCC) operating mode using the **gc_ReqModifyCall( )** function.

```
.
.
/* Dialogic Header Files */
#include <gcip.h>
#include <gclib.h>
.
.
.

/* Request remote SIP client to change call to G.729:                     */
/* Assumes: 1) caller has verified call to be in connected state          */
/*          2) caller has enabled event handler for GCEV_MODIFY_CALL_ACK, */
/*             GCEV_MODIFY_CALL_REJ, and GCEV_MODIFY_CALL_FAIL.           */

int reqChangeToG729(CRN crn)
{
   IP_CAPABILITY   ipcap;
   GC_PARM_BLK    *parmblkp = NULL;

   memset(&ipcap, 0, sizeof(IP_CAPABILITY));
   ipcap.capability = GCCAP_AUDIO_g729;
   ipcap.type = GCCAPTYPE_AUDIO;
   ipcap.extra.audio.frames_per_pkt = 1;
   ipcap.extra.audio.VAD = 0;
   /* Specify TX direction */
   ipcap.direction = IP_CAP_DIR_LCLTRANSMIT;

   /* append the GC_PARM_BLK with the respective local TX codec */
   gc_util_insert_parm_ref(&parmblkp,
                           GCSET_CHAN_CAPABILITY,
                           IPPARM_LOCAL_CAPABILITY,
                           sizeof(IP_CAPABILITY),
                           &ipcap);
   if (NULL == parmblkp)  return FAILURE;

   /* Specify local RX direction */
   ipcap.direction = IP_CAP_DIR_LCLRECEIVE;
   /* append the GC_PARM_BLK with the respective RX codec */
   gc_util_insert_parm_ref(&parmblkp,
                           GCSET_CHAN_CAPABILITY,
                           IPPARM_LOCAL_CAPABILITY,
                           sizeof(IP_CAPABILITY),
                           &ipcap);
   if (NULL == parmblkp)  return FAILURE;

   if (gc_ReqModifyCall(crn, parmblkp, EV_ASYNC) < 0)  return FAILURE;

   gc_util_delete_parm_blk(parmblkp);

} /* End of function. */
```

The following code illustrates how this coder change is initiated in the third party call control (3PCC) operating mode using the **gc_ReqModifyCall( )** function.

```
.
.
.
/* Dialogic Header Files */
#include <gcip.h>
#include <gclib.h>
.
.
.
```

```
/* Request remote SIP client to change call to G.729:                  */
/* Assumes: 1) caller has verified call to be in connected state          */
int reqChangeToG729(CRN crn)
{
   GC_PARM_BLK    *parmblkp = NULL;

   char sdpG729[] =
      "v=0\r\n" \
      "o=Dialogic_IPCCLib 146430240 146430241 IN IP4 169.254.00.01\r\n" \
      "s=Dialogic_SIP_CCLLIB\r\n" \
      "i=session information\r\n" \
      "c=IN IP4 169.254.00.01\r\n" \
      "t=0 0\r\n" \
      "m=audio 2002 RTP/AVP 18\r\n" \
      "a=ptime:10\r\n";

   frc = gc_util_insert_parm_ref_ex(&parmblkp,
                                  IPSET_SDP,
                                  IPPARM_SDP_OFFER,
                                  sizeof(sdpG729),
                                  sdpG729);

   if (NULL == parmblkp)  return FAILURE;

   if (gc_ReqModifyCall(crn, parmblkp, EV_ASYNC) < 0)  return FAILURE;

   gc_util_delete_parm_blk(parmblkp);

   return SUCCESS;

} /* End of function. */
```

The following code example illustrates the use of **gc_ReqModifyCall( )** to place the current media session on hold using the SDP media attribute “inactive”.

```
.
.
/* Dialogic Header Files */
#include <gcip.h>
#include <gclib.h>
.
.
.

/* Request remote SIP client to place call on hold:                     */
/* Assumes: 1) caller has verified call to be in connected state          */
/*          2) caller has enabled event handler for GCEV_MODIFY_CALL_ACK, */
/*             GCEV_MODIFY_CALL_REJ, and GCEV_MODIFY_CALL_FAIL.           */

int holdReq(CRN crn, IP_CAPABILITY * pIpcap)
{
    GC_PARM_BLK *parmblkp = NULL;

    /* Change direction to "inactive" direction */
    pIpcap->direction = IP_CAP_DIR_LCLRTPINACTIVE;

    /* append the GC_PARM_BLK with the respective modified codec direction */
    gc_util_insert_parm_ref(&parmblkp,
                        GCSET_CHAN_CAPABILITY,
                        IPPARM_LOCAL_CAPABILITY,
                        sizeof(IP_CAPABILITY),
                        pIpcap);
    if (NULL == parmblkp) return FAILURE;

    if (gc_ReqModifyCall(crn, parmblkp, EV_ASYNC) < 0) return FAILURE;
    gc_util_delete_parm_blk(parmblkp);

} /* End of function. */
```

The following example illustrates the use of **gc_ReqModifyCall( )** to refresh the Contact header:

```
.
.
/* Dialogic Header Files */
#include <gcip.h>
#include <gclib.h>
.
.
.

/* Request Contact refresh:                                            */
/* Assumes: 1) caller has verified call to be in connected state       */
/*          2) caller has enabled event handler for GCEV_MODIFY_CALL_ACK, */
/*             GCEV_MODIFY_CALL_REJ, and GCEV_MODIFY_CALL_FAIL.          */

int refreshToHomeLocation (CRN crn)
{

   char *pContactHeader = "Contact: Rich <sip:r.intelligent@myhomeISP.com>";

   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_SIP_MSGINFO,
                           IPPARM_SIP_HDR,
                           (unsigned char)strlen(pContactIdHeader) + 1,
                           pContactHeader);

   if (NULL == parmblkp)  return FAILURE;

   if (gc_ReqModifyCall(crn, parmblkp, EV_ASYNC) < 0)  return FAILURE;

   gc_util_delete_parm_blk(parmblkp);

} /* End of function. */
```

■ See Also

- **gc_AcceptModifyCall( )**
- **gc_RejectModifyCall( )**

gc_SetAuthenticationInfo( )

Name:	int gc_SetAuthenticationInfo(target_type, target_id, infoparmblkp)	
Inputs:	int target_type	• type of target object (virtual board)
	long target_id	• target object ID
	GC_PARM_BLKP infoparmblkp	• pointer to GC_PARM_BLK with user information
Returns:	0 if successful <0 if failure	
Includes:	gclib.h gcerr.h	
Mode:	synchronous	

■ Description

The **gc_SetAuthenticationInfo( )** function is used to configure or remove authentication information on an IPT virtual board. This is the only Global Call function that can be used to set this information; the generic Global Call functions **gc_SetConfigData( )** and **gc_SetUserInfo( )** functions cannot be used for this IP-specific configuration operation.

This function should be called before using any Global Call function that sends a SIP request which may provoke a 401/407 response. A 401/407 response to any SIP request that was sent before authentication is configured causes the request to be terminated (with the reason code IPEC_SIPReasonStatus401Unauthorized or IPEC_SIPReasonStatus407ProxyAuthenticationRequired), and Global Call will not attempt to re-send the request.

Parameter	Description
target_type	specifies the type of target object; must be set to GCTGT_CCLIB_NETIF.
target_id	specifies the virtual board ID that the authentication information applies to
infoparmblklp	points to a GC_PARM_BLK structure that contains the authentication information. The parm block contains one or more parameters that use the IPSET_CONFIG set ID and IPPARM_AUTHENTICATION_CONFIGURE or IPPARM_AUTHENTICATION_REMOVE as the parameter ID.

To add a new authentication quadruplet of {realm, identity, username, password} to the Global Call database, or to update an existing quadruplet, the application inserts a parameter element of the following type into the **infoparmblkp** parameter block:

IPSET_CONFIG
- IPPARM_AUTHENTICATION_CONFIGURE
  - value = IP_AUTHENTICATION data structure specifying the quadruplet to create/update

If the realm and identity strings in the IP_AUTHENTICATION structure are unique, the library creates a new authentication quadruplet in the database. If both the realm and identity strings match a quadruplet that already exists, the existing username and password are overwritten with the new strings. If the identity field in the IP_AUTHENTICATION structure is an empty string, the function will set the specified username and password as the defaults for the specified realm.

To remove an authentication quadruplet to the Global Call database, the application inserts a parameter element of the following type into the **infoparmblkp** parameter block:

IPSET_CONFIG
IPPARM_AUTHENTICATION_REMOVE
- value = IP_AUTHENTICATION data structure identifying the realm and identity of the quadruplet to remove

In this case, the library will remove the existing authentication quadruplet that matches the realm and identity strings that are specified in the IP_AUTHENTICATION structure; the username and password elements in the IP_AUTHENTICATION structure are ignored.

Cautions

- The **gc_SetAuthenticationInfo( )** function can only be called on a virtual board device.
- If the GC_PARM_BLK contains multiple parameter elements with the same realm/identity pair in their IP_AUTHENTICATION structures, all of those parameters are ignored except for the one that is last in the GC_PARM_BLK.

Errors

If this function returns <0 to indicate failure, use the **gc_ErrorInfo( )** function to retrieve the reason for the error. See the "Error Handling" section in the *Dialogic® Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.

Possible errors include:

IPERR_BAD_PARM
returned if any of the string pointers in an IP_AUTHENTICATION structure is NULL or if there is any other invalid parameter

IPERR_UNAVAILABLE
returned when the realm/identity does not exist in the Global Call database when the application attempts to remove the quadruplet

IPERR_UNSUPPORTED
returned when the function is called on a line device or CRN rather than a virtual board

Examples

The following code example illustrates how to add or modify a digest authentication quadruplet.

```
#include <gcip.h>
#include <gclib.h>

/* This example adds or modifies the quadruplet with realm "example.com" and
 * identity "sip:bob@example.com". If this realm/identity do not exist on this
 * virtual board, this quadruplet will be added. If this realm/identity exist
 * already, it will be override by this quadruplet.
 */
```

```
void configureAuthQuadruplet (long boardDev)
{
   GC_PARM_BLK *parmblkp = NULL;
   char realm[] = "example.com";
   char identity[] = "sip:bob@example.com";
   char username[] = "bob";
   char password [] = "password1";

   IP_AUTHENTICATION authentication;
   INIT_IP_AUTHENTICATION (&authentication);
   authentication.realm = realm;
   authentication.identity = identity;
   authentication.username = username;
   authentication.password = password;

   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_CONFIG,
                           IPPARM_AUTHENTICATION_CONFIGURE,
                           (unsigned char)(sizeof(IP_AUTHENTICATION)),
                           &authentication);

   gc_SetAuthenticationInfo(GCTGT_CCLIB_NETIF, boardDev, parmblkp);

   gc_util_delete_parm_blk(parmblkp);
}
```

The following code example illustrates how to remove a digest authentication quadruplet.

```
#include <gcip.h>
#include <gclib.h>

/* This example deletes the quadruplet with realm "example.com" and
 * identity "sip:bob@example.com".
 */

void removeAuthQuadruplet (long boardDev)
{
   GC_PARM_BLK *parmblkp = NULL;
   char realm[] = "example.com";
   char identity[] = "sip:bob@example.com";

   IP_AUTHENTICATION authentication;
   INIT_IP_AUTHENTICATION (&authentication);

   authentication.realm = realm;
   authentication.identity = identity;

   gc_util_insert_parm_ref(&parmblkp,
                           IPSET_CONFIG,
                           IPPARM_AUTHENTICATION_REMOVE,
                           (unsigned char)(sizeof(IP_AUTHENTICATION)),
                           &authentication);

   gc_SetAuthenticationInfo(GCTGT_CCLIB_NETIF, boardDev, parmblkp);

   gc_util_delete_parm_blk(parmblkp);
}
```

■ See Also

None.

gc_SipAck( )

Name:	int gc_SipAck(crn, parmblk, mode)	
Inputs:	CRN crn	• call reference number of call targeted for modification
	GC_PARM_BLKP parmblk	• pointer to optional parameter block containing SDP content for the SIP ACK message
	unsigned long mode	• completion mode (EV_ASYNC)
Returns:	0 if successful <0 if unsuccessful	
Includes:	gclib.h	
Category:	Third-party Call Control	
Mode:	Asynchronous only	

■ Description

The **gc_SipAck( )** function is specific to the SIP protocol, and is used only in the third-party call control (3PCC) mode. The function is used to send an explicit SIP ACK message to the remote party on an outbound INVITE or re-INVITE transaction when the library does not automatically send an ACK. In particular, this function must be called in response to the reception of an unsolicited GCEV_SIP_200OK event or else the transaction will time out and fail.

SDP content may be included in the ACK message by passing a pointer to a parameter block that contains a parameter element that uses the IPSET_SDP set ID.

This function is supported only in third-party call control (3PCC) mode. Calling this function when the library has been started in the default first-party call control (1PCC) mode produces an error.

Parameter	Description
crn	call reference number of the call that is involved in the INVITE or re-INVITE transaction
parmblk	pointer to a optional parameter block containing SDP content for the SIP ACK message; must be set to NULL if no SDP content is to be included in the outbound ACK message.
mode	must be EV_ASYNC

This function returns either GC_SUCCESS or GC_ERROR depending upon the validity of the parameters. The function return does not indicate any status as to the success or failure of the sending of the response (that is, the ACK). The final result of sending the response is provided in termination events.

Termination Events

GCEV_SIP_ACK_OK
: Successful termination event for **gc_SipAck( )** indicating that the ACK message was successfully sent

GCEV_ SIP_ACK_FAILED
: Unsuccessful termination event for **gc_SipAck( )** indicating that the ACK message could not be sent. The ACK message could not be sent because the dialog state was invalid for the ACK message generation. No modifications to the existing dialog was performed and the current state remains as it was prior to the **gc_SipAck( )** request.

Unsolicited Events

GCEV_SIP_200OK
: Unsolicited event indicating the application should call **gc_SipAck( )** to complete the SIP transaction.

GCEV_SIP_ACK
: Unsolicited event indicating that a SIP ACK message was received and the SIP transaction is complete.

Cautions

- When a Global Call third-party call control application receives a GCEV_SIP_200OK event, the application must call **gc_SipAck( )** to complete the dialog's transaction, or else that transaction will time out and fail.

Errors

- If the function returns GC_ERROR, one or more of the parameters are invalid.
- Upon receiving GCEV_SIP_ACK_FAILED event, use the **gc_ResultInfo( )** function to retrieve information about the event. See the "Error Handling" section in the *Dialogic® Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file while IP-specific error codes are specified in *gcip_defs.h*.
- On failure, no modifications to the existing dialog or media session are performed and the current state remains as it was prior to the incoming modification request.

Example

```
/*                                                                    */
/* Description: Application event handler that processes new  GC 3PCC events.  */
/*                                                                    */
/* Assumes:  caller has enabled event handler for GCEV_SIP_ACK,       */
/*           GCEV_SIP_ACK_FAILED, GCEV_SIP_ACK_OK, and GCEV_SIP_200OK */
/*                                                                    */

/* Dialogic Header Files */
#include <gcip.h>
#include <gclib.h>
```

```
      /* SRL event handler: */
      for (;;)
      {
         if (-1 != sr_waitevt(500))
            process_event();
      }

   void process_event(void)
   {
      METAEVENT    metaevent;
      GC_INFO      t_info;

      /* Populate the metaEvent structure */
      if(GC_SUCCESS != gc_GetMetaEvent(&metaevent)) return;

      /* process GlobalCall events */
      if ((metaevent.flags & GCME_GC_EVENT) == 0)
         return;

      switch (metaevent.evttype)
      {
         .
         .
         .
         case GCEV_SIP_200OK:  /* request to modify call attribute */
         {
            EXTENSIONEVTBLK *extblkp = metaevent.extevtdatap;
            GC_PARM_BLKP parm_blkp = &extblkp->parmblk;
            GC_PARM_DATA_EXT curParm;
            INIT_GC_PARM_DATA_EXT(&curParm);

            while ((curParm = gc_util_next_parm_ex(parm_blkp, &curParm)) != NULL)
            {
               .
               .
               .
               /* parse and evaluate each proposed attribute change (code not shown)*/
               .
               .
               .
            }

            if ( gc_SipAck(metaevent.crn, NULL, EV_ASYNC) != GC_SUCCESS)
            {
               .
               .
               .
                /* perform error recovery here */
               .
               .
               .
            }

            break;
         }

         case GCEV_SIP_ACK_OK:
            .
            .
            .
            /* remote dialog transaction complete */
            .
            .
            .
            break;
         case GCEV_SIP_ACK_FAILED:
            /* process failure to change attribute */
            if (gc_ResultInfo(&metaevent, &t_info) < 0)
               /* failure logic here */
            /* process information contained in t_info */
            .
            .
            .
            break;

         case GCEV_ACK:
            .
            .
            .
            /* local dialog transaction complete */
            .
            .
            .
            break;

            .
            .
            .
      } /* endof switch */

   } /* endof process_event function */
```

■ See Also

None

gc_SipPrack( )

Name:	int gc_SipPrack (crn, parmblk, mode)	
Inputs:	CRN crn	• call reference number
	GC_PARM_BLKP parmblk	• pointer to an optional parameter block containing SDP content for the SIP PRACK message
	unsigned long mode	• completion mode (EV_ASYNC)
Returns:	0 if successful <0 if unsuccessful	
Includes:	gclib.h	
Category:	Third-party Call Control	
Mode:	Asynchronous	

■ Description

The **gc_SipPrack( )** function is specific to the SIP protocol, and is used only in the third-party call control (3PCC) mode. It sends a PRACK message to the remote party of an outbound INVITE or re-INVITE call.

Parameter	Description
crn	call reference number of the call that is involved in the INVITE or re-INVITE
parmblk	pointer to an optional parameter block containing SDP content for the SIP PRACK message. This is set to NULL if no SDP content is included in the outbound PRACK message.
mode	must be set to EV_ASYNC for asynchronous execution

■ Termination Events

The termination events for this function are:

GCEV_SIP_PRACK_OK
Indicates that the PRACK message was sent successfully..

GCEV_SIP_PRACK_FAILED
Indicates that the PRACK message could not be sent because the state was invalid when calling this function.

■ Errors

If this function returns <0 to indicate failure, use **gc_ErrorInfo( )** for error information.

■ Example

See Section 4.19, “Sending and Receiving SIP PRACK”, on page 274 for example code.

■ See Also

- **gc_SipPrackResponse( )**

gc_SipPrackResponse( )

Name:	int gc_SipPrackResponse (crn, parmblk, mode)	
Inputs:	CRN crn	• call reference number
	GC_PARM_BLKP parmblk	• pointer to an optional parameter block containing SDP content for the SIP PRACK message
	unsigned long mode	• completion mode (EV_ASYNC)
Returns:	0 if successful <0 if unsuccessful	
Includes:	gclib.h	
Category:	Third-party Call Control	
Mode:	Asynchronous	

■ Description

The **gc_SipPrackResponse( )** function is specific to the SIP protocol, and is used only in the third-party call control (3PCC) mode. It sends a PRACK response message to the remote party of an outbound INVITE or re-INVITE call.

Parameter	Description
crn	call reference number of the call that is involved in the INVITE or re-INVITE
parmblk	pointer to an optional parameter block containing SDP content for the SIP PRACK message. This is set to NULL if no SDP content is included in the outbound PRACK message.
mode	must be set to EV_ASYNC for asynchronous execution

■ Termination Events

The termination events for this function are:

GCEV_SIP_PRACK_RESPONSE_OK
Indicates that the PRACK response was sent successfully..

GCEV_SIP_PRACK_RESPONSE_FAILED
Indicates that the PRACK response could not be sent because the state was invalid when calling this function.

■ Errors

If this function returns <0 to indicate failure, use **gc_ErrorInfo( )** for error information.

■ Example

See Section 4.19, “Sending and Receiving SIP PRACK”, on page 274 for example code.

■ See Also

- **gc_SipPrack( )**

gc_SipSessionProgress( )

Name:	int gc_SipSessionProgress(crn, mode)	
Inputs:	CRN crn	• call reference number
	unsigned long mode	• completion mode (EV_ASYNC or EV_SYNC)
Returns:	0 if successful <0 if unsuccessful	
Includes:	gclib.h	
Category:	Optional call handling	
Mode:	Asynchronous or synchronous	

■ Description

The **gc_SipSessionProgress( )** function indicates to the originator that the call will be answered. This function provides a “183 Session Progress” to the destination party request acknowledging that the call has been received but is not yet answered. Upon successful completion of this function, the call state changes from the Offered state to the Accepted state. If the call state is already in the Accepted state, then it will remain in the Accepted state.

Notes: ***1.*** In 1PCC mode, the SDP cannot be controlled by the user application. Therefore, **gc_SipSessionProgress( )** cannot be used to change any SDP information.

2. The SDP can be set and sent when operating in 3PCC mode only.

Parameter	Description
crn	Specifies the call reference number for the call between the remote party A and the local party that received the call.
mode	Set to EV_ASYNC for asynchronous execution or EV_SYNC for synchronous.

■ Termination Events

The termination events for this function are:

GCEV_SIP_SESSIONPROGRESS
Indicates a successful completion of the function.

GCEV_TASKFAIL
Indicates that the function failed.

■ Cautions

The **gc_SipSessionProgress( )** function will fail when called for an unsupported technology. The error value EGC_UNSUPPORTED will be the Global Call value returned when the **gc_ErrorInfo( )** function is used to retrieve the error code.

Errors

If this function returns <0 to indicate failure, use the **gc_ErrorInfo( )** function for error information. If the GCEV_TASKFAIL event is received, use the **gc_ResultInfo( )** function to retrieve information about the event.

Example

```
#include <gclib.h>
#include <gcerr.h>
.
.
.
/*
 * Assume the following has been done:
 * 1. Opened line devices for each time slot
 * 2. Wait for a call using gc_WaitCall( )
 * 3. An event has arrived and has been converted to a
 * metaevent using gc_getMetaEvent() or
 * gc_getMetaEventEx()
 * 4. The event is determined to be a GCEV_OFFERED event
 */
 Int progress_call(void)
 {
     CRNcrn;/* Call reference number */
     GC_INFOgc_error_info;/* GlobalCall error information data */

 /*
  * Send a 183 Session Progress response
  */
      Crn = metaevent.crn;
      if (gc_SipSessionProgress(crn, , EV_ASYNC) != GC_SUCCESS) {
      /* process error returned */
           Gc_ErrorInfo( &gc_error_info );
           Printf("Error gc_SipSessionProgress( ) on device
                     handle: 0x%lx, GC ErrorValue:
                     0x%hx - %s, CCLibID: %i - %s, CC
                     ErrorValue: 0x%1x - %s\n",
                     Metaevent.evtdev,
                     gc_error_info.gcValue,
                     gc_error_info.gcMsg,
                     gc_error_info.ccLibId,
                     gc_error_info.ccLibName,
                     gc_error_info.ccValue,
                     gc_error_info.ccMsg);
           return (gc_error_info.gcvalue);
     }
 /*
  * gc_SipSessionProgress( ) terminates with
  * GCEV_SIP_SESSIONPROGRESS event. When the
  * GCEV_SIP_SESSIONPROGRESS event is received, the
  * state changes to Accepted and the
  * gc_AnswerCall( ) can be issued to complete the
  * connection.
  */
    return (0);
}
```

See Also

None

gc_util_copy_parm_blk( )

Name:	int gc_util_copy_parm_blk(parm_blkpp, parm_blkp)	
Inputs:	GC_PARM_BLKP* parm_blkpp	• pointer to the address of the new GC_PARM_BLK
	GC_PARM_BLKP parm_blkp	• pointer to a valid GC_PARM_BLK to be copied
Returns:	GC_SUCCESS if successful GC_ERROR if unsuccessful	
Includes:	gclib.h gcerr.h	
Category:	GC_PARM_BLK utility	
Mode:	synchronous	

■ Description

The **gc_util_copy_parm_blk**( ) function copies the specified GC_PARM_BLK.

This function **must** be used to copy any GC_PARM_BLK that contains any parameter elements (setID/parmID pairs) that can have data that is potentially larger than 255 bytes. This function can be used for any GC_PARM_BLK, regardless of whether it contains setID/parmID pairs that support parameter data lengths greater than 255 bytes.

The parameters that currently support extended-length values include:

- IPSET_MIME (or IPSET_MIME_200OK_TO_BYE) / IPPARM_MIME_PART_HEADER
- IPSET_MIME (or IPSET_MIME_200OK_TO_BYE) / IPPARM_MIME_PART_TYPE
- IPSET_NONSTANDARDCONTROL / IPPARM_NONSTANDARDDATA_DATA
- IPSET_NONSTANDARDDATA / IPPARM_NONSTANDARDDATA_DATA
- IPSET_SDP / all four parameter IDs (supported in 3PCC operating mode only)
- IPSET_SIP_MSGINFO / IPPARM_SIP_HDR
- IPSET_TUNNELEDSIGNALMSG / IPPARM_TUNNELEDSIGNALMSG_DATA

Parameter	Description
parm_blkpp	pointer to the address of the new GC_PARM_BLK that the specified parm block will be copied to; **must** be set to NULL
parm_blkp	points to a valid, existing GC_PARM_BLK to be copied

■ Cautions

To avoid a memory leak, any GC_PARM_BLK created must eventually be deleted using the **gc_util_delete_parm_blk**( ) function.

Errors

If this function returns GC_ERROR(-1) to indicate failure, use the **gc_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Dialogic® Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.

Example

```
#include "gclib.h"
#include "gcip.h"

void process_event(void)
{
   METAEVENT  metaevent;
   GC_PARM_BLKP my_blkp = NULL;

   if(gc_GetMetaEvent(&metaevent) != GC_SUCCESS)
   {
      /* process error */
   }

   Switch(metaevent.evttype)
   {
      case GCEV_OFFERED:
         /* make a copy of the parm blk */
         if(metaevent.extevtdatap)
         {
            if ( gc_util_copy_parm_blk( &my_blkp,(GC_PARM_BLKP)(metaevent.extevtdatap))
                 != GC_SUCCESS )
            {
               /* Process error */
            }
         }
      ...
   }
   ...
}
```

See Also

- **gc_util_delete_parm_blk( )** (in *Dialogic® Global Call API Library Reference*)

gc_util_find_parm_ex( )

Name:	int gc_util_find_parm_ex(parm_blk, setID, parmID, parm)	
Inputs:	GC_PARM_BLKP parm_blk	• pointer to GC_PARM_BLK to search for the parameter
	unsigned long setID	• parameter set ID of parameter to be found
	unsigned long parmID	• parameter ID of parameter to be found
	GC_PARM_DATA_EXTP parm	• pointer to a valid GC_PARM_DATA_EXT structure that identifies where in the parm block to start searching
Outputs:	GC_PARM_DATA_EXTP parm	• if successful, pointer to a GC_PARM_DATA_EXT structure that contains the ID and value data for the specified parameter
Returns:	GC_SUCCESS if successful EGC_NO_MORE_PARMS if no more parameters exist in GC_PARM_BLK GC_ERROR if failure	
Includes:	gclib.h gcerr.h	
Category:	GC_PARM_BLK utility	
Mode:	synchronous	

■ Description

The **gc_util_find_parm_ex( )** function is used to find a parameter of a particular type in a GC_PARM_BLK and retrieve the parameter data into a GC_PARM_DATA_EXT structure.

This function **must** be used instead of the similar **gc_util_find_parm( )** function if the parameter data can potentially exceed 255 bytes. This function is backward compatible and can be used instead of **gc_util_find_parm( )** for any GC_PARM_BLK, regardless of whether the parameter block contains setID/parmID pairs that support data lengths greater than 255 bytes.

The parameters that currently support extended-length values include:

- IPSET_MIME (or IPSET_MIME_200OK_TO_BYE) / IPPARM_MIME_PART_HEADER
- IPSET_MIME (or IPSET_MIME_200OK_TO_BYE) / IPPARM_MIME_PART_TYPE
- IPSET_NONSTANDARDCONTROL / IPPARM_NONSTANDARDDATA_DATA
- IPSET_NONSTANDARDDATA / IPPARM_NONSTANDARDDATA_DATA
- IPSET_SDP / all four parameter IDs (supported in 3PCC operating mode only)
- IPSET_SIP_MSGINFO / IPPARM_SIP_HDR
- IPSET_TUNNELEDSIGNALMSG / IPPARM_TUNNELEDSIGNALMSG_DATA

The **gc_util_find_parm_ex( )** function can be used to determine whether a particular parameter exists, or to retrieve a particular parameter, or both. If the specified parameter is found in the GC_PARM_BLK, the function fills in the GC_PARM_DATA_EXT structure with the parameter data and returns GC_SUCCESS. If the parameter does not exist in the GC_PARM_BLK, or if no more parameters of the specified type are found, the function returns EGC_NO_MORE_PARMS.

To search from the beginning of the GC_PARM_BLK, initialize the GC_PARM_DATA_EXT structure by using **INIT_GC_PARM_DATA_EXT(parm)** before calling **gc_util_find_parm_ex( )**. If the structure pointed to by **parm** contains parameter information that was retrieved in a previous call to this function, the function will begin its search at that parameter rather than the beginning of the parameter block.

Parameter	Description
parm_blk	points to a valid GC_PARM_BLK that will be searched for a parameter of the specified type
setID	set ID of the parameter to be found
parmID	parameter ID of the parameter to be found
parm	points to a valid GC_PARM_DATA_EXT provided by the application. If a pointer to a newly initialized structure is passed in the function call, the function searches from the beginning of the GC_PARM_BLK; if the structure contains data from a previously found parameter, the function searches from that parameter onward. When the function completes successfully, the structure is updated to contain retrieved information for the parameter that was found.

■ Cautions

- Unlike the similar **gc_util_find_parm( )** function, the **parm** pointer used in this function *cannot* be used to update the parameter itself because it points to a data structure that is in the application's memory rather than a location in the GC_PARM_BLK itself.
- The **parm** parameter must point to a valid GC_PARM_DATA_EXT structure. If it is desired to search from the beginning of the parameter block, the application **must** initialize the structure via **INIT_GC_PARM_DATA_EXT(parm)** before calling **gc_util_find_parm_ex( )**.

■ Errors

If this function returns GC_ERROR to indicate failure, use the **gc_ErrorInfo( )** function to retrieve the reason for the error. See the "Error Handling" section in the *Dialogic® Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.

■ Example

```
#include "gclib.h"
#include "gcip.h"

void search_parm_block(GC_PARM_BLKP parm_blkp)
{
   GC_PARM_DATA_EXT parm_data_ext;
   int ret = 0;

   /* Initialize this structure for two reasons:
    * 1. To search from the first parameter in the parm block
    * 2. The first time this structure is used it must be initialized
    */
   INIT_GC_PARM_DATA_EXT(&parm_data_ext);
```

```
    /* loop to retrieve all of the parameters and associated data in the
     * GC_PARM_BLK that match the set_ID/parm_ID pair for SIP header fields.
     */
    while ( GC_SUCCESS == (ret = gc_util_find_parm_ex(parm_blkp, IPSET_SIP_MSGINFO,
                                                  IPPARM_SIP_HDR, &parm_data_ext)) )
    {
       /* process GC_PARM_DATA_EXT structure */
       .
       .
       .
    }

    /* Check for error */
    if ( GC_ERROR == ret)
    {
        /* process error */
    }

    .
    .
    .
 }
```

See Also

- **gc_util_next_parm_ex( )**

gc_util_insert_parm_ref_ex( )

Name:	int gc_util_insert_parm_ref_ex(parm_blkpp, setID, parmID, data_size, datap)	
Inputs:	GC_PARM_BLKP *parm_blkpp	• pointer to the address of a valid GC_PARM_BLK
	unsigned long setID	• set ID of parameter to be inserted
	unsigned long parmID	• parm ID of parameter to be inserted
	unsigned long data_size	• size in bytes of the parameter data
	void *datap	• pointer to the parameter data
Returns:	GC_SUCCESS if successful GC_ERROR if failure	
Includes:	gclib.h gcerr.h	
Category:	GC_PARM_BLK utility	
Mode:	synchronous	

■ Description

The **gc_util_insert_parm_ref_ex( )** function inserts a parameter element into a GC_PARM_BLK data structure using a reference to the parameter value data.

The **gc_util_insert_parm_ref_ex( )** function **must** be used rather than the similar **gc_util_insert_parm_ref( )** function whenever the parameter value data exceeds 255 bytes in length. The **gc_util_insert_parm_ref_ex( )** function is backwards compatible and can be used with any setID/parmID pair regardless of whether that pair supports values longer than 255 bytes.

The parameters that currently support extended-length values include:

- IPSET_MIME (or IPSET_MIME_200OK_TO_BYE) / IPPARM_MIME_PART_HEADER
- IPSET_MIME (or IPSET_MIME_200OK_TO_BYE) / IPPARM_MIME_PART_TYPE
- IPSET_NONSTANDARDCONTROL / IPPARM_NONSTANDARDDATA_DATA
- IPSET_NONSTANDARDDATA / IPPARM_NONSTANDARDDATA_DATA
- IPSET_SDP / all four parameter IDs (supported in 3PCC operating mode only)
- IPSET_SIP_MSGINFO / IPPARM_SIP_HDR
- IPSET_TUNNELEDSIGNALMSG / IPPARM_TUNNELEDSIGNALMSG_DATA

A new GC_PARM_BLK can be created by inserting the first parameter with ***parm_blkpp** set to NULL. A parameter can be inserted in an existing GC_PARM_BLK by setting ***parm_blkpp** to the address of that block.

Note: Parameters are contained in the GC_PARM_BLK in the order in which they are inserted, and they will also be retrieved via the **gc_util_next_parm_ex( )** function in the same order.

Parameter	Description
parm_blkpp	points to the address of a valid GC_PARM_BLK where the parameter element is to be inserted. Set ***parm_blkpp** to NULL to insert the parameter into a new block.
setID	set ID of the parameter to be inserted
parmID	parameter ID of the parameter to be inserted
data_size	size, in bytes, of the value data associated with this parameter. For certain set ID/parm ID pairs the maximum size is configurable at library start-up using IPCCLIB_START_DATA.max_parm_data_size; for all other parameters, the maximum size is 255 bytes.
datap	points to the value data associated with this parameter

■ Cautions

- To avoid a memory leak, any GC_PARM_BLK created must be deleted using the **gc_util_delete_parm_blk( )** function.
- Insertion of data that exceeds 255 bytes in length is only supported for specific setID/parmID pairs.

■ Errors

- If this function returns GC_ERROR to indicate failure, use the **gc_ErrorInfo( )** function to retrieve the reason for the error. See the "Error Handling" section in the *Dialogic® Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.
- Attempting to insert data greater than 255 bytes in length using a setID/parmID pair that does not support extended-length data produces an error indication. In this situation, the **gc_ErrorInfo( )** function returns the value EGC_INVPARM.

■ Example

```
#include "gclib.h"
#include "gcip.h"

void SetHeader(void)
{
   GC_PARM_BLKP my_blkp = NULL;
   char* pChar = "Remote-Party_ID: This string can be greater than 255 bytes";

   /* Add 1 to strlen for null termination */
   unsigned long data_size = strlen(pChar) + 1;

   /* insert parm and associated data into the GC_PARM_BLK */
   if ( gc_util_insert_parm_ref_ex( &my_blkp, IPSET_SIP_MSGINFO, IPPARM_SIP_HDR, data_size,
                                    (void*)( pChar )) != GC_SUCCESS )
   {
      /* Process error */
   }

   /* At this point the application can overwrite the data pointed to by pChar. */
   pChar = NULL;
   /* Pass the parm block to GC */
   if ( gc_SetUserInfo( GCTGT_GCLIB_CRN, crn, &my_blkp, GC_SINGLECALL) != GC_SUCCESS )
   {
      /* Process error */
   }

   /* GC_PARM_BLK is no longer needed; delete the block */
   gc_util_delete_parm_blk( my_blkp );
}
```

■ See Also

- **gc_util_delete_parm_blk( )** (in *Dialogic® Global Call API Library Reference*)
- **gc_util_insert_parm_ref( )** (in *Dialogic® Global Call API Library Reference*)
- **gc_util_insert_parm_val( )** (in *Dialogic® Global Call API Library Reference*)

gc_util_next_parm_ex( )

Name:	int gc_util_next_parm_ex(parm_blk, parm)	
Inputs:	GC_PARM_BLKP parm_blk	• pointer to GC_PARM_BLK
	GC_PARM_DATA_EXTP parm	• pointer to valid GC_PARM_DATA_EXT structure identifying current parameter
Outputs:	GC_PARM_DATA_EXTP parm	• pointer to GC_PARM_DATA_EXT structure containing retrieved next parameter
Returns:	GC_SUCCESS if successful EGC_NO_MORE_PARMS if no more parameters exist in the GC_PARM_BLK GC_ERROR if failure	
Includes:	gclib.h gcerr.h	
Category:	GC_PARM_BLK utility	
Mode:	synchronous	

■ Description

The **gc_util_next_parm_ex( )** function is used to retrieve the next parameter element (relative to a specified current parameter element) from a GC_PARM_BLK in the form of a GC_PARM_DATA_EXT data structure. Calling this function repetitively and passing a pointer to the GC_PARM_DATA_EXT structure that was returned by the previous call allows an application to sequentially retrieve all of the parameter elements in a GC_PARM_BLK. To begin retrieving parameter elements at the beginning of the GC_PARM_BLK, the application passes a pointer to a GC_PARM_DATA_EXT structure that it has just initialized by calling **INIT_GC_PARM_DATA_EXT(parm)**.

This function **must** be used instead of **gc_util_next_parm**( ) if the parameter value can potentially exceed 255 bytes. This function is backward compatible and can be used instead of **gc_util_next_parm**( ) for any GC_PARM_BLK, regardless of whether the parameter block contains setID/parmID pairs that support values longer than 255 bytes.

The parameters that currently support extended-length values include:

- IPSET_MIME (or IPSET_MIME_200OK_TO_BYE) / IPPARM_MIME_PART_HEADER
- IPSET_MIME (or IPSET_MIME_200OK_TO_BYE) / IPPARM_MIME_PART_TYPE
- IPSET_NONSTANDARDCONTROL / IPPARM_NONSTANDARDDATA_DATA
- IPSET_NONSTANDARDDATA / IPPARM_NONSTANDARDDATA_DATA
- IPSET_SDP / all four parameter IDs (supported in 3PCC operating mode only)
- IPSET_SIP_MSGINFO / IPPARM_SIP_HDR
- IPSET_TUNNELEDSIGNALMSG / IPPARM_TUNNELEDSIGNALMSG_DATA

The **gc_util_next_parm_ex**( ) function updates the data structure referenced by the **parm** pointer and returns GC_SUCCESS if there is another parameter element in the GC_PARM_BLK following the element that was identified in the function call. If the current parameter data structure referenced by **parm** identifies the last parameter element in the GC_PARM_BLK, the next function call returns EGC_NO_MORE_PARMS.

Parameter	Description
parm_blk	points to the valid GC_PARM_BLK structure where data is stored
parm	pointer to a valid GC_PARM_DATA_EXT structure provided by the application. If the pointer that is passed in the function call refers to a structure that was just initialized with **INIT_GC_PARM_DATA_EXT(parm)**, the function retrieves the first parameter element in the GC_PARM_BLK. If the passed pointer references a structure that contains data from a previously found parameter element, the function retrieves the next parameter element in the block (if any). When the function completes successfully, the GC_PARM_DATA_EXT structure is updated to contain the retrieved information for the parameter element.

■ Cautions

Unlike the similar **gc_util_next_parm**( ) function, the **parm** pointer used in this function *cannot* be used to update the parameter itself because it references a data structure that is in the application's memory rather than pointing to a location within the GC_PARM_BLK itself.

■ Errors

- If this function returns GC_ERROR to indicate failure, use the **gc_ErrorInfo**( ) function to retrieve the reason for the error. See the "Error Handling" section in the *Dialogic® Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.
- The **parm** parameter must point to a valid GC_PARM_DATA_EXT structure. If it is desired to search from the beginning of the parameter block, the application **must** initialize the structure via **INIT_GC_PARM_DATA_EXT(parm)** before calling **gc_util_next_parm_ex**( ).

■ Example

```
#include "gclib.h"
#include "gcip.h"

void process_parm_block(GC_PARM_BLKP pparm_blk)
{
   GC_PARM_DATA_EXT parm_data_ext;
   int ret = 0;

   /* Initialize this structure for two reasons:
    * 1. To retrieve the first parameter in the parm block
    * 2. The first time this structure is used it must be initialized
    */
   INIT_GC_PARM_DATA_EXT(&parm_data_ext);

   /* Loop to retrieve all of the parameters and associated data from the GC_PARM_BLK
    */
   while ( GC_SUCCESS == (ret = gc_util_next_parm_ex( pparm_blk, &parm_dat_ext)) )
   {
      /* Process set_ID/parm_ID pairs */
      switch(parm_data_ext.set_ID);
```

```
            {
               .
               .
               .
            }
         }

         /* Check for error */
         if ( GC_ERROR == ret )
         {
            /* Process error */
         }

         .
         .
         .
      }
```

■ See Also

- **gc_util_find_parm_ex( )**

INIT_GC_PARM_DATA_EXT( )

Name:	void INIT_GC_PARM_DATA_EXT(pData)	
Inputs:	GC_PARM_DATA_EXT *pData	• pointer to the structure to be initialized
Returns:	None	
Includes:	gcip.h	
Mode:	synchronous	

■ Description

The **INIT_GC_PARM_DATA_EXT**( ) function is used to initialize a GC_PARM_DATA_EXT data structure, which is used when retrieving parameter elements from the metaevent data associated with many Global Call events using **gc_util_find_parm_ex**( ) and **gc_util_next_parm_ex( )** functions. These functions use the GC_PARM_DATA_EXT structure in order to handle extended-length parameter values (>255 bytes), but always use this structure regardless of the actual length of the parameter value.

Applications **must** use this function to initialize the GC_PARM_DATA_EXT structure before calling **gc_util_find_parm_ex**( ) or before the initial call to **gc_util_next_parm_ex**( ).

Parameter	Description
pData	points to the GC_PARM_DATA_EXT structure to be initialized

■ Cautions

Failure to use this function to initialize the GC_PARM_DATA_EXT structure before calling **gc_util_find_parm_ex**( ) or before the initial call to **gc_util_next_parm_ex**( ) may cause an operational error.

■ Example

```
#include "gclib.h"
#include "gcip.h"

void process_parm_block(GC_PARM_BLKP pparm_blk)
{
   GC_PARM_DATA_EXT parm_data_ext;
   int ret = 0;

   /* Initialize this structure for two reasons:
    * 1. To retrieve the first parameter in the parm block
    * 2. The first time this structure is used it must be initialized
    */
   INIT_GC_PARM_DATA_EXT(&parm_data_ext);

   /* Loop to retrieve all of the parameters and associated data from the GC_PARM_BLK
    */
   while ( GC_SUCCESS == (ret = gc_util_next_parm_ex( pparm_blk, &parm_dat_ext)) )
   {
      /* Process set_ID/parm_ID pairs */
```

```
         switch(parm_data_ext.set_ID);
         {
            .
            .
            .
         }
      }

      /* Check for error */
      if ( GC_ERROR == ret )
      {
         /* Process error */
      }

      .
      .
      .
   }
```

■ See Also

- GC_PARM_DATA_EXT reference page

INIT_IP_VIRTBOARD( )

Name:	void INIT_IP_VIRTBOARD(pIpVb)	
Inputs:	IP_VIRTBOARD *pIpVb	• pointer to the structure to be initialized
Returns:	None	
Includes:	gcip.h	
Mode:	synchronous	

■ Description

The **INIT_IP_VIRTBOARD( )** function is used to initialize an IP_VIRTBOARD data structure, which contains configuration data for a specific virtual IPT board. This function must be called to initialize an IP_VIRTBOARD structure for each virtual board that will be defined by calling **INIT_IPCCLIB_START_DATA( )** before calling **gc_Start( )**.

After the structure is initialized, an application can overwrite any of the defeat values as appropriate to the specific requirements. Among the items controlled by the IP_VIRTBOARD structure and initialized by this function are:

- maximum number of calls (total, H.323, and SIP)
- local IP address and signaling ports for H.323 and SIP
- H.323 Terminal Type (default is Gateway)
- enable access to H.323 message information fields (default is disabled)
- enable call transfer supplementary service (default is disabled)
- enable access to SIP message header fields and MIME-encoded message bodies (default is access disabled for both headers and MIME bodies)
- enable and configure a SIP outbound proxy (default is disabled)
- enable and configure TCP transport for SIP requests (default is disabled)
- configure SIP request retry behavior (default enables all allowable retries)
- enable application access to SIP OPTIONS requests (default is disabled)
- configure maximum number of SIP registrations (default equals max. number of SIP calls)

Parameter	Description
pIpVb	points to the IP_VIRTBOARD data structure to be initialized. See IP_VIRTBOARD, on page 665, for information on the default values and optional values that may be after initialization.

■ Cautions

None.

Example

```
IP_VIRTBOARD ip_virtboard[2];
IPCCLIB_START_DATA ipcclibstart;
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
ip_virtboard[1].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
```

See Also

- **INIT_IPCCLIB_START_DATA( )**
- Section 8.3.27, “gc_Start( ) Variances for IP”, on page 590
- IP_VIRTBOARD, on page 665

INIT_IPCCLIB_START_DATA( )

Name:	void INIT_IPCCLIB_START_DATA(pIpStData, numBoards, pIpVb)	
Inputs:	IPCCLIB_START_DATA *pIpStData	• pointer to the structure to be initialized
	unsigned char numBoards	• number of boards
	IP_VIRTBOARD *pIpVb	• pointer to an array of IP_VIRTBOARD structures
Returns:	None	
Includes:	gcip.h	
Mode:	synchronous	

■ Description

The **INIT_IPCCLIB_START_DATA( )** function is used to initialize an IPCCLIB_START_DATA data structure, which contains configuration information on the virtual IPT boards to be started via **gc_Start( )**. All fields are set to default values described in IPCCLIB_START_DATA, on page 671

Applications **must** use this function to initialize the IPCCLIB_START_DATA structure before calling **gc_Start( )**.

Parameter	Description
pIpStData	points to the IPCCLIB_START_DATA structure to be initialized
numBoards	the number of virtual IPT boards being defined (up to a maximum of 8)
pIpVb	points to an array of IP_VIRTBOARD data structures, one for each virtual IPT board being defined

■ Cautions

None.

■ Example

```
IP_VIRTBOARD ip_virtboard[2];
IPCCLIB_START_DATA ipcclibstart;
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ip_virtboard[0].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
ip_virtboard[1].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
```

■ See Also

- **INIT_IP_VIRTBOARD( )**
- Section 8.3.27, “gc_Start( ) Variances for IP”, on page 590

8.3 Dialogic® Global Call API Function Variances for IP

Note: Except for **gc_Listen( )**, **gc_OpenEx( )**, **gc_ReleaseCallEx( )**, **gc_UnListen( )**, all Dialogic® Global Call API functions that nominally support synchronous and asynchronous mode are supported **only in asynchronous mode** when using the IP technology.

The Dialogic® Global Call API function variances that apply when using IP technology are described in the following sections. See the *Dialogic® Global Call API Library Reference* for generic (technology-independent) descriptions of the Dialogic® Global Call API functions.

8.3.1 gc_AcceptCall( ) Variances for IP

This function is only supported in asynchronous mode.

The **rings** parameter is ignored.

Variance for H.323

The **gc_AcceptCall( )** function is used to send the Q.931 ALERTING message to the originating endpoint.

In addition to the ALERTING message, the library also generates a Q.931 PROGRESS message.

Variance for SIP

The **gc_AcceptCall( )** function is used to send a SIP informational response message to the originating endpoint. This message will generally be either 180 Ringing or 183 Session Progress, but the Dialogic® Global Call API library permits any response code in the range 101-199 to be specified for accept call responses on a given board device or on a per call level. (The 100 Trying response code is not permitted because it is already mapped to the **gc_CallAck( )** function and GCEV_PROCEEDING event.) If the application does not specify a particular response code for call accept messages, 180 Ringing is used by default.

To set the SIP response code, the application invokes **gc_SetConfigData( )** for a board device or **gc_SetUserInfo( )** for a given call with the following parameter:

IPSET_SIP_RESPONSE_CODE
    IPPARM_ACCEPT_RESP_CODE
    - value = unsigned short between 101 and 199

Note: The board level default setting is used when the response message is not explicitly set for a given call.

The following code snippet shows how to set SIP informational response for a board device and for a call.

```
void SetSIPAcceptResponseForBoard(long board, long ProgressAccept) // 180 or 183
{
   GC_PARM_BLK *pParmBlockA = NULL;
   int rc;

   GC_INFO errorinfo;

   long request_id = 0;
   long boardDev;

   char boardstr[255];

   sprintf(boardstr, ":N_iptB%d:P_IP", board);

   rc = gc_OpenEx(&boardDev, boardstr, EV_ASYNC, NULL);

   Sleep(10);

   rc = gc_util_insert_parm_val(&pParmBlockA,
                IPSET_SIP_RESPONSE_CODE,
                IPPARM_ACCEPT_RESP_CODE,
                sizeof(unsigned long),
                ProgressAccept);
   if ( rc != 0 )
   {
         gc_ErrorInfo(&errorinfo);
   }
   rc = gc_SetConfigData(GCTGT_CCLIB_NETIF, boardDev, pParmBlockA, 0,
        GCUPDATE_IMMEDIATE, &request_id, EV_ASYNC);
   if (rc != GC_SUCCESS)
   {
        /* handle error */
   }
   if ( rc != 0 )
   {
         gc_ErrorInfo(&errorinfo);
   }
   gc_Close(boardDev);
}


void SetSIPAcceptResponse(long target_id, long ProgressAccept) // 180 or 183
{
  GC_PARM_BLK *pParmBlockA = NULL;
  int rc;

  GC_INFO errorinfo;

  rc = gc_util_insert_parm_val(&pParmBlockA,
               IPSET_SIP_RESPONSE_CODE,
               IPPARM_ACCEPT_RESP_CODE,
               sizeof(unsigned long),
               ProgressAccept);
  if ( rc != 0 )
  {
          gc_ErrorInfo(&errorinfo);
  }
  rc = gc_SetUserInfo(GCTGT_GCLIB_CHAN, target_id, pParmBlockA, GC_ALLCALLS );

  if ( rc != 0 )
  {
          gc_ErrorInfo(&errorinfo);
  }
}
```

~~The following code example shows how to set the call accept response code to 183 Session Progress instead of the default 180 Ringing **on a board device**:~~

```
~~    .~~
~~    .~~
~~    .~~
~~int              rc =  GC_SUCCESS;~~
~~GC_PARM_BLK *    parmblkp = NULL;~~
~~unsigned short   acceptCode = 183;   /* Session Progress*/~~
~~    .~~
~~    .~~
~~    .~~
~~/* Append/create GC_PARM_BLK with specified 183 response code*/~~
~~gc_util_insert_parm_val(&parmblkp,~~
~~                                 IPSET_SIP_RESPONSE_CODE,~~
~~                                 IPPARM_ACCEPT_RESP_CODE,~~
~~                                 sizeof(unsigned short, &acceptCode);~~

~~rc = gc_SetConfigData(GCTGT_CCLIB_NETIF, boardDev, parmblkp, 0,~~
~~                              GCUPDATE_IMMEDIATE, &request_id, EV_ASYNC);~~
~~if (rc != GC_SUCCESS)~~
~~{~~
~~    /* handle error */~~
~~}~~
~~    .~~
~~    .~~
~~    .~~
```

8.3.2 gc_AcceptInitXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

Variance for H.323 (H.450.2)

Either the rerouting_num (of type char*) or rerouting_addrblkp (of type GCLIB_ADDRESS_BLK*) fields of the GC_REROUTING_INFO structure can be used to specify the rerouting address string to be signaled back to party A and its final destination to party B. The sub_address fields of the GCLIB_ADDRESS_BLK are ignored and not used.

Note: If both fields are used, the rerouting address string will be a concatenation of the information from both fields.

The GCEV_ACCEPT_INIT_XFER event is received by the application on the secondary/consultation call CRN once the transferred call is received as notified via the GCEV_OFFERED event.

If the call transfer is abandoned by parties A or B before the transfer is completed, the GCEV_ACCEPT_INIT_XFER_FAIL event is received with a CCLIB cause value of IPEC_H4502CTAbandon and a Dialogic® Global Call API cause value of GCRV_CALLABANDONED.

If the CTT2 timer (20 seconds) expires before the transfer is completed, the GCEV_ACCEPT_INIT_XFER_FAIL event is received with a CCLIB cause value of IPEC_H450CTT2Timeout and a Dialogic® Global Call API cause value of GCRV_TIMEOUT.

Variance for SIP

This function does not apply to SIP call transfer. In SIP, party A does not notify party C in advance of requesting an attended (supervised) transfer operation with **gc_InvokeXfer( )**, so there is no opportunity for party C to accept or reject the transfer at the initiation stage.

8.3.3 gc_AcceptXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

The **parmblkp** parameter is ignored for IP technology and should be set to NULL.

The **gc_AcceptXfer( )** function can be used at party B only after receiving a GCEV_REQ_XFER event. The application can obtain information on the rerouting number or address in a GC_REROUTING_INFO data structure dereferenced from the extevtdatap in the METAEVENT structure.

Both the rerouting_num (type char *) and the rerouting_addr (type GCLIB_ADDRESS_BLK) fields of the GC_REROUTING_INFO structure contain the same rerouting address string that was explicitly signaled from party A in SIP call transfers or H.450.2 blind call transfers, or from party C via **gc_AcceptInitXfer( )** in H.450.2 supervised call transfers. The rerouting number to be used in the subsequent **gc_MakeCall( )** at party B can be copied from either element, but must not be a concatenation of both elements because they each contain the same character string.

The remaining elements of the GCLIB_ADDRESS_BLK structure dereferenced from rerouting_addr contain the following:

address_type
GCADDRTYPE_IP

address_plan
GCADDRPLAN_UNKNOWN

sub_address
0 (unused)

sub_address_type
0 (unused)

sub_address_plan
0 (unused)

Variance for H.323 (H.450.2)

When party B (the Transferred party) accepts a transfer request via **gc_AcceptXfer( )** no notification is sent to party A (the Transferor or Transferring party). No message is sent to party A until the accepted transfer succeeds or fails.

Variance for SIP

When party B (Transferee or Transferred party) accepts a transfer request via **gc_AcceptXfer( )**, a 202 Accepted message and a NOTIFY(100 Trying) message with Subscription-State=Active is sent to party A (the Transferor or Transferring party). The call control library at party A may optionally generate a GCEV_INVOKE_XFER_ACCEPTED event to notify the application of the acceptance if that event has been enabled for that line device with **gc_SetConfigData( )**.

8.3.4 gc_AnswerCall( ) Variances for IP

This function is only supported in asynchronous mode.

The **rings** parameter is ignored.

Coders can be set in advance of using **gc_AnswerCall( )** by using **gc_SetUserInfo( )**. See Section 8.3.26, "gc_SetUserInfo( ) Variances for IP", on page 587 for more information.

The following code example shows how to use the **gc_SetUserInfo( )** function to set coder information before calls are answered using **gc_AnswerCall( )**.

```
/* Specifying coders before answering calls */
LINEDEV ldev;
CRN crn;
GC_PARM_BLK *target_datap;
/* Define Coder */
IP_CAPABILITY a_DefaultCapability;
gc_OpenEx(&ldev, ":N_iptB1T1:M_ipmB1C1:P_H323", EV_ASYNC, 0);

/* wait for GCEV_OPENEX event ... */

/* Set default coder for this ldev */
target_datap = NULL;
memset(&a_DefaultCapability,0,sizeof(IP_CAPABILITY));
a_DefaultCapability.capability = GCCAP_AUDIO_g7231_5_3k;
a_DefaultCapability.direction = IP_CAP_DIR_LCLTRANSMIT;
a_DefaultCapability.type = GCCAPTYPE_AUDIO;
a_DefaultCapability.extra.audio.frames_per_pkt = 1;
a_DefaultCapability.extra.audio.VAD = GCPV_DISABLE;
gc_util_insert_parm_ref(&target_datap, GCSET_CHAN_CAPABILITY,
IPPARM_LOCAL_CAPABILITY, sizeof(IP_CAPABILITY),
&a_DefaultCapability);

/* set both receive and transmit coders to be the same (since
   the IPTxxx board does not support asymmetrical coders */
memset(&a_DefaultCapability,0,sizeof(IP_CAPABILITY));
a_DefaultCapability.capability = GCCAP_AUDIO_g7231_5_3k;
a_DefaultCapability.direction = IP_CAP_DIR_LCLRECEIVE;
a_DefaultCapability.type = GCCAPTYPE_AUDIO;
a_DefaultCapability.extra.audio.frames_per_pkt = 1;
a_DefaultCapability.extra.audio.VAD = GCPV_DISABLE;
gc_util_insert_parm_ref(&target_datap, GCSET_CHAN_CAPABILITY,
IPPARM_LOCAL_CAPABILITY, sizeof(IP_CAPABILITY),
&a_DefaultCapability);

gc_SetUserInfo(GCTGT_GCLIB_CHAN, ldev, target_datap, GC_ALLCALLS);
gc_util_delete_parm_blk(target_datap);
gc_WaitCall(ldev, NULL, NULL, 0, EV_ASYNC);

/*... Receive GCEV_OFFERED ... */
```

```
/*... Retrieve crn from metaevent... */

gc_AnswerCall(crn, 0, EV_ASYNC);

/*... Receive GCEV_ANSWERED ... */
```

Variance for H.323

The **gc_AnswerCall( )** function is used to send the Q.931 CONNECT message to the originating endpoint.

Variance for SIP

The **gc_AnswerCall( )** function is used to send the 200 OK message to the originating endpoint.

8.3.5 gc_CallAck( ) Variances for IP

This function is only supported in asynchronous mode.

The **callack_blkp** parameter must be a pointer to a GC_CALLACK_BLK structure that contains a type field with a value of GCACK_SERVICE_PROC. The following code example shows how to set up a GC_CALLACK_BLK structure and issue the **gc_CallAck( )** function.

```
GC_CALLACK_BLK gcCallAckBlk;
memset(&gcCallAckBlk, 0, sizeof(GC_CALLACK_BLK));
gcCallAckBlk.type = GCACK_SERVICE_PROC;
rc = gc_CallAck(crn, &gcCallAckBlk, EV_ASYNC);
```

The application can configure whether the Proceeding message is sent manually using the **gc_CallAck( )** function or whether it is sent automatically by the stack. See Section 4.4.6, "Configuring Proceeding Message Generation (H.323)", on page 135 for more information.

Variance for H.323

The **gc_CallAck( )** function is used to send the Proceeding message to the originating endpoint.

Variance for SIP

The **gc_CallAck( )** function is used to send the 100 Trying message to the originating endpoint.

8.3.6 gc_Close( ) Variances for IP

Applications should avoid closing and re-opening devices multiple times. Board devices and channel devices should be opened during initialization and should remain open for the duration of the application.

8.3.7 gc_DropCall( ) Variances for IP

This function is only supported in asynchronous mode.

The **cause** parameter can be any of the generic cause codes documented in the **gc_DropCall( )** function reference page in the *Dialogic® Global Call API Library Reference* or a protocol-specific cause code as described below.

Variance for H.323

Allowable protocol-specific cause codes are prefixed by IPEC_H225 or IPEC_Q931 in Chapter 11, "IP-Specific Event Cause Codes".

Variance for SIP

Cause codes and reasons are only supported when **gc_DropCall( )** is issued while the call is in the Offered state. Allowable protocol-specific cause codes are prefixed by IPEC_SIP in Chapter 11, "IP-Specific Event Cause Codes".

Note: A Global Call application may not always receive a GCEV_DISCONNECTED event when terminating a call, because BYE messages are not retried if lost due to network errors.

8.3.8 gc_Extension( ) Variances for IP

This function is only supported in asynchronous mode.

The **gc_Extension( )** function can be used for the following purposes:

- retrieving call-related information
- getting notification of underlying protocol connection or disconnection state transitions
- getting notification of media streaming initiation and termination in both the transmit and receive directions [not supported in 3PCC operating mode]
- specifying which DTMF types, when detected, provide notification to the application [not supported in 3PCC operating mode]
- sending DTMF digits [not supported in 3PCC operating mode]
- retrieving protocol messages (Q.931, H.245, and registration)
- sending protocol messages (Q.931, H.245, and registration)
- performing T.38 fax server operations

Table 27 shows the valid extension IDs and their purpose.

Table 27. Valid Extension IDs for the gc_Extension( ) Function

Extension ID	Description
IPEXTID_CHANGEMODE	Used with **gc_Extension( )** for the following T.38 fax server operations: • initiating a switch from an audio session to a T.38 fax session • initiating a switch from a T.38 fax session to an audio session • accepting a request to switch from audio to T.38 fax or vice versa • rejecting a request to switch from audio to T.38 fax or vice versa Also used in GCEV_EXTENSION events to provide notification of incoming messages including: • a RequestMode (H.323) or re-INVITE (SIP) message indicating a request to switch from audio to T.38 fax • a RequestMode (H.323) or re-INVITE (SIP) message indicating a request to switch from T.38 fax to audio • a RequestModeAck (H.323) or 200 OK (SIP) message indicating that a switch to audio or T.38 fax has completed successfully See Section 4.38, “T.38 Fax Server”, on page 374 for more information. This extension ID is not supported in 3PCC operating mode.
IPEXTID_FOIP	Used in GCEV_EXTENSION events for notification of information related to fax. See Section 4.7.1, “Enabling and Disabling Unsolicited Notification Events”, on page 158 for more information. This extension ID is not supported in 3PCC operating mode.
IPEXTID_GETINFO	Used to retrieve call-related information. See Section 4.6, “Retrieving Current Call-Related Information”, on page 146 for more information.
IPEXTID_IPPROTOCOL_STATE	Used in GCEV_EXTENSION events for notification of intermediate protocol states, such as, Q.931 and H.245 session connections and disconnections. See Section 4.7.1, “Enabling and Disabling Unsolicited Notification Events”, on page 158 for more information.
IPEXTID_MEDIAINFO	Used in GCEV_EXTENSION events for notification of the initiation and termination of media streaming in the transmit and receive directions. In the case of media streaming connection notification, the datatype of the parameter is IP_CAPABILITY and consists of the coder configuration that resulted from the capability exchange with the remote peer. See Section 4.7.1, “Enabling and Disabling Unsolicited Notification Events”, on page 158 for more information. This extension ID is not supported in 3PCC operating mode.
IPEXTID_MSGINFO	Used in GCEV_EXTENSION events for receiving SIP messages with MIME-encoded information in the message body. See Section 4.10, “Sending and Receiving MIME Bodies in SIP Messages (SIP-T)”, on page 196, for more information. The supported parameter sets are: • IPSET_MIME • IPSET_MIME_200OK_TO_BYE
IPEXTID_RECEIVE_DTMF	Used to select which DTMF types, when detected, provide notification to the application. See Section 4.7.1, “Enabling and Disabling Unsolicited Notification Events”, on page 158 for more information. This extension ID is not supported in 3PCC operating mode.
IPEXTID_RECEIVED_18x_ RESPONSE	Used in GCEV_EXTENSION events for notification of 18x provisional responses. See Section 4.21, “Receiving Multiple SIP 18x Provisional Responses”, on page 287 for more information.

Table 27. Valid Extension IDs for the gc_Extension( ) Function

Extension ID	Description
IPEXTID_RECEIVEMSG	Used in GCEV_EXTENSION events when SIP, Q.931, H.245, and non-standard registration messages are received.
IPEXTID_SEND_DTMF	Used to send DTMF digits. When this call is successful, the sending side receives a GCEV_EXTENSIONCMPLT event with the same ext_id. The remote side receives a GCEV_EXTENSION event with IPEXTID_RECEIVE_DTMF but only when configured for notification of a specific type of DTMF. See Section 4.7.1, “Enabling and Disabling Unsolicited Notification Events”, on page 158 for more information. This extension ID is not supported in 3PCC operating mode.
IPEXTID_SENDMSG	Used to send SIP, H.245, Q.931, and RAS messages. When using this Extension ID, the first parameter inserted into the GC_PARM_BLK must be from one of the following parameter sets: • IPSET_MSG_H245 • IPSET_MSG_Q931 • IPSET_MSG_REGISTRATION • IPSET_MSG_SIP • IPSET_PROTOCOL When the **gc_Extension( )** function completes successfully, the sending side receives a GCEV_EXTENSIONCMPLT event with the same ext_id. The remote side receives a GCEV_EXTENSION event with an ext_id field value of IPEXTID_RECEIVEMSG.

The **gc_Extension**( ) function is only used in the context of a call where the protocol is already known, therefore the protocol does not need to be specified. When protocol-specific information is specified and it is not of the correct protocol type, for example, attempting to send a Q.931 FACILITY message in a SIP call, the operation fails.

See the Section 4.6.2, “Examples of Retrieving Call-Related Information”, on page 149 for a code example showing how to identify the type of extension event and extract the related information.

8.3.9 gc_GetAlarmParm( ) Variances for IP

The **gc_GetAlarmParm**( ) function can be used to get QoS threshold values. The function parameter values in this context are:

linedev
: The media device handle, retrieved using the **gc_GetResourceH**( ) function. See Section 4.26.2, “Retrieving the Media Device Handle”, on page 310 for more information.

aso_id
: The alarm source object ID. Set to ALARM_SOURCE_ID_NETWORK_ID.

ParmSetID
: Must be set to ParmSetID_qosthreshold_alarm.

alarm_parm_list
: A pointer to an ALARM_PARM_FIELD structure. The alarm_parm_number field is not used. The alarm_parm_data field is of type GC_PARM, which is a union. In this context, the type used is void *pstruct, and is cast as a pointer to an IPM_QOS_THRESHOLD_INFO structure,

which includes an IPM_QOS_THRESHOLD_DATA structure that contains the parameters representing threshold values. See the IPM_QOS_THRESHOLD_INFO structure in the *Dialogic® IP Media Library API Library Reference* and the *Dialogic® IP Media Library API Programming Guide* for more information. The thresholds supported by Dialogic® Global Call API for HMP include:

- QOSTYPE_LOSTPACKETS
- QOSTYPE_JITTER
- QOSTYPE_RTCPTIMEOUT
- QOSTYPE_RTPTIMEOUT.

mode
: Must be set to EV_SYNC.

Note: Applications **must** include the *gcipmlib.h* header file before Dialogic® Global Call API can be used to set or retrieve QoS threshold values.

See Section 4.26.3, “Setting QoS Threshold Values”, on page 310 for code examples.

8.3.10 gc_GetCallInfo( ) Variances for IP

The **gc_GetCallInfo( )** function can be used to retrieve calling (ANI) or called party (DNIS) information such as an IP address, an e-mail address, an E.164 number, a URL, or the call identifier (Call ID) used by the underlying protocol to globally, uniquely identify the call. The values of the **info_id** parameter that are supported for both SIP and H.323 are:

ORIGINATION_ADDRESS
: the calling party information (equivalent to ANI)

DESTINATION_ADDRESS
: the called party information (equivalent to DNIS)

IP_CALLID
: the globally unique identifier used by the underlying protocol to identify the call (Call ID or GUID)

Two additional, SIP-specific values for the **info_id** parameter that allow retrieval of information from the From URI and To URI SIP message fields are described below under the “Variance for SIP” heading.

When an **info_id** of ORIGINATION_ADDRESS (ANI) is specified and the function completes successfully, the **valuep** string is a concatenation of values delimited by a pre-determined character. (The delimiter character is configurable in the IPCCLIB_START_DATA data structure that is used by **gc_Start( )**; the default character is a comma.)

When an **info_id** of DESTINATION_ADDRESS (DNIS) is specified and the function completes successfully, the **valuep** string is a concatenation of values delimited by a pre-determined character. (The delimiter character is configurable in the IPCCLIB_START_DATA data structure that is used by **gc_Start( )**; the default character is a comma.) The IP address of the destination gateway (that is processing the DNIS) is **not** included in the string.

When an **info_id** of IP_CALLID (Call ID) is specified and the function completes successfully, the buffer pointed to by the **valuep** argument contains the globally unique identifier used by the underlying protocol to identify the call. The size and datatype of the Call ID depends on the protocol. To assure adequate buffer size when the protocol is unknown, use the IP_CALLIDSIZE define to allocate a buffer that is large enough to hold any type of Call ID value (i.e., either an H.323 array of octets or a SIP string).

Note: For outbound calls the **gc_GetCallInfo( )** function can be used to retrieve valid Call ID information only after the Proceeding state.

The **gc_GetCallInfo( )** function can also be used to query the protocol used by a call. The **info_id** parameter should be set to CALLPROTOCOL and the **valuep** parameter returns a pointer to an integer that is one of the following values:

- CALLPROTOCOL_H323
- CALLPROTOCOL_SIP

Note: For an inbound call, the **gc_GetCallInfo( )** function can be used to determine the protocol any time after the GCEV_OFFERED event is received and before the GCEV_DISCONNECTED event is received.

Variance for H.323

When retrieving calling (ANI) information, the following rules apply. Any section in the string that includes a prefix (TA:, TEL:, or NAME:) has been inserted as an alias by the originating party. Any section in the string that does not include a prefix has been inserted as a **calling party** number (Q.931) by the originating party.

When retrieving called party (DNIS) information, the following rules apply. Any section in the string that includes a prefix (TA:, TEL:, or NAME:) has been inserted as an alias by the originating party. Any section in the string that does not include a prefix has been inserted as a **called party** number (Q.931) by the originating party.

When retrieving Call ID information, the buffer pointed to by the **valuep** argument contains an array of octets. The size of this array is IP_H323_CALLIDSIZE bytes. To assure adequate buffer size when the protocol is unknown, use the IP_CALLIDSIZE define to create a buffer that is large enough to hold any type of Call ID value (i.e., for either H.323 or SIP).

Variance for SIP

When retrieving calling party (ANI) or called party (DNIS) information, prefixes (such as TA:, TEL:, and NAME:) are **not** used.

When retrieving calling party (ANI) information, the address is taken from the SIP From: header, and is accessible in one of two forms by using one of the following parameter IDs in the function call:

ORIGINATION_ADDRESS
    Returns the simple origination address in the form
        alice@192.168.1.10

ORIGINATION_ADDRESS_SIP

Returns a SIP-specific origination address that includes additional From URI parameters and tags. The format used is

sip: alice@192.168.1.10;tag=0-13c4-4059c361-23d07406-72fe

When retrieving called party (DNIS) information, the address is taken from the SIP To: header, and is accessible in one of two forms by using one of the following parameter IDs in the function call:

DESTINATION_ADDRESS

Returns the simple destination address in the form

user@127.0.0.1

DESTINATION_ADDRESS_SIP

Returns a SIP-specific destination address that includes additional To URI parameters in the form

sip: userB@127.0.0.1;user=Steve

When retrieving Call ID information, the buffer pointed to by the **valuep** argument contains a NULL-terminated string. The maximum size of this string is IP_SIP_CALLIDSIZE bytes. To assure adequate buffer size when the protocol is unknown, use the IP_CALLIDSIZE define. This will assure the buffer is large enough to hold any type of Call ID value (i.e., either H.323 or SIP).

Retrieving SIP Call ID via gc_GetCallInfo( )

The following code example illustrates retrieval of the SIP Call ID using a **gc_GetCallInfo( )** call.

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_iptB1T1:P_SIP, :N_iptB1T2:P_SIP, etc...)
 * 2. gc_WaitCall() has been issued to wait for a call.
 * 3. gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows) has been called
 *    to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>

/*
 * Assume the 'crn' parameter holds the CRN associated with the detected GCEV_OFFERED event.
 */

int print_call_info(CRN crn)
{
    GC_INFO gc_error_info;             /* GlobalCall error information data */
    char cid_buff[IP_SIP_CALLIDSIZE];  /* buffer large enough to hold SIP Call-ID value */

    if(gc_GetCallInfo(crn, IP_CALLID, cid_buff) != GC_SUCCESS)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetCallInfo(IP_CALLID) on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,"\
                " CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gcMsg, gc_error_info.ccLibId,
                gc_error_info.ccLibName, gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    printf ("gc_GetCallInfo(IP_CALLID) on crn: 0x%lx, returned - %s\n", crn, cid_buff);

    return(0);
}
```

8.3.11 gc_GetCTInfo( ) Variances for IP

The **gc_GetCTInfo( )** function can be used to retrieve product information (via the CT_DEVINFO structure) for the media sub-device (ipm) attached to the network device (ipt). If no media device is associated with the network device, the function returns as though not supported.

8.3.12 gc_GetResourceH( ) Variances for IP

The **gc_GetResourceH( )** function can be used to retrieve the media device (ipm device) handle, which is required by GCAMS functions, such as, **gc_SetAlarmParm( )** and **gc_GetAlarmParm( )** to set and retrieve QoS threshold values. The function parameter values in this context are:

linedev
: the network device, that is, the Dialogic® Global Call API line device retrieved by the **gc_OpenEx( )** function

resourcehp
: the address where the media device handle is stored when the function completes

resourcetype
: GC_MEDIADEVICE

Note: Applications **must** include the *gcipmlib.h* header file before Dialogic® Global Call API can be used to set or retrieve QoS threshold values.

The other resource types including GC_NETWORKDEVICE (for a network device), GC_VOICEDEVICE (for a voice device), and GC_NET_GCLINEDEVICE (to retrieve the Dialogic® Global Call API line device handle when the media handle is known) are also supported.

Note: The GC_VOICEDEVICE option above applies only if the voice device was opened with the line device or opened separately and subsequently attached to the line device.

8.3.13 gc_GetXmitSlot( ) Variances for IP

The **gc_GetXmitSlot( )** function can be used to get the transmit time slot information for an IP Media device. The function parameter values in this context are:

linedev
: The Dialogic® Global Call API line device handle for an IP device (that is, the handle returned by **gc_OpenEx( )** for a device with :N_iptBxTy in the **devicename** parameter and a media device attached).

sctsinfop
: A pointer to the transmit time slot information for the IP Media device (a pointer to a CT Bus time slot information structure).

8.3.14 gc_InitXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

The **parmblkp** and **ret_rerouting_infopp** parameters are ignored and should be set to NULL. The **gc_InitXfer(**) function returns -1 if invalid parameter are specified.

Variance for H.323 (H.450.2)

The **gc_InitXfer(**) function has an associated GCEV_INIT_XFER termination event that is received on the specified CRN. This termination event indicates that the initiate transfer request was successful and that party C has sent a positive acknowledgement.

Variance for SIP

The **gc_InitXfer(**) function does not cause any SIP message to be sent to either of the remote parties, and is used only for purposes of synchronizing the Global Call state machine. The GCEV_INIT_XFER termination event that the Transferor receives on the specified CRN after calling **gc_InitXfer(**) is a “dummy” event whose only purpose is to allow synchronization of the Global Call state machine.

8.3.15 gc_InvokeXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

Variance for H.323 (H.450.2)

The party A application is notified by GCEV_INVOKE_XFER_REJ if the remote party receiving the call transfer request rejects the request, or by GCEV_INVOKE_XFER_FAIL if the request fails for some reason, but there is **no** notification if the request is accepted. The only notification party A receives in a successful transfer is the GCEV_INVOKE_XFER event, which does not necessarily mean that the transferred call between party B and party C was connected, only that it was confirmed to be delivered. Specifically, it indicates that ALERTING or CONNECT was received from party C on the transferred call.

Table 28 identifies the protocol-specific variances in parameters for **gc_InvokeXfer(**).

Table 28. gc_InvokeXfer( ) Supported Parameters for H.450.2

Parameter	Meaning
crn	For all transfers, CRN of primary call.
extracrn	For a supervised call transfer, parameter value must be the CRN of the secondary/consultation call with party C. For blind call transfers, parameter value must be zero.
numberstr	Ignored in supervised call transfer – set to NULL. For blind call transfer, used to provide address of party C (the rerouting address) as a string. Signaled to party B in the GCEV_REQ_XFER event. Format can be: • transport address, for example, "TA:146.152.0.1" • E.164 alias, for example, "TEL:9739933000" • host address, for example, "NAME: myhostname" **Note:** The prefix must be included in the string to allow correct interpretation. **Note:** When using the GC_MAKECALL_BLK ***makecallp** parameter to specify the rerouting address via a data structure, this parameter must be set to NULL.
makecallp	Ignored in supervised call transfer – set to NULL. For blind call transfer, used to provide address of party C (the rerouting address) in a GC_MAKECALL_BLK data structure. Signaled to party B in the GCEV_REQ_XFER event. **Note:** When using the char ***numberstr** parameter to specify the rerouting address as a string, this parameter must be set to NULL.
timeout	Ignored. H.450.2 timers (T1, T2, T3, T4) are implicitly maintained at 20 seconds – set to zero.

Table 29 through Table 32 list the possible event failure cause values.

Table 29. H.450.2 ctInitiate Errors Received from the Network

ctInitiate Error	Result Values	GC Event
notAvailable	CC: IPEC_H450NotAvailable GC: GCRV_REMOTEREJ_UNAVAIL	GCEV_INVOKE_XFER_REJ
invalidCallState	CC: IPEC_H450InvalidCallState GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_INVOKE_XFER_FAIL
invalidReroutingNumber	CC: IPEC_H4502InvalidReroutingNumber GC: GCRV_REMOTEREJ_INVADDR	GCEV_INVOKE_XFER_REJ
unrecognizedCallIdentity	CC: IPEC_H4502UnrecognizedCallIdentity GC: GCRV_REMOTEREJ_INVADDR	GCEV_INVOKE_XFER_FAIL
establishmentFailure	CC: IPEC_H4502EstablishmentFailure GC: GCRV_REMOTEREJ_UNSPECIFIED	GCEV_INVOKE_XFER_FAIL
supplementaryService InteractionNotAllowed	CC: IPEC_H450SuppServInteractionNotAllowed GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_INVOKE_XFER_REJ
unspecified	CC: IPEC_H4502Unspecified GC: GCRV_REMOTEREJ_UNSPECIFIED	GCEV_INVOKE_XFER_REJ

Table 30. H.450.2 ctIdentify Errors Received From the Network

ctIdentify Error	Result Values	GC Event
notAvailable	CC: IPEC_H450TRTSENotAvailable GC: GCRV_REMOTEREJ_UNAVAIL	GCEV_ INVOKE_XFER_REJ
invalidCallState	CC: IPEC_H450TRTSEInvalidCallState GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_ INVOKE_XFER_FAIL
supplementaryService InteractionNotAllowed	CC: IPEC_H450TRTSESuppServInteractionNotAllowed GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_ INVOKE_XFER_REJ
unspecified	CC: IPECH4502TRTSEUnspecified GC: GCRV_REMOTEREJ_UNSPECIFIED	GCEV_ INVOKE_XFER_REJ

Table 31. H.450.2 ctSetup Errors Received From the Network

ctSetup Error	Result Values	GC Event
notAvailable	CC: IPEC_H450NotAvailable GC: GCRV_REMOTEREJ_UNAVAIL	GCEV_INVOKE_XFER_REJ
invalidCallState	CC: IPEC_H450InvalidCallState GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_INVOKE_XFER_FAIL
invalidReroutingNumber	CC: IPEC_H4502InvalidReroutingNumber GC: GCRV_REMOTEREJ_INVADDR	GCEV_INVOKE_XFER_REJ
unrecognizedCallIdentity	CC: IPEC_H4502UnrecognizedCallIdentity GC: GCRV_REMOTEREJ_INVADDR	GCEV_INVOKE_XFER_FAIL
supplementaryService InteractionNotAllowed	CC: IPEC_H450SuppServInteractionNotAllowed GC: GCRV_REMOTEREJ_NOTALLOWED	GCEV_INVOKE_XFER_REJ
unspecified	CC: IPEC_H4502Unspecified GC: GCRV_REMOTEREJ_UNSPECIFIED	GCEV_INVOKE_XFER_REJ

Table 32. H.450.2 CT Timer Expiry

Endpoint – Timer	Result Values	GC Event
TRGSE – T1	CC: IPEC_H450CTT1Timeout GC: GCRV_TIMEOUT	GCEV_INVOKE_XFER_FAIL
TRGSE – T3	CC: IPEC_H450CTT3Timeout GC: GCRV_TIMEOUT	GCEV_INVOKE_XFER_FAIL

Variance for SIP

The application at party A may optionally be notified by a GCEV_INVOKE_XFER_ACCEPTED event that the transfer request has been accepted by the remote party to which it was sent. (This event has no equivalent in H.450.2.) This event is optional, and is disabled by default. The event may be enabled and disabled on a per-line-device basis via the **gc_SetConfigData( )** function as shown in the following code example.

```
//enable GCEV_INVOKE_XFER_ACCEPTED event for SIP call transfer
GC_PARM_BLK *t_pParmBlk = NULL;
long        request_id;

gc_util_insert_parm_val(&t_parmBlkl, GCSET_CALLEVENT_MSK, GCACT_ADDMSK,
                        sizeof(long), GCMSK_INVOKE_XFER_ACCEPTED);

gc_SetConfigData(GCTGT_GCLIB_CHAN,ldev,t_pParmBlk,0,GCUPDATE_IMMEDIATE,&request_id,EV_SYNC);

gc_util_delete_parm_blk(t_pParmBlk)
```

The specific meaning of the GCEV_INVOKE_XFER termination event for successful transfers is dependent on the application and the transfer scenario(s) it uses. The possible outcomes when Global Call is used by all parties include the following:

- If party A drops the primary call in unattended transfers before the transfer completes, party A does not receive any GCEV_INVOKE_XFER event at all.
- If party B drops the primary call in unattended transfers before the transfer completes, party A receives a GCEV_INVOKE_XFER event that only signifies that party B has sent INVITE to party C.
- For attended transfers or unattended transfers where the primary call is maintained during the transfer, party A receives a GCEV_INVOKE_XFER event which indicates that the transferred call was actually connected between party B and party C.

Table 33 identifies the protocol-specific variances in parameters for **gc_InvokeXfer( )**.

Table 33. gc_InvokeXfer( ) Supported Parameters for SIP

Parameter	Meaning
crn	The CRN of the call between party A and the remote party receiving the transfer request. This is the primary call in an unattended (blind) call transfer, but may be either call for an attended (supervised) transfer.
extracrn	For an attended (supervised) call transfer, the CRN of the call between party A and the remote party *not* receiving the transfer request (i.e. the call not specified in the **crn** parameter). For unattended (blind) call transfers, must be zero.
numberstr	For attended (supervised) call transfers, this parameter is ignored. Set to NULL. For an unattended (blind) call transfer, the address of party C (the rerouting address, which will be signaled to party B) as a string. This address is of the form `user@host; param=value` where • `user` is a user name or phone number • `host` is a domain name or IP address • `param=value` is an optional additional parameter For additional information on rules for destination addresses, see Section 8.3.17.3, "Forming a Destination Address String", on page 564 under the "Variance for SIP" heading. **Note:** When using the GC_MAKECALL_BLK ***makecallp** parameter to specify the rerouting address, this parameter must be set to NULL.

Table 33. gc_InvokeXfer( ) Supported Parameters for SIP (Continued)

Parameter	Meaning
makecallp	For attended (supervised) call transfers, this parameter is Ignored. Set to NULL. For an unattended (blind) call transfer, the address of party C (the rerouting address, which will be signaled to party B) as a GC_MAKECALL_BLK data structure. **Note:** When using the char ***numberstr** parameter to specify the rerouting address, this parameter must be set to NULL.
timeout	Ignored. Set to NULL.

The application may optionally set the specific information in the header fields of the SIP REFER message that is sent by this function by configuring a GC_PARM_BLK before calling **gc_InvokeXfer( )**, as described in Section 4.9, "Setting and Retrieving SIP Message Header Fields", on page 174. Table 34 lists the header fields that can be set in REFER messages and the corresponding parameter IDs along with examples of field values.

Table 34. SIP Header Fields Settable in REFER Messages

Field Name	GC Parameter ID (Set ID: IPSET_SIP_MSGINFO)	Example Field Value
Request URI	IPPARM_REQUEST_URI	`146.152.212.67:5060`
From	IPPARM_FROM	`From: Transferor <sip:146.152.212.43>;tag=0-13c4-408c7921-1026900f-ed5;myname`
To	IPPARM_TO	`To: Transferee <sip:146.152.212.67:5060>;tag=0-13c4-408c7921-10268fdd-6a19`
From Display	IPPARM_FROM_DISPLAY	`Transferor`
To Display	IPPARM_TO_DISPLAY	`Transferee`
Call ID	IPPARM_CALLID_HDR	`48cabd0-0-13c4-408c7921-10268fdd-1563@146.152.212.67`
Contact URI	IPPARM_CONTACT_URI	`sip:146.152.212.43`
Contact Display	IPPARM_CONTACT_DISPLAY	`Transferor`
Referred-By	IPPARM_REFERRED_BY	`Referred-By: <sip:146.152.212.43>`
Replaces	IPPARM_REPLACES	`Replaces: 48cae78-0-13c4-408c7923-1026947b-1078@146.152.212.67;to-tag=0-13c4-408c7923-102694a3-6942;from-tag=0-13c4-408c7923-1026947b-7b6`

8.3.16 gc_Listen( ) Variances for IP

The **gc_Listen( )** function is supported in both synchronous and asynchronous modes. The function is blocking in synchronous mode.

Note: For line devices that comprise media (ipm) and voice (dxxx) devices, routing is only done on the media devices. Routing of the voice devices must be done using the Voice API (dx_ functions).

8.3.17 gc_MakeCall( ) Variances for IP

This function is only supported in asynchronous mode.

The Dialogic® Global Call API supports multiple IP protocols on a single IPT Network device. See Section 2.3.3, "IPT Network Devices", on page 40 for more information. When using a multi-protocol network device (that is, one opened in P_IP mode), the application specifies the protocol in the associated GC_MAKECALL_BLK structure, using the set ID IPSET_PROTOCOL, the parameter ID IPPARM_PROTOCOL_BITMASK, and one of the following values:

- IP_PROTOCOL_SIP
- IP_PROTOCOL_H323

A network device that is opened in multi-protocol mode defaults to IP_PROTOCOL_H323 if the protocol is not explicitly set in the makecall block.

Note: Applications should **not** use the **gc_SetUserInfo( )** function to set the IP protocol.

When making calls on devices that support only one protocol, it is not necessary to include an IPSET_PROTOCOL element in the makecall block. If the application tries to include an IPSET_PROTOCOL element in the makecall block that conflicts with the protocol supported by the device, the application receives an error.

When using SIP, if the remote side does not send a final response to an outgoing INVITE (sent by the call control library) within 64 seconds, the **gc_MakeCall( )** function times out and the library generates a GCEV_DISCONNECTED event to the application. If the application attempts to drop the call before the 64 second timeout is reached, the library's behavior depends on whether a provisional response was received. If no provisional response was received before the application cancels the call, the library cleans up the call immediately. But if a provisional response was received before the application attempts to cancel the call, the library sends a CANCEL to the remote side and generates a GCEV_DROPCALL event to the application after it receives a 200OK response to the CANCEL and a 487RequestTerminated response to the original INVITE, or when a further 32 second timeout expires.

8.3.17.1 Configurable Call Parameters

Call parameters can be specified when using the **gc_MakeCall( )** function. The parameters values specified are only valid for the duration of the current call. At the end of the current call, the default parameter values for the specific line device override these parameter values. The **makecallp** parameter of the **gc_MakeCall( )** function is a pointer to the GC_MAKECALL_BLK structure. The GC_MAKECALL_BLK structure has a gclib field that points to a GCLIB_MAKECALL_BLK structure. The ext_datap field within the GCLIB_MAKECALL_BLK structure points to a GC_PARM_BLK structure with a list of the parameters to be set as call values. The parameters that can be specified through the ext_datap pointer depend on the protocol used (H.323 or SIP) and are described in the following subsections.

Variance for H.323

Table 35 shows the call parameters that can be specified when using **gc_MakeCall( )** with H.323.

Table 35. Configurable Call Parameters When Using H.323

Set ID	Parameter ID(s) and Data Types
GCSET_CHAN_CAPABILITY	IPPARM_LOCAL_CAPABILITY Data structure, type IP_CAPABILITY. See the reference page for IP_CAPABILITY on page 652 for more information. **Note:** If no transmit/receive coder type is specified, any supported coder type is accepted.
IPSET_CALLINFO See Section 9.2.2, “IPSET_CALLINFO”, on page 615 for more information.	IPPARM_CONNECTIONMETHOD Enumeration, with one of the following values: • IP_CONNECTIONMETHOD_FASTSTART • IP_CONNECTIONMETHOD_SLOWSTART See Section 4.2.2, “H.323 Fast Start and Slow Start”, on page 109 for more information.
	IPPARM_CALLID Array of octets, length = MAX_IP_H323_CALLID_LENGTH
	IPPARM_DISPLAY String, max. length = MAX_DISPLAY_LENGTH (82), null-terminated
	IPPARM_FASTSTART_MANDATORY_H245CH Enumeration, with one of the following values: • IP_FASTSTART_MANDATORY_H245CH_OFF • IP_FASTSTART_MANDATORY_H245CH_ON See Section 4.2.3, “H.323 Fast Start with Optional H.245 Channel”, on page 110 for more information.
IPSET_CALLINFO (cont.)	IPPARM_H245TUNNELING Enumeration, with one of the following values: • IP_H245TUNNELING_ON or IP_H245TUNNELING_OFF See Section 4.1.3, “Enabling and Disabling H.245 Tunneling (H.323)”, on page 107 for more information.
	IPPARM_PHONELIST String, max. length = 131.
	IPPARM_USERUSER_INFO String, max. length = MAX_USERUSER_INFO_LENGTH (131 bytes)
IPSET_CONFERENCE	IPPARM_CONFERENCE_GOAL Enumeration with one of the following values: • IP_CONFERENCEGOAL_UNDEFINED • IP_CONFERENCEGOAL_CREATE • IP_CONFERENCEGOAL_JOIN • IP_CONFERENCEGOAL_INVITE • IP_CONFERENCEGOAL_CAP_NEGOTIATION • IP_CONFERENCEGOAL_SUPPLEMENTARY_SRVC
Notes: The term “String” implies the normal definition of a character string which can contain letters, numbers, white space, and a null (for termination).	

Table 35. Configurable Call Parameters When Using H.323 (Continued)

Set ID	Parameter ID(s) and Data Types
IPSET_NONSTANDARDDATA See Section 9.2.19, “IPSET_NONSTANDARDDATA”, on page 631 for more information.	Either: • IPPARM_NONSTANDARDDATA_DATA String, max. length = MAX_NS_PARM_DATA_LENGTH (128) and • IPPARM_NONSTANDARDDATA_OBJID Unsigned Int[ ], max. length =MAX_NS_PARM_OBJID_LENGTH (40) or • IPPARM_NONSTANDARDDATA_DATA String, max. length = MAX_NS_PARM_DATA_LENGTH (128) and • IPPARM_H221NONSTANDARD Data structure, type IP_H221NONSTANDARD
IPSET_NONSTANDARDCONTROL See Section 9.2.18, “IPSET_NONSTANDARDCONTROL”, on page 630 for more information.	Either: • IPPARM_NONSTANDARDDATA_DATA String, max. length = MAX_NS_PARM_DATA_LENGTH (128) and • IPPARM_NONSTANDARDDATA_OBJID Unsigned Int[ ], max. length = MAX_NS_PARM_OBJID_LENGTH (40) or • IPPARM_NONSTANDARDDATA_DATA String, max. length = MAX_NS_PARM_DATA_LENGTH (128) and • IPPARM_H221NONSTANDARD Data structure, type IP_H221NONSTANDARD
Notes: The term “String” implies the normal definition of a character string which can contain letters, numbers, white space, and a null (for termination).	

Variance for SIP

Table 36 shows the call parameters that can be specified when using **gc_MakeCall( )** with SIP.

Table 36. Configurable Call Parameters When Using SIP

Set ID	Parameter ID and Datatype
GCSET_CHAN_CAPABILITY **Note:** This parameter set is not supported in 3PCC operating mode.	IPPARM_LOCAL_CAPABILITY Data structure, type IP_CAPABILITY. See reference page for IP_CAPABILITY on page 652 for more information. **Note:** If no transmit/receive coder type is specified, any supported coder type is accepted.
IPSET_CALLINFO See Section 9.2.2, "IPSET_CALLINFO", on page 615 for more information.	IPPARM_CONNECTIONMETHOD Enumeration, with one of the following values: • IP_CONNECTIONMETHOD_FASTSTART • IP_CONNECTIONMETHOD_SLOWSTART See Section 4.2.4, "SIP Call Setup Modes", on page 111 for more information. This parameter ID is not supported in 3PCC operating mode.
	IPPARM_CALLID String, max. length = MAX_IP_SIP_CALLID_LENGTH **Note:** Directly manipulating the SIP Call ID message header via IPSET_SIP_MSGINFO and IPPARM_CALLID_HDR will override any value provided here.
	IPPARM_DISPLAY String, max. length = MAX_DISPLAY_LENGTH (82), null-terminated
	IPPARM_PHONELIST String, max. length = 131

Notes:
The term "String" implies the normal definition of a character string which can contain letters, numbers, white space, and a null (for termination).
The parameter names used are more closely aligned with H.323 terminology. Corresponding SIP terminology is described in http://www.ietf.org/rfc/rfc3261.txt?number=3261.

8.3.17.2 Origination Address Information

The origination address can be specified in the origination field of type GCLIB_ADDRESS_BLK in the GCLIB_MAKECALL_BLK structure. The address field in the GCLIB_ADDRESS_BLK contains the actual address and the address_type field in the GCLIB_ADDRESS_BLK structure defines the type (IP address, name, telephone number) in the address field.

Note: The total length of the address string is limited by the value MAX_ADDRESS_LEN (defined in *gclib.h*).

The origination address can be set using the **gc_SetCallingNum( )** function, which is a deprecated function. The preferred equivalent is **gc_SetConfigData( )**. See the *Dialogic® Global Call API Library Reference* for more information.

8.3.17.3 Forming a Destination Address String

Variance for H.323

The destination address is formed by concatenating values from three different sources:

- the GC_MAKECALL_BLK
- the **numberstr** parameter of **gc_MakeCall( )**
- the phone list

The order or precedence of these elements and the rules for forming a destination address are described below.

Notes: ***1.*** The following description refers to a delimited string. The delimiter is configurable by setting the value of the delimiter field in the IP_CCLIB_START_DATA structure used by the **gc_Start( )** function.

2. The total length of the address string is limited by the value MAX_ADDRESS_LEN (defined in *gclib.h*).

3. The destination address must be a valid address that can be translated by the remote node.

The destination information string is delimited concatenation of the following strings in the order of precedence shown:

1. A string constructed from the destination field of type GCLIB_ADDRESS_BLK in the GCLIB_MAKECALL_BLK. When specifying the destination information in the GCLIB_ADDRESS_BLK, the address field contains the actual address information and the address_type field defines the type (IP address, name, telephone number) in the address. For example, if the address field is "127.0.0.1", the address_type field must be GCADDRTYPE_IP. The supported address types are:
   - GCADDRTYPE_INTL – international telephone number
   - GCADDRTYPE_NAT – national telephone number
   - GCADDRTYPE_LOCAL – local telephone number
   - GCADDRTYPE_DOMAIN – domain name
   - GCADDRTYPE_URL – URL name
   - GCADDRTYPE_EMAIL – e-mail address
2. The **numberstr** parameter in the **gc_MakeCall( )** function. The **numberstr** parameter is treated as a free string that may be a delimited concatenation of more than one section. The application may include a prefix in a section that maps to a corresponding field in the Setup message. In case of blind call transfer, the application must include a prefix for all sections. See Section 8.3.17.4, "Destination Address Interpretation", on page 567, for more information.
3. Phone list as described in Table 35, "Configurable Call Parameters When Using H.323", on page 561 (and set using IPSET_CALLINFO, IPPARM_PHONELIST). Phone List is treated as a free string that may be a delimited concatenation of more than one section. The application may prefix a section that maps to a corresponding field in the Setup message. In case of blind call transfer, the application must include a prefix for all sections. See the Section 8.3.17.4, "Destination Address Interpretation", on page 567 for more information.

Variance for SIP

The format of the destination address for a SIP call is:

```
        user@host; param=value
```

with the elements representing:

user
: a user name or phone number

host
: a domain name or an IP address

param=value
: an optional additional parameter

When making a SIP call, the destination address is formed according to the following rules in the order of precedence shown:

1. If Phone List (as described in Table 36, "Configurable Call Parameters When Using SIP", on page 563 and identified by IPSET_CALLINFO, IPPARM_PHONELIST) exists, it is taken to construct the global destination-address-string.
2. If the destination address field (of type GCLIB_ADDRESS_BLK in GCLIB_MAKECALL_BLK) exists, it is taken to construct the global destination-address-string. The address_type in GCLIB_ADDRESS_BLK is ignored. If the global destination-address-string is not empty before setting the parameter, an "@" delimiter is used to separate the two parts.
3. If the **numberstr** parameter from the **gc_MakeCall( )** function exists, it is taken to destination-address-string. If the global destination-address-string is not empty before setting the parameter, a ";" delimiter is used to separate the two parts.

Note: To observe the logic described above, the application may use only one of the APIs to send a string that is a valid SIP address.

The following code examples demonstrate the recommended ways of forming the destination string when making a SIP call. Prerequisite code for setting up the GC_MAKECALL_BLK in all the scenarios described in this section is as follows:

```
GC_MAKECALL_BLK gcmkbl;
GCLIB_MAKECALL_BLK gclib_mkbl = {0};
gcmkbl.cclib = NULL;
gcmkbl.gclib = &gclib_mkbl;
GC_PARM_BLK *target_datap = NULL;

gc_util_insert_parm_val(&target_datap,
                        IPSET_PROTOCOL,
                        IPPARM_PROTOCOL_BITMASK,
                        sizeof(char),
                        IP_PROTOCOL_SIP);
```

Scenario 1 – Making a SIP call to a known IP address, where the complete address (user@host) is specified in the makecall block:

```
char *pDestAddrBlk = "11223344@127.0.0.1";  /* where "11223344" is the
                                               phone number of the user
                                               and "127.0.0.1" is the
                                               IP address of the host */
```

```
/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_TRANSPARENT;

/* calling the function with the MAKECALL_BLK, and numberstr parameter=NULL
   the INVITE dest address will be: 11223344@127.0.0.1 */
gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout, EV_ASYNC);
```

Scenario 2 – Making a SIP call to a known IP address, where the complete address (user@host) is formed by the combination of the destination address in the makecall block and the phone list element:

```
char *pDestAddrBlk = "127.0.0.1";   /*host*/
char *IpPhoneList = "003227124311"; /*user*/

/* insert phone list */
gc_util_insert_parm_ref(&target_datap,
                        IPSET_CALLINFO,
                        IPPARM_PHONELIST,
                        (unsigned char)(strlen(IpPhoneList)+1),
                        IpPhoneList);

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_TRANSPARENT;

gclib_mkbl.ext_datap = target_datap;

/* calling the function with the MAKECALL_BLK, and numberstr parameter = NULL
   the INVITE dest address will be: 003227124311@127.0.0.1 */
gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 3 – Making a SIP call to a known IP address, where the complete address (user@host) is formed by the combination of the destination address in the makecall block, a phone list element, and optional parameter (user=phone):

```
char *pDestAddrBlk = "127.0.0.1";  /*host*/
char *IpPhoneList= "003227124311"; /*user*/
char *pDestAddrStr = "user=phone"; /*extra parameter*/

/* insert phone list */
gc_util_insert_parm_ref(&target_datap,
                        IPSET_CALLINFO,
                        IPPARM_PHONELIST,
                        (unsigned char)(strlen(IpPhoneList)+1),
                        IpPhoneList);

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_TRANSPARENT;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK, and numberstr parameter = NULL
   the INVITE dest address will be: 003227124311@127.0.0.1;user=phone */
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

8.3.17.4 Destination Address Interpretation

Note: The following information applies when using H.323 only.

Once a destination string is formed as described in the previous section, the H.323 stack treats the string according to the following rules:

- The **first** section of the string is the destination of the next IP entity (for example, a gateway, terminal, the alias for a remote registered entity, etc.) with which the application attempts to negotiate.
- A non-prefixed section in the string is the Q.931 calledPartyNumber and is the **last** section that is processed. Any section following the first non-prefixed section is ignored. Only **one** Q.931 calledPartyNumber is allowed in the destination string.
- One or more prefixed sections (H.225 destinationAddress fields) must appear **before** the non-prefixed section (Q.931 calledPartyNumber).
- When using free strings (**numberstr** parameter or Phone List), the valid buffer prefixes for H.225 addresses are:
  - TA: – IP transport address
  - TEL: – e164 telephone number
  - NAME: – H.323 ID
  - URL: – Universal Resource Locator
  - EMAIL: – e-mail address
- In case of blind call transfer, all sections must include an appropriate prefix.

The following code examples demonstrate the recommended ways of forming the destination string when making an H.323 call. Prerequisite code for setting up the GC_MAKECALL_BLK in all the scenarios described in this section is as follows:

```
GC_MAKECALL_BLK gcmkbl;
GCLIB_MAKECALL_BLK gclib_mkbl = {0};
gcmkbl.cclib = NULL;
gcmkbl.gclib = &gclib_mkbl;
GC_PARM_BLK *target_datap = NULL;

gc_util_insert_parm_val(&target_datap,
                        IPSET_PROTOCOL,
                        IPPARM_PROTOCOL_BITMASK,
                        sizeof(char),
                        IP_PROTOCOL_H323);
```

Scenario 1 – Making a call to a known IP address, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = "127.0.0.1";
char *pDestAddrStr = "123456";

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_IP;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK*/
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 2 – Making a call to a known IP address, setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = "127.0.0.1";
char *pDestAddrStr = "TEL:111,TEL:222,76543";

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_IP;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK*/
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 3 – Making a call to a known IP address, setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = "127.0.0.1";
char *pDestAddrStr = "TEL:111,TEL:222,NAME:myName";
char *IpPhoneList= "003227124311";

/* insert phone list */
gc_util_insert_parm_ref(&target_datap,
                        IPSET_CALLINFO,
                        IPPARM_PHONELIST,
                        (unsigned char)(strlen(IpPhoneList)+1),
                        IpPhoneList);

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_IP;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK*/
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 4 – Making a call to a known IP address, setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = "127.0.0.1";
char *IpPhoneList= "TEL:003227124311,TEL:444,TEL:222,TEL:1234,171717";
/* insert phone list */
gc_util_insert_parm_ref(&target_datap,
                        IPSET_CALLINFO,
                        IPPARM_PHONELIST,
                        (unsigned char)(strlen(IpPhoneList)+1),
                        IpPhoneList);
gclib_mkbl.ext_datap = target_datap;

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_IP;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK, and numberstr
   parameter = NULL */
gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 5 – While registered, making a call, via the gatekeeper, to a registered entity (using a known H.323 ID), setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = " RegisteredRemoteGW ";  /* The alias of the remote (registered) entity */
char *pDestAddrStr = "TEL:111,TEL:222,987654321";
```

```
/* set GCLIB_ADDRESS_BLK with destination string & type (H323-ID) */
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_DOMAIN;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK */
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 6 – While registered, making a call, via the gatekeeper, to a registered entity (using a known e-mail address), setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = " user@host.com ";  /* The alias of the remote (registered) entity */
char *pDestAddrStr = "TEL:111,TEL:222,987654321";

/* set GCLIB_ADDRESS_BLK with destination string & type (EMAIL) */
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_EMAIL;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK */
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 7 – While registered, making a call via the gatekeeper to a registered entity (using a known URL), setting a number of H.225 aliases, and setting the Q.931 calledPartyNumber:

```
char *pDestAddrBlk = "www.gw1.dialogic.com";  /* The alias of the remote (registered) entity */
char *pDestAddrStr = "TEL:111,TEL:222,987654321";

/* set GCLIB_ADDRESS_BLK with destination string & type (URL) */
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_URL;

gclib_mkbl.ext_datap = target_datap;
/* calling the function with the MAKECALL_BLK */
gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl, MakeCallTimeout,EV_ASYNC);
```

Scenario 8 – Invoking a blind call transfer to a known IP address, H.225 alias and using GC_MAKECALL_BLK:

```
    char *pDestAddrBlk = "127.0.0.1";
    char *IpPhoneList= "TEL:003227124311";

    /* insert phone list */
    gc_util_insert_parm_ref(&target_datap,
                                IPSET_CALLINFO,
                                IPPARM_PHONELIST,
                                (unsigned char)(strlen(IpPhoneList)+1), IpPhoneList);
                                gclib_mkbl.ext_datap = target_datap;

    /* set GCLIB_ADDRESS_BLK with destination string & type*/
    strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
    gcmkbl.gclib->destination.address_type = GCADDRTYPE_IP;
    gclib_mkbl.ext_datap = target_datap;

    /* calling the function with the MAKECALL_BLK, and numberstr parameter = NULL */
    gc_InvokeXfer(crn, 0, NULL, &gcmkbl, CallXferTimeout, EV_ASYNC);
```

Scenario 9 – Invoking a blind call transfer to a known IP address, H.225 alias and using the numberstr parameter:

```
    char *numberstr= "TA: 127.0.0.1,TEL:311";

    /* calling the function with numberstr parameter and MAKECALL_BLK = NULL */
    gc_InvokeXfer(crn, 0, numberstr, NULL, CallXferTimeout, EV_ASYNC);
```

8.3.17.5 Specifying a Timeout

Note: The following information applies when using H.323 only.

The **timeout** parameter of the **gc_MakeCall( )** function specifies the maximum time in seconds to wait for the establishment of a new call, after receiving the first response to the call. This value corresponds to the **Q.931\connectTimeOut** parameter. If the call is not established during this time, the Disconnect procedure is initiated. The default value is 120 seconds.

In addition to the **Q.931\connectTimeOut** parameter described in Section 8.3.17, "gc_MakeCall( ) Variances for IP", on page 560, two other non-configurable parameters affect the timeout behavior:

Q931\responseTimeOut
: The maximum time in seconds to wait for the first response to a new call. If no response is received during this time, the Disconnect procedure is initiated. The default value is 4 seconds.

h245\timeout:
: The maximum time in seconds to wait for the called party to acknowledge receipt of the capabilities it sent. The default value is 40 seconds.

Note: When using the H.323 protocol, the application may receive a timeout when trying to make an outbound call if network congestion is encountered and a TCP connection cannot be established. In this case, the SETUP message is not sent on the network.

8.3.17.6 Code Examples

H.323-Specific Code Example

The following code example shows how to make a call using the H.323 protocol.

```
/* Make an H323 IP call on line device ldev */
void MakeH323IpCall(LINEDEV ldev)
{
   char *IpDisplay = "This is a Display"; /* display data */
   char *IpPhoneList= "003227124311"; /* phone list */
   char *IpUUI = "This is a UUI";    /* user to user information string */
   char *pDestAddrBlk = "127.0.0.1"; /* destination IP address for MAKECALL_BLK*/
   char *pSrcAddrBlk = "987654321";  /* origination address for MAKECALL_BLK*/
   char *pDestAddrStr = "123456";    /* destination string for gc_MakeCall() function*/
   char *IpNSDataData = "This is an NSData data string";
   char *IpNSControlData = "This is an NSControl data string";
   char *IpCommonObjId = "1 22 333 4444"; /* unique format */
   IP_H221NONSTANDARD appH221NonStd;
   appH221NonStd.country_code = 181; /* USA */
   appH221NonStd.extension = 11;
   appH221NonStd.manufacturer_code = 11;
   int ChoiceOfNSData = 1;
   int ChoiceOfNSControl = 1;
   int rc = 0;
   CRN crn;
   GC_MAKECALL_BLK gcmkbl;
   int MakeCallTimeout = 120;
```

```
/* initialize GCLIB_MAKECALL_BLK structure */
GCLIB_MAKECALL_BLK gclib_mkbl = {0};

/* set to NULL to retrieve new parameter block from utility function */
GC_PARM_BLK *target_datap = NULL;
gcmkbl.cclib = NULL; /* CCLIB pointer unused */
gcmkbl.gclib = &gclib_mkbl;

/* set GCLIB_ADDRESS_BLK with destination string & type*/
strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
gcmkbl.gclib->destination.address_type = GCADDRTYPE_IP;

/* set GCLIB_ADDRESS_BLK with origination string & type*/
strcpy(gcmkbl.gclib->origination.address,pSrcAddrBlk);
gcmkbl.gclib->origination.address_type = GCADDRTYPE_NAT;

/* set signaling PROTOCOL to H323. default is H323 if device is multi-protocol */
rc = gc_util_insert_parm_val(&target_datap,
                             IPSET_PROTOCOL,
                             IPPARM_PROTOCOL_BITMASK,
                             sizeof(char),
                             IP_PROTOCOL_H323);

/* initialize IP_CAPABILITY structure */
IP_CAPABILITY t_Capability = {0};
/* configure a GC_PARM_BLK with four coders, display, phone list and UUI message: */
/* specify and insert first capability parameter data for G.7231 coder */
t_Capability.type = GCCAPTYPE_AUDIO;
t_Capability.direction = IP_CAP_DIR_LCLTRANSMIT;
t_Capability.extra.audio.VAD = GCPV_DISABLE;
t_Capability.extra.audio.frames_per_pkt = 1;
t_Capability.capability = GCCAP_AUDIO_g7231_6_3k;

rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

t_Capability.type = GCCAPTYPE_AUDIO;
t_Capability.direction = IP_CAP_DIR_LCLRECEIVE;
t_Capability.extra.audio.VAD = GCPV_DISABLE;
t_Capability.extra.audio.frames_per_pkt = 1;
t_Capability.capability = GCCAP_AUDIO_g7231_6_3k;

rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

/* specify and insert second capability parameter data for G.7229AnnexA coder */
/* changing only frames per pkt and the coder type from first capability: */
t_Capability.extra.audio.frames_per_pkt = 3;
t_Capability.capability = GCCAP_AUDIO_g729AnnexA;
rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

/* specify and insert 3rd capability parameter data for G.711Alaw 64kbit coder */
/* changing only frames per pkt and the coder type from first capability: */
t_Capability.capability = GCCAP_AUDIO_g711Alaw64k;
t_Capability.extra.audio.frames_per_pkt = 10;
```

```
/* For G.711 use frame size (ms) here, frames per packet fixed at 1 fpp */
rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

/* specify and insert fourth capability parameter data for G.711 Ulaw 64kbit coder */
/* changing only the coder type from previous capability */
t_Capability.capability = GCCAP_AUDIO_g711Ulaw64k;
rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

/* insert display string */
rc = gc_util_insert_parm_ref(&target_datap,
                             IPSET_CALLINFO,
                             IPPARM_DISPLAY,
                             (unsigned char)(strlen(IpDisplay)+1),
                             IpDisplay);

/* insert phone list */
rc = gc_util_insert_parm_ref(&target_datap,
                            IPSET_CALLINFO,
                            IPPARM_PHONELIST,
                            (unsigned char)(strlen(IpPhoneList)+1),
                            IpPhoneList);

/* insert user to user information */
rc = gc_util_insert_parm_ref(&target_datap,
                             IPSET_CALLINFO,
                             IPPARM_USERUSER_INFO,
                             (unsigned char)(strlen(IpUUI)+1),
                             IpUUI);

/* setting NS Data elements */
gc_util_insert_parm_ref_ex(&target_datap,
                           IPSET_NONSTANDARDDATA,
                           IPPARM_NONSTANDARDDATA_DATA,
                           (unsigned long)(strlen(IpNSDataData)+1),
                           IpNSDataData);

if(ChoiceOfNSData) /* App chooses in advance which type of */
{                  /* second NS element to use */
   gc_util_insert_parm_ref(&target_datap,
                           IPSET_NONSTANDARDDATA,
                           IPPARM_H221NONSTANDARD,
                           sizeof(IP_H221NONSTANDARD),
                           &appH221NonStd);
}

else
{
   gc_util_insert_parm_ref(&target_datap,
                           IPSET_NONSTANDARDDATA,
                           IPPARM_NONSTANDARDDATA_OBJID,
                           (unsigned char)(strlen(IpCommonObjId)+1),
                           IpCommonObjId);
}
```

```
   /* setting NS Control elements */
   gc_util_insert_parm_ref_ex(&target_datap,
                              IPSET_NONSTANDARDCONTROL,
                              IPPARM_NONSTANDARDDATA_DATA,
                              (unsigned long)(strlen(IpNSControlData)+1),
                              IpNSControlData);

   if(ChoiceOfNSControl) /* App chooses in advance which type of */
   {                    /* second NS element to use */
      gc_util_insert_parm_ref(&target_datap,
                              IPSET_NONSTANDARDCONTROL,
                              IPPARM_H221NONSTANDARD,
                              sizeof(IP_H221NONSTANDARD),
                              &appH221NonStd);
   }
   else
   {
      gc_util_insert_parm_ref(&target_datap,
                              IPSET_NONSTANDARDCONTROL,
                              IPPARM_NONSTANDARDDATA_OBJID,
                              (unsigned char)(strlen(IpCommonObjId)+1),
                              IpCommonObjId);
   }

   if(rc == 0)
   {
      gclib_mkbl.ext_datap = target_datap;
      rc = gc_MakeCall(ldev, &crn, pDestAddrStr, &gcmkbl,
                       MakeCallTimeout, EV_ASYNC);

      /* deallocate GC_PARM_BLK pointer */
      gc_util_delete_parm_blk(target_datap);
   }
}
```

SIP-Specific Code Example

The following code example shows how to make a call using the SIP protocol.

```
/* Make a SIP IP call on line device ldev */
void MakeSipIpCall(LINEDEV ldev)
{
   char *IpDisplay = "This is a Display";  /* display data */
   char *pDestAddrBlk = "12345@127.0.0.1"; /* destination IP address for MAKECALL_BLK */
   char *pSrcAddrBlk = "987654321"; /* origination address for MAKECALL_BLK*/

   int rc = 0;
   CRN crn;
   GC_MAKECALL_BLK gcmkbl;
   int MakeCallTimeout = 120;

   /* initialize GCLIB_MAKECALL_BLK structure */
   GCLIB_MAKECALL_BLK gclib_mkbl = {0};

   /* set to NULL to retrieve new parameter block from utility function */
   GC_PARM_BLK *target_datap = NULL;
   gcmkbl.cclib = NULL; /* CCLIB pointer unused */
   gcmkbl.gclib = &gclib_mkbl;

   /* set GCLIB_ADDRESS_BLK with destination string & type*/
   strcpy(gcmkbl.gclib->destination.address,pDestAddrBlk);
   gcmkbl.gclib->destination.address_type = GCADDRTYPE_TRANSPARENT;
```

```
/* set GCLIB_ADDRESS_BLK with origination string & type*/
strcpy(gcmkbl.gclib->origination.address,pSrcAddrBlk);
gcmkbl.gclib->origination.address_type = GCADDRTYPE_TRANSPARENT;

/* set signaling PROTOCOL to SIP*/
rc = gc_util_insert_parm_val(&target_datap,
                             IPSET_PROTOCOL,
                             IPPARM_PROTOCOL_BITMASK,
                             sizeof(char),
                             IP_PROTOCOL_SIP);

/* initialize IP_CAPABILITY structure */
IP_CAPABILITY t_Capability = {0};
/* configure a GC_PARM_BLK with four coders, display, phone list and UUI message: */
/* specify and insert first capability parameter data for G.7231 coder */
t_Capability.type = GCCAPTYPE_AUDIO;
t_Capability.direction = IP_CAP_DIR_LCLTRANSMIT;
t_Capability.extra.audio.VAD = GCPV_DISABLE;
t_Capability.extra.audio.frames_per_pkt = 1;
t_Capability.capability = GCCAP_AUDIO_g7231_6_3k;

rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

t_Capability.type = GCCAPTYPE_AUDIO;
t_Capability.direction = IP_CAP_DIR_LCLRECEIVE;
t_Capability.extra.audio.VAD = GCPV_DISABLE;
t_Capability.extra.audio.frames_per_pkt = 1;
t_Capability.capability = GCCAP_AUDIO_g7231_6_3k;

rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

/* specify and insert second capability parameter data for G.7229AnnexA coder */
/* changing only frames per pkt and the coder type from first capability: */
t_Capability.extra.audio.frames_per_pkt = 3;
t_Capability.capability = GCCAP_AUDIO_g729AnnexA;
rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);

/* specify and insert 3rd capability parameter data for G.711Alaw 64kbit coder */
/* changing only frames per pkt and the coder type from first capability: */
t_Capability.capability = GCCAP_AUDIO_g711Alaw64k;
t_Capability.extra.audio.frames_per_pkt = 10;

/* For G.711 use frame size (ms) here, frames per packet fixed at 1 fpp */
rc = gc_util_insert_parm_ref(&target_datap,
                             GCSET_CHAN_CAPABILITY,
                             IPPARM_LOCAL_CAPABILITY,
                             sizeof(IP_CAPABILITY),
                             &t_Capability);
```

```
   /* specify and insert fourth capability parameter data for G.711 Ulaw 64kbit coder */
   /* changing only the coder type from previous capability */
   t_Capability.capability = GCCAP_AUDIO_g711Ulaw64k;
   rc = gc_util_insert_parm_ref(&target_datap,
                                GCSET_CHAN_CAPABILITY,
                                IPPARM_LOCAL_CAPABILITY,
                                sizeof(IP_CAPABILITY),
                                &t_Capability);

   /* insert display string */
   rc = gc_util_insert_parm_ref(&target_datap,
                                IPSET_CALLINFO,
                                IPPARM_DISPLAY,
                                (unsigned char)(strlen(IpDisplay)+1),
                                IpDisplay);

   if (rc == 0)
   {
      gclib_mkbl.ext_datap = target_datap;
      /* numberstr parameter may be NULL if MAKECALL_BLK is set, as secondary
         address is ignored in SIP */
      rc = gc_MakeCall(ldev, &crn, NULL, &gcmkbl, MakeCallTimeout,EV_ASYNC);

      /* deallocate GC_PARM_BLK pointer */
      gc_util_delete_parm_blk(target_datap);
   }
}
```

8.3.18 gc_OpenEx( ) Variances for IP

The **gc_OpenEx**( ) function is supported in both synchronous and asynchronous mode, but the use of asynchronous mode is recommended.

The procedure for opening devices is the same regardless of whether H.323 or SIP is used. The IPT network device (N_ipt_BxTy) and IP Media device (M_ipmBxCy) can be opened in the same **gc_OpenEx**( ) call and a voice device (V_dxxxBwCz) can also be included.

The format of the **devicename** parameter is:

```
:P_nnnn:N_iptBxTy:M_ipmBxCy:V_dxxxBwCz
```

Notes: 1. The board and timeslot numbers for network devices do **not** have to be the same as the board and channel numbers for media devices.

2. It is possible to specify :N_iptBx (without any :M component) in the **devicename** parameter to get an IPT board device handle. Certain Dialogic® Global Call API functions, such as **gc_SetConfigData**( ), use the IPT board device to specify call parameters (such as coders) for all devices in one operation or **gc_ReqService( )** to perform registration and deregistration operations. See Section 8.3.25, "gc_SetConfigData( ) Variances for IP", on page 583 and Section 8.3.22, "gc_ReqService( ) Variances for IP", on page 578 for more information.

3. It is also possible to specify :M_ipmBx (without any :N component) in the **devicename** parameter to get an IP Media board device handle.

The prefixes (P_, N_, M_ and V_) are used for parsing purposes. These fields may appear in any order. The conventions described below allow the Dialogic® Global Call API to map subsequent calls made on specific line devices or CRNs to interface-specific libraries. The fields within the **devicename** parameter must each begin with a colon.

The meaning of each field in the **devicename** parameter is as follows:

P_nnnn
: Specifies the IP protocol to be used by the device. This field is mandatory. Possible values are:
  - P_H323 – Use the device for H.323 calls only
  - P_SIP – Use the device for SIP calls only
  - P_IP – Multi-protocol option; use the device for SIP or H.323 calls

  Note: When specifying an IPT board device (see below), use the multi-protocol option, P_IP.

N_iptBxTy
: Specifies the name of the IPT network device where **x** is the logical board number and **y** is the logical channel number. An IPT board device can be specified using N_iptBx, where **x** is the logical board number.

M_ipmBxCy
: Specifies the name of the IP Media device, where **x** is the logical board number and **y** is the logical channel number to be associated with an IPT network device. This field is optional.

V_dxxxBwCz
: Specifies a voice resource, where **w** and **z** are the voice board and channel numbers respectively. This field is optional.

An IPT network device (iptBx) can also used for host LAN disconnect alarms. Note that all other Dialogic® Global Call API alarms for IP are reported on IP Media (ipm) devices, not IPT network (ipt) devices.

Note: Applications should avoid closing and re-opening devices multiple times. Board devices and channel devices should be opened during initialization and should remain open for the duration of the application.

For Windows® operating systems, the SRL function **sr_getboardcnt( )** can be used to retrieve the number of IPT board devices in the system. The **class_namep** parameter in this context should be DEV_CLASS_IPT. The SRL function **ATDV_SUBDEVS( )** can be used to retrieve the number of channels on a board. The **dev** parameter in this context should be an IPT board device handle, that is, a handle returned by **gc_OpenEx( )** when opening an IPT board device.

For Linux operating systems, the SRL device mapper functions **SRLGetAllPhysicalBoards( )**, **SRLGetVirtualBoardsOnPhysicalBoard( )** and **SRLGetSubDevicesOnVirtualBoard( )** can be used to retrieve information about the boards and devices in the system.

8.3.19 gc_RejectInitXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

Variance for H.323

The parameter **parmblkp** is ignored for IP technology and should be set to NULL.

The **gc_RejectInitXfer( )** function can be used at party C only on the receipt of GCEV_REQ_INIT_XFER.

Four of the six Dialogic® Global Call API reasons are supported and result in the following ctIdentify error values signaled back to party A. Values GCVAL_REJREASON_INVADDR and GCVAL_REJREASON_INSUFFINFO cause the function to fail with a subsequent error code of IPERR_BAD_PARAM.

Table 37 lists the ctIdentity error codes that are signaled to party A based on the value of the **reason** parameter passed when the **gc_RejectXfer( )** function is called.

Table 37. ctIdentify Errors Signaled From gc_RejectInitXfer( ) to the Network

GC Value	ctIdentify Error
GCVAL_REJREASON_INSUFFINFO	N/A (will return invalid parameter error)
GCVAL_REJREASON_INVADDR	N/A (will return invalid parameter error)
GCVAL_REJREASON_NOTALLOWED	suppServInteractionNotAllowed
GCVAL_REJREASON_NOTSUBSCRIBED	suppServInteractionNotAllowed
GCVAL_REJREASON_UNAVAIL	notAvailable
GCVAL_REJREASON_UNSPECIFIED	unspecified

Variance for SIP

This function does not apply to SIP call transfer. The SIP stack does not contact the Transfer Target or Transferred-To party (party C) until party A calls **gc_InvokeXfer( )**, so there is no issue of accepting or rejecting the transfer at the initiation stage.

8.3.20 gc_RejectXfer( ) Variances for IP

This function is only available if the call transfer supplementary service was enabled via the sup_serv_mask field in the IP_VIRTBOARD structure when the board device was started.

The parameter **parmblkp** is ignored for IP technology.

The **gc_RejectXfer( )** function can be used at party B only after the receipt of a GCEV_REQ_XFER event.

Variance for H.323 (H.450.2)

All six Dialogic® Global Call API rejection reasons are supported. Table 38 lists the ctInitiate error codes that are signaled to party A based on the value of the **reason** parameter passed when the **gc_RejectXfer( )** function is called.

Table 38. ctInitiate Errors Signaled From gc_RejectXfer( ) to the Network

GC Value	ctInitiate Error
GCVAL_REJREASON_INSUFFINFO	invalidReroutingNumber
GCVAL_REJREASON_INVADDR	invalidReroutingNumber
GCVAL_REJREASON_NOTALLOWED	suppServInteractionNotAllowed
GCVAL_REJREASON_NOTSUBSCRIBED	suppServInteractionNotAllowed
GCVAL_REJREASON_UNAVAIL	notAvailable
GCVAL_REJREASON_UNSPECIFIED	unspecified

Variance for SIP

The value of the **reason** parameter must be between IPEC_SIPReasonStatusMin and IPEC_SIPReasonStatusMax, as defined in the *gcip_defs.h* header file.

8.3.21 gc_ReleaseCallEx( ) Variances for IP

The **gc_ReleaseCallEx**( ) function is supported in both synchronous and asynchronous modes, but the use of asynchronous mode is recommended.

Note: An existing call on a line device must be released before an incoming call can be processed.

8.3.22 gc_ReqService( ) Variances for IP

This function is only supported in asynchronous mode.

The **gc_ReqService**( ) function can be used to register an endpoint with a registration server (gateway in H.323 or registrar in SIP). Function parameters must be set as follows:

target_type
: GCTGT_GCLIB_NETIF

target_ID
: An IPT board device, obtained by using **gc_OpenEx**( ) with a **devicename** parameter of "N_iptBx"

service_ID
: Any valid reference to an unsigned long; must not be NULL

reqdatap
: A pointer to a GC_PARM_BLK containing registration information.

respdatapp
: Not used in asynchronous mode; set to NULL.

mode
: EV_ASYNC

The registration information that can be included is protocol-specific as described in Table 39 and Table 40, below.

To set the protocol type, the following parameter element is inserted into the GC_PARM_BLK referenced by **reqdatap**:

IPSET_PROTOCOL
  IPPARM_PROTOCOL_BITMASK
  and one of the following parameter data values:
  - IP_PROTOCOL_H323
  - IP_PROTOCOL_SIP
  - IP_PROTOCOL_H323 | IP_PROTOCOL_SIP

Note: The default value for the protocol, when not specified by the application, is IP_PROTOCOL_H323.

Registration options are specified by inserting the following parameter element into the GC_PARM_BLK referenced by **reqdatap**:

IPSET_REG_INFO
  IPPARM_OPERATION_REGISTER
  and one of the following parameter data values:
  - IP_REG_SET_INFO – override an existing registration value
  - IP_REG_ADD_INFO – add a registration value
  - IP_REG_DELETE_BY_VALUE – remove a specific registration value (i.e., local alias or supported prefix only)
  - IP_REG_QUERY_INFO – query a SIP Registrar for existing bindings (SIP only)

See Section 4.27.4, “Registration Code Examples”, on page 326 for more information.

Deregister options are specified by inserting the following parameter element into the GC_PARM_BLK referenced by **reqdatap**:

IPSET_REG_INFO
  IPPARM_OPERATION_DEREGISTER
  and one of the following parameter data values:
  - IP_REG_MAINTAIN_LOCAL_INFO – deregister and keep the registration information locally
  - IP_REG_DELETE_ALL – deregister and discard the local registration information

See Section 4.27.4.2, “Deregistration Example”, on page 329 for more information.

The GCEV_SERVICERESP event, which is received on an IPT board device handle, indicates that a service request has been responded to by an H.323 gatekeeper or a SIP registrar. This event does not necessarily mean that the registration operation itself was completed successfully, however; successful completion of the operation is indicated by the result code IPERR_OK. The event data includes a specification of the protocol used in the following parameter element:

IPSET_PROTOCOL
  IPPARM_PROTOCOL_BITMASK
  and one of the following parameter data values:
  - IP_PROTOCOL_H323
  - IP_PROTOCOL_SIP

Variance for H.323

When using H.323, the registration information that can be included in the GC_PARM_BLK associated with the **gc_ReqService( )** function is shown in Table 39.

Table 39. Registration Information When Using H.323

Set ID	Parameter IDs and Values
GCSET_SERVREQ	PARM_REQTYPE † • Value = IP_REQTYPE_REGISTRATION
GCSET_SERVREQ	PARM_ACK †
IPSET_PROTOCOL	IPPARM_PROTOCOL_BITMASK Bitmask composed from one or both of the following values: • IP_PROTOCOL_H323 • IP_PROTOCOL_SIP
IPSET_REG_INFO See Section 9.2.22, "IPSET_REG_INFO", on page 632, for more information.	IPPARM_OPERATION_REGISTER, with defined values: • IP_REG_SET_INFO • IP_REG_ADD_INFO • IP_REG_DELETE_BY_VALUE IPPARM_OPERATION_DEREGISTER, with defined values: • IP_REG_MAINTAIN_LOCAL_INFO • IP_REG_DELETE_ALL IPPARM_REG_ADDRESS • Value = IP_REGISTER_ADDRESS structure See the reference page for IP_REGISTER_ADDRESS on page 661 for more information IPPARM_REG_TYPE, with defined values: • IP_REG_GATEWAY • IP_REG_TERMINAL
IPSET_LOCAL_ALIAS	IPPARM_ADDRESS_DOT_NOTATION IPPARM_ADDRESS_EMAIL IPPARM_ADDRESS_H323_ID IPPARM_ADDRESS_PHONE IPPARM_ADDRESS_TRANSPARENT IPPARM_ADDRESS_URL Data type: String
IPSET_SUPPORTED_PREFIXES	IPPARM_ADDRESS_DOT_NOTATION IPPARM_ADDRESS_EMAIL IPPARM_ADDRESS_H323_ID IPPARM_ADDRESS_PHONE IPPARM_ADDRESS_TRANSPARENT IPPARM_ADDRESS_URL Data type: String
† Mandatory parameters. These parameters are required to support the generic service request mechanism provided by Dialogic® Global Call API and are not sent in any registration message.	

Multiple aliases and supported prefix information is supported when the target protocol for registration is H.323.

Variance for SIP

When using SIP, the registration information that can be included in the GC_PARM_BLK associated with the **gc_ReqService( )** function is shown in Table 40.

Table 40. Registration Information When Using SIP

Set ID	Parameter IDs
GCSET_SERVREQ	PARM_REQTYPE † • Value = IP_REQTYPE_REGISTRATION
GCSET_SERVREQ	PARM_ACK †
IPSET_LOCAL_ALIAS	IPPARM_ADDRESS_DOT_NOTATION IPPARM_ADDRESS_EMAIL IPPARM_ADDRESS_TRANSPARENT Data type: String
IPSET_PROTOCOL	IPPARM_PROTOCOL_BITMASK Bitmask composed from one or both of the following values: • IP_PROTOCOL_H323 • IP_PROTOCOL_SIP
IPSET_REG_INFO See Section 9.2.22, "IPSET_REG_INFO", on page 632, for more information.	IPPARM_OPERATION_REGISTER, with defined values: • IP_REG_ADD_INFO • IP_REG_DELETE_BY_VALUE • IP_REG_QUERY_INFO • IP_REG_SET_INFO IPPARM_OPERATION_DEREGISTER, with defined values: • IP_REG_MAINTAIN_LOCAL_INFO • IP_REG_DELETE_ALL IPPARM_REG_ADDRESS • Value = IP_REGISTER_ADDRESS structure See the reference page for IP_REGISTER_ADDRESS on page 661 for more information IPPARM_REG_AUTOREFRESH, with defined values: • IP_REG_AUTOREFRESH_DISABLE • IP_REG_AUTOREFRESH_ENABLE
† Mandatory parameters. These parameters are required to support the generic service request mechanism provided by Dialogic® Global Call API and are not sent in any registration message.	

Multiple aliases are supported when the target protocol for registration is SIP, but prefix information is **ignored**.

When using SIP, auto-refresh is enabled by default if there is no IPSET_REG_INFO / IPPARM_REG_AUTOREFRESH parameter specified. The default for the requested expiration time is 3600 seconds; the actual expiration time is determined by the Registrar.

8.3.23 gc_RespService( ) Variances for IP

This function is only supported in asynchronous mode.

The **gc_RespService( )** function operates on an IPT board device and is used to respond to requests from an H.323 gatekeeper or a SIP registrar.

The following are the relevant function parameters:

target_type
: GCTGT_CCLIB_NETIF

target_id
: IPT board device

datap
: pointer to GC_PARM_BLK with additional response information

Because some of the data may be protocol specific (in future releases), there is a facility to set the protocol type using the following IP parameter element in the GC_PARM_BLK, **datap**:

IPSET_PROTOCOL
  IPPARM_PROTOCOL_BITMASK
  and one of the following parameter data values:
  - IP_PROTOCOL_H323
  - IP_PROTOCOL_SIP
  - IP_PROTOCOL_H323 | IP_PROTOCOL_SIP

Note: The default value for the protocol when not specified by the application is IP_PROTOCOL_H323.

The GCEV_SERVICEREQ event indicates that a service has been requested by an H.323 gatekeeper or a SIP registrar. The event is received on an IPT board device handle. The event data includes a specification of the protocol used in the following parameter element:

IPSET_PROTOCOL
  IPPARM_PROTOCOL_BITMASK
  and one of the following parameter data values:
  - IP_PROTOCOL_H323
  - IP_PROTOCOL_SIP

8.3.24 gc_SetAlarmParm( ) Variances for IP

The **gc_SetAlarmParm( )** function can be used to set QoS threshold values. The function parameter values in this context are:

linedev
: The media device handle, retrieved using the **gc_GetResourceH( )** function. See Section 4.26.2, "Retrieving the Media Device Handle", on page 310 for more information.

aso_id
: The alarm source object ID. Set to ALARM_SOURCE_ID_NETWORK_ID.

ParmSetID
: Must be set to ParmSetID_qosthreshold_alarm.

alarm_parm_list
: A pointer to an ALARM_PARM_FIELD structure. The alarm_parm_number field is not used. The alarm_parm_data field is of type GC_PARM, which is a union. In this context, the type

used is void *pstruct, and is cast as a pointer to an IPM_QOS_THRESHOLD_INFO structure, which includes an IPM_QOS_THRESHOLD_DATA structure that contains the parameters representing threshold values. See the IPM_QOS_THRESHOLD_INFO data structure pages in the *Dialogic® IP Media Library API Library Reference* and the *Dialogic® IP Media Library API Programming Guide* for more information.

The thresholds supported by Dialogic® Global Call API for HMP are:

- QOSTYPE_JITTER
- QOSTYPE_LOSTPACKETS
- QOSTYPE_RTCPTIMEOUT
- QOSTYPE_RTPTIMEOUT

mode

Must be set to EV_SYNC.

Note: Applications **must** include the *gcipmlib.h* header file before Dialogic® Global Call API can be used to set or retrieve QoS threshold values.

See Section 4.26.3, "Setting QoS Threshold Values", on page 310 for code examples.

8.3.25 gc_SetConfigData( ) Variances for IP

This function is only supported in asynchronous mode.

The **gc_SetConfigData( )** function is used for a number of different purposes:

- setting parameters for all board devices, including devices that are already open
- enabling and disabling unsolicited GCEV_EXTENSION events on a board device basis
- setting the type of DTMF support and the RFC 2833 payload type on a board device basis [not supported in 3PCC operating mode]
- setting T.38 fax server operating mode [not supported in 3PCC operating mode], which also enables/disables application access to re-INVITE requests
- masking and unmasking call state events on a line device basis

Notes:

1. The **gc_SetConfigData( )** function operates on board devices, that is, devices opened using **gc_OpenEx( )** with :N_iptBx:P_IP in the **devicename** parameter. By its nature, a board device is multi-protocol, that is, it applies to both the H.323 and SIP protocols and is not directed to one specific protocol. You *cannot* open a board device (with :P_H323 or :P_SIP in the **devicename** parameter) to target a specific protocol.
2. When using the **gc_SetConfigData( )** function to set parameters, the parameter values apply to all board devices, including devices that are already open. The parameters can be overridden by specifying new values in the **gc_SetUserInfo( )** function (on a per line device basis) or the **gc_MakeCall( )** function (on a per call basis).
3. Coder information can be specified for a device when using **gc_SetConfigData( )**, or when using **gc_MakeCall( )** to make a call, or when using **gc_AnswerCall( )** to answer a call. Note that this capability is not supported in 3PCC operating mode.
4. Use **gc_SetUserInfo( )** to set parameters on line devices.

When using the **gc_SetConfigData( )** function on a board device (the first three bullets above), use the following function parameter values:

target_type
: GCTGT_CCLIB_NETIF

target_id
: An IPT board device that can be obtained by using the **gc_OpenEx( )** function with :N_iptBx:P_IP in the **devicename** parameter. See Section 8.3.18, "gc_OpenEx( ) Variances for IP", on page 575 for more information.

target_datap
: A pointer to a GC_PARM_BLKP structure that contains the parameters to be configured. The parameters that can be included in the GC_PARM_BLK are protocol specific. See the following "Variance for H.323" and "Variance for SIP" sections.

As in other technologies supported by the Dialogic® Global Call API, the **gc_SetConfigData( )** function can be used to mask call state events, such as GCEV_ALERTING, on a line device basis. When used for this purpose, the **target_type** is GCTGT_GCLIB_CHAN and the **target_ID** is a line device. See the "Call State Event Configuration" section in the *Dialogic® Global Call API Programming Guide* for more information on masking events in general.

Variance for H.323

Table 39 describes the call parameters that can be included in the GC_PARM_BLK associated with the **gc_SetConfigData( )** function. These parameters are in addition to the call parameters described in Table 35, "Configurable Call Parameters When Using H.323", on page 561 that can also be included.

Table 41. Parameters Configurable Using gc_SetConfigData( ) When Using H.323

Set ID	Parameter IDs	Use Before †
GCSET_CALL_CONFIG	GCPARM_CALLPROC †† Enumeration with one of the following values: • GCCONTROL_APP – The application must use **gc_CallAck( )** to send the Proceeding message. This is the default. • GCCONTROL_TCCL – The stack sends the Proceeding message automatically.	**gc_AnswerCall( )**
† Information can be set in any state but it is only used in certain states. See the "variances" section for the specific function for more information. †† This is a system configuration parameter for the terminating side, not a call configuration parameter. It cannot be overwritten by setting a new value in **gc_SetUserInfo( )** or **gc_MakeCall( )**. ††† Applies to the configuration of tunneling for inbound calls only. See Section 4.1.3, "Enabling and Disabling H.245 Tunneling (H.323)", on page 107 for more information.		

Table 41. Parameters Configurable Using gc_SetConfigData( ) When Using H.323 (Continued)

Set ID	Parameter IDs	Use Before †
IPSET_CALLINFO	IPPARM_H245TUNNELING ††† Enumeration with one of the following values: • IP_H245TUNNELINGON • IP_H245TUNNELINGOFF	**gc_AnswerCall( )**

	IPPARM_CONNECTIONMETHOD
	 Enumeration with one of the following values:
	 • IP_CONNECTIONMETHOD_FASTSTART
	 • IP_CONNECTIONMETHOD_SLOWSTART
	 IPPARM_FASTSTART_MANDATORY_H245CH
	 Enumeration with one of the following values:
	 • IP_FASTSTART_MANDATORY_H245CH_ON
	 • IP_FASTSTART_MANDATORY_H245CH_OFF
		gc_AnswerCall( )
		 gc_MakeCall( )

IPSET_CONFIG	IPPARM_OPERATING_MODE
	 Enumeration with one of the following values:
	 • IP_AUTOMATIC_MODE
	 • IP_MANUAL_MODE
		gc_AnswerCall( )
		 gc_MakeCall( )

IPSET_DTMF	IPPARM_SUPPORT_DTMF_BITMASK
	 Datatype: Uint8[ ]
	 IPPARM_DTMF_RFC2833_PAYLOAD_TYPE
	 Datatype: Uint8[ ]
		gc_AnswerCall( )
		 gc_MakeCall( )

IPSET_VENDORINFO	IPPARM_VENDOR_PRODUCT_ID
	 String, max. length =
	 MAX_PRODUCT_ID_LENGTH (32)
	 IPPARM_VENDOR_VERSION_ID
	 String, max. length =
	 MAX_VERSION_ID_LENGTH (32)
	 IPPARM_H221NONSTD
	 Datatype IP_H221NONSTANDARD.
		gc_AnswerCall( )
		 gc_MakeCall( )

IPSET_EXTENSIONEVT_MSK	GCACT_ADDMSK Datatype: Uint8[ ] GCACT_SETMSK Datatype: Uint8[ ] GCACT_SUBMSK Datatype: Uint8[ ]	**gc_AnswerCall( )**

† Information can be set in any state but it is only used in certain states. See the "variances" section for the specific function for more information.
†† This is a system configuration parameter for the terminating side, not a call configuration parameter. It cannot be overwritten by setting a new value in **gc_SetUserInfo( )** or **gc_MakeCall( )**.
††† Applies to the configuration of tunneling for inbound calls only. See Section 4.1.3, "Enabling and Disabling H.245 Tunneling (H.323)", on page 107 for more information.

Variance for SIP

The **gc_SetConfigData( )** function can be used to enable and disable the optional GCEV_INVOKE_XFER_ACCEPTED event on a line device basis. This event is only relevant when the call transfer supplementary service is enabled, and is generated to notify the Transferor or Transferring application (party A) that the Transferee or Transferred party (party B) has received and accepted a call transfer request. As with other maskable call state events, the parameter set ID to use is GCSET_CALLEVENT_MSK, and the parameter IDs that may be used are GCACT_ADDMSK, GCACT_SUBMSK, and GCACT_SETMSK. The specific parameter value that is used to enable or disable the GCEV_INVOKE_XFER_ACCEPTED event is GCMSK_INVOKE_XFER_ACCEPTED. Note that there is no corresponding event for H.450.2 call transfers.

Table 42 describes the call parameters that can be included in the GC_PARM_BLK associated with the **gc_SetConfigData**( ) function. These parameters are in addition to the call parameters described in Table 36, "Configurable Call Parameters When Using SIP", on page 563 that can also be included.

Table 42. Parameters Configurable Using gc_SetConfigData( ) When Using SIP

Set ID	Parameter IDs	Use Before †
GCSET_CALL_CONFIG	GCPARM_CALLPROC †† Enumeration with one of the following values: • GCCONTROL_APP – The application must use **gc_CallAck( )** to send the Proceeding message. This is the default. • GCCONTROL_TCCL – The stack sends the Proceeding message automatically.	**gc_AnswerCall( )**

IPSET_CALLINFO	IPPARM_CONNECTIONMETHOD
	Enumeration with one of the following values:
	• IP_CONNECTIONMETHOD_FASTSTART
	• IP_CONNECTIONMETHOD_SLOWSTART
		gc_AnswerCall( )
		gc_MakeCall( )

IPSET_CONFIG	IPPARM_OPERATING_MODE
	Enumeration with one of the following values:
	• IP_AUTOMATIC_MODE
	• IP_MANUAL_MODE
	• IP_T38_MANUAL_MODE
	• IP_T38_MANUAL_MODIFY_MODE
		gc_AnswerCall( )
		gc_MakeCall( )

IPSET_DTMF
Note: This parameter set is not supported in 3PCC operating mode
	IPPARM_SUPPORT_DTMF_BITMASK
	Datatype: Uint8[ ]
	IPPARM_DTMF_RFC2833_PAYLOAD_TYPE
	Datatype: Uint8[ ]
		gc_AnswerCall( )
		gc_MakeCall( )

IPSET_EXTENSIONEVT_MSK	GCACT_ADDMSK Datatype: Uint8[ ] GCACT_SETMSK Datatype: Uint8[ ] GCACT_SUBMSK Datatype: Uint8[ ]	**gc_AnswerCall( )**

† Information can be set in any state but it is only used in certain states. See the "variances" section for the specific function for more information.
†† This is a system configuration parameter for the terminating side, not a call configuration parameter. It cannot be overwritten by setting a new value in **gc_SetUserInfo( )** or **gc_MakeCall( )**.

8.3.26 gc_SetUserInfo( ) Variances for IP

The **gc_SetUserInfo( )** function can be used to:

- set call values for all calls on the specified line device
- set call values for the duration of a single call
- set SIP message information fields
- set IP Media Library parameters (for example, echo cancellation parameters) for a specified line device
- associate and disassociate a T.38 Fax device with a Media device

The **gc_SetUserInfo( )** function is used to set the values of call-related information, such as coder information, display information, phone list, etc. before a call has been initiated. The information is not transmitted until the next Dialogic® Global Call API function that initiates the transmission of information on the line, such as, **gc_AnswerCall( )**, **gc_AcceptCall( )**, or **gc_CallAck( )**.

The parameters that are configurable using **gc_SetUserInfo( )** are given in Table 35, "Configurable Call Parameters When Using H.323", on page 561 and Table 36, "Configurable Call Parameters When Using SIP", on page 563. In addition, the DTMF support bitmask, (see Table 41 and Table 42) is also configurable using **gc_SetUserInfo( )**.

Note: The **gc_SetUserInfo( )** function may **not** be used to set the IP protocol for a multi-protocol line device (i.e., one that was opened in P_IP mode). The only mechanism for selecting the protocol to use is the GC_MAKECALL_BLK structure associated with the **gc_MakeCall( )** function.

The **gc_SetUserInfo( )** function operates on either a CRN or a line device:

- If the target of the function is a CRN, the information in the function is automatically directed to the protocol associated with that CRN.
- If the target of the function is a line device, then:
  - If the line device was opened as a multi-protocol device (:P_PIP), the information in the function is automatically directed to each protocol and is used by either H.323 or SIP calls made subsequently.
  - If the line device was opened as a single-protocol device (:P_H323 or :P_SIP), then the information in the function automatically applies to that protocol only and is used by calls made using that protocol.

Note: Use **gc_SetConfigData( )** to set parameters on board devices.

In the SIP third party call control (3PCC) operating mode, **gc_SetUserInfo( )** is used to set SDP content that will be sent in outbound SIP messages when the function that sends the message does not take a parameter block as one of its parameters. To support this specific use of **gc_SetUserInfo( )** an additional duration parameter value, GC_NEXT_OUTBOUND_MSG, has been defined for the function.

gc_SetUserInfo( ) is also used to set Information Elements (IEs) in Q.931 messages. See Section 4.42.3, "Setting Q.931 Message IEs", on page 395 for more information.

8.3.26.1 Setting Call Parameters for the Next Call

The relevant function parameter values in this context are:

target_type
: GCTGT_GCLIB_CRN (if a CRN exists) or GCTGT_GCLIB_CHAN (if a CRN does not exist)

target_id
: CRN (if it exists) or line device (if a CRN does not exist)

duration
: GC_SINGLECALL

infoparmblkp
: a pointer to a GC_PARM_BLK with a list of parameters (including coder information) to be set for the line device.

Note: If a call is in the Null state, the new parameter values apply to the next call. If a call is in a non-Null state, the new parameter values apply to the remainder of the current call only.

8.3.26.2 Setting Call Parameters for the Next and Subsequent Calls

When the **duration** parameter is set to GC_ALLCALLS, the new call values become the default values for the line device and are used for all subsequent calls on that device. The pertinent function parameter values in this context are:

target_type
: GCTGT_GCLIB_CHAN

target_id
: line device

duration
: GC_ALLCALLS

infoparmblkp
: a pointer to a GC_PARM_BLK with a list of parameters (including coder information) to be set for the line device.

Note: If a call is in the Null state, the new parameter values apply to the next call and all subsequent calls. If a call is in a non-Null state, the new parameter values apply to the remainder of the current call and all subsequent calls.

8.3.26.3 Setting SIP Message Information Fields

The **gc_SetUserInfo( )** function can be used to set SIP message information fields. The relevant function parameter values in this context are:

target_type
: GCTGT_GCLIB_CHAN

target_id
: line device

duration

GC_SINGLECALL

infoparmblkp

A pointer to a GC_PARM_BLK that contains one or more parameter elements, each of which contains the IPSET_SIP_MSGINFO parameter set ID and one of the following parameter IDs to identify the header field to be set:

- IPPARM_CALLID_HDR (deprecated)
- IPPARM_CONTACT_DISPLAY (deprecated)
- IPPARM_CONTACT_URI (deprecated)
- IPPARM_CONTENT_DISPOSITION (deprecated)
- IPPARM_CONTENT_ENCODING (deprecated)
- IPPARM_CONTENT_LENGTH (deprecated)
- IPPARM_CONTENT_TYPE (deprecated)
- IPPARM_DIVERSION_URI (deprecated)
- IPPARM_EVENT_HDR (deprecated)
- IPPARM_EXPIRES_HDR (deprecated)
- IPPARM_FROM (deprecated)
- IPPARM_FROM_DISPLAY (deprecated)
- IPPARM_REFER_TO (deprecated)
- IPPARM_REFERRED_BY (deprecated)
- IPPARM_REPLACES (deprecated)
- IPPARM_REQUEST_URI (deprecated)
- IPPARM_SIP_HDR
- IPPARM_SIP_HDR_REMOVE
- IPPARM_TO (deprecated)
- IPPARM_TO_DISPLAY (deprecated)

In each case, the parameter data is a string that represents the specified contents of the header field.

See Section 4.9.5, "Setting SIP Header Fields for Outbound Messages", on page 185 for more information and a code example.

8.3.26.4 Setting Persistent SIP Headers for a Session

The **gc_SetUserInfo( )** function can be used to set persistent SIP headers for a session. The relevant function parameter values in this context are:

target_type

GCTGT_GCLIB_CRN or GCTGT_GCLIB_CHAN

target_id

CRN or line device

duration

GC_SINGLE_SIP_SESSION

infoparmblkp

a pointer to a GC_PARM_BLK with a list of parameters (including coder information) to be set for the line device.

See Section 4.9.8, “Configuring Persistent SIP Headers”, on page 192 for more information and code examples.

8.3.26.5 Associating and Disassociating a T.38 Fax Device with a Media Device

To support T.38 fax server operation, the **gc_SetUserInfo( )** function is used to associate a T.38 Fax device with a Media device to facilitate a switch from an audio session to a T.38 fax session. Similarly, when switching form a T.38 fax session to an audio session, the **gc_SetUserInfo( )** function is used to disassociate the T.38 Fax device from the Media device. The relevant function parameter values in this context are:

target_type
: GCTGT_GCLIB_CRN

target_id
: CRN

duration
: GC_SINGLECALL

infoparmblkp
: a pointer to a GC_PARM_BLK that contains:
  - the IPSET_FOIP parameter set ID and one of the following parameter IDs:
    – IPPARM_T38_CONNECT when switching from audio to T.38 fax
    – IPPARM_T38_DISCONNECT when switching from T.38 fax to audio
  - an associated IP_CONNECT structure that contains the fax and media handles and the connection type (half-duplex or full-duplex)

See Section 4.38.3, “Initiating a Switch from Audio to T.38 Fax”, on page 377 for more information and a code example.

8.3.27 gc_Start( ) Variances for IP

The **gc_Start( )** function is used to configure the Dialogic® Global Call API library on a system level and on a virtual board level.

At the system level, the following items can be configured:

- the number of IPT board devices (virtual boards) to create in the system (see Section 2.3.2, “IPT Board Devices”, on page 39 for the meaning of an IPT board device)
- the maximum size of parameter data for certain Global Call parameter types, such as SIP message headers, H.323 non-standard data, and MIME part headers
- first party call control or third party call control operating mode

Note: The maximum value of the num_boards field in the IPCCLIB_START_DATA structure, which defines the number of IPT board devices and the number of NIC addresses, is 8.

On a virtual board level, the application can configure a number of characteristics for each IPT board device. Among the major capabilities and features that can be configured for each virtual board when starting the system are:

- the total number of IPT line devices that can be open concurrently
- the maximum number of IPT devices that can be used for H.323 calls and for SIP calls
- the local address and signaling port for H.323 and for SIP
- enable/disable call transfer supplementary services
- enable/disable access to H.323 message information fields and to SIP message header fields
- enable/disable and configure access to MIME-encoded message bodies in SIP messages
- enable/disable and configure SIP outbound proxy
- enable/disable and configure use of TCP transport protocol for SIP messages
- configure SIP request retry behavior
- enable/disable application access to SIP OPTIONS messages

If NULL is passed to **gc_Start( )** the system is started in a default configuration that has a single virtual board which supports both H.323 and SIP protocols. This virtual board will have the default parameters listed at the end of this section. If the default configuration is not appropriate for the application, if the application needs to use the third party call control operating mode, or if the application requires a non-default configuration for any of the parameters (for example, if it needs to use one or more of the features that are disabled by default), the application must explicitly configure the system before calling **gc_Start( )**.

To configure a non-default system, the application starts by creating an IPCCLIB_START_DATA structure and an array of IP_VIRTBOARD structures, one for each virtual board in the system. The application **must** then use the convenience functions **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** (defined in the *gcip.h* header file) to initialize each of the structures with the default value for each field in the structure. After initialization, the application can override the default value for any fields in any of these data structures to configure the virtual boards as desired. After the fields in the IPCCLIB_START_DATA and IP_VIRTBOARD structures have been configured, the IPCCLIB_START_DATA structure is passed to **gc_Start( )** via pointers in CCLIB_START_STRUCT and GC_START_STRUCT data structures.

As a simple example, the following code illustrates the **INIT_IPCCLIB_START_DATA( )** and **INIT_IP_VIRTBOARD( )** convenience functions being used to initialize the data structures for a two-board system and default field values being modified to enable long parameter values, to enable access to H.323 information elements and SIP message headers, and to enable the call transfer supplementary service:

```
IP_VIRTBOARD ip_virtboard[2];
IPCCLIB_START_DATA ipcclibstart;
INIT_IPCCLIB_START_DATA(&ipcclibstart, 2, ip_virtboard);
INIT_IP_VIRTBOARD(&ip_virtboard[0]);
INIT_IP_VIRTBOARD(&ip_virtboard[1]);
ipcclibstart.max_parm_data_size = 1024;  /* override 255 byte default for max parameter size */
ip_virtboard[0].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE; /* enable SIP header access */
ip_virtboard[1].sip_msginfo_mask = IP_SIP_MSGINFO_ENABLE; /* enable SIP header access */
ip_virtboard[0].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE; /* enable H.323 IE access */
ip_virtboard[1].h323_msginfo_mask = IP_H323_MSGINFO_ENABLE; /* enable H.323 IE access */
ip_virtboard[0].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
ip_virtboard[1].sup_serv_mask = IP_SUP_SERV_CALL_XFER; /* override supp services default */
```

When calling **gc_Start( )** with configuration data that has been set by the application, the array of CCLIB_START_STRUCT structures that is pointed to by GC_START_STRUCT must include two mandatory members to start the libraries for IP call control signaling and for IP media devices. One of these structures contains "GC_IPM_LIB" as the cclib_name field and NULL as the cclib_data field. The other structure contains "GC_H3R_LIB" as cclib_name and a pointer to the configured IPCCLIB_START_DATA structure as cclib_data.

Notes: ***1.*** When using Global Gall over IP, the GC_LIB_START structure must include both the GC_H3R_LIB and GC_IPM_LIB libraries since there are inter-dependencies.

2. The maximum value of the num_boards field is 8.

The total_max_calls, h323_max_calls, and sip_max_calls fields in the IP_VIRTBOARD structure can be used to allocate the number and types of calls among the available devices. The following #defines have been provided as a convenience to application developers:

IP_CFG_DEFAULT
: indicates to the call control library that it should determine and fill in the correct value

IP_CFG_MAX_AVAILABLE_CALLS
: indicates to the call control library that it should use the maximum available resources

Note: Do not use the value IP_CFG_MAX_AVAILABLE_CALLS with applications running on Dialogic® HMP Software. That value initializes the stack for 2016 channels, which results in a lengthy initialization time and is an inefficient use of memory and other system resources.

IP_CFG_NO_CALLS
: indicates to the call control library that it should not allocate **any** resources

The total number of IPT devices (total_max_calls) is not necessarily equal to the number of IPT devices used for H.323 calls (h323_max_calls) plus the number of IPT devices used for SIP calls (sip_max_calls). Each IPT device can be used for both H.323 and SIP. For example, if there are 2016 devices available (total_max_calls = 2016, three Dialogic® IPT boards), you can specify that all 2016 devices can be used for both H.323 calls and SIP calls (h323_max_calls = sip_max_calls = 2016), or half are used for H.323 only (h323_max_calls = 1008) and half are used for SIP only (sip_max_calls = 1008), or any other such combination. The only restriction is that total_max_calls must not exceed the sum of the other two parameters.

Note: When the library is started in 3PCC operating mode, both total_max_calls and sip_max_calls must be set to a value that is no greater than the number of channels that are licensed, and h323_max_calls should be set to 0.

The default value for the maximum number of IPT devices (total_max_calls) is 120, but this can be set to a value up to 2016. See the reference page for IP_VIRTBOARD on page 665 for more information.

The following restrictions apply when overriding values in the IPCCLIB_START_DATA and IP_VIRTBOARD structures. The **gc_Start( )** function will fail if these restrictions are not observed.

- The total number of devices (total_max_calls) must not be larger than the sum of the values for the maximum number of H.323 calls and the maximum number of SIP calls (h323_max_calls + sip_max_calls).

- The total number of devices (total_max_calls) cannot be set to IP_CFG_NO_CALLS.
- The maximum number of H.323 calls (h323_max_calls) and maximum number of SIP calls (sip_max_calls) values cannot both be set to IP_CFG_NO_CALLS.
- When configuring multiple board devices, IP_CFG_DEFAULT cannot be used as an address specifier.
- If different IP addresses or port numbers are not used when running multiple instances of an application for any one technology (H.323 or SIP), then the xxx_max_calls (xxx = h323 or sip) parameter for the other technology must be set to IP_CFG_NO_CALLS.

Default configuration parameter values

The following parameter values are set for a single virtual board that supports both H.323 and SIP if NULL is passed to **gc_Start( )**. If this configuration is not appropriate, or if the application requires any of the disabled features to be enabled, it must define and initialize an IPCCLIB_START_DATA structure and an array of IP_VIRTBOARD structures, then override the default values as necessary before passing the information to **gc_Start( )**.

The following parameters are set in the IPCCLIB_START_DATA structure and apply to the entire system:

- delimiter = , [default parsing delimiter for address strings is a comma]
- num_boards = 1
- max_parm_data_size = 255
- media_operational_mode = OPERATIONAL_MODE_1PCC

The following parameters set in IP_VIRTBOARD for the default virtual board apply to both protocols:

- total_max_calls = 120
- localIP.ip_ver = IPVER4
- localIP.u_ipaddr.ipv4 — set via DCM configuration manager utility
- sup_serv_mask = IP_SUP_SERV_DISABLED

The following parameters set in IP_VIRTBOARD for the default virtual board apply to H.323 operation:

- h323_max_calls = 120
- h323_signaling_port = 1720
- h323_msginfo_mask = IP_H323_MSGINFO_DISABLE
- terminal_type = IP_TT_GATEWAY

The following parameters set in IP_VIRTBOARD for the default virtual board apply to SIP operations:

- sip_max_calls = 120
- sip_signaling_port = 5060
- sip_msg_info_mask = IP_SIP_MSGINFO_DISABLE

- sip_mime_mem = Disabled
- outbound_proxy_IP = Disabled
- outbound_proxy_port = 5060
- outbound_proxy_hostname = NULL
- E_SIP_tcpenabled = ENUM_Disabled
- E_SIP_OutboundProxyTransport = ENUM_UDP
- E_SIP_Persistence = ENUM_PERSISTENCE_TRANSACT_USER
- SIP_maxUDPmsgLen = 1300
- E_SIP_DefaultTransport = UNUM_UDP
- E_SIP_RequestRetry = ENUM_REQUEST_RETRY_ALL
- E_SIP_OPTIONS_Access = ENUM_Disabled
- SIP_TLS_ENGINE = NULL

8.3.28 gc_Stop( ) Variances for IP

Applications must not attempt to re-start the Dialogic® Global Call API library with a call to **gc_Start( )** after calling **gc_Stop( )**, even in a debug environment. There must be no more than one call to **gc_Start( )** per application execution. After the library has been stopped, the application should be exited and re-started before any attempt is made to re-start the library.

8.3.29 gc_UnListen( ) Variances for IP

The **gc_UnListen( )** function is supported in both synchronous and asynchronous modes. The function is blocking in synchronous mode.

Note: For line devices that comprise media (ipm) and voice (dxxx) devices, routing is only done on the media devices. Routing of the voice devices must be done using the Voice API (dx_ functions).

8.4 Dialogic® Global Call API States Supported by IP

The following Dialogic® Global Call API call states are supported when using Dialogic® Global Call API with IP technology:

- GCST_ACCEPTED
- GCST_ACCEPT_XFER
- GCST_ALERTING
- GCST_CALLROUTING
- GCST_CONNECTED
- GCST_DETECTED
- GCST_DIALING
- GCST_DISCONNECTED
- GCST_IDLE

- GCST_INVOKE_XFER_ACCEPTED
- GCST_INVOKE_XFER
- GCST_NULL
- GCST_OFFERED
- GCST_PROCEEDING
- GCST_REQ_INIT_XFER
- GCST_REQ_XFER
- GCST_XFER_CMPLT

See the *Global Call API Programming Guide* for more information about the call state models.

8.5 Dialogic® Global Call API Events Supported by IP

The following Dialogic® Global Call API events are supported when using Dialogic® Global Call API with IP technology:

- GCEV_ACCEPT
- GCEV_ACCEPT_INIT_XFER (supported in H.323/H.450.2 only)
- GCEV_ACCEPT_INIT_XFER_FAIL (supported in H.323/H.450.2 only)
- GCEV_ACCEPT_MODIFY_CALL (supported in SIP only)
- GCEV_ACCEPT_MODIFY_CALL_FAIL (supported in SIP only)
- GCEV_ACCEPT_XFER
- GCEV_ACCEPT_XFER_FAIL
- GCEV_ACKCALL (deprecated; equivalent is GCEV_CALLPROC)
- GCEV_ALARM
- GCEV_ALERTING (maskable event)
- GCEV_ANSWERED
- GCEV_ATTACH (not supported in 3PCC operating mode)
- GCEV_ATTACHFAIL (not supported in 3PCC operating mode)
- GCEV_BLOCKED
- GCEV_CANCEL_MODIFY_CALL (supported in SIP only)
- GCEV_CANCEL_MODIFY_CALL_FAIL (supported in SIP only)
- GCEV_CONNECTED
- GCEV_CALLPROC
- GCEV_DETECTED (maskable event)
- GCEV_DETACH (not supported in 3PCC operating mode)
- GCEV_DETACHFAIL (not supported in 3PCC operating mode)
- GCEV_DIALING (maskable event)
- GCEV_DISCONNECTED

- GCEV_DROPCALL
- GCEV_ERROR
- GCEV_EXTENSION [unsolicited extension event]
- GCEV_EXTENSIONCMPLT [termination event for **gc_Extension( )]**
- GCEV_FATALERROR
- GCEV_INIT_XFER
- GCEV_INIT_XFER_FAIL (supported in H.323/H.450.2 only)
- GCEV_INIT_XFER_REJ (supported in H.323/H.450.2 only)
- GCEV_INVOKE_XFER
- GCEV_INVOKE_XFER_ACCEPTED (maskable event, supported in SIP only)
- GCEV_INVOKE_XFER_FAIL
- GCEV_INVOKE_XFER_REJ
- GCEV_LISTEN (not supported in 3PCC operating mode)
- GCEV_MODIFY_CALL_ACK (supported in SIP only)
- GCEV_MODIFY_CALL_CANCEL (supported in SIP only)
- GCEV_MODIFY_CALL_FAIL (supported in SIP only)
- GCEV_MODIFY_CALL_REJ (supported in SIP only)
- GCEV_OFFERED
- GCEV_OPENEX
- GCEV_OPENEX_FAIL
- GCEV_PROCEEDING (maskable event)
- GCEV_REQ_MODIFY_CALL (supported in SIP only)
- GCEV_REQ_MODIFY_UNSUPPORTED (supported in SIP only)
- GCEV_REJ_INIT_XFER (supported in H.323/H.450.2 only)
- GCEV_REJ_INIT_XFER_FAIL (supported in H.323/H.450.2 only)
- GCEV_REJ_XFER
- GCEV_REJ_XFER_FAIL
- GCEV_REJECT_MODIFY_CALL (supported in SIP only)
- GCEV_REJECT_MODIFY_CALL_FAIL (supported in SIP only)
- GCEV_RELEASECALL
- GCEV_REQ_INIT_XFER (supported in H.323/H.450.2 only)
- GCEV_REQ_XFER
- GCEV_RESETLINEDEV
- GCEV_SERVICEREQ
- GCEV_SERVICERESP
- GCEV_SERVICERESPCMPLT
- GCEV_SETCONFIGDATA
- GCEV_SETCONFIGDATAFAIL

- GCEV_SIP_200OK (maskable event, supported in SIP 3PCC operating mode only)
- GCEV_SIP_ACK (maskable event, supported in SIP 3PCC operating mode only)
- GCEV_SIP_ACK_FAILED (supported in SIP 3PCC operating mode only)
- GCEV_SIP_ACK_OK (supported in SIP 3PCC operating mode only)
- GCEV_TASKFAIL
- GCEV_UNBLOCKED
- GCEV_UNLISTEN (not supported in 3PCC operating mode)
- GCEV_XFER_CMPLT
- GCEV_XFER_FAIL

See the *Dialogic® Global Call API Library Reference* for more information about Dialogic® Global Call API events and event types that are not specific to the IP technology.

IP-Specific Parameters 9

This chapter describes the Dialogic® Global Call API parameter set IDs and parameter IDs (parm IDs) that are used with IP technology. Topics include:

The "Overview of Parameter Usage" section presents information on when and how each parameter can be set, sent, and retrieved. The subsections in the "Parameter Set Reference" provide details on the types, values, and use of each individual parameter in each parameter set.

9.1 Overview of Parameter Usage

The parameter set IDs and parameter IDs described in this chapter are defined in the *gcip.h* header file. Table 43 summarizes the parameter sets and parameters used by Dialogic® Global Call API in an IP environment, organized alphabetically by set ID and then by parameter ID.

The meaning of the columns in Table 43 are:

- **Set ID** – An identifier for a group of related parameters.
- **Parameter ID** – An identifier for a specific parameter.
- **Set** – Indicates the Dialogic® Global Call API functions used to set the parameter information in the library. Parameters that affect the operation of the board or the Dialogic® Global Call API library will have only a Set entry. Parameters that are sent to a peer endpoint will also have a Send entry.
- **Send** – Indicates the Dialogic® Global Call API functions used to send the information to a peer endpoint.
- **Retrieve** – Indicates the Dialogic® Global Call API function or event used to retrieve information that was sent by a peer endpoint.
- **H.323/SIP** – Indicates if the parameter is supported when using H.323, SIP (1PCC and/or 3PCC mode), or both H.323 and SIP.

Table 43. Summary of Parameter Sets and Parameter Usage

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
GCSET_ CALL_CONFIG	GCPARM_ CALLPROC	**gc_SetConfigData( )**	---	---	both
GCSET_ CHAN_ CAPABILITY	IPPARM_ LOCAL_CAPABILITY	**gc_SetConfigData( )** **gc_SetUserInfo( )** †	**gc_AnswerCall( )** **gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323, SIP 1PCC

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_ CALLINFO	IPPARM_ BEARERCAP	**gc_SetUserInfo( )** (GC_SINGLECALL only)	**gc_MakeCall( )**	from GCEV_OFFERED via **gc_GetMetaEvent( )**	H.323
	IPPARM_ CALLDURATION	---	---	**gc_Extension( )** (IPEXTID_GETINFO)	both
	IPPARM_CALLID	**gc_MakeCall( )** **gc_SetUserInfo( )** (GC_SINGLECALL only)	**gc_MakeCall( )**	**gc_GetCallInfo( )** (IP_CALLID) –or– **gc_Extension( )** (IPEXTID_GETINFO) **Note:** The use of **gc_Extension( )** to retrieve the Call ID is being deprecated; use **gc_GetCallInfo( )**.	both
	IPPARM_CDPN_ NUMBERING_PLAN_ ID	GC_PARM_BLK **gc_SetUserInfo( )**	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO) and asynchronous GCEV_EXTENSION CMPLT event	H.323
	IPPARM_CDPN_ TYPE_OF_NUMBER	GC_PARM_BLK **gc_SetUserInfo( )**	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO) and asynchronous GCEV_EXTENSION CMPLT event	H.323
	IPPARM_CGPN_ NUMBERING_PLAN_ ID	GC_PARM_BLK **gc_SetUserInfo( )**	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO) and asynchronous GCEV_EXTENSION CMPLT event	H.323
	IPPARM_CGPN_ TYPE_OF_NUMBER	GC_PARM_BLK **gc_SetUserInfo( )**	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO) and asynchronous GCEV_EXTENSION CMPLT event	H.323
	IPPARM_CGPN_ PRESENTATION_ INDICATOR	GC_PARM_BLK **gc_SetUserInfo( )**	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO) and asynchronous GCEV_EXTENSION CMPLT event	H.323
	IPPARM_CGPN_ SCREENING_ INDICATOR	GC_PARM_BLK **gc_SetUserInfo( )**	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO) and asynchronous GCEV_EXTENSION CMPLT event	H.323

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
	IPPARM_ CONNECTION METHOD	**gc_MakeCall( )** **gc_SetUserInfo( )** †	**gc_AnswerCall( )** **gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323, SIP 1PCC
	IPPARM_DISPLAY	**gc_SetUserInfo( )** † **gc_MakeCall( )**	**gc_AnswerCall( )** **gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	both
	IPPARM_FACILITY	**gc_SetUserInfo( )** (GC_SINGLECALL only)	**gc_AnswerCall( )** **gc_MakeCall( )**	**gc_GetMetaEvent( )** for GCEV_OFFERED, GCEV_CONNECTED, or GCEV_EXTENSION (IPEXTID_ RECEIVEMSG) event.	H.323
	IPPARM_ FASTSTART_ MANDATORY_ H245CH	**gc_SetConfigData( )** **gc_SetUserInfo( )** **gc_MakeCall( )**	---	---	H.323
	IPPARM_ H245TUNNELING	**gc_SetUserInfo( )** † **gc_MakeCall( )** **gc_SetConfigData( )** ‡	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323
	IPPARM_MEDIA WAITFORCONNECT	**gc_SetUserInfo( )**	**gc_MakeCall( )**	**gc_GetMetaEvent( )** (GCEV_OFFERED)	H.323
	IPPARM_OFFERED_ FASTSTART_CODER	---	---	**gc_GetMetaEvent( )** (GCEV_OFFERED)	H.323, SIP 1PCC
	IPPARM_ PHONELIST	**gc_SetUserInfo( )** † **gc_MakeCall( )**	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	both
	IPPARM_ PRESENTATION_IND	**gc_SetUserInfo( )**	**gc_MakeCall( )**	**gc_GetMetaEvent( )** (GCEV_OFFERED)	H.323
	IPPARM_ PROGRESS_IND	---	---	**gc_GetMetaEvent( )** (GCEV_EXTENSION) **Note:** Extension events for Progress messages are masked by default. Enable events via **gc_SetUserInfo( )** with parameter IPSET_ EXTENSIONEVT_MSK, GCACT_SETMSK, EXTENSIONEVT_ CALL_PROGRESS	H.323
	IPPARM_ SIP_PRACK_ MANDATORY	**gc_SetConfigData( )** **gc_SetUserInfo( )**			SIP
	IPPARM_ SIP_TRANSPORT_ ADDR			GCEV_OFFERED GCEV_REQ_MODIFY_ CALL	SIP

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
	IPPARM_ USERUSER_INFO	**gc_SetUserInfo( )** † **gc_MakeCall( )**	**gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323
	IPPARM_UUIE_ASN1	---	---	GCEV_OFFERED GCEV_PROCEEDING GCEV_ALERTING GCEV_CONNECTED GCEV_ DISCONNECTED GCEV_EXTENSION (IPEXTID_RECEIVEMSG) for Facility msg only	H.323
IPSET_ CONFERENCE	IPPARM_ CONFERENCE_GOAL	**gc_MakeCall( )** **gc_SetUserInfo( )** †	**gc_AnswerCall( )** **gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323
	IPPARM_ CONFERENCE_ID	---	---	**gc_Extension( )** (IPEXTID_GETINFO)	H.323
IPSET_CONFIG	IPPARM_1PCC_ REJECT_VIDEO	**gc_SetConfigData( )**	---	---	SIP
	IPPARM_ AUTHENTICATION_ CONFIGURE	**gc_SetAuthentication Info( )**	---	---	SIP
	IPPARM_ AUTHENTICATION_ REMOVE	**gc_SetAuthentication Info( )**	---	---	SIP
	IPPARM_CONFIG_TOS (deprecated–use IPPARM_IPMPARM)	**gc_MakeCall( )** **gc_SetUserInfo( )** †	**gc_AnswerCall( )** **gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	both
	IPPARM_ GETCALLINFO UPDATE	**gc_SetConfigData( )**			SIP
	IPPARM_H323_ AUTO_PROGRESS_ DISABLE				H.323
	IPPARM_IPMPARM	**gc_SetUserInfo( )**	---	---	both
	IPPARM_ OPERATING_MODE	**gc_SetConfigData( )**	---	---	H.323, SIP 1PCC

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
	IPPARM_MIME_ OVERLAP_RECEIVE	**gc_SetConfigData( )**			SIP
	IPPARM_OVERLAP SIGNALING	**gc_SetConfigData( )**			SIP
	IPPARM_ REGISTER_ SIP_HEADER	**gc_SetConfigData( )** **gc_SetUserInfo( )** †	---	---	SIP
	IPPARM_SEND_ G723_UNSPECIFIED _BITRATE	**gc_SetConfigData( )** **gc_SetUserInfo( )**			SIP
	IPPARM_SEND_ SIP_UPDATE_POST CONNECTION	**gc_SetConfigData( )**			SIP 3PCC
	IPPARM_ SIGNALING_ DEFERRED	**gc_SetConfigData( )**			SIP
IPSET_DTMF	IPPARM_ DTMF_ ALPHANUMERIC	---	**gc_Extension( )** (IPEXTID_SEND_ DTMF)	**gc_Extension( )** (IPEXTID_ RECEIVE_DTMF)	H.323, SIP 1PCC
	IPPARM_ DTMF_RFC2833_ PAYLOAD_TYPE	**gc_SetConfigData( )** **gc_SetUserInfo( )** †	---	---	H.323, SIP 1PCC
	IPPARM_ SUPPORT_DTMF_ BITMASK	**gc_SetConfigData( )** **gc_SetUserInfo( )** †	---	---	H.323, SIP 1PCC
	IPPARM_ TELEPHONY_ EVENT_DTMF	**gc_SetUserInfo( )** †	---	---	H.323, SIP 1PCC
	IPPARM_ TELEPHONY_ EVENT_INFO	**gc_SetUserInfo( )** †	---	---	H.323, SIP 1PCC
IPSET_ EXTENSIONEVT_ MSK	GCACT_ADDMSK	**gc_SetConfigData( )**	---	---	both
	GCACT_GET_MSK	**gc_SetConfigData( )**	---	---	both
	GCACT_SETMSK	**gc_SetConfigData( )**	---	---	both
	GCACT_SUBMSK	**gc_SetConfigData( )**	---	---	both

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_FOIP	IPPARM_ T38_OFFERED	---	---	GCEV_OFFERED	H.323, SIP 1PCC
	IPPARM_ T38_CONNECT	**gc_SetUserInfo( )**	---	---	H.323, SIP 1PCC
	IPPARM_ T38_DISCONNECT	**gc_SetUserInfo( )**	---	---	H.323, SIP 1PCC
IPSET_ H323_ RESPONSE_ CODE	IPPARM_BUSY_ CAUSE	**gc_SetConfigData( )**	---	---	H.323
IPSET_ IPPROTOCOL_ STATE	IPPARM_ CONTROL_ CONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ IPPROTOCOL_STATE)	H.323
	IPPARM_ CONTROL_ DISCONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ IPPROTOCOL_STATE)	H.323
	IPPARM_ EST_CONTROL_ FAILED	---	---	GCEV_EXTENSION (IPEXTID_ IPPROTOCOL_STATE)	H.323
	IPPARM_ SIGNALING_ CONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ IPPROTOCOL_STATE)	H.323
	IPPARM_ SIGNALING_ DISCONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ IPPROTOCOL_STATE)	H.323
IPSET_ LOCAL_ALIAS	IPPARM_ ADDRESS_ DOT_NOTATION	---	**gc_ReqService( )**	---	both
	IPPARM_ ADDRESS_EMAIL	---	**gc_ReqService( )**	---	both
	IPPARM_ ADDRESS_H323_ID	---	**gc_ReqService( )**	---	H.323
	IPPARM_ ADDRESS_PHONE	---	**gc_ReqService( )**	---	H.323
	IPPARM_ ADDRESS_ TRANSPARENT	---	**gc_ReqService( )**	GCEV_SERVICERESP	both
	IPPARM_ ADDRESS_URL	---	**gc_ReqService( )**	---	H.323

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_ MEDIA_STATE	IPPARM_RX_ CONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	H.323, SIP 1PCC
	IPPARM_RX_ DISCONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	H.323, SIP 1PCC
	IPPARM_RX_ INACTIVE	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	H.323, SIP 1PCC
	IPPARM_RX_ RECVONLY	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	H.323, SIP 1PCC
	IPPARM_TX_ CONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	H.323, SIP 1PCC
	IPPARM_TX_ DISCONNECTED	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	H.323, SIP 1PCC
	IPPARM_TX_ INACTIVE	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	H.323, SIP 1PCC
	IPPARM_TX_ SENDONLY	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	H.323, SIP 1PCC

IPSET_MIME
IPSET_MIME_ 200OK_TO_BYE
	IPPARM_MIME_PART	---	**gc_MakeCall( )**
			gc_SetInfo( )
			gc_CallAck( )
			gc_AcceptCall( )
			gc_AnswerCall( )
			gc_DropCall( )
			gc_Extension( )
				GCEV_OFFERED
				GCEV_PROCEEDING
				GCEV_ALERTING
				GCEV_CONNECTED
				GCEV_ DISCONNECTED
				GCEV_DROPCALL
				GCEV_TASKFAIL
				GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)
					SIP

	IPPARM_MIME_ PART_BODY	---	---	GC_PARM_BLK pointed to by IPPARM_MIME_PART	SIP
	IPPARM_MIME_ PART_BODY_SIZE	---	---	GC_PARM_BLK pointed to by IPPARM_MIME_PART	SIP
	IPPARM_MIME_ PART_HEADER	---	---	GC_PARM_BLK pointed to by IPPARM_MIME_PART	SIP

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
	IPPARM_MIME_ PART_TYPE	---	---	GC_PARM_BLK pointed to by IPPARM_MIME_PART	SIP
IPSET_MIME_ ACK_TO_ REJECTION	IPPARM_MIME_PART	**gc_SetUserInfo( )**			SIP
	IPPARM_MIME_PART _BODY	**gc_SetUserInfo( )**			SIP
	IPPARM_MIME_PART _BODY_SIZE	**gc_SetUserInfo( )**			SIP
	IPPARM_MIME_PART _HEADER	**gc_SetUserInfo( )**			SIP
	IPPARM_MIME_PART _TYPE	**gc_SetUserInfo( )**			SIP
IPSET_ MSG_H245	IPPARM_MSGTYPE	---	**gc_Extension( )** (IPEXTID_ SENDMSG)	GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)	H.323
IPSET_ MSG_Q931	IPPARM_MSGTYPE	---	**gc_Extension( )** (IPEXTID_ SENDMSG)	GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)	H.323
IPSET_ MSG_ REGISTRATION	IPPARM_MSGTYPE	---	**gc_Extension( )** (IPEXTID_ SENDMSG)	GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)	both
IPSET_ MSG_SIP	IPPARM_MSG_SIP_ RESPONSE_CODE	---	**gc_Extension( )** (IPEXTID_ SENDMSG)	GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)	SIP
	IPPARM_MSGTYPE	---	**gc_Extension( )** (IPEXTID_ SENDMSG)	GCEV_EXTENSION (IPEXTID_ RECEIVEMSG)	SIP
	IPPARM_SIP_ METHOD	---	**gc_ReqModify Call( )**	---	SIP
IPSET_ NONSTANDARD CONTROL	IPPARM_ H221NON STANDARD	**gc_SetConfigData( )** **gc_MakeCall( )** **gc_SetUserInfo( )** †	**gc_AnswerCall( )** **gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323
	IPPARM_ NONSTANDARD DATA_DATA	**gc_SetConfigData( )** **gc_SetUserInfo( )** † **gc_MakeCall( )**	**gc_AnswerCall( )** **gc_MakeCall( )** **gc_DropCall( )** **gc_ReqService( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323
	IPPARM_ NONSTANDARD DATA_OBJID	**gc_SetConfigData( )** **gc_SetUserInfo( )** † **gc_MakeCall( )**	**gc_AnswerCall( )** **gc_MakeCall( )** **gc_DropCall( )** **gc_ReqService( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323
IPSET_ NONSTANDARD DATA	IPPARM_ H221NON STANDARD	**gc_SetConfigData( )** **gc_MakeCall( )** **gc_SetUserInfo( )** †	**gc_AnswerCall( )** **gc_MakeCall( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
	IPPARM_ NONSTANDARD DATA_DATA	**gc_SetConfigData( )** **gc_SetUserInfo( )** † **gc_MakeCall( )**	**gc_AnswerCall( )** **gc_MakeCall( )** **gc_DropCall( )** **gc_ReqService( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323
	IPPARM_ NONSTANDARD DATA_OBJID	**gc_SetConfigData( )** **gc_SetUserInfo( )** † **gc_MakeCall( )**	**gc_AnswerCall( )** **gc_MakeCall( )** **gc_DropCall( )** **gc_ReqService( )**	**gc_Extension( )** (IPEXTID_GETINFO)	H.323
IPSET_ PROTOCOL	IPPARM_ PROTOCOL_ BITMASK	**gc_SetConfigData( )** **gc_SetUserInfo( )** † **gc_MakeCall( )**	**gc_ReqService( )** **gc_MakeCall( )**	---	both
IPSET_ PROXY_ INFO	IPPARM_ PROXY_ACTION	**gc_SetConfigData( )**			SIP
	IPPARM_ PROXY_INFO	**gc_SetConfigData( )**			SIP
IPSET_ REG_INFO	IPPARM_ OPERATION_ DEREGISTER	---	**gc_ReqService( )**	---	both
	IPPARM_ OPERATION_ REGISTER	---	**gc_ReqService( )**	---	both
	IPPARM_ REG_ADDRESS	---	**gc_ReqService( )**	---	both
	IPPARM_REG_ AUTOREFRESH	---	**gc_ReqService( )**	---	SIP
	IPPARM_ REG_TYPE	---	**gc_ReqService( )**	---	H.323
	IPPARM_ REG_SERVICEID	---	---	Forwarded automatically in a GCEV_SERVICERESP	SIP
	IPPARM_ REG_STATUS	---	---	Forwarded automatically in a GCEV_SERVICERESP	both
IPSET_RTP_ ADDRESS	IPPARM_LOCAL	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	both
	IPPARM_REMOTE	---	---	GCEV_EXTENSION (IPEXTID_ MEDIAINFO)	both

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_SDP	IPPARM_ SDP_ANSWER	**gc_SetUserInfo( )** with duration of GC_NEXT_ OUTBOUND_MSG	Directly via: **gc_AcceptModify Call( )** **gc_MakeCall( )** **gc_ReqModify Call( )** **gc_SipAck( )** After setting, via: **gc_AcceptCall( )** **gc_AnswerCall( )** **gc_CallAck( )** **gc_RejectModify Call( )**	From event type: GCEV_ALERTING GCEV_ANSWERED GCEV_CONNECTED GCEV_MODIFY_ CALL_ACK GCEV_OFFERED GCEV_PROCEEDING GCEV_REQ_MODIFY_ CALL GCEV_SIP_ACK	SIP 3PCC
	IPPARM_ SDP_OFFER				
	IPPARM_SDP_ OPTION_ANSWER	---	**gc_Extension( )** (OPTIONS or 200OK to OPTIONS)	GCEV_EXTENSION (IPEXTID_ RECEIVEMSG for OPTIONS or 200OK to OPTIONS)	SIP 3PCC
	IPPARM_SDP_ OPTION_OFFER				
	IPPARM_SDP_ IP_TYPE	**gc_SetUserInfo( )**			SIP 1PCC
IPSET_SIP_ MSGINFO	IPPARM_ CALLID_HDR (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_Extension( )**	From event type GCEV_OFFERED or GCEV_EXTENSION	SIP
	IPPARM_ CONTACT_DISPLAY (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, or GCEV_EXTENSION	SIP
	IPPARM_ CONTACT_URI (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP
	IPPARM_ DIVERSION_URI (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, or GCEV_EXTENSION	SIP
	IPPARM_EVENT_ HDR (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_Extension( )**	From event type GCEV_EXTENSION	SIP
	IPPARM_EXPIRES_ HDR (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP
	IPPARM_FROM (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_REQ_XFER or GCEV_EXTENSION	SIP

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
	IPPARM_ FROM_DISPLAY (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, or GCEV_EXTENSION	SIP
	IPPARM_ REFERRED_BY (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP
	IPPARM_ REPLACES (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_Extension( )**	From event type GCEV_OFFERED or GCEV_EXTENSION	SIP
	IPPARM_ REQUEST_URI (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, or GCEV_EXTENSION	SIP
	IPPARM_SIP_HDR	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, GCEV_REQ_XFER, or GCEV_EXTENSION	SIP
	IPPARM_SIP_HDR_ REMOVE	**gc_SetUserInfo( )** (GC_ALLCALLS) (GC_SINGLE_SIP_ SESSION)			SIP
	IPPARM_TO (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_InvokeXfer( )** **gc_Extension( )**	From event type GCEV_REQ_XFER or GCEV_EXTENSION	SIP
	IPPARM_ TO_DISPLAY (deprecated)	**gc_SetUserInfo( )** (GC_SINGLECALL) **gc_Extension( )**	**gc_MakeCall( )** **gc_Extension( )**	From event type GCEV_OFFERED, GCEV_CALLINFO, or GCEV_EXTENSION	SIP
IPSET_ SIP_REQUEST_ ERROR	IPPARM_SIP_DNS_ CONTINUE	---	---	From event type GCEV_EXTENSION	SIP
	IPPARM_SIP_SVC_ UNAVAIL	---	---	From event type GCEV_EXTENSION	SIP
IPSET_ SIP_ RESPONSE_ CODE	IPPARM_ACCEPT_ RESP_CODE	**gc_SetUserInfo( )** **gc_SetConfigData( )**	---	---	SIP
	IPPARM_BUSY_ REASON	**gc_SetConfigData( )**	---	---	SIP
	IPPARM_RECEIVED_ RESPONSE_ STATUS_CODE	---	---	GCEV_ALERTING	SIP

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_ SIP_ SESSION_ TIMER	IPPARM_MIN_SE	**gc_SetConfigData( )** **gc_SetUserInfo( )**	**gc_MakeCall( )** **gc_AnswerCall( )**		SIP
	IPPARM_ REFRESH_ METHOD				SIP
	IPPARM_ REFRESH_ WITHOUT_ PREFERENCE				SIP
	IPPARM_ REFRESH_ WITHOUT_ REMOTE_ SUPPORT				SIP
	IPPARM_ REFRESHER_ PREFERENCE				SIP
	IPPARM_ SESSION_ EXPIRES				SIP
	IPPARM_ TERMINATE_ CALL_ WHEN_ EXPIRES				SIP
IPSET_ SUPPORTED_ PREFIXES	IPPARM_ ADDRESS_DOT_ NOTATION	---	**gc_ReqService( )**	---	H.323
	IPPARM_ ADDRESS_EMAIL	---	**gc_ReqService( )**	---	H.323
	IPPARM_ ADDRESS_ H323_ID	---	**gc_ReqService( )**	---	H.323
	IPPARM_ ADDRESS_PHONE	---	**gc_ReqService( )**	---	H.323
	IPPARM_ ADDRESS_ TRANSPARENT	---	**gc_ReqService( )**	---	H.323
	IPPARM_ ADDRESS_URL	---	**gc_ReqService( )**	---	H.323

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_ SWITCH_ CODEC	IPPARM_ACCEPT	---	**gc_Extension( )** (IPEXTID_ CHANGE_MODE)	---	H.323, SIP 1PCC
	IPPARM_ AUDIO_INITIATE	---	**gc_Extension( )** (IPEXTID_ CHANGE_MODE)	---	H.323, SIP 1PCC
	IPPARM_ AUDIO_ REQUESTED	---	---	GCEV_EXTENSION (IPEXTID_ CHANGE_MODE)	H.323, SIP 1PCC
	IPPARM_READY	---	---	GCEV_EXTENSION (IPEXTID_ CHANGE_MODE)	H.323, SIP 1PCC
	IPPARM_REJECT	---	**gc_Extension( )** (IPEXTID_ CHANGE_MODE)	---	H.323, SIP 1PCC
	IPPARM_ T38_INITIATE	---	**gc_Extension( )** (IPEXTID_ CHANGE_MODE)	---	H.323, SIP 1PCC
	IPPARM_ T38_REQUESTED	---	---	GCEV_EXTENSION (IPEXTID_ CHANGE_MODE)	H.323, SIP 1PCC
IPSET_ TRANSACTION	IPPARM_ TRANSACTION_ID	---	---	**gc_Extension( )** (Any ext_id)	both
IPSET_ TUNNELED SIGNALMSG	IPPARM TUNNELEDSIGNAL MSG_ALTERNATEID	GC_MAKECALL_BLK for **gc_MakeCall( )** **gc_SetUserInfo( )** for other functions	**gc_MakeCall( )**, **gc_CallAck( )**, **gc_AcceptCall( )**, **gc_AnswerCall( )**, **gc_DropCall( )**, **gc_Extension (IPEXTID_SEND MSG)** for Q.931 Facility message	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG) GCEV_EXTENSION (IPEXTID_ RECEIVEMSG) for Facility message only	H.323
	IPPARM TUNNELEDSIGNAL MSG_CONTENT	GC_MAKECALL_BLK for **gc_MakeCall( )** **gc_SetUserInfo( )** for other functions	**gc_MakeCall( )**, **gc_CallAck( )**, **gc_AcceptCall( )**, **gc_AnswerCall( )**, **gc_DropCall( )**, **gc_Extension (IPEXTID_SEND MSG)** for Q.931 Facility message	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG) GCEV_EXTENSION (IPEXTID_ RECEIVEMSG) for Facility message only	H.323

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
	IPPARM TUNNELEDSIGNAL MSG_NSDATA_DATA	GC_MAKECALL_BLK for **gc_MakeCall( )** **gc_SetUserInfo( )** for other functions	**gc_MakeCall( )**, **gc_CallAck( )**, **gc_AcceptCall( )**, **gc_AnswerCall( )**, **gc_DropCall( )**, **gc_Extension (IPEXTID_SEND MSG)** for Facility message only	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG) GCEV_EXTENSION (IPEXTID_ RECEIVEMSG) for Facility message only	H.323
	IPPARM TUNNELEDSIGNAL MSG_NSDATA_ H221NS	GC_MAKECALL_BLK for **gc_MakeCall( )**; **gc_SetUserInfo( )** for other functions	**gc_MakeCall( )**, **gc_CallAck( )**, **gc_AcceptCall( )**, **gc_AnswerCall( )**, **gc_DropCall( )**, **gc_Extension (IPEXTID_SEND MSG)** for Facility message only	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG) GCEV_EXTENSION (IPEXTID_ RECEIVEMSG) for Facility message only	H.323
	IPPARM TUNNELEDSIGNAL MSG_NSDATA_ OBJID	GC_MAKECALL_BLK for **gc_MakeCall( )**; **gc_SetUserInfo( )** for other functions	**gc_MakeCall( )**, **gc_CallAck( )**, **gc_AcceptCall( )**, **gc_AnswerCall( )**, **gc_DropCall( )**, **gc_Extension (IPEXTID_SEND MSG)** for Facility message only	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG) GCEV_EXTENSION (IPEXTID_ RECEIVEMSG) for Facility message only	H.323
IPSET_ TUNNELED SIGNALMSG	IPPARM TUNNELEDSIGNAL MSG_PROTOCOL_ OBJECTID	GC_MAKECALL_BLK for **gc_MakeCall( )**; **gc_SetUserInfo( )** for other functions	**gc_MakeCall( )** **gc_CallAck( )**, **gc_AcceptCall( )**, **gc_AnswerCall( )**, **gc_DropCall( )**, **gc_Extension (IPEXTID_SEND MSG)** for Facility message only	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG) GCEV_EXTENSION (IPEXTID_ RECEIVEMSG) for Facility message only	H.323
	IPPARM TUNNELEDSIGNAL MSG_PROTOCOL_ OBJID (deprecated)	GC_PARM_BLK	**gc_MakeCall( )**	GCEV_ EXTENSIONCMPLT (IPEXTID_ RECEIVEMSG)	H.323

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

Table 43. Summary of Parameter Sets and Parameter Usage (Continued)

Set ID	Parameter ID	Set	Send	Retrieve	SIP/ H.323
IPSET_ VENDORINFO	IPPARM_ H221NONSTD	**gc_SetConfigData( )**	**gc_Extension( )** (IPEXTID_ SENDMSG)	**gc_Extension( )** (IPEXTID_GETINFO)	H.323
	IPPARM_ VENDOR_ PRODUCT_ID	**gc_SetConfigData( )**	**gc_Extension( )** (IPEXTID_ SENDMSG)	**gc_Extension( )** (IPEXTID_GETINFO)	H.323
	IPPARM_ VENDOR_ VERSION_ID	**gc_SetConfigData( )**	**gc_Extension( )** (IPEXTID_ SENDMSG)	**gc_Extension( )** (IPEXTID_GETINFO)	H.323

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a line device basis).
‡ Tunneling for incoming calls can only be specified using the **gc_SetConfigData( )** function with a board device target ID.

9.2 Parameter Set Reference

This section contains reference information on the parameters in a parameter set used for IP telephony under Global Call. The table in each of the following subsections lists and describes the individual parameters associated with the parameter set as well as indicating the data type, size, and defined values for the parameters.

The parameter sets documented in this section include:

- GCSET_CALL_CONFIG
- IPSET_CALLINFO
- IPSET_CONFERENCE
- IPSET_CONFIG
- IPSET_DTMF
- IPSET_EXTENSIONEVT_MSK
- IPSET_FOIP
- IPSET_H323_RESPONSE_CODE
- IPSET_IPPROTOCOL_STATE
- IPSET_LOCAL_ALIAS
- IPSET_MEDIA_STATE
- IPSET_MIME and IPSET_MIME_200OK_TO_BYE
- IPSET_MIME_ACK_TO_REJECTION
- IPSET_MSG_H245
- IPSET_MSG_Q931
- IPSET_MSG_REGISTRATION
- IPSET_MSG_SIP
- IPSET_NONSTANDARDCONTROL

- IPSET_NONSTANDARDDATA
- IPSET_PROTOCOL
- IPSET_PROXY_INFO
- IPSET_REG_INFO
- IPSET_RTP_ADDRESS
- IPSET_SDP
- IPSET_SIP_MSGINFO
- IPSET_SIP_REQUEST_ERROR
- IPSET_SIP_RESPONSE_CODE
- IPSET_SIP_SESSION_TIMER
- IPSET_SUPPORTED_PREFIXES
- IPSET_SWITCH_CODEC
- IPSET_TRANSACTION
- IPSET_TUNNELEDSIGNALMSG
- IPSET_VENDORINFO

9.2.1 GCSET_CALL_CONFIG

Table 44 shows the parameter IDs in the GCSET_CALL_CONFIG parameter set that are relevant in an IP context.

Table 44. GCSET_CALL_CONFIG Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
GCPARM_CALLPROC	Type: enumeration Size: sizeof(char) Values: • GCCONTROL_APP - The application must use **gc_CallAck( )** to send the Proceeding message. This is the default. • GCCONTROL_TCCL - The stack sends the Proceeding message automatically.	Used to specify if the Proceeding message is sent under application control or automatically by the stack	both

9.2.2 IPSET_CALLINFO

Table 45 shows the parameter IDs in the IPSET_CALLINFO parameter set.

Table 45. IPSET_CALLINFO Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_BEARERCAP	Type: string † Size: max. length = 255	Bearer Capability IE	H.323
IPPARM_CALLDURATION	Type: unsigned int Size: sizeof(unsigned int)	Duration of the call	H.323
IPPARM_CALLID	Type for SIP: string † Size for SIP: max. length = MAX_IP_SIP_CALLID_LENGTH Type for H.323: array of octets Size for H.323: MAX_IP_H323_ CALLID_LENGTH If protocol is unknown, MAX_IP_ CALLID_LENGTH defines the maximum Call ID length for any supported protocol.	Globally unique identifier (Call ID) used by the underlying protocol to identify the call **Note:** When using SIP, direct manipulation of the Call ID message header via IPSET_SIP_MSGINFO / IPPARM_CALLID_HDR overrides any value provided via this parameter.	both
IPPARM_CDPN_ NUMBERING_PLAN_ID	Type: unsigned char Size: 1 byte	Contains the numbering plan ID in the CDPN	H.323
IPPARM_CDPN_ TYPE_OF_NUMBER	Type: unsigned char Size: 1 byte	Contains the type of number in the CDPN	H.323
IPPARM_CGPN_ NUMBERING_PLAN_ID	Type: unsigned char Size: 1 byte	Contains the numbering plan ID in the CGPN	H.323
IPPARM_CGPN_ PRESENTATION_ INDICATOR	Type: unsigned char Size: 1 byte	Contains the presentation indicator in the CGPN	H.323
IPPARM_CGPN_ SCREENING_ INDICATOR	Type: unsigned char Size: 1 byte	Contains the screening indicator in the CGPN	H.323
IPPARM_CGPN_ TYPE_OF_NUMBER	Type: unsigned char Size: 1 byte	Contains the type of number in the CGPN	H.323
IPPARM_ CONNECTIONMETHOD	Type: enumeration Size: sizeof(char) Values: • IP_CONNECTIONMETHOD_ FASTSTART (default) • IP_CONNECTIONMETHOD_ SLOWSTART	The connection method: Fast Start or Slow Start. See Section 4.2, "Fast and Slow Call Setup Modes", on page 108 for more information. This parm ID is not supported in 3PCC mode.	H.323, SIP 1PCC
IPPARM_DISPLAY	Type: string † Size: max. length = MAX_DISPLAY_LENGTH (82), null-terminated	Display information. This information can be used by a peer as additional address information.	both
† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.			

Table 45. IPSET_CALLINFO Parameter Set (Continued)

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_FACILITY	Type: string † Size: max. length = 255	Facility IE associated with SETUP, CONNECT, or FACILITY message. A Global Call Extension ID of EXTID_RECEIVEMSG applies when the IE is in an incoming FACILITY message.	H.323
IPPARM_FASTSTART_ MANDATORY_H245CH	Type: enumeration Size: sizeof(char) Values: • IP_FASTSTART_MANDATORY _H245CH_ON (default) • IP_FASTSTART_MANDATORY _H245CH_OFF	Specifies whether establishment of H.245 channel is mandatory when using H.323 fast start call setup.	H.323
IPPARM_H245TUNNELING	Type: enumeration Size: sizeof(char) Values: • IP_H245TUNNELING_ON • IP_H245TUNNELING_OFF	Specify if tunneling is on or off. For details, see Section 4.1.3, "Enabling and Disabling H.245 Tunneling (H.323)", on page 107.	H.323
IPPARM_ MEDIAWAITFORCONNECT	Size: sizeof(char) Values: • 0 = FALSE • 1 = TRUE	MediaWaitForConnect field in SETUP message.	H.323
IPPARM_OFFERED_ FASTSTART_CODER	Type: IP_CAPABILITY structure Size: sizeof(IP_CAPABILITY)	Coder information received in a FastStart offer if enabled in IP_VIRTBOARD via sip_msginfo_mask and/or h323_msginfo_msk. For details, see Section 4.2.5, "Retrieving Coder Information from Call Offers", on page 112.	H.323, SIP 1PCC
IPPARM_PHONELIST	Type: string † Size: max. length = MAX_ADDRESS_LENGTH (128)	Phone numbers that can be retrieved at the remote end point. **Note:** When issuing a **gc_MakeCall( )**, this information can also be sent through the **numberstr** parameter. See Section 8.3.17, "gc_MakeCall( ) Variances for IP", on page 560 for more information.	both

† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.

Table 45. IPSET_CALLINFO Parameter Set (Continued)

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_ PRESENTATION_IND	Type: enumeration Size: sizeof(char) Values: • IP_PRESENTATIONALLOWED • IP_PRESENTATION RESTRICTED	PresentationIndicator field in incoming and outgoing SETUP messages. An application may use this field to control whether the Caller ID is presented to the user.	H.323
IPPARM_PROGRESS_IND	Type: string † Size: max. length = 255	Progress Indicator IE in incoming PROGRESS messages. **Note:** Extension events for PROGRESS messages are masked by default. Enable via **gc_SetUserInfo( )** with parameter IPSET_ EXTENSIONEVT_MSK, GCACT_SETMSK, EXTENSIONEVT_CALL_ PROGRESS)	H.323
IPPARM_SIP_ PRACK_MANDATORY	Type: unsigned char Size: sizeof(char)	Specifies whether PRACK handling is mandatory or optional for outbound SIP call.	SIP
IPPARM_SIP_ TRANSPORT_ADDR	Type: IP_SIP_TRANSPORT_ADDR Size: sizeof (IP_SIP_TRANSPORT_ADDR)	Extracts the remote SIP transport layer address upon receipt of incoming INVITE and re-INVITE messages.	SIP
IPPARM_USERUSER_INFO	Type: unsigned char[ ] Size: max size = MAX_USERUSER_INFO_ LENGTH (131)	User-to-user information	H.323
IPPARM_UUIE_ASN1	Type: unsigned char[ ] Size: max. length = max_parm_data_size (configured at start-up via IPCCLIB_START_DATA)	User-to-User Information Element (UU-IE) in raw, ASN1 encoded format.	H.323
† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.			

9.2.3 IPSET_CONFERENCE

Table 46 shows the parameter IDs in the IPSET_CONFERENCE parameter set.

Table 46. IPSET_CONFERENCE Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_CONFERENCE_GOAL	Type: enumeration Size: sizeof(char) Values: • IP_CONFERENCEGOAL_UNDEFINED • IP_CONFERENCEGOAL_CREATE • IP_CONFERENCEGOAL_JOIN • IP_CONFERENCEGOAL_INVITE • IP_CONFERENCEGOAL_ CAP_NEGOTIATION • IP_CONFERENCEGOAL_ SUPPLEMENTARY_SRVC	The conference functionality to be achieved	H.323
IPPARM_CONFERENCE_ID	Type: string † Size: max. length = IP_CONFERENCE_ID_LENGTH (16)	The conference identifier	H.323

1. For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.
2. Conference ID retrieval is only relevant when an application is in a conference. In a peer-to-peer call, the conference ID does not signify a call identifier. The application should use IPPARM_CALLID to retrieve the call identifier. See Section 9.2.2, "IPSET_CALLINFO", on page 615 for more information.

9.2.4 IPSET_CONFIG

Table 47 shows the parameter IDs in the IPSET_CONFIG parameter set.

Table 47. IPSET_CONFIG Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_1PCC_ REJECT_VIDEO	Type: None Size: 0	Selects rejection via 488 Not Acceptable Here response for any session offer requesting video media. Selection is persistent until library is re-started.	SIP
IPPARM_AUTHENTICATION_ CONFIGURE	Type: IP_ AUTHENTICATION Size: sizeof(IP_ AUTHENTICATION)	Used to add or modify a SIP authentication quadruplet. This parameter is only valid for the **gc_SetAuthenticationInfo( )** function.	SIP
IPPARM_AUTHENTICATION_ REMOVE	Type: IP_ AUTHENTICATION Size: sizeof(IP_ AUTHENTICATION)	Used to remove a SIP authentication quadruplet based on the realm and identity strings in IP_AUTHENTICATION; the username and password. This parameter is only valid for the **gc_SetAuthenticationInfo( )** function.	SIP

† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.

Table 47. IPSET_CONFIG Parameter Set (Continued)

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_CONFIG_TOS	Type: char Size: sizeof(char)	Deprecated. Used to set the TOS byte in IPv4 packet headers. Byte may be set as TOS/IP Precedence byte or DiffServ field (DSCP). Valid values are in the range 0 to 255. The default value is 0.	both
IPPARM_ GETCALLINFOUPDATE	Type: int Size: sizeof(int)	Enables retrieval of SIP "To tag" for inbound calls. See Section 4.9.9, "Retrieving SIP "To tag" for Inbound Calls", on page 194.	SIP
IPPARM_H323_AUTO_ PROGRESS_DISABLE	Type: None Size: 0	Used to disable H323 automatic PROGRESS message after ALERTING on virtual board. See Section 4.43, "Accessing H.323 ALERTING Progress Indicator Information Element", on page 402 for more information.	H.323
IPPARM_IPMPARM	Type: IPM_PARM_INFO Size: sizeof(IPM_PARM_INFO)	Used to set IP Media Library parameters (e.g., TOS byte or echo cancellation parameters) on a pass-through basis (no checking or validating by Global Call).	both
IPPARM_MIME_OVERLAP_ RECEIVE	Type: int Size: sizeof(int)	Enables the overlap receive feature of SIP-I based on embedded MIME ISUP messages. See Section 4.11, "MIME-Based Overlap Receive Support for Limited SIP-I Interworking Scenarios", on page 218 for information.	SIP
IPPARM_OPERATING_MODE	Type: int Size: sizeof(int)	Sets the method used to transition to/ from T.38 fax mode in 1PCC mode; also enables/disables access to SIP re-INVITE messages in both 1PCC and 3PCC modes. Possible values are: • IP_T38_AUTOMATIC_MODE – Default mode. A request to transition to or from T.38 fax gateway mode is handled automatically without application involvement. This mode disables application handling of SIP re-INVITE messages. • IP_T38_MANUAL_MODE – A request to transition to or from T.38 fax server mode is reported as a GCEV_EXTENSION event with an IPSET_SWITCH_CODEC parameter. This mode must be set to enable application handling of SIP re-INVITE messages.	both
IPPARM_OVERLAP_ SIGNALING	Type: int Size: size of (int)	Enables the overlap feature. For details, see Section 4.12, "Overlap Send and Receive Support", on page 223.	SIP

† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.

Table 47. IPSET_CONFIG Parameter Set (Continued)

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_REGISTER_SIP_ HEADER	Type: string † Size: max. length = IP_SIP_HDR_ MAXLEN (255)	Used to register the names of SIP message header fields that the application needs to retrieve from incoming messages	SIP
IPPARM_SEND_G723_ UNSPECIFIED_BITRATE	Type: int Size: sizeof(int)	Specifies not to send any bit rate for G.723.1 on outgoing SIP requests or responses. For details, see Section 4.32, "Unspecified G.723.1 Bit Rate in Outgoing SIP Requests with SDP", on page 363.	SIP 1PCC
IPPARM_SEND_SIP_ UPDATE_ POSTCONNECTION	Type: int Size: sizeof(int)	Used to enable or disable handling of SIP UPDATE messages post connection. For details, see Section 4.18.3, "Handling SIP UPDATE Post-Connection Messages", on page 273.	SIP 3PCC
IPPARM_SIGNALING_ DEFERRED	Type: int Size: sizeof(int)	Used to enable or disable deferral of SIP signaling messages when **gc_DropCall()** is issued. For details, see Section 4.22, "Defer the Sending of SIP Messages", on page 291.	SIP
† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.			

9.2.5 IPSET_DTMF

Table 48 shows the parameter IDs in the IPSET_DTMF parameter set. This parameter set is used to set DTMF-related parameters for the notification, suppression or sending of DTMF digits.

This parameter set is not supported in SIP third party call control (3PCC) operating mode.

Table 48. IPSET_DTMF Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ DTMF_ ALPHANUMERIC	Type: IP_DTMF_DIGITS Size: sizeof(IP_DTMF_DIGITS)	Used when sending or receiving DTMF via UII alphanumeric messages using the Global Call extension API. The parameter value contains an IP_DTMF_DIGITS structure that includes the digit string.	H.323, SIP 1PCC
IPPARM_ DTMF_RFC2833_ PAYLOAD_TYPE	Type: unsigned char Size: sizeof(char)	Used to specify the RFC2833 RTP payload type. Valid values are in the range from 96 to 127. The default value is IP_USE_STANDARD_PAYLOADTYPE (101).	H.323, SIP 1PCC
† The IP_DTMF_TYPE_ALPHANUMERIC value, which is the default, is only valid when using H.323. ‡ The inband mode cannot be used reliably with low bit-rate coders.			

Table 48. IPSET_DTMF Parameter Set (Continued)

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ SUPPORT_DTMF_ BITMASK	Type: int Size: sizeof(int)	Used to specify a bitmask that defines which DTMF transmission methods are to be supported. Possible values are: • IP_DTMF_TYPE_ALPHANUMERIC † • IP_DTMF_TYPE_INBAND_RTP ‡ • IP_DTMF_TYPE_RFC_2833	H.323, SIP 1PCC
IPPARM_ TELEPHONY_ EVENT_ DTMF	Type: unsigned char Size: sizeof(char)	Used to enable or disable notification of DTMF digits received within an RTP stream in RFC 2833 format. For details, see Section 4.7.1.2, "GCEV_TELEPHONY_EVENT", on page 159 and Section 4.24.7, "Exposing Raw RFC 2833 Data", on page 305. Possible values are: • IP_ENABLE • IP_DISABLE (default)	H.323, SIP 1PCC
IPPARM_ TELEPHONY_ EVENT_ INFO	Type: IPM_TELEPHONY_INFO structure Size: sizeof(IPM_TELEPHONY _INFO)	Used to fetch raw RFC 2833 DTMF digits from the GCEV_TELEPHONY_EVENT event. For details, see Section 4.24.2.2, "Processing RFC 2833 Events", on page 301 and Section 4.24.7, "Exposing Raw RFC 2833 Data", on page 305. The value is IPM_TELEPHONY_INFO.	H.323, SIP 1PCC
† The IP_DTMF_TYPE_ALPHANUMERIC value, which is the default, is only valid when using H.323. ‡ The inband mode cannot be used reliably with low bit-rate coders.			

9.2.6 IPSET_EXTENSIONEVT_MSK

This parameter set is used to enable or disable the events associated with unsolicited notification such as the detection of DTMF or a change of connection state in an underlying protocol. Table 49 shows the parameter IDs in the IPSET_EXTENSIONEVT_MSK parameter set.

Table 49. IPSET_EXTENSIONEVT_MSK Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
GCPARM_GET_MSK	Type: int Size: sizeof(int)	Retrieve the bitmask of enabled events	both
GCACT_SETMSK	Type: int Size: sizeof(int)	Set the bitmask of enabled events.	both
GCACT_ADDMSK	Type: int Size: sizeof(int)	Add to the bitmask of enabled events	both
GCACT_SUBMSK	Type: int Size: sizeof(int)	Remove from the bitmask of enabled events	both
Values that can be used to make up the bitmask are: • EXTENSIONEVT_DTMF_ALPHANUMERIC (0x04) † • EXTENSIONEVT_SIGNALING_STATUS (0x08) • EXTENSIONEVT_SIP_18x_RESPONSE • EXTENSIONEVT_STREAMING_STATUS (0x10) • EXTENSIONEVT_T38_STATUS (0x20)			

9.2.7 IPSET_FOIP

Table 50 shows the parameter IDs in the IPSET_FOIP parameter set.

This parameter set is not supported in the SIP third party call control (3PCC) operating mode.

Table 50. IPSET_FOIP Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_T38_OFFERED	Type: IP_CONNECT Size: sizeof(IP_CONNECT)	Used in a GC_PARM_BLK associated with an GCEV_OFFERED event to indicate that a T.38 session is requested.	H.323, SIP 1PCC
IPPARM_T38_CONNECT	Type: IP_CONNECT Size: sizeof(IP_CONNECT)	Used when associating a T.38 Fax device with a Media device when switching from an audio session to a fax session.	H.323, SIP 1PCC
IPPARM_T38_DISCONNECT	Type: IP_CONNECT Size: sizeof(IP_CONNECT)	Used when disassociating a T.38 Fax device with a Media device when switching from a fax session to an audio session.	H.323, SIP 1PCC

9.2.8 IPSET_H323_RESPONSE_CODE

This parameter set is used to set the busy cause code that is used in the failure message sent when the local system is unable to accept additional incoming sessions.

Table 51. IPSET_H323_RESPONSE_CODE Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_ BUSY_CAUSE	Type: eIP_EC_TYPE Size: sizeof(int)	Used in a GC_PARM_BLK to specify the cause code to send when no additional incoming sessions can be accepted. Values: • IPEC_Q931Cause34NoCircuitChannelAvailable • IPEC_Q931Cause47ResourceUnavailableUnspecified	H.323

9.2.9 IPSET_IPPROTOCOL_STATE

This parameter set is used when retrieving notification of protocol signaling states via GCEV_EXTENSION events with extension ID IPEXTID_IPPROTOCOL_STATE. Table 52 shows the parameter IDs in the IPSET_IPPROTOCOL_STATE parameter set.

Table 52. IPSET_IPPROTOCOL_STATE Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ CONTROL_CONNECTED	Type: int Size: sizeof(int)	Media control signaling for the call has been established with the remote endpoint	H.323
IPPARM_ CONTROL_DISCONNECTED	Type: int Size: sizeof(int)	Media control signaling for the call has been terminated	H.323
IPPARM_ EST_CONTROL_FAILED	Type: int Size: sizeof(int)	Establishment failed for optional H.245 channel in fast start connection mode	H.323
IPPARM_ SIGNALING_CONNECTED	Type: int Size: sizeof(int)	Call signaling for the call has been established with the remote endpoint	H.323
IPPARM_ SIGNALING_DISCONNECTED	Type: int Size: sizeof(int)	Call signaling for the call has been terminated	H.323

9.2.10 IPSET_LOCAL_ALIAS

Table 53 shows the parameter IDs in the IPSET_LOCAL_ALIAS parameter set.

Table 53. IPSET_LOCAL_ALIAS Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ ADDRESS_DOT_NOTATION	Type: string † Size: max. length = 255	A valid IP address	both
IPPARM_ ADDRESS_EMAIL	Type: string † Size: max. length = 255	e-mail address composed of characters from the set "[A-Z][a-z][0-9]_-.@"	both
IPPARM_ ADDRESS_H323_ID	Type: string † Size: max. length = 255	A valid H.323 ID	H.323
IPPARM_ ADDRESS_PHONE	Type: string † Size: max. length = 255	An E.164 telephone number	H.323
IPPARM_ ADDRESS_TRANSPARENT	Type: string † Size: max. length = 255	Unspecified address type	both
IPPARM_ ADDRESS_URL	Type: string † Size: max. length = 255	A valid URL composed of characters from the set "[A-Z][a-z][0-9]-.". Must contain at least one "." and may not begin or end with a "-".	H.323
† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.			

Note: For SIP, IPSET_LOCAL_ALIAS is not used for the alias (or Address of Record), but is used for the transport address or contact.

9.2.11 IPSET_MEDIA_STATE

Table 54 shows the parameter IDs in the IPSET_MEDIA_STATE parameter set. These parameters dispatched to the application in GCEV_EXTENSION events of type IPEXTID_MEDIAINFO. In all cases where the parameter data is an IP_CAPABILITY structure, the structure contains the coder capabilities that were negotiated with the remote peer.

This parameter set is not supported in the SIP third party call control (3PCC) operating mode.

Table 54. IPSET_MEDIA_STATE Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ RX_CONNECTED	Type: IP_CAPABILITY Size: sizeof(IP_CAPABILITY)	Streaming in the receive direction (from the remote endpoint) has been initiated. See Section 4.7.1, "Enabling and Disabling Unsolicited Notification Events", on page 158 for more information.	H.323, SIP 1PCC
IPPARM_ RX_DISCONNECTED	Type: None Size: 0	Streaming in the receive direction (from the remote endpoint) has been terminated. Any data associated with this parameter ID is ignored.	H.323, SIP 1PCC
IPPARM_ RX_INACTIVE	Type: IP_CAPABILITY Size: sizeof(IP_CAPABILITY)	Streaming in the receive direction (from the remote endpoint) is inactive, i.e. has been placed on hold.	H.323, SIP 1PCC
IPPARM_ RX_RECVONLY	Type: IP_CAPABILITY Size: sizeof(IP_CAPABILITY)	Streaming for a half-duplex, receive-only connection (from the remote endpoint) has been initiated.	H.323, SIP 1PCC
IPPARM_ TX_CONNECTED	Type: IP_CAPABILITY Size: sizeof(IP_CAPABILITY)	Streaming in the transmit direction (toward the remote endpoint) has been initiated. See Section 4.7.1, "Enabling and Disabling Unsolicited Notification Events", on page 158 for more information.	H.323, SIP 1PCC
IPPARM_ TX_DISCONNECTED	Type: None Size: 0	Streaming in the transmit direction (toward the remote endpoint) has been terminated. Any data associated with this parameter ID is ignored.	H.323, SIP 1PCC
IPPARM_ TX_INACTIVE	Type: IP_CAPABILITY Size: sizeof(IP_CAPABILITY)	Streaming in the transmit direction (toward the remote endpoint) is inactive, i.e. has been placed on hold.	H.323, SIP 1PCC
IPPARM_ TX_SENDONLY	Type: IP_CAPABILITY Size: sizeof(IP_CAPABILITY)	Streaming for a half-duplex, send-only connection (to the remote endpoint) has been initiated.	H.323, SIP 1PCC

9.2.12 IPSET_MIME and IPSET_MIME_200OK_TO_BYE

Table 55 shows the parameter IDs in the IPSET_MIME and IPSET_MIME_200OK_TO_BYE parameter sets which are used when sending and receiving MIME-encoded SIP messages. The same parameters apply to both parameter sets. When using the IPSET_MIME_200OK_TO_BYE parameter set ID, that same set ID must be used in all parameter elements in all data blocks associated with the message.

Table 55. IPSET_MIME and IPSET_MIME_200OK_TO_BYE Parameter Sets

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_MIME_PART	Type: pointer to GC_PARM_BLK Size: 4 bytes	Required parameter. Used to set or get SIP message MIME part(s). Parameter value is a pointer to a GC_PARM_BLK structure that contains a list of pointers to one or more GC_PARM_BLK structures that contain MIME message parts.	SIP
IPPARM_ MIME_PART_BODY	Type: char * Size: 4 bytes	Required parameter. Used to copy MIME part body between application and Global Call space. Parameter value is a pointer to a MIME part body.	SIP
IPPARM_ MIME_PART_BODY_SIZE	Type: Unsigned int Size: 4 bytes	Required parameter. Used to indicate the actual size of the MIME part body, not including MIME part headers.	SIP
IPPARM_ MIME_PART_HEADER	Type: Null-terminated string † Size: max. length = max_parm_data_size (configured at start-up via IPCCLIB_START_DATA)	Optional parameter. Used to contain MIME part header field in format of “field-name: field-value”. Field-name can be any string other than “Content-type”. Content is not checked by Global Call before insertion into SIP message.	SIP
IPPARM_ MIME_PART_TYPE	Type: Null-terminated string † Size: max. length = max_parm_data_size (configured at start-up via IPCCLIB_START_DATA)	Required parameter. Used to contain name and value of the MIME part content type field. String must begin with the field name “Content-Type:”.	SIP

† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.

9.2.13 IPSET_MIME_ACK_TO_REJECTION

The following table shows the parameter IDs in the IPSET_MIME_ACK_TO_REJECTION parameter set. This parameter set is used to embed a MIME body in an outgoing SIP ACK request message for 4xx/5xx/6xx response messages that terminate certain transactions.

Table 56. IPSET_MIME_ACK_TO_REJECTION Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_MIME_PART	Type: pointer to GC_PARM_BLK Size: 4 bytes	Required parameter. Used to set or get SIP message MIME part(s). Parameter value is a pointer to a GC_PARM_BLK structure that contains a list of pointers to one or more GC_PARM_BLK structures that contain MIME message parts.	SIP
IPPARM_ MIME_PART_BODY	Type: char * Size: 4 bytes	Required parameter. Used to copy MIME part body between application and Global Call space. Parameter value is a pointer to a MIME part body.	SIP
IPPARM_ MIME_PART_BODY_SIZE	Type: Unsigned int Size: 4 bytes	Required parameter. Used to indicate the actual size of the MIME part body, not including MIME part headers.	SIP
IPPARM_ MIME_PART_HEADER	Type: Null-terminated string † Size: max. length = max_parm_data_size (configured at start-up via IPCCLIB_START_DATA)	Optional parameter. Used to contain MIME part header field in format of "field-name: field-value". Field-name can be any string other than "Content-type". Content is not checked by Global Call before insertion into SIP message.	SIP
IPPARM_ MIME_PART_TYPE	Type: Null-terminated string † Size: max. length = max_parm_data_size (configured at start-up via IPCCLIB_START_DATA)	Required parameter. Used to contain name and value of the MIME part content type field. String must begin with the field name "Content-Type:".	SIP
† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.			

9.2.14 IPSET_MSG_H245

Table 57 shows the parameter IDs in the IPSET_MSG_H245 parameter set. This parameter set is used with the **gc_Extension( )** and the IPEXTID_SENDMSG extension and encapsulates all the parameters required to send an H.245 message.

Table 57. IPSET_MSG_H245 Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_MSGTYPE	Type: int Size: sizeof(int)	Possible values for H.245 messages are: • IP_MSGTYPE_H245_INDICATION	H.323

9.2.15 IPSET_MSG_Q931

Table 58 shows the parameter IDs in the IPSET_MSG_Q931 parameter set. This parameter set is used with the **gc_Extension( )** and the IPEXTID_SENDMSG extension and encapsulates all the parameters required to send or receive a Q.931 message.

Table 58. IPSET_MSG_Q931 Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_MSGTYPE	Type: int Size: sizeof(int)	Possible values for Q.931 messages are: • IP_MSGTYPE_Q931_FACILITY • IP_MSGTYPE_Q931_PROGRESS	H.323

9.2.16 IPSET_MSG_REGISTRATION

Table 59 shows the parameter IDs in the IPSET_MSG_REGISTRATION parameter set. This parameter set is used with the **gc_Extension( )** and the IPEXTID_SENDMSG extension and encapsulates all the parameters required to send a registration message. For information on the use of this parameter set, see Section 4.44.3, "Nonstandard Registration Message", on page 408.

Table 59. IPSET_MSG_REGISTRATION Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_MSGTYPE	Type: int Size: sizeof(int)	Possible value for registration messages is: • IP_MSGTYPE_REG_NONSTD	both

9.2.17 IPSET_MSG_SIP

Table 60 shows the parameter IDs in the IPSET_MSG_SIP parameter set. This parameter set is used to set the response code or message type for outgoing SIP messages. In most cases, the parameter set is also used to identify the message type for SIP messages that are passed to the application in Global Call events.

Table 60. IPSET_MSG_SIP Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_MSG_SIP_ RESPONSE_CODE	Type: int Size: sizeof(int)	Used to set the numerical response code to send in a SIP response message, or to extract the code from a received response message.	SIP
IPPARM_MSGTYPE	Type: int Size: sizeof(int)	Sets type of supported SIP message to send using **gc_Extension( )** and the IPEXTID_SENDMSG extension ID. Also used to identify the type of SIP message that is passed to the application as a GCEV_EXTENSION event (or GCEV_CALLINFO event in the case of INFO messages only). Defined values are: • IP_MSGTYPE_SIP_INFO • IP_MSGTYPE_SIP_INFO_FAILED • IP_MSGTYPE_SIP_INFO_OK • IP_MSGTYPE_SIP_NOTIFY • IP_MSGTYPE_SIP_NOTIFY_ACCEPT • IP_MSGTYPE_SIP_NOTIFY_REJECT • IP_MSGTYPE_SIP_OPTIONS • IP_MSGTYPE_SIP_OPTIONS_FAILED • IP_MSGTYPE_SIP_OPTIONS_OK • IP_MSGTYPE_SIP_SUBSCRIBE • IP_MSGTYPE_SIP_SUBSCRIBE_ACCEPT • IP_MSGTYPE_SIP_SUBSCRIBE_EXPIRE (receive only) • IP_MSGTYPE_SIP_SUBSCRIBE_REJECT • IP_MSGTYPE_SIP_UPDATE • IP_MSGTYPE_SIP_UPDATE_FAILED • IP_MSGTYPE_SIP_UPDATE_OK	SIP
IPPARM_SIP_ METHOD	Type: int Size: sizeof(int)	Type of SIP method to send. Defined values are: • IP_MSGTYPE_SIP_CANCEL – sends CANCEL method. Only supported for cancelling pending re-INVITE via **gc_ReqModifyCall( )** function.	SIP 3PCC

9.2.18 IPSET_NONSTANDARDCONTROL

Table 61 shows the parameter IDs in the IPSET_NONSTANDARDCONTROL parameter set.

Table 61. IPSET_NONSTANDARDCONTROL Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ NONSTANDARDDATA_DATA	Type: string † Size: max. length = max_parm_data_size ‡ (configured at start-up via IPCCLIB_START_DATA)	Used to contain the nonstandard data.	H.323
IPPARM_ NONSTANDARDDATA_OBJID	Type: Uint[ ] Size: max. length = MAX_NS_PARM_ OBJID_LENGTH (40)	Used to contain a nonstandard object ID, if any. If an H.221 nonstandard data identifier is being used, this parameter should not be present in the parm block.	H.323
IPPARM_H221NONSTANDARD	Type: IP_H221NONSTANDARD Size: sizeof(IP_H221NONSTANDARD)	Used to contain a H.221 nonstandard data identifier, if any. If a nonstandard object ID is being used, this parameter should not be present in the parm block.	H.323
† For parameters with data of type String, the length of in a GC_PARM_BLK is the length of the data string plus 1. ‡ The full maximum length that is configured may not be usable in practice because the H.323 stack limits total message size to max_parm_data_size + 512 bytes. Longer messages are truncated without notification to the application.			

9.2.19 IPSET_NONSTANDARDDATA

Table 62 shows the parameter IDs in the IPSET_NONSTANDARDDATA parameter set.

Table 62. IPSET_NONSTANDARDDATA Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ NONSTANDARDDATA_DATA	Type: string † Size: max. length = max_parm_data_size ‡ (configured at start-up via IPCCLIB_START_DATA)	Used to contain the nonstandard data.	H.323
IPPARM_ NONSTANDARDDATA_OBJID	Type: Uint[ ] Size: max. length = MAX_NS_PARM_OBJID_ LENGTH (40)	Used to contain a nonstandard object ID, if any. If an H.221 nonstandard data identifier is being used, this parameter should not be present in the parm block.	H.323
IPPARM_H221NONSTANDARD	Type: IP_H221NONSTANDARD Size: sizeof(IP_H221NONSTANDARD)	Used to contain an H.221 nonstandard data identifier, if any. If a nonstandard object ID is being used, this parameter should not be present in the parm block.	H.323
† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1. ‡ The full maximum length that is configured may not be usable in practice because the H.323 stack limits total message size to max_parm_data_size + 512 bytes. Longer messages are truncated without notification to the application.			

9.2.20 IPSET_PROTOCOL

Table 63 shows the parameter IDs in the IPSET_PROTOCOL parameter set.

Table 63. IPSET_PROTOCOL Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_PROTOCOL_BITMASK	Type: char Size: sizeof(char)	The IP protocol to use. Defined values (which may be OR'ed) are: • IP_PROTOCOL_H323 • IP_PROTOCOL_SIP	both

9.2.21 IPSET_PROXY_INFO

Table 64 shows the parameter IDs in the IPSET_PROXY_INFO parameter set.

Table 64. IPSET_PROXY_INFO Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_PROXY_ ACTION	Type: char Size: sizeof(char)	Determines whether to use the proxy for a specific SIP Register request. Possible values are: • IP_PROXY_USE • IP_PROXY_BYPASS • IP_PROXY_SELECT • IP_PROXY_DESELECT	SIP
IPPARM_PROXY_ INFO	Type: char Size: sizeof(char)	Denotes the proxy information structure. Possible value: • IP_PROXY_INFO data structure	SIP

9.2.22 IPSET_REG_INFO

Table 65 shows the parameter IDs in the IPSET_REG_INFO parameter set.

Table 65. IPSET_REG_INFO Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_OPERATION_ REGISTER	Type: char Size: sizeof(char)	Used to specify the type of registration operation to perform with a gatekeeper or registrar. Possible values are: • IP_REG_ADD_INFO • IP_REG_DELETE_BY_VALUE • IP_REG_QUERY_INFO (SIP only) • IP_REG_SET_INFO	both
IPPARM_OPERATION_ DEREGISTER	Type: char Size: sizeof(char)	Used when deregistering an endpoint with a gatekeeper/registrar. Possible values are: • IP_REG_DELETE_ALL – Discard the registration data in the local database • IP_REG_MAINTAIN_LOCAL_INFO – Keep the registration data in the local database	both
IPPARM_REG_ADDRESS	Type: IP_REGISTER_ ADDRESS Size: sizeof(IP_REGISTER_ ADDRESS)	Address information to be registered with a gatekeeper/registrar. See the reference page for IP_REGISTER_ADDRESS on page 661 for details.	both
IPPARM_REG_ AUTOREFRESH	Type: char Size: sizeof(char)	Used to enable/disable autorefresh of SIP registration bindings. Possible values are: • IP_REG_AUTOREFRESH_DISABLE • IP_REG_AUTOREFRESH_ENABLE Default behavior if this parameter is not specified is to autorefresh bindings.	SIP

Table 65. IPSET_REG_INFO Parameter Set (Continued)

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_REG_TYPE	Type: int Size: sizeof(int)	The registration type. Possible values are: • IP_REG_GATEWAY • IP_REG_TERMINAL	H.323
IPPARM_REG_SERVICEID	Type: int Size: sizeof(int)	The Service ID that was handed back to the application when it initiated the registration	SIP
IPPARM_REG_STATUS	Type: char Size: sizeof(char)	Indicates whether or not the endpoint's registration with a gatekeeper/registrar was successful. Possible values are: • IP_REG_CONFIRMED • IP_REG_REJECTED	both

9.2.23 IPSET_RTP_ADDRESS

Table 65 shows the parameter IDs in the IPSET_RTP_ADDRESS parameter set.

Table 66. IPSET_RTP_ADDRESS Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_LOCAL	Type: int Size: sizeof(int)	Used when retrieving RTP address of the local endpoint of an RTP stream as contained in a connection event.	both
IPPARM_REMOTE	Type: int Size: sizeof(int)	Used when retrieving RTP address of the remote endpoint of an RTP stream as contained in a connection event.	both

9.2.24 IPSET_SDP

Table 67 shows the parameter IDs in the IPSET_SDP parameter set. The parameter IDs are used to send and receive SDP content.

Parameter blocks that contain the IPSET_SDP set ID are valid only in specific Global Call APIs:

- **gc_MakeCall( )** in the **makecall** parameter block
- **gc_ReqModifyCall( )** in the **parmblk** parameter block
- **gc_AcceptModifyCall( )** in the **parmblk** parameter block
- **gc_SipAck( )** in the **parmblk** parameter block
- **gc_Extension( )** (IPEXTID_SENDMSG) in the **paramblkp** parameter block (only for 200OK to OPTIONS method)

- **gc_SetUserInfo( )** with duration GC_NEXT_OUTBOUND_MSG
  After setting the SDP with this function, the application can then call one of the following functions to send a SIP method containing the SDP: **gc_CallAck( )**, **gc_AcceptCall( )**, **gc_AnswerCall( )**, **gc_RejectModifyCall( )**.

Note: Because the maximum data length for the parameters in this set is not limited to 255 bytes, applications **must** use the "extended" **gc_util_..._ex( )** utility functions to retrieve these parameters from parameter blocks associated with Global Call events.

Table 67. IPSET_SDP Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ SDP_ANSWER	Type: string † Size: ‡	Identifies parameter data as an SDP answer.	SIP 3PCC
IPPARM_ SDP_OFFER	Type: string † Size: ‡	Identifies parameter data as an SDP offer.	SIP 3PCC
IPPARM_ SDP_OPTION_ ANSWER	Type: string † Size: ‡	Identifies parameter data as an SDP answer being exchanged in an OPTIONS transaction (e.g., outside a dialog)	SIP 3PCC
IPPARM_ SDP_OPTION_ OFFER	Type: string † Size: ‡	Identifies parameter data as an SDP offer being exchanged in an OPTIONS transaction (e.g., outside a dialog)	SIP 3PCC
IPPARM_ SDP_IP_TYPE	Type: string Size: ‡	Specifies IP address type. See Section 4.3.5, "Specifying IPv4 or IPv6 Address", on page 124.	SIP 1PCC
† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1. ‡ The maximum data size for all parameters in this set is defined by the max_parm_data_size parameter that is set in the IPCCLIB_START_DATA structure before the system is started.			

9.2.25 IPSET_SIP_MSGINFO

Table 68 shows the parameter IDs in the IPSET_SIP_MSGINFO parameter set. Note that access to SIP message header info fields is disabled by default and must be explicitly enabled by setting the IP_SIP_MSGINFO_ENABLE mask value in the sip_msginfo_mask field of the IP_VIRTBOARD structure before starting the virtual board.

Notes: ***1.*** All parameter IDs in this parameter set are deprecated except IPPARM_SIP_HDR. The deprecated parameter IDs will remain in the IP Call Control Library for backward compatibility, but there will be no further development in relation to these parameter IDs.

2. All of the MAXLEN defines for the deprecated SIP header fields are equated to 255 bytes.

3. The maximum data length for the IPPARM_SIP_HDR parameter ID is not limited to 255 bytes. Applications using this parameter ID **must** use the "extended" **gc_util_..._ex( )** utility functions, which are capable of handling parameter data longer than 255 bytes.

Table 68. IPSET_SIP_MSGINFO Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_CALLID_HDR (deprecated)	Type: string † Size: max length = IP_CALLID_HDR_MAXLEN	Deprecated parameter to set or retrieve the Call-ID header field in SIP messages. **Note:** This parameter overrides any Call-ID value set via IPSET_CALLINFO/ IPPARM_CALLID.	SIP
IPPARM_CONTACT_DISPLAY (deprecated)	Type: string † Size: max length = IP_CONTACT_DISPLAY_ MAXLEN	Deprecated parameter to set or retrieve display name in Contact header field of SIP messages	SIP
IPPARM_CONTACT_URI (deprecated)	Type: string † Size: max length = IP_CONTACT_URI_MAXLEN	Deprecated parameter to set or retrieve URI in Contact header field of SIP messages	SIP
IPPARM_DIVERSION_URI (deprecated)	Type: string † Size: max length = IP_DIVERSION_URI_ MAXLEN	Deprecated parameter to set or retrieve URI in the Diversion header field of SIP messages	SIP
IPPARM_EVENT_HDR (deprecated)	Type: string † Size: max length = IP_EVENT_HDR_MAXLEN	Deprecated parameter to set or retrieve Event header field of SIP messages	SIP
IPPARM_EXPIRES_HDR (deprecated)	Type: string † Size: max length = IP_EXPIRES_HDR_TYPE_ MAXLEN	Deprecated parameter to set or retrieve Expires header field of SIP messages	SIP
IPPARM_FROM (deprecated)	Type: string † Size: max length = IP_FROM_MAXLEN	Deprecated parameter to set or retrieve complete From header field (display name, URI, parameters) of SIP messages	SIP
IPPARM_FROM_DISPLAY (deprecated)	Type: string † Size: max length = IP_FROM_DISPLAY_ MAXLEN	Deprecated parameter to set or retrieve display name in the From header field of SIP messages	SIP
IPPARM_REFERRED_BY (deprecated)	Type: string † Size: max length = IP_REFERRED_BY_ MAXLEN	Deprecated parameter to set or retrieve Referred-By header field in SIP messages	SIP
IPPARM_REPLACES (deprecated)	Type: string † Size: max length = IP_REPLACES_MAXLEN	Deprecated parameter to set or retrieve Replaces parameter in Refer-To header of SIP REFER messages (attended call transfer only)	SIP

† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.

Table 68. IPSET_SIP_MSGINFO Parameter Set (Continued)

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_REQUEST_URI (deprecated)	Type: string † Size: max length = IP_REQUEST_URI_MAXLEN	Deprecated parameter to set Request-URI of SIP messages	SIP
IPPARM_SIP_HDR	Type: string † Size: max length = IP_CFG_PARM_DATA_ MAXLEN	Used to set or retrieve standard or proprietary header fields in SIP messages	SIP
IPPARM_SIP_VIA_HDR_ REPLACE	Type: NULL terminated String Size: string length	Used for enabling Via header replacement. The value is the desired Via header string.	SIP
IPPARM_TO_DISPLAY (deprecated)	Type: string † Size: max length = IP_TO_DISPLAY_MAXLEN	Deprecated parameter to set or retrieve display name in the To header field of SIP messages	SIP
IPPARM_TO (deprecated)	Type: string † Size: max length = IP_TO_MAXLEN	Deprecated parameter to set or retrieve complete To header field (display name, URI, parameters) of SIP messages	SIP

† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.

9.2.26 IPSET_SIP_REQUEST_ERROR

This parameter set is used to indicate that a SIP request has had a transport failure. These parameters are contained in the parameter block associated with GCEV_EXTENSION events that are sent to the application when a SIP request failed.

The parameter value indicates the busy cause code that was used in the failure message sent when the local system is unable to accept additional incoming SIP sessions.

Table 69. IPSET_SIP_REQUEST_ERROR Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_SIP_ DNS_CONTINUE	Type: REQUEST_ ERROR Size: sizeof(REQUEST_ ERROR)	Used in a GCEV_EXTENSION event to indicate that a SIP request had a transport failure and is being retried using address information from the DNS server. The REQUEST_ERROR structure contains an Error field with one of following parameter values to indicate the cause of the transport failure: • IP_SIP_REQUEST_503_RCVD (503 Service Unavailable response received) • IP_SIP_REQUEST_FAILED (general transport error) • IP_SIP_REQUEST_NETWORK_ERROR (network error or local failure) • IP_SIP_REQUEST_TIMEOUT (timeout before response received)	SIP
IPPARM_SIP_ SVC_UNAVAIL	Type: REQUEST_ ERROR Size: sizeof(REQUEST_ ERROR)	Used in a GCEV_EXTENSION event to indicate that a SIP request had a fatal transport failure.The REQUEST_ERROR structure contains an Error field with one of following parameter values to indicate the cause of the transport failure: • IP_SIP_REQUEST_503_RCVD (503 Service Unavailable response received) • IP_SIP_REQUEST_FAILED (general transport error) • IP_SIP_REQUEST_NETWORK_ERROR (network error or local failure) • IP_SIP_REQUEST_RETRY_FAILED (retry logic error; no retry attempted) • IP_SIP_REQUEST_TIMEOUT (timeout before response received)	SIP

9.2.27 IPSET_SIP_RESPONSE_CODE

This parameter set is used for response codes that are contained and used in certain SIP response messages. When setting a response code, the code is set on the board device level, except for IPPARM_ACCEPT_RESP_CODE, which can be set on the board device level or on a per call level.

To set on the board device level, insert the parameter ID in a GC_PARM_BLK and call **gc_SetConfigData( )**. When receiving a response code, the parameter is contained in a GC_PARM_BLK associated with a Global Call event.

To set on a per call level, insert the IPPARM_ACCEPT_RESP_CODE parameter ID in a GC_PARM_BLK and call **gc_SetUserInfo( )**. The response message is sent when the application accepts the call, and is only applicable for a single call instance when using GC_SINGLECALL. Any subsequent calls using that channel revert to the default informational response. You can set the response code for all calls on a specified channel using GC_ALLCALLS.

Table 70. IPSET_SIP_RESPONSE_CODE Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_ACCEPT_ RESP_CODE	Type: Unsigned short Size: sizeof(int)	Used in to specify the Informational response code to send when accepting a call via **gc_AcceptCall( )**. The parameter value can be any integer from 101 to 199, but the only two commonly used values are: • 180 (Ringing) • 183 (Session Progress)	SIP
IPPARM_ BUSY_REASON	Type: eIP_EC_TYPE Size: sizeof(int)	Used to specify the cause code to send when no additional incoming sessions can be accepted. Values: • IPEC_SIPReasonStatus480TemporarilyUnavailable • IPEC_SIPReasonStatus486BusyHere • IPEC_SIPReasonStatus600BusyEverywhere	SIP
IPPARM_RECEIVED_ RESPONSE_ STATUS_CODE	Type: Unsigned short Size: sizeof(int)	Used to retrieve the status code from a received provisional response reported to the application as a GCEV_ALERTING event. Values: • 180 (Ringing) • 181 (Call is Being Forwarded) • 182 (Queued) • 183 (Session Progress)	SIP

9.2.28 IPSET_SIP_SESSION_TIMER

This parameter set is used with the SIP session timer feature. For more information about this feature, see Section 4.31, “SIP Session Timer”, on page 359.

Table 71. IPSET_SIP_SESSION_TIMER Parameter Set

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_SESSION_ EXPIRES	Type: int Size: sizeof(int)	Session-Expires timer value in seconds.	SIP
IPPARM_MIN_SE	Type: int Size: sizeof(int)	Min-SE timer value in seconds. IPPPARM_MIN_SE does not affect Global Call decision to automatically reject incoming call with 422 Session Interval Too Small. Automatically reject decision is based only on SIP_SessionTimer_MinSE from virtual board parameter (IP_VIRTBOARD structure).	SIP
IPPARM_REFRESHER_ PREFERENCE	Type: enumeration Size: sizeof(char)	Refresher preference. Values: • IP_REFRESHER_LOCAL – The User Agent wishes to be the refresher. • IP_REFRESHER_REMOTE – The User Agent wishes the remote side to be the refresher. • IP_REFRESHER_DONT_CARE – The User Agent does not care who is the refresher.	SIP
IPPARM_REFRESH_ METHOD	Type: enumeration Size: sizeof(char)	Method for refresh. Values: • IP_REFRESH_REINVITE – The refresh method is re-INVITE. • IP_REFRESH_UPDATE – The refresh method is UPDATE.	SIP
IPPARM_REFRESH_ WITHOUT_REMOTE_ SUPPORT	Type: enumeration Size: sizeof(char)	When 2xx final response was received, the server does not support the session timer. The application may choose to execute session timer. The session timer mechanism can be operated as long as one of the two User Agents in the call leg supports the extension. If the application decides to operate the session timer, that side sends the refresh. The other side sees the refreshes as repetitive re-INVITEs. The default behavior is to execute the session timer mechanism for the call. Values: • IP_REFRESH_WITHOUT_REMOTE_ SUPPORT_DISABLE – Do not execute session timer without remote support. • IP_REFRESH_WITHOUT_REMOTE_ SUPPORT_ENABLE – Execute session timer without remote support.	SIP

Table 71. IPSET_SIP_SESSION_TIMER Parameter Set (Continued)

Parameter ID	Data Type & Size	Description	SIP/ H.323
IPPARM_REFRESH_ WITHOUT_PREFEREN CE	Type: enumeration Size: sizeof(char)	When 2xx final response was received, the refresher preference did not match the call refresher. The application may choose to execute the session timer. If the application decides to operate the session timer mechanism, the refresher is different from the application preference. The default behavior is to execute the session timer mechanism for the call. Values: • IP_REFRESH_WITHOUT_PREFERENCE_ DISABLE – Do not execute session timer without preference. • IP_REFRESH_WITHOUT_PREFERENCE_ ENABLE – Execute session timer without preference.	SIP
IPPARM_TERMINATE_ CALL_WHEN_EXPIRES	Type: enumeration Size: sizeof(char)	When the session timer is about to expire, Global Call sends GCEV_SIP_SESSION_EXPIRES event to the application. If this parameter is enabled, Global Call sends BYE to terminate the call. If disabled, Global Call does not send BYE and the call stays connected. The default behavior is to terminate the call. Values: • IP_TERMINATE_CALL_WHEN_EXPIRES_ DISABLE – Do not terminate the call when the session timer is about to expire. • IP_TERMINATE_CALL_WHEN_EXPIRES_ ENABLE – Terminate the call when the session timer is about to expire.	SIP

9.2.29 IPSET_SUPPORTED_PREFIXES

Table 72 shows the parameter IDs in the IPSET_SUPPORTED_PREFIXES parameter set.

Table 72. IPSET_SUPPORTED_PREFIXES Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ ADDRESS_DOT_NOTATION	Type: string † Size: max. length = 255	A valid IP address in dot notation	H.323
IPPARM_ ADDRESS_EMAIL	Type: string † Size: max. length = 255	An e-mail address composed of characters from the set “[A-Z][a-z][0-9]_-.@”	H.323
IPPARM_ ADDRESS_H323_ID	Type: string † Size: max. length = 255	A valid H.323 ID	H.323
† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.			

Table 72. IPSET_SUPPORTED_PREFIXES Parameter Set (Continued)

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ ADDRESS_PHONE	Type: string † Size: max. length = 255	An E.164 telephone number. The number string must include the "TEL:" prefix substring.	H.323
IPPARM_ ADDRESS_TRANSPARENT	Type: string † Size: max. length = 255	Unspecified address type	H.323
IPPARM_ADDRESS_URL	Type: string † Size: max. length = 255	A valid URL composed of characters from the set "[A-Z][a-z][0-9]-.". Must contain at least one "." and may not begin or end with a "-".	H.323
† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.			

9.2.30 IPSET_SWITCH_CODEC

Table 73 shows the parameter IDs in the IPSET_SWITCH_CODEC parameter set. This parameter set is used with the Global Call extension API (**gc_Extension( )** function and GCEV_EXTENSION events) with the extension ID IPEXTID_CHANGE_MODE for manual switching between audio and T.38 fax modes.

This parameter set and the IPEXTID_CHANGE_MODE extension ID are not supported for the third party call control (3PCC) operating mode.

Table 73. IPSET_SWITCH_CODEC Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ACCEPT	Type: int Size: sizeof(int)	Used to accept an incoming coder switch request.	H.323, SIP 1PCC
IPPARM_AUDIO_INITIATE	Type: int Size: sizeof(int)	Used to initiate the sending of a RequestMode (H.323) or REINVITE (SIP) message to the remote side to switch from T.38 fax to audio.	H.323, SIP 1PCC
IPPARM_AUDIO_REQUESTED	Type: int Size: sizeof(int)	Provides notification of an incoming request to switch from T.38 fax to audio.	H.323, SIP 1PCC
IPPARM_READY	Type: int Size: sizeof(int)	Provides notification that the media is ready.	H.323, SIP 1PCC
IPPARM_REJECT	Type: int Size: sizeof(int)	Used to reject an incoming request to switch from audio to T.38 fax or vice versa.	H.323, SIP 1PCC

Table 73. IPSET_SWITCH_CODEC Parameter Set (Continued)

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_T38_INITIATE	Type: int Size: sizeof(int)	Used to initiate the sending of a RequestMode (H.323) or REINVITE (SIP) message to the remote side to switch from audio to T.38 fax.	H.323, SIP 1PCC
IPPARM_T38_REQUESTED	Type: int Size: sizeof(int)	Provides notification of an incoming request to switch from audio to T.38 fax.	H.323, SIP 1PCC

9.2.31 IPSET_TRANSACTION

Table 74 shows the parameter IDs in the IPSET_TRANSACTION parameter set.

Table 74. IPSET_TRANSACTION Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_TRANSACTION_ID	Type: int Size: sizeof(int)	Used to uniquely identify any transaction	H.323

9.2.32 IPSET_TUNNELEDSIGNALMSG

Table 75 shows the parameter IDs in the IPSET_TUNNELEDSIGNALMSG parameter set, which is used when sending or receiving tunneled signaling messages (TSMs) in the H.323 protocol.

Table 75. IPSET_TUNNELEDSIGNALMSG Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ TSM_CONTENT_EVENT	Type: enum	Used to identify the type of Global Call event to retrieve TSM content from. Values include: • TSM_CONTENT_OFFERED • TSM_CONTENT_PROCEEDING • TSM_CONTENT_ALERTING • TSM_CONTENT_CONNECTED • TSM_CONTENT_DISCONNECTED • TSM_CONTENT_EXTENSION	H.323
IPPARM_ TUNNELEDSIGNALMSG_ ALTERNATEID	Type: IP_TUNNEL PROTOCOL_ ALTID Size: sizeof(IP_TUNNEL PROTOCOL_ ALTID)	Used to contain a tunneled protocol alternate identifier in a tunneled signaling message (TSM). Either this or the tunneled protocol object ID must exist in a TSM. If the application is using a tunneled protocol object ID when sending a TSM, this parameter should not be inserted in the GC_PARM_BLK.	H.323
IPPARM_ TUNNELEDSIGNALMSG_ CONTENT	Type: string † Size: max length= MAX_IE_ LENGTH (255)	Used to contain any data content of a tunneled signaling message (TSM), which is a sequence of octet strings.	H.323
IPPARM_ TUNNELEDSIGNALMSG_ NSDATA_DATA	Type: string † Size: max. length= max_parm_data_ size ‡ (configured via IPCCLIB_ START_DATA)	Used to contain any non-standard data in a tunneled signaling message (TSM). If no non-standard data is being sent in a TSM, this parameter should not be inserted in the GC_PARM_BLK.	H.323
IPPARM_ TUNNELEDSIGNALMSG_ NSDATA_H221NS	Type: IP_H221 NONSTANDARD Size: sizeof(IP_H221NON STANDARD)	Used to contain an H.221 non-standard data identifier in a tunneled signaling message (TSM). When sending non-standard data in a TSM, ether this ID or the non-standard data object ID must exist in the non-standard data. If non-standard data is not being sent, or if a non-standard data object ID is being used when sending a TSM, this parameter should not be inserted in the GC_PARM_BLK.	H.323

† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.
‡ The full maximum length that is configured may not be usable in practice because the H.323 stack limits total message size to max_parm_data_size + 512 bytes. Longer messages are truncated without notification to the application.

Table 75. IPSET_TUNNELEDSIGNALMSG Parameter Set (Continued)

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_ TUNNELEDSIGNALMSG_ NSDATA_OBJID	Type: string † Size: max length = MAX_NS_ PARAM_OBJID_ LENGTH (40)	Used to contain a non-standard data object identifier in a tunneled signaling message (TSM). When sending non-standard data in a TSM, either this ID or an H.221 non-standard data ID must exist in the non-standard data. If non-standard data is not being sent, or if an H.221 non-standard data ID is being used when sending a TSM, this parameter should not be inserted in the GC_PARM_BLK.	H.323
IPPARM_ TUNNELEDSIGNALMSG_ PROTOCOL_OBJECTID	Type: IP_TUNNEL PROTOCOL_ OBJECTID Size: sizeof(IP_TUNNEL PROTOCOL_ OBJECTID)	Used to contain a tunneled protocol object identifier in a tunneled signaling message (TSM). Either this or the tunneled protocol alternate ID must exist in a TSM. If the application is using an alternate identifier when sending a TSM, this parameter should not be inserted in the GC_PARM_BLK.	H.323
IPPARM_ TUNNELEDSIGNALMSG_ PROTOCOL_OBJID	Type: string † Size: max length = MAX_TSM_ POID_PARAM_ LENGTH (128)	Deprecated parameter previously used to contain a tunneled protocol object identifier in a tunneled signaling message. Superseded by IPPARM_TUNNELEDSIGNALMSG_ PROTOCOL_OBJECTID.	H.323
† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1. ‡ The full maximum length that is configured may not be usable in practice because the H.323 stack limits total message size to max_parm_data_size + 512 bytes. Longer messages are truncated without notification to the application.			

9.2.33 IPSET_VENDORINFO

Table 76 shows the parameter IDs in the IPSET_VENDORINFO parameter set.

Table 76. IPSET_VENDORINFO Parameter Set

Parameter IDs	Data Type & Size	Description	SIP/ H.323
IPPARM_H221NONSTD	Type: IP_H221NONSTANDARD Size: sizeof(IP_ H221NONSTANDARD)	Contains country code, extension code and manufacturer code. See the reference page for IP_H221NONSTANDARD on page 658 for details.	H.323
IPPARM_ VENDOR_PRODUCT_ID	Type: string † Size: max. length = MAX_PRODUCT_ID_LENGTH (32)	Vendor product identifier	H.323
IPPARM_ VENDOR_VERSION_ID	Type: string † Size: max. length = MAX_VERSION_ID_LENGTH (32)	Vendor version identifier	H.323
† For parameters with data of type String, the length in a GC_PARM_BLK is the length of the data string plus 1.			

IP-Specific Data Structures 10

This chapter describes the data structures that are specific to IP technology.

Note: These data structures are defined in the *gcip.h* header file.

GC_PARM_DATA_EXT

```
typedef struct
{
    unsigned long     version;
    void*             pInternal;
    unsigned long     set_ID;
    unsigned long     parm_ID;
    unsigned long     data_size;
    void*             pData;
}GC_PARM_DATA_EXT, *GC_PARM_DATA_EXTP;
```

■ Description

The GC_PARM_DATA_EXT structure contains parameter data retrieved from a GC_PARM_BLK by the **gc_util_find_parm_ex( )** and **gc_util_next_parm_ex( )** functions. These functions were added to the Dialogic® Global Call API library to support the retrieval of parameters whose values may exceed 255 bytes in length. The functions always return the retrieved parameter information in a GC_PARM_DATA_EX structure regardless of whether the parameter value actually exceeds 255 bytes.

The set ID and parm ID as a pair identify the parameter. Set IDs and parm IDs that are common to multiple Global Call technologies are listed in the *Dialogic® Global Call API Library Reference*, and additional technology-specific parameters are listed in each of the various Global Call Technology Guides. Unless a particular set ID/parm IP pair specifically indicates that it supports parameter data that exceeds 255 bytes in length, users should assume that the parameter data length does not exceed 255.

The parameters that currently support extended-length values include:

- IPSET_MIME (or IPSET_MIME_200OK_TO_BYE) / IPPARM_MIME_PART_HEADER
- IPSET_MIME (or IPSET_MIME_200OK_TO_BYE) / IPPARM_MIME_PART_TYPE
- IPSET_NONSTANDARDCONTROL / IPPARM_NONSTANDARDDATA_DATA
- IPSET_NONSTANDARDDATA / IPPARM_NONSTANDARDDATA_DATA
- IPSET_SDP / all four parameter IDs (supported in 3PCC operating mode only)
- IPSET_SIP_MSGINFO / IPPARM_SIP_HDR
- IPSET_TUNNELEDSIGNALMSG / IPPARM_TUNNELEDSIGNALMSG_DATA

Applications **must** use the **INIT_GC_PARM_DATA_EXT( )** function to initialize the structure with the correct version number and default field values before using the structure in a call to **gc_util_find_parm_ex( )** or **gc_util_next_parm_ex( )**. Passing a pointer to an uninitialized structure in the function call may cause an operational error.

■ Field Descriptions

The fields of GC_PARM_DATA_EXT are described as follows:

version
: identifies the version of the data structure implementation. This field is reserved for library use and should **not** be modified by applications.

pInternal
: pointer used to identify the parameter's position within the GC_PARM_BLK structure. This field is reserved for library use and should **not** be used or modified by applications.

set_id
: the set ID of the retrieved parameter

parm_id
: the parameter ID of the retrieved parameter

data_size
: the size of the retrieved parameter data in bytes

pData
: pointer to the first byte of the parameter value buffer

IP_ADDR

```
typedef struct
{
    unsigned char     ip_ver;
    union
    {
        unsigned int      ipv4;
        unsigned int      ipv6[4]
    }u_ipaddr;
}IP_ADDR, *IP_ADDRP;
```

■ Description

The IP_ADDR structure is used to specify a local IP address.

■ Field Descriptions

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

ip_ver
: The version of the local IP address. Possible values are:

- IPVER4
- IPVER6

u_ipaddr
: A union that contains the actual address. The datatype is different depending on whether the address is an IPv4 or an IPv6 address.

For an **IPv4 address**, the address must be stored in memory using the network byte order (big endian) rather than the little-endian byte order of the Dialogic® architecture. A socket API, **htonl**( ), is available to convert from host byte order to network byte order. As an example, to specify an IP address of 127.10.20.30, you may use either of the following C statements:

```
ipv4 = 0x1e140a7f -or-
ipv4 = htonl(0x7f0a141e)
```

For more information on the byte order of IPv4 addresses, see RFC 791 and RFC 792.

A typical **IPv6 address** consists of two parts: the Prefix and the Interface-Identifier. The Prefix identifies a subnet (Length = n bits). The Interface-Identifier identifies an interface on a subnet and must be unique on that subnet (Length = 128 – n bits). For example:

```
Unspecified Address     0:0:0:0:0:0:0:0
Loopback Address        0:0:0:0:0:0:0:1
```

The following IPv6 address format allows the IPv4 address to be represented in the IPv6 domain. IPv4 addresses are encoded into the low-order 32 bits of the IPv6 address, and the high-order 96 bits hold the fixed prefix (that is, IPv4 address in compressed form "::ffff:10.10.20.55").

```
0:0:0:0:0:0:ipaddress (32 bits)    IPv4-compatible IPv6 address
0:0:0:0:0:FFFF:ipaddress (32 bits) IPv4-mapped IPv6 address, for
                                       IPv4-only nodes
```

IP_AUDIO_CAPABILITY

```
typedef struct
{
    unsigned long   frames_per_pkt;
    long            VAD;
} IP_AUDIO_CAPABILITY;
```

■ Description

The IP_AUDIO_CAPABILITY data structure is used to allow some minimum set of information to be exchanged together with the audio codec identifier.

■ Field Descriptions

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

frames_per_pkt

When bundling more than one audio frame into a single transport packet, this value should represent the maximum number of frames per packet that will be sent on the wire. When set to zero, indicates that the exact number of frames per packet is not known, or that the data is not applicable. This field can also be set to GCCAP_dontCare to indicate that any supported value is valid.

Note: For G.711 coders, this field represents the frame size (for example, 10 msec); the frames per packet value is fixed at 1 fpp. For other coders, this field represents the frames per packet and the frame size is fixed. See Section 4.3.2, "Setting Coder Information", on page 118 for more information.

VAD

Identifies whether voice activated detection (VAD) is enabled or disabled. Possible values are:

- GCPV_ENABLE – VAD enabled
- GCPV_DISABLE – VAD disabled
- GCCAP_dontCare – Any supported value is valid

IP_AUTHENTICATION

```
typedef struct
{
    unsigned short  version;
    char*           realm;
    char*           identity;
    char*           username;
    char*           password;
} IP_AUTHENTICATION;
```

■ Description

The IP_AUTHENTICATION data structure is used when setting or removing SIP authentication quadruplets.

Applications should use the **INIT_IP_AUTHENTICATION( )** function to initialize the structure with the correct version number and void pointers for each of the strings before setting the appropriate values.

■ Field Descriptions

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

version
: The version number of the data structure. The correct value is set by the **INIT_IP_AUTHENTICATION( )** initialization function and should not be overridden.

realm
: A null-terminated string that defines the protected domain. This string is case-insensitive and must always be supplied.

identity
: A null-terminated string that allows applications to optionally specify different username/ password pairs for different identities in the same realm. The identity is a URI and must conform to URI syntax, including starting with the scheme (namely “sip:” or “sips:”). If only one username and password applies to a given realm or if setting a default username and password for a multi-identity realm, use an empty string ("") for this field. This field is case-insensitive.

username
: A null-terminated string providing the user’s name in the specified realm. This field is case-sensitive. This field must always contain a non-empty string when the structure is associated with an IPPARM_AUTHENTICATION_CONFIGURE parameter. This field is ignored when the structure is associated with an IPPARM_AUTHENTICATION_REMOVE parameter.

password
: A null-terminated string providing password associated with the user’s name in the specified realm. This field is case-sensitive. This field is ignored when the structure is associated with an IPPARM_AUTHENTICATION_REMOVE parameter.

IP_CAPABILITY

```
typedef struct
{
    int                     capability;
    int                     type;
    int                     direction;
    int                     payload_type;
    IP_CAPABILITY_UNION     extra;
    char                    rfu[0x10];
} IP_CAPABILITY;
```

■ Description

The IP_CAPABILITY data structure provides basic media capability information, including the capability or codec identification and the direction. The IP_CAPABILITY structure is used as the value of one or more parameter element in a GC_PARM_BLK structure when communicating coder capabilities between endpoints.

Note: The IP_CAPABILITY data structure is not intended to provide all the flexibility of the H.245 terminal capability structure or SDP, but provides a first level of useful information in addition to the capability or codec identifier.

■ Field Descriptions

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

capability
The IP Media capability for this structure. Possible values are:
- GCCAP_AUDIO_g711Alaw64k
- GCCAP_AUDIO_g711Ulaw64k
- GCCAP_AUDIO_g7231_5_3k
- GCCAP_AUDIO_g7231_6_3k
- GCCAP_AUDIO_g726_16k
- GCCAP_AUDIO_g726_24k
- GCCAP_AUDIO_g726_32k
- GCCAP_AUDIO_g726_40k
- GCCAP_AUDIO_g729AnnexA
- GCCAP_AUDIO_g729AnnexAwAnnexB
- GCCAP_AUDIO_NO_AUDIO
- GCCAP_DATA_t38UDPFax
- GCCAP_dontCare

type
The category of capability specified in this structure. Indicates which member of the IP_CAPABILITY_UNION union is being used in the extra field. Possible values are:
- GCCAPTYPE_AUDIO – Audio
- GCCAPTYPE_RDATA – Data

Note: Video is not supported in this data structure because video requires the use of the 3PCC operating mode, which does not support this structure.

direction

Identifies the direction and state of the stream that the media attributes in this structure apply to. Possible values are:

- IP_CAP_DIR_LCLRECEIVE – Capabilities specified in the structure refer to receive direction of a full duplex media session.
- IP_CAP_DIR_LCLRECVONLY – Capabilities refer to a half-duplex, receive-only media session.
- IP_CAP_DIR_LCLSENDONLY – Capabilities refer to a half-duplex, send-only media session.
- IP_CAP_DIR_LCLTRANSMIT – Capabilities specified in the structure refer to transmit direction of a full duplex media session.
- IP_CAP_DIR_LCLTXRX – Capabilities specified in the structure refer to both transmit and receive directions of a symmetrical full duplex media session. Supported for T.38 only.
- IP_CAP_DIR_LCLRTPINACTIVE – Capabilities refer to a media session that has been put on hold but with RTCP still active. RTP streaming is temporarily disabled until direction value is changed again. This value is only valid when using SIP, and only when sending or responding to a re-INVITE request.
- IP_CAP_DIR_LCLRTPRTCPINACTIVE – Capabilities refer to a media session that has been put on hold with RTCP as well as RTP inactive. Both RTP and RTCP streaming are disabled until direction value is changed again. This value is only valid when using SIP, and only when sending or responding to a re-INVITE request.
- IP_CAP_DIR_RMTRECEIVE – Coder in a FastStart offer was specified by the remote end to be Receive-only. Only supported when retrieving FastStart coder information from GCEV_OFFERED events.
- IP_CAP_DIR_RMTTRANSMIT – Coder in a FastStart offer was specified by the remote end to be Transmit-only. Only supported when retrieving FastStart coder information from GCEV_OFFERED events.
- IP_CAP_DIR_RMTTXRX – Coder in a FastStart offer was specified by the remote end to be capable of both Transmit and Receive. Only supported when retrieving FastStart coder information from GCEV_OFFERED events.
- IP_CAP_DIR_RMTRTPINACTIVE – Coder in a FastStart SDP offer was specified by the remote end to have a direction attribute of “a=inactive” in the “m=” line, which is used to deactivate RTP streaming. Only supported when retrieving FastStart coder information from GCEV_OFFERED events and only when using SIP.
- IP_CAP_DIR_RMTRTPRTCPINACTIVE – Coder in a FastStart SDP offer was specified by the remote end to have an RTP address of 0.0.0.0 in the “c=” line, which is used to deactivate both RTP and RTCP. Only supported when retrieving FastStart coder information from GCEV_OFFERED events, and only when using SIP.

payload_type

Not currently supported.

extra

The contents of this IP_CAPABILITY_UNION will be indicated by the type field.

rfu

Reserved for future use. Must be set to zero.

IP_CAPABILITY_UNION

```
typedef union
{
    IP_AUDIO_CAPABILITY         audio;
    IP_VIDEO_CAPABILITY         video;
    IP_DATA_CAPABILITY          data;
} IP_CAPABILITY_UNION;
```

■ Description

The IP_CAPABILITY_UNION union enables different capability categories to define their own additional parameters or interest.

■ Field Descriptions

The fields of the IP_CAPABILITY_UNION union are described as follows:

audio
: A structure that represents the audio capability. See IP_AUDIO_CAPABILITY, on page 650 for more information.

video
: Not supported.

data
: Not supported.

IP_CONNECT

```
typedef struct
{
   unsigned short     version;
   int                mediaHandle;
   int                faxHandle;
   eIPConnectType_e   connectType;
} IP_CONNECT;
```

Description

The IP_CONNECT data structure contains information required when associating a Media device with a T.38 Fax device required when switching from an audio coder to a T.38 coder and vice versa.

Field Descriptions

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

version
: reserved for library use; applications should not manipulate this field. The current version number is 0x100.

mediaHandle
: the Media device handle

faxHandle
: the T.38 Fax device handle

connectType
: the connection type. Possible values are:
  - IP_FULLDUP
  - IP_HALFDUP

Note: When disassociating a Media device from a T.38 Fax device, the faxHandle and connectType fields are ignored.

IP_DATA_CAPABILITY

```
typedef struct
{
   int      max_bit_rate;
} IP_DATA_CAPABILITY;
```

Description

The IP_DATA_CAPABILITY data structure provides additional information about the data capability.

Field Descriptions

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

max_bit_rate
Possible values are:

- 2400
- 4800
- 9600
- 14400

The recommended value for T.38 coders is 14400.

IP_DTMF_DIGITS

```
typedef struct
{
    char          digit_buf[IP_MAX_DTMF_DIGITS];
    unsigned int  num_digits;
} IP_DTMF_DIGITS;
```

■ Description

The IP_DTMF_DIGITS data structure is used to provide DTMF information when the digits are received in a User Input Indication (UII) message with alphanumeric data.

■ Field Descriptions

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

digit_buf
: The DTMF digit string buffer; 32 characters in size

num_digits
: The number of DTMF digits in the string buffer

IP_H221NONSTANDARD

```
typedef struct
{
    int   country_code;
    int   extension;
    int   manufacturer_code;
} IP_H221NONSTANDARD;
```

■ Description

The IP_H221NONSTANDARD data structure is used to store H.221 data associated with H.323 nonstandard data.

■ Field Descriptions

The fields of the IP_H221NONSTANDARD data structure are described as follows:

country_code
: The country code. Range: 0 to 255; any value x>255 is treated as x%256.

extension
: The extension number. Range: 0 to 255; any value x>255 is treated as x%256.

manufacturer_code
: The manufacturer code. Range: 0 to 65535; any value x>65535 is treated as x%65636.

IP_PROXY_INFO

```
typedef struct
{
        unsigned long           ulVersion;
        unsigned short          usPort;
        EnumProxyProtocol       eProtocol;
        EnumProxyAddressType    eAddrType;
        char                    arrAddr[CCLIB_SIP_TRANSPORT_IPSTRING_LEN];
        char                    arrHostName[CCLIB_SIP_MAX_HOST_NAME_LEN];
} IP_PROXY_INFO, *pIP_PROXY_INFO;
```

■ Description

The IP_PROXY_INFO structure is used to specify information for dynamic SIP outbound proxy.

■ Field Descriptions

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

ulVersion
: The version of the data structure implementation.

usPort
: The port to which the proxy protocol is assigned.

eProtocol
: The proxy protocol.

eAddrType
: Qualifies the eProtocol field; use ePROXY_ADDRESS_TYPE_IP.

arrAddr
: Used to enter a hard-coded proxy IP address.

arrHostName
: Sets the specified hostname as the SIP outbound proxy instead of arrAddr. If arrAddr is set to 0, this hostname is resolved as the outbound proxy address; otherwise this field is ignored. The default value is NULL.

IP_REGISTER_ADDRESS

```
typedef struct
{
    char              reg_client [IP_REG_CLIENT_ADDR_LENGTH];
    char              reg_server [IP_REG_SERVER_ADDR_LENGTH];
    int               time_to_live;
    int               max_hops;
} IP_REGISTER_ADDRESS;
```

■ Description

The IP_REGISTER_ADDRESS data structure is used to store registration information.

■ Field Descriptions

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

reg_client
: The meaning is protocol dependent:
  - When using H.323, this field is not used; any value specified is ignored
  - When using SIP, this field is an alias for the subscriber

reg_server
: The address of the registration server. Possible value are:
  - An IP address in dot notation. A port number can also be specified as part of the address, for example, 10.242.212.216:1718.
  - IP_REG_MULTICAST_DEFAULT_ADDR

time_to_live
: The time to live value in seconds. The number of seconds for which a registration is considered to be valid when repetitive registration is selected.

  In H.323, the default value of this field is 0, which disables repetitive registration.

  In SIP, if this field is left at its default value 0, the call control library automatically enables auto-refresh with an Expires value of 3600 unless the application explicitly disables auto-refresh. Setting this to a non-zero value sets the Expires header in the REGISTER request to the specified value.

max_hops
: The multicast time to live value in hops. The maximum number of hops (connections between routers) that a packet can take before being discarded or returned when using multicasting.

  This field applies only to H.323 applications using gatekeeper discovery (H.225 RAS) via the default multicast registration address.

IP_SIP_TRANSPORT_ADDR

```
typedef struct
{
        unsigned long                   version;
        unsigned short                  remTAport;
        EnumSipTransport                eremTransportType;
        EnumSipTransportAddressType     eAddrType;
        char                            remTA[CCLIB_SIP_TRANSPORT_IPSTRING_LEN];
} IP_SIP_TRANSPORT_ADDR;
```

■ Description

The IP_SIP_TRANSPORT_ADDR structure contains the transport layer address information of an inbound SIP call.

■ Field Descriptions

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

version
: The version of the data structure implementation. The value is IP_SIP_TA_VERSION.

remTAport
: The transport layer IP port of the remote party.

eremTransportType
: The transport layer protocol type for the inbound SIP call. Valid values are of EnumSipTransport type.

eAddrType
: The transport layer IP address type of the remote party. Valid values are of EnumSipTransportAddressType type.

remTA
: The transport layer IP address (IPv4 or IPv6) of the remote party.

IP_TUNNELPROTOCOL_ALTID

```
typedef struct
{
    unsigned long  version;
    char           protocolType[MAX_TSM_ALTID_VARS_LENGTH];
    int            protocolTypeLength;
    char           protocolVariant[MAX_TSM_ALTID_VARS_LENGTH];
    int            protocolVariantLength;
    char           subIdentifier[MAX_TSM_ALTID_VARS_LENGTH];
    int            subIdentifierLength;
} IP_TUNNELPROTOCOL_ALTID;
```

■ Description

The IP_TUNNELPROTOCOL_ALTID data structure is used in H.323 Annex M tunneled signaling to identify the protocol using alternate ID information. This data structure is used as the value of a Global Call parameter element of type IPSET_TUNNELEDSIGNALMSG / IPPARM_TUNNELEDSIGNALMSG_ALTERNATEID. This data structure is not used when the tunneled signaling message uses a protocol object ID to identify the protocol.

Applications should use the **INIT_IP_TUNNELPROTOCOL_ALTID( )** function to initialize the structure with the correct version number and initial field values.

■ Field Descriptions

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

version
: the version number of the data structure; the correct value is set by the **INIT_IP_TUNNELPROTOCOL_ALTID( )** initialization function and should not be overridden by the application

protocolType
: a string that identifies the tunneled protocol type
maximum length: MAX_TSM_ALTID_VARS_LENGTH

protocolTypeLength
: the length of the protocolType string

protocolVariant
: a string that identifies the tunneled protocol variant
maximum length: MAX_TSM_ALTID_VARS_LENGTH

protocolVariantLength
: the length of the protocolVariant string

subIdentifier
: a string that provides additional tunneled protocol identification
maximum length: MAX_TSM_ALTID_VARS_LENGTH

subIdentifierLength
: the length of the subIdentifier string

IP_TUNNELPROTOCOL_OBJECTID

```
typedef struct
{
    unsigned long  version;
    char           TunneledProtocol_Oid[MAX_TSM_OBJID_VARS_LENGTH];
    int            TunneledProtocol_OidLength;
    char           subIdentifier[MAX_TSM_OBJID_VARS_LENGTH];
    int            subIdentifierLength;
} IP_TUNNELPROTOCOL_OBJECTID;
```

■ Description

The IP_TUNNELPROTOCOL_OBJECTID data structure is used in H.323 Annex M tunneled signaling to identify the tunneling protocol using a protocol object ID. This data structure is used as the value of a Global Call parameter element of type IPSET_TUNNELEDSIGNALMSG / IPPARM_TUNNELEDSIGNALMSG_PROTOCOL_OBJECTID. This data structure is not used when the tunneled signaling message uses the alternate ID method to identify the protocol.

Applications should use the **INIT_IP_TUNNELPROTOCOL_OBJECTID( )** function to initialize the structure with the correct version number and initial field values.

■ Field Descriptions

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

version
: the version number of the data structure; the correct value is set by the **INIT_IP_TUNNELPROTOCOL_OBJECTID( )** initialization function and should not be overridden by applications

TunneledProtocol_Oid
: a string that identifies the tunneled protocol object
: maximum length: MAX_TSM_OBJECTID_VARS_LENGTH

TunneledProtocol_OidLength
: the actual length of the TunneledProtocol_Oid string

subIdentifier
: a string that provides additional tunneled protocol identification
: maximum length: MAX_TSM_OBJECTID_VARS_LENGTH

subIdentifierLength
: the actual length of the subIdentifier string

IP_VIRTBOARD

```
typedef struct
{
    unsigned short                          version;
    unsigned int                            total_max_calls;
    unsigned int                            h323_max_calls;
    unsigned int                            sip_max_calls;
    IP_ADDR                                 localIP;
    unsigned short                          h323_signaling_port;
    unsigned short                          sip_signaling_port;
    void                                    *reserved;
    unsigned short                          size;
    unsigned int                            sip_msginfo_mask;
    unsigned int                            sup_serv_mask;
    unsigned int                            h323_msginfo_mask;
    MIME_MEM                                sip_mime_mem;
    unsigned short                          terminal_type;
    IP_ADDR                                 outbound_proxy_IP;
    unsigned short                          outbound_proxy_port;
    char *                                  outbound_proxy_hostname;
    EnumSIP_Enabled                         E_SIP_tcpenabled;
    EnumSIP_TransportProtocol               E_SIP_OutboundProxyTransport;
    EnumSIP_Persistence                     E_SIP_Persistence;
    unsigned short                          SIP_maxUDPmsgLen;
    EnumSIP_TransportProtocol               E_SIP_DefaultTransport;
    EnumSIP_RequestRetry                    E_SIP_RequestRetry;
    EnumSIP_Enabled                         E_SIP_OPTIONS_Access;
    int                                     sip_registrar_registrations;
    SIP_TLS_ENGINE                          *sip_tls_engine;
    EnumSIP_Enabled                         E_SIP_PrackEnabled;
    EnumSIP_Enabled                         E_SIP_SessionTimer_Enabled;
    unsigned int                            SIP_SessionTimer_SessionExpires;
    unsigned int                            SIP_SessionTimer_MinSE;
    SIP_STACK_CFG                           *sip_stack_cfg;
    EnumSIP_Enabled                         E_SIP_UPDATE_Access;
    EnumSIP_Enabled                         E_SIP_dynamic_outbound_proxy_enabled;
    EnumSIP_Enabled                         E_SIP_IPv6;
    IP_ADDR                                 localIPv6;
    char*                                   localIPv6_iface_name;

}IP_VIRTBOARD;
```

■ Description

The IP_VIRTBOARD data structure is used to store configuration and capability information about an IPT board device that is used when the device is started. An array of IP_VIRTBOARD structures (one for each virtual board in the system) is referenced by the IPCCLIB_START_DATA structure, which is passed to the **gc_Start( )** function. The IP_VIRTBOARD structure must be initialized to default values by the **INIT_IP_VIRTBOARD( )** initialization function; those default values can be overridden by the application before calling **gc_Start( )**.

■ Field Descriptions

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

version
: The version of the structure. The correct version number is populated by the **INIT_IP_VIRTBOARD( )** function and should not be overriden by the application.

total_max_calls
: The maximum total number of IPT devices that can be open concurrently using either the H.323 or SIP protocol. Valid values range from 1 to IP_CFG_MAX_AVAILABLE_CALLS (=2016). The default value is 120. This field must not be set to IP_CFG_NO_CALLS (=0) and must not be set to a value larger than the sum of h323_max_calls and sip_max_calls. When the library is being started in 3PCC operating mode, this field must be set to a value that is no greater than the number of licensed channels.

h323_max_calls
: The maximum number of IPT devices that can be used for H.323 calls. Valid values are in the range from IP_CFG_NO_CALLS (=0) to IP_CFG_MAX_AVAILABLE_CALLS (=2016). The default value is 120. This field must not be set to IP_CFG_NO_CALLS if sip_max_calls is also set to that value. When the library is being started in 3PCC operating mode, this field should be set to IP_CFG_NO_CALLS (=0).

sip_max_calls
: The maximum number of IPT devices that can be used for SIP calls. Possible values are in the range IP_CFG_NO_CALLS (=0) to IP_CFG_MAX_AVAILABLE_CALLS (=2016). The default value is 120. This field must not be set to IP_CFG_NO_CALLS if h323_max_calls is also set to that value. When the library is being started in 3PCC operating mode, this field must be set to a value that is no greater than the number of licensed channels.

localIP
: The local IP address of type IP_ADDR. See the reference page for IP_ADDR, on page 648.

h323_signaling_port
: The H.323 call signaling port. Possible values are a valid port number or IP_CFG_DEFAULT. The default H.323 signaling port is 1720.

sip_signaling_port
: The SIP call signaling port. Possible values are a valid port number or IP_CFG_DEFAULT. The default SIP signaling port is 5060.

reserved
: For library use only

size
: For library use only

sip_msginfo_mask (structure version ≥ 0x101 only)
: Enables and disables access to SIP message information. Access is disabled by default. The following mask values, which may be OR'ed together, are defined to enable these features:
  - IP_SIP_FASTSTART_CODERS_IN_OFFERED – enable receiving coder information from a SIP "FastStart" call offer via the GCEV_OFFERED event
  - IP_SIP_MIME_ENABLE – enable sending and receiving of SIP messages that contain MIME information
  - IP_SIP_MSGINFO_ENABLE – enable access to supported SIP message information fields
  - IP_SIP_TRANSADDR_ENABLE – enable receipt of SIP transport layer address for incoming INVITE or re-INVITE

sup_serv_mask (structure version ≥ 0x102 only)
: Enables and disables the call transfer supplementary service. The service is disabled by default. Use the following value to enable the feature:
  - IP_SUP_SERV_CALL_XFER – enable call transfer service

h323_msginfo_mask (structure version ≥ 0x103 only)
: Enables and disables reception of H.323 message information. Access is disabled by default. The following mask values, which may be OR'ed together, are defined to enable the features:
  - IP_H323_ANNEXMMSG_ENABLE – Enable reception of H.323 Annex M tunneled signaling messages in H.225 messages
  - IP_H323_FASTSTART_CODERS_IN_OFFERED – enable receiving coder information from an H.323 fastStart call offer via the GCEV_OFFERED event
  - IP_H323_MSGINFO_ENABLE – enable access to H.323 message information fields
  - IP_H323_RETRIEVE_UUIE_ENABLE – Enable receiving User-to-User Information Elements from incoming H.323 messages

sip_mime_mem (structure version ≥ 0x104 only)
: Sets the number and size of buffers that will be allocated for the MIME memory pool when the SIP MIME feature is enabled (no buffers are allocated if the feature is not enabled). The default values indicated below are set by the **INIT_MIME_MEM( )** macro, which is called by the **INIT_IP_VIRTBOARD( )** initialization function. The MIME_MEM data structure is defined as follows:

```
typedef struct
{
     unsigned short   version;     /* Version set by INIT_MIME_MEM   */
     unsigned int     size;        /* Default = 1500                 */
     unsigned int     number;      /* Default = (sip_max_calls * 5)  */
}MIME_MEM;
```

terminal_type (structure version ≥ 0x104 only)
: Sets the Terminal Type for the virtual board which will be used during RAS registration (H.323 terminal type) and during Master Slave determination (H.245 terminal type). The value may only be changed from the default that is set by the **INIT_IP_VIRTBOARD( )** initialization function before calling **gc_Start( )**. Unsigned shorts from 0 to 255 are valid values, but the specific values 0 and 255 are reserved and will result in the terminal type being set to the default. Values larger than 255 are truncated to 8 bits. The following symbolic values are defined:
  - IP_TT_GATEWAY (Default) – Value = 60, for operation as terminal type Gateway
  - IP_TT_TERMINAL – value = 50, for operation as terminal type Terminal

outbound_proxy_IP (structure version ≥ 0x105 only)
: Sets the IP address of the SIP outbound proxy, which is used instead of the original Request URI for outbound SIP requests. The default value is 0, which disables outbound proxy unless the outbound_proxy_hostname field is set to a non-NULL name.

outbound_proxy_port (structure version ≥ 0x105 only)
: Sets the port number of the SIP outbound proxy specified by outbound_proxy_IP. The default value is 5060, which is the same as the default SIP signaling port number.

outbound_proxy_hostname (structure version ≥ 0x105 only)
: Sets the specified hostname as the SIP outbound proxy instead of a hard-coded IP address. If outbound_proxy_IP is set to 0, this hostname is resolved as the outbound proxy address. If outbound_proxy_IP is set to an IP address, this field is ignored and outbound_proxy_IP and outbound_proxy_port are used instead. The default value is NULL.

E_SIP_tcpenabled (structure version ≥ 0x106 only)
: Enables the handling of incoming SIP messages that use TCP (received on the port number specified in sip_signaling_port), and the ability to specify TCP transport for SIP requests. The following symbolic values are defined:

- ENUM_Disabled (default) – disable TCP transport support (use default UDP transport)
- ENUM_Enabled – enable TCP transport support for incoming and outgoing messages

E_SIP_OutboundProxyTransport (structure version ≥ 0x106 only)
Selects the default transport protocol for SIP requests when an outbound proxy has been set up via the outbound_proxy_IP or outbound_proxy_hostname field (assuming that TCP is enabled via E_SIP_tcpenabled). The following symbolic values are defined:

- ENUM_TCP – use TCP protocol for the outbound proxy; if this value is set when TCP is not enabled or when TCP is enabled but no SIP proxy is configured, **gc_Start( )** returns an IPERR_BAD_PARM error
- ENUM_TLS – use TLS for the outbound proxy; if this value is set when either TCP or TLS is not enabled (TLS operates on top of TCP), **gc_Start( )** returns an IPERR_BAD_PARM error
- ENUM_UDP (default) – use UDP protocol for the outbound proxy

E_SIP_Persistence (structure version ≥ 0x106 only)
Sets the persistence of TCP connections (assuming that TCP has been enabled via E_SIP_tcpenabled). This field has no effect on whether TCP is used for requests; it only affects the connections that are made when TCP is actually used. The following symbolic values are defined:

- ENUM_PERSISTENCE_NONE – no persistence; TCP connection is closed after each request
- ENUM_PERSISTENCE_TRANSACT – transaction persistence; TCP connection is closed after each transaction
- ENUM_PERSISTENCE_TRANSACT_USER (default) – user persistence; TCP connection is maintained for the lifetime of the "user" of the transaction (the CallLeg, for example)

SIP_maxUDPmsgLen (structure version ≥ 0x106 only)
Sets the maximum size for UDP SIP requests; above this threshold, the TCP transport protocol is automatically used instead of UDP (assuming that TCP is enabled via E_SIP_tcpenabled). The default value is 1300 (as recommended by RFC3261). Value may be set to 0 or VIRTBOARD_SIP_NOUDPMSGSIZECHECK to disable the size checking and reduce the message processing overhead.

E_SIP_DefaultTransport (structure version ≥ 0x106 only)
Sets the default transport protocol that is used when there is no proxy set (assuming that TCP is enabled by E_SIP_tcpenabled). The application can override the default for a particular request by explicitly specifying the transport protocol with a "transport= " header parameter. The following symbolic values are defined:

- ENUM_TCP – use TCP unless ";transport=udp" is set by application; if this value is set when TCP is not enabled, **gc_Start( )** returns an IPERR_BAD_PARM error
- ENUM_UDP (default) – use UDP unless ";transport=tcp" is set by application

E_SIP_RequestRetry (structure version ≥ 0x107 only)
Sets the behavior that the SIP stack follows when a particular address-transport combination has failed for a SIP request; this may be a UDP failure after multiple retries or a TCP failure. The following symbolic values are defined:

- ENUM_REQUEST_RETRY_ALL (default) – there will be a retry if the DNS server has provided a list of IP addresses with transports, and there will also be a retry on the last (or only) address if the transport was TCP and the failure reason qualifies for retry
- ENUM_REQUEST_RETRY_DNS – there will be a retry if the DNS server has provided a list of IP addresses with transports

- ENUM_REQUEST_RETRY_FORCEDTCP – there will be a retry if the DNS server has provided a list of IP addresses with transports, and there will also be a retry on the last (or only) address if the transport was forced to be TCP because of message length and the failure reason qualifies for retry
- ENUM_REQUEST_RETRY_NONE – there will be no retry on request failure

E_SIP_OPTIONS_Access (structure version ≥ 0x108 only)
: Enables application access to incoming OPTIONS, and the ability to send OPTIONS requests. The following symbolic values are defined:
  - ENUM_Disabled (default) – disable application access to OPTIONS messages
  - ENUM_Enabled – enable application access to OPTIONS messages

sip_registrar_registrations (structure version ≥ 0x109 only)
: Specifies the number of unique SIP registrations that can be created. A unique registration is defined as a unique Address Of Record/Registrar pair, so registering the same AOR on a different Registrar is counted as a second unique registration. The range for this field is 1 to 10000. The default value is sip_max_calls.

sip_tls_engine (structure version ≥ 0x10a only)
: Identifies a SIP_TLS_ENGINE data structure that specifies a number of parameters needed for SIP Transport Layer Security (TLS), including TLS port number, engine certificate, and trusted root certificate authorities. The default value is a NULL pointer (SIP TLS disabled). If the structure is not properly configured for TLS server or TLS client operation (or both), **gc_Start( )** will fail with error IPERR_INVALID_TLS_PARAM. If this field identifies a properly configured data structure but TCP is not enabled via E_SIP_tcpenabled, **gc_Start( )** will fail with error IPERR_INVALID_TLS_WITHOUT_TCP.

E_SIP_PrackEnabled
: Enables SIP PRACK method. The default is disabled.

E_SIP_SessionTimer_Enabled
: Enables session timer. The default is disabled.

SIP_SessionTimer_SessionExpires
: Sets the session expires value. Used in timer negotiation for outgoing and incoming calls. Default timer value is 1800 seconds for all calls enabled.

SIP_SessionTimer_MinSE
: Sets the minimum session timer value. Used in timer negotiation for outgoing and incoming calls. Default timer value is 90 seconds for all calls enabled. A 422 response code indicates that the session timer duration is too small. That response contains a Min-SE header field identifying the minimum session interval it is willing to support.

sip_stack_cfg
: Identifies a SIP_STACK_CFG data structure that is used to configure SIP stack parameters.

E_SIP_UPDATE_Access
: Enables the application to send and receive SIP UPDATE requests and responses to modify the state of a pending or pre-established session. The default is disabled.

E_SIP_IPv6
: Enables the application to use IPv6. The default is disabled.

localIPv6
: Specifies the local IPv6 address to be used in SIP signaling. See the Example section.

localIPv6_iface_name
For Link-Local IPv6 address only. Specifies the network interface to use when sending IPv6 packets. The value can be an interface name or a scope identifier string value. See the Example section.

■ Example

The following shows an example for localIPv6 field.

```
// Example for Global Unicast IPv6 Address

  ipVirtualBoard[0].E_SIP_IPv6 = TRUE;
  ipVirtualBoard[0].localIPv6.ip_ver = IPVER6;
  inet_pton(AF_INET6, "2001:db8:0:f101::2",
               &ipVirtualBoard[0].localIPv6.u_ipaddr.ipv6[0]);
```

The following shows an example for localIPv6_iface_name field.

```
// Example for Link-Local IPv6 Address

 ipVirtualBoard[0].E_SIP_IPv6 = TRUE;
 ipVirtualBoard[0].localIPv6.ip_ver = IPVER6;
 inet_pton(AF_INET6, "fe80::21e:4fff:fe9a:4a59",
               &ipVirtualBoard[0].localIPv6.u_ipaddr.ipv6[0]);

#ifdef INTERFACE_NAME
   ipVirtualBoard[0].localIPv6_iface_name = "eth0";
else
   #ifdef SCOPEID
      ipVirtualBoard[0].localIPv6_iface_name = "2";
   else
      // Error iface_name must be specified for Link-Local Address
   #endif
 #endif
```

IPCCLIB_START_DATA

```
typedef struct
{
    unsigned short    version;
    unsigned char     delimiter;
    unsigned char     num_boards;
    IP_VIRTBOARD      *board_list;
    unsigned long     max_parm_data_size;
    unsigned short    media_operational_mode;
} IPCCLIB_START_DATA;
```

■ Description

The IPCCLIB_START_DATA structure is used to configure the IP call control library when starting Dialogic® Global Call API. The IPCCLIB_START_DATA structure is passed to the **gc_Start( )** function via the CCLIB_START_STRUCT and GC_START_STRUCT data structures. Applications **must** use the **INIT_IPCCLIB_START_DATA( )** function to populate a IPCCLIB_START_DATA structure with default values before overriding the default values as desired.

■ Field Descriptions

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

version
: The version of the start structure. The correct version number is populated by the **INIT_IPCCLIB_START_DATA( )** function and should not be used by applications.

delimiter
: An ANSI character that specifies the address string delimiter; the default delimiter is the comma (,). The specified delimiter character is used to separate the components of the destination information when using **gc_MakeCall( )**, for example.

num_boards
: The number of IPT virtual board devices to create. See Section 2.3.2, “IPT Board Devices”, on page 39 for more information on IPT board devices. The maximum value is 8, and the default value is 2.

board_list
: A pointer to an array of IP_VIRTBOARD structures, one structure for each of num_boards IPT board devices. See IP_VIRTBOARD, on page 665 for more information.

max_parm_data_size (structure version ≥ 0x200)
: The maximum data size (in bytes) for Global Call parameters that support values longer than 255 bytes. The default value for this field is 255 for backwards compatibility; the maximum value is 4096.

  Only specific Global Call parameters support >255 byte values. These parameters include:
  - IPSET_MIME or IPSET_MIME_200OK_TO_BYE / IPPARM_MIME_PART_HEADER
  - IPSET_MIME or IPSET_MIME_200OK_TO_BYE / IPPARM_MIME_PART_TYPE
  - IPSET_NONSTANDARDCONTROL / IPPARM_NONSTANDARDDATA_DATA
  - IPSET_NONSTANDARDDATA / IPPARM_NONSTANDARDDATA_DATA
  - IPSET_SDP / all four parameter IDs (supported in 3PCC operating mode only)
  - IPSET_SIP_MSGINFO / IPPARM_SIP_HDR

- IPSET_TUNNELEDSIGNALMSG / IPPARM_TUNNELEDSIGNALMSG_DATA

Note: When using H.323, the stack limits the total size of messages to the value of this field + 512 bytes. Because of the presence of other payload in the message, it may not be possible to use the maximum parameter data size defined in this field for H.323 Nonstandard Data or Annex M Tunneled Signaling Message data. If the total size of an H.323 message is greater than max_parm_data_size + 512 bytes, the stack truncates the message with no notification to the application.

media_operational_mode (structure version ≥ 0x201)

The library's media operational mode. The following symbolic values are defined:

- MEDIA_OPERATIONAL_MODE_1PCC – initializes the library in first party call control mode (1PCC). This is the default value.
- MEDIA_OPERATIONAL_MODE_3PCC – initializes the library in third party call control mode (3PCC).

REQUEST_ERROR

```
typedef struct
{
    unsigned short      version;
    unsigned int        error;
    char                method[IP_SIP_METHODSIZE]
}REQUEST_ERROR, *REQUEST_ERRORP;
```

■ Description

The REQUEST_ERROR structure is used to contain information about the conditions that exist when the transmission of a SIP request fails.

■ Field Descriptions

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

version
: identifies the version of the data structure implementation. This field is reserved for library use and should **not** be modified by applications.

error
: an enumeration that identifies the error condition that caused the transmission of the SIP request to fail. Possible values include:
  - IP_SIP_REQUEST_503_RCVD – connection failed due to 503 Service Unavailable or other fatal error cause
  - IP_SIP_REQUEST_FAILED – connection failed due to general or unclassified error
  - IP_SIP_REQUEST_NETWORK_ERROR – connection failed due to network error or local failure
  - IP_SIP_REQUEST_RETRY_FAILED – failure in request retry logic; retry not attempted
  - IP_SIP_REQUEST_TIMEOUT – connection failed due to connection timeout

method
: an array that contains all or part of the failed method's name

RTP_ADDR

```
typedef struct
{
     int              version
     unsigned short   port;
     unsigned char    ip_ver;
     union
     {
          unsigned int     ipv4;
          unsigned int     ipv6[4];
     } u_ipaddr;
} RTP_ADDR, *RTP_ADDRP;
```

■ Description

The RTP_ADDR data structure contains a complete RTP address, which includes both the port number and the IP address. The RTP_ADDR structure is used when retrieving the local and remote RTP addresses from the Global Call completion event when a call is connected.

■ Field Descriptions

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

version
: data structure version identification, for library use only

port
: the port number used by an RTP stream

ip_ver
: format of the IP address; currently, the only valid value is IPVER4

ipv4
: the IP address used by an RTP stream, in IPv4 format

ipv6[4]
: reserved for future use

SIP_STACK_CFG

```
typedef struct {
    unsigned long version;  /* version set by INIT_SIP_STACK_CFG */
    int retransmissionT1;
    int retransmissionT2;
    int retransmissionT4;
    int generalLingerTimer;
    int inviteLingerTimer;
    int provisionalTimer;
    int cancelGeneralNoResponseTimer;
    int cancelInviteNoResponseTimer;
    int generalRequestTimeoutTimer;
} SIP_STACK_CFG;
```

■ Description

The SIP_STACK_CFG data structure is used to configure selected SIP stack parameters such as timers.

The SIP_STACK_CFG data structure is referenced by the IP_VIRTBOARD data structure, which stores configuration and capability information about an IPT (virtual) board device that is populated when the device is started. An array of IP_VIRTBOARD structures (one per virtual board in the system) is referenced by the IPCCLIB_START_DATA structure, which is passed to the **gc_Start( )** function.

Applications should use the **INIT_SIP_STACK_CFG( )** function to initialize the structure with the correct version number and initial field values before setting the appropriate values.

■ Field Descriptions

The fields of the SIP_STACK_CFG data structure are:

version
: The version number of the data structure. The correct value is set by the **INIT_SIP_STACK_CFG( )** initialization function and should not be overridden.

retransmissionT1
: Determines several timers as defined in RFC 3261. For example, when an unreliable transport protocol is used, a Client Invite transaction retransmits requests at an interval that starts at T1 milliseconds and doubles after every retransmission. A Client General transaction retransmits requests at an interval that starts at T1 and doubles until it reaches T2. The default value is 1000.

retransmissionT2
: Determines the maximum retransmission interval as defined in RFC 3261. For example, when an unreliable transport protocol is used, general requests are retransmitted at an interval that starts at T1 and doubles until it reaches T2. If a provisional response is received, retransmissions continue but at an interval of T2. The parameter value cannot be less than 4000. The default value is 8000.

retransmissionT4
: Determines the amount of time the network takes to clear messages between client and server transactions as defined in RFC 3261. For example, when working with an unreliable transport

protocol, T4 determines the time that a UAS waits after receiving an ACK message and before terminating the transaction. The default value is 10000.

generalLingerTimer
: After a server sends a final response, the server cannot be sure that the client has received the response message. The server should be able to retransmit the response upon receiving retransmissions of the request for generalLingerTimer milliseconds. The default value is 32000.

inviteLingerTimer
: After sending an ACK for an INVITE final response, a client cannot be sure that the server has received the ACK message. The client should be able to retransmit the ACK upon receiving retransmissions of the final response for inviteLingerTimer milliseconds. The default value is 32000.

provisionalTimer
: The provisionalTimer is set when receiving a provisional response on an Invite transaction. The transaction will stop retransmissions of the Invite request and will wait for a final response until the provisionalTimer expires. If you set the provisionalTimer to 0, no timer is set, and the Invite transaction will wait indefinitely for the final response. The default value is 180000.

cancelGeneralNoResponseTimer
: When sending a CANCEL request on a General transaction, the User Agent waits cancelGeneralNoResponseTimer milliseconds before timeout termination if there is no response for the canceled transaction. The default value is 32000.

cancelInviteNoResponseTimer
: When sending a CANCEL request on an Invite request, the User Agent waits cancelInviteNoResponseTimer milliseconds before timeout termination if there is no response for the canceled transaction. The default value is 32000.

generalRequestTimeoutTimer
: After sending a General request, the User Agent waits for a final response generalRequestTimeoutTimer milliseconds before timeout termination (in this time the User Agent retransmits the request every T1, 2*T1, ... , T2, ... milliseconds). The default value is 32000.

SIP_TLS_ENGINE

```
typedef struct
{
   unsigned long       version;                     /* system use only */
   unsigned short      sip_tls_port;
   EnumSIP_TLS_METHOD  E_sip_tls_method;
   char *              local_rsa_private_key_filename;
   char *              local_rsa_private_key_password;
   char *              local_rsa_cert_filename;
   char *              local_dss_private_key_filename;
   char *              local_dss_private_key_password;
   char *              local_dss_cert_filename;
   unsigned int        ca_cert_number;
   char **             ca_cert_filename;
   unsigned int        chain_cert_number;
   char **             chain_cert_filename;
   unsigned int        crl_number;
   char **             crl_filename;
   char *              local_cipher_suite;
   char *              dh_param_512_filename;
   char *              dh_param_1024_filename;
   char *              session_id;
   EnumSIP_Enabled     E_client_cert_required;
   EnumSIP_Enabled     E_block_udp_port;
   EnumSIP_Enabled     E_block_tcp_port;
} SIP_TLS_ENGINE;
```

■ Description

The SIP_TLS_ENGINE data structure is used to specify a set of parameters that are used for SIP Transport Layer Security (TLS). To enable the SIP TLS feature, an application configures an IP_VIRTBOARD structure to reference a SIP_TLS_ENGINE structure before calling **gc_Start( )**. If TLS is not required, the sip_tls_engine field in IP_VIRTBOARD should be NULL.

Applications should use the **INIT_SIP_TLS_ENGINE( )** function to initialize a SIP_TLS_ENGINE structure with the correct version number and initial field values. The application must then configure the structure to specify the certificates required for TLS server and/or TLS client operation before calling **gc_Start( )**. Failure to configure the minimum certificate requirements for either server or client operation will prevent the Dialogic® Global Call API library from starting; the **gc_Start( )** call will fail with error IPERR_INVALID_TLS_PARAM.

■ Field Descriptions

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

version
: the version number of the data structure; the correct value is set by the **INIT_SIP_TLS_ENGINE( )** initialization function and should not be overridden by applications

sip_tls_port
: port number of TLS port Global Call will listen to. Default port number is 5061.

sip_tls_method
: indicates the version of SSL to use. Defined enumerations are:

- ENUM_TLS_METHOD_TLS_V1 – use TLS ver. 1 (Default value)

local_rsa_private_key_filename
: name of file containing TLS RSA private key of local certificate. File must be PEM (base64 encoded) X509 format, in plain text or encrypted. Default is NULL.

local_rsa_private_key_password
: password string used to read TLS RSA private key of local certificate if it is encrypted. Default is NULL

local_rsa_cert_filename
: name of file containing TLS RSA certificate representing local identity. File must be PEM (base64 encoded) X509 format, in plain text. Default is NULL.

local_dss_private_key_filename
: name of file containing TLS DSS private key of local certificate. File must be PEM (base64 encoded) X509 format, in plain text or encrypted. Default is NULL.

local_dss_private_key_password
: password string used to read TLS DSS private key of local certificate if it is encrypted. Default is NULL

local_dss_cert_filename
: name of file containing TLS DSS certificate representing local identity. File must be PEM (base64 encoded) X509 format, in plain text. Default is NULL.

ca_cert_number
: number of trusted certificates, which are usually root certificates. TLS engine can trust zero, one, or more root certificates. Once an engine trusts a root certificate, it will approve all valid certificates issued by that root certificate. Use this field to specify the number of trusted certificates in the ca_cert_filename array. This field must be configured when operating as a TLS client. Default value is 0.

ca_cert_filename
: array of filenames for trusted certificates. Files must be PEM (base64 encoded) X509 format, in plain text. The size of the array is specified by ca_cert_number. This array must be configured when operating as a TLS client. Default is NULL.

chain_cert_number
: number of chained certificates. An engine may hold a certificate that is not issued directly by a root certificate, but rather by a certificate authority delegated by that root certificate. To add one or more intermediate certificate to the chain of certificates that the engine will present during a handshake, use this field to specify the number of chained certificates in the chain_cert_filename array. This field is optional, but is commonly needed when operating as a TLS server and may also be needed for a TLS client if mutual authentication is being used. Default value is 0.

chain_cert_filename
: array of filenames for chained certificates. Files must be PEM (base64 encoded) X509 format, in plain text. The size of the array is specified by chain_cert_number. This array is optional, but is commonly needed when operating as a TLS server and may also be needed for a TLS client if mutual authentication is being used. Default is NULL.

crl_number
: number of optional certificate revocation list (CRL) files. An engine may look up CRLs while examining the incoming certificates. To add one or more CRL files, use this field to specify the number of files in the crl_filename array. Default value is 0.

crl_filename
: array of filenames for optional certificate revocation lists (CRLs). Files must be PEM format in plain text. The size of the array is specified by crl_number. Default is NULL.

local_cipher_suite
: optionally specifies a list of ciphers to use when negotiating encryption algorithms with the remote UA. The ciphers are specified in a specially formatted string defined by OPENSSL; OPENSSL allows for several keywords in the elist, which are shortcuts for sets of ciphers. Default is NULL, which uses OPENSSL default string.

dh_param_512_filename
: name of file containing optional DH parameter with 512-bit key length. Default is NULL, in which case Global Call uses pre-built DH parameter with 512-bit key length.

dh_param_1024_filename
: name of containing optional DH parameter with 1024-bit key length. Default is NULL, in which case Global Call uses pre-built DH parameter with 1024-bit key length.

session_id
: optionally specifies a session ID to enable session caching on the server side. When configured, the session ID is provided to the client during handshake so that client may reuse the session for future connection. Default is NULL (server session caching disabled).

E_client_cert_required
: specifies whether the Dialogic® Global Call API library will require the client's certificate for mutual authentication when acting as a TLS server. Defined values are:
  - ENUM_Disabled (Default value) – Do not require client's certificate during TLS handshake (mutual authentication disabled)
  - ENUM_Enabled – Require client's certificate during TLS handshake (mutual authentication enabled)

E_block_udp_port
: specifies whether the UDP port is disabled (both send and receive directions) to block insecure communications and prevent downgrade attack. Defined values are:
  - ENUM_Disabled (Default value) – Do not block UDP port
  - ENUM_Enabled – Block UDP port to prevent insecure communications

E_block_tcp_port
: specifies wether the TCP port will be disabled (in both send and receive directions) to block insecure communications and prevent downgrade attack. Defined values are:
  - ENUM_Disabled (Default value) – Do not block TCP port
  - ENUM_Enabled – Block TCP port to prevent insecure communications

IP-Specific Event Cause Codes 11

This chapter lists the IP-specific error and event cause codes and provides a description of each code. The codes described in this chapter are defined in the *gcip_defs.h* header file.

When a GCEV_DISCONNECTED event is received, use the **gc_ResultInfo( )** function to retrieve the reason or cause of that event.

When using **gc_DropCall( )** with H.323, only event cause codes prefixed by IPEC_H2250 or IPEC_Q931 should be specified in the **cause** parameter.

When using **gc_DropCall( )** with SIP, if the application wants to reject a call during call establishment, the relevant cause value for the **gc_DropCall( )** function can be either one of the generic Dialogic® Global Call API cause values for dropping a call (see the **gc_DropCall( )** function description in the *Dialogic® Global Call API Library Reference*), or one of the cause codes prefixed by IPEC_SIP in this chapter. If the application wants to drop a call that is already connected (simply hanging up normally) the same rules apply, but the cause is not relevant in the BYE message.

When using **gc_Extension( )** to reject an incoming request to switch from audio to T.38 fax or vice versa, use only the cause codes prefixed by "IPEC_Q931Cause" for H.323, or the cause codes prefixed by "IPEC_SIPReason" for SIP.

11.1 IP-Specific Error Codes

The following IP-specific error codes are supported:

IPERR_ADDRESS_IN_USE
: The address specified is already in use. For IP networks, this will usually occur if an attempt is made to open a socket with a port that is already in use.

IPERR_ADDRESS_RESOLUTION
: Unable to resolve address to a valid IP address.

IPERR_BAD_PARAM
: Call failed because of a bad parameter.

IPERR_CALLER_ID
: Unable to allocate or copy caller ID string.

IPERR_CANT_CLOSE_CHANNEL
: As a result of the circumstances under which this channel was opened, it cannot be closed. This could occur for some protocols in the scenario when channels are opened before the call is connected. In this case, the channels should be closed and deleted after hang-up.

IPERR_CHANNEL_ACTIVE
: Media channel is already active.

IPERR_COPYING_OCTET_STRING
: Unable to copy octet string.

IPERR_COPYING_OR_RESOLVING_ALIAS
: An error occurred while copying the alias. The error could be the result of a memory allocation failure or it could be an invalid alias format.

IPERR_DESTINATION_UNKNOWN
: Failure to locate the host with the address given.

IPERR_DIAL_ADDR_MUST_BE_ALIAS
: The address being dialed in this case may not be an IP address or domain name. It must be an alias because two intermediate addresses have already been specified, that is, Local Proxy, Remote Proxy and Gateway Address.

IPERR_DLL_LOAD_FAILED
: Dynamic load of a DLL failed.

IPERR_DTMF_PENDING
: Already in a DTMF generate state.

IPERR_DUP_CONF_ID
: A conference ID was specified that matches an existing conference ID for another conference.

IPERR_FRAMESPERPACKET_NOT_SUPP
: Setting frames-per-packet is not supported on the specified audio capability.

IPERR_GC_INVLINEDEV
: Invalid line device.

IPERR_HOST_NOT_FOUND
: Could not reach the party with the given host address.

IPERR_INCOMING_CALL_HANDLE
: The handle passed as the incoming call handle does not refer to a valid incoming call.

IPERR_INTERNAL
: An internal error occurred.

IPERR_INVALID_ADDRESS_TYPE
: The address type specified did not map to any known address type.

IPERR_INVALID_CAPS
: Channel open or response failed due to invalid capabilities.

IPERR_INVALID_DEST_ADDRESS
: The destination address did not conform to the type specified.

IPERR_INVALID_DOMAIN_NAME
: The domain name given is invalid.

IPERR_INVALID_DTMF_CHAR
: Invalid DTMF character sent.

IPERR_INVALID_EMAIL_ADDRESS
: The e-mail address given is invalid.

IPERR_INVALID_HOST_NAME
: The host name given is invalid.

IPERR_INVALID_ID
: An invalid ID was specified.

IPERR_INVALID_IP_ADDRESS
: The IP address given is invalid.

IPERR_INVALID_MEDIA_HANDLE
: The specified media handle is different from the already attached media handle.

IPERR_INVALID_PHONE_NUMBER
: The phone number given is invalid.

IPERR_INVALID_PROPERTY
: The property ID is invalid.

IPERR_INVALID_STATE
: Invalid state to make this call.

IPERR_INVALID_URL_ADDRESS
: The URL address given is invalid.

IPERR_INVDEVNAME
: Invalid device name.

IPERR_IP_ADDRESS_NOT_AVAILABLE
: The network socket layer reports that the IP address is not available. This can happen if the system does not have a correctly configured IP address.

IPERR_LOCAL_INTERNAL_PROXY_ADDR
: Local internal proxy specified could not be resolved to a valid IP address or domain name.

IPERR_MEDIA_NOT_ATTACHED
: No media resource was attached to the specified line device.

IPERR_MEMORY
: Memory allocation failure.

IPERR_MULTIPLE_CAPS
: Attaching a channel with multiple capabilities is not supported by this stack or it is not supported in this mode.

IPERR_MULTIPLE_DATATYPES
: Attaching a channel with multiple data types (such as audio and video) is not permitted. All media types proposed for a single channel must be of the same type.

IPERR_NO_AVAILABLE_PROPOSALS
: No available proposals to respond to.

IPERR_NO_CAPABILITIES_SPECIFIED
: No capabilities have been specified yet. They must either be pre-configured in the configuration file or they must be set using an extended capability API.

IPERR_NO_DTMF_CAPABILITY
: The remote endpoint does not have DTMF capability.

IPERR_NO_INTERSECTING_CAPABILITIES
: No intersecting capability found.

IPERR_NOANSWER
: Timeout due to no answer from peer.

IPERR_NOT_IMPLEMENTED
: The function or property call has not been implemented. This differs from IPERR_UNSUPPORTED in that there is the implication that this is an early release which intends to implement the feature or function.

IPERR_NOT_MULTIPOINT_CAPABLE
: The call cannot be accepted into a multipoint conference because there is no known multipoint controller, or the peer in a point-to-point conference is not multipoint capable.

IPERR_NULL_ADDRESS
: Address given is NULL.

IPERR_NULL_ALIAS
: The alias specified is NULL or empty.

IPERR_OK
: Successful completion.

IPERR_PEER_REJECT
: Peer has rejected the call placed from this endpoint.

IPERR_PENDING_RENEGOTIATION
: A batched channel renegotiation is already pending. This implementation does not support queuing of batched renegotiation.

IPERR_PROXY_GATEWAY_ADDR
: Two intermediate addresses were already specified in the local internal proxy and remote proxy addresses. The gateway address in this case cannot be used.

IPERR_REMOTE_PROXY_ADDR
: Remote proxy specified could not be resolved to a valid IP address or domain name.

IPERR_SERVER_REGISTRATION_FAILED
: Attempt to register with the registration and admission server (RAS) failed.

IPERR_STILL_REGISTERED
: The address object being deleted is still registered and cannot be deleted until it is unregistered.

IPERR_TIMEOUT
: Timeout occurred while executing an internal function.

IPERR_UNAVAILABLE
: The requested data is unavailable.

IPERR_UNDELETED_OBJECTS
: The object being deleted has child objects that have not been deleted.

IPERR_UNICODE_TO_ASCII
: Unable to convert the string or character from unicode or wide character format to ASCII.

IPERR_UNINITIALIZED
: The stack has not been initialized.

IPERR_UNKNOWN_API_GUID
: This is the result of either passing in a bogus GUID or one that is not found in the current DLL or executable.

IPERR_UNRESOLVABLE_DEST_ADDRESS
: No Gateway, Gatekeeper, or Proxy is specified, therefore the destination address must be a valid resolvable address. In the case of IP based call control, the address specified should be an IP address or a resolvable host or domain name.

IPERR_UNRESOLVABLE_HOST_NAME)
: The host or domain name could not be resolved to a valid address. This will usually occur if the host or domain name is not valid or is not accessible over the existing network.

IPERR_UNSUPPORTED
: This function or property call is unsupported in this configuration or implementation of stack. This differs from IPERR_NOT_IMPLEMENTED in that it implies no future plan to support this feature of property.

11.2 Error Codes When Using H.323

The following error codes are supported:

IPEC_addrRegistrationFailed
: Registration with the Registration and Admission server failed.

IPEC_addrListenFailed
: Stack was unable to register to listen for incoming calls.

IPEC_CHAN_REJECT_unspecified
: No cause for rejection specified.

IPEC_CHAN_REJECT_dataTypeNotSupported
: The terminal was not capable of supporting the dataType indicated in OpenLogicalChannel.

IPEC_CHAN_REJECT_dataTypeNotAvailable
: The terminal was not capable of supporting the dataType indicated in OpenLogicalChannel simultaneously with the dataTypes of logical channels that are already open.

IPEC_CHAN_REJECT_unknownDataType
: The terminal did not understand the dataType indicated in OpenLogicalChannel.

IPEC_CHAN_REJECT_insuffientBandwdith
: The channel could not be opened because permission to use the requested bandwidth for the logical channel was denied.

IPEC_CHAN_REJECT_unsuitableReverseParameters
: This code shall only be used to reject a bi-directional logical channel request when the only reason for rejection is that the requested parameters are inappropriate.

IPEC_CHAN_REJECT_dataTypeALCombinationNotSupported
: The terminal was not capable of supporting the dataType indicated in OpenLogicalChannel simultaneously with the Adaptation Layer type indicated in H223LogicalChannelParameters.

IPEC_CHAN_REJECT_multicastChannelNotAllowed
: Multicast Channel could not be opened.

IPEC_CHAN_REJECT_separateStackEstablishmentFailed
A request to run the data portion of a call on a separate stack failed.

IPEC_CHAN_REJECT_invalidSessionID
Attempt by the slave to set the SessionID when opening a logical channel to the master.

IPEC_CHAN_REJECT_masterSlaveConflict
Attempt by the slave to open logical channel in which the master has determined a conflict may occur.

IPEC_CHAN_REJECT_waitForCommunicationMode
Attempt to open a logical channel before the MC has transmitted the CommunicationModeCommand.

IPEC_CHAN_REJECT_invalidDependentChannel
Attempt to open a logical channel with a dependent channel specified that is not present.

IPEC_CHAN_REJECT_replacementForRejected
A logical channel of the type attempted cannot be opened using the replacement **For** parameter.The transmitter may wish to re-try by first closing the logical channel that is to be replaced, and then opening the replacement.

IPEC_CALL_END_timeout
A callback was received because a local timer expired.

IPEC_H245EstChannelFailure_MSDError
Establishment of optional H.245 channel in H.323 fast start connection failed due to error in MasterSlaveDetermination (MSD) exchange.

IPEC_H245EstChannelFailure_RemoteReject
Establishment of optional H.245 channel in H.323 fast start connection failed due to rejection on remote side.

IPEC_H245EstChannelFailure_TCSError
Establishment of optional H.245 channel in H.323 fast start connection failed due to error in TerminalCapabilitySet (TCS) exchange.

IPEC_H245EstChannelFailure_TransportError
Establishment of optional H.245 channel in H.323 fast start connection failed due to transport error.

IPEC_InternalError
An internal error occurred while executing asynchronously.

IPEC_INFO_NONE_NOMORE
No more digits are available.

IPEC_INFO_PRESENT_MORE
The requested digits are now available. More/additional digits are available.

IPEC_INFO_PRESENT_ALL
The requested digits are now available.

IPEC_INFO_NONE_TIMEOUT
No digits are available; timed out.

IPEC_INFO_SOME_NOMORE
Only some digits are available, no more digits will be received.

IPEC_INFO_SOME_TIMEOUT
: Only some digits are available; timed out.

IPEC_NO_MATCHING_CAPABILITIES
: No intersection was found between the proposed and matching capabilities.

IPEC_REG_FAIL_duplicateAlias
: The alias used to register with the Registration and Admission server is already registered. This failure typically results if the endpoint is already registered. It could also occur with some servers if a registration is attempted too soon after unregistering using the same alias.

IPEC_REG_FAIL_invalidCallSigAddress
: Server registration failed due to an invalid call signalling address specified.

IPEC_REG_FAIL_invalidAddress
: The local host address specified for communicating with the server is invalid.

IPEC_REG_FAIL_invalidAlias
: The alias specified did not conform to the format rules for the type of alias specified.

IPEC_REG_FAIL_invalidTermType
: An invalid terminal type was specified with the registration request.

IPEC_REG_FAIL_invalidTransport
: The transport type of the local host's address is not supported by the server.

IPEC_REG_FAIL_qosNotSupported
: The registration request announced a transport QoS that was not supported by the server.

IPEC_REG_FAIL_reRegistrationRequired
: Registration permission has expired. Registration should be performed again.

IPEC_REG_FAIL_resourcesUnavailable
: The server rejected the registration request due to unavailability of resources. This typically occurs if the server has already reached the maximum number of registrations it was configured to accept.

IPEC_REG_FAIL_securityDenied
: The server denied access for security reasons. This can occur if the password supplied does not match the password on file for the alias being registered.

IPEC_REG_FAIL_unknown
: The server refused to allow registration for an unknown reason.

IPEC_REG_FAIL_serverDown
: The server has gone down or is no longer responding.

IPEC_MEDIA_startSessionFailed
: Attempt to call **gc_media_StartSession( )** (an internal function) after establishing media channel returned error.

IPEC_MEDIA_TxFailed
: Attempt to establish or terminate a Tx channel with attached capabilities failed. The application is expected to keep the Rx capabilities unchanged in the next call to **gc_AttachEx( )**.

IPEC_MEDIA_RxFailed
: Attempt to establish or terminate an Rx channel with attached capabilities failed. The application is expected to keep the Tx capabilities unchanged in the next call to **gc_AttachEx( )**.

IPEC_MEDIA_TxRxFailed
: Attempts to establish or terminate Tx and Rx channels with attached capabilities failed.

IPEC_MEDIA_OnlyTxFailed
: Attempts to establish a Tx channel with attached capabilities failed. The status of other media channel is unavailable. Relevant to the GCEV_MEDIA_REJ event.

IPEC_MEDIA_OnlyRxFailed
: Attempts to establish an Rx channel with attached capabilities failed. The status of other media channel is unavailable. Relevant to the GCEV_MEDIA_REJ event.

IPEC_MEDIA_TxRequired
: Attempts to establish a Tx channel with attached capabilities failed.

IPEC_MEDIA_RxRequired
: Attempts to establish an Rx channel with attached capabilities failed.

IPEC_TxRx_Fail
: Both channels have failed to open.

IPEC_Tx_FailTimeout
: A Tx channel failed to open because of timeout.

IPEC_Rx_FailTimeout
: An Rx channel failed to open because of timeout.

IPEC_Tx_Fail
: A Tx channel failed to open for an unknown reason.

IPEC_Rx_Fail
: An Rx channel failed to open for an unknown reason.

IPEC_TxRx_FailTimeout
: Both the Tx and Rx channels failed because of a timeout.

IPEC_TxRx_Rej
: Both the Tx and Rx channels were rejected for an unknown reason.

IPEC_Tx_Rej
: Opening of a Tx channel was rejected for unknown reasons.

IPEC_Rx_Rej
: Opening of an Rx channel was rejected for unknown reasons.

IPEC_CHAN_FAILURE_unspecified
: The channel failed to open/close because of an unspecified reason.

IPEC_CHAN_FAILURE_timeout
: The channel failed to open/close because of a timeout.

IPEC_CHAN_FAILURE_localResources
: The channel failed to open/close because of limited resources.

IPEC_FAIL_TxRx_unspecified
Both the Tx and Rx channels failed to open for unspecified reasons.

IPEC_FAIL_TxUnspecifiedRxTimeout
A Tx channel failed to open for unspecified reasons and the Rx channel failed to open because of a timeout.

IPEC_FAILTxUnspecifiedRxResourceUnsuff
A Tx channel failed to open for unspecified reasons and the Rx channel failed to open because of insufficient resources.

IPEC_FAIL_RxUnspecifiedTxTimeout
An Rx channel failed to open for unspecified reasons and the Tx channel failed to open because of a timeout.

IPEC_FAIL_RXUnspecifiedTxResourceUnsuff
An Rx channel failed to open for unspecified reasons and the Tx channel failed to open because of insufficient resources.

IPEC_FAIL_TxTimeoutRxUnspecified
A Tx channel failed to open because of a timeout and the Rx channel failed to open for unspecified reasons.

IPEC_FAIL_TxRxTimeout
The Tx and Rx channels both failed to open because of a timeout.

IPEC_FAIL_TxTimeoutRxResourceUnsuff
A Tx channel failed to open because of a timeout and the Rx channel failed to open because of insufficient resources.

IPEC_FAIL_RxTimeoutTXUnspecified
An Rx channel failed because of a timeout and the Tx channel failed for unspecified reasons.

IPEC_FAIL_RxTimeoutTxResourceUnsuff
A Tx channel failed to open because of a timeout and the Rx channel failed to open because of insufficient resources.

IPEC_FAIL_TxResourceUnsuffRxUnspecified
A Tx channel failed to open because of insufficient resources and the Rx channel failed to open for unspecified reasons.

IPEC_FAIL_TxResourceUnsuffRxTimeout
A Tx channel failed to open because of insufficient resources and the Rx channel failed to open because of a timeout.

IPEC_FAIL_TxRxResourceUnsuff
Tx and Rx channels failed to open because of insufficient resources.

IPEC_FAIL_RxResourceUnsuffTxUnspecified
A Tx channel failed to open for unspecified reasons and the Rx channel failed to open because of insufficient resources.

IPEC_FAIL_RxResourceUnsuffTxTimeout
A Tx channel failed to open because of a timeout and the Rx channel failed to open because of insufficient resources.

11.3 Internal Disconnect Reasons

The following internal disconnect reasons are supported when using H.323:

IPEC_InternalReasonBusy (0x3e9, 1001 decimal)
    Cause 01; Busy

IPEC_InternalReasonCallCompletion (0x3ea, 1002 decimal)
    Cause 02; Call Completion

IPEC_InternalReasonCanceled (0x3eb, 1003 decimal)
    Cause 03; Cancelled

IPEC_InternalReasonCongestion (0x3ec, 1004 decimal)
    Cause 04; Network congestion

IPEC_InternalReasonDestBusy (0x3ed, 1005 decimal)
    Cause 05; Destination busy

IPEC_InternalReasonDestAddrBad (0x3ee, 1006 decimal)
    Cause 06; Invalid destination address

IPEC_InternalReasonDestOutOfOrder (0x3ef, 1007 decimal)
    Cause 07; Destination out of order

IPEC_InternalReasonDestUnobtainable (0x3f0, 1008 decimal)
    Cause 08; Destination unobtainable

IPEC_InternalReasonForward (0x3f1, 1009 decimal)
    Cause 09; Forward

IPEC_InternalReasonIncompatible (0x3f2, 1010 decimal)
    Cause 10; Incompatible

IPEC_InternalReasonIncomingCall, (0x3f3, 1011 decimal)
    Cause 11; Incoming call

IPEC_InternalReasonNewCall (0x3f4, 1012 decimal)
    Cause 12; New call

IPEC_InternalReasonNoAnswer (0x3f5, 1013 decimal)
    Cause 13; No answer from user

IPEC_InternalReasonNormal (0x3f6, 1014 decimal)
    Cause 14; Normal clearing

IPEC_InternalReasonNetworkAlarm (0x3f7, 1015 decimal)
    Cause 15; Network alarm

IPEC_InternalReasonPickUp (0x3f8, 1016 decimal)
    Cause 16; Pickup

IPEC_InternalReasonProtocolError (0x3f9, 1017 decimal)
    Cause 17; Protocol error

IPEC_InternalReasonRedirection (0x3fa, 1018 decimal)
    Cause 18; Redirection

IPEC_InternalReasonRemoteTermination (0x3fb, 1019 decimal)
Cause 19; Remote termination

IPEC_InternalReasonRejection (0x3fc, 1020 decimal)
Cause 20; Call rejected

IPEC_InternalReasonSIT (0x3fd, 1021 decimal)
Cause 21; Special Information Tone (SIT)

IPEC_InternalReasonSITCustIrreg (0x3fe, 1022 decimal)
Cause 22; SIT, Custom Irregular

IPEC_InternalReasonSITNoCircuit (0x3ff, 1023 decimal)
Cause 23; SIT, No Circuit

IPEC_InternalReasonSITReorder (0x400, 1024 decimal)
Cause 24; SIT, Reorder

IPEC_InternalReasonTransfer (0x401, 1025 decimal)
Cause 25; Transfer

IPEC_InternalReasonUnavailable (0x402, 1026 decimal)
Cause 26; Unavailable

IPEC_InternalReasonUnknown (0x403, 1027 decimal)
Cause 27; Unknown cause

IPEC_InternalReasonUnallocatedNumber (0x404, 1028 decimal)
Cause 28; Unallocated number

IPEC_InternalReasonNoRoute (0x405, 1029 decimal)
Cause 29; No route

IPEC_InternalReasonNumberChanged (0x406, 1030 decimal)
Cause 30; Number changed

IPEC_InternalReasonOutOfOrder (0x407, 1031 decimal)
Cause 31; Destination out of order

IPEC_InternalReasonInvalidFormat (0x408, 1032 decimal)
Cause 32; Invalid format

IPEC_InternalReasonChanUnavailable (0x409, 1033 decimal)
Cause 33; Channel unavailable

IPEC_InternalReasonChanUnacceptable (0x40a, 1034 decimal)
Cause 34; Channel unacceptable

IPEC_InternalReasonChanNotImplemented (0x40b, 1035 decimal)
Cause 35; Channel not implemented

IPEC_InternalReasonNoChan (0x40c, 1036 decimal)
Cause 36; No channel

IPEC_InternalReasonNoResponse (0x40d, 1037 decimal)
Cause 37; No response

IPEC_InternalReasonFacilityNotSubscribed (0x40e, 1038 decimal)
Cause 38; Facility not subscribed

IPEC_InternalReasonFacilityNotImplemented (0x40f, 1039 decimal)
Cause 39; Facility not implemented

IPEC_InternalReasonServiceNotImplemented (0x410, 1040 decimal)
Cause 40; Service not implemented

IPEC_InternalReasonBarredInbound (0x411, 1041 decimal)
Cause 41; Barred inbound calls

IPEC_InternalReasonBarredOutbound (0x412, 1042 decimal)
Cause 42; Barred outbound calls

IPEC_InternalReasonDestIncompatible (0x413, 1043 decimal)
Cause 43; Destination incompatible

IPEC_InternalReasonBearerCapUnavailable (0x414, 1044 decimal)
Cause 44; Bearer capability unavailable

11.4 Event Cause Codes and Failure Reasons When Using H.323

The following event cause codes apply when using H.323.

H.225.0 Cause Codes

IPEC_H2250ReasonNoBandwidth (0x7d0, 2000 decimal)
Maps to Q.931/Q.850 cause 34 - No circuit or channel available; indicates that there is no appropriate circuit/channel presently available to handle the call.

IPEC_H2250ReasonGatekeeperResource (0x7d1, 2001 decimal)
Maps to Q.931/Q.850 cause 47 - Resource unavailable; used to report a resource unavailable event only when no other cause in the resource unavailable class applies.

IPEC_H2250ReasonUnreachableDestination (0x7d2, 2002 decimal)
Maps to Q.931/Q.850 cause 3 - No route to destination; indicates that the called party cannot be reached because the network through which the call has been routed does not serve the destination desired.

IPEC_H2250ReasonDestinationRejection (0x7d3, 2003 decimal)
Maps to Q.931/Q.850 cause 16 - Normal call clearing - indicates that the call is being cleared because one of the users involved in the call has requested that the call be cleared.

IPEC_H2250ReasonInvalidRevision (0x7d4, 2004 decimal)
Maps to Q.931/Q.850 cause 88 - Incompatible destination; indicates that the equipment sending this cause has received a request to establish a call which has low layer compatibility, high layer compatibility, or other compatibility attributes (for example, data rate) which cannot be accommodated.

IPEC_H2250ReasonNoPermission (0x7d5, 2005 decimal)
Maps to Q.931/Q.850 cause 111 - Interworking, unspecified.

IPEC_H2250ReasonUnreachableGatekeeper (0x7d6, 2006 decimal)
Maps to Q.931/Q.850 cause 38 - Network out of order; indicates that the network is not functioning correctly and that the condition is likely to last a relatively long period of time, for example, immediately re-attempting the call is not likely to be successful.

IPEC_H2250ReasonGatewayResource (0x7d7, 2007 decimal)
Maps to Q.931/Q.850 cause 42 - Switching equipment congestion; indicates that the switching equipment generating this cause is experiencing a period of high traffic.

IPEC_H2250ReasonBadFormatAddress (0x7d8, 2008 decimal)
Maps to Q.931/Q.850 cause 28 - Invalid number format; indicates that the called party cannot be reached because the called party number is not in a valid format or is incomplete.

IPEC_H2250ReasonAdaptiveBusy (0x7d9, 2009 decimal)
Maps to Q.931/Q.850 cause 41 - Temporary failure; indicates that the network is not functioning correctly and that the condition is not likely to last for a long period of time, for example, the user may wish to try another call attempt almost immediately.

IPEC_H2250ReasonInConf (0x7da, 2010 decimal)
Maps to Q.931/Q.850 cause 17 - User busy; used to indicate that the called party is unable to accept another call because the user busy condition has been encountered. This cause value may be generated by the called user or by the network.

IPEC_H2250ReasonUndefinedReason (0x7db, 2011 decimal)
Maps to Q.931/Q.850 cause 31 - Normal, unspecified; Normal, unspecified; used to report a normal event only when no other cause in the normal class applies.

IPEC_H2250ReasonFacilityCallDeflection (0x7dc, 2012 decimal)
Maps to Q.931/Q.850 cause 16 - Normal call clearing - indicates that the call is being cleared because one of the users involved in the call has requested that the call be cleared.

IPEC_H2250ReasonSecurityDenied (0x7dd, 2013 decimal)
Maps to Q.931/Q.850 cause 31 - Normal, unspecified; Normal, unspecified; used to report a normal event only when no other cause in the normal class applies.

IPEC_H2250ReasonCalledPartyNotRegistered (0x7de, 2014 decimal)
Maps to Q.931/Q.850 cause 20 - Subscriber absent; used when a mobile station has logged off, radio contact is not obtained with a mobile station or if a personal telecommunication user is temporarily not addressable at any user-network interface.

IPEC_H2250ReasonCallerNotRegistered (0x7df, 2015 decimal)
Maps to Q.931/Q.850 cause 31 - Normal, unspecified; used to report a normal event only when no other cause in the normal class applies.

Q.931 Cause Codes

IPEC_Q931Cause01UnassignedNumber (0xbb9, 3001 decimal)
Q.931 cause 01 - Unallocated (unassigned) number; indicates that the called party cannot be reached because. Although the called party number is in a valid format, it is not currently allocated (assigned).

IPEC_Q931Cause02NoRouteToSpecifiedTransitNetwork (0xbba, 3002 decimal)
Q.931 cause 02 - No route to specified transit network (national use); indicates that the equipment sending this cause has received a request to route the call through a particular transit network which it does not recognize. The equipment sending this cause does not

recognize the transit network either because the transit network does not exist or because that particular transit network, while it does exist, does not serve the equipment which is sending this cause. This cause is supported on a network-dependent basis.

IPEC_Q931Cause03NoRouteToDestination (0xbbb, 3003 decimal)
Q.931 cause 03 - No route to destination; indicates that the called party cannot be reached because the network through which the call has been routed does not serve the destination desired. This cause is supported on a network-dependent basis.

IPEC_Q931Cause06ChannelUnacceptable (0xbbe, 3006 decimal)
Q.931 cause 06 - Channel unacceptable; indicates that the channel most recently identified is not acceptable to the sending entity for use in this call.

IPEC_Q931Cause07CallAwardedAndBeingDeliveredInAnEstablishedChannel (0xbbf, 3007 decimal)
Q.931 cause 07 - Call awarded and being delivered in an established channel; indicates that the user has been awarded the incoming call, and that the incoming call is being connected to a channel already established to that user for similar calls (e.g. packet-mode X.25 virtual calls).

IPEC_Q931Cause16NormalCallClearing (0xbc8, 3016 decimal)
Q.931 cause 16 - Normal call clearing; indicates that the call is being cleared because one of the user’s involved in the call has requested that the call be cleared. Under normal situations, the source of this cause is not the network.

IPEC_Q931Cause17UserBusy (0xbc9, 3017 decimal)
Q.931 cause 17 - User busy; used to indicate that the called party is unable to accept another call because the user busy condition has been encountered. This cause value may be generated by the called user or by the network.

IPEC_Q931Cause18NoUserResponding (0xbca, 3018 decimal)
Q.931 cause 18 - No user responding; used when a called party does not respond to a call establishment message with either an alerting or connect indication within the prescribed period of time allocated.

IPEC_Q931Cause19UserAlertingNoAnswer (0xbcb, 3019 decimal)
Q.931 cause 19 - No answer from user (user alerted); used when the called party has been alerted but does not respond with a connect indication within a prescribed period of time. This cause is not necessarily generated by Q.931 procedures but may be generated by internal network timers.

IPEC_Q931Cause21CallRejected (0xbcd, 3021 decimal)
Q.931 cause 21 - Call rejected; indicates that the equipment sending this cause does not wish to accept this call, although it could have accepted the call because the equipment sending this cause is neither busy nor incompatible. This cause may also be generated by the network, indicating that the call was cleared due to a supplementary service constraint. The diagnostic field may contain additional information about the supplementary service and reason for rejection.

IPEC_Q931Cause22NumberChanged (0xbce, 3022 decimal)
Q.931 cause 22 - Number changed; returned to a calling party when the called party number indicated by the calling party is no longer assigned. The new called party number may optionally be included in the diagnostic field. If a network does not support this cause value, cause No. 1, unallocated (unassigned) number should be used.

IPEC_Q931Cause26NonSelectUserClearing (0xbd2, 3026 decimal)
Q.931 cause 26 - Non-selected user clearing; indicates that the user has not been awarded the incoming call.

IPEC_Q931Cause27DestinationOutOfOrder (0xbd3, 3027 decimal)
Q.931 cause 27 - Destination out of order; indicates that the destination indicated by the user cannot be reached because the interface to the destination is not functioning correctly. The term "not functioning correctly" indicates that a signalling message was unable to be delivered to the remote party, for example, a physical layer or data link layer failure at the remote party, or user equipment off-line.

IPEC_Q931Cause28InvalidNumberFormatIncompleteNumber (0xbd4, 3028 decimal)
Q.931 cause 28 - Invalid number format (address incomplete); indicates that the called party cannot be reached because the called party number is not in a valid format or is not complete. Note: This condition may be determined immediately after reception of an ST signal or on time-out after the last received digit.

IPEC_Q931Cause29FacilityRejected (0xbd5, 3029 decimal)
Q.931 cause 29 - Facility rejected; returned when a supplementary service requested by the user cannot be provided by the network.

IPEC_Q931Cause30ResponseToSTATUSENQUIRY (0xbd6, 3030 decimal)
Q.931 cause 30 - Response to STATUS ENQUIRY; included in the STATUS message when the reason for generating the STATUS message was the prior receipt of a STATUS ENQUIRY message.

IPEC_Q931Cause31NormalUnspecified (0xbd7, 3031 decimal)
Q.931 cause 31 - Normal, unspecified; used to report a normal event only when no other cause in the normal class applies.

IPEC_Q931Cause34NoCircuitChannelAvailable (0xbda, 3034 decimal)
Q.931 cause 34 - No circuit/channel available; indicates that there is no appropriate circuit/channel presently available to handle the call.

IPEC_Q931Cause38NetworkOutOfOrder (0xbde, 3038 decimal)
Q.931 cause 38 - Network out of order; indicates that the network is not functioning correctly and that the condition is likely to last a relatively long period of time, that is, immediately re-attempting the call is not likely to be successful.

IPEC_Q931Cause41TemporaryFailure (0xbe1, 3041 decimal)
Q.931 cause 41 - Temporary failure; indicates that the network is not functioning correctly and that the condition is not likely to last a long period of time, that is, the user may wish to try another call attempt almost immediately.

IPEC_Q931Cause42SwitchingEquipmentCongestion (0xbe2, 3042 decimal)
Q.931 cause 42 - Switching equipment congestion; indicates that the switching equipment generating this cause is experiencing a period of high traffic.

IPEC_Q931Cause43AccessInformationDiscarded (0xbe3, 3043 decimal)
Q.931 cause 43 - Access information discarded; indicates that the network could not deliver access information to the remote user as requested, that is, user-to-user information, low layer compatibility, high layer compatibility, or sub-address, as indicated in the diagnostic. The particular type of access information discarded is optionally included in the diagnostic.

IPEC_Q931Cause44RequestedCircuitChannelNotAvailable (0xbe4, 3044 decimal)
Q.931 cause 44 - Requested circuit/channel not available; returned when the circuit or channel indicated by the requesting entity cannot be provided by the other side of the interface.

IPEC_Q931Cause47ResourceUnavailableUnspecified (0xbe7, 3047 decimal)
Q.931 cause 47 - Resource unavailable, unspecified; used to report a resource unavailable event only when no other cause in the resource unavailable class applies.

IPEC_Q931Cause57BearerCapabilityNotAuthorized (0xbf1, 3057 decimal)
Q.931 cause 57 - Bearer capability not authorized; indicates that the user has requested a bearer capability that is implemented by the equipment that generated this cause but the user is not authorized to use.

IPEC_Q931Cause58BearerCapabilityNotPresentlyAvailable (0xbf2, 3058 decimal)
Q.931 cause 58 - Bearer capability not presently available; indicates that the user has requested a bearer capability that is implemented by the equipment that generated this cause but it is not available at this time.

IPEC_Q931Cause63ServiceOrOptionNotAvailableUnspecified (0xbf7, 3063 decimal)
Q.931 cause 63 - Service or option not available, unspecified; used to report a service or option not available event only when no other cause in the service or option not available class applies.

IPEC_Q931Cause65BearCapabilityNotImplemented (0xbf9, 3065 decimal)
Q.931 cause 65 - Bearer capability not implemented; indicates that the equipment sending this cause does not support the bearer capability requested.

IPEC_Q931Cause66ChannelTypeNotImplemented (0xbfa, 3066 decimal)
Q.931 cause 66 - Channel type not implemented; indicates that the equipment sending this cause does not support the channel type requested.

IPEC_Q931Cause69RequestedFacilityNotImplemented (0xbfd, 3069 decimal)
Q.931 cause 69 - Requested facility not implemented; indicates that the equipment sending this cause does not support the requested supplementary service.

IPEC_Q931Cause70OnlyRestrictedDigitalInformationBearerCapabilityIsAvailable (0xbfe, 3070 decimal)
Q.931 cause 70 - Only restricted digital information bearer capability is available (national use); indicates that the calling party has requested an unrestricted bearer service but that the equipment sending this cause only supports the restricted version of the requested bearer capability.

IPEC_Q931Cause79ServiceOrOptionNotImplementedUnspecified (0xc07, 3079 decimal)
Q.931 cause 79 - Service or option not implemented, unspecified; used to report a service or option not implemented event only when no other cause in the service or option not implemented class applies.

IPEC_Q931Cause81InvalidCallReferenceValue (0xc09, 3081 decimal)
Q.931 cause 81 - Invalid call reference value; indicates that the equipment sending this cause has received a message with a call reference that is not currently in use on the user-network interface.

IPEC_Q931Cause82IdentifiedChannelDoesNotExist (0xc0a, 3082 decimal)
Q.931 cause 82 - Identified channel does not exist; indicates that the equipment sending this cause has received a request to use a channel not activated on the interface for a call. For example, if a user has subscribed to those channels on a primary rate interface numbered from

1 to 12 and the user equipment or the network attempts to use channels 13 through 23, this cause is generated.

IPEC_Q931Cause83AsuspendedCallExistsButThisCallIdentityDoesNot (0xc0b, 3083 decimal)
Q.931 cause 83 - A suspended call exists, but this call identity does not; indicates that a call resume has been attempted with a call identity that differs from that in use for any presently suspended call(s).

IPEC_Q931Cause84CallIdentityInUse (0xc0c, 3084 decimal)
Q.931 cause 84 - Call identity in use; indicates that the network has received a call suspended request containing a call identity (including the null call identity) that is already in use for a suspended call within the domain of interfaces over which the call might be resumed.

IPEC_Q931Cause85NoCallSuspended (0xc0d, 3085 decimal)
Q.931 cause 85 - No call suspended; indicates that the network has received a call resume request containing a call identity information element that presently does not indicate any suspended call within the domain of interfaces over which calls may be resumed.

IPEC_Q931Cause86CallHavingTheRequestedCallIdentityHasBeenCleared (0xc0e, 3086 decimal)
Q.931 cause 86 - Call having the requested call identity has been cleared; indicates that the network has received a call resume request containing a call identity information element indicating a suspended call that has in the meantime been cleared while suspended (either by network timeout or by the remote user).

IPEC_Q931Cause88IncompatibleDestination (0xc10, 3088 decimal)
Q.931 cause 88 - Incompatible destination; indicates that the equipment sending this cause has received a request to establish a call that has low layer compatibility, high layer compatibility, or other compatibility attributes (for example, data rate) that cannot be accommodated.

IPEC_Q931Cause91InvalidTransitNetworkSelection (0xc13, 3091 decimal)
Q.931 cause 91 - Invalid transit network selection (national use); indicates that a transit network identification was received that is of an incorrect format as defined by Annex C/Q.931.

IPEC_Q931Cause95InvalidMessageUnspecified (0xc17, 3095 decimal)
Q.931 cause 95 - Invalid message, unspecified; used to report an invalid message event only when no other cause in the invalid message class applies.

IPEC_Q931Cause96MandatoryInformationElementMissing (0xc18, 3096 decimal)
Q.931 cause 96 - Mandatory information element is missing; indicates that the equipment sending this cause has received a message that is missing an information element that must be present in the message before that message can be processed.

IPEC_Q931Cause97MessageTypeNonExistentOrNotImplemented (0xc19, 3097 decimal)
Q.931 cause 97 - Message type non-existent or not implemented; indicates that the equipment sending this cause has received a message with a message type it does not recognize either because 1) the message type is not defined or 2) the message type is defined but not implemented by the equipment sending this cause.

IPEC_Q931Cause100InvalidInformationElementContents (0xc1c, 3100 decimal)
Q.931 cause 100 - Invalid information element contents; indicates that the equipment sending this cause has received an information element that it has implemented; however, one or more fields in the information element are coded in such a way that has not been implemented by the equipment sending this cause.

IPEC_Q931Cause101MessageNotCompatibleWithCallState (0xc1d, 3101 decimal)
: Q.931 cause 101 - Message not compatible with call state; indicates that a message that is incompatible with the call state has been received.

IPEC_Q931Cause102RecoveryOnTimeExpiry (0xc1e, 3102 decimal)
: Q.931 cause 102 - Recovery on timer expiry; indicates that a procedure has been initiated by the expiry of a timer in association with error handling procedures.

IPEC_Q931Cause111ProtocolErrorUnspecified (0xc27, 3111 decimal)
: Q.931 cause 111 - Protocol error, unspecified; used to report a protocol error event only when no other cause in the protocol error class applies.

IPEC_Q931Cause127InterworkingUnspecified (0xc37, 3127 decimal)
: Q.931 cause 127 - Interworking, unspecified; indicates that there has been interworking with a network that does not provide causes for the actions it takes. Thus, the precise cause for a message that is being sent cannot be ascertained.

RAS Failure Reasons

IPEC_RASReasonResourceUnavailable (0xfa1, 4001 decimal)
: Resources have been exhausted. (In GRJ, RRJ, ARJ, and LRJ messages.)

IPEC_RASReasonInsufficientResources (0xfa2, 4002 decimal)
: Insufficient resources to complete the transaction. (In BRJ messages.)

IPEC_RASReasonInvalidRevision (0xfa3, 4003 decimal)
: The registration version is invalid. (In GRJ, RRJ, and BRJ messages.)

IPEC_RASReasonInvalidCallSignalAddress (0xa4, 4004 decimal)
: The call signal address is invalid. (In RRJ messages.)

IPEC_RASReasonInvalidIPEC_RASAddress (0xfa5, 4005 decimal)
: The supplied address is invalid. (In RRJ messages.)

IPEC_RASReasonInvalidTerminalType (0xfa6, 4006 decimal)
: The terminal type is invalid. (In RRJ messages.)

IPEC_RASReasonInvalidPermission (0xfa7, 4007 decimal)
: Permission has expired. (In ARJ messages.)

: A true permission violation. (In BRJ messages.)

: Exclusion by administrator or feature. (In LRJ messages.)

IPEC_RASReasonInvalidConferenceID (0xfa8, 4008 decimal)
: Possible revision. (In BRJ messages.)

IPEC_RASReasonInvalidEndpointID (0xfa9, 4009 decimal)
: The endpoint registration ID is invalid. (In ARJ messages.)

IPEC_RASReasonCallerNotRegistered (0xfaa, 4010 decimal)
: The call originator is not registered. (In ARJ messages.)

IPEC_RASReasonCalledPartyNotRegistered (0xfab, 4011 decimal)
: Unable to translate the address. (In ARJ messages.)

IPEC_RASReasonDiscoveryRequired (0xfac, 4012 decimal)
: Registration permission has expired. (In RRJ messages.)

IPEC_RASReasonDuplicateAlias (0xfad, 4013 decimal)
The alias is registered to another endpoint. (In RRJ messages.)

IPEC_RASReasonTransportNotSupported (0xfae, 4014 decimal)
One or more of the transport addresses are not supported. (In RRJ messages.)

IPEC_RASReasonCallInProgress (0xfaf, 4015 decimal)
A call is already in progress. (In URJ messages.)

IPEC_RASReasonRouteCallToGatekeeper (0xfb0, 4016 decimal)
The call has been routed to a gatekeeper. (In ARJ messages.)

IPEC_RASReasonRequestToDropOther (0xfb1, 4017 decimal)
Unable to request to drop the call for others. (In DRJ messages.)

IPEC_RASReasonNotRegistered (0xfb2, 4018 decimal)
Not registered with a gatekeeper. (In DRJ, LRJ, and INAK messages.)

IPEC_RASReasonUndefined (0xfb3, 4019 decimal)
Unknown reason. (In GRJ, RRJ, URJ, ARJ, BRJ, LRJ, and INAK messages.)

IPEC_RASReasonTerminalExcluded (0xfb4, 4020 decimal)
Permission failure and not a resource failure. (In GRQ messages.)

IPEC_RASReasonNotBound (0xfb5, 4021 decimal)
Discovery permission has expired. (In BRJ messages.)

IPEC_RASReasonNotCurrentlyRegistered (0xfb6, 4022 decimal)
The endpoint is not registered. (In URJ messages.)

IPEC_RASReasonRequestDenied (0xfb7, 4023 decimal)
No bandwidth is available. (In ARJ messages.)

Unable to find location. (In LRJ messages.)

IPEC_RASReasonLocationNotFound (0xfb8, 4024 decimal)
Unable to find location. (In LRJ messages.)

IPEC_RASReasonSecurityDenial (0xfb9, 4025 decimal)
Security access has been denied. (In GRJ, RRJ, URJ, ARJ, BRJ, LRJ, DRJ, and INAK messages.)

IPEC_RASTransportQOSNotSupported (0xfba, 4026 decimal)
QOS is not supported by this gatekeeper. (In RRJ messages.)

IPEC_RASResourceUnavailable (0xfbb, 4027 decimal)
Resources have been exhausted. (In GRJ, RRJ, ARJ and LRJ messages.)

IPEC_RASInvalidAlias (0xfbc, 4028 decimal)
The alias is not consistent with gatekeeper rules. (In RRJ messages.)

IPEC_RASPermissionDenied (0xfbd, 4029 decimal)
The requesting user is not allowed to unregistered the specified user. (In URJ messages.)

IPEC_RASQOSControlNotSupported (0xfbe, 4030 decimal)
QOS control is not supported. (In ARJ messages.)

IPEC_RASIncompleteAddress (0xfbf, 4031 decimal)
The user address is incomplete. (In ARJ messages.)

IPEC_RASFullRegistrationRequired (0xfc0, 4032 decimal)
Registration permission has expired. (In RRJ messages.)

IPEC_RASRouteCallToSCN (0xfc1, 4033 decimal)
The call was routed to a switched circuit network. (In ARJ and LRJ messages.)

IPEC_RASAliasesInconsistent (0xfc2, 4034 decimal)
Multiple aliases in the request identify separate people. (In ARJ and LRJ messages.)

11.5 Failure Response Codes When Using SIP

The following failure response codes apply when using SIP. Each code is followed by a description. The codes are listed in code value order.

Request Failure Response Codes (4xx)

IPEC_SIPReasonStatus400BadRequest (0x1518, 5400 decimal)
SIP Request Failure Response 400 - Bad Request - The request could not be understood due to malformed syntax. The Reason-Phrase should identify the syntax problem in more detail, for example, “Missing Call-ID header field”.

IPEC_SIPReasonStatus401Unauthorized (0x1519, 5401 decimal)
SIP Request Failure Response 401 - Unauthorized - The request requires user authentication. This response is issued by User Agent Servers (UASs) and registrars, while 407 (Proxy Authentication Required) is used by proxy servers.

IPEC_SIPReasonStatus402PaymentRequired (0x151a, 5402 decimal)
SIP Request Failure Response 402 - Payment Required - Reserved for future use.

IPEC_SIPReasonStatus403Forbidden (0x151b, 5403 decimal)
SIP Request Failure Response 403 - Forbidden - The server understood the request, but is refusing to fulfill it. Authorization will not help, and the request should not be repeated.

IPEC_SIPReasonStatus404NotFound (0x151c, 5404 decimal)
SIP Request Failure Response 404 - Not Found - The server has definitive information that the user does not exist at the domain specified in the Request-URI. This status is also returned if the domain in the Request-URI does not match any of the domains handled by the recipient of the request.

IPEC_SIPReasonStatus405MethodNotAllowed (0x151d, 5405 decimal)
SIP Request Failure Response 405 - Method Not Allowed - The method specified in the Request-Line is understood, but not allowed for the address identified by the Request-URI. The response must include an Allow header field containing a list of valid methods for the indicated address.

IPEC_SIPReasonStatus406NotAcceptable (0x151e, 5406 decimal)
SIP Request Failure Response 406 - Not Acceptable - The resource identified by the request is only capable of generating response entities that have content characteristics not acceptable according to the Accept header field sent in the request.

IPEC_SIPReasonStatus407ProxyAuthenticationRequired (0x151f, 5407 decimal)
SIP Request Failure Response 407 - Proxy Authentication Required - This code is similar to 401 (Unauthorized), but indicates that the client must first authenticate itself with the proxy.

This status code can be used for applications where access to the communication channel (for example, a telephony gateway) rather than the callee, requires authentication.

IPEC_SIPReasonStatus408RequestTimeout (0x1520, 5408 decimal)
SIP Request Failure Response 408 - Request Timeout - The server could not produce a response within a suitable amount of time, for example, if it could not determine the location of the user in time. The client may repeat the request without modifications at any later time.

IPEC_SIPReasonStatus410Gone (0x1522, 5410 decimal)
SIP Request Failure Response 410 - Gone - The requested resource is no longer available at the server and no forwarding address is known. This condition is expected to be considered permanent. If the server does not know, or has no facility to determine, whether or not the condition is permanent, the status code 404 (Not Found) should be used instead.

IPEC_SIPReasonStatus413RequestEntityTooLarge (0x1525, 5413 decimal)
SIP Request Failure Response 413 - Request Entity Too Large - The server is refusing to process a request because the request entity-body is larger than the server is willing or able to process. The server may close the connection to prevent the client from continuing the request. If the condition is temporary, the server should include a Retry-After header field to indicate that it is temporary and after what time the client may try again.

IPEC_SIPReasonStatus414RequestUriTooLong (0x1526, 5414 decimal)
SIP Request Failure Response 414 - Request-URI Too Long - The server is refusing to service the request because the Request-URI is longer than the server is willing to interpret.

IPEC_SIPReasonStatus415UnsupportedMediaType (0x1527, 5415 decimal)
SIP Request Failure Response 415 - Unsupported Media Type - The server is refusing to service the request because the message body of the request is in a format not supported by the server for the requested method. The server must return a list of acceptable formats using the Accept, Accept-Encoding, or Accept-Language header field, depending on the specific problem with the content.

IPEC_SIPReasonStatus416UnsupportedURIScheme (0x1528, 5416 decimal)
SIP Request Failure Response 416 - Unsupported URI Scheme - The server cannot process the request because the scheme of the URI in the Request-URI is unknown to the server.

IPEC_SIPReasonStatus420BadExtension (0x153c, 5420 decimal)
SIP Request Failure Response 420 - Bad Extension - The server did not understand the protocol extension specified in a Proxy-Require or Require header field. The server must include a list of the unsupported extensions in an Unsupported header field in the response.

IPEC_SIPReasonStatus421ExtensionRequired (0x153d, 5421 decimal)
SIP Request Failure Response 421 - Extension Required - The User Agent Server (UAS) needs a particular extension to process the request, but this extension is not listed in a Supported header field in the request. Responses with this status code must contain a Require header field listing the required extensions. A UAS should not use this response unless it truly cannot provide any useful service to the client. Instead, if a desirable extension is not listed in the Supported header field, servers should process the request using baseline SIP capabilities and any extensions supported by the client.

IPEC_SIPReasonStatus422SessionIntervalTooSmall (0x153e, 5422
SIP Request Failure Response 422 - Session interval too small - The SIP session timer duration set in the SIP_SessionTimer_MinSE field (IP_VIRTBOARD structure) is too small. The response contains a Min-SE header field identifying the minimum session interval supported.

IPEC_SIPReasonStatus423IntervalTooBrief (0x153f, 5423 decimal)
SIP Request Failure Response 423 - Interval Too Brief - The server is rejecting the request because the expiration time of the resource refreshed by the request is too short. This response can be used by a registrar to reject a registration whose Contact header field expiration time was too small.

IPEC_SIPReasonStatus480TemporarilyUnavailable (0x1568, 5480 decimal)
SIP Request Failure Response 480 - Temporarily Unavailable - The callee's end system was contacted successfully but the callee is currently unavailable (for example, is not logged in, logged in but in a state that precludes communication with the callee, or has activated the "do not disturb" feature). The response may indicate a better time to call in the Retry-After header field. The user could also be available elsewhere (unbeknownst to this server). The reason phrase should indicate a more precise cause as to why the callee is unavailable. This value should be settable by the User Agent (UA). Status 486 (Busy Here) may be used to more precisely indicate a particular reason for the call failure. This status is also returned by a redirect or proxy server that recognizes the user identified by the Request-URI, but does not currently have a valid forwarding location for that user.

IPEC_SIPReasonStatus481CallTransactionDoesNotExist (0x1569, 5481 decimal)
SIP Request Failure Response 481 - Call/Transaction Does Not Exist - This status indicates that the User Agent Server (UAS) received a request that does not match any existing dialog or transaction.

IPEC_SIPReasonStatus482LoopDetected (0x156a, 5482 decimal)
SIP Request Failure Response 482 - Loop Detected - The server has detected a loop.

IPEC_SIPReasonStatus483TooManyHops (0x156b, 5483 decimal)
SIP Request Failure Response 483 - Too Many Hops - The server received a request that contains a Max-Forwards header field with the value zero.

IPEC_SIPReasonStatus484AddressIncomplete (0x156c, 5484 decimal)
SIP Request Failure Response 484 - Address Incomplete - The server received a request with a Request-URI that was incomplete. Additional information should be provided in the reason phrase. This status code allows overlapped dialing. With overlapped dialing, the client does not know the length of the dialing string. It sends strings of increasing lengths, prompting the user for more input, until it no longer receives a 484 (Address Incomplete) status response.

IPEC_SIPReasonStatus485Ambiguous (0x156d, 5485 decimal)
SIP Request Failure Response 485 - The Request-URI was ambiguous. The response may contain a listing of possible unambiguous addresses in Contact header fields. Revealing alternatives can infringe on privacy of the user or the organization. It must be possible to configure a server to respond with status 404 (Not Found) or to suppress the listing of possible choices for ambiguous Request-URIs.

IPEC_SIPReasonStatus486BusyHere (0x156e, 5486 decimal)
SIP Request Failure Response 486 - Busy Here - The callee's end system was contacted successfully, but the callee is currently not willing or able to take additional calls at this end system. The response may indicate a better time to call in the Retry-After header field. The user could also be available elsewhere, such as through a voice mail service. Status 600 (Busy Everywhere) should be used if the client knows that no other end system will be able to accept this call.

IPEC_SIPReasonStatus487RequestTerminated (0x156f, 5487 decimal)
SIP Request Failure Response 487 - Request Terminated - The request was terminated by a BYE or CANCEL request. This response is never returned for a CANCEL request itself.

IPEC_SIPReasonStatus488NotAcceptableHere (0x1570, 5488 decimal)
SIP Request Failure Response 488 - Not Acceptable Here - The response has the same meaning as 606 (Not Acceptable), but only applies to the specific resource addressed by the Request-URI and the request may succeed elsewhere. A message body containing a description of media capabilities may be present in the response, which is formatted according to the Accept header field in the INVITE (or application/SDP if not present), the same as a message body in a 200 (OK) response to an OPTIONS request.

IPEC_SIPReasonStatus491RequestPending (0x1573, 5491 decimal)
SIP Request Failure Response 491 - Request Pending - The request was received by a User Agent Server (UAS) that had a pending request within the same dialog.

IPEC_SIPReasonStatus493Undecipherable (0x1575, 5493 decimal)
SIP Request Failure Response 493 - Undecipherable - The request was received by a User Agent Server (UAS) that contained an encrypted MIME body for which the recipient does not possess or will not provide an appropriate decryption key. This response may have a single body containing an appropriate public key that should be used to encrypt MIME bodies sent to this User Agent (UA).

Server Failure Response Codes (5xx)

IPEC_SIPReasonStatus500ServerInternalError (0x157c, 5500 decimal)
Server Failure Response 500 - Server Internal Error - The server encountered an unexpected condition that prevented it from fulfilling the request. The client may display the specific error condition and may retry the request after several seconds. If the condition is temporary, the server may indicate when the client may retry the request using the Retry-After header field.

IPEC_SIPReasonStatus501NotImplemented (0x157d, 5501 decimal)
Server Failure Response 501 - Not Implemented - The server does not support the functionality required to fulfill the request. This is the appropriate response when a User Agent Server (UAS) does not recognize the request method and is not capable of supporting it for any user. Proxies forward all requests regardless of method. Note that a 405 (Method Not Allowed) is sent when the server recognizes the request method, but that method is not allowed or supported.

IPEC_SIPReasonStatus502BadGateway (0x157e, 5502 decimal)
Server Failure Response 502 - Bad Gateway - The server, while acting as a gateway or proxy, received an invalid response from the downstream server it accessed in attempting to fulfill the request.

IPEC_SIPReasonStatus503ServiceUnavailable (0x157f, 5503 decimal)
Server Failure Response 503 - Service Unavailable - The server is temporarily unable to process the request due to a temporary overloading or maintenance of the server or the use of an unsupported transport protocol (for example, TCP). The server may indicate when the client should retry the request in a Retry-After header field. If no Retry-After is given, the client must act as if it had received a 500 (Server Internal Error) response. A client (proxy or User Agent Client) receiving a 503 (Service Unavailable) should attempt to forward the request to an alternate server. It should not forward any other requests to that server for the

duration specified in the Retry-After header field, if present. Servers may refuse the connection or drop the request instead of responding with 503 (Service Unavailable).

IPEC_SIPReasonStatus504ServerTimeout (0x1580, 5504 decimal)
Server Failure Response 504 - Server Time-out - The server did not receive a timely response from an external server it accessed in attempting to process the request. 408 (Request Timeout) should be used instead if there was no response within the period specified in the Expires header field from the upstream server.

IPEC_SIPReasonStatus505VersionNotSupported (0x1581, 5505 decimal)
Server Failure Response 505 - Version Not Supported - The server does not support, or refuses to support, the SIP protocol version that was used in the request. The server is indicating that it is unable or unwilling to complete the request using the same major version as the client, other than with this error message.

IPEC_SIPReasonStatus513MessageTooLarge (0x1589, 5513 decimal)
Server Failure Response 513 - Message Too Large - The server was unable to process the request since the message length exceeded its capabilities.

Global Failure Response Codes (6xx)

IPEC_SIPReasonStatus600BusyEverywhere (0x15e0, 5600 decimal)
SIP Global Failure Response 600 - Busy Everywhere - The callee's end system was contacted successfully but the callee is busy and does not wish to take the call at this time. The response may indicate a better time to call in the Retry-After header field. If the callee does not wish to reveal the reason for declining the call, the callee uses status code 603 (Decline) instead. This status response is returned only if the client knows that no other end point (such as a voice mail system) will answer the request. Otherwise, 486 (Busy Here) should be returned.

IPEC_SIPReasonStatus603Decline (0x15e3, 5603 decimal)
SIP Global Failure Response 603 - 603 Decline - The callee's machine was successfully contacted but the user explicitly does not wish to or cannot participate. The response may indicate a better time to call in the Retry-After header field. This status response is returned only if the client knows that no other end point will answer the request.

IPEC_SIPReasonStatus604DoesNotExistAnywhere (0x15e4, 5604 decimal)
SIP Global Failure Response 604 - Does Not Exist Anywhere - The server has authoritative information that the user indicated in the Request-URI does not exist anywhere.

IPEC_SIPReasonStatus606NotAcceptable (0x15e6, 5606 decimal)
SIP Global Failure Response 606 - Not Acceptable - The user's agent was contacted successfully but some aspects of the session description such as the requested media,

bandwidth, or addressing style were not acceptable. A 606 (Not Acceptable) response means that the user wishes to communicate, but cannot adequately support the session described.

The 606 (Not Acceptable) response may contain a list of reasons in a Warning header field describing why the session described cannot be supported.

A message body containing a description of media capabilities may be present in the response, which is formatted according to the Accept header field in the INVITE (or application/SDP if not present), the same as a message body in a 200 (OK) response to an OPTIONS request.

It is hoped that negotiation will not frequently be needed, and when a new user is being invited to join an already existing conference, negotiation may not be possible. It is up to the invitation initiator to decide whether or not to act on a 606 (Not Acceptable) response.

This status response is returned only if the client knows that no other end point will answer the request.

Other SIP Codes (8xx)

IPEC_SIPReasonStatusBYE (0x16a8, 5800 decimal)
SIP reason status 800. BYE code.

IPEC_SIPReasonStatusCANCEL (0x16a9, 5801 decimal)
SIP reason status 801. CANCEL code.

SIP Message Error Codes

IPEC_MIME_BUFF_TOO_SMALL
MIME buffer size is smaller than the incoming MIME part in a SIP message.

IPEC_MIME_POOL_EMPTY
MIME memory pool is exhausted.

IPEC_SipHeaderTruncation
A SIP header field exceeded the configured maximum parameter length and was truncated.

SIP Registration Error Codes

IPEC_REG_FAIL_insufficientInternalResources
The SIP stack ran out of resources to process request.

IPEC_REG_FAIL_internalError
An internal IP Call Control Library error was encountered while attempting to form an outgoing REGISTER request.

IPEC_REG_FAIL_invalidExpires
The value of the “expires=” parameter in the Contact: header field was invalid for the current operation.

IPEC_REG_FAIL_networkError
A network error prevented the REGISTER request from being sent.

IPEC_REG_FAIL_registrationTransactionInProgress
A REGISTER transaction is currently in progress with the specified Registrar and Address of Record. A new request to this same Registrar and AOR cannot be generated at this time, and you should try again after the current pending request completes.

IPEC_REG_FAIL_responseTimeout
: There was a timeout error while waiting for a REGISTER response from the Registrar.

IPEC_REG_FAIL_serverResponseDataMismatch
: There was a mismatch between the internal IP Call Control library data and the data contained in the Registrar's response.

Supplementary Reference Information 12

This chapter lists related publications and includes other reference information as follows:

12.1 References to More Information

The following publications provide related information:

- IETF RFC 2246, *The TLS Protocol*, *http://www.ietf.org/rfc/rfc2246.txt*
- IETF RFC 2327, *SDP: Session Description Protocol*, *http://ietf.org/rfc/rfc2327.txt*
- IETF RFC 2976, *The SIP INFO Method*, *http://ietf.org/rfc/rfc2976.txt*
- IETF RFC 3261, *Session Initiation Protocol (SIP)*, *http://ietf.org/rfc/rfc3261.txt*
- IETF RFC 3265, *Session Initiation Protocol (SIP)-Specific Event Notification*, *http://ietf.org/rfc/rfc3265.txt*
- IETF RFC 3515, *The Session Initiation Protocol (SIP) Refer Method*, *http://ietf.org/rfc/rfc3515.txt*
- IETF RFC 3550, *RTP: A Transport Protocol for Real-Time Applications*, *http://ietf.org/rfc/rfc3550.txt*
- ITU-T Recommendation H.225.0 (09/99) - Call signaling protocols and media stream packetization for packet-based multimedia communications systems
- ITU-T Recommendation H.245 (07/01) - Control protocol for multimedia communication
- ITU-T Recommendation H.323 (11/00) - Packet-based multimedia communications systems
- ITU-T Recommendation H.450.2, Call transfer supplementary service for H.323
- ITU-T Recommendation T.30 (07/96) - Procedures for document facsimile transmission in the general switched telephone network
- ITU-T Recommendation T.38 (06/98) - Procedures for real-time Group 3 facsimile communication over networks
- Black, Uyless, *Voice over IP*, Prentice Hall PTR, Prentice-Hall, Inc., 2000
- Douskalis, Bill, *IP Telephony; The Integration of Robust VoIP Services*, Prentice Hall PTR, Prentice-Hall, Inc., ISBN 0-13-014118-6
- Galtieri, Paolo*, Introduction to Voice Over the Internet Protocol*, Applied Computing Technologies, Winter 2000

12.2 SIP Transaction Timer Values

This section provides information on the various timers that affect Global Call's behavior when handling SIP transactions. The duration values for these timers are preconfigured and are not configurable by applications.

retransmissionT1

T1 determines several timers as defined in RFC3261. For example, when an unreliable transport protocol is used, a Client Invite transaction retransmits requests at an interval that starts at T1 seconds and doubles after every retransmission. A Client General transaction retransmits requests at an interval that starts at T1 and doubles until it reaches T2.

- Standard duration: 1000 ms

retransmissionT2

T2 determines the maximum retransmission interval as defined in RFC 3261. For example, when an unreliable transport protocol is used, general requests are retransmitted at an interval which starts at T1 and doubles until it reaches T2. If a provisional response is received, retransmissions continue, but at an interval of T2.

- Standard duration: 8000 ms

retransmissionT4

T4 represents the amount of time the network takes to clear messages between client and server transactions as defined in RFC 3261. For example, when working with an unreliable transport protocol, T4 determines the time that a UAS waits after receiving an ACK message and before terminating the transaction.

- Standard duration: 10000 ms

generalLingerTimer

After a server sends a final response, the server cannot be sure that the client has received the response message. The server should be able to retransmit the response upon receiving retransmissions of the request for generalLingerTimer milliseconds.

- Standard duration: 32000 ms

inviteLingerTimer

After sending an ACK for an INVITE final response, a client cannot be sure that the server has received the ACK message. The client should be able to retransmit the ACK upon receiving retransmissions of the final response for inviteLingerTimer milliseconds.

- Standard duration: 32000 ms

provisionalTimer

The provisionalTimer is set when receiving a provisional response on an Invite transaction. The transaction will stop retransmissions of the Invite request and will wait for a final response until the provisionalTimer expires. If you set the provisionalTimer to zero (0), no timer is set. The Invite transaction will wait indefinitely for the final response.

- Standard duration: 32000 ms

cancelGeneralNoResponseTimer

When sending a CANCEL request on a General transaction, the User Agent waits cancelGeneralNoResponseTimer milliseconds before timeout termination if there is no response for the cancelled transaction.

- Standard duration: 32000 ms

cancelInviteNoResponseTimer
: When sending a CANCEL request on an Invite request, the User Agent waits cancelInviteNoResponseTimer milliseconds before timeout termination if there is no response for the cancelled transaction.

- Standard duration: 32000 ms

generalRequestTimeoutTimer
: After sending a General request, the User Agent waits for a final response generalRequestTimeoutTimer milliseconds before timeout termination (in this time the User Agent retransmits the request every T1, 2*T1, ... , T2, ... milliseconds).

- Standard duration: 32000 ms

12.3 DNS Configuration for SIP

SIP uses DNS procedures to allow a client to resolve a SIP Uniform Resource Identifier (URI) into the IP address, port, and transport protocol of the next hop to contact. The behavior of the Dialogic® Global Call API library complies with the client DNS procedures defined in IETF RFC 3263, including use of NAPTR and SRV DNS queries.

To work with DNS servers, Global Call requires the following information, which it will attempt to obtain from the operating system:

A list of DNS servers
: tells Global Call which DNS servers to work with. The maximum number of DNS servers is 20.

A list of domain suffixes
: tells Global Call which domain suffixes should be appended to FQDNs. (For example, “radvision.com” is the suffix for the “host1.radvision.com” host.) Using suffixes allows the use of a short version of a name that is within the suffix domain. The maximum number of domain suffixes is 20.

When working in Windows®, Global Call tries to fetch the DNS server addresses and suffixes from the registry. For DNS server addresses, it looks in HKEY_LOCAL_MACHINE\\SYSTEM \\CurrentControlSet\\Services\\Tcpip\\Parameters\\NameServer, and for DNS domain suffixes it looks in HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\Tcpip\\Parameters \\Domain.

When working in Linux, Global Call tries to fetch the DNS server addresses and domain suffixes from the file */etc/resolv.conf*.

12.4 Called and Calling Party Address List Format When Using H.323

This section provides reference information about called and calling party address list format:

- Called Party Address List
- Calling Party Address List
- Examples of Called and Calling Party Addresses

Called Party Address List

Called party address lists are formatted as follows:

```
Called Party Address list ::= Called Party Address |
    Called Party Address Delimiter Party Address list

Called Party Address ::= Dialable Address | Name |
    E164ALIAS | Extension | Subaddress | Transport
    Address | Email Address | URL | Party Number |
    Transport Name
```

where:

- Dialable Address ::= E164Address | E164Address ";" Dialable Address
- Name ::= "NAME:" H323ID
- E164ALIAS ::= "TEL:" E164Address
- Extension ::= "EXT:" E164Address | "EXTID : " H323ID
- Subaddress ::= "SUB:" E164Address
- Transport Address ::= "TA:" Transport Address Spec | "FTH : " Transport address Spec.
  - Transport Address Spec ::= Host Name":" Port Number | Host Name
    - Host Name ::= Host IP in decimal dotted notation.
- Email Address ::= "EMAIL :" email address
- URL Address ::= "URL : " URL
- PN Address ::= "PN :" party number ["$" party number type]

– Party Number Type ::= (select either the numerical or string value from the following list):
  - **0.PUU** - The numbering plan follows the E.163 and E.164Recommendations.
  - **PUI** - The number digits carry a prefix indicating type of number according to national recommendations.
  - **PUN** - The number digits carry a prefix indicating the type of number according to national recommendations.
  - **PUNS** - The number digits carry a prefix indicating the type of number according to network specifications.
  - **PUA** - Valid only for the called party number at the outgoing access; the network substitutes appropriate number.
  - **D** - Valid only for the called party number at the outgoing access; the network substitutes appropriate number.
  - **PRL2** - Level 2 regional subtype of private number.
  - **PRL1** - Level 1 regional subtype of private number.
  - **PRP** - PISN subtype of private number.
  - **PRL** - Local subtype of private number.
  - **PRA** - Abbreviated subtype of private number.
  - **N** - The number digits carry a prefix indicating standard type of number according to national recommendations.

- Transport Name ::= “TNAME :” Transport Address Spec

Notes: ***1.*** The delimiter is “,” by default, but it may be changed by setting the value of the delimiter field in the IPCCLIB_START_DATA used by the **gc_Start( )** function. See Section 8.3.27, “gc_Start( ) Variances for IP”, on page 590 for more information.

2. If the Dialable Address form of the address is used, it should be the last item in the list of address alternatives.

Calling Party Address List

Calling party address lists are formatted as follows:

```
Calling Party address list ::= Calling Party address |
    Calling Party address Delimiter |
    Calling Party address list

Calling Party address ::= Dialable Address | Name |
    E164ALIAS | Extension | Subaddress | Transport
    Address | Email Address | URL | Party Number |
    Transport Name
```

where the format options Dialable Address, Name, etc. are as described in the Called Party Address List section.

Note: If the Dialable Address form of the Party address is used, it should be the last item in the list of Party address alternatives.

Examples of Called and Calling Party Addresses

Some examples of called party and calling party addresses are:

- Called and Calling Party addresses: 1111;1111
- NAME: John, NAME: Jo
- TA:192.114.36.10

Glossary

alias: A nickname for a domain or host computer on the Internet.

blind transfer: See *unsupervised transfer.*

call transfer: See *supervised transfer* and *unsupervised transfer.*

codec: A device that converts analog voice signals to a digital form and vice versa. In this context, analog signals are converted into the payload of UDP packets for transmission over the internet. The codec also performs compression and decompression on a voice stream.

H.225.0: Specifies messages for call control including signaling, Registration Admission and Status (RAS), and the packetization and synchronization of media streams.

en-bloc mode: A mode where the setup message contains all the information required by the network to process the call, such as the called party address information.

H.245: H.245 is a standard that provides the call control mechanism that allows H.323-compatible terminals to connect to each other. H.245 provides a standard means for establishing audio and video connections. It specifies the signaling, flow control, and channeling for messages, requests, and commands. H.245 enables codec selection and capability negotiation within H.323. Bit rate, frame rate, picture format, and algorithm choices are some of the elements negotiated by H.245.

gateway: Translates communication procedures and formats between networks, for example the interface between an IP network and the circuit-switched network (PSTN).

Gatekeeper: Manages a collection of H.323 entities (terminals, gateway, multipoint control units) in an H.323 zone.

H.255.0: The H.255.0 standard defines a layer that formats the transmitted audio, video, data, and control streams for output to the network, and retrieves the corresponding streams from the network.

H.323: H.323 is an ITU recommendation for a standard for interoperability in audio, video and data transmissions as well as Internet phone and voice-over-IP (VoIP). H.323 addresses call control and management for both point-to-point and multipoint conferences as well as gateway administration of IP Media traffic, bandwidth and user participation.

IP: Internet Protocol

IP Media Library: Dialogic API library used to control RTP streams.

Multipoint Control Unit (MCU): An endpoint that support conferences between three or more endpoints.

prefix: One or several digits dialed in front of a phone number, usually to indicate something to the phone system. For example, dialing a zero in front of a long distance number in the United States indicates to the phone company that you want operator assistance on a call.

Q.931: The Q.931 protocol defines how each H.323 layer interacts with peer layers, so that participants can interoperate with agreed upon formats. The Q.931 protocol resides within H.225.0. As part of H.323 call control, Q.931 is a link layer protocol for establishing connections and framing data.

RTP: Real-time Transport Protocol. Provides end-to-end network transport functions suitable for applications transmitting real-time data such as audio, video or simulation data, over multicast or unicast network services. RTP does not address resource reservation and does not guarantee quality-of-service for real-time services.

RTCP: RTP Control Protocol (RTCP). Works in conjunction with RTP to allow the monitoring of data delivery in a manner scalable to large multicast networks, and to provide minimal control and identification functionality. RTCP is based on the periodic transmission of control packets to all participants in the session, using the same distribution mechanism as the data packets.

silence suppression: See Voice Activation Detection (VAD).

supervised transfer: A call transfer in which the person transferring the call stays on the line, announces the call, and consults with the party to whom the call is being transferred before the transfer is completed.

UA: In a SIP context, user agents (UAs) are appliances or applications, such as, SIP phones, residential gateways and software that initiate and receive calls over a SIP network.

SIP: Session Initiated Protocol. An ASCII-based, peer-to-peer protocol designed to provide telephony services over the Internet.

split call control: An IP telephony software architecture in which call control is done separately from IP Media stream control, for example, call control is done on the host and IP Media stream control is done on the board.

tunneling: The encapsulation of H.245 messages within Q.931/H.225 messages so that H.245 media control messages can be transmitted over the same TCP port as the Q.931/H.225 signaling messages.

unsupervised transfer: A transfer in which the call is transferred without any consultation or announcement by the person transferring the call.

VAD: Voice Activation Detection. In Voice over IP (VoIP), voice activation detection (VAD) is a technique that allows a data network carrying voice traffic over the Internet to detect the absence of audio and conserve bandwidth by preventing the transmission of *silent packets* over the network.

Index