Global Call IP over Host-based Stack Technology User's Guide

for Linux and Windows Operating Systems

05-1512-004

Publication Date: February 2003

Intel Converged Communications
1515 Route 10
Parsippany, NJ 07054

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

For **Products and Services Information**, visit the Intel Communications Systems Products website at:
http://www.intel.com/network/csp/

For **Sales Offices** and other contact information, visit the Intel Telecom Building Blocks Sales Offices page at:
http://www.intel.com/network/csp/sales/

Table of Contents

List of Figures

List of Tables

Preface

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

For information on porting an application developed using System Release 5.x and the embedded (on board) stack to the host-based stack implementation provided in System Release 6.0 and later, see the *Porting Global Call H.323 Applications from Embedded Stack to Host-Based Stack Application Note*.

Organization of this Guide

This guide contains the following chapters:

Chapter 1 provides a overview of VoIP technology and brief introductions to the H.323 and SIP standards for novice users.

Chapter 2 describes how Global Call can be used with IP technology and provides an overview of the architecture.

Chapter 3 describes the additional functionality of specific Global Call API functions used for IP technology.

Chapter 4 describes how to use Global Call to perform IP-specific tasks, such as setting call related information, registering with a registration server, etc.

Chapter 5 provides guidelines for building Global Call applications that use IP technology.

Chapter 6 provides a reference for IP-specific parameter set IDs and their associated parameter IDs.

Chapter 7 provides a data structure reference for Global Call IP-specific data structures.

Chapter 8 provides descriptions of IP-specific event cause codes.

Chapter 9 provides information for debugging Global Call IP applications.

Chapter 10 gives the formats for called and calling party address lists.

A Glossary and an Index can be found at the end of the document.

Global Call Product Support

The Global Call software provides a consistent interface across Intel® NetStructure™ and Intel® Dialogic® products that interface to various networks for example, T-1 ISDN, E-1 ISDN, E-1 CAS and T-1 robbed bit, SS7 and IP (using H.323 and SIP). See the *Release Guide* for your system software for the products that support Global Call and provide IP signaling capabilities.

1. VoIP Overview

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 Switch Telephone Network (PSTN). At the time of writing this document, there are two major international groups defining standards for VoIP:

- International Telecommunications Union (ITU) - The ITU has defined the following:
  - H.323 standard, which covers VoIP
  - H.248 standard, which covers the Megaco Protocol
- Internet Engineering Task Force (IETF) - The IETF has defined drafts of the following RFC (Request for Comment) documents:
  - RFC 3261, the Session Initiation Protocol (SIP)
  - RFC 27053, the Media Gateway Control Protocol (MGCP)

The H.323 standard was developed in the mid 1990s and is more mature than any of the protocols mentioned above.

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).

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 support 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.

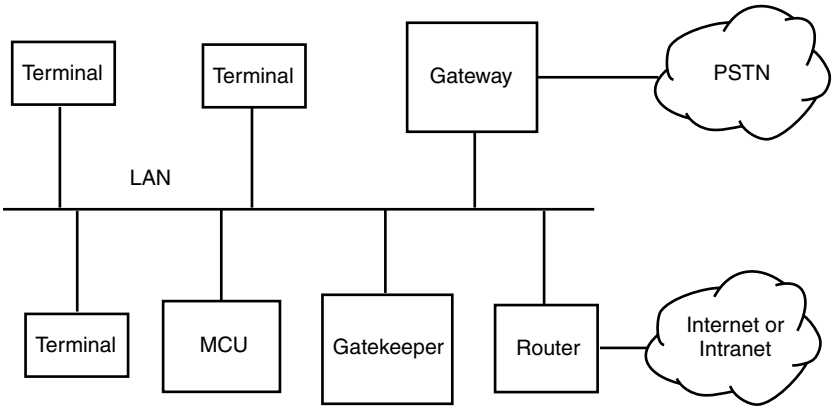


Figure 1. Typical H.323 Network

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.

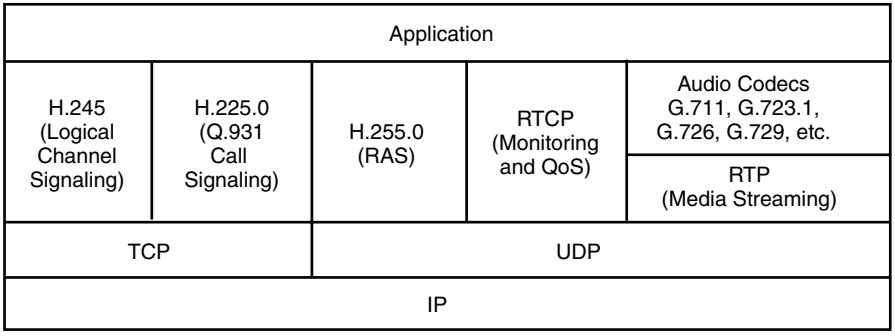


Figure 2. H.323 Protocol Stack

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, Registration Admission and Status (RAS), and the packetization and synchronization of media streams.
- **Real Time Protocol (RTP)** - The RTP specification (RFC 18890) is an IETF draft standard 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 provides 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 RTP specification (RFC 18890) 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 the 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 Kbits/s to 64 Kbits/s. 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 Kbits/s.
- **G.723.1** - Provides bit rates of either 5.3 or 6.4 Kbits/s. Voice communication using this codec typically exhibits some form of degradation.
- **G.729** - Provides a bit rate of 8 Kbits/s. 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 Kbits.s. This codec is based on a telephony standard defined by the European Telecommunications Standards Institute (ETSI). The 13 Kbits/s 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 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 requires two TCP connections between the endpoints:

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

NOTE: It is also possible to encapsulate H.245 media control messages within Q.931/H.225 signaling messages. The concept is known as H.245 **tunneling**. If tunneling is enabled, one less TCP port is required for incoming connections.

The scenario described in this section assumes a slow start connection procedure. See Section 4.2, "Using Fast Start and Slow Start Setup", on page 78 for more information on the difference between the slow start and fast start connection procedure.

The caller at endpoint A connects to the callee at endpoint B on a well-known port, 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 and 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.

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.

Both endpoint A and endpoint B then send 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 unregistration
- Endpoint location
- Admission request
- Bandwidth request
- Disengage request

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 and alias addresses. 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 5 is useful for explaining the fundamentals of an H.323 call, is not a realistic call scenario. The IP addresses of both endpoints were defined to be known. 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.

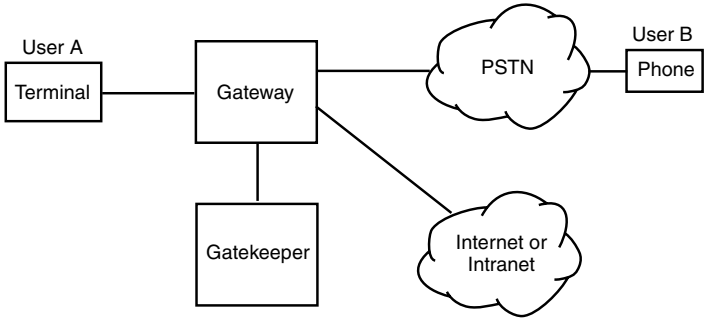


Figure 3. Basic H.323 Network with a Gateway

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 example, 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, can begin.

Call Signaling and Data Exchange

Endpoint A can now sends 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 gatekeeper 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 Section , “Capability Exchange”, on page 7, Section , “Call Initiation”, on page 7, and Section , “Data Exchange”, on page 8 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.

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 is 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.
- SIP is Internet-enabled.

1.3.1. 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.

1.3.2. 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.

1.3.3. Basic SIP Call Scenario

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

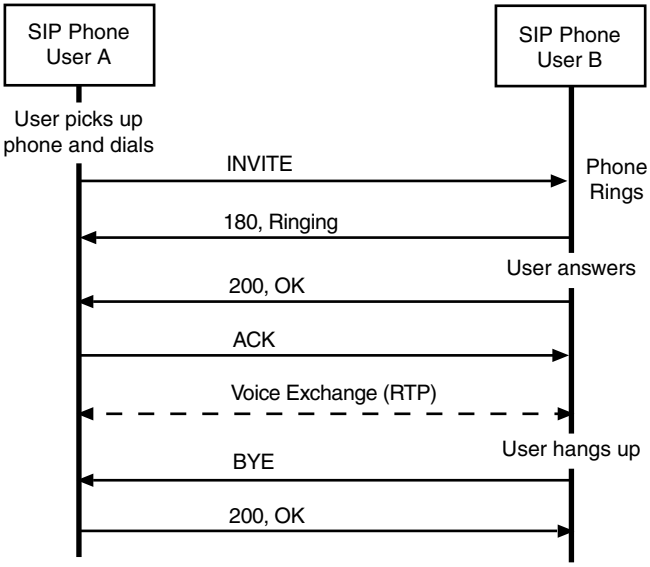


Figure 4. Basic SIP Call Scenario

1.3.4. 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 see RFC 3261 at
http://www.ietf.org/rfc/rfc3261.txt?number=3261.

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 see RFC 3261.

1.4. References

The following publications provide related information.

- ITU-T Recommendation H.323 (11/00) - *Packet-based multimedia communications systems*
- ITU-T Recommendation H.245 (07/01) - *Control protocol for multimedia communication*
- ITU-T Recommendation H.225.0 (09/99) - *Call signaling protocols and media stream packetization for packet-based multimedia communications systems*
- ITU-T Recommendation T.38 (06/98) - *Procedures for real-time Group 3 facsimile communication over IP networks*
- ITU-T Recommendation T.30 (07/96) - *Procedures for document facsimile transmission in the general switched telephone network*
- RFC 1889, RTP: *A Transport Protocol for Real-Time Applications*, IETF Publication, http://www.ietf.org/rfc/rfc1889.txt.

- RFC 3261, *Session Initiation Protocol (SIP)*, IETF Publication, draft reference http://www.ietf.org/rfc/rfc3261.txt?number=3261.
- Cisco Systems, *Signaled Digits in SIP*, draft reference http://www.ietf.org/internet-drafts/draft-mahy-sipping-signaled-digits-00.txt
- Black, Uyless (Copyright 2000), *Voice over IP*, Prentice Hall PTR, Prentice-Hall, Inc.
- Douskalis, Bill, *IP Telephony; The Integration of Robust VoIP Services*, Prentice Hall PTR, Prentice-Hall, Inc., ISBN 0-13-014118-6
- *Introduction to Voice Over the Internet Protocol*, Paolo Galtieri, Applied Computing Technologies, Winter 2000

2. Using Global Call with IP Technology

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

Global Call 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.

2.1. Global Call Over IP Architecture with a Host-Based Stack

Global Call supports a system configuration where the IP signaling stack (provided with the Intel® Dialogic® System Software) is running on the host and a DM3 board or Intel® NetStructure™ IPT board provides the IP resources for media processing.

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

Figure 5 shows the Global Call over IP architecture when using a DM3 board or an Intel® NetStructure IPT board and a host-based stack provided with the system software.

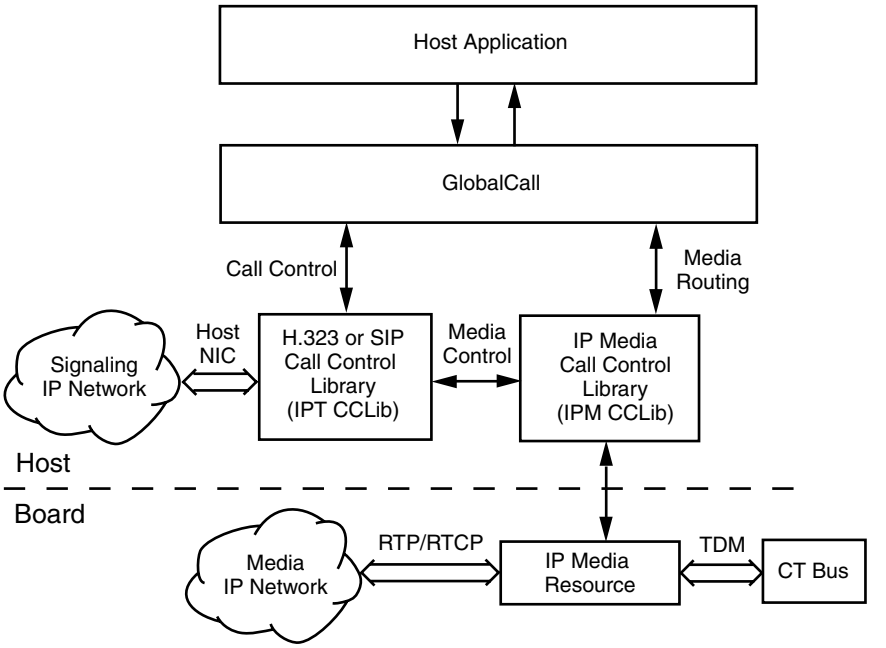


Figure 5. Global Call over IP Architecture Using a Host-Based Stack

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

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

2.1.1. Host Application

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

- Initializes Global Call
- Opens and closes IP line devices

- Opens and closes IP Media devices
- Opens and closes 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.

2.1.2. Global Call

Global Call 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.

2.1.3. IP Signaling Call Control Library (IPT CCLib)

The IP Signaling call control library (IPT CCLib) implements IP technology. It performs the following tasks:

- Controls the H.323 and/or SIP stack.
- Manages IP Media resources as required by the Global Call call state model and the IP signaling protocol model.
- Translates between the Global Call call model and IP signaling protocol model.
- Processes Global Call call control library interface commands.
- Generates call control library interface events.

2.1.4. IP Media Call Control Library (IPM CCLib)

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

- Processes Global Call call control library interface commands for the opening, closing, and timeslot routing of media devices.
- Configures QoS thresholds.
- Translates QoS alarms to Global Call alarm events.

2.1.5. 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.

2.2. Device Types and Usage

When using Global Call with IP technology, three types of devices 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 2.2.1, "IPT Board Devices", on page 24 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 2.2.2, "IPT Network Devices", on page 26 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 3.4.12, “gc_OpenEx( )”, on page 60 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 4.10, “Quality of Service Alarm Management”, on page 116 for more information.

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

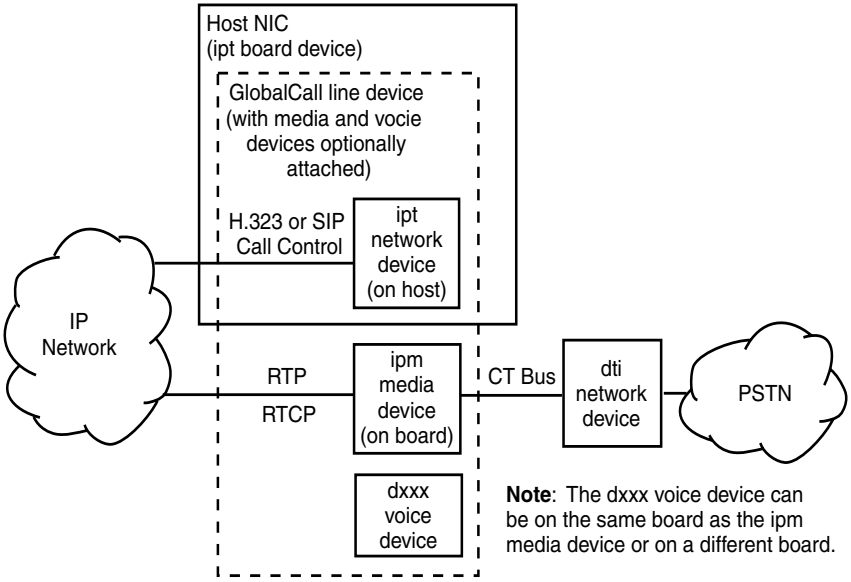


Figure 6. Global Call Devices

2.2.1. IPT Board Devices

An IPT board device is a virtual entity that corresponds to a NIC or NIC address and is capable of handling both H.323 and SIP protocols. The application uses the **gc_Start( )** function to bind NIC 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 Global Call 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 3.4.19, "gc_Start( )", on page 72 for more information on how to configure IPT board devices.

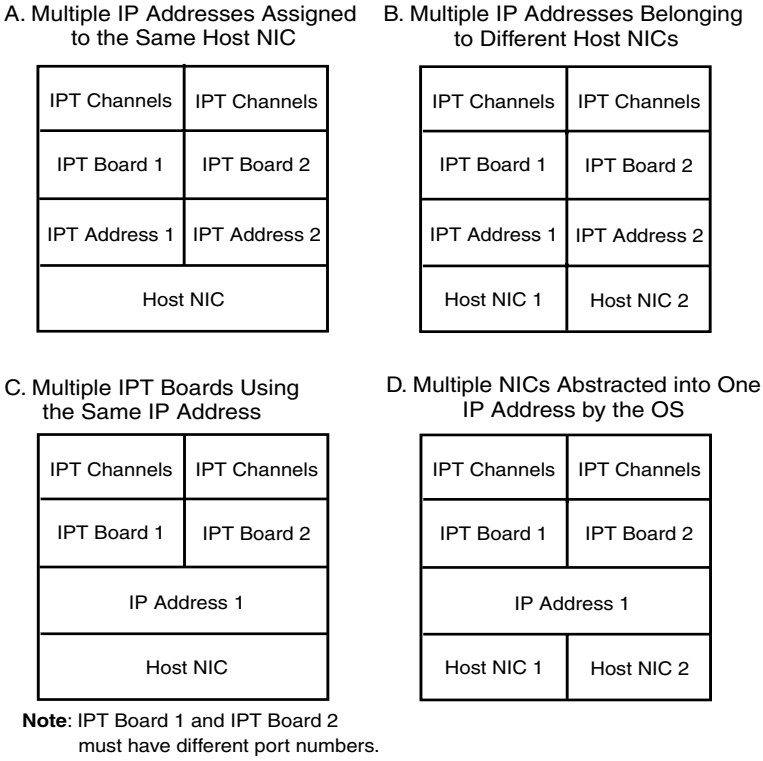


Figure 7. Configurations for Binding IPT Boards to NIC IP Addresses

Once the IPT board devices are configured, the application can open line devices with the appropriate IPT network device (ipt channel) and optionally IPT 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 IPT 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 3.4.14, "gc_ReqService( )", on page 62 for more information.

2.2.2. IPT Network Devices

Global Call 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.

2.2.3. IPT Start Parameters

The application determines the number of boards that will be created by the IPT call control library (up to the number of available IP addresses). For each 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
- Listen port for H.323
- Listen port for SIP

3. Applying Global Call Functions to IP Technology

Certain Global Call functions have additional functionality or perform differently when used with IP technology. The generic function descriptions in the *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 *Global Call API Library Reference* to obtain the complete information when developing Global Call applications that use IP technology.

3.1. Supported Global Call Functions

The following is a full list of the Global Call functions supported when using Global Call with IP technology:

NOTE: All functions are supported in asynchronous mode. **gc_OpenEx( )**, **gc_Listen( )**, **gc_ReleaseCallEx( )** and **gc_Unlisten( )** are also supported in synchronous mode.

- **gc_AcceptCall( )** - See Section 3.4.1, “gc_AcceptCall( )”, on page 34 for variances.
- **gc_AlarmName( )**
- **gc_AlarmNumber( )**
- **gc_AlarmNumberToName( )**
- **gc_AlarmSourceObjectID( )**
- **gc_AlarmSourceObjectIDToName( )**
- **gc_AlarmSourceObjectName( )**
- **gc_AlarmSourceObjectNameToID( )**
- **gc_AnswerCall( )** - See Section 3.4.2, “gc_AnswerCall( )”, on page 34 for variances.
- **gc_AttachResource( )**

- **gc_CallAck( )** - See Section 3.4.3, "gc_CallAck( )", on page 36 for variances.
- **gc_CCLibIDToName( )**
- **gc_CCLibNameToID( )**
- **gc_CCLibStatus( )**
- **gc_CCLibStatusAll( )**
- **gc_CCLibStatusEx( )**
- **gc_Close( )**
- **gc_CRN2LineDev( )**
- **gc_Detach( )**
- **gc_DropCall( )** - See Section 3.4.4, "gc_DropCall( )", on page 36.
- **gc_ErrorInfo( )**
- **gc_ErrorValue( )**
- **gc_Extension( )** - See Section 3.4.5, "gc_Extension( )", on page 36 for variances.
- **gc_GetAlarmConfiguration( )**
- **gc_GetAlarmFlow( )**
- **gc_GetAlarmParm( )** - See Section 3.4.6, "gc_GetAlarmParm( )", on page 38 for variances.
- **gc_GetAlarmSourceObjectList( )**
- **gc_GetAlarmSourceObjectNetworkID( )**
- **gc_GetCallInfo( )** - See Section 3.4.7, "gc_GetCallInfo( )", on page 39 for variances.
- **gc_GetCallState( )**
- **gc_GetCRN( )**
- **gc_GetLineDev( )**
- **gc_GetMetaEvent( )**
- **gc_GetMetaEventEx( )**
- **gc_GetResourceH( )** - See Section 3.4.8, "gc_GetResourceH( )", on page 41 for variances.
- **gc_GetUsrAttr( )**
- **gc_GetVer( )**

- **gc_GetXmitSlot( )** - See Section 3.4.9, “gc_GetXmitSlot( )”, on page 41 for variances.
- **gc_LinedevToCCLIBID( )**
- **gc_Listen( )** - See Section 3.4.10, “gc_Listen( )”, on page 42 for variances.
- **gc_MakeCall( )** - See Section 3.4.11, “gc_MakeCall( )”, on page 42 for variances.
- **gc_OpenEx( )** - See Section 3.4.12, “gc_OpenEx( )”, on page 60 for variances.
- **gc_ReleaseCallEx( )** - See Section 3.4.13, “gc_ReleaseCallEx( )”, on page 62 for variances.
- **gc_ReqService( )** - See Section 3.4.14, “gc_ReqService( )”, on page 62 for variances.
- **gc_ResetLineDev( )**
- **gc_RespService( )** - See Section 3.4.15, “gc_RespService( )”, on page 66 for variances.
- **gc_ResultInfo( )**
- **gc_SetAlarmConfiguration( )**
- **gc_SetAlarmFlow( )**
- **gc_SetAlarmNotifyAll( )**
- **gc_SetAlarmParm( )** - See Section 3.4.16, “gc_SetAlarmParm( )”, on page 67 for variances.
- **gc_SetConfigData( )** - See Section 3.4.17, “gc_SetConfigData( )”, on page 68 for variances.
- **gc_SetUserInfo( )** - See Section 3.4.18, “gc_SetUserInfo( )”, on page 71 for variances.
- **gc_SetUsrAttr( )**
- **gc_Start( )** - See Section 3.4.19, “gc_Start( )”, on page 72 for variances.
- **gc_Stop( )**
- **gc_UnListen( )** - See Section 3.4.20, “gc_UnListen( )”, on page 76 for variances.
- **gc_util_delete_parm_blk( )**
- **gc_util_find_parm( )**
- **gc_util_insert_parm_ref( )**
- **gc_util_insert_parm_val( )**

- **gc_util_next_parm( )**
- **gc_WaitCall( )**

See the *Global Call API Library Reference* for more information about Global Call functions.

3.2. Supported Global Call Call States

The following Global Call call states are supported when using Global Call with IP technology:

- GCST_ACCEPTED
- GCST_ALERTING
- GCST_CALLROUTING
- GCST_CONNECTED
- GCST_DETECTED
- GCST_DIALING
- GCST_DISCONNECTED
- GCST_IDLE
- GCST_NULL
- GCST_OFFERED
- GCST_PROCEEDING

See the *Global Call API Programming Guide* for more information about the call state models.

3.3. Supported Global Call Events

The following Global Call events are supported when using Global Call with IP technology:

- GCEV_ACCEPT
- GCEV_ACKCALL (deprecated; equivalent is GCEV_CALLPROC)
- GCEV_ALARM
- GCEV_ALERTING

- GCEV_ANSWERED
- GCEV_ATTACH
- GCEV_ATTACHFAIL
- GCEV_BLOCKED
- GCEV_CONNECTED
- GCEV_CALLPROC
- GCEV_DETECTED
- GCEV_DETACH
- GCEV_DETACHFAIL
- GCEV_DIALING
- GCEV_DISCONNECTED
- GCEV_DROPCALL
- GCEV_ERROR
- GCEV_EXTENSION (unsolicited event)
- GCEV_EXTENSIONCMPLT (termination event for **gc_Extension(**))
- GCEV_FATALERROR
- GCEV_LISTEN
- GCEV_OFFERED
- GCEV_OPENEX
- GCEV_OPENEX_FAIL
- GCEV_PROCEEDING
- GCEV_RELEASECALL
- GCEV_RESETLINEDEV
- GCEV_SERVICEREQ
- GCEV_SERVICERESP
- GCEV_SERVICERESPCMPLT
- GCEV_SETCONFIGDATA
- GCEV_SETCONFIGDATAFAIL
- GCEV_TASKFAIL
- GCEV_UNBLOCKED
- GCEV_UNLISTEN

See the *Global Call API Library Reference* for more information about Global Call events.

3.4. Function Variances

The following sections describe the details of using certain Global Call API functions in applications that use IP technology. See the *Global Call API Library Reference* for generic (technology-independent) descriptions of the Global Call API functions.

NOTE: All functions are supported in asynchronous mode. Functions that also support synchronous mode (**gc_OpenEx( )**, **gc_Listen( )**, **gc_ReleaseCallEx( )** and **gc_Unlisten( )**) are noted explicitly.

3.4.1. gc_AcceptCall( )

When using H.323, the **gc_AcceptCall( )** function is used to send the Q.931 ALERTING message to the originating endpoint.

When using SIP, the **gc_AcceptCall( )** function is used to send the 180 Ringing message to the originating endpoint.

Also, the **rings** parameter is ignored.

3.4.2. gc_AnswerCall( )

When using H.323, the **gc_AnswerCall( )** function is used to send the Q.931 CONNECT message to the originating endpoint.

When using SIP, the **gc_AnswerCall( )** function is used to send the 200 OK message to the originating endpoint.

Also, the **rings** parameter is ignored.

Coders can be set in advance of using **gc_AnswerCall( )** by using **gc_SetUserInfo( )**. See Section 3.4.18, "gc_SetUserInfo( )", on page 71 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 ... */
```

3.4.3. gc_CallAck( )

When using H.323, the **gc_CallAck( )** function is used to send the Proceeding message to the originating endpoint.

When using SIP, the **gc_CallAck( )** function is used to send the 100 Trying message to the originating endpoint.

The **callack_blkp** parameter must be a pointer to a GC_CALLACK_BLK structure that contains a **type** parameter 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 if the Proceeding message is sent manually using the **gc_CallAck( )** function or if it is sent automatically by the stack. See Section 4.15, "Configuring the Sending of the Proceeding Message", on page 141 for more information.

3.4.4. gc_DropCall( )

In addition to the drop call causes documented in the *Global Call API Library Reference*, the **cause** parameter can be any of the cause codes prefixed by IPEC_H225 or IPEC_Q931 (for H.323) or IPEC_SIP (for SIP) specified in Table 48, "IP-Specific Event Cause Codes", on page 189.

NOTE: When using SIP, cause codes and reasons are only supported when **gc_DropCall( )** is issued while the call is in the Offered state.

3.4.5. gc_Extension( )

The **gc_Extension( )** function can be used for the following purposes:

- Getting notification for T.38 fax events
- Retrieve 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
- Specifying which DTMF types, when detected, provide notification to the application
- Sending DTMF digits
- Retrieving protocol messages (Q.931, H.245 and registration)
- Sending protocol messages (Q.931, H.245 and registration)

Table 1 shows the valid extension IDs and their purpose.

Table 1. Valid Extension IDs for the gc_Extension( ) Function

Extension ID	Description
IPEXTID_FOIP	Used in GCEV_EXTENSION events for notification of information related to fax. See Section 4.14, “Enabling and Disabling Unsolicited Notification Events”, on page 140 for more information.
IPEXTID_GETINFO	Used to retrieve call-related information. See Section 4.5, “Retrieving Current Call-Related Information”, on page 92 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 4.14, “Enabling and Disabling Unsolicited Notification Events”, on page 140 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 4.14, “Enabling and Disabling Unsolicited Notification Events”, on page 140 for more information.
IPEXTID_RECEIVE_DTMF	Used to select which DTMF types, when detected, provide notification to the application. See Section 4.14, “Enabling and Disabling Unsolicited Notification Events”, on page 140 for more information.

Table 1. Valid Extension IDs for the gc_Extension( ) Function

Extension ID	Description
IPEXTID_RECEIVEMSG	Used in GCEV_EXTENSION events when 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_RECEIV_DTMF but only when configured for notification of a specific type of DTMF. See Section 4.14, "Enabling and Disabling Unsolicited Notification Events", on page 140 for more information.
IPEXTID_SENDMSG	Used to send Q.931, H.245 and RAS messages. The supported parameter sets are: • IPSET_MSG_H245 • IPSET_MSG_Q931 • IPSET_MSG_RAS When the gc_Extension( ) call 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 4.5.2, "Example of Retrieving Call-Related Information", on page 97 for a code example showing how to identify the type of extension event and extract the related information.

3.4.6. gc_GetAlarmParm( )

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 4.10.2, "Retrieving the Media Device Handle", on page 117 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 *IP Media Library API Library Reference* and the *IP Media Library API Programming Guide* for more information. The thresholds supported by Global Call are QOSTYPE_LOSTPACKETS and QOSTYPE_JITTER.
- **mode**: Must be set to EV_SYNC.

NOTE: Applications **must** include the *gcipmlib.h* header file before Global Call can be used to set or retrieve QoS threshold values.

See Section 4.10.3, “Setting QoS Threshold Values”, on page 118 for code examples.

3.4.7. gc_GetCallInfo( )

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, and E.164 number, a URL, etc. The supported values of the **info_id** parameter are:

- ORIGINATION_ADDRESS - the calling party information (equivalent to ANI)
- DESTINATION_ADDRESS - the called party information (equivalent to DNIS)

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 (configurable in the IPCCLIB_START_DATA data structure used by **gc_Start**( ); the default is a comma).

For H.323, 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.

For SIP, the address is taken from the From: header and is of the form user@host; prefixes (TA:, TEL; or NAME:) are **not** used.

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 (configurable in the IPCCLIB_START_DATA data structure used by **gc_Start( )**; the default is a comma). The IP address of the destination gateway (that is processing the DNIS) is **not** included in the string.

For H.323, 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.

For SIP, the address is taken from the To: header and is of the form user@host; prefixes (TA:, TEL:, or NAME:) are **not** used.

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.

3.4.8. gc_GetResourceH( )

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 Global Call 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 Global Call 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 Global Call 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.

3.4.9. gc_GetXmitSlot( )

The **gc_GetXmitSlot( )** function can be used to get the transmit time slot information for an IP Media device. The format of the **gc_GetXmitSlot( )** function in this context is:

gc_GetXmitSlot(linedev, sctsinfop)

where,

- **linedev** is the Global Call 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** is a pointer to the transmit time slot information for the IP Media device (a pointer to a CT Bus time slot information structure).

3.4.10. gc_Listen( )

The **gc_Listen**( ) function is supported in both asynchronous and synchronous 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).

3.4.11. gc_MakeCall( )

Global Call supports multiple IP protocols on a single device. See Section 2.2, "Device Types and Usage", on page 22 for more information. When using multi-protocol devices only, the protocol can be specified in the associated GC_MAKECALL_BLK structure. The set ID and parameter ID that should be included in the parameter block are:

- set_ID: IPSET_PROTOCOL
- parm_ID: IPPARM_PROTOCOL_BITMASK, with one of the following values:
  - IP_PROTOCOL_SIP
  - IP_PROTOCOL_H323

When making calls on devices that support multiple protocols, if the application does not explicitly specify a protocol in the makecall block, the default protocol is IP_PROTOCOL_H323. 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.

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. Table 2 shows the parameters that are configurable when using H.323 or SIP.

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. Table 2 describes the parameters that can be accessed through the ext_datap pointer.

NOTES: 1. In Table 2, the term "String" implies the normal definition of a character string which can contain letters, numbers, white space, and a null (for termination).

2. The parameter names in Table 2 are more closely associated with H.323 terminology. Corresponding SIP terminology is described in http://www.ietf.org/rfc/rfc3261.txt?number=3261.

Table 2. Configurable Call Parameters

Set ID	Parameter IDs	H.323/SIP
GCSET_CHAN_ CAPABILITY	• IPPARM_LOCAL_CAPABILITY **Datatype**: IP_CAPABILITY. See Section 7.4, "IP_CAPABILITY", on page 181 for more information. **Note**: If no transmit/receive coder type is specified, any supported coder type is accepted.	H.323 and SIP
IPSET_CALLINFO See Section 6.3, "IPSET_CALLINFO Parameter Set", on page 159 for more information.	• IPPARM_CONNECTIONMETHOD **Datatype**: Enumeration, with one of the following values: IPPARM_CONNECTIONMETHOD_ FASTSTART or IPPARM_CONNECTIONMETHOD_ SLOWSTART See Section 4.2, "Using Fast Start and Slow Start Setup", on page 78 for more information.	H.323 and SIP
	• IPPARM_DISPLAY **Datatype**: String, max. length = MAX_DISPLAY_LENGTH (82), null-terminated.	H.323 and SIP

Table 2. Configurable Call Parameters

Set ID	Parameter IDs	H.323/SIP
	• IPPARM_H245TUNNELING **Datatype**: Enumeration with one of the following values: IP_H245TUNNELING_ON or IP_H245TUNNELING_OFF. See Section 4.16, "Enabling and Disabling Tunneling in H.323", on page 142 for more information.	H.323 only
	• IPPARM_PHONELIST **Datatype**: String, max. length = 131.	H.323 and SIP
	• IPPARM_USERUSER_INFO **Datatype**: String, max. length = MAX_USERUSER_INFO_LENGTH (131 octets).	H.323 only
IPSET_CONFERENCE	IPPARM_CONFERENCE_GOAL **Datatype**: 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	H.323 only

Table 2. Configurable Call Parameters

Set ID	Parameter IDs	H.323/SIP
IPSET_NONSTANDARD DATA See Section 6.14, "IPSET_NONSTANDARD DATA Parameter Set", on page 167 for more information.	EITHER • IPPARM_NONSTANDARDDATA_DATA **Datatype**: String, max. length = MAX_NS_PARM_DATA_LENGTH (128). AND • IPPARM_NONSTANDARDDATA_OBJID **Datatype**: Unsigned Int[ ], max. length = MAX_NS_PARM_OBJID_LENGTH (40). OR • IPPARM_NONSTANDARDDATA_DATA **Datatype**: String, max. length = MAX_NS_PARM_DATA_LENGTH (128). AND • IPPARM_H221NONSTANDARD **Datatype**: IP_H221NONSTANDARD	H.323 only
IPSET_NONSTANDARD CONTROL See Section 6.15, "IPSET_NONSTANDARD CONTROL Parameter Set", on page 168 for more information.	EITHER • IPPARM_NONSTANDARDDATA_DATA **Datatype**: String, max. length = MAX_NS_PARM_DATA_LENGTH (128). AND • IPPARM_NONSTANDARDDATA_OBJID **Datatype**: Unsigned Int[ ], max. length = MAX_NS_PARM_OBJID_LENGTH (40). OR • IPPARM_NONSTANDARDDATA_DATA **Datatype**: String, max. length = MAX_NS_PARM_DATA_LENGTH (128). AND • IPPARM_H221NONSTANDARD **Datatype**: IP_H221NONSTANDARD	H.323 only

Forming a Destination Address String for an H.323 Call

The destination address is formed by concatenating values from three different sources: the GC_MAKECALL_BLK, the **numberstr** parameter of **gc_MakeCall(**), and 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. Other 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 - email 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. See the Destination Address Interpretation for H.323 and Forming a Destination Address String for a SIP Call sections for more information.
3. Phone list as described in Table 2, "Configurable Call Parameters", on page 43 (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. See for more information. See the Destination Address

Interpretation for H.323 and Forming a Destination Address String for a SIP Call sections for more information.

Destination Address Interpretation for H.323

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), if the application wants to prefix buffers, 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

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.intel.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);
```

Forming a Destination Address String for a SIP Call

The format of the destination address for a SIP call is:

```
user@host; param=value
```

where,

- **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 2, “Configurable Call Parameters”, on page 43 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 API’s 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);
```

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 *Global Call API Library Reference* for more information.

Specifying a Timeout

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.

When using H.323, two other parameters that affect the timeout behavior, but are not configurable are:

- **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.

Code Example for H.323

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(&target_datap,
                        IPSET_NONSTANDARDDATA,
                        IPPARM_NONSTANDARDDATA_DATA,
                        (unsigned char)(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(&target_datap,
                           IPSET_NONSTANDARDCONTROL,
                           IPPARM_NONSTANDARDDATA_DATA,
                           (unsigned char)(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);
   }
}
```

Code Example for SIP

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);
   }
}
```

3.4.12. gc_OpenEx( )

The **gc_OpenEx( )** function is supported in both asynchronous and synchronous mode. Using the function is 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 Global Call functions use the IPT board device, such as **gc_SetConfigData( )** to specify call parameters (such as coders) for all devices in one operation or **gc_ReqService( )** to perform registration and deregistration operations. See Section 3.4.17, "gc_SetConfigData( )", on page 68 and Section 3.4.14, "gc_ReqService( )", on page 62 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 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.

In other technologies, an IPT board device can be used for alarms. However, for IP technology, the use of an IPT board device (iptBx) for alarms is not supported. When using Global Call with IP, alarms are reported on IP Media (ipm) devices, not IPT network (ipt) devices.

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.

3.4.13. gc_ReleaseCallEx( )

The **gc_ReleaseCallEx( )** function is supported in both asynchronous and synchronous mode. Using the function in asynchronous mode is recommended.

NOTE: An existing call on a line device must be released before an incoming call can be processed.

3.4.14. gc_ReqService( )

The **gc_ReqService( )** function can be used to register an endpoint with a registration server (gateway in H.323 or registrar in SIP). Function parameter 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** - Set to NULL for asnychronous mode. This function is not supported in synchronous mode.
- **mode** - EV_ASYNC

The registration information that can be included in the associated GC_PARM_BLK is shown in Table 3.

Table 3. Registration Information

Set ID	Parameter IDs	H.323/SIP
GCSET_SERVREQ	PARM_REQTYPE † **Datatype**: IP_REQTYPE_REGISTRATION	H.323 and SIP
GCSET_SERVREQ	PARM_ACK †	H.323 and SIP
IPSET_PROTOCOL	IPPARM_PROTOCOL_BITMASK **Datatype**: Bitmask composed from the following values: • IP_PROTOCOL_H323 • IP_PROTOCOL_SIP	H.323 and SIP
IPSET_REG_INFO See Section 6.17, "IPSET_REG_INFO Parameter Set", on page 169, for more information.	IPPARM_OPERATION_REGISTER **Datatype**: One of the following values: • IP_REG_SET_INFO • IP_REG_ADD_INFO • IP_REG_DELETE_BY_VALUE IPPARM_OPERATION_DEREGISTER **Datatype**: One of the following values: • IP_REG_MAINTAIN_LOCAL_INFO • IP_REG_DELETE_ALL IPPARM_REG_ADDRESS **Datatype**: IP_REGISTER_ADDRESS See Section 7.9, "IP_REGISTER_ADDRESS", on page 185, for more information	H.323 and SIP

† indicates mandatory parameters. These parameters are required to support the generic service request mechanism provided by Global Call and are not sent in any registration message.

Table 3. Registration Information

Set ID	Parameter IDs	H.323/SIP
IPSET_LOCAL_ALIAS	IPPARM_ADDRESS_DOT_NOTATION IPPARM_ADDRESS_EMAIL IPPARM_ADDRESS_H323_ID IPPARM_ADDRESS_PHONE IPPARM_ADDRESS_TRANSPARENT IPPARM_ADDRESS_URL **Datatype**: String	H.323
IPSET_SUPPORTED_ PREFIXES	IPPARM_ADDRESS_DOT_NOTATION IPPARM_ADDRESS_EMAIL IPPARM_ADDRESS_H323_ID IPPARM_ADDRESS_PHONE IPPARM_ADDRESS_TRANSPARENT IPPARM_ADDRESS_URL **Datatype**: String	H.323
† indicates mandatory parameters. These parameters are required to support the generic service request mechanism provided by Global Call and are not sent in any registration message.		

Registration options include:

- Overriding an existing registration value (IPPARM_OPERATION_REGISTER = IP_REG_SET_INFO)
- Adding a registration value (IPPARM_OPERATION_REGISTER = IP_REG_ADD_INFO)
- Removing a registration value; local alias or supported prefix only (IPPARM_OPERATION_REGISTER = IP_REG_DELETE_BY_VALUE)

See Section 4.12.4, "Registration Code Example", on page 129 for more information.

The **gc_ReqService( )** function also provides the following deregister options:

- Deregister and keep the registration information locally (IPPARM_OPERATION_DEREGISTER = IP_REG_MAINTAIN_LOCAL_INFO)
- Deregister and discard the registration information locally (IPPARM_OPERATION_DEREGISTER = IP_REG_DELETE_ALL)

See Section 4.12.5, “Deregistration Code Example”, on page 132 for more information.

Alias and supported prefix information is supported when the target protocol for registration is H.323 only. The application should not specify this type of information when the target protocol includes SIP.

When using SIP, periodic registration is not supported. A time_to_live value can be specified in the IP_REGISTER_ADDRESS structure and an expired header is added. Registration is not refreshed. The application can re-register periodically to refresh registration before the time_to_live value times out. One-time registration is supported.

Since some of the registration data may be protocol specific, there is a facility to set the protocol type using IP parameters in **reqdatap** and **respdatapp**, which are of type GC_PARM_BLK.

The following are the relevant parameters for the GC_PARM_BLK:

- set_ID: IPSET_PROTOCOL
- parm_ID: IPPARM_PROTOCOL_BITMASK, which should be one of the following 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_SERVICERESP event indicates that a service has been responded to 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. The **sr_getevtdatap( )** function returns a pointer to a GC_PARM_BLK that includes the following parameter information:

- set_ID - IPSET_PROTOCOL
- parm_ID - IPPARM_PROTOCOL_BITMASK, with one of the following values:
  - IP_PROTOCOL_H323
  - IP_PROTOCOL_SIP

3.4.15. gc_RespService( )

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. Since some of the data may be protocol specific (in future releases), there is a facility to set the protocol type using IP parameters in **datap**, which is of type GC_PARM_BLK.

The following are the relevant function parameters:

- **target_type** - GCTGT_CCLIB_NETIF
- **target_id** - IPT board device

The following are the relevant parameters for the GC_PARM_BLK:

- set_id: IPSET_PROTOCOL
- parm_id: IPPARM_PROTOCOL_BITMASK, which can be have one of the following 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. The **sr_getevtdatap( )** function returns a pointer to a GC_PARM_BLK that includes the following parameter information:

- set_ID - IPSET_PROTOCOL
- parm_ID - IPPARM_PROTOCOL_BITMASK, with one of the following values:
  - IP_PROTOCOL_H323
  - IP_PROTOCOL_SIP

3.4.16. gc_SetAlarmParm( )

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 4.10.2, “Retrieving the Media Device Handle”, on page 117 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 *IP Media Library API Library Reference* and the *IP Media Library API Programming Guide* for more information. The thresholds supported by Global Call are QOSTYPE_LOSTPACKETS and QOSTYPE_JITTER.
- **mode**: Must be set to EV_SYNC.

NOTE: Applications **must** include the *gcipmlib.h* header file before Global Call can be used to set or retrieve QoS threshold values.

See Section 4.10.3, "Setting QoS Threshold Values", on page 118 for code examples.

3.4.17. gc_SetConfigData( )

The **gc_SetConfigData( )** function is used for a number of different purposes:

- Setting parameters for all devices on a board, 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
- Masking call state events on a line device basis

NOTES: 1. The **gc_SetConfigData( )** operates on a board device, that is, a device 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 can **not** 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.

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 3.4.12, "gc_OpenEx( )", on page 60 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 all the parameters in Table 2, "Configurable Call Parameters", on page 43. In additon, Table 4 shows additional parameters that are only configurable using **gc_SetConfigData( )**.

Table 4. Parameters Configurable Using gc_SetConfigData( )

Set ID	Parameter IDs	Use Before†	H.323/ SIP
GCSET_CALL_CONFIG	GCPARM_CALLPROC †† **Datatype**: 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( )	H.323 and SIP
IPSET_CALLINFO	IPPARM_H245TUNNELING ††† **Datatype**: Enumeration with one of the following values: • IP_H245TUNNELINGON • IP_H245TUNNELINGOFF	gc_AnswerCall( )	H.323 only
IPSET_DTMF	IPPARM_SUPPORT_DTMF_ BITMASK **Datatype**: Uint8[ ] IPPARM_DTMF_RFC2833_ PAYLOAD_TYPE **Datatype**: Uint8[ ]	gc_AnswerCall( ) gc_MakeCall( )	H.323 and SIP

† Information can be set in any state but it is only used in certain states. See Section 6, "IP-Specific Parameter Set Reference", on page 147 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 4.16, "Enabling and Disabling Tunneling in H.323", on page 142 for more information.

Table 4. Parameters Configurable Using gc_SetConfigData( )

Set ID	Parameter IDs	Use Before†	H.323/ SIP
IPSET_VENDORINFO	IPPARM_VENDOR_PRODUCT_ID **Datatype**: String, max. length = MAX_PRODUCT_ID_LENGTH (32). IPPARM_VENDOR_VERSION_ID **Datatype**: String, max. length = MAX_VERSION_ID_LENGTH (32). IPPARM_H221NONSTD **Datatype**: IP_H221NONSTANDARD.	gc_AnswerCall( ) gc_MakeCall( )	H.323 only
IPSET_ EXTENSIONEVT_MSK	GCACT_ADDMSK **Datatype**: Uint8[ ] GCACT_SETMSK **Datatype**: Uint8[ ] GCACT_SUBMSK **Datatype**: Uint8[ ]	gc_AnswerCall( )	H.323 and SIP

† Information can be set in any state but it is only used in certain states. See Section 6, "IP-Specific Parameter Set Reference", on page 147 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 4.16, "Enabling and Disabling Tunneling in H.323", on page 142 for more information.

As in other technologies supported by Global Call, 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 *Global Call API Programming Guide* for more information on masking events in general.

3.4.18. gc_SetUserInfo( )

The **gc_SetUserInfo( )** function can be used to:

- Set call values for the duration of a single call
- Set call values for all calls on the specified line device

The **gc_SetUserInfo( )** function is used only 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 Global Call function, such as, **gc_AnswerCall( )**, **gc_AcceptCall( )**, **gc_CallAck( )** etc. that initiates the transmission of information on the line.

NOTE: If no receive coder type is specified, any supported coder type is accepted.

The parameters that are configurable using **gc_SetUserInfo( )** are given in Table 2, "Configurable Call Parameters", on page 43. In addition, the DTMF support bitmask, see Table 4, "Parameters Configurable Using gc_SetConfigData( )", on page 69, is also configurable using **gc_SetUserInfo( )**.

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 (:N_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 (:N_H323 or :N_SIP), then the information in the function automatically applies to that protocol only and is used by calls made using that protocol.

Setting Call Parameters for the Next Call

The pertinent 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.

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.

3.4.19. gc_Start( )

The **gc_Start( )** function is used to specify the number of IPT boards to create. See Section 2.2.1, "IPT Board Devices", on page 24 for the meaning of an IPT board device. The number of IPT board devices is specified in the IPCCLIB_START_DATA structure; the maximum is 8. See Section 7.2, "IPCCLIB_START_DATA", on page 180 for more information.

NOTE: When using the SIP protocol, multiple IPT boards are **not** currently supported.

The IPCCLIB_START_DATA structure contains a pointer to an IP_VIRTBOARD structure for each IPT board device that contains board information such as:

- Total number of IPT devices that can be open concurrently

- Maximum number of IPT devices to be used for H.323 calls
- H.323 local address and signaling port
- Maximum number of IPT devices to be used for SIP calls
- SIP local address and signaling port

See Section 7.11, “IP_VIRTBOARD”, on page 187 for more information.

The total number of IPT devices is not necessarily the number of IPT devices used for H.323 calls plus the number of IPT devices used for SIP calls. Each IPT device can be used for both H.323 and SIP. If there are 2016 devices available (total_max_calls=2016, three Intel® NetStructure™ IPT boards), you can specify that all 2016 devices can be used for both H.323 calls (max_h323=2016) and SIP (max_sip=2016), or half are used for H.323 only (max_h323=1008) and half are used for SIP only (max_sip=1008), or any other such combination.

The default value for the maximum number of IPT devices is 120, but this can be set to a value up to 2016. See Section 7.11, “IP_VIRTBOARD”, on page 187 for more information. The local IP address for each IPT board device is a parameter of type IPADDR in the IP_VIRTBOARD structure. See Section 7.1, “IPADDR”, on page 179 for more information.

NOTES:

1. When using Intel® NetStructure™ IPT boards that have higher numbers of IP resources, it is important to remember to change the default maximum number of IPT devices (120) to take advantage of the larger number of IP resources.
2. The GC_LIB_START structure must include both the GC_H3R_LIB and GC_IPM_LIB libraries since there are inter-dependencies.
3. The maximum value of the num_boards field is 8.
4. When using the H.323 stack, the maximum number of simultaneous calls recommended is 240.

The following code example shows how to create and populate an IPCCLIB_START_DATA structure when using the **gc_Start( )** function.

```
/* only 1 board is currently supported */
#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);

cclibStartData.version     = 0x0100;   // must be set to 0x0100
cclibStartData.delimiter   = ',';
cclibStartData.num_boards  = BOARDS_NUM;
cclibStartData.board_list  = virtBoards;

virtBoards[0].version = 0x0100;
virtBoards[0].total_max_calls = IP_CFG_MAX_AVAILABLE_CALLS;
virtBoards[0].h323_max_calls = IP_CFG_MAX_AVAILABLE_CALLS;
virtBoards[0].sip_max_calls = IP_CFG_MAX_AVAILABLE_CALLS;
virtBoards[0].localIP.ip_ver = IPVER4;    // must be set to IPVER4
virtBoards[0].localIP.u_ipaddr.ipv4 = IP_CFG_DEFAULT;
virtBoards[0].h323_signaling_port = IP_CFG_DEFAULT;
virtBoards[0].sip_signaling_port = IP_CFG_DEFAULT;
virtBoards[0].reserved = NULL;            // must be set to NULL

CCLIB_START_STRUCT cclibStartStruct[] = {
    {"GC_IPM_LIB", NULL},
    {"GC_H3R_LIB", &cclibStartData}
};

GC_START_STRUCT gcStartStruct;
gcStartStruct.cclib_list = cclibStartStruct;
gcStartStruct.num_cclibs = 2;
int rc = gc_Start(&gcStartStruct);
if(GC_SUCCESS != rc)
{
    // handle the error
}
```

Some variations on the code above are as follows:

```
/* open 120 IPT devices, 120 H323 calls, 120 SIP calls */
    virtBoards[0].total_max_calls = IP_CFG_DEFAULT;
    virtBoards[0].h323_max_calls = IP_CFG_DEFAULT;
    virtBoards[0].sip_max_calls = IP_CFG_DEFAULT;

/* open 2016 IPT devices, 2016 H323 calls, 2016 SIP calls */
    virtBoards[0].total_max_calls = 2016;
    virtBoards[0].h323_max_calls = 2016;
    virtBoards[0].sip_max_calls = 2016;
```

```
/* open 2016 IPT devices, 2016 H323 calls, no SIP calls */
    virtBoards[0].total_max_calls = 2016;
    virtBoards[0].h323_max_calls = IP_CFG_MAX_AVAILABLE_CALLS;
    virtBoards[0].sip_max_calls = IP_CFG_NO_CALLS;

/* open 2016 IPT devices, 1008 H323 calls, 1008 SIP calls */
    virtBoards[0].total_max_calls = 2016;
    virtBoards[0].h323_max_calls = 1008;
    virtBoards[0].sip_max_calls = 1008;
```

The following #defines have been provided as a convenience to the application developer:

- 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.
- IP_CFG_NO_CALLS indicates to the call control library that it should **not** allocate any resources.

The following restrictions apply when populating the IPCCLIB_START_DATA structure. 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 (h323_max_calls) and the maximum number of SIP 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.

3.4.20. gc_UnListen( )

The **gc_UnListen( )** function is supported in both asynchronous and synchronous 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).

4. Using Global Call for IP-Specific Tasks

This chapter describes how to use Global Call to perform certain tasks in an IP environment. These tasks include:

- Call Control Configuration
- Using Fast Start and Slow Start Setup
- Basic Call Control Scenarios When Using Global Call
- Setting Call-Related Information
- Retrieving Current Call-Related Information
- Handling DTMF
- Getting Media Streaming Status and Negotiated Coder Information
- Getting Notification of Underlying Protocol State Changes
- Sending Protocol Messages
- Quality of Service Alarm Management
- Specifying RTP Stream Establishment
- Sending and Receiving Faxes over IP
- Registration
- Enabling and Disabling Unsolicited Notification Events
- Configuring the Sending of the Proceeding Message
- Enabling and Disabling Tunneling in H.323
- Using Object Identifiers

Each task is described in more detail in the following sections.

4.1. Call Control Configuration

When starting Global Call, certain configuration parameters such as, the maximum number of IPT devices available, the local IP address, and the call signaling port are configurable using an IPCCLIB_START_DATA structure associated with the **gc_Start( )** function. The default maximum number of IPT devices is 120, but it is

configurable up to a maximum of 2016 depending on the values of other configuration parameters.

NOTE: When using an Intel® NetStructure™ IPT board, the IPCCLIB_START_DATA structure **must** be used to take advantage of the higher numbers of IPT devices available on the board (up to 672).

To change the default values, create a structure of type IPCCLIB_START_DATA, populate it with new desired configuration values, and use it with the **gc_Start( )** function. See Section 7.2, "IPCCLIB_START_DATA", on page 180 for more information about the data structure and see Section 3.4.19, "gc_Start( )", on page 72 for a code example that shows how to populate the IPCCLIB_START_DATA structure.

NOTES:
1. In the IPCCLIB_START_DATA structure, the maximum value of the num_boards field, which defines the number of NICs or NIC addresses, is 8.
2. When using the SIP protocol, multiple IPT boards are **not** currently supported.

4.2. Using Fast Start and Slow Start Setup

Fast start and slow start are supported in both the H.323 and SIP protocols.

In H.323, fast start and slow start setup are supported depending on the version of H.323 standard supported at the remote side. If the remote side supports H.323 version 2 and above, fast start setup can be used, otherwise a slow start setup is used. Fast start connection reduces the time required to set up a call to one round-trip delay following the H.225 TCP connection. The concept is to include all the necessary parameters for the logical channel to be opened (H.245 information) in the Setup message. The logical channel information represents a set of supported capabilities from which the remote end can choose the most appropriate capability. If the remote side decides to use Fast start connection, it returns the logical channel parameters it wants to use in the Alerting, Proceeding, or Connect messages. Fast start connection is preferable to slow start connection because fewer network round trips are required to set up a call and the local exchange can generate messages when circumstances prevent a connection to the endpoint.

In SIP, fast and slow start setup is also supported. In slow start setup, the INVITE message will have no Session Description Protocol (SDP) and therefore the remote side will propose the message set 100, 180, 200, and the call is answered with the SDP in the ACK message.

In Global Call, fast start and slow start connection are supported on a call-by-call basis. Fast start connection is used by default, but slow start connection can be forced by including the IPPARM_CONNECTIONMETHOD parameter with a value of IP_CONNECTIONMETHOD_SLOWSTART in the ext_datap field (of type GC_PARM_BLK) in the GCLIB_MAKECALL_BLK structure associated with a call.

The following code segment shows how to specify a slow start connection explicitly by including the IPPARM_CONNECTIONMETHOD parameter when populating the ext_datap field (of type GC_PARM_BLK):

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

In addition, the IPPARM_CONNECTIONMETHOD parameter can be set to a value of IP_CONNECTIONMETHOD_FASTSTART to force a fast start connection on a line device configured to use a slow start connection (using **gc_SetUserInfo( )** with a **duration** parameter of GC_ALLCALLS).

NOTE: In SIP, only the calling side can choose fast start or slow start, unlike H.323 where both sides can select either fast start or slow start.

4.3. Basic Call Control Scenarios When Using Global Call

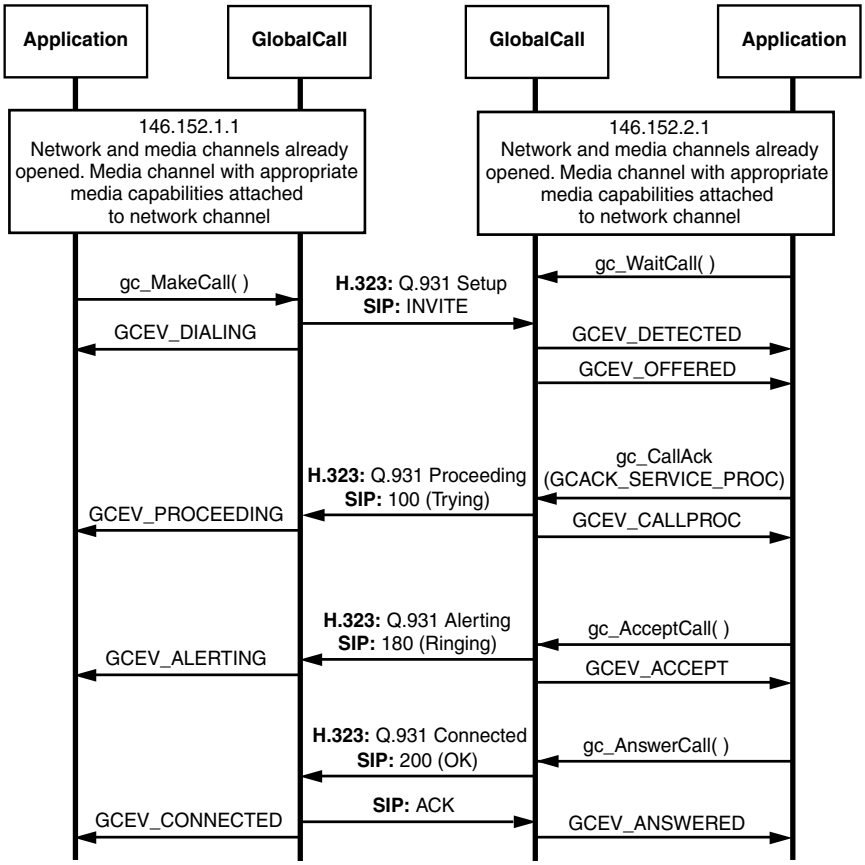
The call control scenarios described in this section include:

- Basic Call Setup When Using H.323 or SIP
- Basic Call Teardown When Using H.323 or SIP

4.3.1. Basic Call Setup When Using H.323 or SIP

Figure 8 shows the basic call setup scenario when using Global Call with H.323 or SIP. See Section 3.4.11, “gc_MakeCall( )”, on page 42 for detailed information on the specification of the destination address etc.

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



Note:
Media stream establishment is dependent on whether fast start or slow start connection is used: If fast start is selected, media streaming can be established after capability exchange in the Setup, Alerting or Connected messages in H.323 or the INVITE, Ringing, or OK messages in SIP. If slow start is selected, media streaming can only be established after the Connected message in H.323 or the OK message in SIP.

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

4.3.2. Basic Call Teardown When Using H.323 or SIP

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

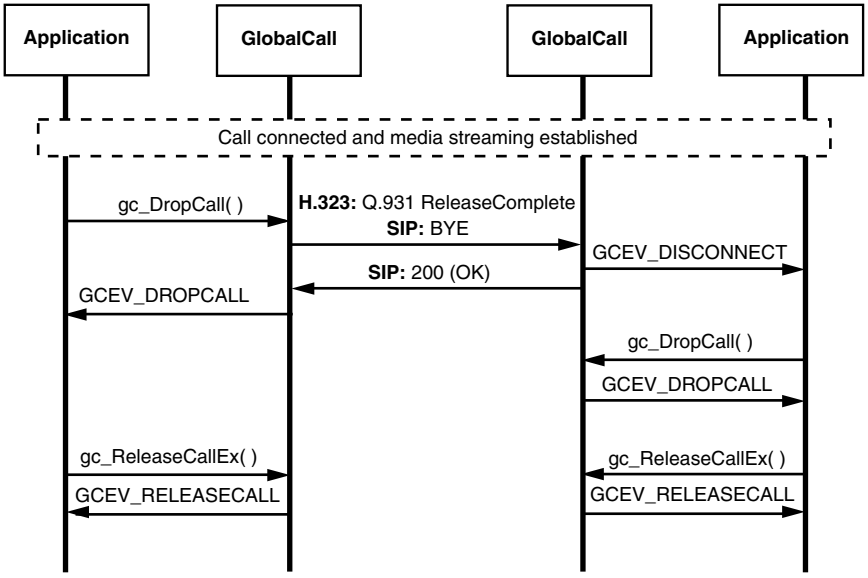


Figure 9. Basic Call Teardown When Using H.323 or SIP

4.4. Setting Call-Related Information

Table 5 summarizes the types of information elements that can be specified, the corresponding set IDs and parameter IDs used to set the information and 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.

Table 5. Summary of Call-Related Information that can be Set

Type of Information	Set ID and Parameter IDs	Functions Used to Set Information	H.323 /SIP
Coder Information	GCSET_CHAN_CAPABILITY • IPPARM_LOCAL_CAPABILITY	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	H.323 and SIP
Conference Goal	IPSET_CONFERENCE • IPPARM_CONFERENCE_GOAL	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	H.323 only
Connection Method	IPSET_CALLINFO • IPPARM_CONNECTIONMETHOD	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	H.323 and SIP
DTMF Support	IPSET_DTMF • IPPARM_SUPPORT_DTMF_ BITMASK	gc_SetConfigData( ) gc_SetUserInfo( )†	H.323 and SIP
Display Information	IPSET_CALLINFO • IPPARM_DISPLAY	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	H.323 and SIP
Enabling/Disabling Unsolicited Events	IPSET_EXTENSIONEVT_MSK • GCACT_ADDMSK • GCACT_SETMSK • GCACT_SUBMSK	gc_SetConfigData( )	H.323 and SIP

Table 5. Summary of Call-Related Information that can be Set

Type of Information	Set ID and Parameter IDs	Functions Used to Set Information	H.323 /SIP
Nonstandard Control Information	Either IPSET_NONSTANDARDCONTROL • IPPARM_NONSTANDARDDATA_DATA AND • IPPARM_NONSTANDARDDATA_OBJID or • IPPARM_NONSTANDARDDATA_DATA AND • IPPARM_H221NONSTANDARD	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	H.323 only
Nonstandard Data	Either IPSET_NONSTANDARDDATA • IPPARM_NONSTANDARDDATA_DATA • IPPARM_NONSTANDARDDATA_OBJID or • IPPARM_NONSTANDARDDATA_DATA • IPPARM_H221NONSTANDARD	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	H.323 only
Phone List	IPSET_CALLINFO • IPPARM_PHONELIST	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	H.323 and SIP
Tunnelling††	IPSET_CALLINFO • IPPARM_H245TUNNELING	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	H.323 only
Type of Service (ToS)	IPSET_CONFIG • IPPARM_CONFIG_TOS	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	H.323 only

Table 5. Summary of Call-Related Information that can be Set

Type of Information	Set ID and Parameter IDs	Functions Used to Set Information	H.323 /SIP
User to User Information	IPSET_CALLINFO • IPPARM_USERUSER_INFO	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	H.323 only
Vendor Information	IPSET_VENDORINFO • IPPARM_H221NONSTD • IPPARM_VENDOR_PRODUCT_ID • IPPARM_VENDOR_VERSION_ID	gc_SetConfigData( )	H.323 only

† The **duration** parameter can be set to GC_SINGLECALL (to apply on a call basis) or to GC_ALLCALLS (to apply on a live device basis).

†† On the terminating side, can only be set using gc_SetConfigData( ) on a board device. See Section 4.16, "Enabling and Disabling Tunneling in H.323", on page 142 for more information.

NOTE: 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.

4.4.1. Setting Call Parameters on a System-Wide Basis

Use the **gc_SetConfigData( )** function to configure call-related parameters including coder information. The values set by the **gc_SetConfigData( )** function are used by the call control library as default values for each line device.

See Section 3.4.17, "gc_SetConfigData( )", on page 68 for more information about the values of function parameters to set in this context.

4.4.2. 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. See Section 3.4.18, “gc_SetUserInfo( )”, on page 71 for more information about the values of function parameters to set in this context.

4.4.3. 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( )**

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. At the end of the call, the values set as default values for the specified line device override these values. This is useful since the **gc_AnswerCall( )** function does not have a parameter to specify a GC_PARM_BLK.

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 3.4.18, “gc_SetUserInfo( )”, on page 71 for more information about the values of function parameters to set in this context.

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 3.4.11, “gc_MakeCall( )”, on page 42 for more information about the values of function parameters to set in this context.

4.4.4. Setting Coder Information

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

Table 6 shows the coders that are supported when using the Global Call API with Intel® NetStructure™ IPT boards.

Table 6. Coders Supported for Intel® NetStructure™ IPT Boards

Coder and Rate	Global Call # Define	Frames Per Packet (fpp) or Frame Size (ms)	VAD Support
G.711 ALaw	GCCAP_AUDIO_g711Alaw64k	Frame Size: 10, 20, and 30 ms (Frames Per Packet: Fixed at 1 fpp)	Not supported
G.711 ULaw	GCCAP_AUDIO_g711Ulaw64k	Frame Size: 10, 20, and 30 ms (Frames Per Packet: Fixed at 1 fpp)	Not supported
G.723.1 5.3 Kbps	GCCAP_AUDIO_g7231_5_3k	Frames Per Packet: 1, 2, 3 or 4 (Frame Size: Fixed at 30 ms)	Supported
G.723.1, 6.3 Kbps	GCCAP_AUDIO_g7231_6_3k	Frames Per Packet: 1, 2, 3 or 4 (Frame Size: Fixed at 30 ms)	Supported
G.729 or G.729a	GCCAP_AUDIO_g729AnnexA	Frames Per Packet: 1, 2, 3, or 4 (Frame Size: Fixed at 30 ms)	Not applicable
G.729b or G.729a+b	GCCAP_AUDIO_g729AnnexA wAnnexB	Frames Per Packet: 1, 2, 3, or 4 (Frame Size: Fixed at 30 ms)	Supported
T.38	GCCAP_DATA_t38UDPFax	Not applicable	Not applicable

Notes:
1. For G.711 coders, the frame size value (not the frames per packet value) is specified in the frames_per_pkt field of the IP_AUDIO_CAPABILITY structure. See Section 7.3, "IP_AUDIO_CAPABILITY", on page 180 for more information.
2. Intel® NetStructure™ IPT boards support symmetrical coder definitions only, that is, the transmit and receive coder definitions must be the same.

Table 7 shows the coders that are supported when using the Global Call API with Intel® NetStructure™ DM/IP boards.

Table 7. Coders Supported for Intel® NetStructure™ DM/IP Boards

Coder and Rate	Global Call # Define	Frames Per Packet (fpp) or Frame Size (ms)	VAD Support
G.711 ALaw	GCCAP_AUDIO_g711Alaw64k	Frame Size: 10, 20 or 30 ms (Frames Per Packet: Fixed at 1 fpp)	Not applicable
G.711 ULaw	GCCAP_AUDIO_g711Ulaw64k	Frame Size: 10, 20 or 30 ms (Frames Per Packet: Fixed at 1 fpp)	Not applicable
G.723.1 5.3 Kbps	GCCAP_AUDIO_g7231_5_3k	Frames Per Packet: 1, 2 or 3 (Frame Size: Fixed at 30 ms)	Supported
G.723.1, 6.3 Kbps	GCCAP_AUDIO_g7231_6_3k	Frames Per Packet: 1, 2 or 3 (Frame Size: Fixed at 30 ms)	Supported
G.729 or G.729a	GCCAP_AUDIO_g729AnnexA	Frames Per Packet: 1, 2, 3 or 4 (Frame Size: Fixed at 10 ms)	Not applicable
G.729b or G.729a+b	GCCAP_AUDIO_g729AnnexA wAnnexB	Frames Per Packet: 3 or 4 (Frame Size: Fixed at 10 ms)	Supported
GSM Full Rate (TIPHON)	GCCAP_AUDIO_gsmFullRate	Frames Per Packet: 1, 2 or 3 (Frame Size: Fixed at 20 ms)	Disabled (default) or Enabled
T.38	GCCAP_DATA_t38UDPFax	Not applicable	Not applicable

Notes:
1. For G.711 coders, the frame size value (not the frames per packet value) is specified in the frames_per_pkt field of the IP_AUDIO_CAPABILITY structure. See Section 7.3, “IP_AUDIO_CAPABILITY”, on page 180 for more information.
2. GSM Telecommunications and Internet Protocol Harmonization over Networks (TIPHON) is a sub-group of the European Telecommunications Standards Institute (ETSI) GSM specification.
3. Intel® NetStructure™ DM/IP boards support G.726 play and record functionality only. Transcoding using G.726 is not supported.
4. GCCAP_dontCare can be used to indicate that any supported coder is valid.

Intel® NetStructure™ Host Media Processing (HMP) software performs voice, conferencing and IVR processing on general-purpose servers based on Intel® architecture without the use of specialized hardware. Table 8 shows the coders that are supported when using the Global Call API with HMP.

Table 8. Coders Supported for Host Media Processing (HMP)

Coder and Rate	Global Call # Define	Frames Per Packet (fpp) or Frame Size (ms)	VAD Support
G.711 ALaw	GCCAP_AUDIO_g711Alaw64k	Frame Size: 10, 20, and 30 ms (Frames Per Packet: Fixed at 1 fpp)	Not applicable
G.711 ULaw	GCCAP_AUDIO_g711Ulaw64k	Frame Size: 10, 20, and 30 ms (Frames Per Packet: Fixed at 1 fpp)	Not applicable
Note: For G.711 coders, the frame size value (not the frames per packet value) is specified in the frames_per_pkt field of the IP_AUDIO_CAPABILITY structure. See Section 7.3, "IP_AUDIO_CAPABILITY", on page 180 for more information.			

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, the following parameter set ID and parameter IDs are used to set up a GC_PARM_BLK containing the coder information:

- GCSET_CHAN_CAPABILITY
  - IPPARM_LOCAL_CAPABILITY - a parameter of type IP_CAPABILITY

Possible values for fields in the IP_CAPABILITY structure are:

- capability - One of the following:
  - GCCAP_AUDIO_g711Alaw64k
  - GCCAP_AUDIO_g711Ulaw64k
  - GCCAP_AUDIO_g7231_5_3k (at 5.3 kbits/s)
  - GCCAP_AUDIO_g7231_6_3k (at 6.3 kbits/s)
  - GCCAP_AUDIO_g729AnnexA

  - GCCAP_AUDIO_gsmFullRate
  - GCCAP_DATA_t38UDPFax
  - GCCAP_dontCare
- type - One of the following:
  - GCCAPTYPE_AUDIO
  - GCCAPTYPE_RDATA
- direction - One of the following:
  - IP_CAP_DIR_LCLTRANSMIT - transmit capability
  - IP_CAP_DIR_LCLRECEIVE - receive capability
  - IP_CAP_DIR_LCLRXTX - transmit and receive capability (T.38 only)

  NOTE: It is recommended to specify both the transmit and receive capabilities.
- payload_type - Not supported.
- extra - Must be of type IP_AUDIO_CAPABILITY.
  - extra.frames_per_packet - The number of frames per packet.

  NOTE: For G.711 coders, this field is the frame size (in ms).
  - extra.VAD - 0 (disabled) or 1 (enabled)

See Section 7.4, “IP_CAPABILITY”, on page 181 for more information.

4.4.5. Specifying Nonstandard Data Information When Using H.323

The following parameter set ID and parameter IDs can be used when setting up the GC_PARM_BLK pointed by the **infoparmblkp** function parameter:

- IPSET_NONSTANDARDDATA
  Either
  - IPPARM_NONSTANDARDDATA_DATA. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).
  - IPPARM_NONSTANDARDDATA_OBJID. The maximum length is MAX_NS_PARM_OBJID_LENGTH (40).

  or
  - IPPARM_NONSTANDARDDATA_DATA. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).

- IPPARM_H221NONSTANDARD

See Section 6.14, "IPSET_NONSTANDARDDATA Parameter Set", on page 167 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(&pParmBlock,
                        IPSET_NONSTANDARDDATA,
                        IPPARM_NONSTANDARDDATA_DATA,
                        (unsingned char)(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,
                                (unsingned char)sizeof(IP_H221NONSTANDARD),
                                &appH221NonStd);
    }
    else
    {
        gc_util_insert_parm_ref(&pParmBlock,
                                IPSET_NONSTANDARDDATA,
                                IPPARM_NONSTANDARDDATA_OBJID,
                                (unsingned char)(strlen(pOid)+1),
                                pOid);
    }
```

4.4.6. Specifying Nonstandard Control Information When Using H.323

Use the **gc_SetUserInfo( )** function with a **duration** parameter set to GC_SINGLECALL to set nonstandard control information. If the **duration** parameter is set to GC_ALLCALLS, the function will fail.

The following parameter set ID and parameter IDs should be used when setting up the GC_PARM_BLK pointed by the **infoparmblkp** function parameter:

- IPSET_NONSTANDARDCONTROL
  Either
  - IPPARM_NONSTANDARDDATA_DATA. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).
  - IPPARM_NONSTANDARDDATA_OBJID. The maximum length is MAX_NS_PARM_OBJID_LENGTH (40).

  or
  - IPPARM_NONSTANDARDDATA_DATA. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).
  - IPPARM_H221NONSTANDARD

See Section 6.15, “IPSET_NONSTANDARDCONTROL Parameter Set”, on page 168 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* 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(&pParmBlock,
                        IPSET_NONSTANDARDCONTROL,
                        IPPARM_NONSTANDARDDATA_DATA,
                        (unsingned char)(strlen(pControl)+1),
                        pControl);

    if (choiceOfNSControl)  /* App decide the CHOICE of
                               OBJECTIDENTIFIER. It can not 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);
    }
```

4.4.7. 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 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.

4.5. Retrieving Current Call-Related Information

To support large numbers of channels, the call control 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:

- 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.
- Use the **gc_Extension( )** function to request the data. The **target_type** should be GCTGT_GCLIB_CRN and the **target_id** should be the actual CRN. The **ext_id** function parameter (extension ID) should be set to IPEXTID_GETINFO, the **parmblkp** function parameter should point to the

GC_PARM_BLK set up in step 1, and the **mode** function parameter should be set to EV_ASYNC (asynchronous).

- 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.
- 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, but includes the size of the data following for each parameter requested.

Table 9 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.

Table 9. Retrievable Call Information

Para-meter	Set ID	Parameter ID	When Information Can Be Retrieved	Datatype in value_ buf Field (See Note Below)	H.323/ SIP
Call ID	IPSET_ CALLINFO	IPPARM_ CALLID	Any state after Offered or Proceeding.	String, max. length = IP_CALLID_ LENGTH (16 octets)	H.323 only
Call Duration	IPSET_ CALLINFO	IPPARM_ CALL DURATION	After Disconnected, before Idle.	Unsigned long (value in ms)	H.323 only
Conference Goal	IPSET_ CONFER ENCE	IPPARM_ CONFER ENCE_GOAL	Any state after Offered or Proceeding.	Uint[8]	H.323 only

Table 9. Retrievable Call Information

Para-meter	Set ID	Parameter ID	When Information Can Be Retrieved	Datatype in value_ buf Field (See Note Below)	H.323/ SIP
Conference ID	IPSET_ CONFER ENCE	IPPARM_ CONFER ENCE_ID	Any state after Offered or Proceeding.	char*, max. length = IP_CONFER ENCE_ID_ LENGTH (16)	H.323 only
Display Information	IPSET_ CALLINFO	IPPARM_ DISPLAY	Any state after Offered or Proceeding.	char*, max. length = MAX_ DISPLAY_ LENGTH (82), null-terminated	H.323 and SIP
Nonstandard Control	IPSET_ NON STANDARD CONTROL	IPPARM_ NON STANDARD DATA_DATA IPPARM_ NON STANDARD DATA_OBJID or IPPARM_ H221NON STANDARD	See Section 4.5.1, "Retrieving Nonstandard Data From Protocol Messages When Using H.323", on page 97 for more information.	char*, max. length = MAX_NS_ PARM_DATA _LENGTH (128) char*, max. length = MAX _NS_PARM_ OBJID_ LENGTH (40)	H.323 only

Table 9. Retrievable Call Information

Para-meter	Set ID	Parameter ID	When Information Can Be Retrieved	Datatype in value_buf Field (See Note Below)	H.323/ SIP
Nonstandard Data	IPSET_ NON STANDARD DATA	IPPARM_ NON STANDARD DATA_DATA IPPARM_ NON STANDARD DATA_OBJID or IPPARM_ H221NON STANDARD	See Section 4.5.1, "Retrieving Nonstandard Data From Protocol Messages When Using H.323", on page 97 for more information.	char*, max. length = MAX_NS_ PARM_DATA _LENGTH (128) char*, max. length = MAX _NS_PARM_ OBJID_ LENGTH (40)	H.323 only
Phone List	IPSET_ CALLINFO	IPPARM_ PHONELIST	Any state after Offered or Proceeding.	char*, maximum length = 131	H.323 and SIP
User to User Information	IPSET_ CALLINFO	IPPARM_ USERUSER_ INFO	Any state after Offered or Proceeding.	char*, max. length = MAX_USER USER_INFO_ LENGTH (131 octets)	H.323 only
Vendor Product ID	IPSET_ VENDOR INFO	IPPARM_ VENDOR_ PRODUCT_ ID	Any state after Offered or Proceeding.	char*, maximum length = MAX_ PRODUCT_ ID_LENGTH (32)	H.323 only

Table 9. Retrievable Call Information

Para-meter	Set ID	Parameter ID	When Information Can Be Retrieved	Datatype in value_ buf Field (See Note Below)	H.323/ SIP
Vendor Version ID	IPSET_ VENDOR INFO	IPPARM_ VENDOR_ VERSION_ID	Any state after Offered or Proceeding.	char*, maximum length = MAX_ VERSION_ID _LENGTH (32)	H.323 only
H.221 Nonstandard Information	IPSET_ VENDOR INFO	IPPARM_ H221NON STD	Any state after Offered or Proceeding.	IP_H221_ NON STANDARD (see note 4)	H.323 only

Note:

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 generated 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_ReleaseCall( )** is issued.

4.5.1. Retrieving Nonstandard Data From Protocol Messages When Using H.323

Any Q.931 message can include nonstandard data. The application can use the **gc_Extension( )** function with and **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 IPPARM_NONSTANDARDDATA_DATA in the extension request. It is not necessary to specify IPPARM_NONSTANDARDDATA_OBJID or IPPARM_H221NONSTANDARD. The call control library ensures that the GCEV_EXTENSIONCMPLT event includes the correct information.

4.5.2. Example of Retrieving Call-Related Information

The following code demonstrates how to:

- 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 9, “Retrievable Call Information”, on page 93).

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

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

```
                            sizeof(int),1);
    if (GC_SUCCESS != frc)
    {
        return GC_ERROR;
    }
    gc_util_insert_parm_val(&gcParmBlk,
                            IPSET_VENDORINFO,
                            IPPARM_VENDOR_VERSION_ID,
                            sizeof(int),1);
    if (GC_SUCCESS != frc)
    {
        return GC_ERROR;
    }
    gc_util_insert_parm_val(&gcParmBlk,
                            IPSET_VENDORINFO,
                            IPPARM_H221NONSTD,
                            sizeof(int),1);
    if (GC_SUCCESS != frc)
    {
        return GC_ERROR;
    }
    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;
    }
    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 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:
                        if(parmp->value_size != 0)
                        {
                            printf("\tReceived extension data CALLID: %s\n",
                                    parmp->value_buf);
                        }
                    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;
}
```

4.6. Handling DTMF

DTMF handling is described under the following topics:

- Specifying DTMF Support
- Getting Notification of DTMF Detection
- Generating DTMF

4.6.1. Specifying DTMF Support

Global Call can be used to configure which DTMF modes (UII Alphanumeric, RFC 2833, or Inband) are supported by the application. The DTMF mode can be specified:

- For all line devices simultaneously 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_SetUserInfo( )** with a **duration** parameter value of GC_SINGLECALL.

The GC_PARM_BLK associated with the **gc_SetConfigData( )** or **gc_SetUserInfo( )** function identifies which DTMF modes are supported. The GC_PARM_BLK includes the following parameter set ID and parameter ID:

- IPSET_DTMF - The parameter set specific to DTMF.
  - IPPARM_SUPPORT_DTMF_BITMASK - Parameter ID that specifies the DTMF transmission mode(s). The possible values are:
    - IP_DTMF_TYPE_ALPHANUMERIC - This is the default.
      - For H.323, DTMF digits are sent and received in H.245 User Input Indication (UII) Alphanumeric messages.
      - For SIP, this value is **not** supported, therefore, one of the two options following must be explicitly specified.
    - IP_DTMF_TYPE_INBAND_RTP - DTMF digits are sent and received inband via standard RTP transcoding.
    - IP_DTMF_TYPE_RFC_2833 - DTMF digits are send and received in the RTP stream as defined in RFC 2833.

NOTE: The IPPARM_SUPPORT_DTMF_BITMASK can only be replaced; it cannot be modified. For each **gc_SetConfigData( )** or **gc_SetUserInfo( )** call, the IPPARM_SUPPORT_DTMF_BITMASK parameter is overwritten.

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.

Table 10 summarizes the DTMF mode settings and associated behavior.

Table 10. Summary of DTMF Mode Settings and Behavior

IP_DTMF_ TYPE_ RFC_2833	IP_DTMF_ TYPE_ ALPHA NUMERIC†	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
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

† Applies to H.323 only.

Table 10. Summary of DTMF Mode Settings and Behavior

IP_DTMF_ TYPE_ RFC_2833	IP_DTMF_ TYPE_ ALPHA NUMERIC†	IP_DTMF_ TYPE_ INBAND	Transmit (Tx) DTMF Mode	Receive (Rx) DTMF Mode
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 set and parameter ID:

- IPSET_DTMF
  - IPPARM_DTMF_RFC2833_PAYLOAD_TYPE - Identifies the payload type. Possible values are:
    - IP_USE_STANDARD_PAYLOADTYPE (default payload type, 101)
    - A value in the range 96 to 127 (dynamic payload type)

4.6.2. Getting Notification of DTMF Detection

Once DTMF support has been configured (see Section 4.6.1, “Specifying DTMF Support”, on page 104), 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 4.14, “Enabling and Disabling Unsolicited Notification Events”, on page 140.

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 as follows:

- IPSET_DTMF - The parameter set specific to DTMF.
  - IPPARM_DTMF_ALPHANUMERIC - For H.323, DTMF digits are received in H.245 User Input Indication (UII) alphanumeric messages.

The parameter value is of type IP_DTMF_DIGITS. See Section 7.7, "IP_DTMF_DIGITS", on page 184 for more information. For SIP, this parameter is **not** supported.

- IPPARM_DTMF_RFC_2833 - DTMF digits received in the RTP stream as defined in RFC 2833. The parameter value is of type IP_RFC2833_EVENT.

4.6.3. Generating DTMF

Once DTMF support has been configured (see Section 4.6.1, "Specifying DTMF Support", on page 104), 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 following parameter set ID and one of the following parameter IDs:

- IPSET_DTMF - The parameter set specific to DTMF.
  - 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.
  - IPPARM_DTMF_RFC_2833 - Specifies that DTMF digits are to be sent in the RTP stream as defined in RFC 2833. The DTMF information is contained in the parameter value which is of type IP_RFC2833_EVENT.

4.7. Getting Media Streaming Status and Negotiated Coder Information

The application can receive notification of changes in the status (connection and disconnection) of media streaming in the transmit and receive directions. When notification of the connection of the media stream in either direction is received, information about the coders negotiated for that direction is also available.

The events for this notification must be enabled; see Section 4.14, “Enabling and Disabling Unsolicited Notification Events”, on page 140. 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:

- extID - IPEXID_MEDIAINFO
- parmblk - A GC_PARM_BLK containing protocol connection status with the following parameter set ID and one of the following parameter IDs:
  - IPSET_MEDIA_STATE - A parameter set specific to underlying protocol states.
    - IPPARM_TX_CONNECTED - Media streaming has been initiated in transmit direction. The datatype of the parameter is IP_CAPABILITY and contains the coder configuration that resulted from the capability exchange with the remote peer.
    - IPPARM_TX_DISCONNECTED - Media streaming has been terminated in transmit direction. The parameter value is not used.
    - IPPARM_RX_CONNECTED - Media streaming has been initiated in receive direction. The datatype of the parameter is IP_CAPABILITY and contains the coder configuration that resulted from the capability exchange with the remote peer.
    - IPPARM_RX_DISCONNECTED - Media streaming has been terminated in receive direction. The parameter value is not used.

4.8. 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 4.14, “Enabling and Disabling Unsolicited Notification Events”, on page 140.

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 protocol connection status with the following parameter set ID and one of the following parameter IDs:
  - IPSET_IPPROTOCOL_STATE - A parameter set specific to underlying protocol states.
    - 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 with the remote endpoint. 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).

4.9. Sending Protocol Messages

The following message types are supported:

- Nonstandard User Input Indication (UII) Message (H.245)
- Nonstandard Facility Messages (Q.931)
- Nonstandard Registration Messages

Table 11 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 11. Summary of Protocol Messages that Can be Sent

Type	Set ID	Parameter ID	When Message Should be Sent
Nonstandard UII Message (H.245)	IPSET_MSG_H245	IPPARM_MSGTYPE (set to IP_MSGTYPE_H245_ INDICATION)	When the call is in the Connected state only.
Nonstandard Facility message (Q.931)	IPSET_MSG_Q931	IPPARM_MSGTYPE (set to IP_MSGTYPE_Q931_ FACILITY)	In any call state.
Nonstandard registration message	IPSET_MSG_RAS	IPPARM_MSGTYPE (set to IP_MSGTYPE_REG_ NONSTD	

4.9.1. Nonstandard UII Message (H.245)

Use the **gc_Extension( )** function in asynchronous mode with an **ext_ id** (extension ID) of IPEXTID_SENDMSG to send nonstandard UII messages. The **target_type** should be GCTGT_GCLIB_CRN and the **target_id** should be the actual CRN. At the sending end, a GCEV_EXTENSIONCMPLT event is received indicating that the message has been sent. At the receiving end, a GCEV_EXTENSION event with the same **ext_id** value is generated. 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.

The relevant parameter set IDs and parameter IDs for this purpose are:

- IPSET_MSG_H245
  - IPPARM_MSGTYPE - Set to IP_MSGTYPE_H245_INDICATION.
- IPSET_NONSTANDARDDATA

  Either

- IPPARM_NONSTANDARDDATA_DATA - Actual nonstandard data. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).
- IPPARM_NONSTANDARDDATA_OBJID - Object ID string. The maximum length is MAX_NS_PARM_OBJID_LENGTH (40).

or

- IPPARM_NONSTANDARDDATA_DATA - Actual nonstandard data. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).
- IPPARM_H221NONSTANDARD - H.221 nonstandard data identifier.

NOTE: The message type (IPPARM_MSGTYPE) is mandatory. At least one other information element must be included.

See Section 6.12, "IPSET_MSG_Q931 Parameter Set", on page 166 and Section 6.14, "IPSET_NONSTANDARDDATA Parameter Set", on page 167 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(&t_PrmBlkp, IPSET_NONSTANDARDDATA,
                             IPPARM_NONSTANDARDDATA_OBJID, ObjLen+1, ObjId);

rc = gc_util_insert_parm_ref(&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);
.
.
.
```

4.9.2. Nonstandard Facility Message (Q.931)

Use the **gc_Extension( )** function in asynchronous mode with an **ext_ id** (extension ID) of IPEXTID_SENDMSG to send nonstandard facility (Q.931 Facility) messages. The **target_type** should be GCTGT_GCLIB_CRN and the **target_id** should be the actual CRN. At the sending end, a GCEV_EXTENSIONCMPLT event is received indicating that the message has been sent. At the receiving end, a GCEV_EXTENSION event with the same **ext_id** value is generated. 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.

The relevant parameter set IDs and parameter IDs are:

- IPSET_MSG_Q931
  - IPPARM_MSGTYPE - Set to IP_MSGTYPE_Q931_FACILITY.
- IPSET_NONSTANDARDDATA

  Either
  - IPPARM_NONSTANDARDDATA_DATA - Actual nonstandard data. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).
  - IPPARM_NONSTANDARDDATA_OBJID - Object ID string. The maximum length is MAX_NS_PARM_OBJID_LENGTH (40).

  or
  - IPPARM_NONSTANDARDDATA_DATA - Actual nonstandard data. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).
  - IPPARM_H221NONSTANDARD - H.221 nonstandard data identifier.

NOTE: The message type (IPPARM_MSGTYPE) is mandatory. At least one other information element must be included.

See Section 6.12, "IPSET_MSG_Q931 Parameter Set", on page 166 and Section 6.14, "IPSET_NONSTANDARDDATA Parameter Set", on page 167 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(&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);
```

4.9.3. Nonstandard Registration Message

Use the **gc_Extension( )** function in asynchronous mode with an **ext_ id** (extension ID) of IPEXTID_SENDMSG to send nonstandard registration messages. The **target_type** should be GCTGT_GCLIB_CRN and the **target_id** should be the actual CRN. At the sending end, a GCEV_EXTENSIONCMPLT event is received indicating that the message has been sent. At the receiving end, a

GCEV_EXTENSION event with the same **ext_id** value is generated. 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.

The relevant parameter set IDs and parameter IDs for this purpose are:

- IPSET_MSG_REGISTRATION
  - IPPARM_MSGTYPE - Set to IP_MSGTYPE_REG_NONSTD.
- IPSET_NONSTANDARDDATA
  - IPPARM_NONSTANDARDDATA_DATA - Actual nonstandard data. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).
  - IPPARM_NONSTANDARDDATA_OBJID - Object ID string. The maximum length is MAX_NS_PARM_OBJID_LENGTH (40).

  OR
  - IPPARM_NONSTANDARDDATA_DATA - Actual nonstandard data. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).
  - IPPARM_H221NONSTANDARD - H.221 nonstandard data identifier.

NOTE: The message type (IPPARM_MSGTYPE) is mandatory. At least one other information element must be included.

See Section 6.13, “IPSET_MSG_REGISTRATION Parameter Set”, on page 167 and Section 6.14, “IPSET_NONSTANDARDDATA Parameter Set”, on page 167 for more information.

4.9.4. 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 10 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.

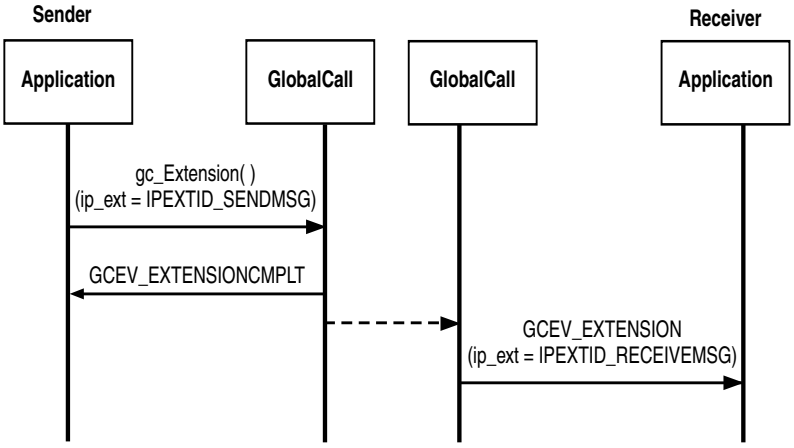


Figure 10. Sending Protocol Messages

4.10. Quality of Service Alarm Management

Global Call 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 Global Call are:

- Lost packets
- Jitter

When developing applications that use Intel® NetStructure™ IPT boards, the only threshold attribute supported is the fault threshold value. Similarly, when developing applications that use Intel® NetStructure™ DM/IP boards, the supported threshold attributes are: time interval, debounce on, debounce off, fault threshold, percent success threshold, and percent fail threshold.

See the *IP Media Library API Library Reference* and the *IP Media Library API Programming Guide* for more information on the supported thresholds.

When using Global Call with other technologies (such as E-1 CAS, T-1 Robbed Bit etc.), 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 Global Call line device. This Global Call line device can be used directly in Global Call Alarm Management System (GCAMS) functions to manage alarms on the network device.

When using Global Call with IP technology, alarms such as QoS alarms, are more directly related to the media processing and are therefore reported on the media device not 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 Global Call line devices are created:

- The first Global Call line device corresponds to the network device and is retrieved in the **gc_OpenEx( )** function.
- The second Global Call 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 Global Call can be used to set or retrieve QoS threshold values.

4.10.1. Alarm Source Object Name

In Global Call, alarms are managed using the Global Call Alarm Management System (GCAMS). Each alarm source is represented by an Alarm Source Object (ASO) that has an associated name. When using Global Call 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.

4.10.2. Retrieving the Media Device Handle

To retrieve the Global Call line device corresponding to the media device, use the **gc_GetResourceH( )** function. See Section 3.4.8, “gc_GetResourceH( )”, on page 41 for more information.

The Global Call line device corresponding to the media device is the device that must be used with GCAMS functions to manage QoS alarms.

4.10.3. Setting QoS Threshold Values

To set QoS threshold values, use the **gc_SetAlarmParm( )** function. See Section 3.4.16, "gc_SetAlarmParm( )", on page 67 for more information.

The following code demonstrates how to set QoS threshold values.

NOTES: 1. The following code uses the IPM_QOS_THRESHOLD_INFO structure from the IP Media Library (IPML). See the *IP Media Library API Library Reference* and the *IP Media Library API Programming Guide* for more information.

2. The unTimeInterval, unDebounceOn, unDebounceOff, unPercentSuccessThreshold, unPercentFailThreshold fields are not supported when using Intel® NetStructure™ IPT boards. These values should be set to 0.

```
/*******************************************************************************
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_LOSTPACKETS;
    QoS_info.QoSThresholdData[0].unTimeInterval = 50;
    QoS_info.QoSThresholdData[0].unDebounceOn = 100;
    QoS_info.QoSThresholdData[0].unDebounceOff = 100;
    QoS_info.QoSThresholdData[0].unFaultThreshold = 10;
    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_qosthresholdalarm, &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);
}
```

4.10.4. Retrieving QoS Threshold Values

To retrieve QoS threshold values, use the **gc_GetAlarmParm( )** function. See Section 3.4.6, "gc_GetAlarmParm( )", on page 38 for more information.

The following code demonstrates how to retrieve QoS threshold values.

NOTES: 1. The following code uses the IPM_QOS_THRESHOLD_INFO structure from the IP Media Library (IPML). See the *IP Media Library API Library Reference* and the *IP Media Library API Programming Guide* for more information.

2. The unTimInterval, unDebounceOn, unDebounceOff, unPercentSuccessThreshold, unPercentFailThreshold fields are not supported when using Intel® NetStructure™ IPT boards.

```
/*******************************************************************************
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_qosthresholdalarm, &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");
    }
}
```

4.10.5. 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 Global Call 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 *Global Call API Programming Guide* for more information about the operation of GCAMS and the *Global Call API Library Reference* for more information about GCAMS functions.

4.11. Specifying RTP Stream Establishment

When using Global Call, 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 3.4.18, “gc_SetUserInfo( )”, on page 71 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

4.12. 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 performs a similar function. Global Call provides applications with the ability to perform endpoint registration. Registration tasks supported include:

- Performing registration-related operations
- Receiving notification of registration

When using Global Call to perform endpoint registration, the following 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".
- An application must perform registration before using **gc_OpenEx( )** on any other line device.
- Once an application chooses to be registered with a gatekeeper, it may change its gatekeeper/registrar by deregistering and reregistering with another gatekeeper/registrar, but it cannot handle calls without being registered with some gatekeeper/registrar.

  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 must issue a **gc_Stop( )**, then a **gc_Start( )**.
- Once an application is registered and has active calls, deregistration or switching to a different gatekeeper must be done only when all calls are in the Idle state. The **gc_ResetLineDev( )** function can be used to put all channels in the Idle state.
- When setting alias information, if the protocol is H.323 only, the **gc_ReqService( )** function can include more than one alias in the GC_PARM_BLK associated with the function. If the registration target includes SIP, aliases and prefixes should not be included.
- When using the **gc_ReqService( )** function, two mandatory parameters, PARM_REQTYPE and PARM_ACK, both in the GCSET_SERVREQ parameter set, are required in the GC_PARM_BLK parameter block. These parameters are required by the generic service request mechanism provided by Global Call and are not sent in any registration message.
- Registration operations cannot be included in the preset registration information using **gc_SetConfigData( )**.

4.12.1. Performing Registration Operations

Global Call provides a number of options for registration and manipulation of registration information. The Global Call API simplifies and abstracts the network RAS messages in H.323 and Registrar messages in SIP. The following functionality is supported:

- Locating a registration server (gatekeeper in H.323 or registrar in SIP) via unicast or multicast (RAS messages: LRQ/LCF/LRJ, 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*.

Locating a Registration Server

A Global Call application can choose to use a known address for the registration server (gatekeeper in H.323 or registrar in SIP) 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 of the IPSET_REG_INFO parameter set. The IPPARM_REG_ADDRESS is of type IP_REGISTER_ADDRESS which contains the reg_server field that is the address value. A specific range of IP addresses are 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.

NOTES: 1. The application can specify the maximum number of hops (connections between routers) in the max_hops field of the IP_REGISTER_ADDRESS structure. This field applies only to H.323 applications using gatekeeper discovery (H.225 RAS) via the default multicast registration address.

2. When using H.323, the port number used for RAS is one less than the port number used for signaling. Consequently, to avoid a conflict when configuring multiple IPT board devices in the IPCCLIB_START_DATA structure, do not assign consecutive H.323 signaling port numbers to IPT board devices. See Section 3.4.19, "gc_Start( )", on page 72 for more information.

Registration

An application can use 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 4.12.4, "Registration Code Example", on page 129 for more information.

Specifying One-Time or Periodic Registration

Global Call enables an application to specify a one-time registration or periodic registration where information is re-registered with the gatekeeper/registrar at the interval (in seconds) specified by the application. This is achieved by setting the **time_to_live** parameter in the IP_REGISTER_ADDRESS structure. If the parameter is set to zero, then the stack uses one-time registration functionality. If the parameter is set to a value greater than zero, for example 5, then each registration with the server is valid for 5 seconds and the stack will automatically refresh its request before timeout. Registered applications are not notified of the refresh transactions.

When using SIP, periodic registration is also supported. The behavior depends on the time_to_live value specified in the IP_REGISTER_ADDRESS structure as follows:

- If the time_to_live value is specified, registration is done with this value set in the Expires header.
- If the time_to_live value is zero, the call control library automatically sets the Expires header to a value of 3600 seconds, which is treated as an application-specified time_to_live value.

NOTE: The actual expiration time for registration is determined by the registrar. The expiration time received from the registrar is stored and when half of this time expires, re-registration occurs.

Changing Registered Information

Global Call provides the ability to modify or add to the registration information after it has been registered with the gatekeeper/registrar. To change registration information, use the **gc_ReqService( )** function. The GC_PARM_BLK in this context should contain an element with a set ID of IPSET_REG_INFO and a parameter ID of IPPARM_OPERATION_REGISTER with a value of:

- IP_REG_SET_INFO - To override existing registration.
- IP_REG_ADD_INFO - To add to existing registration information.

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 3, “Registration Information”, on page 63.

Removing Registered Information by Value

When an application needs to delete one (or more) of its aliases or supported prefixes from the list, it may use the **gc_ReqService( )** function. The GC_PARM_BLK in this context should contain an element with a set ID of IPSET_REG_INFO and a parameter ID of IPPARM_OPERATION_REGISTER with a value of IP_REG_DELETE_BY_VALUE. If the string is registered, it will be deleted from the database and an updated list will be sent to the gatekeeper.

NOTE: When using IPPARM_OPERATION_REGISTER, the value IP_REG_DELETE_ALL is prohibited.

Sending Nonstandard Registration Messages

Global Call 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 6.13, "IPSET_MSG_REGISTRATION Parameter Set", on page 167. Other elements are set as in conventional nonstandard messages; see Section 6.14, "IPSET_NONSTANDARDDATA Parameter Set", on page 167.

Deregistering

Global Call provides the ability to deregister from a gatekeeper/registrar. To deregister, an application uses the **gc_ReqService( )** function. When deregistering, the application can decide whether to keep the registration information locally or delete it. The GC_PARM_BLK in this context should contain an element with a set ID of IPSET_REG_INFO and a parameter ID of IPPARM_OPERATION_DEREGISTER with a value set to either:

- IP_REG_MAINTAIN_LOCAL_INFO - To keep the registration information locally.
- IP_REG_DELETE_ALL - To delete the registration information stored locally.

See Section 4.12.5, "Deregistration Code Example", on page 132 for more information.

4.12.2. Receiving Notification of Registration

An application that sends a registration request to a gatekeeper/registrar will receive notification of whether the registration is successful or not. When using Global Call the application will receive 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
- IP_REG_REJECTED

4.12.3. Receiving Nonstandard Registration Messages

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 as follows. A message identifier element of the form:

- IPSET_MSG_REGISTRATION
  - IPPARM_MSGTYPE = IP_MSGTYPE_REG_NONSTD

One or more additional elements that contain the message data of the form:

- IPSET_NONSTANDARDDATA
  - IPPARM_NONSTANDARDDATA_DATA. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).
  - IPPARM_NONSTANDARDDATA_OBJID. The maximum length is MAX_NS_PARM_OBJID_LENGTH (40).

OR

- IPSET_NONSTANDARDDATA
  - IPPARM_NONSTANDARDDATA_DATA. The maximum length is MAX_NS_PARM_DATA_LENGTH (128).
  - IPPARM_H221NONSTANDARD

4.12.4. Registration Code Example

The following code example shows how to populate a GC_PARM_DATA structure that can be used to register an endpoint with a gatekeeper (H.323) or registrar (SIP). The GC_PARM_DATA structure contains the following registration information:

- Two mandatory parameters required by the generic **gc_ReqService( )** function
- The protocol type (H.323, SIP or both)
- The type of operation (register/deregister) and sub-operation (set information, add information, delete by value, delete all)

- The IP address to be registered
- A number of local aliases
- A number of supported prefixes

```
int boardRegistration(IN LINEDEV boarddev)
{
    GC_PARM_BLKP pParmBlock = NULL;
    int frc = GC_SUCCESS;

    /****** 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),
                                  IP_PROTOCOL_H323); /*can be H323, SIP or Both*/


    /****** 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;
    strcpy(registerAddress.reg_server,"101.102.103.104"); /* set server address*/
    strcpy(registerAddress.reg_client,"10.20.30.40");     /* set client (self)
                                                             address */
    registerAddress.max_hops = regMulticastHops;
    registerAddress.time_to_live = regUnicastTTL;

    frc = gc_util_insert_parm_ref(&pParmBlock,
                                  IPSET_REG_INFO,
```

```
                                IPPARM_REG_ADDRESS,
                                (UINT8)sizeof(IP_REGISTER_ADDRESS),
                                &registerAddress);

    /**** Setting terminalAlias information ****/
    /**** With H.323 - may repeat this line with different aliases and
            alias types ****/
    /**** SIP does not allow setting of this parm block ****/
    frc = gc_util_insert_parm_ref(&pParmBlock,
                                IPSET_LOCAL_ALIAS,
                                (unsigned short)IPPARM_ADDRESS_EMAIL,
                                (UINT8)(strlen("someone@someplace.com")+1),
                                "someone@someplace.com");

    /****** Setting supportedPrefixes information ***********/
    /**** With H.323 - may repeat this line with different supported prefixes and
           supported prefix types ****/
    /**** SIP does not allow setting of this parm block ****/
    frc = gc_util_insert_parm_ref(&pParmBlock,
                                IPSET_SUPPORTED_PREFIXES,
                                (unsigned short)IPPARM_ADDRESS_PHONE,
                                (UINT8)(strlen("011972")+1),
                                "011972");

    /****** 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;
}
```

4.12.5. Deregistration Code Example

The following code example shows how to populate a GC_PARM_DATA structure that can be used to deregister an endpoint with a gatekeeper (H.323). The GC_PARM_DATA 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),
                                  IP_REQTYPE_REGISTRATION);

    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;
}
```

4.13. Sending and Receiving Faxes over IP

The functionality described in this section are the mechanisms that support the sending are receiving of fax information over IP (FoIP). Separate fax resources are required to handle fax transmission and reception.

NOTE: Sending and receiving faxes using SIP is not currently supported.

A fax over IP (FoIP) call can be initiated in the following ways:

- At call setup time, the local side requests FoIP (T.38 only) for either an outgoing or incoming call.
- At call setup time, the remote side requests FoIP (T.38 only) for either an outgoing or incoming call.
- A voice call is connected and fax tones are detected on the local endpoint; the call switches to FoIP transcoding.
- A voice call is connected and the remote endpoint requests a switch to FoIP transcoding; the call switches to FoIP transcoding.

In any one of these scenarios, the local application must specify T.38 coder capability in advance if FoIP exchange is to be allowed.

4.13.1. Specifying T.38 Coder Capability

Using Global Call, T.38 coder support is specified in the same manner as any other coder capability, that is:

- 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.

To support the initiation of a T.38-only call, the application must specifically disable audio capability. This cannot be achieved by specifying no audio capability, since specifying no audio capability is equivalent to a "don't care" condition meaning all capabilities are enabled. Consequently, the audio capabilities must be explicitly disabled by specifying a GCCAP_AUDIO_disabled capability in the capabilities list.

When specifying the capability on a line device basis or on a per call basis, a GC_PARM_BLK with the following parameter set ID and parameter ID must be set up:

- GCSET_CHAN_CAPABILITY
  - IPPARM_LOCAL_CAPABILITY

The GCPARM_LOCAL_CAPABILITY parameter is of type IP_CAPABILITY and should include the following field values:

- capability - set to GCCAP_DATA_t38UDPFax
- type - GCCAPTYPE_RDATA
- direction - IP_CAP_DIR_LCLTXRX
- payload - Not supported
- extra - A parameter of type IP_DATA_CAPABILITY with the following field values:
  - max_bit_rate - set to a value of 144 (in units of 100 bits/sec)

See Section 7.4, “IP_CAPABILITY”, on page 181 for more information.

4.13.2. Initiating Fax Transcoding

Calls initiated or answered using the Global Call API support fax transcoding transparently without intervention by the application. For fax transcoding to occur, the line device or call must have specified and exchanged the T.38 UDP coder as one of the supported channel capabilities.

If this coder has been specified, fax transcoding will be initiated upon detection of a CED, CNG or V.21 tone from the local endpoint. Upon detection of one of these fax tones, the current audio RTP stream will be terminated and fax transmission will be initiated. If the remote endpoint does not support T.38 UDP fax capability, no T.38 transcoding change occurs.

4.13.3. Termination of Fax Transcoding

Fax termination can be triggered in the following ways:

- A call disconnection from either endpoint, that is, **gc_DropCall( )** from the local endpoint or a GCEV_DISCONNECTED event from the remote endpoint.
- The detection of a fax termination event on the local endpoint.
- The remote endpoint sends a signal (via the signaling protocol, for example, H.323 or SIP) to terminate fax transcoding.

In the last two cases, once fax transcoding using T.38 is completed, Global Call transitions back to the audio transcoding in use prior to the fax call. This occurs automatically without any intervention by the application.

NOTE: The call in this context refers to all communication with the remote endpoint, that is, both media transcoding and signaling.

4.13.4. T.38-Only Transcoding

To initiate a T.38-only call, the audio capabilities must be specifically disabled by the application by specifying a capability of GCCAP_AUDIO_disabled in the capabilities list. In this call scenario, no audio transcoding is done before or after the T.38 transmission. The call is initiated and terminated while using the T.38 transcoder; no transition back to an audio transcoder is necessary.

4.13.5. Getting Notification of Audio-to-Fax Transition

Audio transcoding to fax transcoding is done automatically with no intervention necessary by the application, but the application can be configured to receive notification when the transition takes place. The events for this notification must be enabled; see Section 4.14, "Enabling and Disabling Unsolicited Notification Events", on page 140 for information on enabling streaming connection and disconnection events (EXTENSIONEVT_STREAMING_STATUS).

Once the notification events have been enabled, when an audio transcoding session transitions to fax transcoding, four GCEV_EXTENSION events are received, each with the extID of IPEXTID_MEDIAINFO and a parameter set ID of IPSET_MEDIA_STATE.

Each GCEV_EXTENSION event contains a parameter. The parameter for each event in order of reception is as follows:

1. IPPARM_TX_DISCONNECTED - The transmit audio RTP stream is terminated. The GC_PARM_BLK does not contain any additional information.
2. IPPARM_RX_DISCONNECTED - The receive audio RTP stream is terminated. The GC_PARM_BLK does not contain any additional information.
3. IPPARM_TX_CONNECTED - Transmit fax transcoding is initiated. The datatype of the parameter is an IP_CAPABILITY structure representing the T.38 transcoder being used. See Section 4.13.1, “Specifying T.38 Coder Capability”, on page 134 for more information.
4. IPPARM_RX_CONNECTED - Receive fax transcoding is initiated. The datatype of the parameter is an IP_CAPABILITY structure representing the T.38 transcoder in use. See Section 4.13.1, “Specifying T.38 Coder Capability”, on page 134 for more information.

4.13.6. Getting Notification of Fax-to-Audio Transition

Fax transcoding to audio transcoding is done automatically with no intervention necessary by the application, but the application can be configured to receive notification when the transition takes place. The events for this notification must be enabled; see Section 4.14, “Enabling and Disabling Unsolicited Notification Events”, on page 140 for information on enabling streaming connection and disconnection events (EXTENSIONEVT_STREAMING_STATUS).

Once the notification events have been enabled, when a fax transcoding session transitions back to the prior audio transcoding session, four GCEV_EXTENSION events are received, each with the extID of IPEXTID_MEDIAINFO and a parameter set ID of IPSET_MEDIA_STATE.

Each GCEV_EXTENSION event contains a parameter. The parameter for each event in order of reception is as follows:

1. IPPARM_TX_DISCONNECTED - The transmit fax RTP stream is terminated. No more information is contained in the GC_PARM_BLK.
2. IPPARM_RX_DISCONNECTED - The receive fax RTP stream is terminated. No more information is contained in the GC_PARM_BLK.

3. IPPARM_TX_CONNECTED - Transmit audio transcoding is initiated. The datatype of the parameter is an IP_CAPABILITY structure representing the audio transcoder setting in use before fax transcoding was initiated. See Section 4.13.1, "Specifying T.38 Coder Capability", on page 134 for more information.
4. IPPARM_RX_CONNECTED - Receive audio transcoding is initiated. The datatype of the parameter is an IP_CAPABILITY structure representing the audio transcoder setting in use before fax transcoding was initiated. See Section 4.13.1, "Specifying T.38 Coder Capability", on page 134 for more information.

4.13.7. Getting Notification of T.38 Status Changes

The application can receive notification of underlying T.38 status changes including: tone detection on the TDM or IP sides, T.38 capability frame status, T.38 information frame status, and T.38 HDLC frame status. The events for this notification must be enabled; see Section 4.14, "Enabling and Disabling Unsolicited Notification Events", on page 140 for information on enabling T.38 fax status changes (EXTENSIONEVT_T38_STATUS).

Once these events are enabled, when the T.38 status 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_FOIP
- A GC_PARM_BLK that contains information about the T.38 status change. The GC_PARM_BLK can contain the following parameter set IDs and parameter IDs:
  - IPSET_TDM_TONEDET - A parameter set identifying a tone detected on the TDM side as identified by one of the following parameter IDs:
    - IPPARM_TDMDET_CED - CED tone detected from TDM side.
    - IPPARM_TDMDET_CNG - CNG tone detected from TDM side.
    - IPPARM_TDMDET_V21 - V.21 tone detected from TDM side.

    NOTE: The parameter value field in the GC_PARM_BLK in each case is unused (NULL).

- IPSET_T38_TONEDET - A parameter set identifying a tone detected on the IP side as identified by one of the following parameter IDs:
  - IPPARM_T38DET_CED - CED tone detected from IP side.
  - IPPARM_T38DET_CNG - CNG tone detected from IP side.
  - IPPARM_T38DET_V21 - V21 tone detected from IP side.

  NOTE: The parameter value field in the GC_PARM_BLK in each case is unused (NULL).
- IPSET_T38CAPFRAME_STATUS - A parameter set identifying the T.38 capability frame type identified by one of the following parameter IDs:
  - IPPARM_T38CAPFRAME_TX_DIS_DTC
  - IPPARM_T38CAPFRAME_TX_DCS
  - IPPARM_T38CAPFRAME_TX_CTC
  - IPPARM_T38CAPFRAME_RX_DIX_DTC
  - IPPARM_T38CAPFRAME_RX_DCS
  - IPPARM_T38CAPFRAME_RX_CTC

  NOTE: The parameter value field in the GC_PARM_BLK in each case includes the capability frame status structure, IPM_T38CAPFRAM_STATUS_INFO.
- IPSET_T38INFOFRAME_STATUS - A parameter set identifying the T.38 information frame type identified by one of the following parameter IDs:
  - IPPARM_T38INFOFRAME_TX_SUB
  - IPPARM_T38INFOFRAME_RX_SUB
  - IPPARM_T38INFOFRAME_TX_SEP
  - IPPARM_T38INFOFRAME_RX_SEP
  - IPPARM_T38INFOFRAME_TX_PWD
  - IPPARM_T38INFOFRAME_RX_PWD
  - IPPARM_T38INFOFRAME_TX_TSI
  - IPPARM_T38INFOFRAME_RX_TSI
  - IPPARM_T38INFOFRAME_TX_CSI
  - IPPARM_T38INFOFRAME_RX_CSI
  - IPPARM_T38INFOFRAME_TX_CIG

- IPPARM_T38INFOFRAME_RX_CIG

NOTE: The parameter value field in this GC_PARM_BLK in each case includes the frame buffer.

- IPSET_T38HDLCFRAME_STATUS - A parameter set identifying the T.38 HDLC frame type identified by one of the following parameter IDs:
  - IPPARM_T38HDLCFRAME_TX
  - IPPARM_T38HDLCFRAME_RX

4.14. Enabling and Disabling Unsolicited Notification Events

The application can enable and disable the GCEV_EXTENSION events associated with unsolicited notification including:

- DTMF digit detection
- Underlying protocol (Q.931 and H.245) connection state changes
- Media streaming connection state changes
- T.38 fax events

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 the following parameter set ID and parameter IDs:

- IPSET_EXTENSIONEVT_MSK - the parameter set ID for all extension event parameters. The parameter ID that can be included in this set are:
  - 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) 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_DTMF_RFC2833 - For notification of DTMF digits received in the RTP stream as described in RFC 2833.
- EXTENSIONEVT_SIGNALING_STATUS - For notification of intermediate protocol state changes in signaling (for example in H.323, Q.931 Connected and Disconnected) and control (for example in H.323, 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.
- EXTENSIONEVT_T38_STATUS - For notification of fax tones, capability frame type, info frame type, and HDLC frame type detected on T.38 fax.

4.15. Configuring the Sending of the Proceeding Message

The application can configure if the Proceeding message is sent under application control (using the **gc_CallAck( )** function) or automatically by the stack. The **gc_SetConfigData( )** function can be used for this purpose. The relevant set ID and parameter ID that must be included in the associated GC_PARM_BLK are:

- GCSET_CHAN_CONFIG
  - GCPARM_CALLPROC - Possible values are:
    - 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.

4.16. Enabling and Disabling Tunneling in 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 set ID and parameter ID in the GCLIB_MAKECALL_BLK used by the **gc_MakeCall( )** function:

- IPSET_CALLINFO
  - IPPARM_H245TUNNELING - Possible values are:
    - 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 2.2.1, "IPT Board Devices", on page 24). 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).

4.17. 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/index.html.

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.

Global Call 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. Global Call 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.

5. Building Applications

5.1. Required System Software

The Intel® Dialogic® system software must be installed on the development system. See the *Software Installation Guide* for the system software for more information.

5.2. Global Call IP-Specific Header Files

In addition to the generic header files required by Global Call, the following technology-specific header files are also required when building IP applications that use Global Call for call control:

- *gcip.h*
- *gcip_defs.h*
- *gccfgparm.h*
- *gcipmlib.h* - for Quality of Service (QoS) features

5.3. Required Libraries

libgc.lib is the only Global Call library that is required to be linked with the application. Other libraries, including IP-specific libraries, are loaded automatically. See the *Global Call API Programming Guide* for more information on building applications.

6. IP-Specific Parameter Set Reference

The parameter set IDs and parameter IDs described in this chapter are defined in the *gcip.h* header file. Table 12 summarizes the parameter set IDs and parameter IDs used by Global Call in an IP environment.

The meaning of the columns in the table following are:

- **Set ID** - An identifier for a group of related parameters.
- **Parameter ID** - A identifier for a specific parameter.
- **Set** - Indicates the Global Call functions used to set the parameter information.
- **Send** - Indicates the Global Call functions used to send the information to a peer endpoint.
- **Retrieve** - Indicates the Global Call function used to retrieve information that was sent by a peer endpoint.
- **H.323/SIP** - Indicates if the parameter is supported when using H.323 or SIP.

Table 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
GCSET_ CALL_ CONFIG	GCPARM_ CALLPROC	gc_SetConfigData( )	---	---	H.323 and SIP
GCSET_ CHAN_ CAPABILITY	IPPARM_ LOCAL_ CAPABILITY	gc_SetConfigData( ) gc_SetUserInfo( )†	gc_AnswerCall( ) gc_MakeCall( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 and SIP
IPSET_ CALLINFO	IPPARM_ CALL DURATION	---	---	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 and 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 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
IPSET_ CALLINFO	IPPARM_ CALLID	---	---	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 only
IPSET_ CALLINFO	IPPARM_ CONN ECTION METHOD	gc_MakeCall( ) gc_SetUserInfo( )†	gc_AnswerCall( ) gc_MakeCall( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 and SIP
IPSET_ CALLINFO	IPPARM_ DISPLAY	gc_SetUserInfo( )† gc_MakeCall( )	gc_AnswerCall( ) gc_MakeCall( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 and SIP
IPSET_ CALLINFO	IPPARM_ H245 TUNNELING	gc_SetUserInfo( )† gc_MakeCall( ) gc_SetConfigData( ) ‡	gc_MakeCall( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323
IPSET_ CALLINFO	IPPARM_ PHONELIST	gc_SetUserInfo( )† gc_MakeCall( )	gc_MakeCall( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 and SIP
IPSET_ CALLINFO	IPPARM_ USERUSER_ INFO	gc_SetUserInfo( )† gc_MakeCall( )	gc_MakeCall( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 only
IPSET_CONF ERENCE	IPPARM_ CON FERENCE_ GOAL	gc_MakeCall( ) gc_SetUserInfo( )†	gc_AnswerCall( ) gc_MakeCall( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 only
IPSET_CONF ERENCE	IPPARM_ CON FERENCE_ ID	---	---	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 only

† 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 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
IPSET_ CONFIG	IPPARM_ CONFIG_TOS	gc_SetConfigData( ) gc_MakeCall( ) gc_SetUserInfo( )†	gc_AnswerCall( ) gc_MakeCall( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 and SIP
IPSET_ DTMF	IPPARM_ DTMF_ ALPHA NUMERIC	---	gc_Extension( ) (IPEXTID_ SEND_DTMF)	gc_Extension( ) (IPEXTID_ RECEIVE_ DTMF)	H.323 and SIP
IPSET_ DTMF	IPPARM_ DTMF_ RFC_2833	---	gc_Extension( ) (IPEXTID_ SEND_DTMF)	gc_Extension( ) (IPEXTID_ RECEIVE_ DTMF)	H.323 and SIP
IPSET_DTMF	IPPARM_ DTMF_ RFC2833_ PAYLOAD_ TYPE	gc_SetConfigData( ) gc_SetUserInfo( )†	---	---	H.323 and SIP
IPSET_DTMF	IPPARM_ SUPPORT_ DTMF_ BITMASK	gc_SetConfigData( ) gc_SetUserInfo( )†	---	---	H.323 and SIP
IPSET_ EXTENSION EVT_MSK	GCACT_ ADDMSK	gc_SetConfigData( )	---	---	H.323 and SIP
IPSET_ EXTENSION EVT_MSK	GCACT_ GET_MSK	gc_SetConfigData( )	---	---	H.323 and SIP
IPSET_ EXTENSION EVT_MSK	GCACT_ SETMSK	gc_SetConfigData( )	---	---	H.323 and 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 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
IPSET_ EXTENSION EVT_MSK	GCACT_ SUBMSK	gc_SetConfigData( )	---	---	H.323 and SIP
IPSET_IP PROTOCOL_ STATE	IPPARM_ CONTROL_ CONNECTED	---	---	GCEV_ EXTENSION (IPEXTID_ IPPROTOCOL _STATE)	H.323
IPSET_IP PROTOCOL_ STATE	IPPARM_ CONTROL_ DISCONN ECTED	---	---	GCEV_ EXTENSION (IPEXTID_ IPPROTOCOL _STATE)	H.323
IPSET_IP PROTOCOL_ STATE	IPPARM_ SIGNALING_ CONNECTED	---	---	GCEV_ EXTENSION (IPEXTID_ IPPROTOCOL _STATE)	H.323 only
IPSET_IP PROTOCOL_ STATE	IPPARM_ SIGNALING_ DISCONN ECTED	---	---	GCEV_ EXTENSION (IPEXTID_ IPPROTOCOL _STATE)	H.323 only
IPSET_ LOCAL_ ALIAS	IPPARM_ ADDRESS_ DOT_ NOTATION	---	gc_ReqService( )	---	H.323 Only
IPSET_ LOCAL_ ALIAS	IPPARM_ ADDRESS_ EMAIL	---	gc_ReqService( )	---	H.323 Only

† 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 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
IPSET_ LOCAL_ ALIAS	IPPARM_ ADDRESS_ H323_ID	---	gc_ReqService( )	---	H.323 Only
IPSET_ LOCAL_ ALIAS	IPPARM_ ADDRESS_ PHONE	---	gc_ReqService( )	---	H.323 Only
IPSET_ LOCAL_ ALIAS	IPPARM_ ADDRESS_ TRANS PARENT	---	gc_ReqService( )	---	H.323 Only
IPSET_ LOCAL_ ALIAS	IPPARM_ ADDRESS_ URL	---	gc_ReqService( )	---	H.323 Only
IPSET_ MEDIA_ STATE	IPPARM_ RX_ CONNECTED	---	---	GCEV_ EXTENSION (IPEXTID_ MEDIAINFO)	H.323 and SIP
IPSET_ MEDIA_ STATE	IPPARM_ RX_DIS CONNECTED	---	---	GCEV_ EXTENSION (IPEXTID_ MEDIAINFO)	H.323 and SIP
IPSET_ MEDIA_ STATE	IPPARM_ TX_ CONNECTED	---	---	GCEV_ EXTENSION (IPEXTID_ MEDIAINFO)	H.323 and SIP
IPSET_ MEDIA_ STATE	IPPARM_ TX_DIS CONNECTED	---	---	GCEV_ EXTENSION (IPEXTID_ MEDIAINFO)	H.323 and 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 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
IPSET_ MSG_H245	IPPARM_ MSGTYPE	---	gc_Extension( ) (IPEXTID_ SENDMSG)	GCEV_ EXTENSION (IPEXTID_ RECEIVE MSG)	H.323
IPSET_ MSG_Q931	IPPARM_ MSGTYPE	---	gc_Extension( ) (IPEXTID_ SENDMSG)	GCEV_ EXTENSION (IPEXTID_ RECEIVE MSG)	H.323
IPSET_MSG_ REGISTRA TION	IPPARM_ MSGTYPE	---	gc_Extension( ) (IPEXTID_ SENDMSG)	GCEV_ EXTENSION (IPEXTID_ RECEIVE MSG)	H.323 and SIP
IPSET_NON STANDARD CONTROL	IPPARM_ H221NON STANDARD	gc_SetConfigData( ) gc_MakeCall( ) gc_SetUserInfo( )†	gc_AnswerCall( ) gc_MakeCall( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323
IPSET_NON STANDARD CONTROL	IPPARM_ NON STANDARD DATA_DATA	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	gc_AnswerCall( ) gc_MakeCall( ) gc_DropCall( ) gc_ReqService( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 only
IPSET_NON STANDARD DATA	IPPARM_ H221NON STANDARD	gc_SetConfigData( ) gc_MakeCall( ) gc_SetUserInfo( )†	gc_AnswerCall( ) gc_MakeCall( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323
IPSET_NON STANDARD DATA	IPPARM_ NON STANDARD DATA_DATA	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	gc_AnswerCall( ) gc_MakeCall( ) gc_DropCall( ) gc_ReqService( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 only

† 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 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
IPSET_NON STANDARD DATA	IPPARM_ NON STANDARD DATA_ OBJID	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	gc_AnswerCall( ) gc_MakeCall( ) gc_DropCall( ) gc_ReqService( )	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 only
IPSET_ PROTOCOL	IPPARM_ PROTOCOL_ BITMASK	gc_SetConfigData( ) gc_SetUserInfo( )† gc_MakeCall( )	gc_ReqService( ) gc_MakeCall( )	---	H.323 and SIP
IPSET_REG_ INFO	IPPARM_ OPERATION_ DEREGISTER	---	gc_ReqService( )	---	H.323 and SIP
IPSET_REG_ INFO	IPPARM_ OPERATION_ REGISTER	---	gc_ReqService( )	---	H.323 and SIP
IPSET_REG_ INFO	IPPARM_ REG_ ADDRESS		gc_ReqService( )	---	H.323 and SIP
IPSET_REG_ INFO	IPPARM_ REG_ STATUS	---	---	Forwarded automatically in a GCEV_ SERVICE RESP event	H.323 and SIP
IPSET_ SUPPORTED_ PREFIXES	IPPARM_ ADDRESS_ DOT_ NOTATION	---	gc_ReqService( )	---	H.323 Only
IPSET_ SUPPORTED_ PREFIXES	IPPARM_ ADDRESS_ EMAIL	---	gc_ReqService( )	---	H.323 Only

† 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 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
IPSET_ SUPPORTED_ PREFIXES	IPPARM_ ADDRESS_ H323_ID	---	gc_ReqService( )	---	H.323 only
IPSET_ SUPPORTED_ PREFIXES	IPPARM_ ADDRESS_ PHONE	---	gc_ReqService( )	---	H.323 Only
IPSET_ SUPPORTED_ PREFIXES	IPPARM_ ADDRESS_ TRANS PARENT	---	gc_ReqService( )	---	H.323 Only
IPSET_ SUPPORTED_ PREFIXES	IPPARM_ ADDRESS_ URL	---	gc_ReqService( )	---	H.323 Only
IPSET_T38_ TONEDET	IPPARM_ T38DET_ CED	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_T38_ TONEDET	IPPARM_ T38DET_ CNG	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_T38_ TONEDET	IPPARM_ T38DET_ V21	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38CAP FRAME STATUS	IPPARM_ T38CAP FRAME_ RX_CTC	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and 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 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
IPSET_ T38CAP FRAME STATUS	IPPARM_ T38CAP FRAME_ RX_DCS	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38CAP FRAME STATUS	IPPARM_ T38CAP FRAME_ RX_DIX_DTC	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38CAP FRAME STATUS	IPPARM_ T38CAP FRAME_ TX_CTC	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38CAP FRAME STATUS	IPPARM_ T38CAP FRAME_ TX_DCS	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38CAP FRAME STATUS	IPPARM_ T38CAP FRAME_ TX_DIS_DTC	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38HLDC FRAME STATUS	IPPARM_ T38HLDC FRAME_ RX	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38HLDC FRAME STATUS	IPPARM_ T38HLDC FRAME_ TX	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ RX_CSI	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and 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 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ RX_PWD	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ RX_SEP	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ RX_SIG	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ RX_SUB	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ RX_TSI	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ TX_CSI	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ TX_PWD	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ TX_SEP	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and 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 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ TX_SIG	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ TX_SUB	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ T38INFO FRAME STATUS	IPPARM_ T38INFO FRAME_ TX_TSI	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_TDM_ TONEDET	IPPARM_ TDMDET_ CED	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_TDM_ TONEDET	IPPARM_ TDMDET_ CNG	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_TDM_ TONEDET	IPPARM_ TDMDET_ V21	---	---	GCEV_ EXTENSION (IPEXTID_ FOIP)	H.323 and SIP
IPSET_ TRANSACT ION	IPPARM_ TRANSACT ION_ID			gc_Extension( ) (Any ext_id)	H.323 and SIP
IPSET_ USERINPUT INDICATION	IPPARM_ UII_ ALPHA NUMERIC	---	gc_Extension( ) (IPEXTID_ SENDMSG)	gc_Extension( ) (IPEXTID_ SENDMSG)	H.323 only

† 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 12. Summary of Parameter IDs and Set IDs

Set ID	Parameter ID	Set	Send	Retrieve	H323 /SIP
IPSET_ VENDOR INFO	IPPARM_ H221 NONSTD	gc_SetConfigData( )	gc_Extension( ) (IPEXTID_ SENDMSG)	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 only
IPSET_ VENDOR INFO	IPPARM_ VENDOR_ PRODUCT_ID	gc_SetConfigData( )	gc_Extension( ) (IPEXTID_ SENDMSG)	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 only
IPSET_ VENDOR INFO	IPPARM_ VENDOR_ VERSION_ID	gc_SetConfigData( )	gc_Extension( ) (IPEXTID_ SENDMSG)	gc_Extension( ) (IPEXTID_ GETINFO)	H.323 only

† 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.

6.1. GCSET_CALL_CONFIG Parameter Set

Table 13 shows the parameter IDs in the GCSET_CALL_CONFIG parameter set that are relevant in an IP context.

Table 13. GCSET_CALL_CONFIG Parameter Set

Parameter ID	Type	Description
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.	Used to specify if the Proceeding message is sent under application control or automatically by the stack.

6.2. GCSET_CHAN_CAPABILITY Parameter Set

Table 14 shows the parameter IDs in the GCSET_CHAN_CAPABILITY parameter set.

Table 14. GCSET_CHAN_CAPABILITY Parameter Set

Parameter ID	Description	Type
IPPARM_LOCAL_CAPABILITY	Used to pass the local capabilities to be associated with a line device, or a subset thereof when used to communicate selected capabilities for a call. It can also be used to retrieve capabilities when a call is connected.	IP_CAPABILITY

6.3. IPSET_CALLINFO Parameter Set

Table 15 shows the parameter IDs in the IPSET_CALLINFO parameter set.

Table 15. IPSET_CALLINFO Parameter Set

Parameter ID	Type	Description
IPPARM_CALLID	String, max. length = IP_CALLID_LENGTH (16 octets)	The call ID. **Note**: In SIP, setting the call ID is not supported.
IPPARM_CALLDURATION	unsigned long	Duration of the call.
IPPARM_CONNECTIONMETHOD	Enumeration, with one of the following values: • IPPARM_CONNECTION METHOD_FASTSTART • IPPARM_CONNECTION METHOD_SLOWSTART	The connection method: Fast Start or Slow Start. See Section 4.2, “Using Fast Start and Slow Start Setup”, on page 78 for more information.
IPPARM_DISPLAY	String, max. length = MAX_DISPLAY_LENGTH (82), null-terminated.	Display information. This information can be used by a peer as additional address information.

Table 15. IPSET_CALLINFO Parameter Set

Parameter ID	Type	Description
IPPARM_H245TUNNELING	Enumeration, with one of the following values: • IP_H245TUNNELING_ON • IP_H245TUNNELING_OFF	Specify if tunneling is on or off. See Section 4.16, "Enabling and Disabling Tunneling in H.323", on page 142 for more information.
IPPARM_PHONELIST	String, 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 3.4.11, "gc_MakeCall( )", on page 42 for more information.
IPPARM_USERUSER_INFO	Uint8[ ], max size = MAX_USERUSER_INFO_ LENGTH (131)	User-to-user information.
Notes: For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1. See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.4. IPSET_CONFERENCE Parameter Set

Table 16 shows the parameter IDs in the IPSET_CONFERENCE parameter set.

Table 16. IPSET_CONFERENCE Parameter Set

Parameter ID	Type	Description
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	The conference functionality to be achieved.
IPPARM_CONFERENCE_ID	String, max. length = IP_CONFERENCE_ID_LENGTH (16).	The conference identifier.

Note: For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1.
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.
Note: 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 6.3, "IPSET_CALLINFO Parameter Set", on page 159 for more information.

6.5. IPSET_CONFIG Parameter Set

Table 17 shows the parameter IDs in the IPSET_CONFIG parameter set.

Table 17. IPSET_CONFIG Parameter Set

Parameter ID	Type	Description
IPPARM_CONFIG_TOS	Uint8	Set the Type of Service (TOS) byte. Valid values are in the range 0 to 255. The default value is 0.
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.6. IPSET_DTMF Parameter Set

Table 18 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.

Table 18. IPSET_DTMF Parameter Set

Parameter IDs	Type	Description
IPPARM_DTMF_ALPHANUMERIC	int	Used when sending or receiving DTMF via UII alphanumeric messages. The parameter value contains an IP_DTMF_DIGITS structure that includes the digit string.
IPPARM_DTMF_RFC_2833	int	Used when sending or receiving DTMF via RFC2833 messages. The parameter value contains an IP_DTMF_RFC2833 structure that includes the digit string.
IPPARM_DTMF_RFC2833_PAYLOAD_TYPE	int	Used to specify the RFC2833 RTP payload type. The data field is an unsigned char with a valid range of 96 to 127. The default value is IP_USE_STANDARD_PAYLOADTYPE (101).

Table 18. IPSET_DTMF Parameter Set

Parameter IDs	Type	Description
IPPARM_SUPPORT_DTMF_BITMASK	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
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability. † Not supported when using SIP.		

NOTE: A User Input Indication (UII) messages (H.245) can also be used to send DTMF digits in an information signal message. See Section 6.24, "IPSET_USERINPUTINDICATION Parameter Set", on page 175 for more information.

6.7. 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 20 shows the parameter IDs in the IPSET_EXTENSIONEVT_MSK parameter set.

Table 19. IPSET_EXTENSIONEVT_MSK Parameter Set

Parameter IDs	Type	Description
GCPARM_GET_MSK	int	Retrieve the bitmask of enabled events.
GCACT_SETMSK	int	Set the bitmask of enabled events.
GCACT_ADDMSK	int	Add to the bitmask of enabled events.
GCACT_SUBMSK	int	Remove from the bitmask of enabled events.

Table 19. IPSET_EXTENSIONEVT_MSK Parameter Set

Parameter IDs	Type	Description
Values that can be used to make up the bitmask are: • EXTENSIONEVT_DTMF_RFC2833 (0x02) • EXTENSIONEVT_DTMF_ALPHANUMERIC (0x04) † • EXTENSIONEVT_SIGNALING_STATUS (0x08) • EXTENSIONEVT_STREAMING_STATUS (0x10) • EXTENSIONEVT_T38_STATUS (0x20)		
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability. † Not supported for SIP.		

6.8. IPSET_IPPROTOCOL_STATE Parameter Set

This parameter set is used when retrieving notification of protocol signaling states via GCEV_EXTENSION events. Table 20 shows the parameter IDs in the IPSET_IPPROTOCOL_STATE parameter set.

Table 20. IPSET_IPPROTOCOL_STATE Parameter Set

Parameter IDs	Type	Description
IPPARM_SIGNALING_CONNECTED	int	Call signaling for the call has been established with the remote endpoint.
IPPARM_SIGNALING_DISCONNECTED	int	Call signaling for the call has been terminated.
IPPARM_CONTROL_CONNECTED	int	Media control signaling for the call has been established with the remote endpoint.
IPPARM_CONTROL_DISCONNECTED	int	Media control signaling for the call has been terminated.
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.9. IPSET_LOCAL_ALIAS Parameter Set

Table 22 shows the parameter IDs in the IPSET_LOCAL_ALIAS parameter set.

Table 21. IPSET_LOCAL_ALIAS Parameter Set

Parameter IDs	Type	Description
IPPARM_ADDRESS_DOT_NOTATION	String	A valid IP address.
IPPARM_ADDRESS_EMAIL	String	Email address composed of characters from the set "[A-Z][a-z][0-9]_-.@".
IPPARM_ADDRESS_H323_ID	String	A valid H.323 ID.
IPPARM_ADDRESS_PHONE	String	An E.164 telephone number.
IPPARM_ADDRESS_TRANSPARENT	String	Unspecified address type.
IPPARM_ADDRESS_URL	String	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 "-".
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.10. IPSET_MEDIA_STATE Parameter Set

Table 22 shows the parameter IDs in the IPSET_MEDIA_STATE parameter set.

Table 22. IPSET_MEDIA_STATE Parameter Set

Parameter IDs	Type	Description
IPPARM_RX_CONNECTED	int	Streaming has been initiated in the receive direction from the remote endpoint. The datatype of this parameter is IP_CAPABILITY which includes coder information negotiated with the remote peer. See Section 4.14, "Enabling and Disabling Unsolicited Notification Events", on page 140 for more information.

Table 22. IPSET_MEDIA_STATE Parameter Set

Parameter IDs	Type	Description
IPPARM_RX_DISCONNECTED	int	Streaming in the receive direction from the remote endpoint has been terminated.
IPPARM_TX_CONNECTED	int	Streaming has been initiated in the transmit direction toward the remote endpoint. The datatype of this parameter is IP_CAPABILITY which includes coder information negotiated with the remote peer. See Section 4.14, "Enabling and Disabling Unsolicited Notification Events", on page 140 for more information.
IPPARM_TX_DISCONNECTED	int	Streaming in the transmit direction toward the remote endpoint has been terminated.
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.11. IPSET_MSG_H245 Parameter Set

Table 23 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 23. IPSET_MSG_H245 Parameter Set

Parameter IDs	Type	Description
IPPARM_MSGTYPE	int	Possible values for H.245 messages are: • IP_MSGTYPE_H245_INDICATION
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.12. IPSET_MSG_Q931 Parameter Set

Table 24 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 an Q.931 message.

Table 24. IPSET_MSG_Q931 Parameter Set

Parameter IDs	Type	Description
IPPARM_MSGTYPE	int	Possible values for Q.931 messages are: • IP_MSGTYPE_Q931_FACILITY
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.13. IPSET_MSG_REGISTRATION Parameter Set

Table 25 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.

Table 25. IPSET_MSG_REGISTRATION Parameter Set

Parameter IDs	Type	Description
IPPARM_MSGTYPE	int	Possible value for registration messages is: • IP_MSGTYPE_REG_NONSTD
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.14. IPSET_NONSTANDARDDATA Parameter Set

Table 26 shows the parameter IDs in the IPSET_NONSTANDARDDATA parameter set.

Table 26. IPSET_NONSTANDARDDATA Parameter Set

Parameter IDs	Type	Description
IPPARM_NONSTANDARDDATA_DATA	String, max. length = MAX_NS_PARM_DATA_LENGTH (128)	Contains the nonstandard data supplied, if any. If nonstandard data was not supplied, this parameter should not be present in the parm block.
IPPARM_NONSTANDARDDATA_OBJID	Uint[ ], max. length = MAX_NS_PARM_OBJID_LENGTH (40)	Contains the nonstandard object ID supplied, if any. If a nonstandard object ID was not provided, this parameter should not be present in the parm block.
IPPARM_H221NONSTANDARD	String	Contains an H.221 nonstandard data identifier.
Note: For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1. See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.15. IPSET_NONSTANDARDCONTROL Parameter Set

Table 27 shows the parameter IDs in the IPSET_NONSTANDARDCONTROL parameter set.

Table 27. IPSET_NONSTANDARDCONTROL Parameter Set

Parameter IDs	Type	Description
IPPARM_NONSTANDARDDATA_DATA	String, max. length = MAX_NS_PARM_DATA_LENGTH (128)	Contains the nonstandard data supplied, if any. If nonstandard data was not supplied, this parameter should not be present in the parm block.

Table 27. IPSET_NONSTANDARDCONTROL Parameter Set

Parameter IDs	Type	Description
IPPARM_NONSTANDARDDATA_OBJID	Uint[ ], max. length = MAX_NS_ PARM_OBJID_ LENGTH (40)	Contains the nonstandard object ID supplied, if any. If a nonstandard object ID was not provided, this parameter should not be present in the parm block.
PPARM_H221NONSTANDARD	String	Contains an H.221 nonstandard data identifier.

Note: For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1.
See Table 12, “Summary of Parameter IDs and Set IDs”, on page 147 for H.323 and SIP applicability.

6.16. IPSET_PROTOCOL Parameter Set

Table 28 shows the parameter IDs in the IPSET_PROTOCOL parameter set.

Table 28. IPSET_PROTOCOL Parameter Set

Parameter IDs	Type	Description
IPPARM_PROTOCOL_BITMASK	String	The IP protocol to use. Possible values are: • IP_PROTOCOL_H323 • IP_PROTOCOL_SIP

6.17. IPSET_REG_INFO Parameter Set

Table 29 shows the parameter IDs in the IPSET_REG_INFO parameter set.

Table 29. IPSET_REG_INFO Parameter Set

Parameter IDs	Type	Description
IPPARM_OPERATION_REGISTER	char	Used to manipulate registration information when registering an endpoint with a gatekeeper/registrar. Possible values are: • IP_REG_ADD_INFO • IP_REG_DELETE_BY_VALUE • IP_REG_SET_INFO
IPPARM_OPERATION_DEREGISTER	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.
IPPARM_REG_ADDRESS	IP_REGISTER_ADDRESS. See Section 7.9, "IP_REGISTER_ADDRESS", on page 185 for more information.	Address information to be registered with a gatekeeper/registrar.
IPPARM_REG_STATUS	int	Provides an indication of whether the endpoint registration with a gatekeeper/registrar was successful or not. Possible values are: • IP_REG_CONFIRMED • IP_REG_REJECTED
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.18. IPSET_SUPPORTED_PREFIXES Parameter Set

Table 30 shows the parameter IDs in the IPSET_SUPPORTED_PREFIXES parameter set.

Table 30. IPSET_SUPPORTED_PREFIXES Parameter Set

Parameter IDs	Type	Description
IPPARM_ADDRESS_DOT_NOTATION	String	A valid IP address.
IPPARM_ADDRESS_EMAIL	String	Email address composed of characters from the set "[A-Z][a-z][0-9]_-.@"
IPPARM_ADDRESS_H323_ID	String	A valid H.323 ID.
IPPARM_ADDRESS_PHONE	String	An E.164 telephone number
IPPARM_ADDRESS_TRANSPARENT	String	Unspecified address type.
IPPARM_ADDRESS_URL	String	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 "-".
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.19. IPSET_TDM_TONEDET Parameter Set

Table 31 shows the parameter IDs in the IPSET_TDM_TONEDET parameter set.

Table 31. IPSET_TDM_TONEDET Parameter Set

Parameter IDs	Type	Description
IPPARM_TDMDET_CED	int	Indicates Called Terminal Identification (CED) tone detection on the TDM side.
IPPARM_TDMDET_CNG	int	Indicates Calling Tone (CNG) detection on the TDM side.
IPPARM_TDMDET_V21	int	Indicates V21 tone detection on the TDM side.

Table 31. IPSET_TDM_TONEDET Parameter Set

Parameter IDs	Type	Description
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.20. IPSET_T38_TONEDET Parameter Set

Table 32 shows the parameter IDs in the IPSET_T38_TONEDET parameter set.

Table 32. IPSET_T38_TONEDET Parameter Set

Parameter IDs	Type	Description
IPPARM_T38DET_CED	int	Indicates Called Terminal Identification (CED) tone detection on the IP.
IPPARM_T38DET_CNG	int	Indicates Calling Tone (CNG) detection on the IP side.
IPPARM_T38DET_V21	int	Indicates V21 tone detection on the IP side.
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.21. IPSET_T38CAPFRAMESTATUS Parameter Set

Table 33 shows the parameter IDs in the IPSET_T38CAPFRAMESTATUS parameter set. These parameters correspond to commands described in the *ITU T.30 Standard*.

Table 33. IPSET_T38CAPFRAMESTATUS Parameter Set

Parameter IDs	Type	Description
IPPARM_T38CAPFRAME_TX_DIS_DTC	int	Digital Transmit Command (DTC) – The digital command response to the standard capabilities identified by the DIS† signal.
IPPARM_T38CAPFRAME_TX_DCS	int	Digital Command Signal (DCS) – The digital set-up command responding to the standard capabilities identified by the DIS† signal.
IPPARM_T38CAPFRAME_TX_CTC	int	Continue To Correct (CTC) – This digital command is only used in the optional T.4 error correction mode.
IPPARM_T38CAPFRAME_RX_DIX_DTC	int	Digital Transmit Command (DTC) – The digital command response to the standard capabilities identified by the DIS† signal.
IPPARM_T38CAPFRAME_RX_DCS	int	Digital Command Signal (DCS) – The digital set-up command responding to the standard capabilities identified by the DIS† signal.
IPPARM_T38CAPFRAME_RX_CTC	int	Continue To Correct (CTC) – This digital command is only used in the optional T.4 error correction mode.

See Table 12, “Summary of Parameter IDs and Set IDs”, on page 147 for H.323 and SIP applicability.

† The Digital Identification Signal (DIS) characterizes the standard ITU-T capabilities of the called terminal.

6.22. IPSET_T38HDLCFRAMESTATUS Parameter Set

Table 34 shows the parameter IDs in the IPSET_T38HDLCFRAMESTATUS parameter set.

Table 34. IPSET_T38HDLCFRAMESTATUS Parameter Set

Parameter IDs	Type	Description
IPPARM_T38HDLCFRAME_TX	int	T.38 HDLC transmit frame.

Table 34. IPSET_T38HDLCFRAMESTATUS Parameter Set

Parameter IDs	Type	Description
IPPARM_T38HDLCFRAME_RX	int	T.38 HDLC receive frame.
See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.23. IPSET_T38INFOFRAMESTATUS Parameter Set

Table 35 shows the parameter IDs in the IPSET_T38INFOFRAMESTATUS parameter set. These parameters correspond to sections in the Facsimile Information Field (FIF) as described in the *ITU T.30 Standard*.

Table 35. IPSET_T38INFOFRAMESTATUS Parameter Set

Parameter IDs	Type	Description
IPPARM_T38INFOFRAME_TX_SUB	int	Subaddress (SUB) - A subaddress in the called subscriber's domain. Used to provide additional routing information in the facsimile procedure.
IPPARM_T38INFOFRAME_RX_SUB	int	Subaddress (SUB) - A subaddress in the called subscriber's domain. Used to provide additional routing information in the facsimile procedure.
IPPARM_T38INFOFRAME_TX_SEP	int	Selective Polling (SEP) - A subaddress for the polling mode that may be used to indicate if a specific document will be polled at the called terminal.
IPPARM_T38INFOFRAME_RX_SEP	int	Selective Polling (SEP) - A subaddress for the polling mode that may be used to indicate if a specific document will be polled at the called terminal.
IPPARM_T38INFOFRAME_TX_PWD	int	Passwrod (PWD) - A password for the polling mode that may be used to provide additional security to the facsimile procedure.

Table 35. IPSET_T38INFOFRAMESTATUS Parameter Set

Parameter IDs	Type	Description
IPPARM_T38INFOFRAME_RX_PWD	int	Password (PWD) - A password for the polling mode that may be used to provide additional security to the facsimile procedure.
IPPARM_T38INFOFRAME_TX_TSI	int	Transmitting Subscriber Identification (TSI) - The identification of the transmitting terminal that may be used to provide additional security to the facsimile procedures.
IPPARM_T38INFOFRAME_RX_TSI	int	Transmitting Subscriber Identification (TSI) - The identification of the transmitting terminal that may be used to provide additional security to the facsimile procedures.
IPPARM_T38INFOFRAME_TX_CSI	int	Called Subscriber Identification (CSI) - Identifies the called subscriber by its international telephone number.
IPPARM_T38INFOFRAME_RX_CSI	int	Called Subscriber Identification (CSI) - Identifies the called subscriber by its international telephone number.
IPPARM_T38INFOFRAME_TX_CIG	int	Calling Subscriber Identification (CIG) - Identifies the calling terminal and may be used to provide additional security to the facsimile procedure.
IPPARM_T38INFOFRAME_RX_CIG	int	Calling Subscriber Identification (CIG) - Identifies the calling terminal and may be used to provide additional security to the facsimile procedure.
See Table 12, “Summary of Parameter IDs and Set IDs”, on page 147 for H.323 and SIP applicability.		

6.24. IPSET_USERINPUTINDICATION Parameter Set

Table 36 shows the parameter IDs in the IPSET_USERINPUTINDICATION parameter set.

Table 36. IPSET_USERINPUTINDICATION Parameter Set

Parameter IDs	Type	Description
IPPARM_UII_ALPHANUMERIC	String	Used to send user input indication (UII) messages. Can be used to: • Send DTMF digits • Send message information that a peer can retrieve by examining the structure associated with the GCEV_EXTENSION event generated at the peer.
Note: For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1. See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

6.25. IPSET_VENDORINFO Parameter Set

Table 37 shows the parameter IDs in the IPSET_VENDORINFO parameter set.

Table 37. IPSET_VENDORINFO Parameter Set

Parameter IDs	Type	Description
IPPARM_H221NONSTD	IP_H221NONSTANDARD. See Section 7.8, "IP_H221NONSTANDARD", on page 185 for more information.	Contains country code, extension code and manufacturer code.
IPPARM_VENDOR_ PRODUCT_ID	String, max. length = MAX_PRODUCT_ID_ LENGTH (32).	Vendor product identifier.
IPPARM_VENDOR_ VERSION_ID	String, max. length = MAX_VERSION_ID_ LENGTH (32).	Vendor version identifier.

Table 37. IPSET_VENDORINFO Parameter Set

Parameter IDs	Type	Description
Note: For parameter IDs of type String, the length of the string when used in a GC_PARM_BLK is the length of the string plus 1. See Table 12, "Summary of Parameter IDs and Set IDs", on page 147 for H.323 and SIP applicability.		

7. Data Structure Reference

This chapter describes data structures that are specific to IP technology. These data structures are defined in the *gcip.h* header file.

7.1. IPADDR

The IPADDR structure is used to specify a local IP address. The data structure definition is as follows:

```
typedef struct
{
    unsigned char     ip_ver;
    union{
            unsigned int     ipv4;
            unsigned int     ipv6[4]
         }u_ipaddr;
}IPADDR, *PIPADDR;
```

Table 38 describes the meaning of each field in the IPCCLIB_START_DATA data structure.

Table 38. IPADDR Field Descriptions

Field	Description
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 more information on the byte order of IPv4 addresses, see RFC 791 and RFC 792.

7.2. IPCCLIB_START_DATA

The IPCCLIB_START_DATA structure is used to configure the IP H.323/SIP call control library when starting Global Call. The data structure definition is as follows:

```
typedef struct
{
    unsigned short    version;
    unsigned char     delimiter;
    unsigned char     num_boards;
    IP_VIRTBOARD      *board_info;
} IPCCLIB_START_DATA;
```

Table 39 describes the meaning of each field in the IPCCLIB_START_DATA data structure.

Table 39. IPCCLIB_START_DATA Field Descriptions

Field	Description
version	The version of the start structure. Set to 0x100.
delimiter	An ANSI character used to change the default address string delimiter, that is, ",". The delimiter is used to separate the components of the destination information when using **gc_MakeCall( )** for example.
num_boards	The number of IPT board devices. See Section 2.2.1, "IPT Board Devices", on page 24 for more information on IPT board devices. the maximum value is 8.
board_info	Information associated with each board. See Section 7.11, "IP_VIRTBOARD", on page 187 for more information.

7.3. IP_AUDIO_CAPABILITY

The IP_AUDIO_CAPABILITY data structure is used to allow some minimum set of information to be exchanged together with the audio codec identifier. The data structure definition is as follows:

```
typedef struct
{
    unsigned long   frames_per_pkt;
    long            VAD;
} IP_AUDIO_CAPABILITY;
```

Table 40 describes the meaning of each field in the IP_AUDIO_CAPABILITY data structure.

Table 40. IP_AUDIO_CAPABILITY Field Descriptions

Field	Description
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 4.4.4, “Setting Coder Information”, on page 85 for more information.
VAD	Applies to audio algorithms that support the concept of voice activated detection (VAD) only. Possible values are: • GCPV_ENABLE - VAD enabled • GCPV_DISABLE - VAD disabled This field can also be set to GCCAP_dontCare to indicate that any supported value is valid.

7.4. IP_CAPABILITY

The IP_CAPABILITY data structure provides a level of capability information in addition to simply the capability or codec identifier.

NOTE: The IP_CAPABILITY data structure is not intended to provide all the flexibility of the H.245 terminal capability structure, but provides a first level of useful information in addition to the capability or codec identifier.

The data structure definition is as follows.

```
typedef struct
{
    int                     capability;
    int                     type;
    int                     direction;
    int                     payload_type;
    IP_CAPABILITY_UNION     extra;
    char                    rfu[0x10];
} IP_CAPABILITY;
```

Table 41 describes the meaning of each field in the IP_CAPABILITY data structure.

Table 41. IP_CAPABILITY Field Descriptions

Field	Description
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_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. Possible values are: • GCCAPTYPE_AUDIO - Audio • GCCAPTYPE_RDATA - Data
direction	The capability direction code for this capability. Possible values are: • IP_CAP_DIR_LCLTRANSMIT - Indicates a transmit capability for the local endpoint. • IP_CAP_DIR_LCLRECIEVE - Indicates a receive capability for the local endpoint. • IP_CAP_DIR_LCLRXTX - Indicates a receive and transmit capability for the local endpoint. Supported for T.38 only.

Table 41. IP_CAPABILITY Field Descriptions (Continued)

Field	Description
payload_type	The payload type. When using a standard payload type, set the value of this field to IP_USE_STANDARD_PAYLOADTYPE. When using a nonstandard payload type, use this field to specify the RTP payload type that will be used in conjunction with the coder specified in the capability field in this structure. Not currently supported.
extra	The contents of the IP_CAPABILITY_UNION will be indicated by the type field.
rfu	Reserved for future use. Must be set to zero when not used.

7.5. IP_DATA_CAPABILITY

The IP_DATA_CAPABILITY data structure provides additional information about the data capability.

The data structure definition is as follows.

```
typedef struct
{
   int     max_bit_rate;
} IP_DATA_CAPABILITY;
```

Table 42 describes the meaning of each field in the IP_DATA_CAPABILITY data structure.

Table 42. IP_DATA_CAPABILITY Field Descriptions

Field	Description
max_bit_rate	Possible values are: • 2400 • 4800 • 9600 • 14400 The recommended value for T.38 coders is 14400.

7.6. IP_CAPABILITY_UNION

The IP_CAPABILITY_UNION union enables different capability categories to define their own additional parameters or interest. The union definition is as follows:

```
typedef union
{
    IP_AUDIO_CAPABILITY         audio;
    IP_VIDEO_CAPABILITY         video;
    IP_DATA_CAPABILITY          data;
} IP_CAPABILITY_UNION;
```

Table 43 describes the meaning of each field in the IP_CAPABILITY_UNION union.

Table 43. IP_CAPABILITY_UNION Field Descriptions

Field	Description
audio	A structure that represents the audio capability. See Section 7.3, "IP_AUDIO_CAPABILITY", on page 180 for more information.
video	Not supported.
data	Not supported.

7.7. IP_DTMF_DIGITS

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. The data structure definition is as follows:

```
typedef struct
{
    char          digit_buf[IP_MAX_DTMF_DIGITS];
    unsigned int  num_digits;
} IP_DTMF_DIGITS;
```

Table 44 describes the meaning of each field in the IP_DTMF_DIGITS data structure.

Table 44. IP_DTMF_DIGITS Field Descriptions

Field	Description
digit_buf	The DTMF digit string buffer; 32 characters in size.
num_digits	The number of DTMF digits in the string buffer.

7.8. IP_H221NONSTANDARD

The IP_H221NONSTANDARD data structure is used to store H.221 nonstandard data. The data structure definition is as follows:

```
typedef struct
{
    int   country_code;
    int   extension;
    int   manufacturer_code;
} IP_H221NONSTANDARD;
```

Table 45 describes the meaning of each field in the IP_H221NONSTANDARD data structure.

7.9. IP_REGISTER_ADDRESS

The IP_REGISTER_ADDRESS data structure is used to store registration information. The data structure definition is as follows:

```
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;
```

Table 45. IP_REGISTER_ADDRESS Field Descriptions

Field	Description
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 unicast time to live value in seconds. The number of seconds for which a registration is considered to be valid when repetitive registration is selected.
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.

7.10. IP_RFC2833_EVENT

The IP_RFC2833_EVENT data structure is used to provide information about events associated with DTMF digit detection. The application receives an event to mark the beginning of a DTMF digit, then receives another event to mark the end of the DTMF digit. See Section 4.6.2, "Getting Notification of DTMF Detection", on page 107 for more information. The data structure definition is as follows:

```
typedef struct {
    unsigned char  event;
    unsigned char  state;
} IP_RFC2833_EVENT;
```

Table 46 describes the meaning of each field in the IP_RFC2833_EVENT data structure.

Table 46. IP_RFC2833_EVENT Field Descriptions

Field	Description
event	The type of the event as specified in RFC 2833.
state	The state indicated by the event. Possible values are: • GCPV_DISABLE • GCPV_ENABLE

7.11. IP_VIRTBOARD

The IP_VIRTBOARD data structure is used to store information about an IPT board device. The data structure definition is as follows:

```
typedef struct
{
    unsigned short    version;
    unsigned int      total_max_calls;
    unsigned int      h323_max_calls;
    unsigned int      sip_max_calls;
    IPADDR            localIP;
    unsigned short    h323_signaling_port;
    unsigned short    sip_signaling_port;
    void              *reserved;
}IP_VIRTBOARD;
```

Table 47 describes the meaning of each field in the IP_VIRTBOARD data structure.

Table 47. IP_VIRTBOARD Field Descriptions

Field	Description
version	The version of the structure. Set to 0x100.
total_max_calls	The maximum total number of IPT devices that can be open concurrently. Possible values are in the range 1 to 2016 (IP_CFG_MAX_AVAILABLE_CALLS). Each IPT device can support both the H.323 and SIP protocols.

Table 47. IP_VIRTBOARD Field Descriptions

Field	Description
h323_max_calls	The maximum number of IPT devices used for H.323 calls. Possible values are in the range 1 to 2016 (IP_CFG_MAX_AVAILABLE_CALLS).
sip_max_calls	The maximum number of IPT devices used for SIP calls. Possible values are in the range 1 to 2016 (IP_CFG_MAX_AVAILABLE_CALLS).
localIP	The local IP address of type IPADDR. See Section 7.1, "IPADDR", on page 179.
h323_signaling_port	The H.323 call signaling port. Possible values are the port number or IP_CFG_DEFAULT. For H.323, the default port is 1720.
sip_signaling_port	The SIP call signaling port. Possible values are the port number or IP_CFG_DEFAULT. For SIP, the default port is 5060.
reserved	Reserved for future use. Must be set to NULL.

8. IP-Specific Event Cause Codes

The event cause 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. Table 48 lists the supported IP-specific event cause codes and provides a description of each code.

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, the following behavior is defined:

- 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 Global Call cause values for dropping a call (see the **gc_DropCall( )** function description in the *Global Call API Library Reference*), or one of the cause values prefixed by IPEC_SIP in Table 48.
- If the application wants to drop a call that is already connected (simply hanging up normally) the same rules as described in the first bullet apply, but the cause is not relevant in the BYE message.

Table 48. IP-Specific Event Cause Codes

Event Cause Code	Description
IPEC_H2250ReasonAdaptiveBusy	H2250 Reason.
IPEC_H2250ReasonBadFormatAddress	H2250 Reason.
IPEC_H2250ReasonCalledPartyNotRegistered	H2250 Reason.
IPEC_H2250ReasonCallerNotRegistered	H2250 Reason.
IPEC_H2250ReasonDestinationRejection	H2250 Reason.
IPEC_H2250ReasonFacilityCallDeflection	H2250 Reason.
IPEC_H2250ReasonGatekeeperResource	H2250 Reason.
IPEC_H2250ReasonGatewayResource	H2250 Reason.

Table 48. IP-Specific Event Cause Codes (Continued)

Event Cause Code	Description
IPEC_H2250ReasonInConf	H2250 Reason.
IPEC_H2250ReasonInvalidRevision	H2250 Reason.
IPEC_H2250ReasonNoBandwidth	H2250 Reason.
IPEC_H2250ReasonNoPermission	H2250 Reason.
IPEC_H2250ReasonSecurityDenied	H2250 Reason.
IPEC_H2250ReasonUndefinedReason	H2250 Reason.
IPEC_H2250ReasonUnreachableDestination	H2250 Reason.
IPEC_H2250ReasonUnreachableGatekeeper	H2250 Reason.
IPEC_Q931Cause01UnassignedNumber	Q.931 cause 01.
IPEC_Q931Cause02NoRouteToSpecifiedTransit Network	Q.931 cause 02.
IPEC_Q931Cause03NoRouteToDestination	Q.931 cause 03.
IPEC_Q931Cause06ChannelUnacceptable	Q.931 cause 06.
IPEC_Q931Cause07CallAwardedAndBeing DeliveredInAnEstablishedChannel	Q.931 cause 07.
IPEC_Q931Cause16NormalCallClearing	Q.931 cause 16.
IPEC_Q931Cause17UserBusy	Q.931 cause 17.
IPEC_Q931Cause18NoUserResponding	Q.931 cause 18.
IPEC_Q931Cause19UserAlertingNoAnswer	Q.931 cause 19.
IPEC_Q931Cause21CallRejected	Q.931 cause 21.
IPEC_Q931Cause22NumberChanged	Q.931 cause 22.
IPEC_Q931Cause26NonSelectUserClearing	Q.931 cause 26.
IPEC_Q931Cause27DestinationOutOfOrder	Q.931 cause 27.
IPEC_Q931Cause28InvalidNumberFormat IncompleteNumber	Q.931 cause 28.
IPEC_Q931Cause29FacilityRejected	Q.931 cause 29.
IPEC_Q931Cause30ResponseTo STATUSENQUIRY	Q.931 cause 30.

Table 48. IP-Specific Event Cause Codes (Continued)

Event Cause Code	Description
IPEC_Q931Cause31NormalUnspecified	Q.931 cause 31.
IPEC_Q931Cause34NoCircuitChannelAvailable	Q.931 cause 34.
IPEC_Q931Cause38NetworkOutOfOrder	Q.931 cause 38.
IPEC_Q931Cause41TemporaryFailure	Q.931 cause 41.
IPEC_Q931Cause42SwitchingEquipment Congestion	Q.931 cause 42.
IPEC_Q931Cause43AccessInformationDiscarded	Q.931 cause 43.
IPEC_Q931Cause44RequestedCircuitChannel NotAvailable	Q.931 cause 44.
IPEC_Q931Cause47ResourceUnavailable Unspecified	Q.931 cause 47.
IPEC_Q931Cause57BearerCapability NotAuthorized	Q.931 cause 57.
IPEC_Q931Cause58BearerCapability NotPresentlyAvailable	Q.931 cause 58.
IPEC_Q931Cause63ServiceOrOption NotAvailableUnspecified	Q.931 cause 63.
IPEC_Q931Cause65BearCapability NotImplemented	Q.931 cause 65.
IPEC_Q931Cause66ChannelType NotImplemented	Q.931 cause 66.
IPEC_Q931Cause69RequestedFacility NotImplemented	Q.931 cause 69.
IPEC_Q931Cause70OnlyRestrictedDigital InformationBearerCapabilityIsAvailable	Q.931 cause 70.
IPEC_Q931Cause79ServiceOrOption NotImplementedUnspecified	Q.931 cause 79.
IPEC_Q931Cause81InvalidCallReferenceValue	Q.931 cause 81.
IPEC_Q931Cause82IdentifiedChannel DoesNotExist	Q.931 cause 82.

Table 48. IP-Specific Event Cause Codes (Continued)

Event Cause Code	Description
IPEC_Q931Cause83AsuspendedCallExists ButThisCallIdentityDoesNot	Q.931 cause 83.
IPEC_Q931Cause84CallIdentityInUse	Q.931 cause 84.
IPEC_Q931Cause85NoCallSuspended	Q.931 cause 85.
IPEC_Q931Cause86CallHavingTheRequested CallIdentityHasBeenCleared	Q.931 cause 86.
IPEC_Q931Cause88IncompatibleDestination	Q.931 cause 88.
IPEC_Q931Cause91InvalidTransitNetwork Selection	Q.931 cause 91.
IPEC_Q931Cause95InvalidMessageUnspecified	Q.931 cause 95.
IPEC_Q931Cause96MandatoryInformation ElementMissing	Q.931 cause 96.
IPEC_Q931Cause97MessageTypeNonExistent OrNotImplemented	Q.931 cause 97.
IPEC_Q931Cause100InvalidInformationElement Contents	Q.931 cause 100.
IPEC_Q931Cause101MessageNotCompatible WithCallState	Q.931 cause 101.
IPEC_Q931Cause102RecoveryOnTimeExpiry	Q.931 cause 102.
IPEC_Q931Cause111ProtocolErrorUnspecified	Q.931 cause 111.
IPEC_Q931Cause127InterworkingUnspecified	Q.931 cause 127.
IPEC_RASReasonResourceUnavailable	RAS failure reason
IPEC_RASReasonInsufficientResources	RAS failure reason
IPEC_RASReasonInvalidRevision	RAS failure reason
IPEC_RASReasonInvalidCallSignalAddress	RAS failure reason
IPEC_RASReasonInvalidIPEC_RASAddress	RAS failure reason
IPEC_RASReasonInvalidTerminalType	RAS failure reason
IPEC_RASReasonInvalidPermission	RAS failure reason
IPEC_RASReasonInvalidConferenceID	RAS failure reason

Table 48. IP-Specific Event Cause Codes (Continued)

Event Cause Code	Description
IPEC_RASReasonInvalidEndpointID	RAS failure reason
IPEC_RASReasonCallerNotRegistered	RAS failure reason
IPEC_RASReasonCalledPartyNotRegistered	RAS failure reason
IPEC_RASReasonDiscoveryRequired	RAS failure reason
IPEC_RASReasonDuplicateAlias	RAS failure reason
IPEC_RASReasonTransportNotSupported	RAS failure reason
IPEC_RASReasonCallInProgress	RAS failure reason
IPEC_RASReasonRouteCallToGatekeeper	RAS failure reason
IPEC_RASReasonRequestToDropOther	RAS failure reason
IPEC_RASReasonNotRegistered	RAS failure reason
IPEC_RASReasonUndefined	RAS failure reason
IPEC_RASReasonTerminalExcluded	RAS failure reason
IPEC_RASReasonNotBound	RAS failure reason
IPEC_RASReasonNotCurrentlyRegistered	RAS failure reason
IPEC_RASReasonRequestDenied	RAS failure reason
IPEC_RASReasonLocationNotFound	RAS failure reason
IPEC_RASReasonSecurityDenial	RAS failure reason
IPEC_RASTransportQOSNotSupported	RAS failure reasona
IPEC_RASResourceUnavailable	RAS failure reason
IPEC_RASInvalidAlias	RAS failure reason
IPEC_RASPermissionDenied	RAS failure reason
IPEC_RASQOSControlNotSupported	RAS failure reason
IPEC_RASIncompleteAddress	RAS failure reason
IPEC_RASFullRegistrationRequired	RAS failure reason
IPEC_RASRouteCallToSCN	RAS failure reason
IPEC_RASAliasesInconsistent	RAS failure reason
IPEC_RASReasonMax	RAS failure reason

Table 48. IP-Specific Event Cause Codes (Continued)

Event Cause Code	Description
IPEC_SIPReasonStatus400BadRequest = 5400	SIP Request Failure Response 400
IPEC_SIPReasonStatus401Unauthorized = 5401	SIP Request Failure Response 401
IPEC_SIPReasonStatus402PaymentRequired = 5402	SIP Request Failure Response 402
IPEC_SIPReasonStatus403Forbidden = 5403	SIP Request Failure Response 403
IPEC_SIPReasonStatus404NotFound = 5404	SIP Request Failure Response 404
IPEC_SIPReasonStatus405MethodNotAllowed =5405	SIP Request Failure Response 405
IPEC_SIPReasonStatus406NotAcceptable = 5406	SIP Request Failure Response 406
IPEC_SIPReasonStatus407ProxyAuthenticationRequired = 5407	SIP Request Failure Response 407
IPEC_SIPReasonStatus408RequestTimeout = 5408	SIP Request Failure Response 408
IPEC_SIPReasonStatus410Gone = 5410	SIP Request Failure Response 410
IPEC_SIPReasonStatus413RequestEntityTooLarge = 5413	SIP Request Failure Response 413
IPEC_SIPReasonStatus414RequestUriTooLong = 5414	SIP Request Failure Response 414
IPEC_SIPReasonStatus415UnsupportedMediaType = 5415	SIP Request Failure Response 415
IPEC_SIPReasonStatus416UnsupportedURIScheme = 5416	SIP Request Failure Response 416
IPEC_SIPReasonStatus420BadExtension = 5420	SIP Request Failure Response 420
IPEC_SIPReasonStatus421ExtensionRequired = 5421	SIP Request Failure Response 421
IPEC_SIPReasonStatus423IntervalTooBrief = 5423	SIP Request Failure Response 423
IPEC_SIPReasonStatus480TemporarilyUnavailable = 5480	SIP Request Failure Response 480
IPEC_SIPReasonStatus481CallTransactionDoesNotExist = 5481	SIP Request Failure Response 481

Table 48. IP-Specific Event Cause Codes (Continued)

Event Cause Code	Description
IPEC_SIPReasonStatus482LoopDetected = 5482	SIP Request Failure Response 482
IPEC_SIPReasonStatus483TooManyHops = 5483	SIP Request Failure Response 483
IPEC_SIPReasonStatus484AddressIncomplete =5484	SIP Request Failure Response 484
IPEC_SIPReasonStatus485Ambiguous = 5485	SIP Request Failure Response 485
IPEC_SIPReasonStatus486BusyHere = 5486	SIP Request Failure Response 486
IPEC_SIPReasonStatus487RequestTerminated = 5487	SIP Request Failure Response 487
IPEC_SIPReasonStatus488NotAcceptableHere = 5488	SIP Request Failure Response 488
IPEC_SIPReasonStatus491RequestPending = 5491	SIP Request Failure Response 491
IPEC_SIPReasonStatus493Undecipherable = 5493	SIP Request Failure Response 493
IPEC_SIPReasonStatus500ServerInternalError = 5500	SIP Server Failure Response 500
IPEC_SIPReasonStatus501NotImplemented = 5501	SIP Server Failure Response 501
IPEC_SIPReasonStatus502BadGateway = 5502	SIP Server Failure Response 502
IPEC_SIPReasonStatus503ServiceUnavailable = 5503	SIP Server Failure Response 503
IPEC_SIPReasonStatus504ServerTimeout = 5504	SIP Server Failure Response 504
IPEC_SIPReasonStatus505VersionNotSupported = 5505	SIP Server Failure Response 505
IPEC_SIPReasonStatus513MessageTooLarge = 5513	SIP Server Failure Response 513
IPEC_SIPReasonStatus600BusyEverywhere = 5600	SIP Global Failure Response 600
IPEC_SIPReasonStatus603Decline = 5603	SIP Global Failure Response 603
IPEC_SIPReasonStatus604DoesNotExistAnywhere = 5604	SIP Global Failure Response 604

Table 48. IP-Specific Event Cause Codes (Continued)

Event Cause Code	Description
IPEC_SIPReasonStatus606NotAcceptable = 5606	SIP Global Failure Response 606
IPEC_SIPReasonStatusBYE = 5800	SIP Failure Response 800
IPEC_SIPReasonStatusCANCEL = 5801	SIP Failure Response 801
IPEC_SIPReasonStatusMax	

9. Debugging Applications

9.1. Overview

The Global Call software can be configured to write underlying call control library and stack information to log files while an application is running. This information can help trace the sequence of events and identify the source of a problem. These log files are also useful when reporting problems to technical support personnel.

Table 49 shows the log files that can be generated, the directory into which each log file is written, the purpose of each log file, and the configuration files that can be used to customize the output each log file.

Table 49. Summary of Log File Options

Log File Name	Purpose	Where Generated	Configuration File	Placement of Configuration File for Log Generation
gc_h3r.log	Call control library and SIP stack debugging in both Linux and Windows operating systems.	Application directory	*gc_h3r.cfg*	Application directory
logfile.log	H.323 stack debugging on Linux operating systems.	Application directory	*msg.conf*	Application directory
rvtsp1.log	H.323 stack debugging in a Windows environment.	Application directory	*rvtele.ini*	WINNT directory, for example, *C:\Winnt\rvtele.ini*

NOTE: Log file generation is enabled by placing the configuration file in the respective directory as indicated in Table 49 above.

9.2. Log Files

The following topics provide more information about the use of each log file:

- Call Control Library and SIP Stack Debugging
- H.323 Stack Debugging on Linux Operating Systems
- H.323 Stack Debugging in a Windows Environment

9.2.1. Call Control Library and SIP Stack Debugging

The *gc_h3r.cfg* file can be used to customize the information written to the *gc_h3r.log* file by one or both of the following:

- Call Control Library
- SIP Stack

Customizing Call Control Library Logging to the gc_h3r.log File

The call control library comprises three library files; *libgch3r.so*, *libsigal.so*, and *libsipsigal.so* for Linux operating systems, and *libgch3r.dll*, *libsigal.dll*, and *libsipsigal.dll* for Windows operating systems. Table 50 shows the modules in *libgch3r.dll* or *libgch3r.so* and identifies the type of log information generated by each module.

Table 50. Modules in libgch3r.dll or libgch3r.so

Module	Type of Information
M_CRN	Call Reference Number information and states.
M_SHM	Interface to User information (for example, when calling gc_ReleaseCallEx( ), ShmReleaseCallEx is output to the log.
M_LD	Line device operation and states.
M_MEDIA	Media channel related information.
M_PDL	Predefined Library information.
M_PACKER	Packed event information to the application.
M_SH_DB	Preconfiguration information saved in the control library.

Table 50. Modules in libgch3r.dll or libgch3r.so

Module	Type of Information
M_SH_DEC	Internal process communication information.
M_SH_ENC	Internal process communication information.
M_SH_IPC	Internal process communication information.
M_SH_UNPACK	Information unpacked from the application.
M_BOARD	Board-related information.

Table 51 shows the modules in *libsigal.dll* or *libsigal.so* and identifies the type of log information generated by each module.

Table 51. Modules in libsigal.dll or libsigal.so

Module	Type of Information
M_SIG_MAN	DLL manager information.
M_CALL_MAN	Call manager information.
M_SIGNAL	Q.931 manager information.
M_CONTROL	H.245 manager information.
M_CHAN_MAN	Logical channel manager information.
M_CHAN	Logical channel information.
M_IE	Information element information.
M_SIG_DEC	Internal process communication information.
M_SIG_ENC	Internal process communication information.
M_SIG_IPC	Internal process communication information.
M_RAS	Registration, Admission and Status information.
M_CAPS	Capability matching algorithm information.

Table 52 shows the modules in *libsipsigal.dll* or *libsipsigal.so* and identifies the type of log information generated by each module.

Table 52. Modules in libsipsigal.dll or libsipsigal.so

Module	Type of Information
M_S_SIGAL	DLL manager information.
M_S_CALLM	Call manager information.
M_S_SIGNL	SIP manager information.
M_S_CHMGR	Logical channel manager information.
M_SIP_IE	Information element information.
M_SIP_CAP	Capability matching algorithm information.
M_SIP_DEC	Internal process communication information.
M_SIP_ENC	Internal process communication information.
M_SIP_IPC	Internal process communication information.
M_INFO	Message send information (Not yet implemented).
M_REFER	Call transfer information (Not yet implemented).
M_PRACK	Provisional response information (Not yet implemented).
M_AUTHENT	Authentication information (Not yet implemented).

In the *gc_h3r.cfg* file, you can set a different debug level for each module. Table 53 shows the valid bebug levels.

Table 53. Levels of Debug for Call Control Library Logging

Debug Level †	Debug Information	Call Control Library Output to Log File
0	L_NONE	No information
1	L_SPECIAL	Limited information describing call control library configuration
2	L_ERROR	Error information. This is the default level.
3	L_WARNING	Warning information

† Selecting a debug level automatically includes all lower debug levels. For example, selecting level 3 automatically includes levels 0, 1, and 2.

Table 53. Levels of Debug for Call Control Library Logging

Debug Level †	Debug Information	Call Control Library Output to Log File
4	L_INFO	Significant state transition information
5	L_EXTEND	Additional relevant information
6	L_ALL	All information

† Selecting a debug level automatically includes all lower debug levels. For example, selecting level 3 automatically includes levels 0, 1, and 2.

To set a module to the desired debug level for a module, use the following syntax:

```
Module name = Debug Level Number
```

Some examples are:

- m_sip_enc = 2
  sets the m_sip_enc module to the LEVEL_ERROR debug level
- m_call_man = 6
  sets the m_call_man module to the LEVEL_ALL debug level
- m_media = 4
  sets the m_media module to the LEVEL_INFO debug level

Customizing SIP Stack Logging to the gc_h3r.log File

The SIP stack comprises a number of modules as follows:

- RvSipStack_Messge
- RvSipStack_Core
- RvSipStack_Transport
- RvSipStack_Transaction
- RvSipStack_Call
- RvSipStack_Parser
- RvSipStack_Stack
- RvSipStack_Authenticator

- RvSipStack_RegClient
- RvSipStack_MsgBuilder

In the *gc_h3r.cfg* file, you can set different debug levels for each module. Table 54 shows the valid bebug levels that can be set. More than one level of debug can be set for each module. This is achieved by specifying a decimal number that is the binary equivalent of the binary values of each desired level ORed together. For example, a value of 31 decimal (11111 binary) enables all debug levels.

Table 54. Levels of Debug Information for SIP Stack Logging

Debug Level †	Debug Information	SIP Stack Output to Log File
0	None	No information. This is the default level.
1	DEBUG	Detailed information about SIP stack activity
2	INFO	Information about SIP stack activity
4	WARNING	Warnings about possible non-fatal errors
8	ERROR	A non-fatal error occurred
16	EXCEP	A fatal error occurred that blocks stack operation

† Uses a decimal representation of a bit mask, that is level 2 is 010, level 8 is 01000, level 16 is 010000 etc.

Special gc_h3r.cfg Configuration Parameters

In the *gc_h3r.cfg* file, two parameters that have special significance are:

- **outputdest** - This parameter should be always be set to 0.
- **print_file_n_line** - Controls whether filename and line numbers are written to the log. Possible values are:
  - 1 - Filenames and line numbers are written to the log file.
  - 0 - Filenames and line numbers are not written to the log file.

Sample Extract from gc_h3r.log File

The following is an extract from a *gc_h3r.log* file:

```
4 ! 09:35:54.558 ! M_MEDIA   ! L_INFO    !    2 ! >> eventHandler:
ev CNTRL_EV_RX_CONNECT st TRL_ST_TXCONNECT
4 ! 09:35:54.558 ! M_MEDIA   ! L_INFO    !    2 ! >> Media::connectALL
4 ! 09:35:54.558 ! M_MEDIA   ! L_INFO    !    2 ! >> mediaEventHandler:
ev EV_CONNECTALL st ST_WAIT_FOR_CALL
6 ! 09:35:54.558 ! M_MEDIA   ! L_ALL     !    2 ! >> MediaState::startTransaction
6 ! 09:35:54.558 ! M_MEDIA   ! L_ALL     !    2 ! >> Mediaipml::setEventMaskCALL
: mode 32768,maskEvt 0x7f
6 ! 09:35:54.558 ! M_MEDIA   ! L_ALL     !    2 ! << Mediaipml::setEventMaskCALL
:m_EventMaskState 0x22 : [0]
6 ! 09:35:54.558 ! M_MEDIA   ! L_ALL     !    2 ! >>
Mediaipml::setRemoteMediaInfoCALL
5 ! 09:35:54.558 ! M_MEDIA   ! L_EXTEND  !    2 ! LOCAL_RTP Port = 2720 Address :
10.242.212.44
5 ! 09:35:54.568 ! M_MEDIA   ! L_EXTEND  !    2 ! LOCAL_RTCP Port = 2721 Address
: 10.242.212.44
5 ! 09:35:54.568 ! M_MEDIA   ! L_EXTEND  !    2 ! REMOTE_RTP  Port = 2710 Address
: 10.242.212.44
5 ! 09:35:54.568 ! M_MEDIA   ! L_EXTEND  !    2 ! LOCAL_CODER, G711ULAW64K,FSize
20,FPP 1,V 0,PT 0,RedPT 0
5 ! 09:35:54.568 ! M_MEDIA   ! L_EXTEND  !    2 ! REMOTE_RTCP Port = 2711 Address
: 10.242.212.44
5 ! 09:35:54.568 ! M_MEDIA   ! L_EXTEND  !    2 ! REMOTE_CODER, G711ULAW64K,FSize
20,FPP 1,V 0,PT 0,RedPT 0
6 ! 09:35:54.568 ! M_MEDIA   ! L_ALL     !    2 ! <<
Mediaipml::setRemoteMediaInfoCALL: [0]
6 ! 09:35:54.568 ! M_MEDIA   ! L_ALL     !    2 ! <<
MediaState::startTransaction: [0]
4 ! 09:35:54.568 ! M_MEDIA   ! L_INFO    !    2 ! << mediaEventHandler: ev
EV_CONNECTALL st ST_STARTING: [0
```

NOTES: **1.** Lines that begin with 4 (level 4) indicate state machine transitions.

2. Lines that begin with 5 (level 5) provide extended information (in this case the remote coder and the local coder and RTP/RTCP information starting the media channel).

3. Lines that begin with 6 (level 6) provide additional information (in this case the entry to and exit from functions).

9.2.2. H.323 Stack Debugging on Linux Operating Systems

The *msg.conf* file can be used to customize the logging of H.323 stack information to the *logfile.log* file. You can use the *msg.conf* file for the following:

- Selecting Modules that Write to logfile.log
- Selecting the Debug Level

- Selecting the Debug Output Type

Selecting Modules that Write to logfile.log

The H.323 stack comprises a number of modules as follows:

- EMA
- MEMORY
- RA
- CAT
- CM†
- CMAPI†
- CMAPICB†
- CMERR†
- TPKTCHAN†
- CONFIG†
- APPL
- FASTSTART†
- VT
- UNREG
- RAS†
- UDPCHAN
- TCP
- TRANSPORT
- ETIMER
- PER†
- PERERR†
- TUNNCTRL†
- Q931†
- Q931ERR
- L1
- TIMER
- AnnexE

- SSEERR
- SSEAPI
- SSEAPICS
- SSCHAN
- SUPS

NOTE: † indicates the most commonly used modules.

In the *msg.conf* file, you can enable or disable the modules that write information to *logfile.log*. A module is disabled by including a pound symbol (#) in front of the module name. For example, in the following segment of the *msg.conf* file, the TPKTCHAN and UDPCHAN modules write to the log file, but the CMAPICS and CMAPI modules do not.

```
TPKTCHAN
UDPCHAN
#CMAPICB
#CMAPI
```

Selecting the Debug Level

In the *msg.conf* file, you can set the debug level by including lines similar to the following:

```
#set up debug level
%2
```

In this example, the debug level is set to 2. Table 55 shows the valid bebug levels and their meaning.

Table 55. Debug Levels for H.323 Stack Logging to logfile.log

Debug Level	H.323 Stack Output
0	Do not display or print debug information. This is the default level.
1	Do not display trees or do not check trees for ASN.1 consistency.
2	Display all information including trees, but do not check trees for ASN.1 consistency.

Table 55. Debug Levels for H.323 Stack Logging to logfile.log

Debug Level	H.323 Stack Output
3	Display all information and check trees for ASN.1 consistency. Display any inconsistencies found.
4	Display all messages.

Selecting the Debug Output Type

In the *msg.conf* file, you can direct where the debug output will be written by including lines similar to the following:

```
#debug output definition
>file
```

In this example, the debug output is directed to a file. Table 56 shows the valid output options.

Table 56. Debug Output Options for H.323 Stack Logging to logfile.log

Output Option	H.323 Stack Output
file	Write the debug output to a file. The name of the file is *<current directory>logfile.log*.
logger	Write the debug output to a logger, such as the debug logger, or to any other printing tool.
terminal	Display the debug output on a terminal.

9.2.3. H.323 Stack Debugging in a Windows Environment

The *rvtele.ini* file can be used to customize the logging of H.323 stack information to the *rvtsp1.log* file. You can use the *rvtele.ini* file for the following:

- Selecting Modules that Write to the rvtsp1.log
- Selecting the Debug Level
- Selecting the Debug Output Type
- Configuring Cyclic Mode Parameters

Selecting Modules that Write to the rvtsp1.log

The H.323 stack comprises a number of modules as follows:

- EMA
- MEMORY
- RA
- CAT
- CM†
- CMAPI†
- CMAPICB†
- CMERR†
- TPKTCHAN†
- CONFIG†
- APPL
- FASTSTART†
- VT
- UNREG
- RAS†
- UDPCHAN
- TCP
- TRANSPORT
- ETIMER
- PER†
- PERERR†
- TUNNCTRL†
- Q931†
- Q931ERR
- L1
- TIMER
- AnnexE
- SSEERR

- SSEAPI
- SSEAPICS
- SSCHAN
- SUPS

NOTE: † indicates the most commonly used modules.

In the *rvtele.ini* file, you can enable or disable modules from writing information to *rvtsp1.log*. A module is enabled by including a line with the <module name>=1 under a section labelled [insertIntoFile]. For example:

```
[insertIntoFile]
TPKTCHAN=1
UDPCHAN=1
CMPAPICS=0
CMPAIP=0
```

In this example, the TPKTCHAN and UDPCHAN modules will writer to the log file, but the CMAPICS and CMAPI modules will not.

NOTE: Only one section labelled [insertIntoFile] is allowed in the *rvtele.ini* file.

Selecting the Debug Level

In the *rvtele.ini* file, you can set the debug level by including lines similar to the following:

```
#set up debug level
deblevel=2
```

In this example, the debug level is set to 2. Table 57 shows the valid bebug levels and their meaning.

Table 57. Debug Levels for H.323 Stack Logging to rvtsp1.log

Debug Level	H.323 Stack Output
0	Do not display or print debug information. This is the default level.
1	Display messages from all source modules, except those source modules in the list given with the filtering level instructions.

Table 57. Debug Levels for H.323 Stack Logging to rvtsp1.log

Debug Level	H.323 Stack Output
2	Display messages from all source modules according to the list given with the filtering level instructions.
3	Display messages from all source modules.

Selecting the Debug Output Type

In the *rvtele.ini* file, the following section must exist to direct the debug output:

```
[supserve]
...
msgfile=1
msgdeb=1
msgwin=0
```

In this example, the debug output is directed to the *rvtsp1.log* file and writes all debug output to the debugger window, such as the Windows debugger when running the application in a Windows environment. Table 58 shows the valid output options.

Table 58. Debug Output Options for H.323 Stack Logging to rvtsp1.log

Output Option	H.323 Stack Output
msgfile	The stack writes all debug messages to the *rvtsp1.log* file.
msgdeb	The stack writes all debug messages to a debugger window.
msgwin	The stack writes all debug messages to a special window that it creates.

Configuring Cyclic Mode Parameters

When using a debug output of msgfile=1, it is possible to work in cyclic mode and set a limit to the physical size of the *rvtsp1.log* file. When the log file expands to this size, information will be logged to the beginning of the file overwriting older logging information. The lines in the *rvtele.ini* file that control these parameters are as follows:

```
[fileParams]
fileSize=20000000
fileCyclic=1
```

To disable this option, set the parameter values as follows:

```
[fileParams]
fileSize=-1
fileCyclic=0
```

10. Called and Calling Party Address List Format When Using H.323

10.1. Called Party Address List

Called party address lists are formatted as shown 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 3.4.19, "gc_Start( )", on page 72 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.

10.2. 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 Section 10.1, “Called Party Address List”, on page 211.

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.

10.3. 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.

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: 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).

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.

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

C

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V